THE UNIVERSITY *of* LIVERPOOL

Fortran 90 Course Notes

— — —

AC Marshall with help from JS Morgan and JL Schonfelder.
Thanks to Paddy O'Brien.

© University of Liverpool, 1997

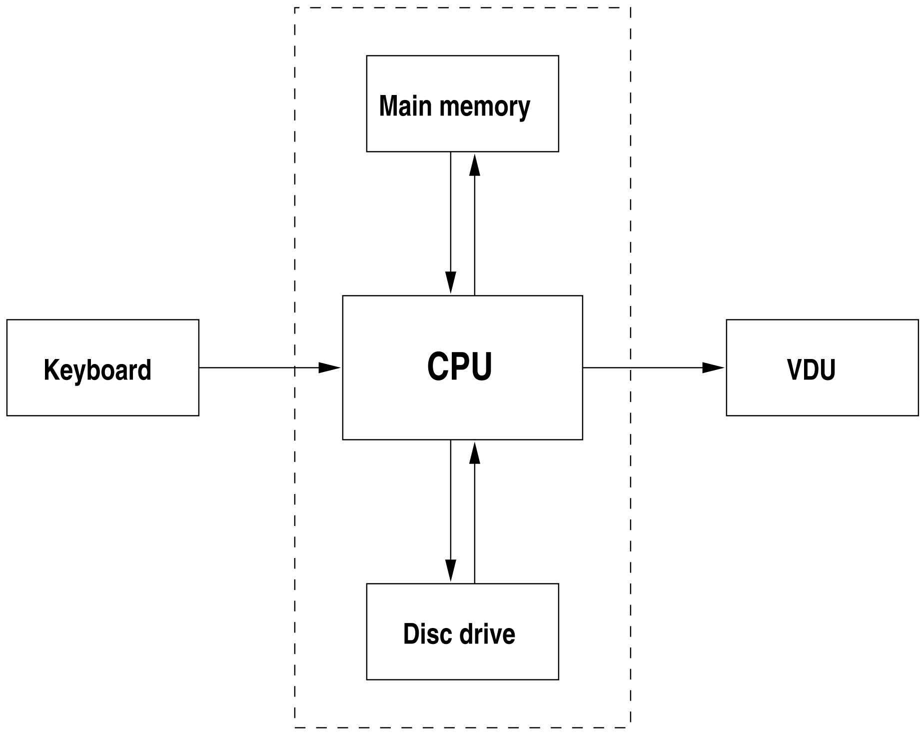
Contents

Module 1: Fundamentals Of Computer Programming

1 Introduction to Computer Systems

1.1 What is a Computer?

The following schematic diagram gives the layout of a Personal Computer (PC), most single user systems follow this general design:



The components perform the following tasks:

- **CPU (Central Processor Unit)** — does the 'work', fetches, stores and manipulates values that are stored in the computers memory. Processors come in all different 'shapes and sizes' — there are many different types of architectures which are suited to a variety of different tasks. We do not consider any particular type of CPU is this course.
- **Main memory (RAM — Random Access Memory)** — used to store values during execution of a program. It can be written to and read from at any time.

- **Disc drive (hard or floppy)** — 'permanently' stores files (programs and data). Hard discs are generally located inside the machine and come in a variety of different sizes and speeds. They do not, in fact, store files permanently — they often go wrong and so must undergo a **back-up** at regular intervals. The floppy disc drive allows a user to make his or her own back up of important files and data. It is *very important* to keep back-ups. Do not be caught out — you may well lose all your work!
- **Keyboard** — allows user to **input** information. Nowadays, most keyboards have more or less the same functionality.
- **VDU (Visual Display Unit)** — visually **outputs** data. There are numerous types of VDU differing in the resolution (dots per inch) and the number of colours that can be represented.
- **Printer** — allows a hard copy to be made. Again, there are many different types of printers available, for example, line printers, dot-matrix printers, bubble jet printers and laser printers. These also differ in their resolution and colour palette.

The last four are known as *peripheral devices*.

A good PC could contain:

- Intel Pentium P166 CPU
- 32MBytes RAM (main memory)
- 2.1GByte hard disc
- SVGA monitor
- IBM PC keyboard

In addition a system may include,

- printer, for example, an HP LaserJet
- soundcard and speakers
- CD ROM drive (Read Only Memory)
- SCSI (*'scuzzy'*) disc (fast),
- floppy disc drive (for backing up data)
- network card

1.2 What is Hardware and Software?

A computer system is made up from **hardware** and **software**.

Hardware is the physical medium, for example:

- circuit boards
- processors

- keyboard

A piece of software is a computer program, for example:

- an operating system
- an editor
- a compiler
- a Fortran 90 program

The software allows the hardware to be used. Programs vary enormously in size and complexity.

1.3 Telling a Computer What To Do

To get a computer to perform a specific task it must be given a sequence of unambiguous instructions or a **program**.

We meet many examples of programs in everyday life, for example, instructions on how to assemble a bedside cabinet. These instructions are generally numbered, meaning that there is a specific order to be followed, they are also (supposed to be) precise so that there is no confusion about what is intended:

1. insert the spigot into hole 'A',
2. apply glue along the edge of side panel,
3. press together side and top panels
4. attach toggle pin 'B' to gromit 'C'
5. ... and so on

If these instructions are not followed 'to the letter', then the cabinet would turn out wonky.

1.4 Some Basic Terminology

It is necessary to cover some terminology. Hopefully, much of it will be familiar — you will hear many of the terms used throughout the course.

- **Bit** is short for Binary Digit. Bits have value of `1` or `0`, (or *on* or *off*, or, *true* or *false*),
- 8 Bits make up 1 **Byte**,

  1024 Bytes make up 1 KByte (1 KiloByte or 1K), ("Why 1024?" I hear you ask. Because $2^{10} = 1024$.

  1024 KBytes make up 1 MByte (1 MagaByte or 1M),

  1024 MBytes make up 1 GByte (1 GigaByte or 1G),
- all machines have a **wordsize** — a fundamental unit of storage, for example, 8-bits, 16-bits, etc. The size of a **word** (in Bytes) differs between machines. A Pentium based machine is 32-bit.

- a **flop** is a floating point operation per second. A floating point operation occurs when two real numbers are added. Today, we talk of megaflops or even gigaflops.

- **parallel processing** occurs when two or more CPUs work on solution of the same problem at the same time.

1.5 How Does Computer Memory Work?

In this hypothetical example, wordsize is taken to be 8-bits:

3F29	1	0	1	0	1	1	1	0	3F2A								
3F2B									3F2C								

A computers memory is addressable. Each location is given a specific 'number' which is often represented in hexadecimal [base-16], for example, `3F2C`. (Hexadecimal digits are as follows: 0, 1, 2, 3, ..., 9, A, B, C, D, E, F, 10, 11, ..., 19, 1A, 1B, ..., 1F, 20, ...). The CPU is able to read to and write from a specified memory location at will. Groups of memory locations can be treated as a whole to allow more information to be stored. Using the cryptic hexadecimal identifiers for memory locations is very awkward so Fortran 90 allows (English) names to be used instead, this allows programs to make sense to a casual reader.

Even when the computer has just been turned on, each memory location will contain some sort of 'value'. In this case the values will be random. In the general case the values will be those that remain from the previous program that used the memory. For this reason it is *very important* to initialise memory locations before trying to use the values that they are storing.

All CPUs have an **instruction set** (or language) that they understand. Eventually *all* Fortran 90 programs must be translated (or compiled) into instructions from this set. Roughly speaking, all processors have the same sort of instructions available to them. The CPU can say things like, 'fetch me the contents of memory location `3F2C`' or 'write this value to location `3AF7`'. This is the basis of how all programs work.

Consider the following sequence of **assembler code** instructions:

```
LDA '3F2C'   load (fetch) the contents of 3F2C
ADD '3F29'   add to this the contents of 3F29
STO '3F2A'   store the value in location 3F2A
```

The sequence of instructions, which is meant only for illustrative purposes, effectively adds two numbers together and stores the result in a separate memory location. Until 1954 when the first dialect of Fortran was developed, all computer programs were written using assembler code. It was John Backus, then working for IBM, who proposed that a more economical and efficient method of programming their computer should be developed. The idea was to design a language that made it possible to express mathematical formulae in a more natural way than that currently supported by assembler languages. The result of their experiment was Fortran (short for IBM Mathematical *For*mula *Tran*slation System).

This new language allowed the above assembler instructions to be written less cryptically as, for example:

```
K = I + J
```

A compiler would then translate the above assignment statement into something that looks like the assembler code given above.

1.6 Numeric Storage

In general, there are two types of numbers used in Fortran 90 programs `INTEGER`s (whole numbers) and `REAL`s (floating point numbers).

`INTEGER`s are stored *exactly*, usually in range (-32767,32767), however, they are not stored in the format hinted at in the previous section but in a special way that allows addition and subtraction to be performed in a straightforward way.

`REAL`s are stored *approximately* and the space they occupy is partitioned into a **mantissa** and an **exponent**. (In the following number: 0.31459×10^1, the mantissa is 0.31459 and the exponent is 1.) In a `REAL` number, the exponent can only take a small range of values — if the exponent is expected to be large then it may be necessary to use a numeric storage unit that is capable of representing larger values, for example `DOUBLE PRECISION` values.

It is possible for a program to throw a **numeric exception** such as **overflow** which occurs when number is outside of supported range (for example, a real number where exponent is too big) or **underflow** which is the opposite, the number is too close to zero for representation (for example, a real number where the exponent is too small).

In Fortran 90, the `KIND` mechanism allows the user to specify what numeric range is to be supported. This allows programs to be numerically portable. Different computer systems will allow numbers of differing sizes to be represented by one word. It is possible that when a program has been moved from one computer system to another, it may fail to execute correctly on the new system because underflow or overflow exceptions are generated. The `KIND` mechanism combats this serious problem.

`CHARACTER`s variables are stored differently.

2 What Are Computer Programs

2.1 Programming Languages

Computers only understand binary codes so the first programs were written using this notation. This form of programming was soon seen to be extremely complex and error prone so assembler languages were developed. Soon it was realised that even assembler languages could be improved on. Today, a good programming language must be:

- totally unambiguous (unlike natural languages, for example, English — 'old women and men suck eggs', does this mean that *men* or *old* men suck eggs?).
- expressive — it must be fairly easy to program common tasks,
- practical — it must be an easy language for the compiler to translate,
- simple to use.

All programming languages have a very precise **syntax** (or grammar). This will ensure that a syntactically-correct program only has a single meaning.

2.2 High-level Programming Languages

Assembler code is a **Low-Level Language**. It is so-called because the structure of the language reflects the instruction set (and architecture) of the CPU. A programmer can get very close to the physical hardware. Low-level languages allow very efficient use of the machine but are difficult to use.

Fortran 90, FORTRAN 77, ADA, C and Java are examples of **High-Level Languages**. They provide a much higher degree of abstraction from the physical hardware. They also allow for **portable** programs to be written, i.e., programs that will run on a variety of different systems and which will produce the same results regardless of the platform. As well as having the above benefits, high-level languages are more expressive and secure and are much quicker to use than low-level languages. In general, however, a well written assembler program will run faster than a high-level program that performs the same task.

At the end of the day, an executable program that runs on a CPU must still be represented as a series of binary digits. This is achieved by **compiling** (translating) a high-level program with a special piece of software called a **compiler**. Compilers are incredibly complicated programs that accept other programs as input and generate a binary executable object file as output.

2.3 An Example Problem

Consider the following problem which is suitable for solution by computer.

To convert from ^oF (Fahrenheit) to ^oC (Centigrade) we can use the following formula:

$$c = 5 \times (f - 32)/9$$

To convert from ^oC to K (Kelvin) we add 273.

A specification of the program could be that it would prompt the user for a temperature expressed in degrees Fahrenheit, perform the necessary calculations and print out the equivalent temperatures in Centigrade and Kelvin.

2.4 An Example Program

A program which follows the above specification is given below.

```
PROGRAM Temp_Conversion
IMPLICIT NONE
 INTEGER :: Deg_F, Deg_C, K
 PRINT*, "Please type in the temp in F"
 READ*, Deg_F
 Deg_C = 5*(Deg_F-32)/9
 PRINT*, "This is equal to", Deg_C, "C"
 K = Deg_C + 273
 PRINT*, "and", K, "K"
END PROGRAM Temp_Conversion
```

This program, in this case, called `Temp.f90`, can be compiled using the NAg v2.2 Fortran 90 compiler. The compiler is invoked on the Unix command line as follows:

```
chad2-13{adamm} 26> f90 Temp.f90
NAg Fortran 90 compiler v2.2. New Debugger: 'dbx90'
```

If the program has been written exactly as above then there will not be any error messages (or **diagnostics**). (The case where a program contains mistakes will be covered later.) The compiler produces executable code which is stored (by default) in a file called `a.out`. It is possible to give this file a different name, this is discussed later. The executable file can be run by typing its name at the Unix prompt:

```
chad2-13{adamm} 27> a.out
 Please type in the temp in F
45
 This is equal to 7 C
 and 280 K
```

The program will start executing and will print out `Please type in the temp in F` on the screen, the computer will then wait for a value to be entered via the keyboard. As soon as the required input has been supplied and the `Return` key pressed, the program will continue. Almost instantaneously, the equivalent values in Centigrade and Kelvin will appear on the screen.

2.5 Analysis of Temperature Program

The code is delimited by `PROGRAM` ... `END PROGRAM` statements, in other words these two lines mark the beginning and end of the code. Between these lines there are *two* distinct areas.

- **Specification Part**

  This is the area of the program that is set aside for the declarations of the named memory locations (or **variables**) that are used in the executable area of the program. As well as supplying variable names, it is also necessary to specify the **type** of data that a particular variable will hold. This is important so that the compiler can work out how store the value held by a variable. It is also possible to assign an initial values to a variable or specify that a particular memory location holds a constant value (a read-only value that will not change whilst the program is executing). These techniques are very useful.

- **Execution Part**

  This is the area of code that performs the useful work. In this case the program does the following:

  1. prompts the user for input,
  2. reads input data (`Deg_F`),
  3. calculates the temperature in ^{o}C,
  4. prints out this value,
  5. calculates the temperature in Kelvin and
  6. prints out this value.

2.5.1 A Closer Look at the Specification Part

There are a couple more points that should be raised. The `IMPLICIT NONE` statement should *always* be present in *every* Fortran 90 program unit. Its presence means that *all* variables mentioned in the program must be declared in the specification part. A common mistake whilst coding is to mistype a variable name, if the `IMPLICIT NONE` statement is not present then the compiler will assume that the mistyped variable name is a new, hitherto unmentioned, variable and will use whatever value happened to be in the memory location that the variable refers to. An error such as this caused the US Space Shuttle to crash.

The third line of the program declares three `INTEGER` (whole number) variables, there are a number of other variable types that can be used in different situations:

- `REAL` — real numbers, i.e., no whole numbers, e.g., 3.1459, 0.31459×10^1. `REAL` variables can hold larger numbers than `INTEGER`s.
- `LOGICAL` — can only take one of two vaules: `.TRUE.` or `.FALSE.`,
- `CHARACTER` — contains single alphanumeric character, e.g., `'a'`,
- `CHARACTER(LEN=12)` — contains 12 alphanumeric characters. A number of consecutive alphanumeric characters are known as a **string**,
- user defined types — combination of the above (see later).

Fortran 90 is *not* case sensitive. `K` is the same as `k`.

2.5.2 A Closer Look at the Execution Part

Let us now look at the part of the program that does the actual 'work'.

- `PRINT*, "Please type in the temp in F"` — writes the string (between the quotes) to the VDU screen,
- `READ*, Deg_F` — reads a value from the keyboard and assigns it to the `INTEGER` variable `Deg_F`,
- `Deg_C = 5*(Deg_F-32)/9` — this is an **assignment statement**. The **expression** on the RHS is evaluated and then assigned to the `INTEGER` variable `Deg_C`. This statement contains a number of operators:
  - `*` is the multiplication operator,
  - `-` is the subtraction operator,
  - `=` is the assignment operator.

  The Fortran 90 language also contains `+` and `/` operators for addition and division.

  Division is much harder than any other operation for a CPU to perform - it takes longer so uses more resources, for example, if there is a choice between dividing by `2.0` or multiplying by `0.5` the latter should be selected.
- `PRINT*, "This is equal to", Deg_C, "C"` — displays a string on the screen followed by the value of a variable (`Deg_C`) followed by a second string (`"C"`).

By default, input is from the keyboard and output to the screen.

2.6 How to Write a Computer Program

There are four general phases during the development of any computer program:

1. specify the problem,
2. analyse and break down into a series of steps towards solution, (i.e., design an **algorithm**),
3. write the Fortran 90 code,
4. compile and run (i.e., test the program).

When writing a computer program it is absolutely vital that the problem to be solved is fully understood. A good percentage of large programming projects run into trouble owing to a misunderstanding of what is actually required. The specification of the problem is crucial. It is usual to write a detailed 'spec' of the problem using a 'natural language' (e.g., English). Unfortunately, it is very easy to write an ambiguous specification which can be interpreted in a number of ways by different people. In order to combat this, a number of Formal Methods of specification have been developed. These methods are often high-level abstract mathematical languages which are relatively simple to convert to high-level programs; in some cases this can be done automatically. Examples of such languages are Z and VDM. We do not propose to cover such specification languages here. For the purposes of learning Fortran 90, a simple natural language specification is adequate.

Once the problem has been specified, it needs to be broken down into small steps towards the solution, in other words an **algorithm** should be designed. It is perfectly reasonable to use English to specify each step. This is known as *pseudo-code*.

The next two phases go hand-in-hand, they are often known as the *code-test-debug* cycle and it is often necessary to perform the cycle a number of times. It is very rare that a program is written correctly on the first attempt. It is common to make typographical errors which are usually unearthed by the compiler. Once the typo's have been removed, the program will be able to be compiled and an executable image generated. Again, it is not uncommon for execution to expose more errors or bugs. Execution may either highlight run-time errors which occur when the program tries to perform illegal operations (e.g., divide by zero) or may reveal that the program is generating the wrong answers. The program must be thoroughly tested to demonstrate that it is indeed correct. The most basic goal should be to supply test data that executes *every* line of code. There are many software tools that generate statistical reports of code coverage, such as the Unix `tcov` utility or the more comprehensive LDRA Testbed.

2.7 A Quadratic Equation Solver

This section shows the various steps that are needed when designing and writing a computer program to generate the roots of a quadratic equation.

2.7.1 The Problem Specification

Write a program to calculate the roots of a quadratic equation of the form:

$$ax^2 + bx + c = 0$$

The roots are given by the following formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

2.7.2 The Algorithm

1. READ the values of the coefficients a, b and c from the keyboard.
2. if a is zero then stop as we do not have a quadratic equation,
3. calculate value of discriminant $D = b^2 - 4ac$
4. if D is zero then there is one root: $\frac{-b}{2a}$,
5. if D is $>$ 0 then there are two real roots: $\frac{-b+\sqrt{D}}{2a}$ and $\frac{-b-\sqrt{D}}{2a}$,
6. if D is $<$ 0 there are two complex roots: $\frac{-b+\mathbf{i}\sqrt{-D}}{2a}$ and $\frac{-b-\mathbf{i}\sqrt{-D}}{2a}$,
7. PRINT solution.

2.7.3 The Program

The following program is one solution to the problem:

```
PROGRAM QES
 IMPLICIT NONE
  INTEGER :: a, b, c, D
  REAL    :: Real_Part, Imag_Part
  PRINT*, "Type in values for a, b and c"
  READ*, a, b, c
  IF (a /= 0) THEN
! Calculate discriminant
   D = b*b - 4*a*c
   IF (D == 0) THEN                  ! one root
    PRINT*, "Root is ", -b/(2.0*a)
   ELSE IF (D > 0) THEN              ! real roots
    PRINT*, "Roots are",(-b+SQRT(REAL(D)))/(2.0*a),&
            "and",      (-b-SQRT(REAL(D)))/(2.0*a)
   ELSE                              ! complex roots
    Real_Part = -b/(2.0*a)
! Since D < 0, calculate SRQT of -D which will be +ve
    Imag_Part = (SQRT(REAL(-D))/(2.0*a))
    PRINT*, "1st Root", Real_Part, "+", Imag_Part, "i"
    PRINT*, "2nd Root", Real_Part, "-", Imag_Part, "i"
   END IF
  ELSE                               ! a == 0
! a is equal to 0 so ...
   PRINT*, "Not a quadratic equation"
  END IF
END PROGRAM QES
```

2.8 The Testing Phase

As the basic goal we want to ensure that each and every line of the program has been executed. Looking at the code we need the following test cases:

1. discriminant greater than zero: real valued roots.
2. discriminant equals zero: single real valued root.
3. discriminant less than zero: complex valued roots.
4. coefficient a is zero: not a quadratic equation.

These four situations can be seem to correspond to each of the four `PRINT` statements in the program. The following values of coefficients should exercise each of these lines:

1. $a = 1$, $b = -3$ and $c = 2$,
2. $a = 1$, $b = -2$ and $c = 1$,
3. $a = 1$, $b = 1$ and $c = 1$,
4. $a = 0$, $b = 2$ and $c = 3$.

Below is what happens when the above test data is supplied as input to our program.

```
uxa{adamm} 35> a.out
 Type in values for a, b and c
1 -3 2
 Roots are   2.0000000 and   1.0000000
uxa{adamm} 36> a.out
 Type in values for a, b and c
1 -2 1
 Root is    1.0000000
uxa{adamm} 37> a.out
 Type in values for a, b and c
1 1 1
 1st Root  -0.5000000 +   0.8660254 i
 2nd Root  -0.5000000 -   0.8660254 i
uxa{adamm} 38> a.out
 Type in values for a, b and c
0 2 3
 Not a quadratic equation
```

It can be seen that every line has been executed and in each case the correct results have been produced.

2.8.1 Discussion

The previous program introduces some new ideas,

- **comments** — anything on a line following a `!` is ignored. Comments are added solely for the benefit of the programmer — often a program will be maintained by a number of different people during its lifetime. Adding useful and descriptive comments will be invaluable to anybody who has to modify the code. There is nothing wrong with having more comments in a program than there are executable lines of code!
- `IF` **construct** — different lines are executed depending on the value of the Boolean expression. `IF` statements and constructs are explained more fully in Sections 11.1 and 11.2.
- **relational operators** — == ('is equal to') or > ('is greater than'). Relational operators, (see 10.4,) allow comparisons to be made between variables. There are six such operators for use between numeric variables, the remaining four are
  - >= — 'greater than or equal to'
  - /= — 'not equal to'
  - < — 'less than'
  - <= — 'less than or equal to'

  These operators can also be written in non-symbolic form, these are `.EQ.`, `.GT.`, `.GE.`, `.NE.`, `.LT.` and `.LE.` respectively.
- **nested constructs** — one control construct can be located inside another. This is explained more fully in Section 11.3.
- **procedure call** — `SQRT(X)` returns square root of `X`. A procedure is a section of code that performs a specific and common task. Calculating the square root of a number is a relatively common requirement so the code to do this has been written elsewhere and can be called at any point in the program. This procedure requires an **argument**, `X`, to be supplied — this is the number that is to be square rooted. In this case, the routine is an **intrinsic procedure**, this means that it is part of the Fortran 90 language itself — every compiler is bound to include this procedure. For more information see Section 17.2.
- **type conversion** — in the above call, the argument `X` must be `REAL` (this is specified in the Fortran 90 standard language definition). In the program, `D` is `INTEGER`, `REAL(D)` converts `D` to be real valued so that `SQRT` can be used correctly. For a more detailed explanation see Section 13.1.

Another useful technique demonstrated here is the pre-calculation of common sub-expressions, for example, in order to save CPU time we only calculate the discriminant, `D`, once.

2.9 Software Errors

Errors are inescapable! There are two types of error: **compile-time** or **syntax** errors and **run-time** errors. Errors are often known as 'bugs' because one of the very earliest computers started producing the wrong answers when a moth flew inside it and damaged some of its valves!

2.9.1 Compile-time Errors

In quadratic equation solver program, if we accidentally mistyped a variable name:

```
Rael_Part = -b/(2.0*a)
```

then the compiler would generate a syntax error:

```
uxa{adamm} 40> f90 Quad.f90
NAg Fortran 90 compiler v2.2.
Error: Quad.f90, line 16:
       Implicit type for RAEL_PART
       detected at RAEL_PART@=
Error: Quad.f90, line 24:
       Symbol REAL_PART referenced but never set
       detected at QES@<end-of-statement>
[f90 terminated - errors found by pass 1]
```

The compiler is telling us that RAEL_PART has not been given an explicit type, in other words, it has not been declared in the specification area of the code. It is also telling us that we are trying to print out the value of REAL_PART but that we have not actually assigned anything to it. As you can see from the above compiler output, error messages tend to be obtuse at times. A sign of a good compiler is one that can give specific and useful error messages rather than simply telling you that your program doesn't make sense!

2.9.2 Run-time Errors

If we had made the following typo in our quadratic equation solver,

```
Real_Part = -b/(.0*a)
```

then the program would compile but upon execution we would get a run-time error,

```
uxa{adamm} 43> a.out
 Type in values for a, b and c
1 1 1
*** Arithmetic exception:
Floating divide by zero - aborting
Abort
```

the program is telling the CPU to divide b by zero (.0*a). As this is impossible, our program crashes and an error message is sent to the screen. It is run-time errors like this that mean it is absolutely essential to thoroughly test programs before committing them for serious use.

It is also possible to write a program that gives the wrong results — this is likely to be a bug in the algorithm!

2.10 The Compilation Process

The NAg Fortran 90 compiler is invoked by f90 followed by the filename of the program to be compiled:

```
f90 Quad.f90
```

This command:

1. checks the program syntax
2. generates object code
3. passes object code to linker which attached libraries (system, I/O, etc) and generates executable code in a file called `a.out`.

Step 3 attaches code which deals with, amongst other things, mathematical calculations and input from the keyboard and output to the screen. This code, which is resident in a system library, is used with all available compilers, in other words, the same I/O library will be used with C, ADA or C++ programs. This alleviates the need for each compiler to include a separate instance of these I/O routines. It is possible that this step may give a linker error if procedures used in the code cannot be found in a library.

Executable code is processor specific. Code complied for an Intel Pentium will not run an a Sun SPARC and *vice-versa*. If a program is moved to a different platform then it will need to be recompiled. If the program conforms to the Fortran 90 standard then the source code will not need editing at all. Some compilers accept extensions to the Fortran 90 standard. If any of these extensions are used the program will be non-standard conforming and may need significant rewriting when compiled on a different platform or by a different compiler.

2.11 Compiler Switches

All compilers can be invoked with a number of different options. The effect of these options is explained in the `man` pages or in the compiler manual. The NAg compiler has a number, for example,

- `-o` <*output-filename*>: gives executable a different name, for example, `Quad`

```
f90 Quad.f90 -o Quad
```

- `-dryrun`: show but do not execute commands constructed by the compiler.
- `-hpf`: accept the extensions to Fortran 90 as specified by the High Performance Fortran Forum.
- `-info` <*level*>: set level of information messages generated by the compiler, from 0 to 9.
- `-time`: report execution times

```
f90 Quad.f90 -o Quad -time
```

For more information about the compiler type:

```
man f90
```

Question 1: Compilation and Editing

The following program calculates the roots of a quadratic equation:

```
PROGRAM QES
 IMPLICIT NONE
  INTEGER :: a, b, c, D
  REAL    :: Real_Part, Imag_Part
  PRINT*, "Type in values for a, b and c"
  READ*, a, b, c
  IF (a /= 0) THEN
! Calculate discriminant
   D = b*b - 4*a*c
   IF (D == 0) THEN                  ! one root
    PRINT*, "Root is ", -b/(2.0*a)
   ELSE IF (D > 0) THEN              ! real roots
    PRINT*, "Roots are",(-b+SQRT(REAL(D)))/(2.0*a),&
            "and",      (-b-SQRT(REAL(D)))/(2.0*a)
   ELSE                              ! complex roots
    Real_Part = -b/(2.0*a)
    Imag_Part = (SQRT(REAL(-D))/(2.0*a))
    PRINT*, "1st Root", Real_Part, "+", Imag_Part, "i"
    PRINT*, "2nd Root", Real_Part, "-", Imag_Part, "i"
   END IF
  ELSE                               ! a == 0
   PRINT*, "Not a quadratic equation"
  END IF
END PROGRAM QES
```

1. Using an editor, type the above program into a file called `QuadSolver.f90`
2. Compile and run the program. Verify the correctness of the code by supplying the following test data:
   (a) $a = 1$, $b = -3$ and $c = 2$,
   (b) $a = 1$, $b = -2$ and $c = 1$,
   (c) $a = 1$, $b = 1$ and $c = 1$,
   (d) $a = 0$, $b = 2$ and $c = 3$.
3. Copy `QuadSolver.f90` into a new file called `NewQuadSolver.f90`.
4. Edit this file and declare a new `REAL` variable called `one_over_2a`.
5. In the executable part of the code, set `one_over_2a` to be equal to the value of `1/(2.0*a)`. Where ever this expression occurs replace it with `one_over_a2`. Why is this a good idea?
6. Define another new `REAL` variable called `sqrt_D` and *where appropriate* pre-calculate `SQRT(REAL(D))` and substitute this new variable in place of this expression.
7. Use a different set of test data to that given above to convince yourself that `NewQuadSolver.f90` is a correct program.
8. Change the name of the program to be `FinalQuadSolver.f90` and compile the code to produce an executable file called `FinalQuadSolver`.
9. Delete the original program `QuadSolver.f90` and the executable file `a.out`.

Question 2: The Hello World Program

Write a Fortran 90 program to print out `Hello World` on the screen.

Module 2: Introduction to Fortran 90

3 Introduction

FORTRAN 77 has been widely used by scientists and engineers for a number of years now. It has been a very successful language but is now showing signs of age, in the last few years there has been a tendency for people to drift away from FORTRAN 77 and begin to use C, Ada or C++. The Fortran standard has now been revised to bring it up to date with a new improved language, known informally as Fortran 90, being defined.. Comparisons have been (rightly) drawn between the new Fortran standard and APL, ADA and C++. All three languages contain elements which make them the 'flavour of the month', they are, to some degree, *object oriented*. The criteria for what makes a language actually object oriented is debatable but the new operator overloading, user defined typing and `MODULE` packaging features of Fortran 90 certainly help to forward its case. Along with ADA and APL it uses the concept of array operations and reduction operators; Fortran 90 also supports user defined generic procedures to enhance usability but these are implemented in a simpler and more efficient way than in ADA.

Here we highlight the new Fortran 90 features which make language more robust and usable, for example, of those mentioned above, many are object-based facilities which make it more difficult to make mistakes or do daft things. Fortran 90 is also comparable to ADA in the sense that it too has many restrictions detailed in the standard document meaning that mistakes, which in other languages (such as C, C++ and FORTRAN 77) would be syntactically correct but cause an odd action to be taken, are ruled out at compile time. As an example of this, if explicit interfaces are used then it is not possible to associate dummy and actual arguments of different types — this was possible in FORTRAN 77 and was sometimes used in anger but generally indicated an error.

3.1 The Course

Throughout the course it is assumed that the user has experience of FORTRAN 77 or least one other high level programming language such as Modula, Pascal, APL, Ada, C,or Algol and understands terms like *bits*, *real numbers*, *precision* and *data structures*.

4 Fortran Evolution

4.1 A Brief History of FORTRAN 77

Fortran, which originally stood for *IBM Mathematical FORmula TRANslation System* but has been abbreviated to *FORmula TRANslation*, is the oldest of the established "high-level" languages, having been designed by a group in IBM during the late 1950s. The language became so popular in the early 1960s that other vendors started to produce their own versions and this led to a growing divergence of dialects (by 1963 there were 40 different compilers). It was recognised that such divergence was not in the interests of either the computer users or the computer vendors and so FORTRAN 66 became the first language to be officially standardised in 1972 (it is quite common for the Fortran version number to be out of step with the standardisation year). The publication of the standard meant that Fortran became more widely implemented than any other language. By the mid 1970s virtually every computer, mini or mainframe, was supplied with a standard-conforming FORTRAN 66 language processing system. It was therefore possible to write programs in Fortran on any one system and be reasonably confident that these could be moved fairly easily to work on any other system. This, and the fact that Fortran programs could be processed very efficiently, led to Fortran being the most heavily-used programming language for non-commercial applications.

The standard definition of Fortran was updated in the late 1970's and a new standard, ANSI X3.9-1978, was published by the American National Standards Institute. This standard was subsequently (in 1980) adopted by the International Standards Organisation (ISO) as an International Standard (IS 1539 : 1980). The language is commonly known as FORTRAN 77 (since the final draft was actually completed in 1977) and is the version of the language now in widespread use. Compilers, which usually support a small number of extensions have, over the years, become *very* efficient. The technology which has been developed during the implementation of these compilers will not be wasted as it can still be applied to Fortran 90 programs.

The venerable nature of Fortran and the rather conservative character of the 1977 standard revision left FORTRAN 77 with a number of old-fashioned facilities that might be termed deficiencies, or at least infelicities. Many desirable features were not available, for example, in FORTRAN 77 it is very difficult to represent data structures succinctly and the lack of any dynamic storage means that all arrays must have a fixed size which can not be exceeded; it was clear from a very early stage that a new, more modern, language needed to be developed. Work began in early 80's on a language known as 'Fortran 8x'. ('x' was expected to be 8 in keeping with the previous names for Fortran.) The work took 12 years partly because of the desire to keep FORTRAN 77 a strict subset and also to ensure that efficiency (one of the bonus's of a simple language) was not compromised. Languages such as Pascal, ADA and Algol are a treat to use but cannot match Fortran for efficiency.

Fortran 90 is a major development of the language but nevertheless it includes all of FORTRAN 77 as a strict subset and so any standard conforming FORTRAN 77 program will continue to be a valid Fortran 90 program. The main reason for this is due to the vast amount of so-called 'dusty deck' programs that populate Scientific installations all over the world. Many man-years have been put into writing these programs which, after so many years of use (i.e., in-the-field testing,) are very reliable. Most of the dusty deck codes will be FORTRAN 77 but there are doubtless many lines of FORTRAN 66 too.

In addition to the old FORTRAN 77 constructs, Fortran 90 allows programs to be expressed in ways that are more suited to a modern computing environment and has rendered obsolete many of the mechanisms that were appropriate in FORTRAN 77. Here, we have tried to present an approach to programming which uses the new features of Fortran 90 to good effect and which will result in portable, efficient, safe and maintainable code. We do not intended to present a complete description of every possible feature of the language.

In Fortran 90 some features of FORTRAN 77 have been replaced by better, safer and more efficient features. Many of these features are to be labelled as depracated and should not be used in new programs. Tools exist to effect automatic removal of such obsolescent features, examples of these are (Pacific Sierra Research's) VAST90 and (NA Software's) LOFT90 In addition, these tools perform differing amounts of vectorisation (transliteration of serial structures to equivalent parallel structures).

As the Fortran 90 standard is very large and complex there are (inevitably) a small number of ambiguities / conflicts / grey areas. Such anomalies often only come to light when compliers are developed. Since standardisation many compilers have been under development and a number of these anomalies have been identified. A new standard, Fortran 95, will recitify these faults and extend the language in the appropriate areas. In the last couple of years the Fortran 90 based language known as High Performance Fortran (HPF) has been developed. This language contains the whole of Fortran 90 and also includes other desirable extensions. Fortran 95 will include many of the new features from HPF [4].

(HPF is intended for programming distributed memory machines and introduced "directives" (Fortran 90 structured comments) to give hints on how to distribute data (arrays) amongst grids of (non-homogeneous) processors. The idea is to relieve the programmer of the burden of writing explicit message-passing code; the compilation system does this instead. HPF also introduced a small number of executable statements (parallel assignments, side effect free (PURE) procedures) which have been adopted by Fortran 95.)

4.2 Drawbacks of FORTRAN 77

By todays standards FORTRAN 77 is outmoded — many other languages have been developed which allow greater expressiveness and ease of programming. The main drawbacks have been identified as:

1. FORTRAN 77 awkward 'punched card' or 'fixed form' source format.

   Each line in a FORTRAN 77 program corresponds to a single punched card with 72 columns meaning that each line can only be 72 characters long. This causes problems because any text in columns 73 onwards is simply ignored (line numbers used to be placed in these columns). In this day and age, this restriction is totally unnecessary.

   Other restrictions of FORTRAN 77:

   - the first 5 columns are reserved for line numbers;
   - the 6th column can only be used to indicate a continuation line;
   - comments can only be initiated by marking the first column with a special character;
   - only upper case letters are allowed anywhere in the program;
   - variable names can only be 6 characters long meaning that mnemonic and cryptic names must be used all the time — maintenance unfriendly!
   - no in-line comments are allowed, comments must be on a line of their own;

2. Lack of inherent parallelism.

   Fortran is supposed to be a performance language — today High Performance Computing is implemented on Parallel machines — FORTRAN 77 has no in-built way of expressing parallelism. In the past calls to specially written vector subroutines have been used or reliance has been made on the compiler to vectorise (parallelise) the sequential (serial) code. It is much more efficient to give the user control of parallelism. This has been done, to a certain extent, by the introduction of parallel 'array syntax'.

3. Lack of dynamic storage.

   FORTRAN 77 only allows static storage for example, this means temporary short-lived arrays cannot be created on-the-fly nor can pointers, which are useful for implementing intuitive data structures, be used. All FORTRAN 77 programs must declare arrays 'big enough' for any future problem size which is an awkward and very unattractive restriction absent in virtually all of the current popular high-level languages.

4. Lack of numeric portability.

   Problems arise with precision when porting FORTRAN 77 code from one machine to another. Many FORTRAN 77 systems implement their own extensions to give greater precision, this means that the code becomes non-portable. Fortran 90 has taken ideas for the various FORTRAN 77 extensions and improved them so that the new language is much more portable that before.

5. Lack of user-defined data structures.

   In FORTRAN 77 intuitive (user-defined) data types were not available as they are in ADA, Algol, C, Pascal etc.. Their presence would make programming more robust and simpler. In FORTRAN 77 there is no way of defining compound objects.

6. Lack of explicit recursion.

   FORTRAN 77 does not support recursion which is a very useful and succinct mathematical technique. In the past this had to be simulated using a user defined stack and access routines to manipulate stack entries. Recursion is a fairly simple and a code efficient concept and was, to all intents and purposes, unavailable.

7. Reliance on unsafe storage and sequence association features.

   In FORTRAN 77, global data is only accessible via the notoriously open-to-abuse `COMMON` block. The rules which applied to `COMMON` blocks are very lax and the user could inadvertently do quite horrendous things! Fortran 90 presents a new method of obtaining global data. Another technique which is equally as open to abuse as `COMMON` blocks is that a user is allowed to alias an array using an `EQUIVALENCE` statement. A great deal of errors stem for mistakes in these two areas which are generally regarded as unsafe, however, there is no real alternative in FORTRAN 77.

4.3 New Fortran 90 Features

Fortran 90 is a major revision of its predecessor many of the inadequacies of FORTRAN 77 have been addressed:

1. Fortran 90 defines a new code format which is much more like every other language;

   - free format — no reliance on specific positioning of special characters;
   - upto 132 columns per line;
   - more than one statement per line;
   - in-line comments allowed (this makes it easier to annotate code);
   - upper and lower case letters allowed (makes code more readable);
   - it is virtually impossible to misplace a character now;
   - longer and more descriptive object names (upto 31 characters);
   - names can be punctuated by underscores making then more readable.

2. Parallelism can now be expressed using whole array operations which include extensive slicing and sectioning facilities. Arithmetic may now be performed on whole arrays and array sections. Operations in an array valued expression are conceptually performed in parallel.

   To support this feature, many parallel intrinsic functions have been introduced including reduction operations such as `SUM` (add all elements in an array and return one value — the sum) and `MAX-VAL` (scan all elements in an array and return one value — the biggest). This concept comes from APL.

   The masked (parallel) assignment (`WHERE`) statement is also a new feature.

3. The introduction of dynamic (heap) storage means that pointers and allocatable arrays can be implemented. Temporary arrays can now be created on-the-fly, pointers can be used for aliasing arrays or sections of arrays and implementing dynamic data structures such as linked lists and trees.

4. To combat non-portability, the `KIND` facility has been introduced. This mechanism allows the user to specify the desired precision of an object in a portable way. Type specifications can be parameterised meaning that the precision of a particular type can be changed by altering the value of one constant. Appropriate intrinsic inquiry functions are defined to query the precision of given objects.

5. User-defined types, which are constructed from existing types, can now be constructed. For example, a type can be defined to represent a 3D coordinate which has three components (x, y, z) or a different type can be used to represent personal details: name, age, sex address, phone number etc. Defining objects in this way is more intuitive and makes programming easier and less error prone.

6. Explicit recursion is now available. For reasons of efficiency the user must declare a procedure to be recursive but it can then be used to call itself.

7. Facility packaging has been introduced — `MODULE`s replace many features of FORTRAN 77. They can be used for global definitions (of types, objects, operators and procedures), and can be used to provide functionality whose internal details are hidden from the user (data hiding).

   Reshaping and retyping functions have been added to the language which means that features that relied on storage association (such a `EQUIVALENCE`) do not have to be used.

- procedure interfaces which declare procedure and argument names and types can now be specified, this means that:
  - programs and procedures can be separately compiled — more efficient development;
  - array bounds do not have to be passed as arguments;
  - array shape can be inherited from the actual argument;
  - better type checking exists across procedure boundaries;
  - more efficient code can be generated.
- new control constructs have been introduced, for example,
  - `DO` ... `ENDDO` (not part of FORTRAN 77) this will reduce use of numeric labels in the program;
  - `DO` ... `WHILE` loop;
  - the `EXIT` command for gracefully exiting a loop;
  - the `CYCLE` command for abandoning the current iteration of a loop and commencing the next one;
  - named control constructs which are useful for code readability;
  - `SELECT CASE` control block. which is more succinct, elegant and efficient than an `IF` .... `ELSEIF` ... `ELSEIF` block.
- Internal procedures are now allowed. A procedure is allowed to contain a further procedure with local scope — it cannot be accessed from outside of the procedure within which it is defined.
- It is now possible to define and overload operators for derived and intrinsic data types which mean that so-called 'semantic extension' can be performed, For example, an arbitrary length integer can be implemented using a linked list structure (one digit per cell) and then all the intrinsic operators defined (overloaded) so that this type can be treated in the same way as all other types. The language thus appears to have been extended. All intrinsic operators `+`, `-`, `*`, `/` and `**` and assignment, `=`, can be overloaded (defined for new types).

- MODULEs also provide object based facilities for Fortran 90, for example, it is possible to use a module to implement the abovementioned long integer data type — all required facilities (for example, definitions, objects, operators, overloaded intrinsics and manipulation procedures) may be packaged together in a MODULE, the user simple has to USE this module to have all the features instantly available. The module can now be used like a library unit.
- An ancillary standard, 'The varying strings module', has been defined and is implemented in a module. When using this module, the user can simply declare an object of the appropriate type (VARYING_STRING) which can be manipulated in exactly the same fashion as the intrinsic character type. All intrinsic operations and procedures are included in the module and are therefore available for all objects of this type. All the implementation details are hidden from the user.
- useful libraries can be written and placed in a module.
- BLOCK DATA subprograms are now redundant since a MODULE can be used for the same purpose.

4.4 Advantages of Additions

The introduction of the previously mentioned new features have had an impact on the general standing of the language. Fortran 90 can be thought of as being: more natural, more flexible, more expressive, more usable, more portable and, above all, safer.

Even though FORTRAN 77 is a subset of Fortran 90 the user should steer away from 'unsafe' features of the new language. Many 'dangerous' features have been superseeded by new features of Fortran 90, for example,

- COMMON blocks — use a module;
- EQUIVALENCE — use TRANSFER; (using EQUIVALENCE also reduces portability);
- reshaping of arrays across procedure boundaries (this used to be possible when using assumed sized arrays) — use the RESHAPE function at the call site;
- retyping of arrays across procedure boundaries — use TRANSFER;
- reliance on labels has been decreased;
- fixed form source — use free form;
- IMPLICIT NONE statement introduced which, if preset, disallows implicit typing of variables. (In FORTRAN 77 undeclared variables were implicitly declared with the type being derived from the first letter of the variable name — this meant that wrongly spelled variable names could pass through a compiler with no warning messages issued at all. If the variable was referenced then a totally random value (the incumbent value of the memory location where the variable resides) would be used with potentially disastrous consequences.
- internal procedures (procedures within procedures with local scope) alleviate the need for the awkward ENTRY statements ans statement functions.

5 Language Obsolescence

Fortran 90 has carried forward the whole of FORTRAN 77 and also a number of features from existing Fortran compilers. This has been done to protect the investment in the millions of lines of code that have been written in Fortran since it was first developed. Inevitably, as modern features have been added, many of the older features have become redundant and programmers, especially those using Fortran 90 for the first time, need to be aware of the possible pit-falls in using them.

Fortran 90 has a number of features marked as obsolescent, this means,

- they are already redundant in FORTRAN 77;
- better methods of programming already existed in the FORTRAN 77 standard;
- programmers should stop using them;
- the standards committee's intention is that many of these features will be removed from the next revision of the language, Fortran 95;

5.1 Obsolescent Features

Fortran 90 includes the whole of FORTRAN 77 as a subset, warts and all, but the specification also flagged some facilities as obsolescent. The following features may well be removed from the next revision of Fortran and *should not be used* when writing new programs. Fortran 90 retains the functionality of these features which can all be better expressed using new syntax:

5.1.1 Arithmetic IF Statement

It is a three way branch statement of the form,

IF(<*expression*>) <*label1*>,<*label2*>,<*label3*>

Here <*expression*> is any expression producing a result of type INTEGER, REAL or DOUBLE PRECISION, and the three labels are statement labels of executable statements. If the value of the expression is negative, execution transfers to the statement labelled <*label1*>. If the expression is zero, transfer is to the statement labelled <*label2*>, and a positive result causes transfer to <*label3*>. The same label can be repeated.

This relic of the original Fortran has been redundant since the early 1960s when the logical IF and computed GOTO were introduced and it should be replaced by an equivalent CASE or IF construct.

5.1.2 ASSIGN Statement

Used to assign a statement label to an INTEGER variable (the label cannot be used as an integer though it is generally used in a GOTO or FORMAT statement.

ASSIGN <*label*> TO <*integer-variable*>

5.1.3 ASSIGNed GOTO Statement

Historically this was used to simulate a procedure call before Fortran had procedures — its use should be replaced by either an IF statement or by a procedure call.

5.1.4 ASSIGNed FORMAT Statement

The ASSIGN statement can be used to assign a label to an integer which is subsequently referred to in an input/output statement.

The same functionality can be obtained by using CHARACTER strings to hold FORMAT specifications. The FORMAT specification can either be this string or a pointer to this string.

5.1.5 Hollerith Format Strings

Used to represent strings in a format statement like this:

```
      WRITE(*,100)
 100  FORMAT(17H TITLE OF PROGRAM)
```

The use of Hollerith strings is out-of-date as strings can now be delimited by single or double quotes:

```
      WRITE(*,100)
 100  FORMAT('TITLE OF PROGRAM')
```

5.1.6 PAUSE Statement

PAUSE was used to suspend execution until a key was pressed on the keyboard.

PAUSE <*stop code*>

The <*stop code*> is written out at the PAUSE new code should use a PRINT statement for the <*stop code*> and a READ statement which waits for input to signify that execution should recommence.

5.1.7 REAL and DOUBLE PRECISION DO-loop Variables

In FORTRAN 77 REAL and DOUBLE PRECISION variables can be used in DO-loop control expressions and index variables. This is unsafe because a loop with real valued DO-loop control expressions could easily iterate a different number of times on different machines — a loop with control expression 1.0,2.0,1.0 may loop once or twice because real valued numbers are only stored approximately, 1.0 + 1.0 could equal, say, 1.99, or 2.01. The first evaluation would execute 2 times whereas the second would only give 1 execution. The solution to this problem is to use INTEGER variables and construct REAL or DOUBLE PRECISION variables within the loop.

The following loop is obsolescent:

```
DO x = 1.0,2.0,1.0
 PRINT*, INT(x)
END DO
PRINT*, x
```

5.1.8 Shared DO-loop Termination

A number of DO loops can currently be terminated on the same (possibly executable) statement — this causes all sorts of confusion, when programs are changed so that the loops do not logically end on a single statement any more.

```
      IF (N < 1) GOTO 100
      DO 100 K=1,N
      DO 100 J=1,N
      DO 100 I=1,N
      ...
  100 A(I,J,K)=A(I,J,K)/2.0
```

The simple solution is to use END DO instead.

5.1.9 Alternate RETURN

This allows a calling program unit to specify labels as arguments to a called procedure as shown. The called procedure can then return control to different points in the calling program unit by specifying an integer parameter to the RETURN statement which corresponds to a set of labels specified in the argument list.

```
       ...
      CALL SUB1(x,y,*98,*99)
       ...
   98 CONTINUE
       ...
   99 CONTINUE
       ...

       SUBROUTINE SUB1(X,Y,*,*)
       ...
       RETURN 1
       ...
       RETURN 2
       END
```

Use an INTEGER return code and a GOTO statement or some equivalent control structure.

5.1.10 Branching to an END IF

FORTRAN 77 allowed branching to an END IF from outside its block, this feature is deemed obsolete so, instead, control should be transferred to the next statement instead or, alternatively, a CONTINUE statement could be inserted.

5.2 Undesirable Features

The following features are not marked as obsolescent yet can, and indeed should be, expressed using new syntax:

5.2.1 Fixed Source Form

Use free form source form as this is less error prone and more intuitive.

5.2.2 Implicit Declaration of Variables

Always use the `IMPLICIT NONE` statement in each program unit, any undeclared variables will be reported at compile time.

5.2.3 `COMMON` Blocks

One of the functions of modules is to provide global data, this will be expanded upon later.

5.2.4 Assumed Size Arrays

It is recommended that instead of assumed-size arrays, assumed-shape arrays be used in new programs instead. With this class of array it is not necessary to explicitly communicate any information regarding bounds to a procedure, however, in this case an explicit interface *must* be provided. If an array is to be reshaped or retyped across a procedure boundary then the new intrinsics `RESHAPE`, `TRANSFER`, `PACK` and `UNPACK` should be used to achieve the desired effect.

5.2.5 `EQUIVALENCE` Statement

`EQUIVALENCE` is often used for retyping or aliasing and sometimes for simulating dynamic workspace. This feature is considered to be unsafe (and non-portable) and should not be used. Fortran 90 offers:

- the `TRANSFER` intrinsic for retyping,
- a `POINTER` variable for aliasing,
- the `ALLOCATABLE` attribute for temporary workspace arrays.

5.2.6 `ENTRY` Statement

`ENTRY` statements allow procedures to be entered at points other than the first line. This facility can be very confusing; if this effect is desired then the code should be reorganised to take advantage of internal procedures.

6 Object Oriented Programming

In the last ten years Object Oriented Design and Object Oriented Programming has become increasingly popular; languages such as C++ and Eiffel have been hailed as the programming languages of the future, but how does Fortran 90 compare?

Programming paradigms have evolved over the years. With each successive refinement of programming style have come improvements in readability, maintainability, reliability, testability, complexity, power, structure and reusability. The first computer programs were written in assembler languages and primitive languages such as (the original) Fortran which possessed only the single statement IF and GOTO statement to control the flow of execution; even procedures had to be simulated using a cunning combination of ASSIGN and GOTO. Clearly the directed graphs of these early programs resembled a bowl of spaghetti with a vast number of logical paths passing through the code in an unstructured fashion. Each path of a program represents a unique combination of predicates which (some argue) must all be tested in order to gain confidence that the program will function correctly. In [3] Davis presents some estimations of the possible number of paths through such an unstructured program: a 100 line program can have up to 10^{158} possible paths.

The next phase of programming style introduced the concept of a **function**; code modules could be written and the program partitioned into logical (and to some extent) reusable blocks. This had the effect of reducing the complexity of the program (less paths) and improving the maintainability, readability and overall structure. A good example of an advanced functional language is FORTRAN 77. If the 100 line program of above could be split into four separate functions then the number of paths would be reduced to 10^{33}.

Functional programming forced the user to adopt better programming practises, however, there still existed many dimly lit back alleys where programmers can perform dirty tricks. In FORTRAN 77 use of the COMMON block is one classic example. Due to the absence of high level data structures (see next paragraph) users were often reluctant to supply seemingly endless lists of objects as arguments to functions; it was much easier to hide things 'behind-the-scenes' in a COMMON block which allows contiguous areas of memory to be available to procedures. In FORTRAN 77 it is also easy to inadvertently retype parts of the memory leading to all sort of problems. For example, the following program was supplied to the Sun f77 compiler which generated one warning:

```
PROGRAM Duffer
 DOUBLE PRECISION ::  INIT
 COMMON /GOOF/ INIT, A, B
  CALL WHOOPS()
  PRINT*, "INIT=", INIT, "A=", A, "B=", B
END

SUBROUTINE WHOOPS()
 COMMON /GOOF/ INIT, A, B
  INIT = 3
  A = 4
  B = 5
END
```

the following output was generated:

```
INIT=    6.9006308436664-314A=    5.00000B=  0.
```

With the introduction of Algol the concept of **structured programming** was born; chaotic jumps around the code were replaced by compact and succinct blocks with a rigidly defined structure. Examples of such structures are loops and if blocks. Data was also allowed to be structured through the use of pointers and structures. Both these features allowed for greater modularity, clarity and more compact code. FORTRAN 77 missed the boat with regards to structured data, however, Fortran 90 has corrected this oversight. With this new structure applied, our 100 line program will now have a maximum of 10^4 paths.

The latest and most fashionable programming paradigm is **object-oriented programming**. This style of programming promotes software reusability, modularity and precludes certain error conditions by packaging together type definitions and procedures for the manipulation of objects of that type. Fortran 90 does not have such an extensive range of object-oriented capabilities as, say, C++ but is comparable to (original) ADA and provides enough to be of great use to the programmer.

6.1 Fortran 90's Object Oriented Facilities

Fortran 90 has some degree of object oriented facilities such as:

- *data abstraction* — user-defined types;
- *data hiding* — PRIVATE and PUBLIC attributes;
- *encapsulation* — Modules and data hiding facilities;
- *inheritance* and *extensibility* — super-types, operator overloading and generic procedures;
- *polymorphism* — users can program their own polymorphism by generic overloading;
- *reusability* — Modules;

FORTRAN 77 had virtually no object oriented features at all, Fortran 90 adds much but by no means all the required functionality. As usual there is a trade off with efficiency. One of the ultimate goals of Fortran 90 is that the code *must* be efficient.

6.1.1 Data Abstraction

It is convenient to use objects which mirror the structure of the entities which they model. In Fortran 90 user derived types provide a certain degree of abstraction and the availability of pointers allows fairly complex data structures to be defined. Pointers are implemented in a different way from many languages and are considerably less flexible (but more efficient) than, say, pointers in C. As Fortran 90 pointers are strongly typed, their targets are limited but this leads to more secure and faster code. Fortran 90 does not support enumerated types but these can be simulated in a (semantic extension) module without too much trouble.

Two main problems are the lack of parameterised derived types and the lack of subtypes.

Parameterised derived types are user-defined types where the individual type components have selectable kinds. The idea would be to define a 'skeleton' type which could be supplied with KIND values in such a way that the individual components are declared with these KINDs. As currently defined, derived types can have kind values for the components but they are fixed, for example the following two types are totally separate:

```
TYPE T1
 INTEGER(KIND=1) :: sun_zoom
 REAL(KIND=1)    :: spock
END TYPE T1
TYPE T2
 INTEGER(KIND=2) :: sun_zoom
 REAL(KIND=2)    :: spock
END TYPE T2
```

If the kind selection could be deferred until object declaration then they could be considered to be parameterised.

Subtypes are, as their name suggests, a subclass of a parent type. A common example may be a positive integer type which has exactly the same properties as an intrinsic `INTEGER` type but with a restricted range. Subtypes are expensive to implement but provide range checking security.

6.1.2 Data Hiding

Fortran 90 supports fairly advanced data hiding. Any object or procedure can have its accessibility defined in a `MODULE` by being given a `PRIVATE` or `PUBLIC` attribute. There is, however, a drawback in that `MODULE`s may inherit functionality from previously written `MODULE`s by using them but objects and functionality that are `PRIVATE` in the use-associated module cannot be seen in the new module. There really needs to be a third type of accessibility which allows `PRIVATE` objects to be accessible outside their module under certain circumstances. This would allow for much greater extensibility.

6.1.3 Encapsulation

Encapsulation refers to bundling related functionality into a library or other such self contained package whilst shielding the user from any worries about the internal structure.

Encapsulation very closely aligned with data hiding and is available through `MODULE`s / `MODULE PROCEDURE`s and `USE` statements and backed up by the ability to rename module entities or to selectively import only specified objects.

6.1.4 Inheritance and Extensibility

Fortran 90 supports supertypes meaning that user-defined types can include other defined types and a hierarchy can be built up, however, this does not apply in the other direction; Fortran 90 does not have subtypes. Due to the lack of subtyping functional and object inheritance cannot be transmitted in this fashion, however, ADA has subtypes but is still not considered to be an object oriented language. Being able to define subtypes which inherit access functions from the parent type is a very important aspect of object oriented programming and Fortran 90 does not support this.

6.1.5 Polymorphism

The generic capabilities of Fortran 90 are very advanced, however, polymorphism is not inherent in the language and must be programmed by the user. Specific procedures must always be user-written and added to the generic interface. In addition Fortran 90 does not support dynamic binding, an example of this is the inability to resolve generic procedure calls at run-time. (The standard forbids generic procedure names to be used as actual procedure arguments, the specific name must be used instead.) Dynamic binding is generally thought to be an expensive feature to have in the language and is not included owing to efficiency concerns.

Many instances of polymorphism are ruled out because of the lack of subtypes.

6.1.6 Reusability

`MODULE`s provide an easy method to access reusable code. Modules can contain libraries of types, object declarations and functions which can be used by other modules. Use of the module facility must be encouraged and traditional FORTRAN 77 programmers must try hard to program in this new way.

6.2 Comparisons with C++

In a paper, *Fortran 90 and Computational Science*, [2] available on the World Wide Web,

`http://csep1.phy.ornl.gov/csep.html`

C++ and Fortran 90 are compared and contrasted in their suitability for computational scientific work.

Module 3: Elements of Fortran 90

7 Fortran 90 Programming

7.1 Example of a Fortran 90 Program

Consider the following example Fortran 90 program:

```
MODULE Triangle_Operations
 IMPLICIT NONE
CONTAINS
 FUNCTION Area(x,y,z)
  REAL :: Area            ! function type
  REAL, INTENT( IN ) :: x, y, z
  REAL :: theta, height
  theta = ACOS((x**2+y**2-z**2)/(2.0*x*y))
  height = x*SIN(theta); Area = 0.5*y*height
 END FUNCTION Area
END MODULE Triangle_Operations

PROGRAM Triangle
 USE Triangle_Operations
 IMPLICIT NONE
 REAL :: a, b, c, Area
 PRINT *, 'Welcome, please enter the&
          &lengths of the 3 sides.'
 READ *, a, b, c
 PRINT *, 'Triangle''s area:  ', Area(a,b,c)
END PROGRAM Triangle
```

The program highlights the following:

- free format source code

  Executable statements do not have to start in or after column 7 as they do in FORTRAN 77.

- `MODULE Triangle_Operations`

  A program unit used to house procedures. (Similar to a C++ 'class'.)

- `IMPLICIT NONE`

  Makes declaration of variables compulsory throughout the module. It applies globally within the `MODULE`.

- `CONTAINS`

  Specifies that the rest of the `MODULE` consists of procedure definitions.

- `FUNCTION Area(x,y,z)`

  This declares the function name and the number and name of its dummy arguments.

- `REAL :: Area ! function type`

  `FUNCTION`s return a result in a variable which has the same name as the function (in this case `Area`). The type of the function result must be declared in either the header or in the declarations.

  The `!` initiates a comment, everything after this character on the same line is ignored by the compiler.

- `REAL, INTENT( IN ) :: x, y, z`

  The type of the dummy arguments must always be declared, they must be of the same type as the actual arguments.

  The `INTENT` attribute says how the arguments are to be used:

  - `IN` means the arguments are used but not (re)defined;
  - `OUT` says they are defined and not used;
  - `INOUT` says that they are used and then redefined.

  Specifying the `INTENT` of an argument is not compulsory but is good practise as it allows the compiler to detect argument usage errors.

- `REAL :: theta, height`

  The final `REAL` declaration statement declares local variables for use in the `FUNCTION`. Such variables cannot be accessed in the calling program, as they are out of *scope*, (not visible to the calling program).

- `theta = ACOS((x**2+y**2-z**2)/(2.0*x*y))`

  This statement assigns a value to `theta` and uses some mathematical operators:

  - `*` - multiplication,
  - `**` - exponentiation;
  - `/` - division
  - `+` - addition,
  - `-` - subtraction

  The brackets (parenthesis) are used to group calculations (as on a calculator) and also to specify the argument to the *intrinsic function* reference `ACOS`.

  Intrinsic functions are part of the Fortran 90 language and cover many areas, the simplest and most common are mathematical functions such as `SIN` and `COS` or `MIN` and `MAX`. Many are designed to act elementally on an array argument, in other words they will perform the same function to every element of an array at the same time.

- `height = x*SIN(theta); Area = 0.5*y*height`

  This highlights two statements on one line. The `;` indicates that a new statement follows on the same line. Normally only one statement per line is allowed.

  The function variable `Area` must be assigned a value or else the function will be in error.

- `END MODULE` — this signifies the end of the module.

- `PROGRAM Triangle`

  The `PROGRAM` statement is not strictly necessary but its inclusion is good practice. There may only be one per program.

- `USE Triangle_Operations` — tells the program to attach the specified module. This will allow the `Area` function to be called from within the `Triangle` program.
- `IMPLICIT NONE`

  An `IMPLICIT NONE` statement turns off implicit typing making the declaration of variables compulsory.
- `REAL :: a, b, c, Area`

  Declaration of real valued objects. `a, b` and `c` are variables and `Area` is a function name. This function must be declared because its name contains a value.
- `PRINT *, 'Welcome, please enter the& ...`

  This `PRINT` statement writes the string in quotes to the standard output channel (the screen). The `&` at the end of the line tells compiler that the line continues and the `&` at the start of the text tells the compiler to insert one space and continue the previous line at the character following the `&`. If the `&` were at the start of the line, or if there were no `&` on the second line, then the string would have a large gap in it as the indentation would be considered as part of the string.
- `READ *, a, b, c`

  This `READ` statement waits for three things to be input from the standard input (keyboard). The entities should be separated by a space. Digits will be accepted and interpreted as real numbers; things other than valid numbers will cause the program to crash.
- `PRINT *, 'Triangle''s area:  ', Area(a,b,c)`

  This `PRINT` statement contains a function reference, `Area`, and the output of a string.

  The `''` is an *escaped character*, and is transformed, on output, to a single `'`. Typing just a single quote here would generate an error because the compiler would think it had encountered the end of the string and would flag the character `s` as an error. The whole string could be enclosed by double quotes (`"`) which would allow a single quote (`'`) to be used within the string without confusion. The same delimiter must be used at each end of the string.

  The function call invokes the `FUNCTION` with the values of `a, b` and `c` being substituted for `x, y` and `z`.

  - `a, b` and `c` are known as actual-arguments,
  - `x, y` and `z` are known as dummy-arguments.
  - `a, b` and `c` and `x, y` and `z` are said to be argument associated
  - cannot refer to `a, b` and `c` in the function they are not in scope.
- `END PROGRAM Triangle`

  An `END PROGRAM` statement terminates the main program.

7.2 Coding Style

It is recommended that the following coding convention is adopted:

- Fortran 90 keywords, intrinsic functions and user defined type names and operators should be in upper case and user entities should be in lower case but may start with a capital letter.
- indentation should be 1 or 2 spaces and should be applied to the bodies of program units, control blocks, `INTERFACE` blocks, etc.

- the names of program units are always included in their `END` statements,
- argument keywords are always used for optional arguments,
- always use `IMPLICIT NONE`.

Please note: In order that a program fits onto a page these rules are sometimes relaxed here.

Adopting a specific and documented coding style will make the code easier to read, and will allow other people to be instantly familiar with the style of code.

8 Language Elements

8.1 Source Form

The most basic rules in any programming language are those which govern the source form. These rules determine exactly how statements in a program are entered into the computer and how they are displayed. The source form rules are analogous to those fundamentals of natural language which define the alphabet used to express the words in the language, the punctuation symbols to be used, how sentences are to be written (left to right, up and down, right to left, etc.). These are the rules we are going to describe here.

8.2 Free Format Code

Fortran 90 supports the new free format source form:

- 132 characters per line;
- extended character set;
- '&' line continuation character;
- '`!`' comment initiator;
- '`;`' statement separator;
- significant blanks.

Fortran 90 has two basically incompatible source forms, an old form compatible with that used in FORTRAN 77, **fixed format**, and a new form more suited to the modern computing environment, **free format**. The old form used a very strictly defined layout of program statements on lines. This was well suited to the expression of programs when the main method of entering programs into a computer was by the use of stacks of cards with holes punched in them to represent the characters of the program statements. Such punched card systems also worked with a very restricted set of characters; only upper-case letters, for example, were allowed. The new form is designed to be easier to prepare and to read using the sort of keyboard / display that is now ubiquitous. This new form uses a much wider character set and it allows much greater freedom in laying out statements on a line. It is this new form that we are going to describe here since this is the form we would expect any new programmer to employ. The old form will not be described in any great detail.

The old "character in column 6" method of line continuation is replaced by an & at the end of a line. Only one & is essential however if a string is to be split then there should also be an & at the beginning of the text on the next line in order to preserve the spacing. The continued line effectively begins from the character after the &.

```
PRINT*, 'This is a long constant continued from line to line&
        &by use of the continuation mark at both the end of the&
        &continued line and at the start of the continuation line'
```

& is ineffective within a comment and cannot be on a line on its own.

```
INTEGER SQUASHED, UP; REAL CLOSE  ! Two for the price of one
```

The ';' allows two statements (or more) to occupy the same line and ! signifies that the following text is a comment.

In free format blanks become significant. The rules are fairly straightforward: blanks cannot occur in the middle of names, keywords or literals and, in general, blanks must appear between keywords and between keywords and names. See Section 8.4 for more details.

Question 3: Reformatting Code

The following program (which is available by anonymous ftp from `ftp.liv.ac.uk` in the directory `/pub/f90courses/progs`, filename `BasicReformatQuestion.f90`) has been badly laid out, Reformat it so its is neat and readable but performs exactly the same function,

```
PROGRAM MAIN;INTEGER::degreesfahrenheit&
  ,degreescentigrade;READ*,&
  degreesfahrenheit;degreescentigrade&
  =5*(degreesfahrenheit-32)/9;PRINT*,&
  degreesCENtiGrAde;END
```

8.3 Character Set

The statements of a Fortran 90 program are expressed using a carefully defined restricted character set. This character set has been chosen so that it is available on almost every machine currently in use. The Fortran 90 character set can be reproduced by virtually every display, printer or other input / output device which uses characters. It consists of the set of alphanumeric characters and a limited set of punctuation marks and other special symbols. The alphanumeric characters are the upper case letters, A-Z, the lower case letters, a-z, the digits, 0-9, and the underscore character, _. There is no difference between upper and lower case letters in Fortran 90. The allowed punctuation and other special symbols are shown in the following table.

Symbol	Description	Symbol	Description
	space	=	equal
+	plus	-	minus
*	asterisk	/	slash
(	left paren	)	right paren
,	comma	.	period
'	single quote	"	double quote
:	colon	;	semicolon
!	shriek	&	ampersand
%	percent	<	less than
>	greater than	$	dollar
?	question mark		

The last one in each column, $ and ?, do not have any special meaning in the language. Most modern computer systems are able to process a somewhat wider character set than this, however, this set is chosen so that it will be available on virtually every computing system in every country in the world. The characters found on any particular machine in excess of these are very likely to either not exist on some other machine or have very different graphic representation. Apart from textual output, the case of a letter is insignificant to Fortran 90 (unlike say, C,) so the different cases can therefore be used to improve the readability of the program.

A collating sequence is defined ([1] Section 4.3.2) and is standard ASCII; see Section 33.

8.4 Significant Blanks

In free format programs spaces, or blanks, in statements can be thought of as being significant. Consecutive spaces / blanks have the same meaning as one blank. Blanks are not allowed in language keywords. For instance, `INT EGER` is not the same as `INTEGER`. Neither are blanks allowed in names defined by the programmer. The name `hours` could not be written as `ho urs`; not that any one would be likely to do so. If a name is made up from two words it is common practice to use an underscore as a separator. The continuation character & acts as a space and so cannot be used in the middle of names and keywords.

Keywords and names may not be run together without spaces or some other "punctuation" to separate them. Thus,

```
SUBROUTINEClear
```

would be illegal and not equivalent to

```
SUBROUTINE Clear
```

Likewise,

```
INTEGER :: a
```

is valid and

```
INT EGER :: a
```

is not.

Blanks must appear:

- between two separate keywords
- between keywords and names not otherwise separated by punctuation or other special characters.

```
INTEGER FUNCTION fit(i) ! is valid
INTEGERFUNCTION fit(i)  ! is not
INTEGER FUNCTIONfit(i)  ! is not
```

Blanks are optional between certain pairs of keywords such as: END <*construct*>, where <*construct*> is DO, FUNCTION, PROGRAM, MODULE, SUBROUTINE and so on. Blanks are mandatory is other situations. Adding a blank between two valid English words is generally a good idea and will be valid.

Apart from observing the above rules, spaces can be used liberally to increase the readability of programs. Thus the following statements,

```
POLYN=X**3-X**2+3*X-2.0
```

and

```
POLYN = X**3 - X**2 + 3*X - 2.0
```

are equivalent.

To sum up, the rules governing the use of blanks are largely common sense. Judicious use of blanks and sensible use of upper and lower case can often significantly clarify more complicated constructs or statements, and some systematic convention for this use can make it very much easier to read and hence to write correct programs. As statements can start anywhere on a line most programmers use indentation to highlight control structure nesting for example within IF blocks and DO loops.

8.5 Comments

It is always very good practice to add descriptive comments to programs. On any line a ! character indicates that all subsequent characters up to the end of the line are commentary. Such commentary is ignored by the compiler. Comments are part of the program source but are intended for the human reader and not for the machine.

```
    PROGRAM Saddo
!
! Program to evaluate marriage potential
!
    LOGICAL :: TrainSpotter ! Do we spot trains?
    LOGICAL :: SmellySocks  ! Have we smelly socks?
    INTEGER :: i, j         ! Loop variables
```

The one exception to above is if the ! appears in a character context, for example, in

```
PRINT*, "No chance of ever marrying!!!"
```

the ! does not initiate a comment.

A comment can contain any of the characters available on the computer being used. A comment is not restricted to use the Fortran character set. A program containing comments using characters outside this set may still be portable but the comments may look somewhat odd if the program is moved to another system with a different extended set of characters.

With modern compilation systems it is often possible to give the compiler 'hints' by annotating programs with specifically structured comments which are interpreted and acted upon to produce efficient code. The HPF (High Performance Fortran) language is implemented in this way.

```
! Next line is an HPF directive
!HPF$ PROCESSORS P(3,3,3)
! Switch to fixed format
!DIR$ FIXED_FORM
```

8.6 Names

The Fortran language defines a number of names, or keywords, such as PRINT, INTEGER, MAX, etc. The spelling of these names is defined by the language. There are a number of entities that must be named by the programmer, such as variables, procedures, etc. These names must be chosen to obey a few simple rules. A name can consist of up to 31 alphanumeric characters (letters or digits) plus underscore. The first character in any name must be a letter. In names, upper and lower case letters are equivalent. The following are valid Fortran names,

```
A, aAa, INCOME, Num1, N1205, under_score
```

The following declarations are incorrect statements owing to invalid names,

```
INTEGER :: 1A      ! does not begin with a letter
INTEGER :: A_name_made_up_of_more_than_31_letters ! too long, 38 characters
INTEGER :: Depth:0 ! contains an illegal character ":"
INTEGER :: A-3     ! subtract 3 from A is meaningless here
```

The underscore should be used to separate words in long names

```
CHARACTER(LEN=12) :: user_name ! valid name
CHARACTER(LEN=12) :: username  ! different valid name
```

With this and the new long names facility in Fortran 90 symbolic names should be readable, significant and descriptive. It is worth spending a minute or so choosing a good name which other programmers will be able to understand.

8.7 Statement Ordering

Fortran has some quite strict rules about the order of statements. Basically in any program or procedure the following rules must be used:

1. The program heading statement must come first, (PROGRAM, FUNCTION or SUBROUTINE). A PROGRAM statement is optional but its use is recommended.
2. All the specification statements must precede the first executable statement. Even though DATA statements may be placed with executable text it is far clearer if they lie in the declaration area. It is also a good idea to group FORMAT statements together for clarity.
3. The executable statements must follow in the order required by the logic of the program.
4. The program or procedure must terminate with an END statement.

Within the set of specification statements there is relatively little ordering required, however, in general if one entity is used in the specification of another, it is normally required that it has been previously defined. In other words, named constants (PARAMETERs) must be declared before they can be used as part of the declaration of other objects.

The following table details the prescribed ordering:

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER statement</td><td>IMPLICIT statements</td></tr>
<tr><td>PARAMETER and DATA statements</td><td>Derived-Type Definition, Interface blocks, Type declaration statements, Statement function statements and specification statements</td></tr>
<tr><td>DATA statements</td><td>Executable constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal or module procedures</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Execution of a program begins at the first executable statement of the MAIN PROGRAM, when a procedure is called execution begins with the first executable statement after the invoked entry point. The non-executable statements are conceptually 'executed' (!) simultaneously on program initiation, in other words they are referenced once and once only when execution of the main program begins.

There now follows a explanation of the table,

- There can be only 1 main PROGRAM,

- there may be many uniquely named FUNCTIONs and SUBROUTINEs program units (procedures).

- there may be any number of uniquely named MODULEs which are associated with a program through a USE statement. Modules are very flexible program units and are used to package a number of facilities (for example, procedures, type definitions, object declarations, or semantic extensions). Their use is very much encouraged and they replace a number of unsafe features of FORTRAN 77.

- There can be only one BLOCK DATA subprogram — these will not be described as part of this course — their purpose is to define global constants or global initialisation and this is best done by a MODULE and USE statement.

- USE statement:

  This attaches a module whose entities become *use-associated* with the program unit. When a module is used its public contents are accessible as if they had been declared explicitly in the program unit. Modules may be pre-compiled (like a library) or may be written by the programmer.

  Any global entities should be placed in a module and then used whenever access is required.

- The IMPLICIT NONE statement should be placed after a USE statement. Its use is implored.

- FORMAT and ENTRY statements. Format statements should be grouped together somewhere in the code. ENTRY statements provide a mechanism to 'drop' into a procedure halfway through the executable code. Their use is outmoded and strongly discouraged owing to its dangerous nature.

- PARAMETER statement and IMPLICIT statement:

  IMPLICIT statements should not be used, IMPLICIT NONE should be the only form of implicit typing considered. (IMPLICIT statements allow the user to redefine the implication of an object's first letter in determining what its implicit type will be if it is not declared. Cannot have an IMPLICIT statement if there is an IMPLICIT NONE line.)

  PARAMETER statements, it is suggested that the attributed (Fortran 90) style of PARAMETER declaration be used.

- DATA statements should (but do not have to) be placed in the declaration, common practice puts them after the declarations and before the executables.

- Interface blocks are generally placed at the head of the declarations and are grouped together.

  Statement functions are a form of in-line statement definition, internal procedures should be used instead.

- executable statements are things like DO statements, IF constructs and assignment statements.

- CONTAINS separates the "main" program unit from any locally visible internal procedures.

- the internal procedures follow the same layout as a (normal) procedure except that they cannot contain a second level of internal procedure.

- the END statement is essential to delimit the current program unit.

9 Data Objects

9.1 Intrinsic Types

Fortran 90 has three broad classes of object type,

- character;
- boolean;
- numeric.

these give rise to five simple intrinsic types, known a default types,

- `CHARACTER` for strings of one or more characters;
- `LOGICAL` for objects which have the values true or false;
- `REAL` (and `DOUBLE PRECISION`) for approximate, possibly fractional numbers;
- `INTEGER` for exact whole numbers;
- `COMPLEX` for representing numbers of the form: $\mathbf{x} + i\mathbf{y}$.

For example,

```
CHARACTER          :: sex  ! letter
CHARACTER(LEN=12) :: name ! string
LOGICAL            :: wed  ! married?
REAL               :: height
DOUBLE PRECISION   :: pi   ! 3.14...
INTEGER            :: age  ! whole No.
COMPLEX            :: val  ! x + iy
```

Each type has

- a name
- a set of valid values
- a means to denote values
- a set of operators

Note,

- Most programming languages have the same broad classes of objects.
- The three broad classes cannot be intermixed without some sort of type coercion being performed.
- `REAL` and `DOUBLE PRECISION` objects are approximate. `DOUBLE PRECISION` should not now be used. In FORTRAN 77 an object of this type had greater precision than `REAL`, in Fortran 90 the precision of a `REAL` object may be specified making the `DOUBLE PRECISION` data type redundant.
- All numeric types have finite range.
- A default type is not parameterised by a kind value.

9.2 Literal Constants

A literal constant is an entity with a fixed value:

```
+12345      ! INTEGER
2.          ! REAL
1.0         ! REAL
-6.6E-06    ! REAL
-6.6D-06    ! DOUBLE PRECISION
.FALSE.     ! LOGICAL
'Mau''dib'  ! CHARACTER
"Mau'dib"   ! CHARACTER
```

Note,

- there are only two `LOGICAL` values.
- integers are represented by a sequence of digits with a + or − sign, + signs are optional.
- `REAL` constants contain a decimal point or an exponentiation symbol, `INTEGER` constants do not.
- character literals are delimited by the double or single quote symbols, `"` and `'`.
- two occurrences of the delimiter inside a string produce one occurrence on output; for example `'Mau''dib'` but not `"Mau''dib"` because of the differing delimiters;
- there is only a finite range of values that numeric literals can take.
- constants may also include a kind specifier.

9.3 Implicit Typing

If a variable is referenced but not declared (in its scoping unit) then, by default, it is implicitly declared. The type that it assumes depends on the first letter,

- `I, J, K, L, M` and `N` represent integers;
- all other letters define real variables.

This short cut is potentially very dangerous as a misspelling of a variable name can declare and assign to a brand new variable with the user being totally unaware.

Implicit typing should **always** be turned off by adding:

```
IMPLICIT NONE
```

as the first line after any `USE` statements. In this way the user will be informed of any undeclared variables used in the program.

Consider,

```
    DO 30 I = 1.1000
     ...
 30 CONTINUE
```

in fixed format with implicit typing this declares a `REAL` variable `DO30I` and sets it to `1.1000` instead of performing a loop 1000 times! Legend has it that the example sighted caused the crash of the American Space Shuttle. An expensive missing comma!

9.4 Numeric and Logical Declarations

Variables of a given type should be declared in type declaration statements at the start of a program unit. A simplified syntax follows,

< *type* > [,< *attribute*-list >] `::` < *variable*-list > [=< *value* >]

The `::` is actually optional, however, it does no harm to use it, moreover, if < *attribute*-list > or =< *value* > are present then the `::` is obligatory.

The following are all valid declarations,

```
    REAL :: x
    INTEGER :: i,j
    LOGICAL, POINTER :: ptr
    REAL, DIMENSION(10,10) :: y, z(10)
    DOUBLE PRECISION, DIMENSION(0:9,0:9) :: w
```

The `DIMENSION` attribute declares a 10×10 array, this can be overridden as with `z` which is declared as a 1D array with 10 elements.

< *attribute*-list > represents a list of attributes such as `PARAMETER`, `SAVE`, `INTENT`, `POINTER`, `TARGET`, `DIMENSION`, (for arrays) or visibility attributes. An object may be given more than one attribute per declaration but some cannot be mixed (such as `PARAMETER` and `POINTER`).

9.5 Character Declarations

Character variables can be declared in a similar way to numeric types using a `CHARACTER` statement. `CHARACTER` variables can

- refer to one character;
- refer to a string of characters which is achieved by adding a length specifier to the object declaration.

A simplified syntax follows,

< *type* >[(`LEN=`< *length-spec* >)] [,< *attribute*-list >] [`::`] < *variable*-list > [=< *value* >]

If < *attribute*-list > or =< *value* > are present then so must be `::`.

The following are all valid declarations,

```
CHARACTER(LEN=10)  :: name
CHARACTER          :: sex
CHARACTER(LEN=32)  :: str
```

In the same way as the `DIMENSION` attribute was overridden in the example of Section 9.4 so can the string length declaration (specified by `LEN=`); this is achieved using the * notation. If a `DIMENSION` specifier is present it can also be overridden. The length specifier must come after the dimension if both are being overridden.

```
CHARACTER(LEN=10) :: name, sex*1, str*32
CHARACTER(LEN=10), DIMENSION(10,10) :: tom(10)*2, dick, harry(10,10,10)*20
CHARACTER, POINTER :: P2ch
```

The first line is exactly the same as the previous declaration.

There is a substantial difference between a character variable of 10 characters (`CHARACTER(LEN=10)` or `CHARACTER*10`) and an array of 10 elements; the first is scalar and the second is non-scalar.

Other attributes can be added in the same way as for numeric types (see Section 9.4).

9.6 Constants (Parameters)

Symbolic constants, oddly known as *parameters* in Fortran, can easily be set up either in an attributed declaration or in a `PARAMETER` statement (it is recommended that the attributed form be used):

```
INTEGER price_of_fags            ! F77 style - not recommended
PARAMETER (price_of_fags = 252)  ! F77 style - not recommended
REAL, PARAMETER ::  pi = 3.14159 ! F90 style
```

`CHARACTER` constants can assume their length from the associated literal (`LEN=*`), they can also be declared to be fixed length:

```
CHARACTER(LEN=*), PARAMETER :: son = 'bart', dad = "Homer"
```

Parameters should be used:

- if it is known that a variable will only take one value;

  The variable is forced to be unchanged throughout the program. If any statement tries to change the value an error will result.

- for legibility where a 'magic value' occurs in a program such as π;

  Symbolic names are easier to understand than a meaningless number.

- for maintainability when a 'constant' value could feasibly be changed.

  The value of the constant can be adjusted throughout the program by changing 1 line.

9.7 Initialisation

When a program is loaded into the computers memory, the contents of declared variables are normally undefined, it is clearly useful to override this and give variables useful initial values. For example,

```
INTEGER          :: i = 5, j = 100
REAL             :: max = 10.D5
CHARACTER(LEN=5) :: light = 'Amber'
CHARACTER(LEN=9) :: gumboot = 'Wellie'
LOGICAL          :: on = .TRUE., off = .FALSE.
```

For `CHARACTER` variables, if the object and initialisation expression are of different lengths then either:

- the object will be padded with blanks on the RHS, or
- the initialisation expression will be truncated on the right so it will fit.

Variables can be initialised in a number of ways

- `PARAMETER` statements,
- `DATA` statements,
- type declaration statements (with an =< *expression* >) clause.

Limited expressions known as *initialisation expressions* can also be used in type declaration statements. These expression must be able to be evaluated when the program is compiled — if you can't work out the values then neither can the compiler. Initialisation expressions can contain `PARAMETER`s or literals. Arrays may be initialised by specifying a scalar value or by using a conformable array constructor, `(/.../)`.

```
REAL, PARAMETER :: pi = 3.141592
REAL :: radius = 3.5
REAL :: circum = 2 * pi * radius
INTEGER :: a(1:4) = (/1,2,3,4/)
```

In general, intrinsic functions *cannot* be used in initialisation expressions, however, the following intrinsics may be used:

- `REPEAT`,
- `RESHAPE`,
- `SELECTED_INT_KIND`,
- `SELECTED_REAL_KIND`,
- `TRANSFER`,
- `TRIM`.
- `LBOUND`,

- UBOUND,
- SHAPE,
- SIZE,
- KIND,
- LEN,
- BIT_SIZE,
- numeric inquiry intrinsics, for, example, HUGE, TINY, EPSILON.

In this context the arguments to these functions must be initialisation expressions.

9.8 Examples of Declarations

The following declarations show how careful choice of data type, name and inclusion of comments can help readability:

```
CHARACTER(LEN=20) :: Location   ! Name of hospital
CHARACTER :: Ward               ! Ward, e.g., A, B, C etc
INTEGER   :: NumBabiesBorn = 0  ! Sum total of births
REAL      :: TimeElapsed = 0.0  ! Time since 1st birth (hours)
REAL      :: MaxTimeTwixtBirths = 0.0 ! longest gap between births
REAL      :: AveTimeTwixtBirths ! average gap between births
REAL      :: TimeSinceLastBirth ! gap since the last birth
LOGICAL   :: NHS                ! Is it an NHS hospital
```

There is nothing wrong with using verbose variable names and augmenting declarations with comments explaining their use.

It is important to use an appropriate data type for an object, for example, the above variable, Location, is the name of a hospital, it is clearly appropriate to use a CHARACTER string here. (We have assumed that the name can be represented in 20 letters.) The second variable, Ward only needs to contain one letter of the alphabet, this is reflected in its declaration. It is also a good idea to initialise any variables that are used to hold some sort of counter. In the above code fragment, we have 3 such examples: NumBabiesBorn is incremented by one each time there is a new birth, this value must always be a whole number so INTEGER is the appropriate data type; TimeElapsed measures the time in hours since the first birth, in order to be accurate, we need to be able to represent fractions of hours so a REAL variable is in order, when the program begins to run zero time will have elapsed which explains the initialisation; the final example is MaxTimeTwixtBirths the longest spell of time between births, again it is a good idea to initialise the variable to a sensible value owing to the way it will probably be used in the program. It is likely that the following will appear;

```
IF (TimeSinceLastBirth > MaxTimeTwixtBirths) THEN
  MaxTimeTwixtBirths = TimeSinceLastBirth
END IF
```

The AveTimeTwixtBirths variable will also have to represent a non-whole number so REAL is the obvious choice. The variable, TimeSinceLastBirth is used to store the current time gap between births and is, in this sense, a temporary variable. LOGICAL variables are ideal when one of two values needs to be stored, here we wish to know whether the hospital is NHS-funded or not.

Question 4: Declaration Format

Which of the following are incorrect declarations and why? (If you think a declaration may be correct in a given situation then say what the situation would be.)

1. `ReAl :: x`
2. `CHARACTER :: name`
3. `CHARACTER(LEN=10) :: name`
4. `REAL :: var-1`
5. `INTEGER :: 1a`
6. `BOOLEAN :: loji`
7. `DOUBLE :: X`
8. `CHARACTER(LEN=5) :: town = "Glasgow"`
9. `CHARACTER(LEN=*) :: town = "Glasgow"`
10. `CHARACTER(LEN=*), PARAMETER :: city = "Glasgow"`
11. `INTEGER :: pi = +22/7`
12. `LOGICAL :: wibble = .TRUE.`
13. `CHARACTER(LEN=*), PARAMETER :: "Bognor"`
14. `REAL, PARAMETER :: pye = 22.0/7.0`
15. `REAL :: two_pie = pye*2`
16. `REAL :: a = 1., b = 2`
17. `LOGICAL(LEN=12) :: frisnet`
18. `CHARACTER(LEN=6) :: you_know = 'y'know"`
19. `CHARACTER(LEN=6) :: you_know = "y'know"`
20. `INTEGER :: ia ib ic id`
    (in free format source form)
21. `DOUBLE PRECISION :: pattie = +1.0D0`
22. `DOUBLE PRECISION :: pattie = -1.0E-0`
23. `LOGICAL, DIMENSION(2) bool`
24. `REAL :: poie = 4.*atan(1.)`
25. declare the following objects,

Name	**Status**	**Type**	**Initial Value**
feet	variable	integer	-
miles	variable	real	-
town	variable	character ($\leq$ 20 letters)	-
home_town	constant	character	<*your home town*>
in_uk	constant	logical	<*is your home town in UK?*>
sin_half	constant	real	$\sin(0.5) = 0.47942554$

10 Expressions and Assignment

10.1 Expressions

Each of the three broad type classes has its own set of intrinsic (in-built) operators, these are combined with operands to form expressions. Expressions are made from one operator (e.g., +, -, *, /, // and **) and at least one operand. Operands are also expressions.

Expressions have types derived from their operands; they are either of intrinsic type or a user defined type. For example,

- `NumBabiesBorn+1` — numeric valued
- `"Ward "//Ward` — character valued
- `TimeSinceLastBirth .GT. MaxTimeTwixtBirths` — logical valued

In addition to the intrinsic operations:

- operators may be defined by the user, for example, `.INVERSE.`. These defined operators (with 1 or 2 operands) are specified in a procedure and can be applied to any type or combination of types. The operator functionality is given in a procedure which must then be mentioned in an interface block. Such operators are very powerful when used in conjunction with derived types and modules as a package of objects and operators.
- intrinsic operators may be overloaded; when using a derived type the user can specify exactly what each existing operator means in the context of this new type.

10.2 Assignment

Expressions are often used in conjunction with the assignment operator, =, to give values to objects. This operator,

- is defined between all intrinsic numeric types. The two operands of = (the LHS and RHS) do not have to be the same type.
- is defined between two objects of the same user-defined type.
- may be explicitly overloaded so that assignment is meaningful in situations other than those above.

Examples,

```
a = b
c = SIN(.7)*12.7
name = initials//surname
bool = (a.EQ.b.OR.c.NE.d)
```

The LHS is an object and the RHS is an expression.

10.3 Intrinsic Numeric Operations

The following operators are valid for numeric expressions,

- ** exponentiation, a dyadic ("takes two operands") operator, for example, 10**2, (evaluated right to left);
- * and / multiply and divide, dyadic operators, for example, 10*7/4;
- + and - plus and minus or add and subtract, a monadic ("takes one operand") and dyadic operators, for example, -4 and 7+8-3;

All the above operators can be applied to numeric literals, constants, scalar and array objects with the only restriction being that the RHS of the exponentiation operator must be scalar.

Example,

```
a = b - c
f = -3*6/5
```

Note that operators have a predefined precedence, which defines the order that they are evaluated in, (see Section 10.7).

Question 5: Area Of a Circle

Write a simple program to read in the radius and calculate the area of the corresponding circle and volume of the sphere. Demonstrate correctness by calculating the area and volume using radii of 2, 5, 10 and -1.

Area of a circle,

$$\text{area} = \pi r^2 \tag{1}$$

Volume of a sphere,

$$\text{volume} = \frac{4\pi r^3}{3} \tag{2}$$

Hint 1: place the READ, the area calculation and the PRINT statement in a loop as follows. A program template (which is available by anonymous ftp from ftp.liv.ac.uk in the directory /pub/f90courses/progs, filename BasicAreaOfCircleQuestion.f90) is given below.

```
PROGRAM Area
 DO
  PRINT*, "Type in the radius, a negative value will terminate"
  READ*, radius
  IF (radius .LT. 0) EXIT
   ... area calculation
   PRINT*, "Area of circle with radius ",&
           radius, " is ", area
   PRINT*, "Volume of sphere with radius ",&
           radius, " is ", volume
```

```
END DO
END PROGRAM Area
```

In this way when a negative radius is supplied the program will terminate.

Hint 2: use the value 3.14159 for π.

10.4 Relational Operators

The following relational operators deliver a logical result:

- .GT. — greater than.
- .GE. — greater than or equal to.
- .LE. — less than or equal to.
- .LT. — less than.
- .NE. — not equal to.
- .EQ. — equal to.

for example,

```
i .GT. 12
```

is an expression delivering a .TRUE. or .FALSE. result.

These above operators are equivalent to the following:

- $>$ — greater than.
- $>=$ — greater than or equal to.
- $<=$ — less than or equal to.
- $<$ — less than.
- $/=$ — not equal to.
- $=$ — equal to.

for example,

```
i > 12
```

Both sets of symbols may be used in a single statement.

Relational operators:

- compare the values of two operands.

- deliver a logical result.
- can be applied to numeric operands (restrictions on `COMPLEX` which can only use `.EQ.` and `.NE.`).
- can be applied to default `CHARACTER` objects — both objects are made to be the same length by padding the shorter with blanks. Operators refer to ASCII order (see Appendix 33).
- cannot be applied to `LOGICAL` objects, for example,

  ```
  (bool .NE. .TRUE.)
  ```

  is not a valid expression but,

  ```
  (.NOT.bool)
  ```

  is.
- are used (in scalar) form in `IF` statements (see Section 11.1) and elementally in the `WHERE` statement (see Section 15.17).

Consider,

```
bool = i.GT.j
IF (i.EQ.j) c = D
IF (i == j) c = D
```

The example demonstrates,

- simple logical assignment using a relational operator,
- `IF` statements using both forms of relational operators,

When using real-valued expressions (which are approximate) `.EQ.` and `.NE.` have no real meaning.

```
REAL :: Tol = 0.0001
IF (ABS(a-b) .LT. Tol) same = .TRUE.
```

10.5 Intrinsic Logical Operations

A `LOGICAL` or boolean expression returns a `.TRUE.` / `.FALSE.` result. The following operators are valid with `LOGICAL` operands,

- `.NOT.` — monadic negation operator which gives `.TRUE.` if operand is `.FALSE.`, for example, `.NOT.(a .LT. b)`.
- `.AND.` — logical and operator. `.TRUE.` if both operands are `.TRUE.`.
- `.OR.` — logical or operator. `.TRUE.` if at least one operand is `.TRUE.`.
- `.EQV.` — `.TRUE.` if both operands are the same.
- `.NEQV.` — `.TRUE.` if both operands are different.

The following are examples of logical expressions,

```
REAL :: a, b, x, y
LOGICAL :: l1, l2
 ...
l1 = (.NOT.(x.EQ.y.AND.a.EQ.b))
l2 = (l1.EQV.((x.GT.a.OR.y.LT.b).NEQV.a.EQ.b))
```

10.6 Intrinsic Character Operations

10.6.1 Character Substrings

Consider,

```
CHARACTER(LEN=*), PARAMETER :: string = "abcdefgh"
```

substrings can be taken,

- `string` is '`abcdefgh`' (the whole string),
- `string(1:1)` is '`a`' (the first character),
- `string(2:4)` is '`bcd`' (2nd, 3rd and 4th characters),
- `string(1)` is an error (the substring must be specified *from* a position *to* a position, a single subscript is no good).
- `string(1:)` is '`abcdefgh`'.
- `string(:1)` is '`a`'.

10.6.2 Concatenation

There is only one intrinsic character operator, the concatenation operator, //. Most string manipulation is performed using intrinsic functions.

```
CHARACTER(LEN=4), PARAMETER  :: name = "Coal"
CHARACTER(LEN=10) :: song = "Firey "//name
PRINT*, "Rowche "//"Rumble"
PRINT*, song(1:1)//name(2:4)
```

would produce

```
Rowche Rumble
Foal
```

The example joins together two strings and then the first character of `song` and the 2nd, 3rd and 4th of `name`. Note that // cannot be mixed with other types or kinds.

10.7 Operator Precedence

The following table depicts the order in which operators are evaluated:

Operator	Precedence	Example
user-defined monadic	Highest	.INVERSE.A
**	·	10**4
* or /	·	89*55
monadic + or -	·	-4
dyadic + or -	·	5+4
//	·	str1//str2
.GT., >,.LE., <=, etc	·	A > B
.NOT.	·	.NOT.Bool
.AND.	·	A.AND.B
.OR.	·	A.OR.B
.EQV. or .NEQV.	·	A.EQV.B
user-defined dyadic	Lowest	X.DOT.Y

In an expression with no parentheses, the highest precedence operator is combined with its operands first; In contexts of equal precedence left to right evaluation is performed except for **.

Other relevant points are that the intrinsically defined order cannot be changed which means that user defined operators have a fixed precedence (at the top and bottom of the table depending on the operator type). The operator with the highest precedence has the *tightest binding*; from the table user-defined monadic operators can be seen to be the most tightly binding.

The ordering of the operators is intuitive and is comparable to the ordering of other languages. The evaluation order can be altered by using parenthesis; expressions in parentheses are evaluated first. In the context of equal precedence, left to right evaluation is performed except for ** where the expression is evaluated from right to left. This is important when teetering around the limits of the machines representation. Consider A-B+C and A+C-B, if A were the largest representable number and C is positive and smaller than B; the first expression is OK but the second will crash the program with an overflow error.

One of the most common pitfalls occurs with the division operator — it is good practice to put numerator and denominator in parentheses. Note:

```
(A+B)/C
```

is not the same as

```
A+B/C
```

but

```
(A*B)/C
```

is equivalent to

```
A*B/C
```

This is because the multiplication operator binds tighter than the addition operator, however,

```
A/B*C
```

is not equivalent to

```
A/(B*C)
```

because of left to right evaluation.

The syntax is such that two operators cannot be adjacent; one times minus one is written `1*(-1)` and not `1*-1`. (This is the same as in most languages.)

10.8 Precedence Example

The precedence is worked out as follows

1. in an expression find the operator(s) with the tightest binding.
2. if there are more than one occurrence of this operator then the separate instances are evaluated left to right.
3. place the first executed subexpression in brackets to signify this.
4. continue with the second and subsequent subexpressions.
5. move to next most tightly binding operator and follow the same procedure.

It is easy to make mistakes by forgetting the implications of precedence. The following expression,

```
x = a+b/5.0-c**d+1*e
```

is equivalent to

```
x = ((a+(b/5.0))-(c**d))+1*e
```

The following procedure has been followed to parenthesise the expression,

- □ tightest binding operator is `**`. This means `c**d` is the first executed subexpression so should be surrounded by brackets.
- □ `/` and `*` are the second most tightly binding operators and expressions involving them will be evaluated next, put `b/5.0` and `1*e` in brackets.

- + and − have the next highest precedence. Since they are of equal precedence, occurrences are evaluated from left
- at last the assignment is made.

Likewise, the following expression,

```
.NOT.A.OR.B.EQV.C.AND.D.OR..NOT.E
```

is equivalent to

```
((.NOT.A).OR.B).EQV.((C.AND.D).OR.(.NOT.E))
```

here,

- the tightest binding operator is: `.NOT.` followed by `.AND.` followed by `.OR.`.
- the two subexpressions containing the monadic `.NOT.` are effectively evaluated first, as there are two of these the leftmost, `.NOT.A` is done first followed by `.NOT.E`.
- the subexpression `C.AND.D` is evaluated next followed by `.OR.` (left to right)
- finally the `.EQV.` is executed

Parentheses can be added to any expression to modify the order of evaluation.

Question 6: Operator Precedence

Rewrite the following expression so that it contains the equivalent symbolic relational operators and then add parenthesis to indicate the order of evaluation,

```
.NOT.A.AND.B.EQV.C.OR.D.AND.E.OR.x.GT.y.AND.y.eq.z
```

Add parenthesis to this expression to indicate the order of evaluation,

```
-a*b-c/d**e/f+g**h+1-j/k
```

10.9 Precision Errors

Each time two real numbers are combined there is a slight loss of accuracy in the result. After many such operations such 'round-off' errors become noticeable. Catastrophic accuracy loss often arises because two values that are almost equal are subtracted, the subtraction may cancel the leading digits and promotes errors very rapidly from low order digits to high order ones.

For example, consider, the numbers `.123456` and `.123446`; these may be approximated in memory as `.123457` and `.123445` respectively, whereas the true difference is `.100000D-4` the representation may give `.130000D-4`, a 30% error.

```
x = 0.123456; y = 0.123446
PRINT*, "x = ",x," y = ",y
PRINT*, "x-y = ",x-y," but should be 0.100d-4"
```

May produce:

```
x = 0.123457  y = 0.123445
x-y = 0.130d-4 but should be 0.100d-4
```

A whole branch of Numerical Analysis is dedicated to minimising this class of errors in algorithms.

Module 4: Control Constructs, Intrinsics and Basic I/O

11 Control Flow

All structured programming languages need constructs which provide a facility for conditional execution. The simplest method of achieving this functionality is by using a combination of IF and GOTO which is exactly what FORTRAN 66 supported. Fortran has progressed since then and now includes a comprehensive basket of useful control constructs. Fortran 90 supports:

- conditional execution statements and constructs, (IF statements, and IF ... THEN ... ELSEIF... ELSE ... END IF);

  These are the basic conditional execution units. They are useful if a section of code needs to be executed depending on a series of logical conditions being satisfied. If the first condition in the IF statement is evaluated to be true then the code between the IF and the next ELSE, ELSEIF or ENDIF is executed. If the predicate is false then the second branch of the construct is entered. This branch could be either null, an ELSE or an ELSEIF corresponding to no action, the default action or another evaluation of a different predicate with execution being dependent upon the result of the current logical expression. Each IF statement has at least one branch and at most one ELSE branch and may contain any number of ELSEIF branches. Very complex control structures can be built up using multiply nested IF constructs.

  Before using an IF statement, a programmer should be convinced that a SELECT CASE block or a WHERE assignment block would not be more appropriate.

- loops, (DO ... END DO);

  This is the basic form of iteration mechanism. This structure allows the body of the loop to be executed a number of times. The number of iterations can be a constant, a variable (expression) or can be dependent on a particular condition being satisfied. DO loops can be nested.

- multi-way choice construct, (SELECT CASE);

  A particular branch is selected depending upon the value of the case expression. Due to the nature of this construct it very often works out (computationally) cheaper than an IF block with equivalent functionality. This is because in a SELECT CASE block a single control expression is evaluated once and then its (single) result is compared with each branch. With an IF .. ELSEIF block a different control expression must be evaluated at each branch. Even if all control expressions in an IF construct were the same and were simply compared with different values, the general case would dictate that the SELECT CASE block is more efficient.

and less importantly,

- unconditional jump statements, (GOTO);

  Direct jump to a labelled line. This is a very powerful statement, it is very useful and very open to abuse. Unstructured jumps can make a program virtually impossible to follow, the GOTO must be used with care. It is particularly useful for handling exceptions, that is to say, when emergency action is needed to be taken owing to the discovery of an unexpected error.

- I/O exception branching, (ERR=, END=, EOR=);

  This is a slightly oddball feature of Fortran in the sense that there is currently no other form of exception handling in the language. (The feature originated from FORTRAN 77.) It is possible to add qualifiers to I/O statements to specify a jump in control should there be an unexpected I/O data error, end of record or should the end of a file be encountered.

  It is always good practice to use at least the ERR= qualifier in I/O statements.

11.1 IF Statement

This is the most basic form of conditional execution in that there is only an option to execute one statement or not — the general IF construct allows a block of code to be executed. The basic form is,

IF(< *logical-expression* >)< *exec-stmt* >

If the .TRUE. / .FALSE. valued expression, < *logical-expression* >, evaluates to .TRUE. then the < *exec-stmt* > is executed otherwise control of the program passes to the next line. This type of IF statement still has its use as it is a lot less verbose than a block-IF with a single executable statement.

For example,

```
IF (logical_val) A = 3
```

If logical_val is .TRUE. then A gets set to 3 otherwise it does not.

A logical expression is often expressed as:

< *expression* >< *relational-operator* >< *expression* >

For example,

```
IF (x .GT. y) Maxi = x
IF (i .NE. 0 .AND. j .NE. 0) k = 1/(i*j)
IF (i /= 0 .AND. j /= 0) k = 1/(i*j) ! same
```

As REAL numbers are represented approximately, it makes little sense to use the .EQ. and .NE. relational operators between real-valued expressions. For example there is no guarantee that 3.0 and 1.5 * 2.0 are equal. If two REAL numbers / expressions are to be tested for equality then one should look at the size of the difference between the numbers and see if this is less than some sort of tolerance.

```
REAL :: Tol = 0.0001
IF (ABS(a-b) .LT. Tol) same = .TRUE.
```

Consider the IF statement

```
IF (I > 17) Print*, "I > 17"
```

this maps onto the following control flow structure,

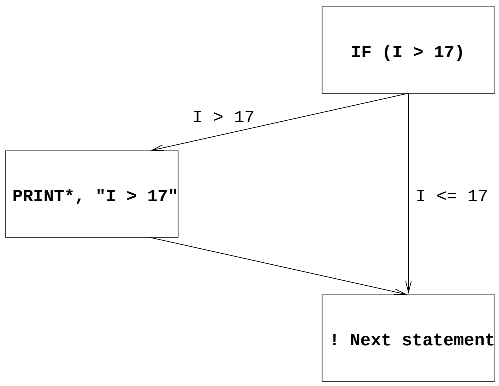


Figure 1: Schematic Diagram of an IF Statement

11.2 IF Construct

The block-IF is more flexible than the single statement IF since there can be a number of alternative mutually exclusive branches guarded by (different) predicates. The control flow is a lot more structured than if a single statement IF plus GOTO statements were used. The scenario is that the predicates in the IF or ELSEIF lines are tested in turn, the first one which evaluates as true transfers control to the appropriate inner block of code; when this has been completed control passes to the ENDIF statement and thence out of the block. If none of the predicates are true then the $<$ *else-block* $>$ (if present) is executed.

The simplest form of the IF construct is equivalent to the IF statement, in other words, if a predicate is .TRUE. then an action is performed.

Consider the IF ... THEN construct

```
IF (I > 17) THEN
  Print*, "I > 17"
END IF
```

this maps onto the following control flow structure,

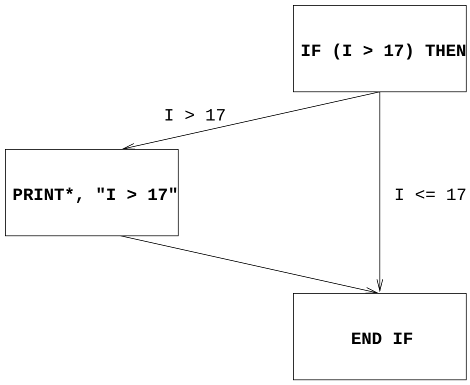


Figure 2: Visualisation of an IF ... THEN Construct

It is a matter of personal taste whether the above construct or just the simple IF statement is used for this sort of case.

The IF construct may contain an ELSE branch, a simple example could be,

```
IF (I > 17) THEN
  Print*, "I > 17"
ELSE
  Print*, "I <= 17"
END IF
```

this maps onto the following control flow structure,

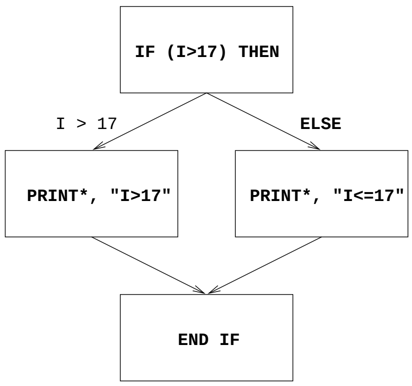


Figure 3: Visualisation of an IF ... THEN ... ELSE Construct

The construct may also have an ELSEIF branch:

```
IF (I > 17) THEN
```

```
  Print*, "I > 17"
ELSEIF (I == 17)
  Print*, "I == 17"
ELSE
  Print*, "I < 17"
END IF
```

Both ELSE and ELSEIF are optional and there can be any number of ELSEIF branches. The above maps to the following control flow structure

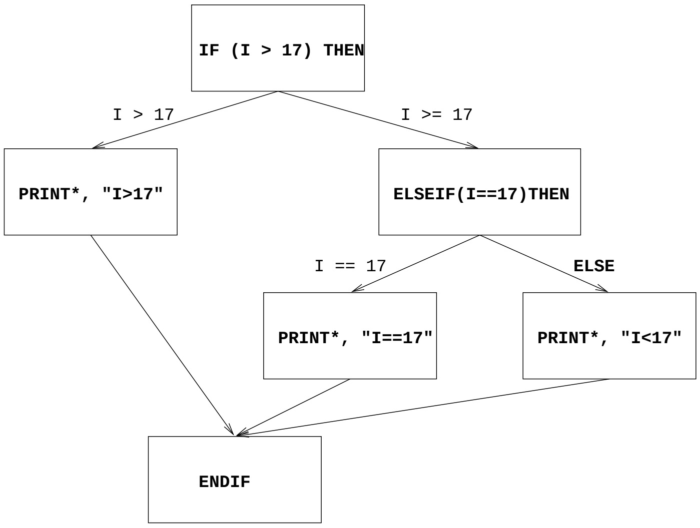


Figure 4: Visualisation of an IF ... THEN ... ELSEIF Construct

The formal syntax is,

[< *name* >:]IF(< *logical-expression* >)THEN
        < *then-block* >
[ELSEIF(< *logical-expression* >)THEN [< *name* >]
        < *elseif-block* >
...]
[ELSE [< *name* >]
        < *else-block* >]
END IF [< *name* >]

The first branch to have a .TRUE. valued < *logical-expression* > is the one that is executed. If none are found then the < *else-block* >, if present, is executed.

For example,

```
IF (x .GT. 3) THEN
 CALL SUB1
ELSEIF (x .EQ. 3) THEN
 CALL SUB2
```

```
ELSEIF (x .EQ. 2) THEN
 CALL SUB3
ELSE
 CALL SUB4
ENDIF
```

(A further IF construct may appear in the < *then-block* >, the < *else-block* > or the < *elseif-block* >. This is now a nested IF structure.)

Statements in either the < *then-block* >, the < *else-block* > or the < *elseif-block* > may be labelled but jumps to such labelled statements are permitted only from within the block containing them. Entry into a block-IF construct is allowed only via the initial IF statement. Transfer out of either the < *then-block* >, the < *else-block* > or the < *elseif-block* > is permitted but only to a statement entirely outside of the whole block defined by the IF...END IF statements. A transfer within the same block-IF between any of the blocks is not permitted.

Certain types of statement, e.g., END SUBROUTINE, END FUNCTION or END PROGRAM, statement are not allowed in the < *then-block* >, the < *else-block* > or the < *elseif-block* >.

11.3 Nested and Named IF Constructs

All control constructs can be both named and nested, for example,

```
outa: IF (a .NE. 0) THEN
        PRINT*, "a /= 0"
        IF (c .NE. 0) THEN
          PRINT*, "a /= 0 AND c /= 0"
        ELSE
          PRINT*, "a /= 0 BUT c == 0"
        ENDIF
      ELSEIF (a .GT. 0) THEN outa
        PRINT*, "a > 0"
      ELSE outa
        PRINT*, "a must be < 0"
      ENDIF outa
```

Here the names are only cosmetic and are intended to make the code clearer (*cf* DO-loop names which do). If a name is given to the IF statement then it *must* be present on the ENDIF statement but not necessarily on the ELSE or ELSEIF statement. If a name is present on the ELSE or ELSEIF then it must be present on the IF statement.

The example has two nested and one named IF block. Nesting can be to any depth (unless the compiler prohibits this, even if it does the limit will almost certainly be configurable).

Even though construct names are only valid within the block that they apply to, their scope is the whole program unit. This means that a name may only be used once in a scoping unit even though no confusion would arise if it were re-used. (See Section 17.8 for a discussion of scope.)

Question 7: The 'Triangle Program'

Write a program to accept three (INTEGER) lengths and report back on whether these lengths could define an equilateral, isosoles or scalene triangle (3, 2 or 0 equal length sides) or whether they cannot form a triangle.

Demonstrate that the program works by classifying the following:

1. (1, 1, 1)
2. (2, 2, 1)
3. (1, 1, 0)
4. (3, 4, 5)
5. (3, 2, 1)
6. (1, 2, 4)

[Hint: If three lengths form a triangle then 2 times the longest side must be less than the sum of all three sides. In Fortran 90 terms, the following must be true:

```
(2*MAX(side1,side2,side3) < side1+side2+side3)
```

]

11.4 Conditional Exit Loops

A loop comprises a block of statements that are executed cyclically. When the end of the loop is reached, the block is repeated from the start of the loop. Loops are differentiated by the way they are terminated. Obviously it would not be reasonable to continue cycling a loop forever. There must be some mechanism for a program to exit from a loop and carry on with the instructions following the End-of-loop.

The block of statements making up the loop is delimited by `DO` and `END DO` statements. This block is executed as many times as is required. Each time through the loop, the condition is evaluated and the first time it is true the `EXIT` is performed and processing continues from the statement following the next `END DO`. Consider,

```
i = 0
DO
  i = i + 1
  IF (i .GT. 100) EXIT
  PRINT*, "I is", i
END DO
! if i>100 control jumps here
PRINT*, "Loop finished. I now equals", i
```

this will generate

```
I is   1
I is   2
```

```
I is   3
  ....
I is   100
Loop finished. I now equals   101
```

This type of conditional-exit loop is useful for dealing with situations when we want input data to control the number of times the loop is executed.

The statements between the DO and its corresponding END DO must constitute a proper block. The statements may be labelled but no transfer of control to such a statement is permitted from outside the loop-block. The loop-block may contain other block constructs, for example, DO, IF or CASE, but they must be contained completely; that is they must be properly nested.

An EXIT statement which is not within a loop is an error.

11.5 Conditional Cycle Loops

Situations often arise in practice when, for some exceptional reason, it is desirable to terminate a particular pass through a loop and continue immediately with the next repetition or cycle; this can be achieved in Fortran 90 by arranging that a CYCLE statement is executed at an appropriate point in the loop.

For example,

```
i = 0
DO
  i = i + 1
  IF (i >= 50 .AND. i <= 59) CYCLE
  IF (i > 100) EXIT
  PRINT*, "I is", i
END DO
PRINT*, "Loop finished. I now equals", i
```

this will generate

```
I is   1
I is   2
  ....
I is   49
I is   60
  ....
I is   100
Loop finished. I now equals   101
```

Here CYCLE forces control to the **innermost** DO statement (the one that contains the CYCLE statement) and the loop begins a new iteration.

In the example, the statement:

```
IF (i >= 50 .AND. i <= 59) CYCLE
```

if executed, will transfer control to the DO statement. The loop must still contain an EXIT statement in order that it can terminate.

A CYCLE statement which is not within a loop is an error.

11.6 Named and Nested Loops

Sometimes it is necessary to jump out of more than the innermost DO loop. To allow this, loops can be given names and then the EXIT statement can be made to refer to a particular loop. An analogous situation also exists for CYCLE,

```
0|    outa: DO
1|     inna: DO
2|      ...
3|      IF (a.GT.b) EXIT outa  ! jump to line 9
4|      IF (a.EQ.b) CYCLE outa ! jump to line 0
5|      IF (c.GT.d) EXIT inna  ! jump to line 8
6|      IF (c.EQ.a) CYCLE      ! jump to line 1
7|     END DO inna
8|    END DO outa
9|    ...
```

The (optional) name following the EXIT or CYCLE highlights which loop the statement refers to.

For example,

```
IF (a.EQ.b) CYCLE outa
```

causes a jump to the first DO loop named outa (line 0).

Likewise,

```
IF (c.GT.d) EXIT inna
```

jumps to line 9.

If the name is missing then the directive is applied, as usual, to the next outermost loop so

```
IF (c.EQ.a) CYCLE
```

causes control to jump to line 1.

The scope of a loop name is the same as that of any construct name.

Question 8: Mathematical Magic

If you take a positive integer, halve it if it is even or triple it and add one if it is odd, and repeat, then the number will eventually become one.

Set up a loop containing a statement to read in a number (input terminated by zero) and to print out the sequence obtained from each input. The number 13 is considered to be very unlucky and if it is obtained *as part of the sequence* then execution should *immediately* terminate with an appropriate message.

Demonstrate that your program works by outputting the sequences generated by the following sets of numbers:

1. 7
2. 106, 46, 3, 0

11.7 DO ... WHILE Loops

If a condition is to be tested at the top of a loop a DO ... WHILE loop could be used,

```
DO WHILE (a .EQ. b)
 ...
END DO
```

The loop only executes if the logical expression evaluates to .TRUE.. Clearly, here, the values of a or b must be modified within the loop otherwise it will never terminate.

The above loop is functionally equivalent to,

```
DO; IF (a .NE. b) EXIT
 ...
END DO
```

EXIT and CYCLE can still be used in a DO WHILE loop, just as there could be multiple EXIT and CYCLE statements in a regular loop.

11.8 Indexed DO Loop

Loops can be written which cycle a fixed number of times. For example,

```
DO i = 1, 100, 1
 ...
END DO
```

is a DO loop that will execute 100 times; it is exactly equivalent to

```
DO i = 1, 100
 ...
END DO
```

The syntax is as follows,

```
DO <DO-var>=<expr1>,<expr2> [ ,<expr3> ]
      <exec-stmts>
END DO
```

The loop can be named and the <*exec-stmts*> could contain EXIT or CYCLE statements, however, a WHILE clause cannot be used but this can be simulated with an EXIT statement if desired.

The number of iterations, which is evaluated **before** execution of the loop begins, is calculated as

```
MAX(INT((<expr2>-<expr1>+<expr3>)/<expr3>),0)
```

in other words the loop runs from <*expr1*> to <*expr2*> in steps of <*expr3*>. If this gives a zero or negative count then the loop is not executed. (It seems to be a common misconception that Fortran loops always have to be executed once — this came from FORTRAN 66 and is now totally incorrect. Zero executed loops are useful for programming degenerate cases.)

If <*expr3*> is absent it is assumed to be 1.

The iteration count is worked out as follows (adapted from the standard, [1]):

1. <*expr1*> is calculated,
2. <*expr2*> is calculated,
3. <*expr3*>, if present, is calculated,
4. the DO variable is assigned the value of <*expr1*>,
5. the iteration count is established (using the formula given above).

The execution cycle is performed as follows (adapted from the standard):

1. the iteration count is tested and if it is zero then the loop terminates.
2. if it is non zero the loop is executed.
3. (conceptually) at the END DO the iteration count is decreased by one and *the* DO *variable is incremented by* <expr3>. (Note how the DO variable can be greater than <*expr2*>.)
4. control passes to the top of the loop again and the cycle begins again.

More complex examples may involve expressions and loops running from high to low:

```
DO i1 = 24, k*j, -1
 DO i2 = k, k*j, j/k
  ...
 END DO
END DO
```

An indexed loop could be achieved using an induction variable and EXIT statement, however, the indexed DO loop is better suited as there is less scope for error.

The DO variable cannot be assigned to within the loop.

11.8.1 Examples of Loop Counts

There now follows a few examples of different loops,

1. upper bound not exact,

   ```
   loopy: DO i = 1, 30, 2
    ... ! 15 iterations
   END DO loopy
   ```

   According to the rules (given earlier) the fact that the upper bound is not exact is not relevant. The iteration count is INT(29/2) = 14, so i will take the values 1,3,..,27,29 and finally 31 although the loop is not executed when i holds this value, this is its final value.

2. negative stride,

   ```
   DO j = 30, 1, -2
    ... ! 15 iterations
   END DO
   ```

   similar to above except the loop runs the other way (high to low). j will begin with the value 30 and will end up being equal to 0.

3. a zero-trip loop,

   ```
   DO k = 30, 1, 2
    ... ! 0 iterations
    ... ! loop skipped
   END DO
   ```

   This is a false example in the sense that the loop bounds are literals and there would be no point in coding a loop of this fashion as it would never ever be executed! The execution is as follows, firstly, k is set to 30 and then the iteration count would be evaluated and set to 0. This would mean that the loop is skipped, the only consequence of its existence being that k holds the value 30.

4. missing stride — assume it is 1,

   ```
   DO l = 1,30
    ... ! i = 1,2,3,...,30
    ... ! 30 iterations
   END DO
   ```

   As the stride is missing it must take its default value which is 1. This loop runs from 1 to (30 so the implied stride means that the loop is executed 30 times.

5. missing stride,

   ```
   DO l = 30,1
    ... ! zero-trip
   END DO
   ```

   As the stride is missing it must take its default value which is 1. This loop runs from high to low (30 to 1) so the implied stride means that the loop is not executed. The final value of l will be 30.

11.9 Scope of DO Variables

Fortran 90 is not block structured; all DO variables are visible after the loop and have a specific value. The index variable is recalculated at the top of the loop and then compared with < *expr2* >, if the loop has finished, execution jumps to the statement after the corresponding END DO. The loop is executed three times and i is assigned to 4 times, the index variable will retain the value that it had just been assigned. For example,

```
DO i = 4, 45, 17
 PRINT*, "I in loop = ",i
END DO
PRINT*, "I after loop = ",i
```

will produce

```
I in loop =  4
I in loop =  21
I in loop =  38
I after loop =  55
```

Elsewhere in the program, the index variable may be used freely but in the loop it can only be referenced and must not have its value changed.

11.10 SELECT CASE Construct

The SELECT CASE Construct is similar to an IF construct. It is a useful control construct if one of several paths through an algorithm must be chosen based on the value of a particular expression.

```
SELECT CASE (i)
  CASE (3,5,7)
    PRINT*,"i is prime"
  CASE (10:)
    PRINT*,"i is > 10"
  CASE DEFAULT
    PRINT*, "i is not prime and is < 10"
END SELECT
```

The first branch is executed if i is equal to 3, 5 or 7, the second if i is greater than or equal to 10 and the third branch if neither of the previous has already been executed.

A slightly more complex example with the corresponding IF structure given as a comment,

```
  SELECT CASE (num)

   CASE (6,9,99,66)
 ! IF(num==6.OR. .. .OR.num==66) THEN
    PRINT*, "Woof woof"

   CASE (10:65,67:98)
```

```
! ELSEIF((num.GE.10.AND.num.LE.65) .OR. ...
   PRINT*, "Bow wow"

  CASE (100:)
! ELSEIF (num.GE.100) THEN
   PRINT*, "Bark"

  CASE DEFAULT
! ELSE
   PRINT*, "Meow"

 END SELECT
! ENDIF
```

Important points are,

- the < *case-expr* > in this case is `num`.
- the first < *case-selector* >, `(6,9,99,66)` means "if num is equal to either 6, 9, 66 or 99 then",
- the second < *case-selector* >, `(10:65,67:98)` means "if num is between 10 and 65 (inclusive) or 67 and 98 (inclusive) then",
- `(100:)` specifies the range of greater than or equal to one hundred.
- if a case branch has been executed then when the next < *case-selector* > is encountered control jumps to the `END SELECT` statement.
- if a particular case expression is not satisfied then the next one is tested.

(An `IF` .. `ENDIF` construct could be used but a `SELECT CASE` is neater and more efficient.)

`SELECT CASE` is more efficient than `ELSEIF` because there is only one expression that controls the branching. The expression needs to be evaluated once and then control is transferred to whichever branch corresponds to the expressions value. An `IF` .. `ELSEIF` ... has the potential to have a different expression to evaluate at each branch point making it less efficient.

Consider the `SELECT CASE` construct,

```
SELECT CASE (I)
 CASE(1);   Print*, "I==1"
 CASE(2:9); Print*, "I>=2 and I<=9"
 CASE(10);  Print*, "I>=10"
 CASE DEFAULT; Print*, "I<=0"
END SELECT CASE
```

this maps onto the following control flow structure,

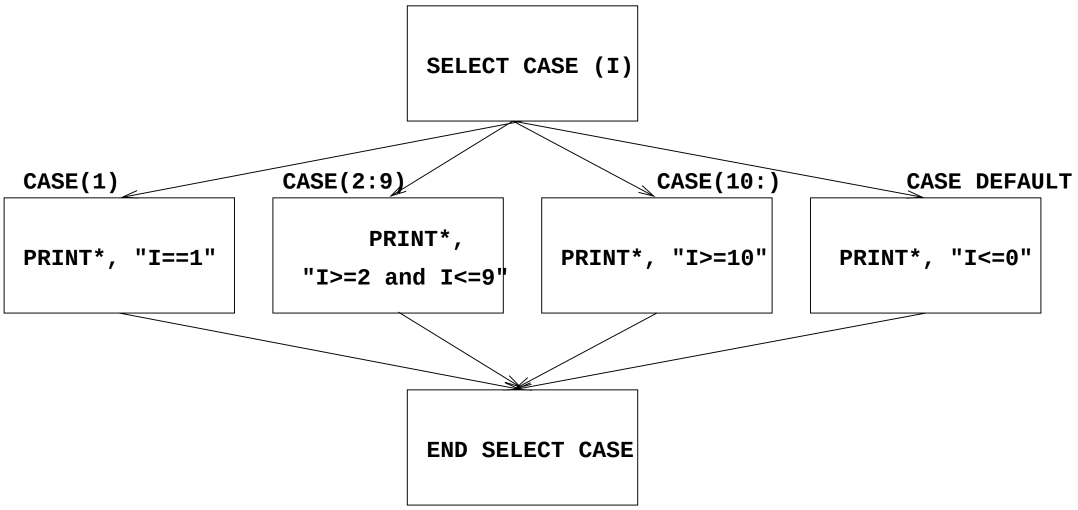


Figure 5: Visualisation of a SELECT CASE Construct

The syntax is as follows,

[<*name*>:] SELECT CASE <*case-expr*>
    [CASE <*case-selector*> [<*name*>]
        <*exec-stmts*>] ...
    [CASE DEFAULT [<*name*>]
        <*exec-stmts*>]
END SELECT [<*name*>]

Note,

- ☐ the <*case-expr*> must be scalar and INTEGER, LOGICAL or CHARACTER valued;
- ☐ there may be any number of general CASE statements but only one CASE DEFAULT branch;
- ☐ the <*case-selector*> must be a parenthesised single value or a range (section with a stride of one), for example, (.TRUE.) or (99:101). A range specifier is a lower and upper limit separated by a single colon. One or other of the bounds is optional giving an open ended range specifier.
- ☐ the <*case-expr*> is evaluated and compared with the <*case-selector*>s in turn to see which branch to take.
- ☐ if no branches are chosen then the CASE DEFAULT is executed (if present).
- ☐ when the <*exec-stmts*> in the selected branch have been executed, control jumps out of the CASE construct (via the END SELECT statement).
- ☐ as with other similar structures it is not possible to jump into a CASE construct.

CASE constructs may be named — if the header is named then so must be the END SELECT statement. If any of the CASE branches are named then so must be the SELECT statement and the END statement

A more complex example is given below, this also demonstrates how SELECT CASE constructs may be named.

```
  ...
 outa: SELECT CASE (n)
  CASE (:-1) outa
   M = -1
  CASE (1:) outa
   DO i = 1, n
    inna: SELECT CASE (line(i:i))
     CASE ('@','&','*','$')
      PRINT*, "At EOL"
     CASE ('a':'z','A':'Z')
      PRINT*, "Alphabetic"
     CASE DEFAULT
      PRINT*, "CHAR OK"
    END SELECT inna
   END DO
  CASE DEFAULT outa
   PRINT*, "N is zero"
  END SELECT outa
```

Analysis:

- the first `SELECT CASE` statement is named `outa` and so is the corresponding `END SELECT` statement.
- the <*case-expr*> in this case is `n` this is evaluated first and its value stored somewhere.
- the first <*case-selector*>, `(:-1)` means "if n is less than or equal to `-1`" so if this is true then this branch is executed — when the next <*case-selector*> is encountered, control jumps to the `END SELECT` statement.
- if the above case expression is not satisfied then the next one is tested. `(1:)` specifies the range of greater than or equal to one. If `n` satisfies this then the `DO` loop containing the nested `CASE` construct is entered. If it does not then the `DEFAULT` action is carried out. In this case this branch corresponds to the value 0, the only value not covered by the other branches.
- the inner case structure demonstrates a scalar `CHARACTER` <*case-expr*> which is matched to a list of possible values or, indeed, a list of possible ranges. If the character substring `line(i:i)` matches a value from either the list or list of ranges then the appropriate branch is executed. If it is not matched then the `DEFAULT` branch is executed. Note a `CHARACTER` substring **cannot** be written as `line(i)`
- this inner case structure is executed `n` times, (as specified by the loop,) and then control passes back to the outer case structure. Since the executable statements of the case branch have now been completed, the construct is exited.

12 Mixing Objects of Different Types

12.1 Mixed Numeric Type Expressions

When an (sub)expression is evaluated, the actual calculation in the CPU must be between operands of the same type, this means if the expression is of mixed type, the compiler must automatically convert (promote or coerce) one type to another. Default types have an implied ordering

1. INTEGER — lowest
2. REAL
3. DOUBLE PRECISION
4. COMPLEX — highest

thus if an INTEGER is mixed with a REAL the INTEGER is promoted to a REAL and then the calculation performed; the resultant expression is of type REAL.

For example,

- INTEGER * REAL gives a REAL, (3*2.0 is 6.0)
- REAL * INTEGER gives a REAL, (3.0*2 is 6.0)
- DOUBLE PRECISION * REAL gives DOUBLE PRECISION,
- COMPLEX * <*anytype*> gives COMPLEX,
- DOUBLE PRECISION * REAL * INTEGER gives DOUBLE PRECISION.

Consider the expression,

```
int*real*dp*c
```

the types are coerced as follows:

1. int to REAL
2. int*real to DOUBLE PRECISION
3. (int*real)*dp to COMPLEX.

The above expression is therefore COMPLEX valued.

Note that numeric and non-numeric types cannot be mixed using intrinsic operators, nor can LOGICAL and CHARACTER.

In general one must think hard and long about mixed mode arithmetic!

12.2 Mixed Type Assignment

When the RHS expression of a mixed type assignment statement has been evaluated it has a specific type, this type must then be converted to fit in with the LHS. This conversion could be either a promotion or a relegation. For example,

- INTEGER = REAL (or DOUBLE PRECISION)

  The RHS needs relegating to be an INTEGER value. The right hand side is evaluated and then the value is truncated (all the decimal places lopped off) then assigned to the LHS.

- ☐ REAL (or DOUBLE PRECISION) = INTEGER

  The INTEGER needs promoting to become a REAL. The right hand side expression is simply stored (approximately) in the LHS.

For example, as real values are stored approximately,

```
REAL :: a = 1.1, b = 0.1
INTEGER :: i, j, k
i = 3.9       ! i will be 3
j = -0.9      ! j will be 0
k = a - b     ! k will be 1 or 0
```

Notes:

- ☐ since i is INTEGER, the value 3.9 must be *truncated*, integers are always formed by truncating *towards* zero.
- ☐ j (an INTEGER,) would be truncated to 0.
- ☐ the result of a - b would be close to 1.0 (it could be 1.0000001 or it could be 0.999999999), so, because of truncation, k could contain either 0 or 1.

Care must be taken when mixing types!

12.3 Integer Division

If one integer divides another in a subexpression then the type of that subexpression is INTEGER. Confusion often arises about integer division; in short, division of two integers produces an integer result by truncation (towards zero).

Consider,

```
REAL :: a, b, c, d, e
a = 1999/1000
b = -1999/1000
c = (1999+1)/1000
d = 1999.0/1000
e = 1999/1000.0
```

- ☐ a is (about) 1.000. The integer expression 1999/1000 is evaluated and then truncated towards zero to produce an integral value, 1. Its says in the Fortran 90 standard, [1], P84 section 7.2.1.1, "The result of such an operation [integer division] is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively."
- ☐ b is (about) -1.000 for the same reasons as above.
- ☐ c is (about) 2.000 because, due to the parentheses 2000/1000 is calculated.
- ☐ d and e are (about) 1.999 because both RHS's are evaluated to be real numbers, in 1999.0/1000 and 1999/1000.0 the integers are promoted to real numbers before the division.

Question 9: Decimal to Roman Numerals Conversion

Using a SELECT CASE block and integer division write a program that reads in a decimal number between 0 and 999 and prints out the equivalent in Roman Numerals.

Demonstrate that your program works with the numbers:

1. 888
2. 0
3. 222
4. 536

The output should contain no embedded spaces.

0					
1	*i*	1·	*x*	1··	*c*
2	*ii*	2·	*xx*	2··	*cc*
3	*iii*	3·	*xxx*	3··	*ccc*
4	*iv*	4·	*xl*	4··	*cd*
5	*v*	5·	*l*	5··	*d*
6	*vi*	6·	*lx*	6··	*dc*
7	*vii*	7·	*lxx*	7··	*dcc*
8	*viii*	8·	*lxxx*	8··	*dccc*
9	*ix*	9·	*xc*	9··	*cm*

Hint: Use a CHARACTER string (or CHARACTER strings) to store the number before output. The 'longest' number is 888, *dccclxxxviii* (12 characters).

13 Intrinsic Procedures

Some tasks in a language are performed frequently, Fortran 90 has efficient implementations of such common tasks built-in to the language, these procedures are called *intrinsic* procedures. Fortran 90 has 113 intrinsic procedures in a number of different classes,

- ☐ elemental such as:
  - ◇ mathematical, such as, trigonometric and logarithms, for example, SIN or LOG.
  - ◇ numeric, for example, SUM or CEILING.
  - ◇ character, for example, INDEX and TRIM.
  - ◇ bit manipulation, for example, IAND and IOR. (There is no BIT data type but intrinsics exist for manipulating integers as if they were bit variables.)

  Elemental procedures apply to scalar objects as well as arrays — when an array argument is supplied the same function is applied to each element of the array at (conceptually) the same time.
- ☐ inquiry, for example, ALLOCATED and SIZE; These report on the status of a program. We can inquire about:

  - ◇ the status of dynamic objects.
  - ◇ array bounds, shape and size.
  - ◇ kind parameters of an object and available kind representations, (useful for portable code).
  - ◇ the numerical model; used to represent types and kinds.
  - ◇ argument presence (for use with OPTIONAL dummy arguments).

- □ transformational, for example, REAL and TRANSPOSE. The functionality includes:
  - ◇ repeat (for characters — repeats strings).
  - ◇ mathematical reduction procedures, i.e., given an array return an object of less rank.
  - ◇ array manipulation — shift operations, RESHAPE, PACK.
  - ◇ type coercion, TRANSFER copies bit-for-bit to an object of a different type. (Stops people doing dirty tricks like changing the type of an object across a procedure boundary which was a popular FORTRAN 77 'trick'.)
  - ◇ PRODUCT and DOT_PRODUCT (arrays).
- □ miscellaneous (non-elemental SUBROUTINEs) including timing routines, for example, SYSTEM_-CLOCK and DATE_AND_TIME.

The procedures vary in what arguments are permitted. Some procedures can be applied to scalars and arrays, some to only scalars and some to only arrays. All intrinsics which take REAL valued arguments also accept DOUBLE PRECISION arguments.

13.1 Type Conversion Functions

In Fortran 90 it is easy to explicitly transform the type of a constant or variable by using the in-built intrinsic functions.

- □ REAL(i) converts the integer i to the corresponding real approximation, the argument to REAL can be INTEGER, DOUBLE PRECISION or COMPLEX.
- □ INT(x) converts real x to the integer equivalent following the truncation rules given before. The argument to INT can be REAL, DOUBLE PRECISION or COMPLEX.
- □ Other functions may form integers from non-integer values:
  - ◇ CEILING(x) — smallest integer greater or equal to x,
  - ◇ FLOOR(x) — largest integer less or equal to x,
  - ◇ NINT(x) — nearest integer to x.
- □ DBLE(a) converts a to DOUBLE PRECISION, the argument to DBLE can be INTEGER, REAL or COMPLEX.
- □ CMPLX(x) or CMPLX(x,y) — converts x to a complex value, $\mathbf{x} + i\mathbf{y}$.
- □ IACHAR(c) returns the position of the CHARACTER variable c in the ASCII collating sequence, the argument must be a single CHARACTER.
- □ ACHAR(i) returns the i^{th} character in the ASCII collating sequence (see 33), the argument ACHAR must be a single INTEGER.

For example,

```
PRINT*, REAL(1), INT(1.7), INT(-0.9999)
PRINT*, IACHAR('C'), ACHAR(67)
```

would give

```
1.000000 1 0
67 C
```

13.2 Mathematical Intrinsic Functions

Summary,

ACOS(x)	arccosine
ASIN(x)	arcsine
ATAN(x)	arctangent
ATAN2(y,x)	arctangent of complex number (x, y)
COS(x)	cosine where x is in radians
COSH(x)	hyperbolic cosine where x is in radians
EXP(x)	e raised to the power x
LOG(x)	natural logarithm of x
LOG10(x)	logarithm base 10 of x
SIN(x)	sine where x is in radians
SINH(x)	hyperbolic sine where x is in radians
SQRT(x)	the square root of x
TAN(x)	tangent where x is in radians
TANH(x)	tangent where x is in radians

- ASIN, ACOS — arcsin and arccos.

  The argument to each must be real and $\leq |1|$, for example, ASIN(0.84147098) has value 1.0 (radians).

- ATAN — arctan.

  The argument must be real valued, for example, ATAN(1.0) is $\frac{\pi}{4}$, ATAN(1.5574077) has value 1.0.

- ATAN2 — arctan; the principle value of the nonzero complex number (X,Y), for example, ATAN2(1.5574077,1.0) has value 1.0.

  The two arguments (Y, X) (note order) must be real valued, if Y is zero then X cannot be. These numbers represent the complex value (X,Y).

- TAN, COS, SIN — tangent, cosine and sine.

  Their arguments must be real or complex and are **measured in radians**, for example, COS(1.0) is 0.5403.

- TANH, COSH, SINH — hyperbolic trigonometric functions.

  The actual arguments must be REAL valued, for example, COSH(1.0) is 1.54308.

- ☐ EXP, LOG, LOG10, SQRT — e^x, natural logarithm, logarithm base 10 and square root.

  The arguments must must be real or complex (with certain constraints), for example, EXP(1.0) is 2.7182.

  Note that SQRT(9) is an invalid expression because the argument to SQRT cannot be INTEGER.

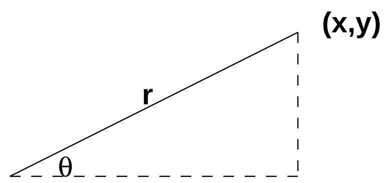
All angles are expressed in radians.

Question 10: Point on a circle

Write a program to read in a vector defined by a length, $\mathbf{r}$ and an angle, θ, in degrees which prints out the corresponding $(\mathbf{x}, \mathbf{y})$ co-ordinates. Recall that arguments to trigonometric functions are in radians.

Demonstrate correctness by giving the $(\mathbf{x}, \mathbf{y})$ co-ordinates for the following vectors

1. $\mathbf{r} = 12$, $\theta = 77°$
2. $\mathbf{r} = 1000$, $\theta = 0°$
3. $\mathbf{r} = 1000$, $\theta = 90°$
4. $\mathbf{r} = 20$, $\theta = 100°$
5. $\mathbf{r} = 12$, $\theta = 437°$



Hint: remember that

$$\sin\theta = \frac{y}{r}$$

and

$$\cos\theta = \frac{x}{r}$$

13.3 Numeric Intrinsic Functions

Summary,

ABS(a)	absolute value
AINT(a)	truncates a to whole REAL number
ANINT(a)	nearest whole REAL number
CEILING(a)	smallest INTEGER greater than or equal to REAL number
CMPLX(x,y)	convert to COMPLEX
DBLE(x)	convert to DOUBLE PRECISION
DIM(x,y)	positive difference
FLOOR(a)	biggest INTEGER less than or equal to real number
INT(a)	truncates a into an INTEGER
MAX(a1,a2,a3,...)	the maximum value of the arguments
MIN(a1,a2,a3,...)	the minimum value of the arguments
MOD(a,p)	remainder function
MODULO(a,p)	modulo function
NINT(x)	nearest INTEGER to a REAL number
REAL(a)	converts to the equivalent REAL value
SIGN(a,b)	transfer of sign — ABS(a)*(b/ABS(b))

As all are elemental they can accept array arguments, the result is the same shape as the argument(s) and is the same as if the intrinsic had been called separately for each array element of the argument.

- ☐ ABS — absolute value.

  The argument can be INTEGER, REAL or COMPLEX, the result is of the same type as the argument except for complex where the result is real valued, for example, ABS(-1) is 1, ABS(-.2) is 0.2 and ABS(CMPLX(-3.0,4.0)) is 5.0.

- ☐ AINT — truncates to a whole number.

  The argument and result are real valued, for example, AINT(1.7) is 1.0 and AINT(-1.7) is -1.0.

- ☐ ANINT — nearest whole number.

  The argument and result are real valued, for example, AINT(1.7) is 2.0 and AINT(-1.7) is -2.0.

- ☐ CEILING, FLOOR — smallest INTEGER greater than (or equal to), or biggest INTEGER less than (or equal to) the argument.

  The argument must be REAL, for example, CEILING(1.7) is 2 CEILING(-1.7) is 1.

- ☐ CMPLX — converts to complex value.

  The argument must be two real numbers, for example, CMPLX(3.6,4.5) is a complex number.

- ☐ DBLE — coerce to DOUBLE PRECISION data type.

  Arguments must be REAL, INTEGER or COMPLEX. The result is the actual argument converted to a DOUBLE PRECISION number.

- ☐ DIM — positive difference.

  Arguments must be REAL or INTEGER. If X bigger than Y then DIM(X,Y) = X-Y, if $Y > X$ and result of X-Y is negative then DIM(X,Y) is zero, for example, DIM(2,7) is 0 and DIM(7,2) is 5.

- ☐ INT truncates to an INTEGER (as in integer division)

  Actual argument must be numeric, for example INT(8.6) is 8 and INTCMPLX(2.6,4.0) is 2.

- ☐ MAX and MIN — maximum and minimum functions.

  These must have at least two arguments which must be INTEGER or REAL. MAX(1.0,2.0) is 2.0.

- `MOD` — remainder function.

  Arguments must be `REAL` or `INTEGER`. `MOD(a,p)` is the remainder when evaluating `a/p`, for example, `MOD(9,5)` is 4, `MOD(-9.0,5.0)` is `-4.0`.

- `MODULO` — modulo function.

  Arguments must be `REAL` or `INTEGER`. `MODULO(a,b)` is $a \bmod b$, for example, `MOD(9,5)` is 4, `MOD(-9.0,5.0)` is `1.0`.

- `REAL` — coverts to `REAL` value.

  For example, `REAL(5)` is `5.0`

- `SIGN` — transfers the sign of the second argument to the first.

  The arguments are real or integer and the result is of the same type and is equal to `ABS(a)*(b/ABS(b))`, for example, `SIGN(6,-7)` is -6, `SIGN(-6,7)` is 6.

Question 11: Quadratic equation solver

Write a program to read in values of a, b and c and calculate the real roots of the corresponding quadratic equation:

$$\mathbf{y} = a^2\mathbf{x} + b\mathbf{x} + c$$

Point out if the equation only has one or no real roots.

The program should repeatedly expect input; $a = 0$, $b = 0$ and $c = 0$ should be used to terminate.

Hint 1: recall that the solution of a general quadratic equation equation is:

$$\mathbf{x} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Hint 2: The program has a single root if

$$b^2 - 4ac = 0$$

two real roots if

$$b^2 - 4ac > 0$$

and imaginary roots if

$$b^2 - 4ac < 0$$

13.4 Character Intrinsic Functions

Summary,

`ACHAR(i)`	i^{th} character in ASCII collating sequence
`ADJUSTL(str)`	adjust left
`ADJUSTR(str)`	adjust right
`CHAR(i)`	i^{th} character in processor collating sequence
`IACHAR(ch)`	position of character in ASCII collating sequence
`ICHAR(ch)`	position of character in processor collating sequence
`INDEX(str,substr)`	starting position of substring
`LEN(str)`	Length of string
`LEN_TRIM(str)`	Length of string without trailing blanks
`LGE(str1,str2)`	lexically `.GE.`
`LGT(str1,str2)`	lexically `.GT.`
`LLE(str1,str2)`	lexically `.LE.`
`LLT(str1,str2)`	lexically `.LT.`
`REPEAT(str,i)`	repeat i times
`SCAN(str,set)`	scan a string for characters in a set
`TRIM(str)`	remove trailing blanks
`VERIFY(str,set)`	verify the set of characters in a string

- ☐ `ACHAR`— i^{th} character in ASCII collating sequence.

  The argument must be between 0 and 127; this function is the inverse of `IACHAR`, for example `ACHAR(100)` is `'d'`. Compare to `CHAR`.

- ☐ `ADJUSTL` — adjust a string left.

  The argument must be a string and the result is the same string with leading blanks removed and inserted as trailing blanks.

- ☐ `ADJUSTR` — adjust a string right.

  The argument must be a string and the result is the same string with trailing blanks removed and inserted as leading blanks.

- ☐ `CHAR` — i^{th} character in the compilers collating sequence

  Takes a single character as an argument. The result is similar to `ACHAR` but uses the compilers collating sequence (this will often be the same as `ACHAR`.)

- ☐ `IACHAR` — position of a character in ASCII collating sequence.

  Takes a single character as an argument which must be an ASCII character, and returns its position of a character in ASCII collating sequence, for example, `IACHAR('d')` is `100`.

- ☐ `ICHAR` — position of a character in the compilers collating sequence.

  Takes a single character as an argument (which must be valid) and returns its position of a character in the compilers collating sequence, for example, `IACHAR('d')` is `100`. (The result is often the same as `IACHAR`.)

- ☐ `INDEX` — starting position of a substring in a string.

  Takes two arguments, both must be of type `CHARACTER` and of the same kind, the result is the *first* occurrence of `substr` in `str`, for example, `INDEX('jibberish','eris')` is 5.

- LEN, LEN_TRIM— length of string

  Both take one string argument the first function returns the length of the string including the trailing blanks and the second discounts the blanks, for example, LEN("Whoosh!! ") is 10, LEN_TRIM("Whoosh!! ") is 8.

- LGE, .., LLT — lexical positional operators.

  These functions accept two strings of the same kind, the result is comparable to that of relational operators in the sense that a LOGICAL value is returned governed by the lexical position of the string in ASCII order. This means there is a difference between the case of a letter, for example, LGT('Tin','Tin') returns .FALSE., LGE('Tin','Tin') and LGE('tin','Tin') return .TRUE..

- REPEAT — concatenate string i times.

  The first argument is a string and the second the number of times it is to be repeated, for example REPEAT('Boutrous ',2) is 'Boutrous Boutrous '.

- TRIM — remove trailing blanks.

- VERIFY — verify that a set of characters contains all the letters in a string.

  The two arguments, set and string, are characters and of the same kind. Given a set of characters (stored in a string) the result is the first position in the string which is a character that is NOT in the set, for example, VERIFY('ABBA','A') is 2 and VERIFY('ABBA','BA') is 0.

Question 12: Concatenate Names

Write a program which accepts two names (Christian name and Family name, a maximum of 10 characters each) and outputs a single string containing the full name separated by one space with the first letter of each name in upper case and the rest of the name in lower case. You may assume that all inputs are valid names.

Hint a comes before A in the ASCII collating sequence.

(This sequence is given in the notes.)

14 Simple Input / Output

14.1 PRINT Statement

This is the simplest form of directing unformatted data to the standard output channel, for example,

```
PROGRAM Outie
 CHARACTER(LEN=*), PARAMETER :: long_name = &
           "Llanfairphwyll...gogogoch"
 REAL :: x, y, z
 LOGICAL :: lacigol
 x = 1; y = 2; z = 3
 lacigol = (y .eq. x)
 PRINT*, long_name
 PRINT*, "Spock says ""illogical&
```

```
          &Captain"" "
    PRINT*, "X = ",x," Y = ",y," Z = ",z
    PRINT*, "Logical val: ",lacigol
   END PROGRAM Outie
```

produces on the screen,

```
    Llanfairphwyll...gogogoch
    Spock says "illogical Captain"
    X =    1.000  Y =    2.000  Z =    3.000
    Logical val:  F
```

As can be seen from the above example, the PRINT statement takes a comma separated list of things to print, the list can be any printable object including user-defined types (as long as they don't contain pointers). The * indicates the output is in free (default) format. Fortran 90 supports a great wealth of output (and input) formatting which is not all described here!

There are a couple of points to raise,

- LOGICAL variables can be printed,

  ```
  lacigol = (y .eq. x)
  ```

  generates an F signifying .FALSE..

- Strings can be split across lines,

  ```
  PRINT*, "Spock says ""illogical&
          &Captain"" "
  ```

  If a CHARACTER string crosses a line indentation can still be used if an & is appended to the end of the first line and the position from where the string is wanted to begin on the second - see the Spock line in the example; the &s act like a single space.

- The double " in the string, the first one *escapes* the second. Strings may be delimited by the double or single quote symbols, " and ', but these may not be mixed in order to delimit a string. The following would produce the same output as the statement in the program,

  ```
  PRINT*, 'Spock says "illogical&
          &Captain" '
  ```

  In this case the " delimiter does not have to be escaped.

- Notice how the output has many more spaces than the PRINT statement indicates. This is because output is unformatted. The default formatting is likely to be different between compilers.

- Each PRINT statement begins a new line, non-advancing I/O is available but we have to specify it in a FORMAT statement.

14.2 READ Statement

This is the simplest form of reading unformatted data from the standard input channel, for example, if the type declarations are the same as for the PRINT example,

```
READ*, long_name
READ*, x, y, z
READ*, lacigol
```

would read the following input from the keyboard

```
Llanphairphwyll...gogogoch
0.4 5. 1.0e12
T
```

Note,

- each READ statement reads from a newline;
- the READ statement can transfer any object of intrinsic type from the standard input;

The * format specifier in the READ statement is comparable to the functionality of PRINT, in other words, unformatted data is read. (Actually this is not strictly true formatted data can be read but the format cannot be specified!) As long as each entity to be read in is blank separated, then the READ statement simply works through its 'argument' list. Each READ statement begins a new line so if there are less arguments to the read statement than there are entries on a line the extra items will be ignored.

Module 5: Arrays

15 Arrays

Arrays (or matrices) hold a collection of different values at the same time. Individual elements are accessed by **subscripting** the array.

A 15 element array can be visualised as:

Figure 6: A One Dimensional (1D) Array

And a 5 $\times$ 3 array as:

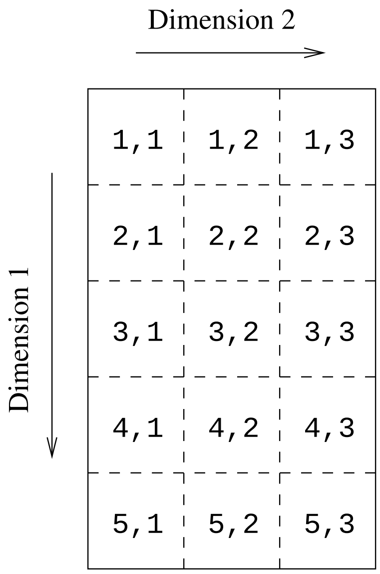


Figure 7: A Two Dimensional (2D) Array

Every array has a type (`REAL`, `INTEGER`, etc) so each element holds a value of that type.

15.1 Array Terminology

Examples of declarations:

```
REAL, DIMENSION(15)      :: X
REAL, DIMENSION(1:5,1:3) :: Y, Z
```

The above are *explicit-shape* arrays.

If the lower bound is not explicitly stated it is taken to be 1.

Terminology:

- *rank* — the number of dimensions up to and including 7 dimensions. `X` has rank 1, `Y` and `Z` have rank 2.
- *bounds* — upper and lower limits of indices, an unspecified bound is 1. `X` has lower bound 1 and upper bound 15, `Y` and `Z` have lower bounds of 1 and 1 with upper bounds 5 and 3.
- *extent* — number of elements in dimension (which can be zero). `X` has extent 15, `Y` and `Z` have extents 5 and 3.
- *size* — either the total number of elements or, if particular dimension is specified, the number of elements in that dimension. All arrays have size 15.
- *shape* — rank and extents. `X` has shape `(/15/)`, `Y` and `Z` have shape `(/5,3/)`.
- *conformable* — two arrays are conformable if they have the same shape — for operations between two arrays the shapes (of the sections) must (generally) conform (just like in mathematics). `Y` and `Z` have the same shape so they conform.
- there is no *storage association* for Fortran 90 arrays.

Explicit-shape arrays can have symbolic bounds so long as they are initialisation expressions — evaluatable at compile time.

Question 13: Rank, Extents etc.

Give the rank, bounds, size and shape of the arrays defined as follows:

```
REAL, DIMENSION(1:10)  :: ONE
REAL, DIMENSION(2,0:2) :: TWO
INTEGER, DIMENSION(-1:1,3,2) :: THREE
REAL, DIMENSION(0:1,3) :: FOUR
```

15.2 Declarations

As long as the value of `lda` is known the following are valid:

```
REAL, DIMENSION(100)       :: R
REAL, DIMENSION(1:10,1:10) :: S
REAL                       :: T(10,10)
REAL, DIMENSION(-10:-1)    :: X
INTEGER, PARAMETER :: lda = 5
REAL, DIMENSION(0:lda-1)   :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z
```

The above example demonstrates:

- bounds can begin and end anywhere, (the array X),
- default lower bound is 1,
- there is a shorthand form of declaration, see T,
- arrays can be zero-sized. If lda were set to be zero,

  ```
  INTEGER, PARAMETER :: lda = 0
  ```

  then the array Y would be zero sized.

  Zero-sized arrays are useful when programming degenerate cases especially when invoking procedures (in this case we would expect lda to be a dummy argument which determines the size of some local (or automatic) array); no extra code has to be added to test for zero extents in any of the dimensions — statements including references to zero sized arrays are simply ignored.

Question 14: Hotel Array

Declare an array of rank 3 which might be suitable for representing a hotel with 8 floors and 16 rooms on each floor and two beds in each room. How would the second bed in the 5th room on floor 7 be referenced?

Consider the following declarations,

```
REAL, DIMENSION(15)       :: A
REAL, DIMENSION(-4:0,0:2) :: B
REAL, DIMENSION(5,3)      :: C
REAL, DIMENSION(0:4,0:2)  :: D
```

Individual array elements are denoted by *subscripting* the array name by an INTEGER, for example, A(7) 7^{th} element of A, or C(3,2), 3 elements down, 2 across.

The arrays can be visualised as below:

The first dimension runs up and down the page and the second dimensions runs across the page.

Question 15: Array References

Given,

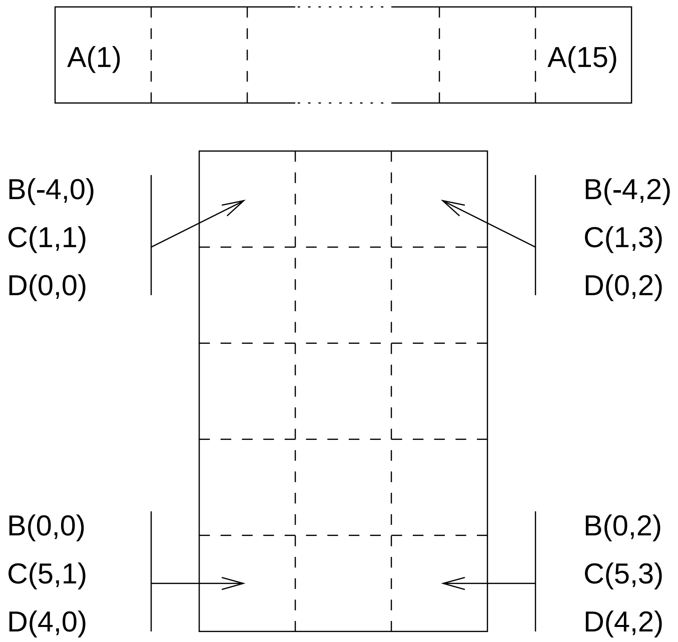


Figure 8: Visualisation of Arrays

```
INTEGER :: i = 3, j = 7
REAL, DIMENSION(1:20) :: A
```

which of the following are valid array references for the array:

- ☐ `A(12)`
- ☐ `A(21)`
- ☐ `A(I)`
- ☐ `A(3.0)`
- ☐ `A(I*J)`
- ☐ `A(1+INT(4.0*ATAN(1.0)))`

[Hint: `4.0*ATAN(1.0)` is π]

15.3 Array Conformance

If an object or sub-object is used directly in an expression then it must conform with all other objects in that expression. (Note that a scalar conforms to any array with the same value for every element.) for two array references to conform both objects must be the same shape.

Using the declarations from before:

```
C = D                   ! is valid
A = B                   ! is not valid
```

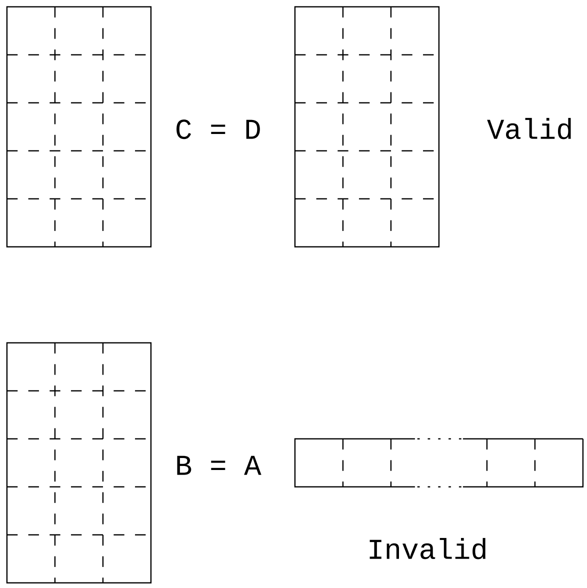


Figure 9: Visualisation of conforming Arrays

Visualisation,

A and B have the same size (15 elements) but have different shapes so cannot be directly equated. To force conformance the array must be used as an argument to a transformational intrinsic to change its shape, for example,

```
B = RESHAPE(A,(/5,3/))  ! is, see later
A = PACK(B,.TRUE.)      ! is, see later
A = RESHAPE(B,(/10,8/)) ! is, see later
B = PACK(A,.TRUE.)      ! is, see later
```

Arrays can have their shapes changed by using transformational intrinsics including, `MERGE`, `PACK`, `SPREAD`, `UNPACK` and `RESHAPE`.

Two arrays of different types conform and if used in the same expression will have the relevant type coercion performed just like scalars.

Question 16: Conformance

Given

```
REAL, DIMENSION(1:10)  :: ONE
REAL, DIMENSION(2,0:2) :: TWO
INTEGER, DIMENSION(-1:1,3,2) :: THREE
REAL, DIMENSION(0:1,3) :: FOUR
```

Which two of the arrays are conformable?

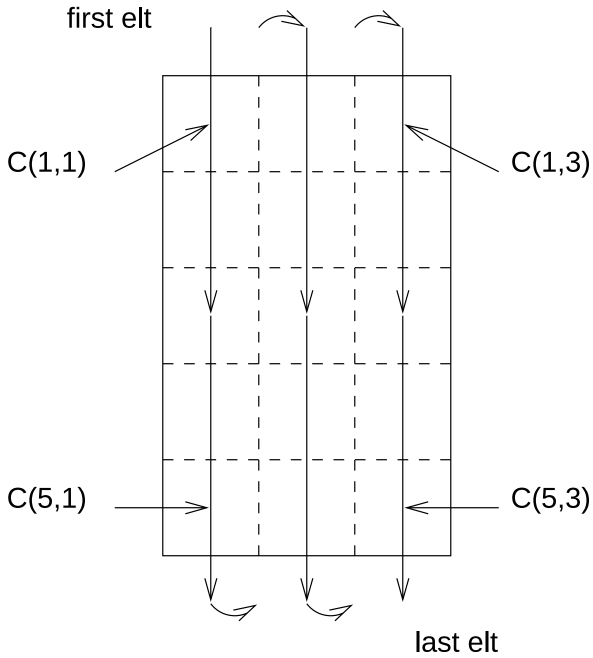


Figure 10: Visualisation Of Array Element Ordering

15.4 Array Element Ordering

Fortran 90 does not have any storage association meaning that, unlike FORTRAN 77, the standard does not specify how arrays are to be organised in memory. This makes passing arrays to a procedure written in a different language *very* difficult indeed.

The lack of implicit storage association makes it easier to write portable programs and allows compiler writers more freedom to implement local optimisations. For example, in distributed memory computers an array may be stored over 100 processors with each processor owning only a small section of the whole array — the standard will allow this.

There are certain situations where an ordering is needed, for example, during input or output and in these circumstances Fortran 90 does define ordering which can be used in such contexts. It is defined in the same manner as the FORTRAN 77 storage association but it does **not** imply anything about how array elements are stored.

The array element ordering is again of column major form:

```
C(1,1),C(2,1),..,C(5,1),C(1,2),C(2,2),..,C(5,3)
```

This ordering is used in array constructors, I/O statements, certain intrinsics (`TRANSFER`, `RESHAPE`, `PACK`, `UNPACK` and `MERGE`) and any other contexts where an ordering is needed.

Question 17: Array Element Ordering

Given

```
REAL, DIMENSION(1:10)  :: ONE
```

```
REAL, DIMENSION(2,0:2) :: TWO
INTEGER, DIMENSION(-1:1,3,2) :: THREE
REAL, DIMENSION(0:1,3) :: FOUR
```

Write down the array element order of each array.

15.5 Array Syntax

Using the earlier declarations, references can be made to:

- whole arrays (conformable)
  - `A = 0.0`
    This statement will set whole array `A` to zero. Each assignment is performed conceptually at the same time. Scalars always conform with arrays.
  - `B = C + D`
    This adds the corresponding elements of `C` and `D` and then assigns each element if the result to the corresponding element of `B`.

    For this to be legal Fortran 90 both arrays in the RHS expression must conform (`B` and `C` must be same shape and size). The assignment could have been written `B(:) = C(:) + D(:)` demonstrating how a whole array can be referenced by subscripting it with a colon. (This is shorthand for `lower_bound:upper_bound` and is exactly equivalent to using only its name with no subscripts or parentheses.)
- elements
  - `A(1) = 0.0`
    This statement sets one element, the first element of `A` (`A(1)`), to zero,
  - `B(0,0) = A(3) + C(5,1)`
    Sets element `B(0,0)` to the sum of two elements.

  If present, array subscripts must be integer valued expressions.

  A particular element of an array is accessed by subscripting the array name with an integer which is within the bounds of the declared extent. Subscripting directly with a `REAL`, `COMPLEX`, `CHARACTER`, `DOUBLE PRECISION` or `LOGICAL` is an error. This, and indeed the previous example, demonstrates how scalars (literals and variables) conform to arrays; scalars can be used in many contexts in place of an array.
- array sections
  - `A(2:4) = 0.0`
    This assignment sets three elements of `A` (`A(2)`, `A(3)` and `A(4)`) to zero.
  - `B(-1:0,1:2)=C(1:2,2:3)+1`
    Adds one to the subsection of `C` and assigns to the subsection of `B`.

  The above examples demonstrate how parts (or subsections) of arrays can be referenced. An array section can be specified using the colon range notation first encountered in the `SELECT CASE` construct. In addition, the sequence of elements can also have a stride (like `DO` loops) meaning that the section specification is denoted by a *subscript-triplet* which is a linear function.

Care must be taken when referring to different sections of the same array on both sides of an assignment statement, for example,

```
DO i = 2,15
 A(i) = A(i) + A(i-1)
END DO
```

is not the same as

```
A(2:15) = A(2:15) + A(1:14)
```

in the first case, a general element i of `A` has the value,

```
A(i) = A(i) + A(i-1) + ... + A(2) + A(1)
```

but in the vectorised statement it has the value

```
A(i) = A(i) + A(i-1)
```

The correct vector equivalent to the original `DO`-loop can be achieved by using the `SUM` intrinsic, `A(2:15) = (/ (SUM(1:i), i=2,15) /)`,

In summary both scalars and arrays can be thought of as objects. (More or less) the same operations can be performed on each with the array operations being performed in parallel. It is not possible to have a scalar on the LHS of an assignment and a non scalar array reference on the RHS unless that section is an argument to a reduction function.

15.6 Whole Array Expressions

A whole (or section of an) array can be treated like a single variable in that all intrinsic operators which apply to intrinsic types have their meaning extended to apply to conformable arrays, for example,

```
B = C * D - B**2
```

as long as `B`, `C` and `D` conform then the above assignment is valid. (Recall that the RHS of the `**` operator must be scalar.) Note that in the above example, `C*D` is **not** matrix multiplication, `MATMUL(C,D)` should be used if this is the desired operation.

The above assignment is equivalent to:

```
!PARALLEL
    B(-4,0) = C(1,1)*D(0,0)-B(-4,0)**2 ! in ||
    B(-3,0) = C(2,1)*D(1,0)-B(-3,0)**2 ! in ||
     ...
    B(-4,1) = C(1,2)*D(0,1)-B(-4,1)**2 ! in ||
     ...
    B(0,2)  = C(5,3)*D(4,2)-B(0,2)**2  ! in ||
!END PARALLEL
```

With array assignment there is no implied order of the individual assignments, they are performed, conceptually, in parallel.

In addition to the above operators, the subset of intrinsic functions termed elemental can also be applied, for example,

```
    B = SIN(C)+COS(D)
```

The functions are also applied element by element, thus the above is equivalent to the parallel execution of,

```
!PARALLEL
    B(-4,0) = SIN(C(1,1))+COS(D(0,0))
      ...
    B(0,2)  = SIN(C(5,3))+COS(D(4,2))
!END PARALLEL
```

Many of Fortran 90's intrinsics are elemental including all numeric, mathematical, bit, character and logical intrinsics.

Again it must be stressed that conceptually there is no order implied in the array statement form — each individual assignment can be thought of as being executed in parallel between corresponding elements of the arrays — this is different from the DO-loop.

15.7 Visualising Array Sections

Consider the declaration

```
    REAL, DIMENSION(1:6,1:8) :: P
```

The sections:

- P(1:3,1:4) is a 3×4 section; the missing stride implies a value of 1,
- P(2:6:2,1:7:3), which could be written: P(2::2,:7:3)) is a 3×3 section.

  (A missing upper bound (in the first dimension) means assume the upper bound as declared, (6), A missing lower bound is the lower bound as declared, (1).)
- P(2:5,7) is a 1D array with 4 elements; P(2:5,7:7) is a 4×1 2D array,
- P(1:6:2,1:8:2) is a 3×4 section. This could also be written as P(::2,::2), here both upper and lower bounds are missing so the values are taken to be the bounds as declared,

Conformance:

- P(1:3,1:4) = P(1:6:2,1:8:2) is a valid assignment; both LHS and RHS are 3×4 sections.
- P(1:3,1:4) = 1.0 is a valid assignment; a scalar on the RHS conforms to any array on the LHS of an assignment.
- P(2:6:2,1:7:3) = P(1:3,1:4) is not a valid assignment; an attempt is made to equate a 3×3 section with a 3×4 section, the array sections do not conform.
- P(2:6:2,1:7:3) = P(2:5,7) is not a valid assignment; an attempt is made to equate a 3×3 section with a 4 element 1D array.

It is important to recognise the difference between an n element 1D array and a $1 \times n$ 2D array:

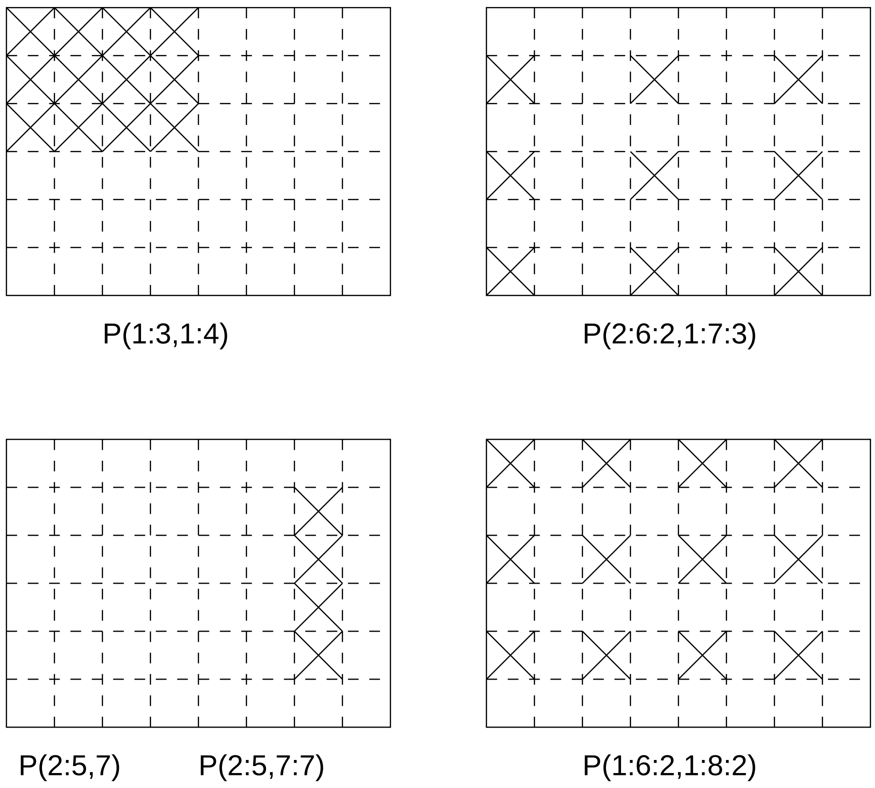


Figure 11: Visualisation of Array Sections

- ☐ P(2:5,7) is a 1D section — the scalar in the second dimension 'collapses' the dimension.
- ☐ P(2:5,7:7) is a 2D section — the second dimension is specified with a section (a range) not a scalar so the resultant sub-object is still two dimensional.

15.8 Array Sections

The general form of a subscript-triplet specifier is::

[< *bound1* >]:[< *bound2* >][:< *stride* >]

The section starts at < *bound1* > and ends at or before < *bound2* >. < *stride* > is the increment by which the locations are selected. < *bound1* >, < *bound2* > and < *stride* > must all be scalar integer expressions. Thus

```
A(:)         ! the whole array
A(3:9)       ! A(m) to A(n) in steps of 1
A(3:9:1)     ! as above
A(m:n)       ! A(m) to A(n)
A(m:n:k)     ! A(m) to A(n) in steps of k
A(8:3:-1)    ! A(8) to A(3) in steps of -1
A(8:3)       ! A(8) to A(3) step 1 => Zero size
A(m:)        ! from A(m) to default UPB
A(:n)        ! from default LWB to A(n)
A(::2)       ! from default LWB to UPB step 2
A(m:m)       ! 1 element section
A(m)         ! scalar element - not a section
```

are all valid.

If the upper bound (< *bound2* >) is not a combination of the lower bound plus multiples of the stride then the actual upper bound is different from that stated; this is the same principle that is applied to DO-loops.

Another similarity with the DO-loops is that when the stride is not specified it is assumed to have a value of 1. In the above example, this means that A(3:8) is the same as A(3:8:1) but A(8:3) is a zero sized section and A(8:3:-1) is a section that runs backwards. Zero strides are **not** allowed and, in any case, are pretty meaningless!

Other bound specifiers can be absent too, if < *bound1* > or < *bound2* > is absent then the lower or upper bound of the dimension (as declared) is implied, if both are missing then the whole dimension is assumed.

Let us examine the above sections in detail,

- ☐ A(:)

  This runs from the declared lower bound to the declared upper bound so refers to the whole array.

- ☐ A(3:9)

  Defines the 7 element section running from A(3) to A(9). The stride is missing therefore is assumed to be 1.

- A(3:9:1)

  Exactly the same section as above.

- A(m:n)

  From element A(m) to A(n).

- A(m:n:k)

  The section runs from m to n in strides of k.

- A(8:3:-1)

  This section runs from 8 to 3 in steps of -1.

- A(8:3)

  This section runs from 8 to 3 in steps of 1, i.e., this is a zero sized section.

- A(m:)

  This section runs from M to the declared upper bound in steps of 1.

- A(:n)

  This section runs from the declared lower bound to n in steps of 1.

- A(::2)

  This section runs from the declared lower bound to the upper bound in strides of 2.

- A(m:m)

  This is a one element array and is distinct from A(m) which is a scalar reference.

Question 18: Array Sections

Declare an array which would be suitable for representing draughts board. Write a program to set all the white squares to zero and the black squares to unity. (A draughts board is 8×8 with alternate black and white squares)

15.9 Printing Arrays

The conceptual ordering of array elements is useful for defining the order in which array elements are output. If A is a 2D array then:

```
PRINT*, A
```

would produce output in Array Element Order:

```
A(1,1), A(2,1), A(3,1), ..., A(1,2), A(2,2), ...
```

Sections of arrays can also be output, for example,

```
PRINT*, A(::2,::2)
```

would produce:

```
A(1,1), A(3,1), A(5,1), ..., A(1,3), A(3,3), A(5,3), ...
```

An array of more than one dimension is not formatted neatly, if it is desired that the array be printed out row-by-row (or indeed column by column) then this must be programmed explicitly.

This order could be changed by using intrinsic functions such as RESHAPE, TRANSPOSE or CSHIFT.

15.10 Input of Arrays

Elements of an array can be read in and assigned to the array in array element order, for example,

```
READ*, A
```

would read data from the standard input and assign to the elements of A. The input data may be punctuated by any number of carriage returns which are simply ignored.

Sections of arrays can also be input, for example,

```
READ*, A(::2,::2)
```

is perfectly valid and will assign to the indicated subsection of A.

15.10.1 Array I/O Example

Consider the matrix A:

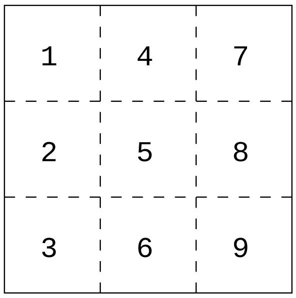


Figure 12: Visualisation of the array A

The PRINT statements in the following program

```
PROGRAM Owt
 IMPLICIT NONE
  INTEGER, DIMENSION(3,3) :: A = RESHAPE((/1,2,3,4,5,6,7,8,9/))
```

```
      PRINT*, 'Array element   =',a(3,2)
      PRINT*, 'Array section   =',a(:,1)
      PRINT*, 'Sub-array       =',a(:2,:2)
      PRINT*, 'Whole Array     =',a
      PRINT*, 'Array Transp''d =',TRANSPOSE(a)
   END PROGARM Owt
```

produce on the screen,

```
    Array element    = 6
    Array section    = 1 2 3
    Sub-array        = 1 2 4 5
    Whole Array      = 1 2 3 4 5 6 7 8 9
    Array Transposed = 1 4 7 2 5 8 3 6 9
```

15.11 Array Inquiry Intrinsics

These intrinsics allow the user to quiz arrays about their attributes and status and are most often applied to dummy arguments in procedures. Consider the declaration:

```
    REAL, DIMENSION(-10:10,23,14:28) :: A
```

the following inquiry intrinsics are available,

- LBOUND(SOURCE[,DIM])

  Returns a one dimensional array containing the lower bounds of an array or, if a dimension is specified, a scalar containing the lower bound in that dimension. For example,

  - LBOUND(A) is (/-10,1,14/) (array);
  - LBOUND(A,1) is -10 (scalar).

- UBOUND(SOURCE[,DIM])

  Returns a one dimensional array containing the upper bounds of an array or, if a dimension is specified, a scalar containing the upper bound in that dimension. For example,

  - UBOUND(A) is (/10,23,28/)
  - UBOUND(A,1) is 10.

- SHAPE(SOURCE)

  Returns a one dimensional array containing the shape of an object. For example,

  - SHAPE(A) is (/21,23,15/) (array);
  - SHAPE((/4/)) is (/1/) (array).

- SIZE(SOURCE[,DIM])

  Returns a scalar containing the total number of array elements either in the whole array or in an optionally specified dimension. For example,

  - SIZE(A,1) is 21.

◇ SIZE(A) is 7245.

◇ SIZE(4) is an error as the argument must not be scalar.

□ ALLOCATED(SOURCE)

Returns a scalar LOGICAL result indicating whether an array is allocated or not. For example,

```
PROGRAM Main
 IMPLICIT NONE
  INTEGER, ALLOCATABLE, DIMENSION(:) :: Vec
   PRINT*, ALLOCATED(Vec)
   ALLOCATE(Vec(10))
   PRINT*, ALLOCATED(Vec)
   DEALLOCATE(Vec)
   PRINT*, ALLOCATED(Vec)
END PROGRAM Main
```

will produce .FALSE., .TRUE. and .FALSE. in that order.

Question 19: Inquiry intrinsics etc.

Given,

```
INTEGER, DIMENSION(-1:1,3,2) :: A
```

Write a small program which contains intrinsic function calls to show:

1. the total number of elements in A,
2. the shape of A
3. the lower bound in dimension 2
4. the upper bound in dimension 3

15.12 Array Constructors

Array constructors are used to give arrays or sections of arrays specific values. An array constructor is a comma separated list of scalar expressions delimited by (/ and /). The results of the expressions are placed into the array in array element order with any type conversions being performed in the same manner as for regular assignment. The constructor must be of the correct length for the array, in other words, the section and the constructor must conform.

For example,

```
PROGRAM MAin
  IMPLICIT NONE
  INTEGER, DIMENSION(1:10)  :: ints
  CHARACTER(len=5), DIMENSION(1:3) :: colours
  REAL, DIMENSION(1:4)      :: heights
```

```
       heights = (/5.10, 5.6, 4.0, 3.6/)
       colours = (/'RED  ','GREEN','BLUE '/)
       ! note padding so strings are 5 chars
       ints    = (/ 100, (i, i=1,8), 100 /)
        ...
    END PROGRAM MAin
```

The array and its constructor must conform.

Notice that all strings in the constructor for `colours` are 5 characters long. This is because the string within the constructor *must* be the correct length for the variable.

`(i, i=1,8)` is an *implied*-`DO` *specifier* and may be used in constructors to specify a sequence of constructor values. There may be any number of separate implied `DO`s which may be mixed with other specification methods. In the above example the vector `ints` will contain the values `(/ 100, 1, 2, 3, 4, 5, 6, 7, 8, 100 /)`. Note the format of the implied `DO`: a `DO`-loop index specification surrounded by parentheses.

There is a restriction that only one dimensional constructors are permitted, for higher rank arrays the `RESHAPE` intrinsic must be used to modify the shape of the result of the RHS so that it conforms to the LHS:

```
    INTEGER, DIMENSION(1:3,1:4) :: board
    board = RESHAPE((/11,21,31,12,22,32,13,23,33,14,24,34/), (/3,4/))
```

The values are specified as a one dimensional constructor and then the shape is modified to be a 3×4 array which conforms with the declared shape of `board`.

Question 20: Array Constructor

Write an array constructor for the 5 element rank 1 array `BOXES` containing the values `1, 4, 6, 12, 23`.

15.13 The RESHAPE Intrinsic Function

`RESHAPE` is a general intrinsic function which delivers an array of a specified shape:

`RESHAPE(SOURCE,SHAPE[,PAD][,ORDER])`

Note,

- the `RESHAPE` intrinsic changes the shape of `SOURCE` to the specified `SHAPE`.
- `SOURCE` must be intrinsic typed array, it cannot be an array of user-defined types.
- `SHAPE` is a one dimensional array specifying the target shape. It is convenient to use an explicit array constructor for this field in order to make things clearer.
- `PAD` is a one dimensional array of values which is used to pad out the resulting array if there are not enough values in `SOURCE`. The `PAD` constructor is used repeatedly (in array element order) to provide enough elements for the result. `PAD` is optional.

- ORDER allows the dimensions to be permuted, in other words, allows the array element ordering to be modified, ORDER is optional.

For example, the following statement assigns SOURCE to A,

```
A = RESHAPE((/1,2,3,4/),(/2,2/))
```

The result of the RESHAPE is a 2×2 array (specified by the second argument (/2,2/)), the result is filled in array element order and looks like:

```
1  3
2  4
```

Visualisation,

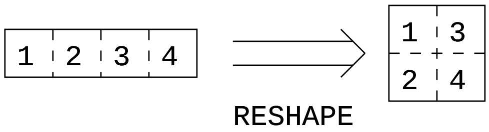


Figure 13: Visualisation of the Effect of the RESHAPE Intrinsic

Also consider

```
A = RESHAPE((/1,2,3,4/),(/2,2/),&
                 ORDER=(/2,1/))
```

This time the array is filled up in row major form, (the subscripts of dimension 2 vary the quickest,) this is specified by the ORDER=(/2,1/) specifier. The default ordering is, of course, ORDER=(/1,2/). The ORDER keyword is necessary because some optional arguments are missing. A looks like

```
1  2
3  4
```

Clearly the result of RESHAPE must conform to the array object on the LHS of the =, consider,

```
RESHAPE((/1,2,3,4,5,6/),(/2,4/),(/0/),(/2,1/))
```

this has the value

```
1  2  3  4
5  6  0  0
```

The source object has less elements than the LHS so the resulting array is padded with the extra values taken repeatedly from the third array argument, PAD, (/0/). Note how this reference does not use keyword arguments, it is directly equivalent to,

```
RESHAPE(SOURCE=(/1,2,3,4,5,6/),&
         SHAPE=(/2,4/),        &
           PAD=(/0/),          &
         ORDER=(/2,1/))
```

and

```
RESHAPE(SOURCE=(/1,2,3,4,5,6/),&
           PAD=(/0/),          &
         SHAPE=(/2,4/),        &
         ORDER=(/2,1/))
```

If one of the optional arguments is absent then keyword arguments should be used for the other optional argument to make it clear to the compiler (and the user) which is the missing argument. The keywords are the names of the dummy arguments.

15.14 Array Constructors in Initialisation Statements

Named array constants of any rank can be created using the RESHAPE function as long as all components can be evaluated at compile time (just like in initialisation expressions):

```
INTEGER, DIMENSION(3), PARAMETER :: Unit_vec = (/1,1,1/)
CHARACTER(LEN=*), DIMENSION(3), PARAMETER :: &
      lights = (/'RED  ','BLUE ','GREEN'/)
REAL, DIMENSION(3,3), PARAMETER :: &
      unit_matrix = RESHAPE((/1,0,0,0,1,0,0,0,1/), (/3,3/))
```

Note how the string length of the PARAMETER lights can be assumed from the length of the constructor values. The strings in the constructor *must* all be the same length.

Previously defined constants (PARAMETERs) may also be used to initialise variables, consider,

```
INTEGER, DIMENSION(3,3) :: &
      unit_matrix_T = RESHAPE(unit_matrix, (/3,3/), ORDER=(/2,1/))
```

This assigns the transpose of unit_matrix to unit_matrix_T.

Question 21: Travelling Salesman Problem

A salesman travels between 5 towns A, B, C, D, E whose distances apart are given in the following table:-

	A	B	C	D	E
A	0	120	180	202	300
B		0	175	340	404
C			0	98	56
D				0	168
E					0

Which is the shortest route which takes in all the towns.

15.15 Allocatable Arrays

Fortran 90 allows arrays to be created on-the-fly; these are known as *deferred-shape* arrays and use dynamic heap storage (this means memory can be grabbed, used and then put back at any point in the program). This facility allows the creation of "temporary" arrays which can be created used and discarded at will.

Deferred-shape arrays are:

- declared like explicit-shape arrays but without the extents and with the ALLOCATABLE attribute:

```
INTEGER, DIMENSION(:), ALLOCATABLE :: ages
REAL, DIMENSION(:,:), ALLOCATABLE :: speed
```

- given a size in an ALLOCATE statement which reserves an area of memory for the object:

```
ALLOCATE(ages(1:10), STAT=ierr)
IF (ierr .NE. 0) THEN
 PRINT*, "ages: Allocation request denied"
END IF
ALLOCATE(speed(-lwb:upb,-50:0),STAT=ierr)
IF (ierr .NE. 0) THEN
 PRINT*, "speed: Allocation request denied"
END IF
```

  In the ALLOCATE statement we could specify a list of objects to create but in general one should only specify one array statement; if there is more than one object and the allocation fails it is not immediately possible to tell which allocation was responsible. The optional STAT= field reports on the success of the storage request, if it is supplied then the keyword must be used to distinguish it from an array that needs allocating. If the result, (ierr,) is zero the request was successful otherwise it failed. This specifier should be used as a matter of course.

There is a certain overhead in managing dynamic or ALLOCATABLE arrays — explicit-shape arrays are cheaper and should be used if the size is known and the arrays are persistent (are used for most of the life of the program). The dynamic or heap storage is also used with pointers and obviously only has a finite size — there may be a time when this storage runs out. If this happens there may be an option of the compiler to specify / increase the size of heap storage.

15.16 Deallocating Arrays

Heap storage should be reclaimed using the DEALLOCATE statement:

```
IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)
```

As a matter of course, the LOGICAL valued intrinsic inquiry function, ALLOCATED, should be used to check on the status of the array before attempting to DEALLOCATE because it is an error to attempt to deallocate an array that has not previously been allocated space or one which does not have the ALLOCATE attribute. Again one should only supply one array per DEALLOCATE statement and the optional STAT= field should always be used. ierr holds a value that reports on the success / failure of the DEALLOCATE request in an analogous way to the ALLOCATE statement.

Memory leakage will occur if a procedure containing an allocatable array (which does not possess the SAVE attribute) is exited without the array being DEALLOCATEd, (this constraint will be relaxed in Fortran 95). The storage associated with this array becomes inaccessible for the whole of the life of the program.

Consider the following sorting program which can handle any number of items,

```
PROGRAM sort_any_number
 !---------------------------------------------------------!
 ! Read numbers into an array, sort into ascending order   !
 ! and display the sorted list                             !
 !---------------------------------------------------------!
 INTEGER, DIMENSION(:), ALLOCATABLE :: nums
 INTEGER :: temp, I, K, n_to_sort, ierr

  PRINT*, 'How many numbers to sort'
  READ*,  n_to_sort

  ALLOCATE( nums(1:n_to_sort), STAT=ierr)
  IF (ierr .NE. 0) THEN
   PRINT*, "nums: Allocation request denied"
   STOP ! halts execution
  END IF

  PRINT*,  'Type in ',n_to_sort, 'values one line at a time'

  DO I=1,n_to_sort
   READ*, nums(I)
  END DO

  DO I = 1, n_to_sort-1
   DO K = I+1, n_to_sort
    IF(nums(I) > nums(K)) THEN
     temp = nums(K)      ! Store in temporary location
     nums(K) = nums(I)   ! Swap the contents over
     nums(I) = temp
    END IF
   END DO
  END DO

  DO I = 1, n_to_sort
   PRINT*, 'Rank ',I,' value is ',nums(I)
  END DO

  IF (ALLOCATED(nums)) DEALLOCATE(nums, STAT=ierr)
  IF (ierr .NE. 0) THEN
   PRINT*, "nums: Deallocation request denied"
  END IF

END PROGRAM sort_any_number
```

15.17 Masked Assignment — Where Statement

The WHERE statement is used when an array assignment is to be performed on a non-regular section of the LHS array elements, in other words, the whole array assignment is masked so that it only applies to specified elements. Masked array assignment is achieved using the WHERE statement:

```
WHERE (I .NE. 0) A = B/I
```

the effect of this statement is to perform the assignment `A(j,k) = B(j,k)/I(j,k)` for all values of `j` and `k` where the mask, `(I(j,k) .NE. 0)`, is `.TRUE.`. In the cases where the mask is `.FALSE.` no action is taken.

For example, if

$$\texttt{B} = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

and,

$$\texttt{I} = \begin{pmatrix} \boxed{2} & 0 \\ 0 & \boxed{2} \end{pmatrix}$$

then

$$\texttt{A} = \begin{pmatrix} \boxed{0.5} & \cdot \\ \cdot & \boxed{2.0} \end{pmatrix}$$

Only the indicated elements, corresponding to the non-zero elements of `I`, have been assigned to.

Conformable array sections may be used in place of the whole arrays, for example,

```
WHERE (I(j:k,j:k) .NE. 0) A(j+1:k+1,j-1:k-1) = B(j:k,j:k)/I(j:k,j:k)
```

is perfectly valid.

Question 22: `WHERE` Statement

Write a WHERE statement that will take a 2D INTEGER array and negate all odd-valued positive numbers.

15.18 Masked Assignment — Where Construct

Masked assignment may also be performed by a WHERE construct:

```
WHERE(A > 0.0)
  B = LOG(A)
  C = SQRT(A)
ELSEWHERE
  B = 0.0 ! C is NOT changed
ENDWHERE
```

the effect of this code block is to perform the assignments `B(j,k) = LOG(A(j,k))` and `C(j,k) = SQRT(A(j,k)` wherever `(A(j,k) > 0.0)` is `.TRUE.`. For the cases where the mask is `.FALSE.` the assignments in the `ELSEWHERE` block are made instead. Note that the `WHERE ... END WHERE` is *not* a control construct and cannot currently be nested. This constraint will be relaxed in Fortran 95.

In all the above examples the mask, (the logical expression,) must conform to the implied shape of each assignment in the body, in other words, in the above example all arrays must all conform.

The execution sequence is as follows: evaluate the mask, execute the `WHERE` block (in full) then execute the `ELSEWHERE` block. The separate assignment statements are executed sequentially but the individual elemental assignments within each statement are (conceptually) executed in parallel. It is not possible to have a scalar on the LHS in a `WHERE` and all statements must be array assignments.

Consider the following example from the Fortran 90 standard (pp296–298).

The code is a 3-D Monte Carlo simulation of state transition. Each gridpoint is a logical variable whose value can be interpreted as spin-up or spin-down. The transition between states is governed by a local probabilistic process where all points change state at the same time. Each spin either flips to the opposite state or not depending on the state of its six nearest neighbours. Gridpoints on the edge of the cube are defined by cubic periodicity — in other words the grid is taken to be replicated in all dimensions in space.

```
MODULE Funkt
CONTAINS
 FUNCTION RAND (m)
  INTEGER m
  REAL, DIMENSION(m,m,m) :: RAND
   CALL RANDOM_NUMBER(HARVEST = RAND)
  RETURN
 END FUNCTION RAND
END MODULE Funkt

PROGRAM TRANSITION
USE Funkt
IMPLICIT NONE
 INTEGER, PARAMETER :: n = 16
 INTEGER            :: iterations, i
 LOGICAL, DIMENSION(n,n,n) :: ising,     flips
 INTEGER, DIMENSION(n,n,n) :: ones,      count
 REAL, DIMENSION(n,n,n)    :: threshold
 REAL, DIMENSION(6)        :: p

 p = (/ 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 /)

 iterations  = 10
 ising = RAND(n) .LE. 0.5

 DO i = 1,iterations
  ones = 0
  WHERE (ising) ones = 1
  count = CSHIFT(ones, -1, 1) + CSHIFT(ones, 1, 1) &
        + CSHIFT(ones, -1, 2) + CSHIFT(ones, 1, 2) &
        + CSHIFT(ones, -1, 3) + CSHIFT(ones, 1, 3)
  WHERE (.NOT.ising) count = 6 - count
  threshold = 1.0
```

```
    WHERE (count == 4) threshold = p(4)
    WHERE (count == 5) threshold = p(5)
    WHERE (count == 6) threshold = p(6)
    flips = RAND(n) .LE. threshold
    WHERE (flips) ising = .NOT. ising
   ENDDO
  END PROGRAM TRANSITION
```

Note CSHIFT performs a circular shift on an array, for example, if

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$$

then

```
CSHIFT(A,-1)
```

is A shifted one place to the left with the left-most number wrapping around to the right,

$$A = \begin{pmatrix} 2 & 3 & 4 & 1 \end{pmatrix}$$

and is A shifted one place to the right

```
CSHIFT(A,1)
```

is

$$A = \begin{pmatrix} 4 & 1 & 2 & 3 \end{pmatrix}$$

It is also possible to specify a dimension for 2D and upward arrays. If

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

then

```
CSHIFT(B,1,1)
```

shifts the array one position in dimension 1 (downwards)

$$B = \begin{pmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

and

```
CSHIFT(B,1,2)
```

$$B = \begin{pmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{pmatrix}$$

and so on.

Question 23: Array Masked Array Assignment

Using an array constructor and the `WHERE` statement, implement the following algorithm for finding prime numbers:

1. define a vector, `Prime`, of size n,
2. initialise `Prime` such that `Prime(i) = i` for `i = 1,n`
3. set `i = 2`
4. for all $j >$ `i`, ($j \in$ `(i+1:n)`) if `Prime(j)` is exactly divisible by `i` then set `Prime(j) = 0`, [hint: use the `MOD` (remainder) intrinsic in conjunction with a `WHERE` statement.]
5. increment `i`,
6. if `i` equals n then exit
7. if `Prime(i)` is zero then goto step 5
8. goto step 4

Print out all non-zero entries of the vector (the prime numbers).

Hint: the `WHERE` statement is an *array assignment statement* and **not** a control construct therefore it cannot contain a `PRINT` statement. The `PACK` intrinsic can accept an array argument and a conformable `MASK` and will return a 1D vector of all the elements of the array where the corresponding mask elements are `.TRUE.`.

```
Print*, PACK(Array,Mask)
```

15.19 Vector-valued Subscripts

Index indirection can be introduced by using vector-valued subscripts. A one dimensional vector can be used to subscript an array in a dimension. The result of this is that an array section can be specified where the order of the elements do not follow a linear pattern. Consider:

```
INTEGER, DIMENSION(5) :: V=(/1,4,8,12,10/)
INTEGER, DIMENSION(3) :: W=(/1,2,2/)
```

then A(V) is shorthand for the irregular section that contains A(1), A(4), A(8), A(12), and A(10) in that order.

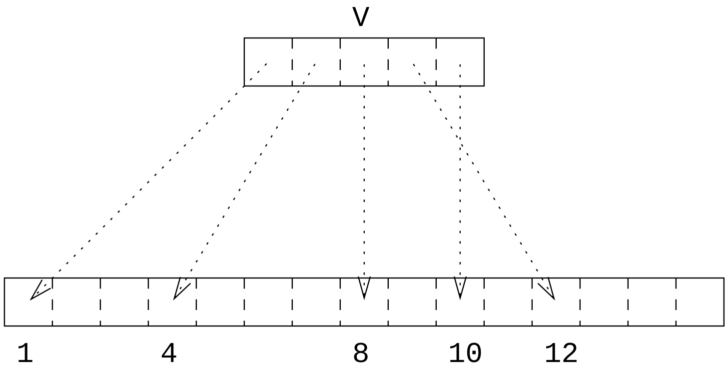


Figure 14: Subscripting Using a Vector

The following statement is a valid assignment to a 5 element section:

```
A(V) = 3.5
```

Likewise, if A contains the elements (5.3,6.4,..) then

```
C(1:3,1) = A(W)
```

would set the subsection C(1:3,1) to (5.3,6.4,6.4).

Vector-valued subscripts can be used on either side of the assignment operator, however, in order to preserve the integrity of parallel array operations it would be invalid to assign values to A(W) because A(2) would be assigned to twice. It must be ensured that subscripts on the LHS of the assignment operator are unique.

It is only possible to use one dimensional arrays as vector subscripts, if a 2D section is to be defined then two 1D vectors must be used, for example,

```
A(1) = SUM(C(V,W))
```

Note, vector subscripting is **very inefficient** and should not be used unless absolutely necessary.

Question 24: Vector Subscripts / MAXLOC

Generate an arbitrary 1D array, vector, filled with random numbers between 0 and 1. By using the MAXLOC intrinsic with a suitable mask set up an integer array, VSubs, which contains a permuted index set of vector such that Vector(VSubs(i)) > Vector(VSubs(i+1)) for all valid i.

You may find it useful to use the MAXLOC intrinsic. For example,

```
MAXLOC(VSubs)
```

returns an array containing the index of the largest element of the array VSubs, and,

```
MAXLOC(VSubs,MASK=VSubs.LT.VSubs(i))
```

returns the position of the largest element that is less than the value of VSubs(i). In both cases the result is a one element 1D array. (The array contains one element because VSub only has one dimension.)

16 Selected Intrinsic Functions

16.1 Random Number Intrinsic

RANDOM_NUMBER(HARVEST) is a useful intrinsic especially when developing and testing code. It is an elemental SUBROUTINE so when invoked with a REAL valued argument (which has INTENT(OUT)), it will return, in its argument, a pseudorandom number or conformable array of pseudorandom numbers in the range $0 \leq x < 1$.

For example,

```
REAL                   :: HARVEST
REAL, DIMENSION(10,10) :: HARVEYS
CALL RANDOM_NUMBER(HARVEST)
CALL RANDOM_NUMBER(HARVEYS)
```

will assign a random number to the scalar variable HARVEST and an array of (different) random numbers to HARVEYS. This subroutine is very useful for numeric applications where large arrays need to be generated in order to test or time codes.

The random number generator can be seeded by user specified values. The seed is an integer array of a compiler dependent size. Using the same seed on separate invocations will generate the same sequence of random numbers.

RANDOM_SEED([SIZE=<*int*>]) finds the size of the seed.

RANDOM_SEED([PUT=<*array*>]) seeds the random number generator.

For example,

```
CALL RANDOM_SEED(SIZE=isze)
CALL RANDOM_SEED(PUT=IArr(1:isze))
CALL RANDOM_NUMBER(HARVEST)

PRINT*, "Type in a scalar seed for the generator"
READ*, iseed
CALL RANDOM_SEED(PUT=(/(iseed, i = 1, isze)/))
CALL RANDOM_NUMBER(HARVEST)

ALLOCATE(ISeedArray(isze))
PRINT*, "Type in a ", isze," element array as a seed for the generator"
READ*, ISeedArray
```

```
CALL RANDOM_SEED(PUT=ISeedArray)
CALL RANDOM_NUMBER(HARVEST)

DEALLOCATE(ISeedArray)
```

Using the same seed on separate executions will generate the same sequence of random numbers.

There are other optional arguments which can be used to report on which seed is in use and report on how big an array is needed to hold a seed. This procedure may also be called with no arguments in order to initialise the random number generator.

Question 25: Random Number Generation

Using the intrinsic subroutine RANDOM_NUMBER, write a program to simulate the throw of a die.

16.2 Vector and Matrix Multiply Intrinsics

There are two types of intrinsic matrix multiplication these should *always* be used when appropriate as they will be the most efficient method of calculation:

- DOT_PRODUCT(VEC1, VEC2)

  This is the inner (dot) product of two rank 1 arrays. Clearly, VEC1, VEC2 must conform in size and must be one dimensional. Care must be taken not to confuse this intrinsic with DPROD the DOUBLE PRECISION product function or PRODUCT the intra-matrix product (see Section 16.5).

  An example of use is,

  ```
  DP = DOT_PRODUCT(A,B)
  ```

  which is equivalent to:

  ```
  DP = A(1)*B(1) + A(2)*B(2) + ...
  ```

  or

  ```
  DP = SUM(A*B)
  ```

  The result is also defined for COMPLEX and LOGICAL array arguments. For COMPLEX the result is,

  ```
  DP = SUM(CONJG(A)*B)
  ```

  and for LOGICAL,

  ```
  DP = LA(1).AND.LB(1) .OR. LA(2).AND.LB(2) .OR. ...
  ```

- MATMUL(MAT1, MAT2)

  This is the 'traditional' matrix-matrix multiplication and is **not** equivalent to MAT1*MAT2. There are certain restrictions placed on the function arguments which say that the arrays must match in specific dimensions, they do not have to be conformable:

- ⋄ if MAT1 has shape (n, m) and MAT2 shape (m, k) then the result has shape (n, k);
- ⋄ if MAT1 has shape (m) and MAT2 shape (m, k) then the result has shape (k);
- ⋄ if MAT1 has shape (n, m) and MAT2 shape (m) then the result has shape (n);

Element (i,j) of the result is,

```
SUM(MAT1(i,:)*MAT2(:,j))
```

The result is also defined for LOGICAL arguments,

```
ANY(MAT1(i,:).AND.MAT2(:,j))
```

If A and B are set up as follows,

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

and

$$B = \begin{pmatrix} 4 & 8 \\ 3 & 7 \\ 2 & 6 \\ 1 & 5 \end{pmatrix}$$

then the following program

```
PROGRAM DEMO
 INTEGER :: A(2,4)
 INTEGER :: B(4,2)
 A(1,:) = (/1,2,3,4/)
 A(2,:) = (/5,6,7,8/)
 B(:,1) = (/4,3,2,1/)
 B(:,2) = (/8,7,6,5/)
 PRINT*, "DOT_PRODUCT(A(1,:),A(2,:)) = ", DOT_PRODUCT(A(1,:),A(2,:))
 PRINT*, "MATMUL(A,B) = ",MATMUL(A,B)
END PROGRAM DEMO
```

gives

```
DOT_PRODUCT(A(1,:),A(2,:)) =  70
MATMUL(A,B) = 20 60 60 164
```

Question 26: MATMUL Intrinsic

For the declarations

```
REAL, DIMENSION(100,100) :: A, B, C
```

what is the difference between C=MATMUL(A,B) and C=A*B.

16.3 Maximum and Minimum Intrinsics

There are two intrinsics in this class:

- `MAX(SOURCE1,SOURCE2[,SOURCE3[,...]])`— returns the maximum values over all source objects
- `MIN(SOURCE1,SOURCE2[,SOURCE3[,...]])`— returns the minimum values over all source objects

For example,

- `MAX(1,2,3)` is 3
- `MIN(1,2,3)` is 1

The list of source objects are searched from left to right (as indicated in the diagram below). If two values are equal then it is the first that is selected (as also indicated in the diagram below).

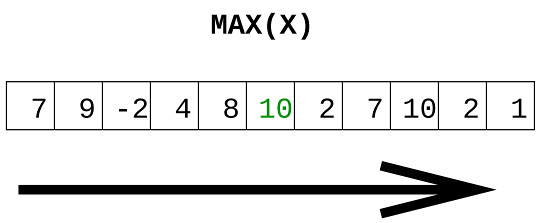


Figure 15: Visualisation of the `MAX` Intrinsic

The `MAX` and `MIN` intrinsics may also accept array arguments. The result is the same shape and size as each argument, for example,

- `MIN((/1,2/),(/-3,4/))` is `(/-3,2/)`
- `MAX((/1,2/),(/-3,4/))` is `(/1,4/)`

Question 27: `MAX` **and** `MIN`

What is the value of:

- `MAX( (/2,7,3,5,9,1/),(/1,9,5,3,7,2/))`?
- `MIN( (/2,7,3,5,9,1/),(/1,9,5,3,7,2/))`?

16.4 Array Location Intrinsics

There are two intrinsics in this class:

- `MAXLOC(SOURCE[,MASK])`

  Returns a one dimensional array containing the location of the *first* maximal value in an array under an optional mask. If the `MASK` is present the only elements considered are where the mask is `.TRUE.`. Note that the result is **always** array valued.

  The source array is searched from left to right and the position of the first occurrence of the maximum value is returned,

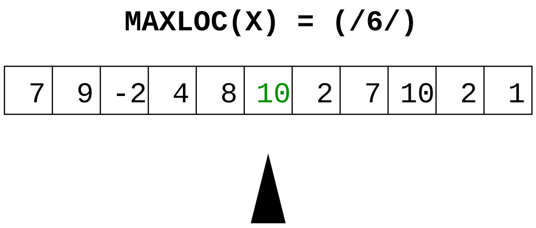


Figure 16: Visualisation of `MAXLOC` When Applied to a 1D Array

  Consider this further 2D example, if

$$\texttt{Array} = \begin{pmatrix} 0 & -1 & 1 & 6 & -4 \\ 1 & -2 & 5 & 4 & -3 \\ 3 & 8 & 3 & -7 & 0 \end{pmatrix}$$

  then

  - `MAXLOC(Array)` is `(/3,2/)` corresponding to the location of value 8.
  - `MAXLOC(Array,Array.LE.7)` is `(/1,4/)`
    Only the following elements are considered,

$$\texttt{Array} = \begin{pmatrix} 0 & -1 & 1 & \boxed{6} & -4 \\ 1 & -2 & 5 & 4 & -3 \\ 3 & & 3 & -7 & 0 \end{pmatrix}$$

    the maximal value is at the location indicated.
  - `MAXLOC(MAXLOC(Array,Array.LE.7))`
    `MAXLOC(Array,Array.LE.7)` gives `(/1,4/)` so the overall result is `(/2/)` (array valued) corresponding to the location that holds the largest element of the array `(/1,4/)`.

- `MINLOC(SOURCE[,MASK])`

  Returns a one dimensional array containing the location of the *first* minimal value in an array under an optional mask.

  - `MINLOC(Array)` is `(/3,4/)`
    The minimal value -7 is element `A(3,4)`.
  - `MINLOC(Array,Array.GE.7)` is gives `(/3,2/)`.
  - `MINLOC(MINLOC(Array,Array.GE.7))`
    This is effectively `MINLOC((/3,2/))` so the result is `(/2/)` (array valued) corresponding to the second element of the array `(/3,2/)`.

Question 28: `MAXLOC`

What is the value of:

- ☐ `MAXLOC((/2,7,3,5,9,1/))`?
- ☐ `MAXVAL((/2,7,3,5,9,1/))`?
- ☐ `MINLOC((/2,7,3,5,9,1/))`?
- ☐ `MINVAL((/2,7,3,5,9,1/))`?

If

$$A = \begin{pmatrix} 0 & -5 & 8 & 3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{pmatrix}$$

what is

- ☐ `MAXLOC(A, MASK = A .LT. 5)`?
- ☐ `MAXVAL(A, MASK = A .LT. 5)`?
- ☐ `MAXLOC(A, MASK = A .LT. 4)`?
- ☐ `MAXVAL(A, MASK = A .LT. 4)`?

16.5 Array Reduction Intrinsics

Reduction functions are aptly named because an array is operated upon and a result obtained which has a smaller rank than the original source array. For a rank n array, if `DIM` is absent or $n = 1$ then the result is scalar, otherwise the result is of rank $n - 1$.

- ☐ `SUM(SOURCE[,DIM][,MASK])`
  - ◇ `SUM` returns the sum of array elements, along an optionally specified dimension under an optionally specified mask.
  - ◇ if `DIM` is absent the whole array is considered and the result is a scalar.

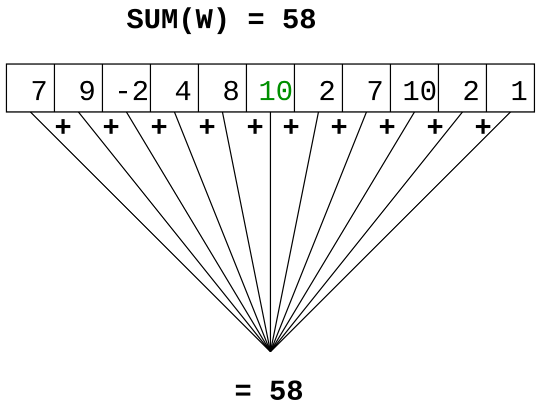


Figure 17: SUM of a 1D Array

If DIM is not specified for the SUM of a 2D array then the result is obtained by adding all the elements together

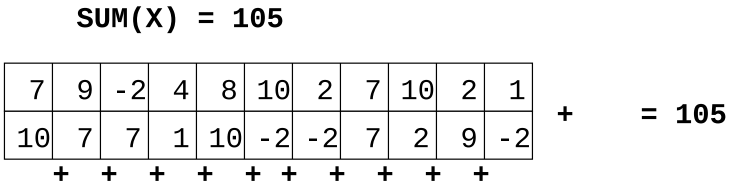


Figure 18: SUM of a 2D Array

◇ if DIM is specified the result is an array of rank $n - 1$ of sums, for example, summing down the columns

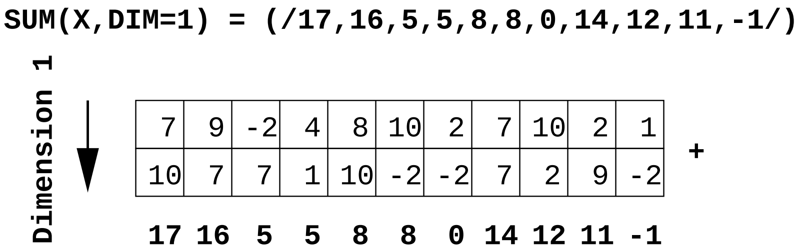


Figure 19: Summing along Dimension 1 of a 2D Array

or along the rows,

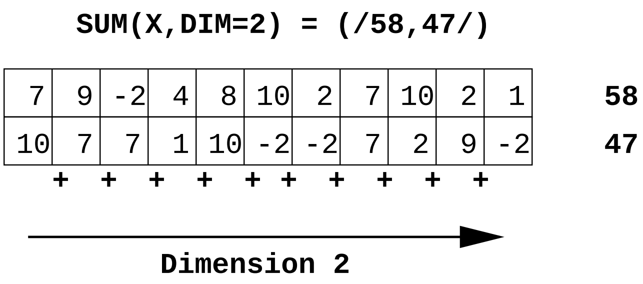


Figure 20: Summing along Dimension 2 of a 2D Array

- ◇ if `MASK` is present then the sum only involves elements of `SOURCE` which correspond to `.TRUE.` elements of `MASK`, for example, only the elements larger than 6 are considered,

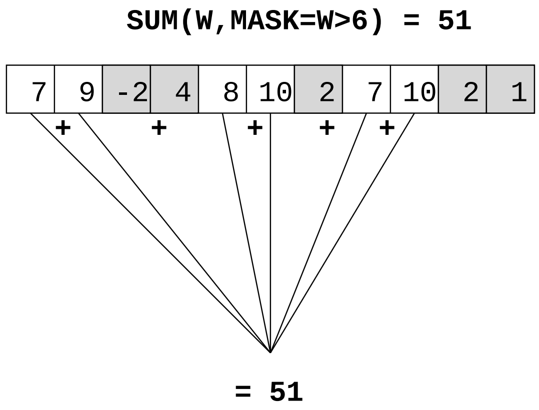


Figure 21: `SUM` Under the Control of a Mask

- ◇ if the array is zero sized then the sum is 0

☐ `PRODUCT(SOURCE[,DIM][,MASK])`

- ◇ `PRODUCT` returns the product of all array elements, along an optionally specified dimension under an optionally specified mask,
- ◇ if `DIM` is absent the whole array is considered and the result is a scalar.
- ◇ if `DIM` is specified the result is an array of rank $n-1$ of products, for example, if

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

```
PRINT*, PRODUCT(A,DIM=1)
PRINT*, PRODUCT(A,DIM=2)
```

gives

```
 2 12 30
15 48
```

- ◇ if `MASK` is present then the product only involves elements of `SOURCE` which correspond to `.TRUE.` elements of `MASK`, for example,

```
PRINT*, PRODUCT(A,MASK=A.LT.4)
```

gives

```
6
```

- ◇ if the array is zero sized then the product is 1.

□ `ALL(MASK[,DIM])`

- ◇ `ALL` returns `.TRUE.` if *all* values of the mask are `.TRUE.` (along an optionally specified) dimension `DIM`. If `DIM` is specified then the result is an array of rank $n-1$, otherwise a scalar is returned.

  For example, consider a 2D array, if `DIM=2` then the function returns a 1D vector with the result being as if the `ALL` function has been applied to each column in turn. If `DIM=1` the result is as if the `ALL` function had been applied to each row in turn.

  If `A` is as before, and

$$B = \begin{pmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{pmatrix}$$

  then the following

```
PRINT*, ALL(A.NE.B,DIM=1)
```

  gives

```
 T F F
```

  recall that dimension 1 runs up and down the page.

  Similarly

```
PRINT*, ALL(A.NE.B,DIM=2)
```

  gives,

```
 F F
```

  where dimension 2 run across the page.

- ◇ if `DIM` is absent then the whole array is considered, for example,

```
PRINT*, ALL(A.NE.B)
```

  gives the scalar value,

```
 F
```

- ◇ if the array is zero sized then `ALL` returns `.TRUE.`,

□ `ANY(MASK[,DIM])`

- ◇ `ANY` returns `.TRUE.` if *any* values of the mask are `.TRUE.` (along an optionally specified dimension `DIM`). If `DIM` is given then the result is an array of rank $n-1$, for example,

```
PRINT*, ANY(A.NE.B,DIM=1)
PRINT*, ANY(A.NE.B,DIM=2)
```

  gives

```
T F T
T T
```

- ◇ if DIM is absent then the whole array is considered, for example,

```
PRINT*, ANY(A.NE.B)
```

gives the scalar value,

```
 T
```

- ◇ if the array is zero sized then ANY returns .FALSE..

□ COUNT(MASK[,DIM])

- ◇ COUNT returns the number of .TRUE. elements in a specified LOGICAL array along dimension DIM. The result is an array of rank $n-1$, for example,

```
PRINT*, COUNT(A.NE.B,DIM=1)
PRINT*, COUNT(A.NE.B,DIM=2)
```

gives

```
 2 0 1
 1 2
```

- ◇ if DIM is absent then the whole array is considered, for example,

```
PRINT*, COUNT(A.NE.B)
```

gives the scalar,

```
 3
```

- ◇ if the array is zero sized then COUNT returns zero.

□ MAXVAL(SOURCE[,DIM][,MASK])

- ◇ MAXVAL returns the maximum values in an array along an optionally specified dimension under an optionally specified mask,
- ◇ if DIM is specified the result is an array of rank $n-1$ of maximum values in other dimensions, for example,

```
PRINT*, MAXVAL(A,DIM=1)
PRINT*, MAXVAL(A,DIM=2)
```

gives

```
 2 4 6
 5 6
```

- ◇ if DIM is absent the whole array is considered and the result is a scalar.
- ◇ if MASK is present then the survey is only performed on elements of SOURCE which correspond to .TRUE. elements of MASK, for example,

```
PRINT*, MAXVAL(A,MASK=A.LT.4)
```

only considers elements of A that are less than 4 and gives

```
3
```

- ◇ the largest negative number of the appropriate kind is returned if the array is zero sized.

□ MINVAL(SOURCE[,DIM][,MASK])

- ◇ `MINVAL` returns the minimum value in an array along an optionally specified dimension under an optionally specified mask,
- ◇ if `DIM` is specified the result is an array of rank $n-1$ of minimum values in other dimensions, for example,

```
PRINT*, MINVAL(A,DIM=1)
PRINT*, MINVAL(A,DIM=2)
```

gives

```
 1 3 5
 1 2
```

- ◇ if `DIM` is absent the whole array is considered and the result is a scalar.
- ◇ if `MASK` is present then the survey is only performed on elements of `SOURCE` which correspond to `.TRUE.` elements of `MASK`, for example,

```
PRINT*, MINVAL(A,MASK=A.GT.4)
```

gives

```
5
```

- ◇ the smallest positive number of the appropriate kind is returned if the array is zero sized.

Question 29: Summation Example

Which five consecutive numbers have the greatest sum:

```
6.3 7.6 9.2 3.4 5.6 7.23 9.76 6.83 5.45 4.56
4.86 5.8 6.4 7.43 7.87 8.6 9.25 8.9 8.4 7.23
```

Question 30: Operations on arrays and array intrinsics

Declare constant arrays `A` and `X` where:

$$\mathtt{A} = \begin{pmatrix} -4 & 5 & 9 \\ 6 & -7 & 8 \end{pmatrix}$$

$$\mathtt{X} = \begin{pmatrix} 1.5 \\ -1.9 \\ 1.7 \\ -1.2 \\ 0.3 \end{pmatrix}$$

1. Using the relevant intrinsics set `M` and `N` to be the extents of `A`,
2. Print the array `A` out row by row,
3. Write a Fortran 90 program which use intrinsics to print out the following:

(a) The sum of the product of the columns of `A` (use intrinsics)
(b) The product of the sum of the row elements of `A` (use intrinsics)
(c) The sum of squares of the elements of `X`
(d) The mean of the elements of `X`
(e) The sum of the positive elements of `X`
(f) The infinity norm of `X` i.e. the largest of $(|x_i|, i = 1, n)$
(g) The one norm of `A` i.e. the largest column sum of $|a_{ij}|$

Question 31: Salaries Example

The salaries received by employees of a company are

10500, 16140, 22300, 15960, 14150, 12180, 13230, 15760, 31000

and the position in the hierarchy of each employee is indicated by a corresponding category thus

1, 2, 3, 2, 1, 1, 1, 2, 3

Write a program to find the total cost to the company of increasing the salary of people in categories 1, 2 and 3 by 5%, 4% and 2% respectively.

Module 6: Procedures

17 Program Units

Fortran 90 has two main program units:

- main PROGRAM,

  The place where execution begins and where control should eventually return before the program terminates. The main program may contain any number of procedures.

- MODULE.

  A program unit which can also contain procedures and declarations. It is intended to be attached to any other program unit where the entities defined within it become accessible. A module is similar to a C++ class.

  MODULE program units are new to Fortran 90 and are supposed to replace the unsafe FORTRAN 77 features such as COMMON, INCLUDE, BLOCK DATA as well as adding a much needed (limited) 'object oriented' aspect to the language. Their importance cannot be overstressed and they should be used whenever possible.

There are two classes of procedure:

- SUBROUTINE,

  A parameterised named sequence of code which performs a specific task and can be invoked from within other program units by the use of a CALL statement, for example,

  ```
  CALL PrintReportSummary(CurrentFigures)
  ```

  Here, control will pass into the SUBROUTINE named PrintReportSummary, after the SUBROUTINE has terminated control will pass back to the line following the CALL.

- FUNCTION,

  As a SUBROUTINE but returns a result in the function name (of any specified type and kind). This can be compared to a mathematical function, say, $f(x)$. An example of a FUNCTION call could be:

  ```
  PRINT*, "The result is", f(x)
  ```

  Here, the value of the function f (with the argument x) is substituted at the appropriate point in the output.

Procedures are generally contained within a main program or a module. It is also possible to have 'stand alone' or EXTERNAL procedures, these will be discussed later (see Section 28).

17.1 Main Program Syntax

This is the only compulsory program unit, every program must have one:

[PROGRAM [< *main program name* >]]
        ...
    < *declaration of local objects* >
        ...
    < *executable stmts* >
        ...
[CONTAINS
    < *internal procedure definitions* >]
END [PROGRAM [< *main program name* >]]

The PROGRAM statement and < *main program name* > are optional, however, it is good policy to always use them. < *main program name* > can be any valid Fortran 90 name.

The main program contains declarations and executable statements and may also contain internal procedures. These internal procedures are separated from the surrounding program unit, the *host* (in this case the main program), by a CONTAINS statement. Internal procedures may only be called from within the surrounding program unit and automatically have access to all the host program unit's declarations but may also override them. Please note that some implementation of HPF may not yet support internal procedures.

```
PROGRAM Main
 ! ...
CONTAINS ! Internal Procs
   SUBROUTINE Sub1(..)
    ! Executable stmts
   END SUBROUTINE Sub1
    ! etc.
   FUNCTION Funkyn(...)
    ! Executable stmts
   END FUNCTION Funkyn
 END PROGRAM Main
```

Figure 22: Schematic Diagram of a Main Program

Internal procedures may not contain further internal procedures, in other words the nesting level is a maximum of 1. The diagram shows two internal procedures, Sub1 and Funkyn however, there may be any number of internal procedures (subroutines or functions) which are wholly contained within the main program.

The main program may also contain calls to external procedures. This will be discussed later (see Section 28).

The main program must contain an END statement as its last (non-blank) line. For neatness sake this should really be suffixed by PROGRAM (so it reads END PROGRAM) and should also have the name of the program attached too. Using as descriptive as possible END statements helps to reduce confusion.

17.1.1 Main Program Example

The following example demonstrates a main program which calls an intrinsic function, (FLOOR), and an internal procedure, (Negative)

```
PROGRAM Main
 IMPLICIT NONE
 REAL x
 INTRINSIC FLOOR
  READ*, x
  PRINT*, FLOOR(x)
  PRINT*, Negative(x)
CONTAINS
 REAL FUNCTION Negative(a)
  REAL, INTENT(IN) :: a
   Negative = -a
 END FUNCTION Negative
END PROGRAM Main
```

Although not totally necessary, the intrinsic procedure is declared in an INTRINSIC statement (the type is not needed — the compiler knows the types of all intrinsic functions).

The internal procedure is 'contained within' the main program so does not require declaring in the main program;

The compiler is able to 'see' the procedure and therefore knows its result type, number and type of arguments.

17.2 Procedures

A procedure, such as an intrinsic function, is an abstracted block of parameterised code that performs a particular task. Procedures should generally be used if a task has to be performed two or more times, this will cut down on code duplication.

Before writing a procedure the first question should be: "Do we really need to write this or does a routine already exist?" Very often a routine with the functionality already exists, for example, as an intrinsic procedure or in a library somewhere. (Fortran 90 has 113 intrinsic procedures covering a variety of functionality and the NAg f190 Numerical Library contains over 300 mathematic procedures so there is generally a wide choice!)

The NAg library deals with solving numerical problems and is ideal for engineers and scientists. f190, the NAg Fortran 90 Mk I library, has just been released as a successor to the well respected and popular FORTRAN 77 library which contains at least 1140 routines.

Other libraries include: BLAS, (Basic Linear Algebra Subroutines,) for doing vector, matrix-vector and matrix-matrix calculations, (these should *always* be used if possible); IMSL (Visual Numerics), akin to NAg Library; LaPACK, linear algebra package; Uniras, graphics routines, very comprehensive. Many of these packages will be optimised and shipped along with the compiler.

Note that the HPFF defined a set of routines that should be available as part of an HPF compilation system. If the target platform is to be a parallel computer it will be worth investigating further; a number of Fortran 90 version of these procedures exist, for example, at LPAC

http://www.lpac.ac.uk/SEL-HPC/Materials/HPFlibrary/HPFlibrary.html

.

As the use of Fortran 90 grows many useful (portable) library modules will be developed which contain routines that can be USEd by any Fortran 90 program. See World Wide Web Fortran Market

http://www.fortran.com/fortran/market.html

.

There is also an auxiliary Fortran 90 standard known as the "Varying String" module. This is to be added to the Fortran 95 standard and will allow users to define and use objects of type VARYING_STRING where CHARACTER objects would normally be used. The Standard has already been realised in a module (by Lawrie Schonfelder at Liverpool University). All the intrinsic operations and functions for character variables have be overloaded so that VARYING_STRING objects can be used in more or less the same way as other intrinsic types. which contain routines that can be USEd by any Fortran 90 program. See World Wide Web Fortran Market

http://www.fortran.com/fortran/market.html

.

If a procedure is to be written from scratch then the following guidelines should be followed:

- It is generally accepted that procedures should be no more than 50 lines long in order to keep the control structure simple and to keep the number of program paths to a minimum.
- Procedures should be as flexible as possible to allow for software reuse. Try to pass as many of the variable entities referenced in a procedure as actual arguments and do not rely on global storage or host association unless absolutely necessary.
- Try to give procedures meaningful names and initial descriptive comments.
- There is absolutely no point in reinventing the wheel — if a procedure or collection of procedures already exist as intrinsic functions or in a library module then they should be used.

17.3 Subroutines

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  CALL OutputFigures(Numbers)
 .....
CONTAINS
```

```
 SUBROUTINE OutputFigures(Numbers)
  REAL, DIMENSION(:), INTENT(IN) :: Numbers
   PRINT*, "Here are the figures", Numbers
 END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

The subroutine here simply prints out its argument. Internal procedures can 'see' all variables declared in the main program (and the `IMPLICIT NONE` statement). If an internal procedure declares a variable which has the same name as a variable from the main program then this supersedes the variable from the outer scope for the length of the procedure.

Using a procedure here allows the output format to be changed easily. To alter the format of all outputs, it is only necessary to change on line within the procedure.

Internal subroutines lie between `CONTAINS` and `END PROGRAM` statements and have the following syntax

`SUBROUTINE` <*procname*>[(<*dummy args*>)]
    <*declaration of dummy args*>
    <*declaration of local objects*>
        ...
    <*executable stmts*>
`END` [`SUBROUTINE` [<*procname*>]]

(Recall that not all HPF compilers implement Internal subroutines.)

A `SUBROUTINE` may include calls to other procedures either from the same main program, from an attached module or from an external file. Note how, in the same way as a main program, a `SUBROUTINE` must terminate with an `END` statement. It is good practice to append `SUBROUTINE` and the name of the routine to this line as well.

Fortran 90 also allows recursive procedures (procedure that call themselves). In order to promote optimisation a recursive procedure must be specified as such — it must have the `RECURSIVE` keyword at the beginning of the subroutine declaration (see Section 18.10).

Question 32: Simple example of Subroutine

Write a main program and internal subroutine that returns, as its first argument, the sum of two real numbers.

17.4 Functions

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  PRINT*, F(a,b)
 .....
CONTAINS
 REAL FUNCTION F(x,y)
  REAL, INTENT(IN) :: x,y
```

```
    F = SQRT(x*x + y*y)
  END FUNCTION F
 END PROGRAM Thingy
```

Functions operate on the same principle as SUBROUTINEs, the only difference being that a function returns a value. In the example, the line

```
PRINT*, F(a,b)
```

will substitute the value returned by the function for F(a,b), in other words, the value of $\sqrt{a^2+b^2}$.

Just like subroutines, functions also lie between CONTAINS and END PROGRAM statements. They have the following syntax:

[< *prefix* >] FUNCTION < *procname* >([< *dummyargs* >])
    < *declaration of dummy args* >
    < *declaration of local objects* >
        ...
    < *executable stmts, assignment of result* >
END [FUNCTION [< *procname* >]]

It is also possible to declare the function type in the declaration area instead of in the header:

FUNCTION < *procname* >([< *dummy args* >])
    < *declaration of dummy args* >
    < *declaration of result type* >
    < *declaration of local objects* >
        ...
    < *executable stmts, assignment of result* >
END [FUNCTION [< *procname* >]]

This would mean that the above function could be equivalently declared as:

```
FUNCTION F(x,y)
 REAL             :: F
 REAL, INTENT(IN) :: x,y
  F = SQRT(x*x + y*y)
END FUNCTION F
```

(Recall that not all HPF compilers implement internal functions.)

Functions may also be recursive, see Section 18.10, and may be either scalar or array valued (including user defined types and pointers). Note that, owing to the possibility of confusion between an array reference and a function reference, the parentheses are **not** optional.

Question 33: Simple example of a Function

Write a main program and an internal function that returns the sum of two real numbers supplied as arguments.

Question 34: Random Number Generation

Write a function which simulates a throw of two dice and returns the total score. Use the intrinsic subroutine RANDOM_NUMBER to obtain pseudo-random numbers to simulate the throw of the dice.

17.5 Argument Association

Recall, in the SUBROUTINE example we had an invocation:

```
CALL OutputFigures(NumberSet)
```

and a declaration,

```
SUBROUTINE OutputFigures(Numbers)
```

An argument in a call statement (in an invocation), for example, NumberSet, is called an *actual argument* since it is the true name of the variable. An argument in a procedure declaration is called a *dummy argument*, for example, Numbers as it is a substitute for the true name. A reference to a dummy argument is really a reference to its corresponding actual argument, for example, changing the value of a dummy argument actually changes the value of the actual argument. Dummys and actuals are said to be *argument associated*. Procedures may have any number of such arguments but actuals and dummys must correspond in number, type, kind and rank. [FORTRAN 77 programs which flouted this requirement were not standard conforming but there was no way for the compiler to check.]

For the above call, Numbers is the dummy argument and NumberSet is the actual argument.

Consider,

```
PRINT*, F(a,b)
```

and

```
REAL FUNCTION F(x,y)
```

here, the actual arguments a and b are associated with the dummy arguments x and y.

If the value of a dummy argument changes then so does the value of the actual argument.

17.6 Local Objects

In the following procedure

```
SUBROUTINE Madras(i,j)
 INTEGER, INTENT(IN) :: i, j
 REAL                :: a
 REAL, DIMENSION(i,j):: x
```

a, and x are known as *local objects* and x will probably have a different size and shape on each call.. They:

- are created each time a procedure is invoked,
- are destroyed when the procedure completes,
- *do not* retain their values between calls,
- do not exist in the programs memory between calls.

So, when a procedure is called, any local objects are bought into existence for the duration of the call. Thus if an object is assigned to on one call, the next time the program unit is invoked a totally different instance of that object is created with no knowledge of what happened during the last procedure call meaning that all values are lost.

The space usually comes from the programs stack.

17.7 Argument Intent

In order to facilitate efficient compilation and optimisation hints, in the form of attributes, can be given to the compiler as to whether a given dummy argument will:

1. hold a value on procedure entry which remains unchanged on exit — `INTENT(IN)`.
2. not be used until it is assigned a value within the procedure — `INTENT(OUT)`.
3. hold a value on procedure entry which may be modified and then passed back to the calling program — `INTENT(INOUT)`.

For example,

```
SUBROUTINE example(arg1,arg2,arg3)
  REAL, INTENT(IN) :: arg1
  INTEGER, INTENT(OUT) :: arg2
  CHARACTER, INTENT(INOUT) :: arg3
  REAL r
   r = arg1*ICHAR(arg3)
   arg2 = ANINT(r)
   arg3 = CHAR(MOD(127,arg2))
END SUBROUTINE example
```

It can be seen here that:

- arg1 is unchanged within the procedure,
- the value of arg2 is not used until it has been assigned to,
- arg3 is used and then reassigned a value.

The use of INTENT attributes is not essential but it allows good compilers to check for coding errors thereby enhancing safety. If an INTENT(IN) object is assigned a value or if an INTENT(OUT) object is not assigned a value then errors will be generated at compile time.

Question 35: Erroneous Code

What is wrong with the following internal procedure?

```
SUBROUTINE Mistaken(A,B,C)
 IMPLICIT NONE
 REAL, INTENT(IN) :: A
 REAL, INTENT(OUT) :: C
 A = 2*C
END SUBROUTINE Mistaken
```

17.8 Scope

The *scope* of an entity is the range of a program within which an entity is visible and accessible. Most entities have a scope which is limited to the program unit in which they are declared, but in special circumstances some entities can have a scope that is wider than this. The best way to provide global data is by implementing a MODULE which contains the required global data declarations and then USEing it wherever required. (Using COMMON to achieve this is *strongly* discouraged as this method of global data declaration is considered to be unsafe, obtuse and outmoded.)

17.8.1 Host Association

In FORTRAN 77, different routines were entirely separate from each other, they did not have access to each others variable space and could only communicate through argument lists or by global storage (COMMON); such procedures are known as external.

Procedures in Fortran 90 may contain internal procedures which are only visible within the program unit in which they are declared, in other words they have a local scope. Consider the following example,

```
PROGRAM CalculatePay
 IMPLICIT NONE
 REAL :: Pay, Tax, Delta
 INTEGER :: NumberCalcsDone = 0
 Pay = ...;  Tax = ... ; Delta = ...
 CALL PrintPay(Pay,Tax)
 Tax = NewTax(Tax,Delta)
  ....
CONTAINS
 SUBROUTINE PrintPay(Pay,Tax)
  REAL, INTENT(IN) :: Pay, Tax
  REAL :: TaxPaid
   TaxPaid = Pay * Tax
   PRINT*, TaxPaid
   NumberCalcsDone = NumberCalcsDone + 1
 END SUBROUTINE PrintPay
 REAL FUNCTION NewTax(Tax,Delta)
  REAL, INTENT(IN) :: Tax, Delta
   NewTax =  Tax + Delta*Tax
   NumberCalcsDone = NumberCalcsDone + 1
 END FUNCTION NewTax
```

```
END PROGRAM CalculatePay
```

PrintPay is an internal subroutine of CalculatePay and has access to NumberCalcsDone. It can be thought of as a *global* variable. It is said to be available to the procedures by *host association*. The variables Pay and Tax, on the other hand, are passed as arguments to the procedures. This means that they are available by argument association.

The decision of whether to give visibility to an object by host association or by argument association is tricky — there are no hard and fast rules. If, for example, Pay, Tax and Delta had not been passed to the procedures as arguments but had been made visible by host association instead, then there would be no discernible difference in the results that the program produces. Likewise, NumberCalcsDone could have been communicated to both procedures by argument association instead of by host association. In a sense, the method that is used will depend on personal taste or a specific 'in-house' coding-style. Here, Pay, Tax and Delta are not used in every single procedure in the same way as NumberCalcsDone which acts in a more global way!

NewTax cannot access any of the local declarations of PrintPay (for example, TaxPaid,) and vice-versa. NewTax and PrintPay can be thought of a resting at the same scoping level whereas the containing program, CalculatePay is at an outer (higher) scoping level (see Figure 17.1).

PrintPay can invoke other internal procedures which are contained by the same outer program unit (but cannot call itself as it is not recursive, see Section 18.10 for discussion about recursion).

Upon return from PrintPay the value of NumberCalcsDone will have increased by one owing to the last line of the procedure.

Question 36: Standard Deviation

Write a program which contains an internal function that returns the standard deviation from the mean of an array of real values. Note that if the mean of a sequence of values $(x_i, i = 1, n)$ is denoted by m then the standard deviation, s, is defined as:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - m)^2}{n}}$$

[Hint: In Fortran 90 SUM(X) is the sum of the elements of X.]

To demonstrate correctness print out the standard deviation of the following numbers (10 of 'em):

```
5.0 3.0 17.0 -7.56 78.1 99.99 0.8 11.7 33.8 29.6
```

and also for the following 14,

```
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0
```

17.8.2 Example of Scoping Issues

Consider the following example,

```
PROGRAM Proggie        ! scope Proggie
 IMPLICIT NONE
 REAL ::  A, B, C      ! scope Proggie
 CALL sub(A)           ! scope Proggie
CONTAINS
 SUBROUTINE Sub(D)     ! scope Sub
  REAL :: D            ! D is dummy (alias for A)
  REAL :: C            ! local C (diff from Proggie's C)
   C = A**3            ! A cannot be changed
   D = D**3 + C        ! D can be changed
   B = C               ! B from Proggie gets new value
 END SUBROUTINE Sub
 SUBROUTINE AnuvvaSub  ! scope AnuvvaSub
   REAL :: C           ! another local C (unrelated)
     .....
 END SUBROUTINE AnuvvaSub
END PROGRAM Proggie
```

This demonstrates most of the issues concerning the scope of names in procedures.

If an internal procedure declares an object with the same name as one in the host (containing) program unit then this new variable supersedes the one from the outer scope for the duration of the procedure, for example, `Sub` accesses `B` from `Proggie` but supersedes `C` by declaring its own local object called `C`. This `C` (in `Sub`) is **totally unrelated** to the `C` of `Proggie`.

Internal procedures may have dummy arguments, however, any object used as an actual argument to an internal procedure cannot be changed by referring to it by its original name in that procedure; it must be assigned to using its dummy name. For example, `Sub` accesses `A`, known locally as `D`, by argument association and is forbidden to assign a value to or modify the availability of `A`; it must be altered through its corresponding dummy argument (see P180 of Fortran 90 standard, [1]). The variable `A` can still be referenced in `Sub` as if it possessed the `INTENT(IN)` attribute (see Section 17.7 for the implications of the `INTENT` attribute).

A local variable called `A` could be declared in `Sub` which would clearly bear no relation to the `A` which is argument associated with `d`. When `Sub` is exited and control is returned to `Proggie`, the value that `C` had before the call to the subroutine is restored.

The `C` declared in `AnuvvaSub` bears no relation to the `C` from `Proggie` or the `C` from `Sub`.

Question 37: Local Variables

At each of the indicated points in the code, give the status (local, dummy argument, host associated or undefined) and, if appropriate, the values of the variables `v1, v2, v3, v4, r` and `i`.

```
PROGRAM PerOg
 IMPLICIT NONE
 REAL    :: V1,V2
 INTEGER :: V3,V4
  V1 = 1.0
  V2 = 2.0
  V3 = 3
  V4 = 4
   ...
```

```
!------ Position 1
    ...
   CALL Inte(V1,V3)
    ...
!------ Position 2
    ...
   CALL Exte(V1,V3)
    ...
!------ Position 3
    ...
 CONTAINS
  SUBROUTINE Inte(r,i)
   REAL, INTENT(INOUT) :: r
   INTEGER, INTENT(INOUT) :: i
   INTEGER :: v2 = 25
    ...
!------ Position 4
    ...
    r = 24.7
    i = 66
    v4 = 77
    ...
  END SUBROUTINE Inte
END PROGRAM PerOg
```

17.9 SAVE Attribute

The SAVE attribute can be:

- applied to a specified variable. In the following example, NumInvocations is initialised only on the **first** call (conceptually at program start-up) and then retains its new value between calls,

```
SUBROUTINE Barmy(arg1,arg2)
 INTEGER, SAVE :: NumInvocations = 0
  NumInvocations = NumInvocations + 1
```

- applied to the whole procedure by appearing on a line on its own. This means that *all* local objects are SAVEd.

```
SUBROUTINE polo(x,y)
 IMPLICIT NONE
 INTEGER :: mint, neck_jumper
 SAVE
 REAL :: stick, car
```

In the above example mint, neck_jumper, stick and car all have the SAVE attribute.

Variables with the SAVE attribute are known as *static* objects and have static storage class.

In fact, the SAVE attribute is given implicitly if an object, which is not a dummy argument or a PARAMETER, appears in an initialising declaration in a procedure, so

```
INTEGER, SAVE :: NumInvocations = 0
```

is equivalent to

```
INTEGER :: NumInvocations = 0
```

however, the former is clearer!

Clearly, the `SAVE` attribute has no meaning in the main program since when it is exited, the program has finished executing.

Objects appearing in `COMMON` blocks or `DATA` statements are automatically static.

Question 38: Save Attribute

Write a skeleton procedure that records how many times it has been called.

17.10 Keyword Arguments

Normal argument correspondence is performed by position; the first actual argument corresponds to the first dummy argument and so on. Fortran 90 includes a facility to allow actual arguments to be specified in any order. At the call site actual arguments may be prefixed by *keywords* (the *dummy* argument name followed by an equals sign) which are used when resolving the argument correspondence. If keywords are used then the usual positional correspondence of arguments is replaced by keyword correspondence, for example, consider the following interface description,

```
SUBROUTINE axis(x0,y0,l,min,max,i)
 REAL, INTENT(IN)   :: x0, y0, l, min, max
 INTEGER, INTENT(IN) :: i
END SUBROUTINE axis
```

`axis` can be invoked as follows,

- using positional argument invocation:

  ```
  CALL AXIS(0.0,0.0,100.0,0.1,1.0,10)
  ```

- using keyword arguments:

  ```
  CALL AXIS(0.0,0.0,Max=1.0,Min=0.1,L=100.0,I=10)
  ```

As soon as one argument is prefixed by a keyword then all subsequent arguments (going left to right) must also be prefixed.

Note: if an `EXTERNAL` procedure (see 28) is invoked with keyword arguments then the `INTERFACE` must be explicit at the call site.

In summary, keyword arguments:

- allow actual arguments to be specified in any order.

- □ help improve the readability of the program.
- □ (when used in conjunction with optional arguments) make it easy to add an extra argument without the need to modify each and every invocation in the calling code.
- □ are the dummy argument names.

Question 39: Keyword Arguments

Write a main program and an internal subroutine that returns, as its first argument, the sum of two real numbers. Invoke this using keyword arguments.

What are the important things to remember when using keyword arguments?

17.11 Optional Arguments

Dummy arguments with the optional attribute can be used to allow default values to be substituted in place of absent actual arguments. Any argument with the `OPTIONAL` attribute may be omitted from an actual argument list but as soon as one argument has been dropped, all subsequent arguments (going left to right) must be keyword arguments to allow the compiler to resolve the argument correspondence. Any use of optional arguments requires the interface of the procedure to be explicit; the compiler needs to know the ordering, type, rank and names of the dummy arguments to work out the correspondence. The status of an optional argument can be found using the `PRESENT` intrinsic function.

Many of the intrinsic procedures have optional arguments and keywords can be used for these procedures in exactly the same way as for user defined procedures.

17.11.1 Optional Arguments Example

Consider the following internal subroutine with two optional arguments,

```
SUBROUTINE SEE(a,b)
 IMPLICIT NONE
 REAL, INTENT(IN), OPTIONAL    :: a
 INTEGER, INTENT(IN), OPTIONAL :: b
 REAL    :: ay; INTEGER :: bee
 ay = 1.0; bee = 1
 IF(PRESENT(a)) ay  = a
 IF(PRESENT(b)) bee = b
  ...
```

Both `a` and `b` have the `OPTIONAL` attribute so the subroutine, `SEE`, can be called in the following ways.

```
CALL SEE()
CALL SEE(1.0,1); CALL SEE(b=1,a=1.0) ! same
CALL SEE(1.0);   CALL SEE(a=1.0)     ! same
CALL SEE(b=1)
```

In the above example of procedure calls, the first call uses both default values, the second and third use none and the remaining two both have one missing argument.

Within the procedure it must be possible to check whether OPTIONAL arguments are missing or not, the PRESENT intrinsic function, which delivers a scalar LOGICAL result, has been designed especially for this purpose.

If an optional argument is missing then it may not be assigned to, for example, the following is invalid:

```
IF (.NOT.PRESENT (up)) up = .TRUE.  ! WRONG!!!
```

Question 40: Erroneous Code

What is wrong with the following internal procedure?

```
SUBROUTINE Mistaken(A,B,C)
 REAL, INTENT(INOUT) :: A
 REAL, INTENT(OUT), OPTIONAL :: B
 REAL, INTENT(IN) :: C
 A = 2*C*B
 IF( PRESENT(B) ) B = 2*A
END
```

Question 41: Draw a Circle

Write an internal subroutine which draws circles. The routine takes two arguments, the first defines the radius of the circle and the second is optional and defines the colour by which the circle is shaded. The default colour for the circle shading should be green. Assume you have two library subroutines available, one called Circle which takes one REAL argument, (the radius,) and draws the circumference and one called Shade_Circle which shades it in and takes one character argument that defines the colour as a character string, e.g. "R" for red or "G" for green.

Module 7: More Procedures

18 Procedures and Array Arguments

There are three types of dummy array argument:

- *explicit-shape* — all bounds specified;

  ```
  INTEGER, DIMENSION(8,8), INTENT(IN) :: explicit_shape
  ```

  The actual argument that becomes associated with an explicit-shape dummy must conform in size and shape. An explicit INTERFACE is not required in this case. This method of declaring dummy arguments is very inflexible as only arrays of one fixed size can be passed. This form of declaration is only very occasionally appropriate.

- *assumed-size* — all bounds passed except the last;

  ```
  INTEGER, INTENT(IN) :: lda ! dummy arg
  INTEGER, DIMENSION(lda,*) :: assumed_size
  ```

  The last bound of an assumed-size array remains unspecified, (it contains a *,) and can adopt an assumed value. An explicit INTERFACE is not required. This was the FORTRAN 77 method of passing arrays but has been totally superseded by the next category.

- *assumed-shape* — no bounds specified;

  ```
  INTEGER, DIMENSION(:,:), INTENT(IN) :: assumed_shape
  ```

  All bounds can be inherited from the actual argument. The actual array that corresponds to the dummy must match in type, kind and rank. An explicit INTERFACE *must* be provided. This type of argument should always used in preference to *assumed-size* arrays.

Note: an actual argument can be an ALLOCATABLE array but a dummy argument cannot be — this means effectively that an ALLOCATABLE array must be allocated before being used as an actual argument.

18.1 Explicit-shape Arrays

A dummy argument that is an explicit-shape array must conform in size and shape to the associated actual argument; no bound information is passed to the procedure. Consider the following examples,

```
PROGRAM Main
 IMPLICIT NONE
  INTEGER, DIMENSION(8,8)   :: A1
```

```
      INTEGER, DIMENSION(64)    :: A2
      INTEGER, DIMENSION(16,32) :: A3
       ...
      CALL subby(A1)                 ! OK
      CALL subby(A2)     ! non conforming
      CALL subby(A3(::2,::4))        ! OK
      CALL subby(RESHAPE(A2,(/8,8/)) ! OK
       ...
    CONTAINS
     SUBROUTINE subby(explicit_shape)
      IMPLICIT NONE
      INTEGER, DIMENSION(8,8) :: explicit_shape
      ...
     END SUBROUTINE subby
    END PROGRAM Main
```

The bottom line is subby can "accept any argument as long as it is an 8×8 default INTEGER array"! This is clearly a very inflexible approach which generally should not be used.

18.2 Assumed-shape Arrays

Declaring dummy arrays as assumed-shape arrays is the recommended method in Fortran 90. Consider,

```
    PROGRAM TV
     IMPLICIT NONE
      ...
      REAL, DIMENSION(40)    :: X
      REAL, DIMENSION(40,40) :: Y
       ...
      CALL gimlet(X,Y)
      CALL gimlet(X(1:39:2),Y(2:4,4:4))
      CALL gimlet(X(1:39:2),Y(2:4,4)) ! invalid
      ...
    CONTAINS
     SUBROUTINE gimlet(a,b)
      REAL, INTENT(IN)   :: a(:), b(:,:)
      ...
     END SUBROUTINE gimlet
    END PROGRAM TV
```

An assumed-shape array declaration must have the same type, rank and kind as the actual argument. The compiler will insist on this.

Note:

- array sections can be passed so long as they are regular, that is, not defined by vector subscripts. The reason for this is concerned with efficiency. A vector subscripted section will be non-trivial to find in the memory, it is likely to be widely scattered and would probably need to be copied on entry to the procedure and then copied back on exit, this will create all sorts of runtime penalties.
- the actual argument cannot be an assumed-size array. If an actual argument were an assumed-size array then the bound / extent information of the last dimension would not be known meaning that the relevant information could not be passed on to a further procedure.

The third call is invalid because the second section reference has one dimensions whereas the declaration of the dummy has two.

18.3 Automatic Arrays

Other arrays can depend on dummy arguments, these are called *automatic* arrays and their size is determined by the values of dummy arguments. These arrays have local scope and a limited lifespan and, as they have their size determined by dummy arguments and are created and destroyed with each invocation of the procedure, cannot have the SAVE attribute (or be initialised). Arrays of this class are traditionally used for workspace.

Consider,

```
PROGRAM Main
 IMPLICIT NONE
  INTEGER :: IX, IY
   ....
   CALL une_bus_riot(IX,2,3)
   CALL une_bus_riot(IY,7,2)
CONTAINS
 SUBROUTINE une_bus_riot(A,M,N)
  INTEGER, INTENT(IN) :: M, N
  INTEGER, INTENT(INOUT) :: A(:,:)
  REAL  :: A1(M,N)             ! automatic
  REAL  :: A2(SIZE(A,1),SIZE(A,2)) ! ditto
  REAL  :: A3(A(1,1),A(1,1))   ! automatic
  ...
 END SUBROUTINE
END PROGRAM Main
```

The bound specifiers of an automatic array can originate from:

- ☐ scalar dummy arguments which are passed as arguments, for example, A1,
- ☐ components of other dummy arguments, for example, A3,
- ☐ characteristics of other dummy arguments (SIZE or LEN intrinsics), for example, A2 will be the same shape as A,
- ☐ a combination of the above.

There is currently no way to tell whether an automatic array has been created. Typically, if there is insufficient memory to allocate an automatic array, the program will ungracefully crash. If this is not desired then the less intuitive allocatable array should be used.

Question 42: Types of Arrays

In the following internal subroutine, which are the automatic and which are the assumed shape arrays?

```
SUBROUTINE Array_Types(A,B,C,D)
```

```
    INTEGER, INTENT(IN) :: D
     REAL, DIMENSION(:,:) :: A,C
     REAL, DIMENSION(:)   :: B
     REAL, DIMENSION(SIZE(A)) :: E
     INTEGER, DIMENSION(1:D,1:D) :: F
    ...
```

18.4 SAVE Attribute and Arrays

Consider,

```
SUBROUTINE sub1(dim)
 INTEGER, INTENT(IN)                        :: dim
 REAL, ALLOCATABLE, DIMENSION(:,:), SAVE :: X
 REAL, DIMENSION(dim)                       :: Y
  ...
  IF (.NOT.ALLOCATED(X)) ALLOCATE(X(20,20))
```

In the same way that a SAVEd variable persists between procedure calls, the array X will remain allocated (and its value preserved) between separate invocations.

Dummy arguments or objects which depend on dummy arguments (for example, automatic objects, see Section 18.3) cannot have the SAVE attribute. This is because the dummy arguments could be different on each invocation of the procedure. As Y depends on a dummy argument it cannot be given the SAVE attribute.

18.5 Explicit Length Character Dummy Arguments

This type of dummy argument declaration is comparable to an explicit-length array declaration in that an *explicit-length* actual CHARACTER array argument must match the corresponding dummy in kind as well as in rank. If the actual is a scalar default character variable then the length of the dummy must be less than or equal to the length of the actual. This approach is clearly inflexible and other methods should really be used, consider:

```
PROGRAM Main
 IMPLICIT NONE
  CHARACTER(LEN=10), DIMENSION(10) :: wurd
   ...
  CALL char_example(wurd(3))
  CALL char_example(wurd(6:))
CONTAINS
 SUBROUTINE char_example(wird,werds)
  CHARACTER(LEN=10), INTENT(INOUT) :: wird
  CHARACTER(LEN=10), INTENT(INOUT) :: werds(:)
   ...
 END SUBROUTINE char_example
END PROGRAM Main
```

The example demonstrates that assumed-shape arrays (or indeed, explicit-length arrays if desired) can still be used for character dummy array arguments.

18.6 Assumed Length Character Dummy Arguments

This is the recommended method of passing strings in Fortran 90.

CHARACTER dummy arguments can inherit the *type-param-value* (LEN=) specifier from the corresponding actual argument, however, the kind parameter and rank must still match. This form of argument declaration is much more flexible than using explicit-length strings and is analogous to using assumed-shape dummies in that the length information is 'inherited' from the actual argument. An argument with this form of inherited size is called an *assumed-length* character dummy:

```
PROGRAM Main
 IMPLICIT NONE
  CHARACTER(LEN=10) :: Vera
  CHARACTER(LEN=20) :: Hilda
  CHARACTER(LEN=30) :: Mavis
    ...
   CALL char_lady(Vera)
   CALL char_lady(Hilda)
   CALL char_lady(Mavis)
CONTAINS
 SUBROUTINE char_lady(word)
  CHARACTER(LEN=*), INTENT(IN) :: word
   ...
   PRINT*, "Length of arg is", LEN(word)
   ...
 END SUBROUTINE char_lady
END PROGRAM Main
```

The actual length of a dummy can be acquired by using the LEN inquiry intrinsic which is comparable to the SIZE array inquiry intrinsic.

18.7 Array-valued Functions

As well as returning 'conventional' scalar results, functions can return pointers, arrays or derived types. For array valued functions the size of the result can be determined in a fashion which is similar to the way that automatic arrays are declared.

Consider the following example of an array valued function:

```
PROGRAM proggie
 IMPLICIT NONE
  INTEGER, PARAMETER      :: m = 6
  INTEGER, DIMENSION(M,M) :: im1, im2
   ...
  IM2 = funnie(IM1,1) ! invoke
   ...
CONTAINS
 FUNCTION funnie(ima,scal)
  INTEGER, INTENT(IN) :: ima(:,:)
  INTEGER, INTENT(IN) :: scal
  INTEGER :: funnie(SIZE(ima,1),SIZE(ima,2))
  funnie(:,:) = ima(:,:)*scal
```

```
  END FUNCTION funnie
 END PROGRAM proggie
```

here the bounds of `funnie` are inherited from the actual argument and used to determine the size of the result array; a fixed sized result could have been returned by declaring the result to be an explicit-shape array but this approach is less flexible.

Question 43: Triangular Numbers — Array Valued Function

The numbers 1, 3, 6, 10, 15, 21, ... are called triangular numbers because the number of units in each can be displayed as a triangular pyramid of blobs. The i^{th}, p_i is computed from $p_i = p_{i-1} + i$. Write an array valued function which takes one argument (N) and returns a vector of the first N triangular numbers.

Make sure to specify the argument `INTENT`.

Write a test program to demonstrate the function and print out the sequence where $N = 23$.

Question 44: Vector Multiplication — Array Valued Function

Write a function `Outer` that forms the outer product of two vectors. If A and B are vectors then the outer product is a matrix C such that $C_{ij} = A_i \times B_j$.

Write a test program which accepts two integers giving the size of the `A` and `B` vectors, uses the `RANDOM_NUMBER` function to assign values and then prints the outer product.

18.8 Character-valued Functions

It is clearly useful to have a function that returns a `CHARACTER` string of a given length. The length of the result can either be fixed or can be inherited from one of the dummy arguments:

```
FUNCTION reverse(word)
 CHARACTER(LEN=*), INTENT(IN) :: word
 CHARACTER(LEN=LEN(word)) :: reverse
 INTEGER :: lw
  lw = LEN(word)
  ! reverse characters
  DO I = 1,lw
    reverse(lw-I+1:lw-I+1) = word(I:I)
  END DO
END FUNCTION reverse
```

In this case the length of the function result is determined automatically to be the same as that assumed by the dummy argument (`word`). (Automatic `CHARACTER` variables could also be created in this way.) Note that `word` cannot be used until after its declaration, it is for this reason that, in this case, the function type cannot appear as a prefix to the function header.

An explicit interface must always be provided if the function uses a `LEN=*` length specification.

18.9 Side Effect Functions

If, in the function invocation

```
rezzy = funky1(a,b,c) + funky2(a,b,c)
```

both funky1 and funky2 modify the value of a and a is used in the calculation of the result, then the order of execution would be important. Consider the following two internal functions:

```
INTEGER FUNCTION funky1(a,b,c)
 REAL, INIENT(INOUT) :: a
 REAL, INIENT(IN)    :: b,c
  a = a*a
  funky1 = a/b
END FUNCTION funky1
```

and

```
INTEGER FUNCTION funky2(a,b,c)
 REAL, INIENT(INOUT) :: a
 REAL, INIENT(IN)    :: b,c
  a = a*2
  funky2 = a/c
END FUNCTION funky2
```

Notice how both functions modify the value of a, this means that the value of rezzy is wholly dependent on the (undefined) order of execution.

With a=4, b=2 and c=4 the following happens:

- if funky1 executed first then rezzy=8+8=16
  - in funky1 a is initially equal to 4,
  - upon exit of funky1 a is equal to 16
  - upon exit funky1 is equal to $16/2 = 8$
  - in funky2 a is initially 16
  - upon exit of funky2 a is equal to 32
  - upon exit funky2 is equal to $32/4 = 8$
  - this means that rezzy is $8+8 = 16$
  - and a equals 32
- if funky2 executed first then rezzy=2+32=34
  - in funky2 a is initially 4
  - upon exit of funky2 a is equal to 8,
  - upon exit funky2 is equal to $8/4 = 2$
  - in funky1 a is initially equal to 8,

- ◇ upon exit of funky1 a is equal to 64,
- ◇ upon exit funky1 is equal to 64/2 = 32
- ◇ this means that rezzy is 2+32=34
- ◇ and a equals 64

A properly constructed function should be such that its result is uniquely determined by the values of its arguments, and the values of its arguments should be unchanged by any invocation as should any global entities of the program. Fortran 95 will introduce the PURE keyword which when applied to a procedure asserts that the routine is 'side-effect-free' in the sense that it does not change any values behind the scenes. (For example, a PURE function will not change any global data nor will it change the values of any of its arguments.) If at all possible all functions should be PURE as this will keep the code simpler.

18.10 Recursive Procedures

Recursion occurs when procedures call themselves (either directly or indirectly). Any procedural call chain with a circular component exhibits recursion. Even though recursion is a neat and succinct technique to express a wide variety of problems, if used incorrectly it may incur certain efficiency overheads.

In FORTRAN 77 recursion had to be simulated by a user defined stack and corresponding manipulation functions, in Fortran 90 it is supported as an explicit feature of the language. For matters of efficiency, recursive procedures (SUBROUTINEs and FUNCTIONs) must be explicitly declared using the RECURSIVE keyword. (See below.)

Declarations of recursive functions have a slightly different syntax to regular declarations, the RESULT keyword must be used with recursive functions and specifies a variable name where the result of the function can be stored. (The RESULT keyword is necessary since it is not possible to use the function name to return the result. Array valued recursive functions are allowed and sometimes a recursive function reference would be indistinguishable from an array reference. The function name implicitly has the same attributes as the result name.)

The fact that a function exhibits recursion must be declared in the header, valid declarations are:

```
INTEGER RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
```

or

```
RECURSIVE INTEGER FUNCTION fact(N) RESULT(N_Fact)
```

(In the above the INTEGER applieds to *both* fact and N_Fact.)

or,

```
RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
INTEGER N_Fact
```

In the last case INTEGER N_Fact implicitly gives a type to fact; it is actually illegal to also specify a type for fact.

Subroutines are declared using the RECURSIVE SUBROUTINE header.

18.10.1 Recursive Function Example

The following program calculates the factorial of a number, $n!$, and uses $n! = n(n-1)!$

```
PROGRAM Mayne
 IMPLICIT NONE
  PRINT*, fact(12) ! etc
CONTAINS
 RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
  INTEGER, INTENT(IN)  :: N
  INTEGER :: N_Fact ! also defines type of fact
   IF (N > 0) THEN
     N_Fact = N * fact(N-1)
   ELSE
     N_Fact = 1
   END IF
 END function FACT
END PROGRAM Mayne
```

The INTEGER keyword in the function header specifies the type for both fact *and* N_fact.

The recursive function repeatedly calls itself, on each call the values of the argument is reduced by one and the function called again. Recursion continues until the argument is zero, when this happens the recursion begins to unwind and the result is calculated.

$4!$ is evaluated as follows,

1. $4!$ is $4 \times 3!$, so calculate $3!$ then multiply by 4,
2. $3!$ is $3 \times 2!$, need to calculate $2!$,
3. $2!$ is $2 \times 1!$, $1!$ is $1 \times 0!$ and $0! = 1$
4. can now work back up the calculation and fill in the missing values.

18.10.2 Recursive Subroutine Example

Subroutines can also be recursive,

```
RECURSIVE SUBROUTINE Factorial(N, Result)
 INTEGER, INTENT(IN)    :: N
 INTEGER, INTENT(INOUT) :: Result
  IF (N > 0) THEN
   CALL Factorial(N-1,Result)
   Result = Result * N
  ELSE
   Result = 1
  END IF
END SUBROUTINE Factorial
```

Question 45: Maggot/Onion Recursive Procedure Conundrum

An onion with an unknown number of layers contains a maggot at an unknown layer. Write a recursive function which will determine the depth of the maggot using the probability of 0.1 of a maggot being at a particular layer and a random number generator.

19 Object Orientation

19.1 Stack Simulation Example

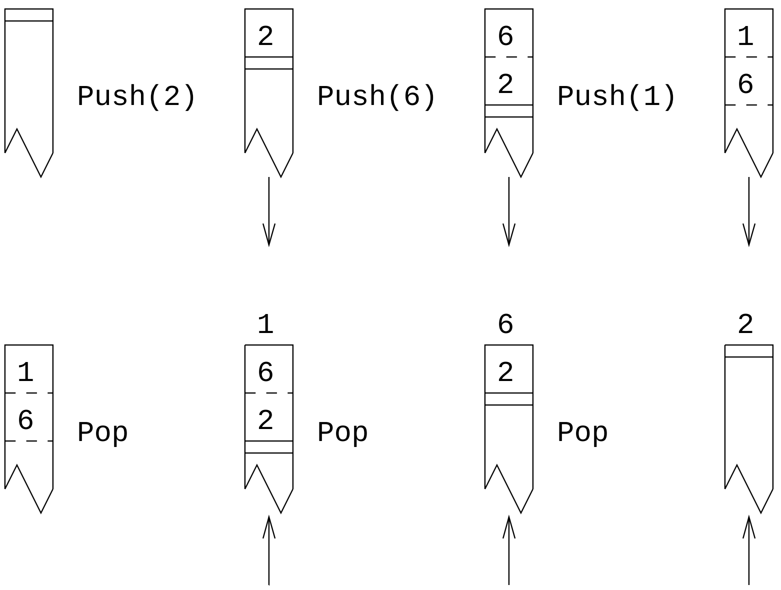


Figure 23: How a Stack Works

A stack can be thought of as a pile of things that can only be accessed from the top. You can put something on the top of the pile: a `Push` action, and you can grab things off the top of the pile: a `Pop` action.

The diagrams show the conventional visualisation of a stack. As elements are *pushed* onto the top of the stack, the previous top elements get pushed down, as things are *popped* off the top the lower stack entries move towards the top.

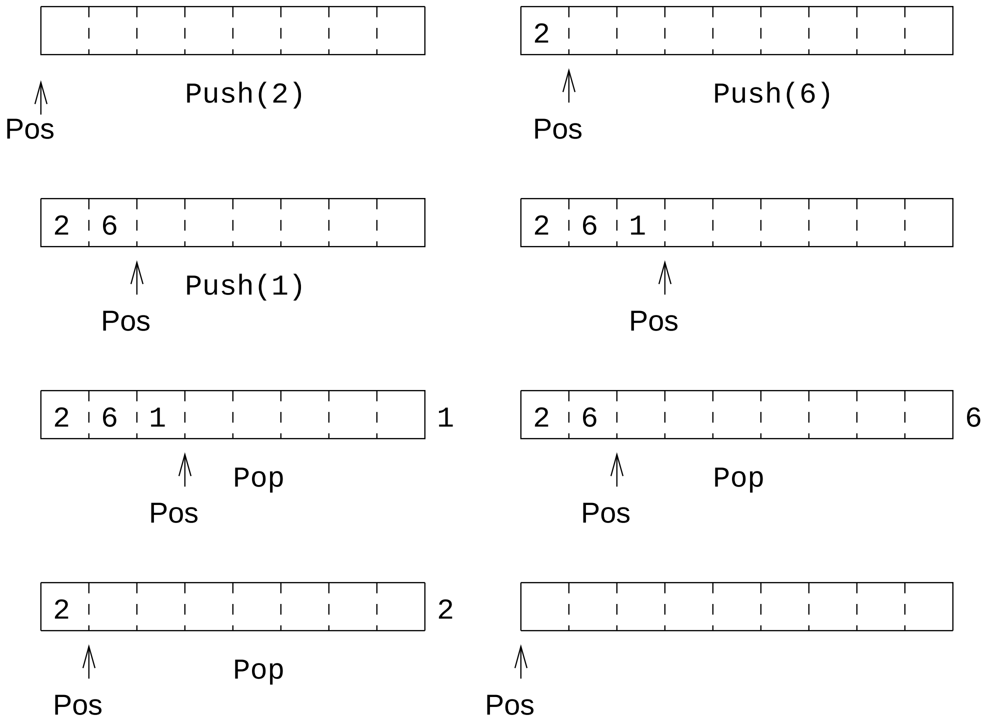


Figure 24: How a Stack Could be Implemented

These diagrams show how we will simulate the stack using an array with a pointer to indicate where the top of the stack is. As elements are pushed onto the stack they are stored in the array and the 'top' position moves right; as things are popped off, the top of the stack moves left. From this description it can be seen that we need three objects to simulate a stack:

- a current position marker,
- the array to hold the data,
- the maximum size of the stack (array).

This data is needed by both the `Push` and `Pop` procedures, and should be made global.

19.1.1 Stack Example Program

For example, the following defines a very simple 100 element integer stack,

```
    PROGRAM stack
     IMPLICIT NONE
     INTEGER, PARAMETER :: stack_size = 100
     INTEGER, SAVE :: store(stack_size), pos = 0
      ....
! stuff that uses stack
```

```
  ....
 CONTAINS
  SUBROUTINE push(i)
   INTEGER, INTENT(IN) :: i
    IF (pos < stack_size) THEN
     pos = pos + 1; store(pos) = i
    ELSE
     STOP 'Stack Full error'
    END IF
  END SUBROUTINE push
  SUBROUTINE pop(i)
   INTEGER, INTENT(OUT) :: i
    IF (pos > 0) THEN
     i = store(pos); pos = pos - 1
    ELSE
     STOP 'Stack Empty error'
    END IF
   END SUBROUTINE pop
 END PROGRAM stack
```

store and pos do not really need the SAVE attribute as here they are declared in the main program and do not go out of scope. The declaration of stack_size as a PARAMETER allows the stack size to be easily modified; it does not need the SAVE attribute (because it is a constant).

The main program can now call push and pop which simulate adding to and removing from a 100 element INTEGER stack. The current state of the stack, that is, the number of elements on the sack and the values of each location, are stored as global data.

Actually this is **not** the ultimate way of simulating a stack but demonstrates global data. (See later for an improvement.)

19.2 Reusability — Modules

A stack is a useful data structure which can be applied to many different situations. In its current state, if we wished to use the stack with a different program then it would be necessary to retype or cut and paste the existing text into this other program. There is a far better way to enable the code to be used elsewhere and that is to convert the existing PROGRAM to a MODULE. This technique is called encapsulation:

```
 MODULE Stack
  IMPLICIT NONE
  INTEGER, PARAMETER :: stack_size = 100
  INTEGER, SAVE :: store(stack_size), pos = 0
 CONTAINS
  SUBROUTINE push(i)
   .....
  END SUBROUTINE push
  SUBROUTINE pop(i)
   ......
  END SUBROUTINE pop
 END MODULE Stack
```

In the above module, the data structures and access functions that relate to the stack have been encapsulated in a MODULE. It is now possible for any other program units that need to utilise a stack to attach this module (with a USE statement) and access its functionality.

```
PROGRAM StackUser
 USE Stack          ! attaches module
 IMPLICIT NONE
  ....
 CALL Push(14); CALL Push(21);
 CALL Pop(i); CALL Pop(j)
  ....
END PROGRAM StackUser
```

All entities declared in Stack, for example push and pop, are now available to StackUser. This form of software reusability is a cornerstone of object oriented programming; in this case the module is like a (C++ or Java) class as it contains the definition of an object (the stack) and all the access functions used to manipulate it (push and pop).

A point of note is that the SAVE statement, in the module, behaves as usual. The variables store and pos are given the SAVE attribute because objects in a module only exist when the module is being used. In the StackUser program, the module is in use for the lifetime of the program — it has the scope of the program. However, if the module was not used in a main program but was instead attached to a procedure, then when that procedure is exited and control passed back to the main program (which does not contain a USE statement,) the module will go out of scope and any non-static objects (non-SAVEd objects) will disappear from memory. The next time the module is brought back into scope, a new instance of these variables will be created, however, any values held by non-static objects will be lost. This is the same principle as when the SAVE attribute is applied to objects declared in a procedure.

Within a module, functions and subroutines are called *module procedures* and follow slightly different rules from regular procedures. The main difference is that they may contain internal procedures (in the same way as PROGRAMs),

All modules must be compiled before any program unit that uses the module.

Modules can also be used to replace the functionality of a COMMON block. COMMON blocks have long been regarded as an unsafe feature of Fortran. In FORTRAN 77 they were the only way to achieve global storage. In Fortran 90 global storage can be achieved by:

- declaring the required global objects within a module.
- giving these objects the SAVE attribute,
- attaching the module (via USE) to each program unit which requires access to the global data. (Wherever the MODULE is used, the global objects are visible.)

For example, to declare X, Y and Z as global, declared them in a module:

```
MODULE  Globby
 REAL, SAVE :: X, Y, Z
END MODULE Globby
```

and use the module

```
PROGRAM YewZing
 USE Globby
 IMPLICIT NONE
  ...
END PROGRAM YewZing
```

If the use of a COMMON block is absolutely essential then a single instance of it should be placed in a MODULE and then used. (A COMMON block names a specific area of memory and splits it up into areas that can be referred to as scalar or array objects, for a single COMMON block it is possible to specify a different partition of data objects each time the block is accessed, each partition can be referred to by any type, in other words the storage can be automatically retyped and repartitioned with gay abandon — this leads to all sorts of insecurities. Putting the block in a MODULE removes any possibility of the above type of misuse.) This will have the effect of ensuring that each time the COMMON block is accessed it will be laid out in **exactly** the same way and mimics the techniques of placing COMMON blocks in INCLUDE files (a common FORTRAN 77 technique).

Question 46: Encapsulation

Define a module called Simple_Stats which contains encapsulated functions for calculating the mean and standard deviation of an arbitrary length REAL vector. The functions should have the following interfaces:

```
REAL FUNCTION mean(vec)
 REAL, INTENT(IN), DIMENSION(:) :: vec
END FUNCTION mean


REAL FUNCTION Std_Dev(vec)
 REAL, INTENT(IN), DIMENSION(:) :: vec
END FUNCTION Std_Dev
```

[Hint: In Fortran 90, SIZE(X) gives the number of elements in the array X.]

You may like to utilise your earlier code as a basis for this exercise.

Add some more code in the module which records how many times each statistical function is called during the lifetime of a program. Record these numbers in the variables: mean_use and std_dev_use.

Demonstrate the use of this module in a test program; in one execution of the program give the mean and standard deviation of the following sequences of numbers:

```
5.0 3.0 17.0 -7.56 78.1 99.99 0.8 11.7 33.8 29.6
```

and

```
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0
```

plus two other sequences.

Print out the values of mean_use and std_dev_use for this run of the program.

Question 47: Complex Arithmetic — Modules

A complex number with solely integer components can be represented as a two element `INTEGER` array. Write a `MODULE` called `Integer_Complex_Arithmetic` which contains 4 `FUNCTION`s each accepting two integer complex number 'operands' and delivering the result of addition, subtraction, multiplication and division. Where appropriate Fortran 90 integer division rules should be followed. The following rules for complex arithmetic may be useful,

$$(a + b\mathbf{i}) + (x + y\mathbf{i}) = (a + x) + (b + y)\mathbf{i}$$

and,

$$(a + b\mathbf{i}) - (x + y\mathbf{i}) = (a - x) + (b - y)\mathbf{i}$$

and,

$$(a + b\mathbf{i}) \times (x + y\mathbf{i}) = (a \times x - b \times y) + (a \times y + b \times x)\mathbf{i}$$

and,

$$\frac{(a + b\mathbf{i})}{(x + y\mathbf{i})} = \frac{(a + b\mathbf{i}) \times (x - y\mathbf{i})}{x^2 + y^2}$$

Also write a procedure to accept one integer complex number and one non-negative integral exponent which delivers the result of `**`.

Note, for $n \geq 0$,

$$(a + b\mathbf{i})^n = 1 \times \underbrace{(a + b\mathbf{i}) \times \cdots \times (a + b\mathbf{i})}_{n}$$

19.3 Restricting Visibility

Data hiding has been introduced to Fortran 90 as a safety feature, it prevents anybody using a module from changing certain module objects such as those used for 'housekeeping'. Restricting the visibility of purely internal details also allows the module to be updated / modified without the user being aware.

For example, the following lines could be added to the stack example,

```
PRIVATE :: pos, store, stack_size  ! hidden
PUBLIC  :: pop, push   ! not hidden
```

This would ensure that `store`, `stack_size` and `pos` are hidden from the user whereas, `pop` and `push` are not. This makes the module **much** safer as the user is unable to 'accidentally' modify the value of `pos` (the index of the top of the stack) or modify the contents of `store` (the data structure holding the contents of the stack).

The default accessibility for an arbitrary module is `PUBLIC` which means that if accessibility is not specificed a user will have access to all entities. This can be reversed by using a `PRIVATE` statement directly after the `IMPLICIT NONE` statement.

```
MODULE Blimp
 IMPLICIT NONE
  PRIVATE
  ! unlessed specified otherwise all
  ! other entities are PRIVATE
   .....
END MODULE Blimp
```

There is also an attributed form of the visibility specifiers.

In our example we can equivalently use statements and attributes;

```
PUBLIC                 ! set default visibility
INTEGER, PRIVATE, SAVE :: store(stack_size), pos
INTEGER, PRIVATE, PARAMETER :: stack_size = 100
```

Which means in the main PROGRAM:

```
CALL Push(21); CALL Pop(i)
```

is perfectly OK but

```
pos = 22; store(pos) = 99
```

would be flagged by the compiler as an error.

Access to the stack is now only possible via the access functions pop and push.

Question 48: Visibility Attributes

Add visibility attributes to your Simple_Stats module so that the usage statistics variables (mean_use and std_dev_use) cannot be accessed directly but must be viewed by calling either of the access subroutines Print_Mean_Use or Print_Std_Dev_Use. In other words,

```
CALL Print_Mean_Use
CALL Print_Std_Dev_Use
```

should print out the values of mean_use and std_dev_use.

What advantage does this approach hold?

19.4 The USE Renames Facility

The USE statement names a module whose public definitions are to be made accessible. It has the following form:

USE < *module-name* > [,< *new-name* > => < *use-name* >...]

If a local entity has the same name as an object in the module, then the module object can be renamed, (as many renames as desired can be made in one USE statement, they are just supplied as a comma separated list),

```
USE Stack, IntegerPop => Pop
```

Here the module object Pop is renamed to IntegerPop when used locally. This renaming facility is essential otherwise user programs would often require rewriting owing to name clashes, in this way the renaming can be done by the compiler. Renaming should not be used unless absolutely necessary as it can add a certain amount of confusion to the program. It is only permissible to rename an object once in a particular scoping unit.

19.5 USE ONLY Statement

It is possible to restrict the availability of objects declared in a module made visible by use association by using the ONLY clause of the USE statement. In this way name clashes can be avoided by only using those objects which are necessary. It has the following form:

USE < *module-name* > [ONLY:< *only*-list >...]

The < *only*-list > can also contain renames (=>).

This method can be used as an alternative to renaming module entities, for example, a situation may arise where a user wishes to use 1 or 2 objects from a module but discovers that if the module is accessed by use association there are a couple of hundred name clashes. The simplest solution to this is to only allow the 1 or 2 objects to be seen by his / her program. These 1 or 2 objects can be renamed if desired.

For example,

```
USE Stack, ONLY:Pos, IntegerPop => Pop
```

Only Pos and Pop are made accessible and the latter is renamed.

The USE statement, with its renaming and restrictive access facilities, gives greater user control enabling two modules with similar names to be accessed by the same program with only minor inconvenience. In addition, the ONLY clause gives the compiler the option of just including the object code associated with the entities specified instead of the code for the whole module. This has the potential to make the executable code smaller and faster.

Question 49: USE Renames Statement

Write a USE statement that accesses the Simple_Stats module. This USE statement should specify that only the procedures Mean and Print_Mean_Use are attached and that they are renamed to be Average and Num_Times_Average_Called!

20 Modules

Modules are new to Fortran 90. They have a very wide range of applications and their use should be encouraged. They allow the user to write object based code which is generally accepted to be more reliable, reusable and easier the write than regular code.

A MODULE is a program unit whose functionality can be exploited by any other program unit which attaches it (via the USE statement). A module can contain the following:

- global object declarations;

  Modules should be used in place of FORTRAN 77 COMMON and INCLUDE statements. If global data is required, for example, to cut down on argument passing, then objects declared in a module can be made visible wherever desired by simply attaching the module. Objects in a module can be given a static storage class so will retain their values between uses.

  Sets of variable declarations which are not globally visible can also be placed in a module and used at various places in the code; this mimics the functionality of the INCLUDE statement. (INCLUDE was an extension to FORTRAN 77 and literally included a specified file at the appropriate place in the code.)

- interface declarations;

  It is sometimes advantageous to package INTERFACE definitions into a module and then use the module whenever an explicit interface is needed. This should be done in conjunction with the ONLY clause but is only really useful when module procedures cannot be used.

- procedure declarations;

  Procedures can be encapsulated into a module which will make them visible to any program unit which uses the module. This approach has the advantage that all the module procedures, and hence their interfaces, are explicit within the module so there will be no need for any INTERFACE blocks to be written.

- controlled object accessibility;

  Variables, procedures and operator declarations can have their visibility controlled by access statements within the module. It is possible for specified objects to be visible only inside the module. This technique is often used to provide security against a user meddling with the values of module objects which are purely internal to the module.

- packages of whole sets of facilities;

  Derived types, operators, procedures and generic interfaces can be defined and packaged to provide simple object oriented capabilities.

- semantic extension;

  A semantic extension module is a collection of user-defined type declarations, access routines and overloaded operators and intrinsics which, when attached to a program unit, allow the user to use the functionality as if it were part of the language.

20.1 Modules — General Form

The syntax of a module program unit is:

```
MODULE < module name >
        < declarations and specifications statements >
 [ CONTAINS
        < definitions of module procedures > ]
END [ MODULE [ < module name > ] ]
```

< *module name* > is the name that appears in the USE statement, it is *not* necessarily the same as the filename.

```
MODULE Nodule
  ! TYPE Definitions
  ! Global data
   ! ..
  ! etc ..
 CONTAINS
    SUBROUTINE Sub(..)
      ! Executable stmts
    CONTAINS
        SUBROUTINE Int1(..)

        END SUBROUTINE Int1

          ! etc.
        SUBROUTINE Intn(..)

        END SUBROUTINE Int2n
    END SUBROUTINE Sub
      ! etc.
    FUNCTION Funky(..)
     ! Executable stmts
    CONTAINS

          ! etc

    END FUNCTION Funky
END MODULE Nodule
```

Figure 25: Schematic Diagram of a Module Program Unit

Entities of other modules can be accessed by a USE statement as the first statement in the specification part. This is called *use-association*.

Non-circular dependent definition chains are allowed (one module USEs another which USEs another and so on) providing a partial-inheritance mechanism.

A module may include the following declarations and specifications:

- USE statements (a module inherit the environment of another module either fully or partially by using USE .. ONLY clause,)
- object definitions, (any declarations and definitions that could appear in a main program can also be found here),
- object initialisations,
- type definitions,
- accessibility statements,
- interface declarations (of external procedures);
- MODULE PROCEDURE declarations (procedures that appear after the CONTAINS specifier). Module procedures are written in exactly the same way as regular (external) procedures and may also contain internal procedures. They can be given visibility attributes so may be private to the module or usable by any program unit that employs the module.

Module 8: Pointers and Derived Types

21 Pointers and Targets

It is often useful to have variables where the space referenced by the variable can be changed as well as the values stored in that space. This is the functionality that a *pointer* variable provides. The space to which a pointer variable points is called its *target*; pointer variables do not hold data, they point to scalar or array variables which themselves may contain data. Pointers are used in much the same way as non-pointer variables with added functionality and a small number of extra constraints. In many situations the pointer has the potential to use less space than the target.

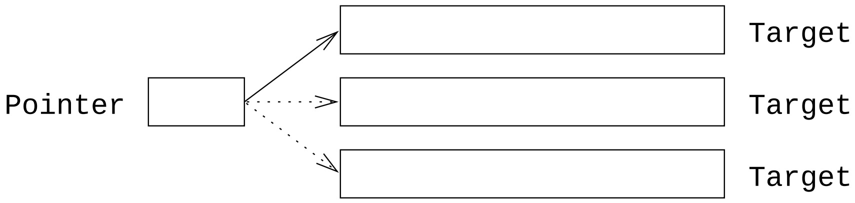


Figure 26: The Relationship Between a Pointer and its Target

Pointers are very useful as they can be used to access variables indirectly (aliasing) and provide a more flexible alternative to ALLOCATABLE arrays. Pointers may also be used in conjunction with user-defined types to create dynamic data structures such as linked lists and trees. These structures are useful, for example, in situations where data is being created and destroyed dynamically and it cannot be predicted in advance how this will occur.

Fortran 90 pointers are unlike C pointers as they are much less flexible and more highly optimised. The need for efficiency has implications in the way that they are used, declared and what they can point to. Pointers are strongly typed in the sense that a pointer to a REAL scalar target may not point to any other data type (INTEGER, LOGICAL etc) nor may it point to an array. A pointer to a one dimensional array may not point to an array of a different dimensionality (unless the array has been sectioned down to the required rank). In order to further enhance optimisation, anything that is to be pointed at must have the TARGET attribute (either explicitly or implicitly). In its lifetime a pointer can be made to point at various different targets, however, they all must be correctly attributed and be of the correct type, kind and rank. At any given time a given pointer may only have one target but a given target may be pointed to by more than one pointer.

The use of pointers may have a detrimental effect on the performance of code but can be used to actually enhance efficiency. For example, when sorting a set of long database records on one field, it is much more efficient to declare an array of pointers (one for each database entry) to define the sorted set than to physically move (copy) large database records. Pointer variables are typically small (probably smaller than the storage required for one default integer variable) so can be used to conserve memory. Moving arrays about in memory will involve a large and costly copy operation whereas retargeting a couple of pointers is a far less expensive solution.

In general, a reference to the pointer will be a reference to the target. Pointers are automatically dereferenced, in other words, when the pointer is referenced it is actually the value of the space that the pointer references that is used.

21.1 Pointer Status

Pointer variables have 3 states:

- *undefined* — the initial status of a pointer.
- *associated* — the pointer has a target.
- *disassociated* — the pointer has no target but is defined.

Visualisation,

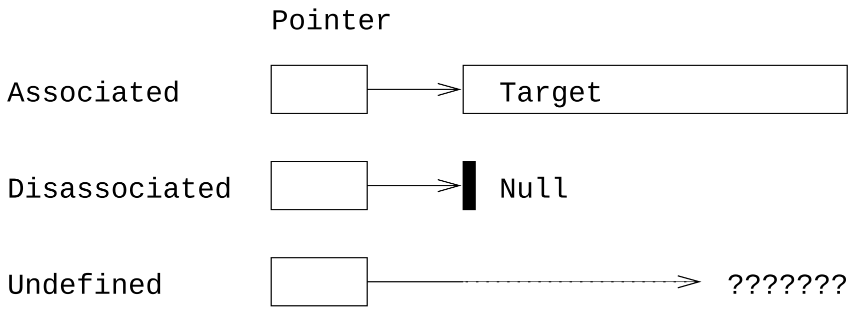


Figure 27: The Three States of a Pointer

There is a scalar `LOGICAL` intrinsic function, `ASSOCIATED`, which returns the (association) status of a pointer. The function returns `.TRUE.` if the pointer is associated and `.FALSE.` if it is disassociated; the result is undefined if the pointer status is itself undefined.

As good practise one should *always* use the `NULLIFY` statement to disassociate pointers before any use. Since an unassociated pointer has an undefined status it should not be referenced at all, not even in an `ASSOCIATED` function call. Fortran 95 will allow pointers to be nullified on declaration.

21.2 Pointer Declaration

A `POINTER` is a variable declared with the `POINTER` attribute, it has static type, kind and rank determined by its declaration:

```
REAL, POINTER :: Ptor
REAL, DIMENSION(:,:), POINTER :: Ptoa
```

The first declaration specifies that the name `Ptor` is a pointer to a scalar `REAL` target; the second specifies that `Ptoa` is a pointer to a rank 2 array of reals. Pointers cannot point to literals.

The above raises three interesting points:

- the declaration fixes the type, kind and rank of any possible target.
- pointers to arrays are always declared with deferred-shape array specifications.
- the rank of a target is fixed but the shape may vary.

A variable with the `POINTER` attribute may not possess the `ALLOCATABLE` or `PARAMETER` attributes.

21.3 Target Declaration

If ordinary variables are to become targets of a pointer they must be declared as such and be given the `TARGET` attribute.

```
REAL, TARGET :: x, y
REAL, DIMENSION(5,3), TARGET :: a, b
REAL, DIMENSION(3,5), TARGET :: c
```

With these declarations (and those from Section 21.2):

- `x` or `y` may become associated with `Ptor`;
- `a, b` or `c` may become associated with `Ptoa`.

As Fortran has always been associated with efficiency it was felt that pointers should be implemented as efficiently as possible and to this end the `TARGET` attribute was included. The Fortran 90 standard, [1], says:

> *"The TARGET attribute ... is defined solely [primarily] for optimization purposes. It allows the processor to assume that any nonpointer object not explicitly declared as a target may be referred to only by way of its original declared name."* Fortran 90 Standard.

The above text is to be changed as indicated in Fortran 95 as the word *solely* is felt to be misleading. The `TARGET` attribute is also used at procedure boundaries to say that one of the arguments is a `TARGET` which clearly means that the `TARGET` attribute has uses other than for optimisation.

21.4 Pointer Manipulation

Pointers have two separate forms of assignment which should not be confused as they have very different results.

- =>, *pointer assignment*
  - is a form of *aliasing* (referring to an object by a different name), the pointer and its target refer to the same space,
  - can take place between a pointer variable and a target variable, or between two pointer variables,
- =, *'normal' assignment*

◇ can take place between a pointer variable and an object of the appropriate type, kind and rank, the object on the RHS does *not* need to have the TARGET attribute. (The TARGET attribute is not needed because the LHS of the assignment is simply an alias for a 'regular' object and the RHS is not being pointed at.) When an alias is used on the LHS of an assignment statement, the reference can be thought of as being a reference to the aliased object.

◇ sets the space pointed at to the specified value. If a particular space is pointed at by a number of pointers then changing the values in the space obviously changes the dereferenced value of all associated pointers.

In summary, pointer assignment makes the pointer and the variable reference the same space whilst normal assignment alters the value contained in that space.

21.5 Pointer Assignment

Consider,

```
Ptor => y
Ptoa => b
```

The first statement associates a pointer Ptor with a target y, (Ptor is made an alias for y,) and the second, associates the pointer Ptoa with b, (Ptoa is an alias for b). From now on now Ptor and y, and Ptoa and b can be used interchangeably in statements since they both reference the same entity. If the value of y or b is changed, the value of Ptor or Ptoa also changes but the location of space pointed at does not.

If Ptor2 is also a pointer to a scalar real then the pointer assignment,

```
Ptor2 => Ptor
```

makes Ptor2 also point to y meaning that y has two aliases. The above statement has exactly the same effect as:

```
Ptor2 => y
```

21.5.1 Pointer Assignment Example

Consider,

```
x = 3.14159
Ptor => y
Ptor = x  ! y = x
```

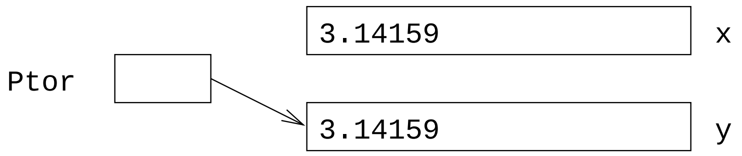


Figure 28: Visualisation of Pointer Assignment

The three statements have the following effect,

1. `x = 3.14159` is a regular assignment, `x` receives a value.
2. `Ptor => y` alias's `y` to be `Ptor`, `y` does not necessarily have a meaningful value.
3. `Ptor = x` sets the space pointed at by `Ptor`, (i.e., `y`) to the value of `x` (`3.14159`), (any assignments / references to `Ptor` are really assignments / references to `y`).

At the end of the above sequence of statements `x` and `Ptor` have the same value, this is because `Ptor` is an alias for `y` so the last statement effectively sets `y = 3.14159`. If the value of `x` is subsequently changed, the value of `Ptor` and `y` do not — `x` is not being pointed at by anything so changing its value has no effect. However,

```
Ptor = 5.0
```

means that, because of aliasing, `y` would also take the value `5.0`

21.6 Association with Arrays

An array pointer may be associated with the whole of a target or a *regular section* of a target as long as the section has the correct rank (and type and kind). (A regular section is an array section that can be defined by a linear function; a *subscript-triplet* defines a regular section.)

An array pointer cannot be associated with a non-regular (vector subscripted) array section.

Assuming the same declarations as before:

- this is valid:

  ```
  Ptoa => a(3:5,::2)
  ```

  This pointer assignment statement aliases the specified section to the name `Ptoa`, for example, `Ptoa(1,1)` refers to element `a(3,1)`; `Ptoa(2,2)` refers to to element `a(5,3)` and so on. `SIZE(Ptoa)` gives 6 and `SHAPE(Ptoa)` gives `(/ 2, 2 /)`.

- this is valid:

  ```
  Ptoa => a(1:1,2:2)
  ```

This means alias Ptoa to a 1×1 2D array. Note that the section specifiers uphold the rank even though they only specify one element.

- these are invalid — the targets have the wrong rank:

```
Ptoa => a(1:1,2)
Ptoa => a(1,2)
Ptoa => a(1,2:2)
```

  The ranks of the targets must be 2.

- this is invalid as the target is not a regular section but is defined by a vector subscript:

```
v = (/2,3,1,2/)
Ptoa => a(v,v)
```

  Even if the vector subscript v did define a regular section, the statement would still be invalid as the sectioning is specified by a vector subscript.

For example,

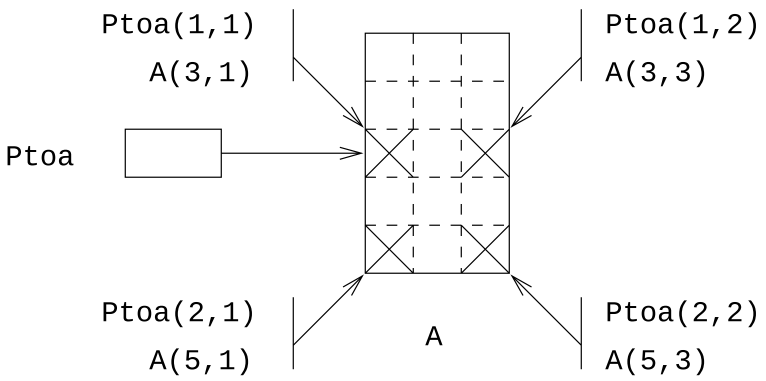


Figure 29: A Pointer to a Section of an Array

```
Ptoa => a(3::2,::2)
```

Here the top left subsection element can be referred to through TWO names

- A(3,1)
- Ptoa(1,1)

21.7 Dynamic Targets

Targets for pointers can also be created dynamically by allocation. As well as allotting space for allocatable arrays, the ALLOCATE statement can reserve space to be the target of a pointer, in this case the pointer is not so much an alias of another variable, it is more a reference to an unnamed part of the heap storage.

```
ALLOCATE(Ptor,STAT=ierr)
ALLOCATE(Ptoa(n*n,2*k-1), STAT=ierr)
```

The above statements allocate new space, the first for a single real and the second for a rank 2 real array. These objects are automatically made the targets of Ptor and Ptoa respectively.

In ALLOCATE statements, the status specifier STAT= should always be used, recall that a zero value means that the allocate request was successful and a positive value means that there was an error and the allocation did not take place. There should only be one object per ALLOCATE statement so if the allocation goes wrong it is easy to tell which object is causing the problem. In a procedure, any allocated space should be deallocated before the procedure is exited otherwise the space will become inaccessible and will be wasted. Allocated space that is to be retained between procedure calls should have the SAVE attribute.

It is not an error to allocate an array pointer that is already associated. If an array pointer is already allocated and is then allocated again, links with the first target are automatically broken and a new target installed in its place. Care must be taken to ensure that the original target is still accessible by another pointer otherwise its storage will be lost for the duration of the program.

21.8 Automatic Pointer Attributing

All pointer variables implicitly have the TARGET attribute. This means that any pointer can be associated with any other pointer as shown below:

```
Ptoa => A(3::2,1::2)
Ptor => Ptoa(2,1)
```

This would associate the element (2,1) of the target of Ptoa with Ptor.

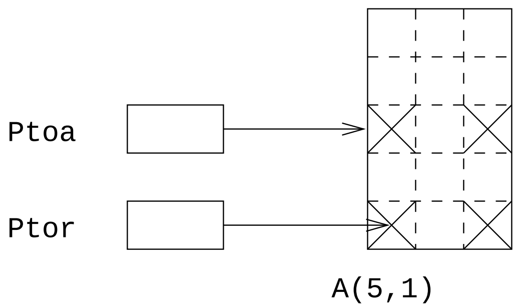


Figure 30: Automatic Attributing of Arrays

Here the bottom left subsection element can be referred to through THREE names:

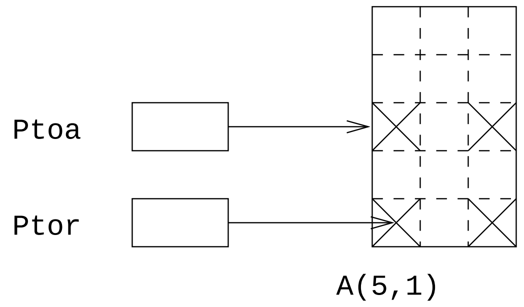


Figure 31: Automatic Attribution of a Pointer

- `Ptor`
- `A(5,1)`
- `Ptoa(2,1)`

This form of aliasing has advantages in both convenience of expression and efficiency. For example, if a particular element was being used frequently within a nested loop, the calculation of the index of particular element need only be done once. Subsequent references to the element can then be made through the scalar pointer.

Question 50: Pointers and Assignment

Given the following declarations,

```
IMPLICIT NONE
REAL, POINTER :: a, b(:), c(:,:)
REAL, TARGET  :: e, f(100), g(100,100), h(100,100,1)
REAL          :: k(100,100)
INTEGER, POINTER :: o(:,:)
INTEGER, TARGET  :: r(100), s(100,100)
INTEGER          :: v(100)
```

7 of the lines below are incorrect. Which are they and why are they wrong?

```
a => h(1,1,:)
b => h(20,:,1:1)
o => s
c => g(100,100)
a => k(20,20)
a => e
b => v
c => g; b => c(32,40:42)
e => f(20)
a => f(5)
c => g(100:100,100:100)
b => REAL(r)
```

21.9 Association Status

The status of a defined pointer may be tested by a scalar `LOGICAL` intrinsic function:

```
ASSOCIATED(Ptoa)
```

If `Ptoa` is defined and associated then this will return `.TRUE.`; if it is defined and disassociated it will return `.FALSE.`. If it is undefined the result is also undefined — the Edinburgh and NAg compilers give opposite results if `ASSOCIATED` is given an undefined pointer argument — neither give an error.

The target of a defined pointer may also be tested:

```
ASSOCIATED(Ptoa, arr)
```

If `Ptoa` is defined and currently associated with the specific target, `arr`, then the function will return `.TRUE.`, otherwise if it will return `.FALSE.`.

The `ASSOCIATED` intrinsic is also useful to avoid deallocation errors:

```
IF(ASSOCIATED(Ptoa)) DEALLOCATE(Ptoa, STAT=ierr)
```

21.10 Pointer Disassociation

Pointers can be disassociated with their targets by:

- nullification

  ```
  NULLIFY(Ptor)
  ```

  The `NULLIFY` statement breaks the connection of its pointer argument with its target (if any) and leaves the pointer in a disassociated state, however, it **does not** deallocate the targets storage. This space will be inaccessible until the program terminates unless it is pointed to by at least one more pointer. A list of pointers to nullify may be specified.

  Immediately after their appearance in a declaration statement, non-dummy argument pointers are in an undefined state; as a matter of course they should be nullified to bring them into a disassociated state. Using an undefined pointer as an argument to the `ASSOCIATED` statement is an error because an undefined pointer cannot be referenced *anywhere*!

- deallocation

  ```
  DEALLOCATE(Ptoa, STAT=ierr)
  ```

  The `DEALLOCATE` statement breaks the connection between the pointer and its target and returns the space to the heap storage for re-use. The statement should be used when the target space is finished with, for example, at the end of a procedure. The target is left in a disassociated (or nullified) state.

  The `STAT=` field should always be used to report on the success / failure of the deallocation request; as usual, zero indicates success and a positive value means failure. It is possible to deallocate `ALLOCATABLE` arrays and `POINTER` variables in the same statement, however, it is recommended that there only be one object per `DEALLOCATE` statement in case there is an error. It is an error to deallocate anything that has not been first allocated or that is in a disassociated or undefined state and it is also an error to deallocate an object that has more than one pointer associated with it; any other pointers should first be nullified before deallocation is performed.

21.11 Pointers to Arrays vs. Allocatable Arrays

In the general case ALLOCATABLE arrays should be used in preference to POINTER arrays in order to facilitate optimisation. Arrays which are the target of pointers may be referred to by a number of different names (alias's) whereas ALLOCATABLE arrays may only have one name. Clearly, if aliasing is desired then POINTERs and TARGETs must be used.

There are two main restrictions imposed upon allocatable arrays which do not apply to POINTER arrays:

- unallocated ALLOCATABLE arrays cannot be passed as actual arguments to procedures,
- ALLOCATABLE arrays cannot be used as components of derived types.

In summary, ALLOCATABLE arrays are more efficient and POINTER arrays are more flexible.

An interesting discussion about pointer efficiency can be found here

http://euclid.math.fsu.edu/ pbismu/RESEARCH/SOFTWARE/POINTER/pointer.html

21.12 Practical Use of Pointers

Pointers can be of great use in iterative problems. Iterative methods:

- make guess at required solution;
- use guess as input to equation to produce better approximation;
- use new approximation to obtain better approximation;
- repeat until degree of accuracy is obtained;

As an example, let us find the square root of a number by iteration:

$$X_n = \frac{X_{n-1} + Y/X_{n-1}}{2}$$

(A stable and rapidly convergent algorithm which produces acceptable value for square root in small number of iterations.)

The following iterative code segment could be found in a solution,

```
REAL :: Y, tol=1.0E-4
REAL :: prev_app, next_app
prev_app = 1.0 ! set initial approx
DO                ! calculate next approx
 next_app = (prev_app + Y/prev_app)/2.0
 IF (ABS((next_app-prev_app)/next_app) < tol) THEN
  EXIT
 ELSE
  prev_app = next_app
 END IF
```

```
    END DO
    ! next_app now contains the required result
    ...
```

This structure is typical of such iterative processes, however, in general the objects involved are not simple real variables and the update process is not just a simple arithmetic assignment. It is frequently the case that the objects which hold the approximations are large arrays and the cost of the copy operation, moving the result of one iteration to become the input to the next, is a significant proportion of the total cost of an iteration. In such cases it is much more efficient to use pointers to avoid having to do the copying operations.

For array problems a more efficient solution can be constructed which uses pointers:

```
PROGRAM Solve
 IMPLICIT NONE
 REAL :: tol=1.0E-4
 REAL, DIMENSION(1000,1000), TARGET :: approx1, approx2
 REAL, DIMENSION(:,:), POINTER :: prev_approx, next_approx, swap
  prev_approx => approx1
  next_approx => approx2
  prev_approx => initial_approximation(.....)
  DO
   next_approx = iteration_function_of(prev_approx)
   IF (ABS(MAXVAL(next_approx-prev_approx)) < tol) EXIT
   swap => prev_approx
   prev_approx => next_approx
   next_approx => swap
  END DO
CONTAINS
 FUNCTION iteration_function_of(in)
  REAL, DIMENSION(:,:) :: in
  REAL, DIMENSION(SIZE(in,1),SIZE(in,2)) :: iteration_function_of
   ......
 END FUNCTION
END PROGRAM Solve
```

The `DO` loop contains the iteration which gradually moves towards the solution. `iteration_function_of` is an array valued function uses the values of `prev_approx` and improves them to be closer to the solution. Convergence is tested by examining the difference between the two most recent approximations. The worst value (chosen by `MAXVAL`) is used. If this worst value differs from the corresponding value from the previous iteration by more than the tolerance then a further iteration is performed. The supposition is that when values remain 'constant' between iterations they have reached their optimum value. In other words when all array elements have stabilised, the solution has been found.

The following statements,

```
  swap        => prev_approx
  prev_approx => next_approx
  next_approx => swap
```

shuffle the arrays around (turns the current approximation in to the previous one for the next iteration). `next_approx` ends up pointing to the old `prev_approx` which has outlived its usefulness — on the next iteration this array can be used to hold the next set of approximations. Here, pointer assignment replaces two potentially expensive copy operations. The net effect is that `prev_approx` and `next_approx` are swapped at the cost of 3 pointer assignments which is quite small, whereas, to swap the actual objects would require $1000 \times 1000 = 1000000$ real assignments, a much more expensive operation.

21.13 Pointers and Procedures

Pointer variables may be used as actual and dummy arguments in much the same way as non-pointer variables. As with all arguments, dummies and actuals must match in type, kind and rank. Note that a `POINTER` dummy argument **cannot** have the `INTENT` attribute. In some areas pointers are slightly more flexible than non-pointer arguments, for example, unallocated `ALLOCATABLE` arrays cannot be passed as actual arguments to a procedure, but unassociated (unallocated) `POINTER` arrays can.

If a pointer is to be used as an actual argument then the interface must be explicit. The reason for this is because a pointer argument can be interpreted in two ways:

1. immediately dereference and pass the target (the corresponding dummy argument does **not** have the `POINTER` attribute).
2. pass the pointer so it can be manipulated as a pointer in the procedure (the corresponding dummy argument does have the `POINTER` attribute).

For example,

```
PROGRAM Supping
 IMPLICIT NONE
 INTEGER, POINTER :: Pint1, Pint2
 ...
 CALL Beer(Pint1,Pint2)
 ...
CONTAINS
 SUBROUTINE Beer(arg1,arg2)
  INTEGER, POINTER :: arg1
  INTEGER, INTENT(IN) :: arg2
   ....
 END SUBROUTINE Beer
END PROGRAM Supping
```

`Pint2` which corresponds to the non-pointer `arg2` is dereferenced before being passed to `Beer`, `Pint1` is not.

Referring to `Pint1` in the main program references exactly the same space as `arg1` in the subroutine, however, it is not guaranteed that if an unassociated pointer is associated in a procedure that on return to the call site the pointer will still be associated. This is not defined to be the case in the standard.

21.14 Pointer Valued Functions

A function may be pointer valued, in the artificial example below the function `ptr` returns a pointer to the larger of `a` and `b`:

```
PROGRAM main
 IMPLICIT NONE
 REAL :: x
 INTEGER, TARGET :: a, b
 INTEGER, POINTER :: largest
  CALL RANDOM_NUMBER(x)
  a = 10000.0*x
  CALL RANDOM_NUMBER(x)
  b = 10000.0*x
 largest => ptr()
 print*, largest
CONTAINS
 FUNCTION ptr()
  INTEGER, POINTER :: ptr
  IF (a .GT. b) THEN
   ptr => a
  ELSE
   ptr => b
  END IF
 END FUNCTION ptr
END PROGRAM main
```

It is not possible to have an attribute in the `FUNCTION` statement so the specification that the function is pointer valued must be made amongst the declarations. The following declaration would also suffice,

```
 ...
INTEGER FUNCTION ptr()
 POINTER :: ptr
  ...
```

Clearly for the function to return a result the correct type, the function name must identify a target by being on the LHS of a pointer assignment.

The interface of an external pointer valued function must always be explicit (see 28).

The following illustrates an external pointer valued function:

```
PROGRAM main
 IMPLICIT NONE
 REAL :: x
 INTEGER, TARGET :: a, b
 INTEGER, POINTER :: largest
 INTERFACE
  FUNCTION ptr(a,b)
   IMPLICIT NONE
   INTEGER, TARGET, INTENT(IN) :: a, b
   INTEGER, POINTER :: ptr
  END FUNCTION ptr
 END INTERFACE
  CALL RANDOM_NUMBER(x)
  a = 10000.0*x
  CALL RANDOM_NUMBER(x)
  b = 10000.0*x
```

```
  largest => ptr(a,b)
  print*, a, b, "Largest: ", largest
END PROGRAM main

FUNCTION ptr(a,b)
 IMPLICIT NONE
 INTEGER, TARGET, INTENT(IN) :: a, b
 INTEGER, POINTER :: ptr
  IF (a .GT. b) THEN
   ptr => a
  ELSE
   ptr => b
  END IF
END FUNCTION ptr
```

21.15 Pointer I / O

When pointers appear in I / O statements they are always immediately dereferenced so their target is accessed instead. Clearly, a pointer must not be disassociated as dereferencing would make no sense.

Consider the following example,

```
PROGRAM Eggie
 IMPLICIT NONE
 INTEGER :: ierr
 REAL, DIMENSION(3), TARGET :: arr = (/1.,2.,3./)
 REAL, DIMENSION(:), POINTER :: p, q
  p => arr
  PRINT*, p
  ALLOCATE(q(5), STAT=ierr)
  IF (ierr .EQ. 0) THEN
   READ*,  q
   PRINT*, q
   DEALLOCATE(q)
   PRINT*, q ! invalid, no valid target
  END IF
END PROGRAM Eggie
```

In the first output statement, `p` is dereferenced and the three real numbers are printed out in array element ordering. The `READ` statement works in a very similar way. Once `q` has been allocated its appearance in the input statement causes the program to expect 5 real numbers to be entered on the standard input channel.

The last `PRINT` statement is an error as it is not permissible to attempt to read or write to or from a deallocated pointer.

It is not possible to print out the address of the target of a pointer.

22 Derived Types

It is often advantageous to express some objects in terms of aggregate structures, for example:

- coordinates: (x, y, z);
- addresses: name, number, street, etc.

Fortran 90 allows compound entities or *derived types* to be defined, (in other languages these may be known as structures or records). Programs can be made simpler, more maintainable and more robust by judicious choice of data structures. Choosing an efficient data structure is a complex process and many books have been written on the subject.

In Fortran 90 a new type can be defined in a *derived-type-statement*:

```
TYPE COORDS_3D
 REAL :: x, y, z
END TYPE COORDS_3D
```

The type COORDS_3D has three REAL components, x, y and z. (These could have been declared on three separate lines and the resultant type would have been identical.) Objects of this new type can be used in much the same way as objects of an intrinsic type, for example, in assignment statements, as procedure arguments or in I/O statements, however, it is not permissible to initialise a component in a derived type declaration statement.

An object of type COORDS_3D can be declared in a type declaration statement,

```
TYPE (COORDS_3D) :: pt
```

Derived type objects can be attributed in the same way as for regular type declarations, for example, the DIMENSION, TARGET or ALLOCATABLE attributes are all valid subject to a small proviso mentioned later.

```
TYPE (COORDS_3D), DIMENSION(10,20), TARGET :: pt_arr
```

Derived type objects can be used as dummy or actual arguments and so can be given the INTENT or OPTIONAL attributes. they cannot, however, possess the PARAMETER attribute.

22.1 Supertypes

A new derived type can be declared that has, as one of its components, a further derived type. This type must have already been declared or must be the type currently being declared. (The latter is how recursive data structures are formed.)

```
TYPE SPHERE
 TYPE (COORDS_3D) :: centre
 REAL             :: radius
END TYPE SPHERE
```

A sphere is characterised by a centre point and a radius, a type with these two components is an intuitive way of describing such an object. This is better than defining a type with 4 real components which are all at the same level:

```
TYPE SPHERE
 REAL :: x, y, z, radius
END TYPE SPHERE
```

Objects of type SPHERE can be declared:

```
TYPE (SPHERE) :: bubble, ball
```

There is no sequence association for derived types, in other words, there is no reason to suppose that objects of type COORDS_3D should occupy 3 contiguous REAL storage locations.

22.2 Derived Type Assignment

Values can be assigned to derived types in two ways:

- component by component;
- as an object.

The % operator can be used to select a single specific component of a derived type object which allows the object to be given values in a piecewise fashion. In order to select a specific component it is necessary to know the names given to the components when the derived type was initially declared.

The object bubble is of type SPHERE which we know is composed of two components: centre and a radius, however, centre is itself a derived type component (so therefore not of intrinsic type) meaning that the individual components must be selected from this component too. In order to assign a value (an intrinsic typed expression) to a derived type component using a regular assignment statement, it must be ensured that each component is fully resolved, in other words, the component selected is of intrinsic type.

In the following two statements,

```
bubble%radius = 3.0
bubble%centre%x = 1.0
```

bubble%radius is of type REAL so can be assigned a REAL value but bubble%centre is of type COORDS_3D which must therefore must be further resolved in order to perform assignment of a single intrinsically typed value. bubble%centre%x (or ..%y or ..%z) is of intrinsic type so can be given a value using the intrinsic assignment operator =.

During assignments of this type, normal type coercion rules still apply, for example, an INTEGER could be assigned to a REAL component of a derived type with the INTEGER value simply being promoted.

As an alternative to the abovementioned component-by-component selection it is possible to use a derived type constructor to assign values to the whole object. This structure constructor takes the form of the derived type name followed by a parenthesised list of values, one for each component. For example, consider,

```
bubble%centre = COORDS_3D(1.0,2.0,3.0)
```

`bubble%centre` is of type `COORDS_3D` so we can use the *automatically defined* derived type constructor (`COORDS_3D`) to coerce the listed values to the correct type. Here, as `COORDS_3D` has three components the constructor must also be supplied with 3 expressions.

The object `bubble` can be set up using the `SPHERE` constructor, this has two components, one of type `COORDS_3D` and one of type `REAL`. As `bubble%centre` is of type `COORDS_3D` this can be used as one component of the constructor. The other component is default `REAL`.

```
bubble = SPHERE(bubble%centre,10.)
```

In the above example, `COORDS_3D(1.,2.,3.)` could be used in place of `bubble%centre`.

```
bubble = SPHERE(COORDS_3D(1.,2.,3.),10.)
```

It is not possible to have,

```
bubble = SPHERE(1.,2.,3.,10.)
```

Even though the type constructor looks like a function call there is no keyword selection or optional fields.

Assignment between two objects of the same derived type is intrinsically defined, `ball` and `bubble` can be equated,

```
ball = bubble
```

22.3 Arrays and Derived Types

It is possible to define derived types which contain arrays of intrinsic or derived type objects. These arrays cannot be `ALLOCATABLE` but can have the `POINTER` attribute.

Consider,

```
TYPE FLOBBLE
 CHARACTER(LEN=6)          :: nom
 INTEGER, DIMENSION(10,10) :: harry
END TYPE FLOBBLE
TYPE (FLOBBLE)                :: bill
TYPE (FLOBBLE), DIMENSION(10) :: ben
```

it is also possible to declare an array of objects where each element of the array also contains an array, however, it is not permissible to refer to an 'array of arrays'. We can refer to an element or subsection of the array component:

```
bill%harry(7,7)
bill%harry(:,::2)
ben(1)%harry(7,7)
ben(9)%harry(:,:)
ben(:)%harry(7,7)
```

in the above example, the five expressions all define a section of some sort, there is only one non-scalar index in the reference. In a derived type that has more levels of derived type nesting there can still only be sectioning of at most one component level.

The following two expressions are not allowed because sectioning is in more than one component,

```
ben(:)%harry(:,::2) ! invalid
ben(9:9)%harry(:,:) ! invalid
```

The first expression is invalid because a section is taken from the array of objects and also from one of the components of each object; the second expression is invalid for the same reason although slightly more subtle, the reference here is a one element subsection of `ben` (not scalar) and the whole of the `harry` component. (Remember that `ben(9)` is scalar and `ben(9:9)` is non scalar.)

22.4 Derived Type I/O

Derived type objects can be used in I/O statements in much the same way as intrinsic objects, the only restrictions are that

- the derived type must not contain a `POINTER` component, (this is because the pointer reference may never be resolved. If, for example, a circular list structure were to appear in an output statement the program would never terminate.)
- the visibility of the internal type components must not be restricted (see Section 19.3).

A derived type object is written out or read in as if all the components were explicitly inserted into the output statement in the same order in which they were defined.

So,

```
PRINT*, bubble
```

is exactly equivalent to

```
PRINT*, bubble%centre%x, bubble%centre%y, &
        bubble%centre%z, bubble%radius
```

Derived types containing arrays are handled in the expected way. Individual components are input / output in order with individual elements of arrays output in array element order.

22.5 Derived Types and Procedures

Derived types can be used as arguments to procedures, however, the type definition *must* be made accessible by either use or host association. Indeed, if the same textual definition of a (non-`SEQUENCE`) type appears in two separate program units, the two types (which have the same names, types and components) are not considered to be the same. (This is because of optimisation — the compiler can represent the type however it sees fit.

(A SEQUENCE type is a special class of derived type which relies on sequence association. The components of a sequence type are bound to be stored in contiguous memory locations in the order of the component declarations.)

All type definitions should be encapsulated in a module:

```
MODULE VecDef
 TYPE vec
  REAL :: r
  REAL :: theta
 END TYPE vec
END MODULE VecDef
```

To make the type definitions visible, the module must be used:

```
PROGRAM Up
 USE VecDef
 IMPLICIT NONE
  TYPE(vec) :: north
   CALL subby(north)
  ...
CONTAINS
 SUBROUTINE subby(arg)
  TYPE(vec), INTENT(IN) :: arg
   ...
 END SUBROUTINE subby
END PROGRAM Up
```

Type definitions can only become accessible by *host* or *use* association.

So long as the type definition is visible, derived type arguments behave like intrinsically typed arguments, they can be given attributes (OPTIONAL, INTENT, DIMENSION, SAVE, ALLOCATABLE, etc.). Interfaces are required in the same situations as for intrinsic types.

22.6 Derived Type Valued Functions

Functions can return results of an arbitrary defined type

```
FUNCTION Poo(kanga, roo)
 USE VecDef
 TYPE (vec) :: Poo
 TYPE (vec), INTENT(IN) :: kanga, roo
 Poo = ...
END FUNCTION Poo
```

Recall that the type definition must be made available by host or use association.

Question 51: Complex arithmetic

Modify your existing Integer_Complex_Arithmetic module and define a type INTCOMPLEX which is able to represent a complex number with integer components.

```
TYPE(INTCOMPLEX)
 INTEGER x,y
END TYPE
```

Modify the module procedures etc. to accept this datatype instead of the previously implemented 2 element array.

Add two functions UPLUS and UMINUS to the module which each take one INTCOMPLEX argument such that:

```
UPLUS(ZI)  = +ZI
UMINUS(ZI) = -ZI
```

Write a test program to demonstrate.

Module 9: Modules and Object-based Programming

22.7 POINTER Components of Derived Types

It is forbidden to include ALLOCATABLE arrays as components of derived types, however, complex data structures can be constructed by including POINTER variables instead. These components can point to scalar or array valued intrinsic or derived types which means that dynamically sized structures can be created and manipulated, As an example consider,

```
TYPE VSTRING
  CHARACTER, DIMENSION(:), POINTER :: chars
END TYPE VSTRING
```

Objects of this type have a component which is a pointer to a dynamically sized 1-D array of characters. This data type is useful for storing strings of different lengths.

```
TYPE(VSTRING) :: Pvs1, Pvs2
 ...
ALLOCATE(Pvs1%chars(5))
Pvs1%chars = (/"H","e","l","l","o"/)
```

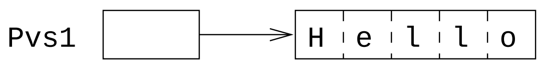


Figure 32: A Ponter to a Variable Length String

There is a restriction concerning I/O of user-defined types which contain pointer components, (see Section 22.4); the object must be resolved to an intrinsic type component. If,

```
TYPE(VSTRING) :: nom
 ...
print*, nom                  ! Illegal
print*, nom%chars            ! fine
```

the composite object nom cannot be output, but since nom%chars is a pointer to a CHARACTER array it can be written.

22.8 Pointers and Lists

The use of pointers with derived types also provides support for structures such as linked lists. Derived types may contain pointers to any other type including the type being currently defined. It is not possible to point to a type which has yet to be defined, the target type must be defined or be being defined.

In the following example, CELL contains a pointer to another object also of type CELL,

```
TYPE CELL
 INTEGER :: val
 TYPE (CELL), POINTER :: next
END TYPE CELL
```

this defines a single linked list of the following schematic structure,

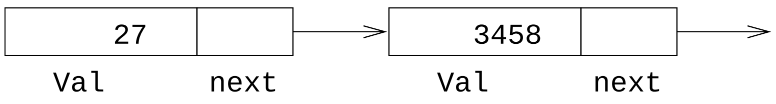


Figure 33: A Linked List With 2 Elements

the above diagram represents two cells, each cell contains a value (INTEGER) and a link to the next cell. The last cell in the list should have a null target (use the NULLIFY command). As long as something points to the head of the structure the whole of the list can be traversed.

Intrinsic assignment between structures containing pointer components is subtlely different from 'normal' assignment. = is always intrinsically defined for assignment between two objects of the same derived type so when the type contains a pointer component = must "behave sensibly", in other words it does what the user would expect it to do:

- non-pointer components are assigned by copying the value from the RHS to the corresponding component on the LHS,
- pointer components are pointer assigned using the => operator.

So:

```
TYPE(CELL) :: A
TYPE(CELL), TARGET :: B
A = B
```

is equivalent to:

```
A%val = B%val
A%next => B%next
```

Other recursive structures can be built up such as singly or doubly linked lists or n-ary trees.

22.8.1 Linked List Example

The following fragment would create a linked list of cells starting at head and terminating with a cell whose next pointer is null (disassociated).

```
PROGRAM Thingy
 IMPLICIT NONE
 TYPE (CELL), TARGET  :: head
 TYPE (CELL), POINTER :: curr, temp
 INTEGER              :: k
 head%val = 0             ! listhead = default
 NULLIFY(head%next)       ! un-undefine
 curr => head             ! curr head of list
 DO
  READ*, k                ! get value of k
  ALLOCATE(temp)          ! create new cell
  temp%val=k              ! assign k to new cell
  NULLIFY(temp%next)      ! set disassociated
  curr%next => temp       ! attach new cell to
                          ! end of list
  curr => temp            ! curr points to new
                          ! end of list
 END DO
END PROGRAM Thingy
```

There now follows a line-by-line dissection of the example,

- `TYPE (CELL), TARGET  :: head` — this is a cell which anchors the list. This is the starting point for traversal of the list.
- `TYPE (CELL), POINTER :: curr, temp` — declare pointers to two cells. `curr` points to the current position in the list (this will tell us where the next list cell is to be added), `temp` is used for receiving a newly allocated cell before it gets tagged onto the list.
- `INTEGER :: k` variable used when reading the data.
- `head%val = 0` — set the value of the list anchor to be zero (null).
- `NULLIFY(head%next)` disassociate (nullify) the head of the list.
- `curr => head` — `curr` marks the position in the list where the next cell is to be added. As we haven't really started yet this is at the head of the list.
- `DO ... END DO` — a loop where the input is read and the list is built up.
- `READ ...` — read value of `k`.
- `ALLOCATE(temp)` — creates a new cell called `temp`, if the loop is not on its first iteration the previously created cell will have been attached to the end of the list. The `ALLOCATE` statement will create a new `temp` elsewhere in the memory which can then (eventually) be attached to the linked list.
- `temp%val=k` — set the `value` part of the cell to the previous input value.
- `NULLIFY(temp%next)` — initialise (disassociate, nullify) the pointer component of the new cell. This cell will be the end of the list for at least one loop iteration.

- `curr%next => temp` — attach the new cell to the end of the list. The list is anchored by `head` and the last cell in the list is nullified and pointed to by `curr`. Adding a new cell to the end of the list is achieved by simply making the last list item point to the new cell. The pointer component of the new cell has already been nullified ready for the next iteration of the loop.
- `curr => temp` reassign the pointer which indicates the end of the list. `temp` is now the last item in the list so `curr` should be made to point to this cell.

To summarise:

```
head%val = 0
NULLIFY(head%next)
curr => head
```

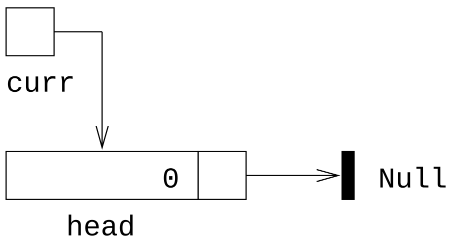


Figure 34: Initialisation of the List

This first set of statements take a `TARGET` cell `head` (the head of the list) and initialise it. Here, initialisation involves setting the numeric component to zero (`head%val = 0`), and `NULLIFY`ing the pointer component. The pointer `curr` is set to point a the current insert point in the list; as the list has no members the current insert point is at the head of the list.

```
ALLOCATE(temp)
temp%val = k
NULLIFY(temp%next)
```

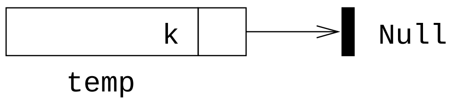


Figure 35: Initialisation of a New Cell

The second sequence of statements creates a new cell from the heap storage (called `temp`). This cell has its value component set to the most recent input value, (`k`,) and is then given the disassociated status (using `NULLIFY`).

```
curr%next => temp
curr => temp
```

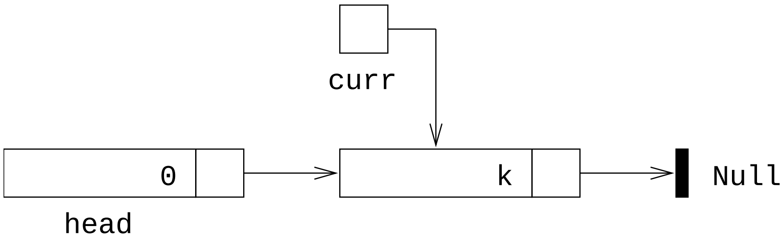


Figure 36: The Addition of a New List Member

The third sequence of statements take the new cell an insert it in the list at the current insert point. The insert point (pointed to by `curr`) is then moved to reflect the addition.

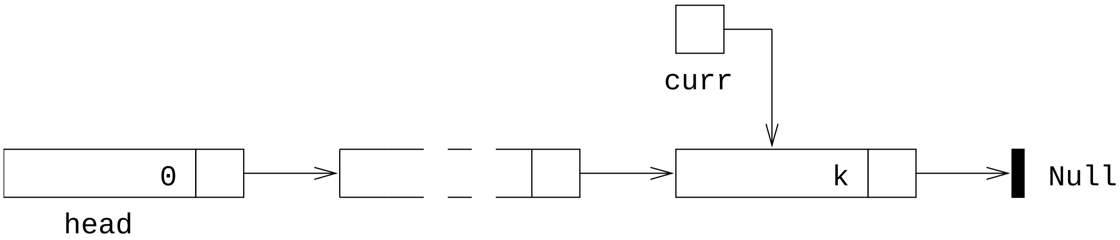


Figure 37: The General Structure of the List

When the list is completed it will have a structure similar to that given in the above diagram. The list can be traversed by starting at the `head` cell and following the pointer links.

It is no good having a linked list if it cannot be navigated. A "walk-through" which prints each cell value out could be programmed as follows:

```
curr => head
DO
 PRINT*, curr%val
 IF(.NOT.ASSOCIATED(curr%next)) EXIT
 curr => curr%next
END DO
```

- `curr => head` — the variable `curr` is used as a place-marker. Since we do not need to know where the end of the list is there is no need to keep `curr` pointing at it. (We can always find the end by starting at the head and traversing the list until we reach a null pointer.) Thus, `curr` is being reused.
- `DO ... END DO` — the list traversal is performed within this loop. The criterion for exiting is that the pointer component of the current cell is disassociated (null).
- `PRINT*, curr%val` — print out the value of the current cell, this first time around the loop this will be the value of the list head which was set to the default value of 0, if this is not required then it should be skipped.

- `IF(.NOT.ASSOCIATED(curr%next)) EXIT` — if the pointer is not associated with a target then the loop is exited. This can be thought of as "if the pointer component is null then `EXIT`".
- `curr => curr%next` — move to the next cell, the statement says "make `curr` point to the target of `curr%next`". (`curr%next` is immediately dereferenced.)

Question 52: Defined Types and Pointers

Write a program to implement a tree sort. The program should read in textual names from standard input and place each name in a binary tree. The algorithm is

1. read in a name
2. `IF` name = 'end' `EXIT`
3. add the name to the tree
4. print the tree

Steps 3 and 4 should be implemented with recursive procedures.

The algorithm for Step 3 is for each node:

1. if the node is null, add the name to the node and return
2. if the name is earlier in the alphabet than the node name
   - recursively go down the left branch

   otherwise
   - recursively go down the right branch

The algorithm for Step 4 is:

1. if the node is null return
2. recursively print the left branch
3. print the node name
4. recursively print the right branch

22.9 Arrays of Pointers

As it is possible to create arrays of objects of derived types, so is possible to create what are in effect arrays of pointers:

```
TYPE iPTR
 INTEGER, POINTER :: compon
END TYPE iPTR
TYPE(iPTR), DIMENSION(100) :: ints
```

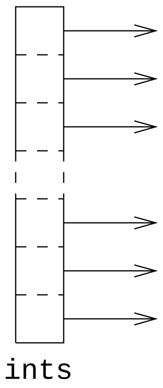


Figure 38: Visualisation of an Array Of Pointers

Visualisation,

Here we have an array of 100 elements each of which can point to an INTEGER. It is not possible to refer to ints(:) as there is no context where a whole array (or sub-array) of pointers like this can be used, dereferencing can only be used on a scalar pointer,

```
ints(10)%compon    ! valid
ints(:)%compon     ! not valid
ints(10:10)%compon ! not valid
```

If desired ints could have been ALLOCATABLE:

```
TYPE iPTR
 INTEGER, POINTER :: compon
END TYPE iPTR
TYPE(iPTR), DIMENSION(:), ALLOCATABLE :: ints
 ....
ALLOCATE(ints(100))
```

The following is also acceptable and defines and array of pointers to integer arrays,

```
TYPE iaPTR
 INTEGER, POINTER, DIMENSION(:) :: acompon
END TYPE iaPTR
TYPE(iaPTR), DIMENSION(100) :: intsarrays
```

Each pointer component can be made to point to an unnamed part of the heap storage or to an existing array,

```
INTEGER :: ierr
 ...
ALLOCATE(intsarrays(1)%acompon(20),STAT=ierr)
 ...
ALLOCATE(intsarrays(2)%acompon(40),STAT=ierr)
```

or

```
INTEGER, TARGET :: Ia(6), Ib(7), Ic(8)
 ...
intsarrays(1)%acompon => Ia
intsarrays(2)%acompon => Ib
intsarrays(3)%acompon => Ic
```

`ALLOCATABLE` arrays cannot be components of derived types.

Question 53: Arrays of Pointers

Rewrite the 'MAXLOC' sorting example using an array of pointers instead of a vector subscript to hold the order of the numbers.

Question 54: Sorting Using Pointers

Using derived types and pointers to minimise storage, write a program to set up a user-specified number of 5 element arrays, and fill them with random numbers between 0. and 1000.0 and then using your SimpleStats MODULE sort them firstly on the basis of their mean and secondly on the basis of their standard deviation. Print out the results of the sorting so that the array with the greatest mean is printed out first alongside the actual value of its mean (to demonstrate correctness) and the array with the smallest mean is printed out last. Do the same with the sorted list of standard deviations, viz:

```
Position 1. Mean = 452.0
            Array is 960.66666 .... 34.87311
Position 2. Mean = 122.5
            Array is 460.92653 .... 50.80036

Position 10. Mean = 781.77
            Array is 160.23456 .... 4.80713
```

and so on.

[**Hint:** Use an array of the following user defined type:

```
TYPE ARRAY_INFO
 REAL, DIMENSION(:), POINTER :: Array
 REAL                        :: ArrayMean
 REAL                        :: ArrayDev
END TYPE ARRAY_INFO
```

to hold the array, its mean and its standard deviation. You should create two arrays where each element can point to objects of type ARRAY_INFO these arrays can be used to represent the sorted mean and standard deviation lists.

You may find it useful to employ other data structures to aid the solution. Do not worry about implementing an efficient sorting mechanism.]

23 Modules — Type and Procedure Packaging

In a 'proper' program it would be very useful to use derived types in the same way as intrinsic types and even use them interchangeably with intrinsic types. To do this we must, for each derived type definition, provide the same functionality that is provided for intrinsic types. For example, there are no intrinsic functions defined for user-defined types and operations between derived and intrinsic types are very limited, however,

- ☐ procedure arguments can be derived types.
- ☐ functions can return results of derived types.
- ☐ operators can be overloaded to apply to derived types.

So we can write a 'package' which contains:

- ☐ type definitions,
- ☐ constructors,
- ☐ overloaded intrinsics,
- ☐ an overload set of operators,
- ☐ generic overload set definitions,
- ☐ other useful procedures.

This form of bundling is known as 'encapsulation'.

There are many advantages to including the above functionality in a module,

- ☐ derived type definitions must be placed in a module so that their structure can be made visible by use association in non `MODULE` program units,
- ☐ the interfaces for generic procedures, user defined operators, operator overloads and assignment overloads must all be explicit,
- ☐ following the same logic it makes excellent sense to put the actual overloaded intrinsic procedures and intrinsic operators in this module too.
- ☐ any other relevant procedures should also be placed in the module to provide an encapsulated package of type definitions, operator definitions, intrinsic procedures and access routines.

This module, when `USE`d, would provide an almost transparent *semantic extension* to Fortran 90.

An example of this kind of module is the varying string module which is to be an ancillary standard to Fortran 95. An implementation of this module has been written at Liverpool and includes all the relevant intrinsic functions, for example, `LEN`, `INDEX`, and operators, for example, `//`, `LGE`, `=`, which have been defined for the new `VARYING_STRING` type. This means that objects of type `VARYING_STRING` can be used (almost) transparently with objects of the same and intrinsic (`CHARACTER`) type.

23.1 Derived Type Constructors

Derived types have their in-built constructors, however, it is often a good idea to write a specific routine instead. These should always be used instead of the default constructor.

Purpose written constructors can support default values, optional and keyword arguments and will not have to be modified if the internal structure of the type is changed. It is also possible to hide the internal details of the type by placing a PRIVATE statement with the type definition:

```
MODULE ThreeDee

 IMPLICIT NONE

 TYPE Coords_3D
  PRIVATE
  REAL :: x, y, z
 END TYPE Coords_3D

CONTAINS

 TYPE(Coords_3D) FUNCTION Init_Coords_3D(x,y,z)
  REAL, INTENT(IN), OPTIONAL :: x,y,z
  ! Set Defaults
  Init_Coords_3D = Coords_3D(0.0,0.0,0.0)
  IF (PRESENT(x)) Init_Coords_3D%x = x
  IF (PRESENT(y)) Init_Coords_3D%y = y
  IF (PRESENT(z)) Init_Coords_3D%z = z
 END FUNCTION Init_Coords_3D

END MODULE ThreeDee
```

If an argument is not supplied then the corresponding component of Coords_3D is set to zero.

The following calls are all valid,

```
PROGRAM Testo3D

 USE ThreeDee
 IMPLICIT NONE

 TYPE(Coords_3D) :: pt1, pt2, pt3

  pt1 = Init_Coords_3D()
  pt2 = Init_Coords_3D(1.0,2.0)
  pt3 = Init_Coords_3D(y=10.0)

  PRINT*, "pt1:", pt1
  PRINT*, "pt2:", pt2
  PRINT*, "pt3:", pt3

END PROGRAM Testo3D
```

This program will produce:

```
pt1: 0.0  0.0  0.0
pt2: 1.0  2.0  0.0
pt3: 0.0 10.0  0.0
```

This approach allows greater flexibility.

23.2 Generic Procedures

Defining a generic interface to a procedure is a way of grouping procedures with similar functionality together under one name. Typically a generic procedure interface has a general name and contains a list of specific procedures with similar functionality which are implemented for all data types in a program. Whenever the generic procedure name is used the compiler is able to tell which specific procedure this instance of use corresponds to by looking the types of the actual arguments. The specific procedure can then be invoked.

Most intrinsics are generic in that their type is determined by their argument(s). For example, `ABS(X)`:

- returns a real value if `X` is `REAL`.
- returns a real value if `X` is `COMPLEX`.
- returns a integer value if `X` is `INTEGER`.

It is possible to ignore the generic interface and still refer to a procedure by its specific name, for example, `DABS`, `CABS`, or `DABS`— the end effect will be the same. Note that if the procedure name is used as an actual argument then the specific name *must* be used. Fortran 90 does not support run-time resolution of generic overloads as this would compromise efficiency.

A user can define his / her own generic names for user procedures. This is implemented through a generic interface specification. The user may then call the procedure with a generic name and the compiler will examine the *number, type, rank* and *kind* of the non-optional arguments to decide which specific procedure to call (at compile-time). (See Section 25.9 for the interaction between generic interfaces and kinds.) If no such procedure exists then an error is generated. The set of procedures in the generic interface is known as an *overload set*. For example,

```
INTERFACE CLEAR
  SUBROUTINE clear_int(a)
    INTEGER, DIMENSION(:), INTENT(INOUT) :: a
  END SUBROUTINE clear_int
  SUBROUTINE clear_real(a)
    REAL, DIMENSION(:), INTENT(INOUT) :: a
  END SUBROUTINE clear_real
END INTERFACE ! CLEAR
```

`clear_int` and `clear_real` can both be called by the generic name `CLEAR`. `clear_int` will be called if the actual argument is `INTEGER`, `clear_real` will be invoked if the argument is `REAL`. The overload set consists of `clear_int` and `clear_real`. It is believed that using generic procedure names makes programming much easier as the user does not have to remember the specific name of a procedure but only the function that it performs.

Procedures in a generic interface must be all functions or all subroutines — they cannot be mixed. In the case of functions, it is the type of the arguments not the type of the result which determines which specific procedure is called.

In order to use user defined generic procedure names the interfaces must be explicit. The best way to do this is to put the generic interface into a module and attach it by USE association.

Note: currently the END INTERFACE statement cannot contain the interface name! This is a language anomaly and is due to be corrected in Fortran 95. It is likely that many compilers will accept code with the name present.

Given,

```
MODULE schmodule
 INTERFACE CLEAR ! generic int
  MODULE PROCEDURE clear_int
  MODULE PROCEDURE clear_real
 END INTERFACE ! CLEAR
CONTAINS
 SUBROUTINE clear_int(a)
  INTEGER, DIMENSION(:), INTENT(INOUT) :: a
   .... ! actual code
 END SUBROUTINE clear_int
 SUBROUTINE clear_real(a)
  REAL, DIMENSION(:), INTENT(INOUT) :: a
   .... ! actual code
 END SUBROUTINE clear_real
END MODULE schmodule

PROGRAM Main
 IMPLICIT NONE
 USE schmodule
 REAL :: prices(100)
 INTEGER :: counts(50)
  CALL CLEAR(prices) ! generic call
  CALL CLEAR(counts) ! generic call
END PROGRAM Main
```

`prices` is a real-valued array so the corresponding specific procedure from the overload set is the one with a single real-valued array as a dummy argument — `clear_real`.

`counts` is integer-valued so the second call will be resolved to `clear_int`.

In order for the compiler to be able to resolve the reference, both module procedures must be unique with respect to their (non-optional) arguments.

23.3 Generic Interfaces — Commentary

In order for a call to a generic interface to be resolved the overload set must be unambiguous. In other words, the specified procedures must all be unique in at least one of: *type*, *kind* or *rank* in their **non-optional** arguments. (In other words, by examining the argument(s), the compiler calculates which specific procedure to invoke.)

When parametrised types are being used in a program, default intrinsic types should **not** be used for arguments in procedures that occur in generic interfaces. This is because the default type corresponds to a particular kind which varies from processor to processor. (See Section 25.8.)

23.4 Derived Type I/O

Derived types with PRIVATE components need special procedures for I/O. They cannot be used in a simple PRINT or WRITE statement:

```
MODULE ThreeDee

 IMPLICIT NONE

 TYPE Coords_3D
  PRIVATE
  REAL :: x, y, z
 END TYPE Coords_3D

 INTERFACE Print
  MODULE PROCEDURE Print_Coords_3D
 END INTERFACE ! Print

CONTAINS

 SUBROUTINE Print_Coords_3D(Coord)
  TYPE(Coords_3D), INTENT(IN) :: Coord
   PRINT*, Coord%x, Coord%y, Coord%z
 END SUBROUTINE Print_Coords_3D

 TYPE(Coords_3D) FUNCTION Init_Coords_3D(x,y,z)
  REAL, INTENT(IN), OPTIONAL :: x,y,z
  ! Set Defaults
  Init_Coords_3D = Coords_3D(0.0,0.0,0.0)
  IF (PRESENT(x)) Init_Coords_3D%x = x
  IF (PRESENT(y)) Init_Coords_3D%y = y
  IF (PRESENT(z)) Init_Coords_3D%z = z
 END FUNCTION Init_Coords_3D

END MODULE ThreeDee
```

Coords_3D may only be output via the generic interface Print which calls the Print_Coords_3D procedure. A generic Print interface can be extended for each and every derived type defined:

```
 CALL Print(init_Coords_3D(1.0,2.0,3.0))
```

Using this approach further abstracts the type definition from the users program. If the internal structure of the type is changed then the output routine can be changed accordingly — the users' program can remain untouched.

23.5 Overloading Intrinsic Procedures

When a new type is added, it is a simple process to add a new overload to any relevant intrinsic procedures. A generic interface is defined with the same name as the existing generic intrinsic name.

Any procedures included in this interface specification are added to the overload set. (Note: The body of the procedures should be PURE in the sense that they should not alter any global data or produce output.) Existing intrinsics may be actually be *redefined* by including a procedure with the same argument types as the original intrinsic. When the generic name is supplied, the new procedure is called instead of the Fortran 90 intrinsic of the same name.

As an example, assume that we wish to extend the LEN_TRIM intrinsic to return the number of letters in the owners name when applied to objects of type HOUSE,

```
MODULE new_house_defs
 IMPLICIT NONE
 TYPE HOUSE
  CHARACTER(LEN=16) :: owner
  INTEGER           :: residents
  REAL              :: value
 END TYPE HOUSE
 INTERFACE LEN_TRIM
  MODULE PROCEDURE owner_len_trim
 END INTERFACE
CONTAINS
 FUNCTION owner_len_trim(ho)
  TYPE(HOUSE), INTENT(IN) :: ho
  INTEGER :: owner_len_trim
  owner_len = LEN_TRIM(ho%owner)
 END FUNCTION owner_len_trim
  .... ! other encapsulated stuff
END MODULE new_house_defs
```

The user defined procedures are added to the existing generic overload set whenever the above module is USEd.

Intrinsic function overloading is used in the VARYING_STRING module. All the intrinsic functions that apply to CHARACTER variables have been overloaded to apply to object of type VARYING_STRING. Thus VARYING_STRING objects may be used in exactly the same way as CHARACTER objects.

Question 55: Complex Arithmetic — Generic Interfaces

Enhance your Integer_Complex_Arithmetic module by adding the overload sets for the following intrinsics:

- ☐ REAL — returns INTEGER result; the real part of a complex number. Eg, REAL(x,y) is x
- ☐ INT — returns INTEGER result; as above.
- ☐ AIMAG — returns INTEGER result; the imaginary part of a complex number.Eg, AIMAG(x,y) is y
- ☐ CONJG — returns INTCOMPLEX result; the conjugate of a complex number. Eg, CONJG(x,y) is (x,-y).
- ☐ ABS — returns REAL result; absolute value of a complex number. Eg, ABS(x,y) is SQRT(x*x+y*y).

so that they accept arguments of type INTCOMPLEX. (REAL and INT should have the same functionality, ie, return the first (non-imaginary) component of the number.)

Demonstrate that the whole module works by USEing it with the following test program (which is available by anonymous ftp from `ftp.liv.ac.uk` in the directory `/pub/f90courses/progs`, filename `IntegerComplex2zProgram.f90`).

```
PROGRAM Testo
 USE Integer_Complex_Arithmetic
  IMPLICIT NONE

   PRINT*, "REAL(3,4)"
   PRINT*, REAL(INTCOMPLEX(3,4))
   PRINT*, "INT(3,4)"
   PRINT*, INT(INTCOMPLEX(3,4))
   PRINT*, "AIMAG(3,4)"
   PRINT*, AIMAG(INTCOMPLEX(3,4))
   PRINT*, "CONJG(3,4)"
   PRINT*, CONJG(INTCOMPLEX(3,4))
   PRINT*, "ABS(3,4)"
   PRINT*, ABS(INTCOMPLEX(3,4))

END PROGRAM Testo
```

23.6 Overloading Operators

Intrinsic operators, such as -, = and *, can be overloaded to apply to all types in a program. This should be encapsulated in a module:

- specify the generic operator symbol (in parentheses) in an INTERFACE OPERATOR statement,
- specify the overload set in a generic interface,
- declare the MODULE PROCEDUREs (FUNCTIONs) which define how the operations are implemented.

These functions have one or two non-optional arguments with INTENT(IN) which correspond to monadic and dyadic operators.

For a dyadic operator the function has two arguments, the first corresponds to the LHS operand and the second to the RHS operand. For example, to define addition between an integer and, say, a representation of a rational number, there would have to be two procedures defined, one for the integer-rational combination and one for for the rational-integer combination. A monadic operator has a corresponding procedure with only one argument.

As usual overloads are resolved by examining the number, type, rank and kind of the arguments so the overload set must be unambiguous. The body of the procedures should be PURE in the sense that they should not alter any arguments, global data or produce output.

23.6.1 Operator Overloading Example

The '*' operator can be extended to apply to the rational number data type as follows:

```
MODULE rational_arithmetic
```

```
 IMPLICIT NONE
 TYPE RATNUM
  INTEGER :: num, den
 END TYPE RATNUM
 INTERFACE OPERATOR (*)
  MODULE PROCEDURE rat_rat, int_rat, rat_int
 END INTERFACE
CONTAINS
 FUNCTION rat_rat(l,r)      ! rat * rat
  TYPE(RATNUM), INTENT(IN) :: l,r
   ...
 END FUNCTION rat_rat
 FUNCTION int_rat(l,r)      ! int * rat
  INTEGER, INTENT(IN)      :: l
  TYPE(RATNUM), INTENT(IN) :: r
  ...
 END FUNCTION int_rat
 FUNCTION rat_int(l,r)      ! rat * int
  TYPE(RATNUM), INTENT(IN) :: l
  INTEGER, INTENT(IN)      :: r
  ...
 END FUNCTION rat_int
END MODULE rational_arithmetic
```

In order for multiplication to be defined between all combinations of integer and rational numbers three new combinations must be defined. Obviously integer-integer multiplication already is defined, the remaining three combinations, (integer-rational, rational-integer and rational-rational) are defined in the procedures `int_rat`, `rat_int` and `rat_rat` respectively. These three new procedures are added to the operator overload set of the `*` operator.

It is not possible to modify the operator precedence, when an operator is overloaded it retains its place in the pecking order of operators.

With the following type declarations in force,

```
USE rational_arithmetic
TYPE (RATNUM) :: ra, rb, rc
```

the following specific function references would be valid:

```
rc = rat_rat(int_rat(2,ra),rb)
```

however, using overloaded intrinsic operators is clearer and should be encouraged:

```
rc = 2*ra*rb
```

And even better still add visibility attributes to force user into good coding:

```
MODULE rational_arithmetic
 TYPE RATNUM
  PRIVATE
```

```
  INTEGER :: num, den
 END TYPE RATNUM
 INTERFACE OPERATOR (*)
  MODULE PROCEDURE rat_rat,int_rat,rat_int
 END INTERFACE
 PRIVATE :: rat_rat,int_rat,rat_int
  ....
```

it is now not possible to use `rat_rat`, etc.

Every intrinsic operator can be overloaded. Some operators exist in both monadic and dyadic forms, for example, + and -, and if they are both to be overloaded they must be handled as separate cases.

Question 56: Complex Arithmetic — Overloading Operators

Modify your `Integer_Complex_Arithmetic` module so that the procedures which perform the five basic arithmetic operations overload the intrinsic operators. Do the same with the procedures that perform unary plus and unary minus.

23.7 Defining New Operators

As well as overloading existing intrinsic operators, new operators can be defined. They follow the dot notation of Fortrans non-symbolic operators, such as `.AND.` or `.OR.`. They have the form,

.< *name* >.

where < *name* > can only contain letters. (Operator names can be the same as object names with no conflict.)

For all operator definitions, there must be one or two `INTENT(IN)` arguments corresponding to the definition of a monadic or dyadic operator. As with generic procedure interfaces there must be a unique set of overloads, no duplicity is allowed. Of all operators, a user-defined monadic has the highest precedence and a user-defined dyadic has the lowest.

It is not possible to redefine the meanings of intrinsic operators (such as `.NOT.` and `.GE.`).

23.7.1 Defined Operator Example

Consider the following module containing the definition of the `.TWIDDLE.` operator in both monadic and dyadic forms,

```
MODULE twiddle_op
 INTERFACE OPERATOR (.TWIDDLE.)
  MODULE PROCEDURE itwiddle, iitwiddle
 END INTERFACE ! (.TWIDDLE.)
CONTAINS
 FUNCTION itwiddle(i)
  INTEGER itwiddle
  INTEGER, INTENT(IN) :: i
```

```
      itwiddle = -i*i
     END FUNCTION
     FUNCTION iitwiddle(i,j)
      INTEGER iitwiddle
      INTEGER, INTENT(IN) :: i,j
      iitwiddle = -i*j
     END FUNCTION
    END MODULE
```

The following program

```
    PROGRAM main
     USE twiddle_op
     print*, 2.TWIDDLE.5, .TWIDDLE.8, .TWIDDLE.(2.TWIDDLE.5), &
              .TWIDDLE.2.TWIDDLE.5
    END PROGRAM
```

produces

```
    -10 -64 -100 20
```

Note the INTENT attribute which is mandatory and that the END INTERFACE statement cannot contain the operator name.

The above example highlights how operator precedence influences the results of .TWIDDLE.(2.TWIDDLE.5) and .TWIDDLE.2.TWIDDLE.5. The parentheses modify the execution order so that in the first case 2.TWIDDLE.5 is evaluated first; the second expression uses intrinsic operator precedence meaning that the first expression to be evaluated is the monadic occurrence of TWIDDLE.

Question 57: Series and Parallel Resistance, Defined Operators

Define a module called LECCY_OPS containing two operators .PARALLEL. and .SERIES. which, given two default REAL resistance values as operands will deliver the resistance obtained by connecting them in parallel or series.

For series resistance:

$$R = R_1 + R_2$$

and for parallel:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$$

Use the following test program (which is available by anonymous ftp from ftp.liv.ac.uk in the directory /pub/f90courses/progs, filename DefinedOperatorResistanceQuestion.f90 to demonstrate correctness of the module.

```
    PROGRAM Testo
     USE Resistance
     Print*, "10.0 .SERIES. 50.0", 10.0 .SERIES. 50.0
     Print*, "10.0 .PARALLEL. 50.0", 10.0 .PARALLEL. 50.0
    END PROGRAM Testo
```

23.7.2 Precedence

There is an intrinsic ordering of operators (see 10.7). Every operator in Fortran 90 has a precedence; the operator with the highest precedence is combined with its operand(s) first. User defined monadic operators have the *highest* precedence of all operators, and user defined dyadic operators have the *lowest* precedence. User defined operators which have the same name as intrinsic operators retain the same precedence.

For example, assume that there are two user defined operators (one monadic the other dyadic), then the expression,

```
.TWIDDLE.e**j/a.TWIDDLE.b+c.AND.d
```

is equivalent to

```
(((.TWIDDLE.e)**j)/a).TWIDDLE.((b+c).AND.d)
```

The monadic `.TWIDDLE.` operator is combined with its operand first whereas the dyadic `.TWIDDLE.` will be the last operator to be considered.

23.8 User-defined Assignment

Assignment between two objects of intrinsic type and between the same user defined type is intrinsically defined, however, assignment between two different user-defined types or between an intrinsic and a user-defined type must be explicitly programmed.

Assignment overloading is done in much the same way as operator overloading except that the procedure which defines the assignment process is a `SUBROUTINE` with two arguments. The body of the procedure should be `PURE` in the sense that it should not alter any arguments, global data or produce output. Specification of the subroutines which describe how the assignment is performed are given in an `INTERFACE ASSIGNMENT` block, these subroutines must have the following properties:

- the first argument is the variable which receives the assigned value, the LHS. It must have the `INTENT(OUT)` attribute;
- the second actual argument is the expression whose value is converted and assigned to the result the RHS expression. The corresponding dummy argument must have the `INTENT(IN)` attribute.

23.8.1 Defined Assignment Example

Take the rational numbers example, defining the mathematical operators (+, -, *, / and **) is of little use if the assignment operator (=) is undefined. A module should be written which specifies an interface giving the overload set for the assignment operator, the rules should be explicitly defined in module procedures for assignment between:

- LHS `REAL`; RHS `RATNUM`
- LHS `RATNUM`; RHS `INTEGER`

The following interface specifies which procedures should be employed for assignment involving rational numbers:

```
INTERFACE ASSIGNMENT (=)
 MODULE PROCEDURE rat_ass_int, real_ass_rat
END INTERFACE
PRIVATE rat_ass_int, real_ass_rat
```

The specific procedures will be given in the `CONTAINS` block of the module:

```
SUBROUTINE rat_ass_int(var, exp)
 TYPE (RATNUM), INTENT(OUT) :: var
 INTEGER, INTENT(IN) :: exp
  var%num = exp
  var%den = 1
END SUBROUTINE rat_ass_int
SUBROUTINE real_ass_rat(var, exp)
 REAL, INTENT(OUT) :: var
 TYPE (RATNUM), INTENT(IN) :: exp
  var = exp%num / exp%den
END SUBROUTINE real_ass_rat
```

the body of each subroutine *must* contain an assignment to the first argument.

Wherever the module is used the following is valid:

```
ra = 50
i = rb*rc
```

If `i` is declared as an `INTEGER` and all other objects are of type `RATNUM`, then the first assignment follows the rules that are laid out in the procedure `rat_ass_int` and the second accesses `real_ass_rat`.

Question 58: Complex Arithmetic — Overloading the Assignment Operator

Modify your `Integer_Complex_Arithmetic` module so that the assignment operator is overloaded to allow objects of type `INTEGER` and `REAL` to be assigned to `INTCOMPLEX` objects. (`REAL` values should be *truncated* before use.)

23.8.2 Semantic Extension Example

The visibility specifiers can be applied to all objects including type definitions, procedures and operators. It is a particularly good idea to deny access to the module procedures which define what overloading a certain operator actually means. For example, consider the rational arithmetic module, we clearly want the `*` operator to be visible by use association (`PUBLIC`) but there is no advantage to be gained by allowing `rat_rat`, `int_rat` and `rat_int` to be visible to the user. An analogous situation arises for the assignment operator:

For example,

```
    MODULE rational_arithmetic
     IMPLICIT NONE

     PUBLIC :: OPERATOR (*)
     PUBLIC :: ASSIGNMENT (=)

     TYPE RATNUM
      PRIVATE
      INTEGER :: num, den
     END TYPE RATNUM

     TYPE, PRIVATE :: INTERNAL
      INTEGER :: lhs, rhs
     END TYPE INTERNAL

     INTERFACE OPERATOR (*)
      MODULE PROCEDURE rat_rat, int_rat, rat_int
     END INTERFACE ! OPERATOR (*)

     INTERFACE ASSIGNMENT (=)
      MODULE PROCEDURE rat_ass_int, real_ass_rat
     END INTERFACE ! ASSIGNMENT (=)

     PRIVATE rat_rat, int_rat, rat_int, rat_ass_int, real_ass_rat

    CONTAINS

     SUBROUTINE rat_ass_int(var, exp)
       ... ! and so on
     END SUBROUTINE rat_ass_int

! etc

    END MODULE rational_arithmetic
```

The type INTERNAL is only accessible from within the module.

The following entities are PUBLIC: RATNUM type (no internal access) *, =; the rest are PRIVATE: rat_rat, int_rat, rat_int, rat_ass_int, real_ass_rat and the type INTERNAL. Note that the visibility statements for named entities must come after their declaration (after the MODULE PROCEDURE statements).

To build a complete "class" should also add:

- constructors for RATNUM: init_RATNUM
- output procedure for RATNUM: Print
- overloaded intrinsics: REAL, CEILING, FLOOR, etc

23.8.3 Yet Another Example

Define a module,

```
MODULE circles
  IMPLICIT NONE
  PRIVATE
  TYPE, PUBLIC :: POINT
   REAL :: x,y
  END TYPE POINT
  TYPE, PUBLIC :: CIRCLE
   TYPE(POINT) :: centre
   REAL        :: radius
  END TYPE CIRCLE
  INTERFACE OPERATOR (.CROSSES.)
   MODULE PROCEDURE circle_crosses
  END INTERFACE
  PUBLIC OPERATOR (.CROSSES.)
CONTAINS
  LOGICAL FUNCTION circle_crosses( c1,c2 )
   TYPE(CIRCLE), INTENT(IN) :: c1,c2
   REAL :: d
   d = (c1%centre%x - c2%centre%x)**2 + &
       (c1%centre%y - c2%centre%y)**2
   IF( d < (c1%radius + c2%radius)**2 .OR.   &
       d > (c1%radius - c2%radius)**2 ) THEN
     circle_crosses = .TRUE.
   ELSE
     circle_crosses = .FALSE.
   END IF
  END FUNCTION circle_crosses
END MODULE circles
```

This is an example demonstrating the visibility attributes, operator definition, encapsulation, and derived type definitions.

- internal components of POINT and CIRCLE are not visible,
- the operator .CROSSES. can be seen but the procedure circle_crosses cannot.

Program calls module,

```
PROGRAM Example
  USE circles
   TYPE(CIRCLE), DIMENSION(8) :: c
   INTEGER :: Xings=0
    ... ! read in data c(1:8)
    DO I= 1,8
     DO J = 1, I-1
      IF( c(I) .CROSSES. c(J) ) THEN
        PRINT*, I, ' crosses ', J
        Xings = Xings + 1
      END IF
     END DO
    END DO
  PRINT*, "Num circles which cross = ", Xings
END PROGRAM
```

When the module appears in a USE statement the type definition and operator can safely be used, however, any reference to the type components or the procedure circle_crosses would be an error.

Question 59: Complex Arithmetic — Accessibility control

Modify your Integer_Complex_Arithmetic module to take advantage of the accessibility statements.

Use accessibility statements to restrict visibility of the module procedures, and type structure. Write a constructor function and I / O subroutines for the INTCOMPLEX type which have the following interfaces:

- construction

```
TYPE(INTCOMPLEX) FUNCTION Setup_INTCOMPLEX(i1,i2)
 INTEGER, INTENT(IN) :: i1, i2
END FUNCTION Setup_INTCOMPLEX
```

- output,

```
SUBROUTINE Put_INTCOMPLEX(ic)
 TYPE(INTCOMPLEX), INTENT(IN) :: ic
END SUBROUTINE Put_INTCOMPLEX
```

- input

```
SUBROUTINE Get_INTCOMPLEX(ic)
 TYPE(INTCOMPLEX), INTENT(OUT) :: ic
END SUBROUTINE Get_INTCOMPLEX
```

Demonstrate that the whole module works by USEing it with the following test program (which is available by anonymous ftp from ftp.liv.ac.uk in the directory /pub/f90courses/progs, filename IntegerComplex3cProgram.f90).

```
PROGRAM Testo
 USE Integer_Complex_Arithmetic
 IMPLICIT NONE
 TYPE(INTCOMPLEX) :: var1, var2, ans

 var1 = 3
 PRINT*, "var1 = 3"
 CALL Put_INTCOMPLEX(var1)
 PRINT*, ""

 var1 = 6.0
 PRINT*, "var1 = 6.0"
 CALL Put_INTCOMPLEX(var1)
 PRINT*, ""

 var1 = 6.2
 PRINT*, "var1 = 6.2"
 CALL Put_INTCOMPLEX(var1)
```

```
    PRINT*, ""

    var1 = 6.6
    PRINT*, "var1 = 6.6"
    CALL Put_INTCOMPLEX(var1)
    PRINT*, ""

    var1 = Setup_INTCOMPLEX(1,2)
    var2 = Setup_INTCOMPLEX(3,4)

    PRINT*, "(1,2)+(3,4)"
    ans = var1 + var2
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""

    PRINT*, "(1,2)-(3,4)"
    ans = var1 - var2
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""

    PRINT*, "(1,2)/(3,4)"
    ans = var1 / var2
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""

    PRINT*, "(3,4)/(3,4)"
    ans = var2 / var2
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""

    PRINT*, "(3,4)/(1,2)"
    ans = var2 / var1
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""

    PRINT*, "(1,2)*(3,4)"
    ans = var1 * var2
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""

    PRINT*, "(1,2)**3"
    ans = var1 ** 3
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""

    PRINT*, "+(1,2)"
    ans = +var1
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""

    PRINT*, "-(1,2)"
    ans = -var1
    CALL Put_INTCOMPLEX(ans)
    PRINT*, ""
```

```
     PRINT*, "Type in the two INTCOMPLEX components"
     CALL Get_INTCOMPLEX(var1)
     PRINT*, ""
     PRINT*, "This is what was typed in"
     CALL Put_INTCOMPLEX(var1)
     PRINT*, ""

     PRINT*, "Your number/(3,4)"
     ans = var1 / var2
     CALL Put_INTCOMPLEX(ans)
     PRINT*, ""

! Intrinsics

     PRINT*, "REAL(3,4)"
     PRINT*, REAL(var2)
     PRINT*, ""

     PRINT*, "INT(3,4)"
     PRINT*, INT(var2)
     PRINT*, ""

     PRINT*, "AIMAG(3,4)"
     PRINT*, AIMAG(var2)
     PRINT*, ""

     PRINT*, "CONJG(3,4)"
     CALL Put_INTCOMPLEX(CONJG(var2))
     PRINT*, ""

     PRINT*, "ABS(3,4)"
     PRINT*, ABS(var2)

    END PROGRAM Testo
```

23.8.4 More on Object Oriented Programming *by J. S. Morgan*

Many Fortran programmers may not realise that it is possible to implement a C++ style of object-oriented programming (OOP) methodology using the features already extant in Fortran 90. In some areas Fortran 90 OO programming is a little more verbose and arguably less elegant than C++ but nevertheless OO Fortran 90 programs are quite easily implemented.

This section is based on a paper, 'How to express C++ Concepts in Fortran 90'

`http://www.cs.rpi.edu/ nortonc/oof90.html`

by V. K. Dec yk, C. D. Norton, and B. K. Szymanski at the Jet Propulsion Laboratory, California Institute of Technology. For those wishing to understand the essentials of OOP the article is a good read.

They demonstrated OO programming using a database example. Here I have used an example related to graphics to illustrate the same principles.

Basically a class in OOP terms is a type (in Fortran 90 terms) which has PRIVATE components together with a constructor for creating instances of a class, a destructor which destroys instances of a class, and a series of methods which perform various manipulations of instances of the class. In the following I have ignored destructors to keep the code fragments down to a minimum.

Thus to create a class such as a point class (for use in graphics applications) a Fortran 90 module can be used. For example:

```
MODULE point_class

   PRIVATE

   TYPE point
     PRIVATE
     REAL :: x,y   ! coordinates of the point
   ENDTYPE point

   INTERFACE new
     MODULE PROCEDURE new_this
   END INTERFACE

   INTERFACE draw
     MODULE PROCEDURE draw_this
   END INTERFACE

   PUBLIC point, new, draw

   CONTAINS

   SUBROUTINE draw_this(this)
     TYPE(point) :: this
     WRITE(*,*) ' Drawing point at', this%x, this%y
   END SUBROUTINE draw_this

   SUBROUTINE new_this(this,x,y)
     TYPE(point) :: this
     REAL        :: x,y
     this%x = x
     this%y = y
   END SUBROUTINE new_this

END MODULE point_class
```

Note that the PRIVATE statement makes all names in the module private by default. The PRIVATE statement in the TYPE definition makes the components of the type private. The public statement makes only those names which the user is allowed access to visible to the user of the module.

The use of the generic names for new, draw, etc. allows us to then use the same names for the methods (procedures in Fortran 90 parlance) of other classes. It allows new classes to inherit the types and procedures of other classes.

Thus a line class can be created as:

```
MODULE line_class

   USE point_class    ! inherit the public entities of point_class

   PRIVATE

   TYPE line
     PRIVATE
     TYPE(POINT) :: p1,p2
   ENDTYPE line

   INTERFACE new      ! overrides new for this object
     MODULE PROCEDURE new_this
   END INTERFACE

   INTERFACE draw
     MODULE PROCEDURE draw_this
   END INTERFACE

   PUBLIC line,new,draw

   CONTAINS

   SUBROUTINE draw_this(this)
     TYPE(line) :: this
     WRITE(*,*) ' Drawing line',this
   END SUBROUTINE draw_this

   SUBROUTINE new_this(this,p1,p2)
     TYPE(line) :: this
     TYPE(point):: p1,p2
     this%p1 = p1
     this%p2 = p2
   END SUBROUTINE new_this

END MODULE line_class
```

Note that the line class is a super-class of the point class. Inheritance in OO programming is usually associated with sub-classing.

We can then write statements such as

```
  TYPE(point)          :: p1,p2
  TYPE(line)           :: a_line

!--- create objects
  CALL new(p1, 5.,10.)
  CALL new(p2,10.,20.)
```

```
  CALL new(a_line,p1,p2)


!--- draw them
  CALL draw(p1)
  CALL draw(a_line)
```

Frequently in graphics programs it is necessary to create structures consisting of objects of different classes. For example, a drawing can be modelled as an array whose elements can contain points or lines (or other graphical objects).

To achieve this in C++ it is possible to use its so-called dynamic binding features which are accessed via virtual functions. Fortran 90 does not support these features directly and so a little extra work is needed.

The approach taken is to create, effectively, a super-type which handles the housekeeping associated with manipulating the different sub-types and keeps this hidden from the user.

Thus we create a `graphic_object_class` which can manifest itself in instances of points or lines. Thus:

```
MODULE graphic_object_class

   USE point_class
   USE line_class

   PRIVATE

   TYPE graphic_object
     PRIVATE
     TYPE(point) ,POINTER :: pp
     TYPE(line)  ,POINTER :: lp
   ENDTYPE graphic_object

   INTERFACE create_graphic_object
     MODULE PROCEDURE create_point, create_line
   END INTERFACE

   PUBLIC graphic_object, create_graphic_object, point, line, new, &
          draw_graphic_object


   CONTAINS

   SUBROUTINE create_point(this,p)
     TYPE(graphic_object) :: this
     TYPE(point),TARGET :: p
     this%pp => p
     NULLIFY(this%lp)
   END SUBROUTINE create_point

   SUBROUTINE create_line(this,l)
     TYPE(graphic_object) :: this
     TYPE(line),TARGET :: l
```

```
    NULLIFY(this%pp)
    this%lp => l
  END SUBROUTINE create_line

  SUBROUTINE draw_graphic_object(this)

    TYPE(graphic_object) :: this

    IF(ASSOCIATED(this%pp))THEN
       CALL draw(this%pp)
    ELSEIF(ASSOCIATED(this%lp))THEN
       CALL draw(this%lp)
    ENDIF

  END SUBROUTINE draw_graphic_object

END MODULE graphic_object_class
```

The above uses a type with pointers to the different objects. When an object of a type is created the appropriate pointer is pointed to that object. The other pointer is NULL. At runtime the draw_graphic_object checks which object is being referenced and calls the appropriate routine.

Finally, putting all this together, we can now write programs such as the following.

```
PROGRAM use_graphic_objects

  USE graphic_object_class

  IMPLICIT NONE

  TYPE(graphic_object) :: g1,g2
  TYPE(point)          :: a_point,p1,p2,c
  TYPE(line)           :: a_line
  REAL                 :: x,y,r

!--- set some specific values
  x = 5; y=10; r = 45

!--- create objects
  CALL new(a_point,x,y)  ! OO equivalent : a_point = new a_point(x,y);
  CALL new(p1,3.,6.)
  CALL new(p2,10.,20.)
  CALL new(c ,100.,150.)
  CALL new(a_line,p1,p2) ! OO equivalent : a_line = new a_line(p1,p2);

!--- create generic objects
  CALL create_graphic_object(g1, a_point)
  CALL create_graphic_object(g2, a_line)

!--- draw them
  CALL draw_graphic_object(g1) ! OO equiv. : g1.draw_graphic_object();
  CALL draw_graphic_object(g2)
```

```
END PROGRAM use_graphic_objects
```

The above, I hope, has demonstrated how, in Fortran 90, by using a well disciplined methodology, and including a disciplined naming scheme, it is possible to mimic the object oriented style of programming to quite a large degree.

The main drawback with the above code can be seen when one tries to add an extra object type (class) to the scheme. It is fairly straightforward to

create a new object such as a circle by writing a module using one of the other modules as a template. However, adding in the necessary code to the `use_graphic_objects` module is much more tricky since changes have to be made in several different places and this can be error prone. This is where the expressibility of a true object-oriented language scores over Fortran 90.

23.9 Semantic Extension Modules

In summary, modules can be defined which provide a *semantic extensions* to the language. Such modules require:

- ☐ a mechanism for defining new types.
- ☐ a method for defining operations on those types.
- ☐ a method of overloading the operations so users can use them in a natural way.
- ☐ a way of encapsulating all these features in such a way that users can access them as a combined set.
- ☐ details of underlying data representation in the implementation of the associated operations to be kept hidden.

Semantic extension allows the user to express a problem in terms of the natural objects and operations relating to that problem. This philosophy is used in so-called object oriented programming. Fortran 90 does provide many features desirable for OOP but falls short of providing all of them.

The `VARYING_STRING` module is a good example of semantic extension, it has has 96 module procedures which are used to extend 25 generic intrinsic procedures, 6 intrinsic relational operators and the concatenation operator. Most entities in the `VARYING_STRING` module are `PRIVATE` except the type and intrinsic entities such as the extended operators and functions.

23.9.1 Semantic Extension Example

The use of semantic extension modules is not restricted to types which are analogous to the intrinsic types such other forms of numbers. The technique can be used to build extensions in a vast range of areas. The following example illustrates this. It defines a module that provides facilities for handling straight line segments as single objects. Lines are assumed to be defined by the x and y coordinates of their endpoints. For simplicity only two functions are defined i.e. a function to create a line given its endpoints, and a logical binary operator which can be used to check whether two lines intersect. The latter is implemented as a function with two arguments of type `LINE` returning a logical result. However, it would be possible to build up a complete module defining a wide variety of facilities for manipulating lines.

```
MODULE Lines

 IMPLICIT NONE

 !-- Definition of derived type for lines ------------!
 TYPE LINE
  PRIVATE
  REAL :: X1, Y1, X2, Y2
  ! (X1,Y1) and (X2,Y2) are the endpoints of the line
 END TYPE LINE

 !-- Interface Definitions for Generic names ---------!
 INTERFACE Create_Line
  ! This Procedure creates a line between two points.
  ! The arguments of the function will be (X1,Y1,X2,Y2)
  ! and they can be either all REAL or all INTEGER.
  MODULE PROCEDURE Real_Line_Constructor, Integer_Line_Constructor
 END INTERFACE

 INTERFACE OPERATOR(.CROSSES.)
  !-- This operator tests if two lines intersect.
  MODULE PROCEDURE Line_Line_Crosses
 END INTERFACE

CONTAINS

 !-- Definitions of specific procedure bodies --------!
 FUNCTION Integer_Line_Constructor(X1,Y1,X2,Y2)
  INTEGER     :: X1,Y1,X2,Y2
  TYPE(LINE)  :: Integer_Line_Constructor
  Integer_Line_Constructor%X1 = X1
  Integer_Line_Constructor%Y1 = Y1
  Integer_Line_Constructor%X2 = X2
  Integer_Line_Constructor%Y2 = Y2
 END FUNCTION Integer_Line_Constructor

 FUNCTION Real_Line_Constructor(X1,Y1,X2,Y2)
  REAL                :: X1,Y1,X2,Y2
  TYPE(LINE)  :: Real_Line_Constructor
  Integer_Line_Constructor%X1 = X1
  Integer_Line_Constructor%Y1 = Y1
  Integer_Line_Constructor%X2 = X2
  Integer_Line_Constructor%Y2 = Y2
 END FUNCTION Real_Line_Constructor

 FUNCTION Line_Line_Crosses(L2,L2)
  TYPE(LINE)  :: L1, L2
  LOGICAL     :: Line_Line_Crosses
  REAL        :: X1L1, X2L1, X1L2, Y1L2, X2L2, Y2L2, m, c
  REAL        :: Cth, Sth
  REAL, PARAMETER :: tol = 1.0E-03    ! a small number
  ! angle L1 makes with x-axis
  IF ((L1%X1 - L1%X2) < tol) THEN
   th = pi/2
```

```
      ELSE
       th = ATAN ((L1%Y2-L1%Y1)/(L1%X2-L1%X1))
      END IF
      Cth = COS(th)
      Sth = SIN(th)
      ! Find x coordinates of L1 in new axis system
      X1L1 = L1%X1*Cth - L1%Y1*Sth
      X2L1 = L1%X2*Sth - L1%Y2*Cth
      ! Find coords of endpoints of L2 in new axis system
      X1L2 = L2%X1*Cth - L2%Y1*Sth
      Y1L2 = L2%X1*Sth - L2%Y1*Cth
      X2L2 = L2%X2*Cth - L2%Y2*Sth
      Y2L2 = L2%X2*Sth - L2%Y2*Cth
      ! find m and c for equation of new line
      Line_Line_Crosses = .FALSE.
      IF ((X2L2 - X1L2) > tol) THEN  ! lines not parallel
       m = (Y2L2-Y1L2)/(X2L2-X1L2)
       c = Y2L2 - m * X2L2
       xcross = -c/m
       ! Test if crossing point is inside line segment
       IF ( X1L1 <= xcross .AND. X2L1 >= xcross) THEN
        Line_Line_Crosses = .TRUE.
       END IF
      END IF
     END FUNCTION Line_Line_Crosses

    END MODULE Lines
```

The module can be used in a program such as the one below,

```
    PROGRAM Using_Lines

      USE Lines

      IMPLICIT NONE

      TYPE(LINE) :: L1, L2  !-- Declare two objects of type Line
      ! Create two lines, one using the INTEGER interface and one
      ! using the REAL interface
      L1 = Create_Line(50,50,100,100)
      L2 = Create_Line(20.4,12.56,150.0,265.75)
      IF (L1.CROSSES.L2) THEN
       PRINT*, 'line 1 crosses line 2'
      ELSE
       PRINT*, 'line 1 does not cross line 2'
      END IF
     END PROGRAM
```

We could modify the module by changing the internal representation of a line, because of the judiciously chosen object accessibility the change in representation is totally hidden from the user,

```
    MODULE Lines
```

```
    IMPLICIT NONE

    TYPE LINE
      PRIVATE
      REAL :: X1, Y1, SLOPE, LENGTH
      ! (X1,Y1) is the start of the line
    END TYPE LINE
    !-- Interface Definitions for Generic names ---------!

    INTERFACE Create_Line
      ! This Procedure creates a line between two points.
      ! The arguments of the function will be (X1,Y1,X2,Y2)
      ! and they can be either all REAL or all INTEGER.
      MODULE PROCEDURE Real_Line_Constructor, Integer_Line_Constructor
    END INTERFACE

    INTERFACE OPERATOR(.CROSSES.)
      !-- This operator tests if two lines intersect.
      MODULE PROCEDURE Line_Line_Crosses
    END INTERFACE

  CONTAINS

    !-- Definitions of specific procedure bodies --------!

    FUNCTION Integer_Line_Constructor(X1,Y1,X2,Y2)
      INTEGER     :: X1,Y1,X2,Y2
      TYPE(LINE)  :: Line_Constructor
      Line_Constructor%X1 = X1
      Line_Constructor%Y1 = Y1
      Line_Constructor%slope = REAL(Y2-Y1)/(X2-X1)
      Line_Constructor%length = SQRT(REAL((Y2-Y1)**2+(X2-X1)**2))
    END FUNCTION Integer_Line_Constructor

    FUNCTION Real_Line_Constructor(X1,Y1,X2,Y2)
      REAL                :: X1,Y1,X2,Y2
      TYPE(LINE)  :: Line_Constructor
      Line_Constructor%X1 = X1
      Line_Constructor%Y1 = Y1
      Line_Constructor%slope = (Y2-Y1)/(X2-X1)
      Line_Constructor%length = SQRT((Y2-Y1)**2+(X2-X1)**2)
    END FUNCTION Real_Line_Constructor
    ...(rewrite of the whole module to use the new structure)
  END MODULE Lines
```

The main program given above could use either of the two Lines modules without modification.

Module 10: Parametrised Intrinsic Types Plus

24 Complex Data Type

This intrinsic data type has the same precision as default `REAL` and has comparable properties to other intrinsic data types.

A `COMPLEX` object is made up from two default `REAL` cells and is declared as follows:

```
COMPLEX :: z, j, za(1:100)
```

Symbolic constants are expressed as a co-ordinate pair:

```
COMPLEX, PARAMETER :: i = (0.0,1.0)
```

Real-valued literals and symbolic constants, and complex valued literals and symbolic constants can all be used in a `COMPLEX` object initialisation statement (for example, `PARAMETER` statement), but it is not permissible to allow a constructed value containing real-valued symbolic constants as components. So,

```
INTEGER, PARAMETER :: a1 = 1, a2 = 2
COMPLEX, PARAMETER :: i = (1.0,2.0)
COMPLEX, PARAMETER :: ii = i
COMPLEX, PARAMETER :: iii = a1
```

is OK, but

```
COMPLEX, PARAMETER :: iv = (a1,a2)
```

is no good. The `CMPLX` constructor cannot be used in initialisation expressions.

Complex values can be constructed elsewhere using the `CMPLX` intrinsic function,

```
z = CMPLX(x,y)
```

This format must be used if the RHS is not a literal constant. It is recommended that the `CMPLX` intrinsic be used even when the RHS *is* a literal value (except in `PARAMETER` statements) as this makes the program more consistent and highlights the type conversion,

```
z = CMPLX(3,6.9)
```

Type coercion behaves exactly as for the REAL data type, for example, if a COMPLEX literal contained integers, (1,1), these would be promoted to REAL values before assignment to the COMPLEX variable.

Complex expressions can be used in the same way as other types

```
REAL :: x; COMPLEX :: a, b, c,
 ...
a = x*((b+c)*CMPLX(0.1,-0.1))
b = 1
```

The real value x will be promoted to the complex value CMPLX(x,0). b will be set to CMPLX(1.0,0). If necessary all other data types are promoted to complex values. When a non-complex value is coerced a 0.0 is placed in the complex part of the coordinate pair.

24.1 Complex Intrinsics

The following intrinsic are relevant to the COMPLEX data type,

- AIMAG(z) — imaginary part of a complex number.

  When supplied with a COMPLEX argument, this function returns the imaginary part, for example, AIMAG(0.9,0.2) is 0.2.

- REAL or DBLE — real part of a complex number.

  When applied to a complex argument returns the real part of the complex value as a REAL or DOUBLE PRECISION number. For example, DBLE(0.9,0.2) is 0.9D0.

- CONJG(z) — conjugate of a complex number.

  When applied to a complex argument returns the complex conjugate of a complex number, for example CONJG(CMPLX(x,y)) equals CMPLX((x,-y)).

- ABS(z) — absolute value of a complex number.

  When applied to a complex argument returns the real valued absolute value, for example, ABS(CMPLX(x,y)) equals SQRT(x**2 + y**2).

- SIN etc. — mathematical functions.

  The set of mathematical intrinsic functions is defined generically for complex arguments. The result type will generally be complex.

  For example, SIN(z) will calculate the complex function result as would be expected, if $\mathbf{z} = \mathbf{x} + i\mathbf{y}$, then

  - real part of $\sin(\mathbf{z}) = \sin(\mathbf{x})\cosh(\mathbf{y})$
  - imaginary part of $\sin(\mathbf{z}) = \cos(\mathbf{x})\sinh(\mathbf{y})$

  As a rule of thumb, if the mathematical function can be applied to complex arguments then it will be implemented as such in Fortran 90.

Question 60: Complex Arithmetic — Mixing Types

Enhance your Integer_Complex_Arithmetic module by adding the overload sets for operations between REAL and INTCOMPLEX operands and INTEGER and INTCOMPLEX operands, these should be implemented for +, -, (dyadic) * and /. Clearly the result of an operation between a REAL and INTCOMPLEX expression and an INTEGER and INTCOMPLEX expression should be of type INTCOMPLEX. (REAL numbers should be *truncated* before use.)

Recall that if a is integer or real valued then,

$$a + (x + y\mathbf{i}) = (a + x) + y\mathbf{i}$$

and,

$$a - (x + y\mathbf{i}) = (a - x) - y\mathbf{i}$$

and,

$$a \times (x + y\mathbf{i}) = (a \times x) + (a \times y)\mathbf{i}$$

and,

$$\frac{a}{(x + y\mathbf{i})} = \left(\frac{a \times x}{x^2 + y^2}\right) - \left(\frac{a \times y}{x^2 + y^2}\right)\mathbf{i}$$

so

$$0.99 \times (3 + 4\mathbf{i}) = 0 + 0\mathbf{i}$$

and

$$1.0 \times (3 + 4\mathbf{i}) = 3 + 4\mathbf{i}$$

Demonstrate that the whole module works by USEing it in the following test program (which is available by anonymous ftp from ftp.liv.ac.uk in the directory /pub/f90courses/progs, filename IntegerComplex5Program.f90).

```
PROGRAM Testo
 USE Integer_Complex_Arithmetic
 IMPLICIT NONE
 TYPE(INTCOMPLEX) :: var1, var2, ans

 var1 = 3
 PRINT*, "var1 = 3"
 CALL Put_INTCOMPLEX(var1)

 var1 = 5.99
 PRINT*, "var1 = 5.99"
```

```
CALL Put_INTCOMPLEX(var1)

var1 = 6.01
PRINT*, "var1 = 6.01"
CALL Put_INTCOMPLEX(var1)

var1 = Setup_INTCOMPLEX(1,2)
var2 = Setup_INTCOMPLEX(3,4)

PRINT*, "(1,2)+(3,4)"
ans = var1 + var2
CALL Put_INTCOMPLEX(ans)

PRINT*, "(1,2)-(3,4)"
ans = var1 - var2
CALL Put_INTCOMPLEX(ans)

PRINT*, "(1,2)/(3,4)"
ans = var1 / var2
CALL Put_INTCOMPLEX(ans)

PRINT*, "(3,4)/(3,4)"
ans = var2 / var2
CALL Put_INTCOMPLEX(ans)

PRINT*, "(3,4)/(1,2)"
ans = var2 / var1
CALL Put_INTCOMPLEX(ans)

PRINT*, "(1,2)*(3,4)"
ans = var1 * var2
CALL Put_INTCOMPLEX(ans)

PRINT*, "(1,2)**3"
ans = var1 ** 3
CALL Put_INTCOMPLEX(ans)

PRINT*, "+(1,2)"
ans = +var1
CALL Put_INTCOMPLEX(ans)

PRINT*, "-(1,2)"
ans = -var1
CALL Put_INTCOMPLEX(ans)

PRINT*, "Type in the two INTCOMPLEX components"
CALL Get_INTCOMPLEX(var1)
PRINT*, "This is what was typed in"
CALL Put_INTCOMPLEX(var1)

PRINT*, "Your number/(3,4)"
ans = var1 / var2
CALL Put_INTCOMPLEX(ans)
```

```
! Intrinsics

     PRINT*, "REAL(3,4)"
     PRINT*, REAL(var2)
     PRINT*, "INT(3,4)"
     PRINT*, INT(var2)
     PRINT*, "AIMAG(3,4)"
     PRINT*, AIMAG(var2)
     PRINT*, "CONJG(3,4)"
     CALL Put_INTCOMPLEX(CONJG(var2))
     PRINT*, "ABS(3,4)"
     PRINT*, ABS(var2)

! REAL | INTEGER .OP. INTCOMPLEX

     PRINT*, "2+(3,4)"
     ans =  2 + var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2-(3,4)"
     ans =  2 - var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2*(3,4)"
     ans =  2 * var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2/(3,4)"
     ans =  2 /var2
     CALL Put_INTCOMPLEX(ans)

     var1 = Setup_INTCOMPLEX(1,2)
     PRINT*, "4/(1,2)"
     ans =  4/var1
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2.5+(3,4)"
     ans =  2.5 + var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2.5-(3,4)"
     ans =  2.5 - var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2.5*(3,4)"
     ans =  2 * var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2.5/(3,4)"
     ans =  2.5 /var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "4.7/(1,2)"
     ans =  4.7/var1
```

```
     CALL Put_INTCOMPLEX(ans)


     PRINT*, "-2.5+(3,4)"
     ans =  -2.5 + var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-2.5-(3,4)"
     ans =  -2.5 - var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-2.5*(3,4)"
     ans =  -2.5 * var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-2.5/(3,4)"
     ans =  -2.5 /var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-4.7/(1,2)"
     ans =  -4.7/var1
     CALL Put_INTCOMPLEX(ans)

! INTCOMPLEX .OP. INTEGER | REAL

     PRINT*, "(3,4)+2"
     ans =  var2 + 2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "3,4)-2"
     ans =  var2-2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4) * 2"
     ans =  var2 * 2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)/2"
     ans =  var2/2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(1,2)/4"
     ans =  var1/4
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)+2.5"
     ans =  var2+2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)-2.5"
     ans =  var2-2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)*2.5"
```

```
     ans =   var2*2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)/2.5"
     ans =  var2/2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(1,2)/4.7"
     ans =  var1/4.7
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-(3,4)-2.5"
     ans =   -var2-2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)*(-2.5)"
     ans =  var2*(-2.5)
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "0.99 * (3,4)"
     ans = 0.99 *var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "1.0 * (3,4)"
     ans = 1.0 *var2
     CALL Put_INTCOMPLEX(ans)

    END PROGRAM Testo
```

25 Parameterised Intrinsic Types

FORTRAN 77 had a problem with numeric portability, the precision (and exponent range) of a data type on one processor would not necessarily be the same on another. For example, a default REAL may be able to support numbers up to (say) 10^{68} on one machine and up to 10^{136} on another. One of the goals of Fortran 90 was to overcome this portability problem. Fortran 90 implements a portable precision selecting mechanism, it supports types which can be parameterised by a kind value (an integer). The kind value is used to select a representation model for a type and, for numeric types, can be specified in terms of the required precision. A processor can support different precisions (or representations) of INTEGER, REAL and COMPLEX, different CHARACTER sets (for example, arabic, musical notation and cryllic script) and different ways of representing LOGICAL objects. DOUBLE PRECISION does not have different precisions and its use is not recommended — use a parameterised REAL type instead.

An example of the parameterisation of an intrinsic type is,

```
    INTEGER(KIND=1) :: ik1
    REAL(4) :: rk4
```

The kind parameters correspond to differing precisions supported by the compiler (details in the compiler manual).

Objects of different kinds can be mixed in arithmetic expressions and rules exist for type coercion, procedure arguments, however, must match in type **and** kind. There is no type coercion across procedure boundaries.

25.1 Integer Data Type by Kind

Selecting kind parameters by an explicit integer is still **not** portable, the explicit integer will correspond to different precisions on different machines, a mechanism is needed to select a kind value on the basis of the desired precision. For integers this can only be achieved by using the `SELECTED_INT_KIND` intrinsic function. For example, `SELECTED_INT_KIND(2)` returns an integer which corresponds to the kind representation capable of expressing numbers in the range, $(-10^2, 10^2)$. In the case of `SELECTED_INT_KIND` the argument specifies the minimum decimal exponent range for the desired model.

For example,

```
INTEGER :: short, medium, long, vlong
PARAMETER (short  = SELECTED_INT_KIND(2),  medium = SELECTED_INT_KIND(4),  &
           long   = SELECTED_INT_KIND(10), vlong  = SELECTED_INT_KIND(100))
INTEGER(short)    :: a,b,c
INTEGER(medium)   :: d,e,f
INTEGER(long)     :: g,h,i
```

The above declarations specify that precision should be at least:

- $(-10^2, 10^2)$ — `short`
- $(-10^4, 10^4)$ — `medium`
- $(-10^{10}, 10^{10})$ — `long`
- $(-10^{100}, 10^{100})$ — `vlong`

If a model with at least the stated precision is not available then the function will return `-1` and any declaration using this kind value will give an error (at compile time).

25.2 Constants of Selected Integer Kind

Constants of a selected kind are denoted by appending underscore followed by the kind number or an integer constant name (better):

```
100_2, 1238_4, 54321_long
```

Be **very careful** not to type a minus sign '-' instead of an underscore '_'!

There are other pitfalls too, the constant

```
1000_short
```

may not be valid as `KIND = short` may not be able to represent numbers greater than 100. Be very careful as the number may overflow.

25.3 Real KIND Selection

This works on a similar principle to integer kind selection, the intrinsic SELECTED_REAL_KIND can be parameterised with two values, the minimum precision and the minimum decimal exponent range, the corresponding kind value will be returned or, if the desired range cannot be supported, the result will be -1.

SELECTED_REAL_KIND(8,9) will return a kind value that supports numbers with a precision of 8 digits and decimal exponent range between -9 and +9.

Declarations are made in the same way as for INTEGERs, for example,

```
INTEGER, PARAMETER :: r1 = SELECTED_REAL_KIND(5,20),  &
                      r2 = SELECTED_REAL_KIND(10,40)
REAL(KIND=r1)       :: x, y, z
REAL(r2), PARAMETER :: diff = 100.0_r2
```

COMPLEX variables are specified in the same way,

```
COMPLEX(KIND=r1) :: cinema
COMPLEX(r2) :: inferiority = (100.0_r2,99.0_r2)
```

Both parts of the complex number have the same numeric range.

It can also be seen how the literal kind specification is achieved by the use of the underscore.

25.4 Kind Functions

It is often useful to be able to interrogate an object to see what kind parameter it has.

KIND is an intrinsic function that returns the integer which corresponds to the KIND of the argument. The argument can be a variable or a literal. For example, KIND(a) will return the integer which corresponds to the kind of a and KIND(20) returns the kind value chosen for the representation used for the default integer type.

The type conversion functions, INT, NINT, REAL, DBLE, etc., have an optional KIND= argument for specifying the representation of the result of the function, for example,

```
INTEGER(KIND=6) :: ik6
REAL x
INTEGER i
 ...
ik6 = REAL(x,KIND=6)
 ! or better
ik6 = REAL(x,KIND(ik6))
```

Retaining the representation precision is important, note the difference between the following two assignments,

```
INTEGER ia, ib
```

```
REAL(KIND=6) rk6
 ...
rk6 = rk6 + REAL(ia,KIND=6)/REAL(ib,KIND=6)
```

and,

```
rk6 = rk6 + REAL(ia)/REAL(ib)
```

In the second example, the RHS operand is less accurate than in the first.

Question 61: Kind Functions and Available Representations

Write a program which quizzes a processor and prints out the available kinds for the INTEGER data type. For each available kind, give the maximum exponent range that is supported. Also give the kind number corresponding to the default INTEGER. If an exponent is too large to be supported then SELECTED_INT_KIND will return -1. (You can assume that no number greater than 10^{32} will be supported.)

25.5 Expression Evaluation

If the two operands of any intrinsic operation have the same type and kind, then the result also has this type and kind. If the kinds are different, then the operand with the lower range is promoted before the operation is performed.

For example, with the following declarations

```
INTEGER(short) :: members, attendees
INTEGER(long)  :: salaries, costs
```

the expression:

- `members + attendees` is of kind `short`,
- `salaries - costs` is of kind `long`,
- `members * costs` is also of kind `long`.

Care must be taken to ensure the LHS is able to hold numbers returned by the RHS.

The rules for normal type coercion still hold, in mixed type expressions INTEGERs are promoted to REALs, REALs to DOUBLE PRECISION and so on.

25.6 Logical KIND Selection

LOGICAL objects follow exactly the same principle as for numeric data types. Even though a LOGICAL variable can only hold one of two values it can still be represented in a number of different ways. For example,

```
LOGICAL(KIND=4) :: yorn = .TRUE._4
LOGICAL(KIND=1), DIMENSION(10) :: mask
IF (yorn .EQ. LOGICAL(mask(1),KIND(yorn))) ....
```

KIND=1 could mean that only one byte is used to store each element of mask which would conserve space. Must refer to the compiler manual.

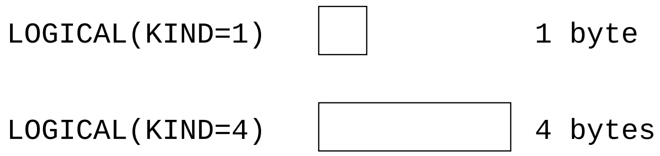


Figure 39: Possible Sizes of Different Logical Kind Variables

There is no SELECTED_LOGICAL_KIND intrinsic, however, the KIND intrinsic can be used to inquire about the representation and, as demonstrated above, the LOGICAL intrinsic, which has an optional KIND= argument, can be used to convert between representations.

25.7 Character KIND Selection

Every compiler must support at least one character set which must include all the Fortran characters. A compiler may also support other character sets:

```
INTEGER, PARAMETER :: greek = 1
CHARACTER(KIND=greek)    :: zeus, athena ! greek
CHARACTER(KIND=2,LEN=25) :: mohammed  ! arabic
```

Normal operations apply individually but characters of different kinds cannot be mixed.

For example,

```
print*, zeus//athena   ! OK
print*, mohammed//athena ! illegal
print*, CHAR(ICHAR(zeus),greek)
```

Recall that CHAR gives the character in the given position in the collating sequence. ICHAR and CHAR must be used when converting between character sets which means that only one character at a time can be converted.

Literals can also be specified:

```
greek_"αδαμ"
```

Notice how the kind is specified first — this is so the compiler has advanced knowledge that some 'funny' characters are about to appear and is able to warn the lexical analyser of this fact.

25.8 Kinds and Procedure Arguments

Dummy and actual arguments must match exactly in kind, type and rank (this is how generic overloads are resolved). In the following example it is assumed that the kind parameters are initialised in a module (good practice),

```
SUBROUTINE subbie(a,b,c)
 USE kind_defs
 REAL(r2), INTENT(IN)  :: a, c
 REAL(r1), INTENT(OUT) :: b
  ...
```

Any dummy arguments to an invocation of subbie must have matching arguments, for example,

```
 USE kind_defs
 REAL(r1) :: arg2
 REAL(r2) :: arg3
  ...
 CALL subbie(1.0_r2, arg2, arg3)
```

Using 1.0 instead of 1.0_r2 will not be correct on all compilers. The default kind always corresponds to a specific kind. Which kind it corresponds to depends on the compiler, this means that in the example, on one processor 1.0 may be identical (and have identical kind value) to 1.0_r2 but on another processor it may be equivalent to 1.0_r1, this would mean that the above program would compile OK sometimes (the first case) and other times would fail owing to the procedure arguments not matching in type, rank *and* kind. A truly portable program should NOT use intrinsic default types!

25.9 Kinds and Generic Interfaces

Note that if a procedure is defined with an argument of default intrinsic type then it is also defined for one parameterised instance of that type. Consider,

```
  ...
 INTERFACE blob1
  MODULE PROCEDURE a
  MODULE PROCEDURE b
 END INTERFACE ! blob1
  ...
CONTAINS
  ...
 SUBROUTINE a(i)
  INTEGER(KIND=1), INTENT(IN) :: i
   ...
 END SUBROUTINE a
 SUBROUTINE b(i)
  INTEGER(KIND=2), INTENT(IN) :: i
   ...
 END SUBROUTINE b
```

here both procedures have a unique interface as the kinds of the two dummy arguments are different. Consider,

```
  ...
 INTERFACE blob2
  MODULE PROCEDURE a
  MODULE PROCEDURE c
 END INTERFACE ! blob2
  ...
CONTAINS
 SUBROUTINE a(i)
  INTEGER(KIND=1), INTENT(IN) :: i
   ...
 END SUBROUTINE a
 SUBROUTINE c(i)
  INTEGER, INTENT(IN) :: i
   ...
 END SUBROUTINE c
```

here, if the default integer type corresponds to a kind value of 2 then the generic interfaces for `blob1` and `blob2` are the same, however, if the default integer type corresponds to a kind value of 1, the module will be in error because the overload set is non unique. In summary, default intrinsic types should not appear in generic interface specifications if *any* parameterised types appear as arguments to other procedures in the overloads set. This will ensure that the program is more portable.

Question 62: Different precision `INTCOMPLEX`

Enhance your `Integer_Complex_Arithmetic` module by converting the code to a portable system which can represent `INTCOMPLEX` values, say in the range $(-10^9, 10^9)$.

Expressions containing `INTCOMPLEX`s may involve integers of two kinds of `INTEGER`s, (`short_int` and `long_int` which correspond to ranges of $(-10^4, 10^4)$ and $(-10^9, 10^9)$ respectively) so the overload set for numeric operators must be extended. There is still only one kind of `REAL` and `COMPLEX` data type.

Modify your program so that `Setup_INTCOMPLEX` is a generic function which constructs values for all combinations of `short_int` and `long_int`.

Demonstrate that the whole module works by `USE`ing it in the following test program (which is available by anonymous ftp from `ftp.liv.ac.uk` in the directory `/pub/f90courses/progs`, filename `IntegerComplex6Program.f90`.

```
PROGRAM Testo
 USE Integer_Complex_Arithmetic
 IMPLICIT NONE
 TYPE(INTCOMPLEX) :: var1, var2, ans

 var1 = 3
 PRINT*, "var1 = 3"
 CALL Put_INTCOMPLEX(var1)

 var1 = 5.99
 PRINT*, "var1 = 5.99"
 CALL Put_INTCOMPLEX(var1)

 var1 = 6.01
```

```
     PRINT*, "var1 = 6.01"
     CALL Put_INTCOMPLEX(var1)

     var1 = Setup_INTCOMPLEX(1,2)
     var2 = Setup_INTCOMPLEX(3,4)

     PRINT*, "(1,2)+(3,4)"
     ans = var1 + var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(1,2)-(3,4)"
     ans = var1 - var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(1,2)/(3,4)"
     ans = var1 / var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)/(3,4)"
     ans = var2 / var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)/(1,2)"
     ans = var2 / var1
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(1,2)*(3,4)"
     ans = var1 * var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(1,2)**3"
     ans = var1 ** 3
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "+(1,2)"
     ans = +var1
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-(1,2)"
     ans = -var1
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "Type in the two INTCOMPLEX components"
     CALL Get_INTCOMPLEX(var1)
     PRINT*, "This is what was typed in"
     CALL Put_INTCOMPLEX(var1)

     PRINT*, "Your number/(3,4)"
     ans = var1 / var2
     CALL Put_INTCOMPLEX(ans)

! Intrinsics

     PRINT*, "REAL(3,4)"
```

```
     PRINT*, REAL(var2)
     PRINT*, "INT(3,4)"
     PRINT*, INT(var2)
     PRINT*, "AIMAG(3,4)"
     PRINT*, AIMAG(var2)
     PRINT*, "CONJG(3,4)"
     CALL Put_INTCOMPLEX(CONJG(var2))
     PRINT*, "ABS(3,4)"
     PRINT*, ABS(var2)

! REAL | INTEGER .OP. INTCOMPLEX

     PRINT*, "2+(3,4)"
     ans =  2 + var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2-(3,4)"
     ans =  2 - var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2*(3,4)"
     ans =  2 * var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2/(3,4)"
     ans =  2 /var2
     CALL Put_INTCOMPLEX(ans)

     var1 = Setup_INTCOMPLEX(1,2)
     PRINT*, "4/(1,2)"
     ans =  4/var1
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2.5+(3,4)"
     ans =  2.5 + var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2.5-(3,4)"
     ans =  2.5 - var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2.5*(3,4)"
     ans =  2 * var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "2.5/(3,4)"
     ans =  2.5 /var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "4.7/(1,2)"
     ans =  4.7/var1
     CALL Put_INTCOMPLEX(ans)
```

```
     PRINT*, "-2.5+(3,4)"
     ans =  -2.5 + var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-2.5-(3,4)"
     ans =  -2.5 - var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-2.5*(3,4)"
     ans =  -2.5 * var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-2.5/(3,4)"
     ans =  -2.5 /var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-4.7/(1,2)"
     ans =  -4.7/var1
     CALL Put_INTCOMPLEX(ans)

! INTCOMPLEX .OP. INTEGER | REAL

     PRINT*, "(3,4)+2"
     ans =  var2 + 2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "3,4)-2"
     ans =  var2-2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4) * 2"
     ans =  var2 * 2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)/2"
     ans =  var2/2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(1,2)/4"
     ans =  var1/4
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)+2.5"
     ans =  var2+2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)-2.5"
     ans =  var2-2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)*2.5"
     ans =   var2*2.5
     CALL Put_INTCOMPLEX(ans)
```

```
     PRINT*, "(3,4)/2.5"
     ans =  var2/2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(1,2)/4.7"
     ans =  var1/4.7
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "-(3,4)-2.5"
     ans =   -var2-2.5
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "(3,4)*(-2.5)"
     ans =  var2*(-2.5)
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "0.99 * (3,4)"
     ans = 0.99 *var2
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "1.0 * (3,4)"
     ans = 1.0 *var2
     CALL Put_INTCOMPLEX(ans)

! Extended precision

     var1 = Setup_INTCOMPLEX(3_short_int,1_long_int)
     PRINT*, "var1 = Setup_INTCOMPLEX(3_short_int,1_long_int)"
     CALL Put_INTCOMPLEX(var1)

     var1 = Setup_INTCOMPLEX(3_short_int,1_short_int)
     PRINT*, "var1 = Setup_INTCOMPLEX(3_short_int,1_short_int)"
     CALL Put_INTCOMPLEX(var1)

     var1 = Setup_INTCOMPLEX(3_long_int,1_short_int)
     PRINT*, "var1 = Setup_INTCOMPLEX(3_long_int,1_short_int)"
     CALL Put_INTCOMPLEX(var1)

     PRINT*, "var1 = Setup_INTCOMPLEX(3_long_int,1_long_int)"
     var1 = Setup_INTCOMPLEX(3_long_int,1_long_int)
     CALL Put_INTCOMPLEX(var1)

     PRINT*, "200_short_int * Setup_INTCOMPLEX(3_long_int,1_long_int)"
     ans = 200_short_int * var1
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "Setup_INTCOMPLEX(3_long_int,1_long_int) - 20_short_int"
     ans = var1 - 20_short_int
     CALL Put_INTCOMPLEX(ans)

     PRINT*, "Setup_INTCOMPLEX(3_long_int,1_long_int) ** 200_short_int"
     ans = var1 ** 6_short_int
     CALL Put_INTCOMPLEX(ans)
```

```
    PRINT*, "var ** 6_long_int"
    ans = var1 ** 6_long_int
    CALL Put_INTCOMPLEX(ans)

    PRINT*, "var1 = Setup_INTCOMPLEX(30000_long_int,10000_long_int)"
    var1 = Setup_INTCOMPLEX(30000_long_int,10000_long_int)
    CALL Put_INTCOMPLEX(var1)

    PRINT*, "400_long_int / Setup_INTCOMPLEX(30000_long_int,10000_long_int)"
    ans = 400_long_int / var1
    CALL Put_INTCOMPLEX(ans)

    PRINT*, "Setup_INTCOMPLEX(30000_long_int,10000_long_int) / 4000000"
    ans = var1 / 4000000_long_int
    CALL Put_INTCOMPLEX(ans)

   END PROGRAM Testo
```

26 More Intrinsics

26.1 Bit Manipulation Intrinsic Functions

Summary,

`BTEST(i,pos)`	bit testing
`IAND(i,j)`	AND
`IBCLR(i,pos)`	clear bit
`IBITS(i,pos,len)`	bit extraction
`IBSET(i,pos)`	set bit
`IEOR(i,j)`	exclusive OR
`IOR(i,j)`	inclusive OR
`ISHFT(i,shft)`	logical shift
`ISHFTC(i,shft)`	circular shift
`NOT(i)`	complement
`MVBITS(ifr,ifrpos, len,ito,itopos)`	move bits (`SUBROUTINE`)

Variables used as bit arguments must be `INTEGER` valued. The model for bit representation is that of an integer so 0 would have a bit representation of all 0's, 1 would have all zeros except for a 1 in the last position (position zero) (`00...0001`). The positions are numbered from zero and go from right to left (just like regular binary numbers.)

The model for bit representation is that of an unsigned integer, for example,

The number of bits in a single variable depends on the compiler — parameterised integers should be used to fix the number of bytes. The intrinsic `BIT_SIZE` gives the number of bits in a variable.

Here is a summary of the bit intrinsics and examples of their use; assume that `A` has the value 5 (`00...000101`) and `B` the value 3 (`00...000011`) in the following:

- ☐ `BTEST` — bit value.

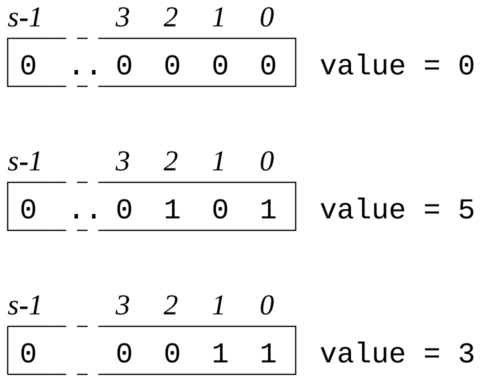


Figure 40: Visualisation of Bit Variables

`.TRUE.` if the bit in position `pos` of `INTEGER i` is 1, `.FALSE.` otherwise, for example, `BTEST(A,0)` has value `.TRUE.` and `BTEST(A,1)` is `.FALSE.`.

- `IAND` — bitwise `.AND.`.

  The two arguments are `INTEGER` the result is an `INTEGER` obtained by doing a logical `.AND.` on each bit of arguments, for example, `IAND(A,B)` is 1 (`00...000001`).

- `IBCLR` — bit clear.

  Set bit in position `pos` to 0, for example, `IBCLR(A,0)` is 4 (`00...000100`).

- `IBITS` — extract sequence of bits.

  `IBITS(A,1,2)` is 2 (10 in binary) or effectively `00..00010`.

- `IBSET` — set bit

  Set bit in position `pos` to 1, for example, `IBSET(A,1)` is 7 (`00...000111`).

- `IEOR`, `IOR` — bitwise `.OR.` (exclusive or inclusive).

  For example, `IEOR(A,B)` is 6 (`00...000110`) and `IOR(A,B)` is 7 (`00...000111`).

- `ISHFT`, `ISHFTC` — logical and circular shift.

  Shift the bits `shft` positions left (if `shft` is negative move bits to the right). `ISHFT` fills vacated positions by zeros `ISHFTC` wraps around. for example, `ISHFT(A,1)` is `00...001010` (10), `ISHFT(A,-1)` is `00...000010` (2). `ISHFTC(A,1)` is `00...001010` (10), `ISHFTC(A,-1)` is `10...000010` (-2147483646).

- `NOT` — compliment of whole word.

  `NOT(A)` is `11...111010` (-6).

- `MVBITS` (`SUBROUTINE`) — copy a sequence of bits between objects.

  For example `MVBITS(A,1,2,B,0)` says "move 2 bits beginning at position 1 in `A` to `B` position 0" (the bits are counted from right to left), this gives `B` the value `00...000010` (2).

26.2 Array Construction Intrinsics

There are four intrinsics in this class:

- MERGE(TSOURCE,FSOURCE,MASK)

  This function has the effect of merging two arrays under a mask, whether to include TSOURCE or FSOURCE in the result depends on LOGICAL array MASK; where it is .TRUE. the element from TSOURCE is included otherwise the element from FSOURCE is used instead.

- SPREAD(SOURCE,DIM,NCOPIES)

  This function replicates an array by adding NCOPIES along a dimension. The effect is analogous to taking a single page and replicating it to form a book with multiple copies of the same page. The result has one more dimension that the source array. SPREAD is useful during vectorisation of DO-loops.

- PACK(SOURCE,MASK[,VECTOR])

  This function packs an arbitrary dimensioned array into a one-dimensional vector under a mask. PACK is useful for compressing data.

- UNPACK(VECTOR,MASK,FIELD)

  This function unpacks a vector into an array under a mask, UNPACK is a complementary function to PACK and is therefore useful for uncompressing data.

26.2.1 MERGE Intrinsic

MERGE(TSOURCE,FSOURCE,MASK)

This function merges two arrays under mask control. TSOURCE, FSOURCE and MASK must all conform and the result is TSOURCE where MASK is .TRUE. and FSOURCE where it is .FALSE..

Consider,

```
INTEGER, DIMENSION(2,3) :: TSOURCE, FSOURCE
LOGICAL, DIMENSION(2,3) :: MASK
LOGICAL, PARAMETER :: T = .TRUE.
LOGICAL, PARAMETER :: F = .FALSE.
TSOURCE = RESHAPE((/1,3,5,7,9,11/), (/2,3/))
FSOURCE = RESHAPE((/0,2,4,6,8,10/), (/2,3/))
MASK    = RESHAPE((/T,F,T,F,F,T/), (/2,3/))
```

Now, as

$$\texttt{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

the highlighted elements are selected from the two source arrays,

$$\texttt{TSOURCE} = \begin{pmatrix} \boxed{1} & \boxed{5} & 9 \\ 3 & 7 & \boxed{11} \end{pmatrix}$$

and

$$\texttt{FSOURCE} = \begin{pmatrix} 0 & 4 & \boxed{8} \\ \boxed{2} & \boxed{6} & 10 \end{pmatrix}$$

thus

$$\texttt{MERGE(TSOURCE,FSOURCE,MASK)} = \begin{pmatrix} 1 & 5 & 8 \\ 2 & 6 & 11 \end{pmatrix}$$

26.2.2 SPREAD Intrinsic

`SPREAD(SOURCE,DIM,NCOPIES)`

This function replicates an array by adding `NCOPIES` of in the direction of a stated dimension.

For example, if `A` is `(/5, 7/)`, then

$$\texttt{SPREAD(A,2,4)} = \begin{pmatrix} 5 & 5 & 5 & 5 \\ 7 & 7 & 7 & 7 \end{pmatrix}$$

As `DIM`= 2 the vector `(/ 5, 7/)` is spread along dimension 2 (the direction of a row).

$$\texttt{SPREAD(A,1,4)} = \begin{pmatrix} 5 & 7 \\ 5 & 7 \\ 5 & 7 \\ 5 & 7 \end{pmatrix}$$

In this case `DIM`= 1 so the vector `(/ 5, 7/)` is spread along dimension 1 (the direction of a column).

A good use for this intrinsic is during vectorisation when, say, a doubly nested `DO` loop is combining a vector with each column of an array at the centre of the loop, the statement can be rewritten as a 2D array assignment by creating a copy of the vector for each column of the array. For example,

```
DO i = 1,100
 DO j = 1,100
  A(i,j) = B(i)*2
 END DO
END DO
```

can be transformed to

```
A(1:100,1:100) = SPREAD(B(:),2,100)*2
```

which exploits Fortran 90 array syntax.

26.2.3 PACK Intrinsic

`PACK(SOURCE,MASK[,VECTOR])`

This function packs a arbitrary shaped array into a one-dimensional array under a mask. `VECTOR`, if present, must be 1-D and must be of same type and kind as `SOURCE`.

Element i of the result is the element of `SOURCE` that corresponds to the ith `.TRUE.` element of `MASK`, (in array element order,) for $i = 1, ..., t$, where t is the number `.TRUE.` elements in `MASK`.

If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is t unless MASK is scalar with the value .TRUE. in which case the result size is the size of SOURCE. If VECTOR has size $n > t$, element i of the result has the value VECTOR(i), for $i = t + 1, ..., n$.

For example, if

$$\texttt{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

and

$$\texttt{A} = \begin{pmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{pmatrix}$$

then

- PACK(A,MASK) is (/1, 5, 11/);

  VECTOR is absent meaning that size of the result is the number of .TRUE. elements in MASK, 3. The elements of A which correspond to the .TRUE. elements of MASK are

  $$\texttt{A} = \begin{pmatrix} \boxed{1} & \boxed{5} & 9 \\ 3 & 7 & \boxed{11} \end{pmatrix}$$

  It can be seen that the highlighted elements are taken from A in array element order and packed into the result.

- PACK(A,MASK,(/3,4,5,6/)) is (/1, 5, 11, 6/).

  Here VECTOR is present so the result is of size 4. The first t elements of the results are as before corresponding to the .TRUE. elements of the mask. The remaining values are taken from VECTOR the 4th value of the result is the $t + 1$th element of VECTOR, 6. Hence the result (/1, 5, 11, 6/).

- PACK(A,.TRUE.,(/1,2,3,4,5,6,7,8,9/)) is (/1,3,5,7,9,11,7,8,9/).

  Here the mask is scalar so the first 6 elements are A and the rest from the end of VECTOR.

26.2.4 UNPACK Intrinsic

UNPACK(VECTOR,MASK,FIELD)

This function unpacks a vector into an array under a mask. FIELD, must conform to MASK and and must be of same type and kind as VECTOR. The result is the same shape as MASK.

If

$$\texttt{FIELD} = \begin{pmatrix} 9 & 5 & 1 \\ 7 & 7 & 3 \end{pmatrix}$$

and

$$\texttt{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

then

$$\texttt{UNPACK((/6,5,4/),MASK,FIELD)} = \begin{pmatrix} \boxed{6} & \boxed{5} & 1 \\ 7 & 7 & \boxed{4} \end{pmatrix}$$

The highlighted elements originate from VECTOR and correspond to the .TRUE. values of MASK, the other values are from the corresponding elements of FIELD.

Also

$$\mathtt{UNPACK((/3,2,1/),.NOT.MASK,FIELD)} = \begin{pmatrix} 9 & 5 & \boxed{1} \\ \boxed{3} & \boxed{2} & 3 \end{pmatrix} \tag{3}$$

again highlighted elements correspond to the .TRUE. values of NOTMASK, the other values are from the corresponding elements of FIELD.

26.3 TRANSFER Intrinsic

Most languages have a facility to change the type of an area of storage, in FORTRAN 77 people used EQUIVALENCE. Fortran 90 adds a different facility. The TRANSFER intrinsic converts (not coerces) a physical representation between data types; it is a retyping facility. The intrinsic takes the bit pattern of the underlying representation and interprets it as a different type. The intrinsic has the following syntax:

```
TRANSFER(SOURCE,MOLD)
```

where SOURCE is the object to be retyped and MOLD is an object of the target type. For example,

```
    REAL, DIMENSION(10)    :: A, AA
    INTEGER, DIMENSION(20) :: B
    COMPLEX, DIMENSION(5)  :: C
     ...
    A  = TRANSFER(B, (/ 0.0 /))
    AA = TRANSFER(B, 0.0)
    C  = TRANSFER(B, (/ (0.0,0.0) /))
     ...
```

The same bit pattern is used for each variable type:

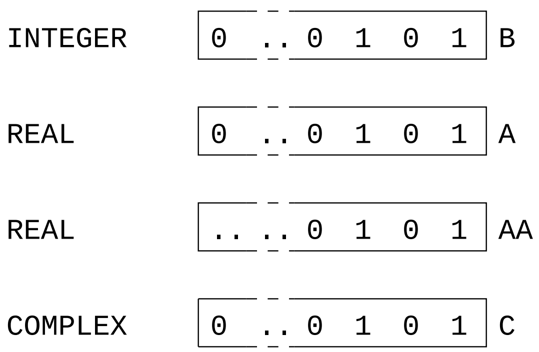


Figure 41: Visualisation of the TRANSFER Intrinsic

the variable will take on the number defined by its bit pattern.

The above example highlights the following points:

- `SOURCE` can be array or scalar valued of any type,
- `MOLD` can be array or scalar valued of any type and specifies the result type (which is the same shape as `SOURCE`),
- in the example there is a big difference between `MOLD` being array valued `(/ 0.0 /)` and scalar `0.0`
  - the result of `TRANSFER(B, (/ 0.0 /))` is an array, this is assigned element-by-element to `A`
  - the result of `TRANSFER(B, 0.0)` is a scalar, this scalar is assigned to every element of `AA`. (The first half of `B(1)` (4 bytes) is interpreted as a `REAL` number.)
- `(/ (0.0,0.0) /)` represents an array valued `COMPLEX` variable.

Module 11: Odds and Ends

27 Input / Output

Fortran 90 has a wealth of I/O, too much to cover here. It has many statements, (for example, `READ`, `WRITE`, `OPEN`, `CLOSE`, `REWIND` and `BACKSPACE`) built in to the language and also has very powerful formatting commands for 'pretty printing' or formatted output.

So far, all examples have performed I/O with a terminal. Fortran 90 allows a number of different streams (files) to be connected to a program for both reading and writing. In Fortran 90 a file is connected to a *logical unit* denoted by a number. This number must be positive and is often limited to be between 1 and 100, the exact number will be given in the compiler manual. Each logical unit can have many properties, for example,

- *file* — the name of the file connected to the unit,

  The name is specified in the `OPEN` statement.

- *action* — read, write, read and write,

  If a file is opened for one sort of action and another is attempted then an error will be generated, for example, it is not permissible to write to a file opened solely for reading.

- *status* — old, new, replace, etc.

  similar to above, if we open a new file but the file already exists then an error results.

- *access* method — sequential, direct,

  - *sequential* — is 'normal' access, each write / read causes a pointer to move down the file, the next line which is written / read appears after the previous line.
  - *direct* — each line is accessed by a number (a record number) which must be specified in the read / write statement made.

The maximum number of files which can be open at any one time will also be specified in the compiler manual.

27.1 `OPEN` Statement

The `OPEN` statement is used to connect a named file to a logical unit. It is often possible to pre-connect a file before the program has begun, if this is the case then there is no need for an `OPEN` statement, however, there are many default I/O settings many of which are processor dependent. Its good practice not to rely on defaults but to specify exactly what is required in an *explicit* `OPEN` statement. This will make the program more portable.

The syntax is

OPEN([UNIT=]< *integer* >, FILE=< *filename* >, ERR=< *label* >, &
STATUS=< *status* >, ACCESS=< *method* >, ACTION=< *mode* >, RECL=< *int-expr* >)

where,

- UNIT=< *integer* > specifies a numeric reference for the named file. The number must be one that is not currently in use.
- FILE=< *filename* > gives the filename to be associated with the logical unit. The name must match exactly with the actual file, sometimes this means that the filename needs to contain a specific number of blanks.
- ERR=< *label* > specifies a numeric label where control should be transferred if there is an error opening the named file.
- STATUS=< *status* > specifies the status of the named file, the status may be one of the following,
  - 'OLD', — file exists.
  - 'NEW', — file does not exist.
  - 'REPLACE' — file will be overwritten.
  - 'SCRATCH' — file is temporary and will be deleted when closed.
  - 'UNKNOWN' — unknown.
- ACCESS=< *method* > specifies the access method,
  - 'DIRECT' — the file consists of tagged records accessed by an ID number, in this case the record length *must* be specified (RECL). Individual records can be specified and updated without altering the rest of the file.
  - 'SEQUENTIAL' — the file is written / read (sequentially) line by line.
- ACTION=< *mode* > specifies what can be done to the file, the mode may be one of the following,
  - 'READ',— open for reading.
  - 'WRITE'— open for writing.
  - 'READWRITE' — open for both reading and writing.

There are other less important specifiers which are not covered here.

There now follows an example of use,

```
OPEN(17,FILE='output.dat',ERR=10, STATUS='REPLACE', &
        ACCESS='SEQUENTIAL',ACTION='WRITE')
```

A file output.dat is opened for writing, it is connected to logical unit number 17. The file is accessed on a line by line basis and already exists but is to be replaced. The label 10 must pertain to a valid executable statement.

```
OPEN(14,FILE='input.dat',ERR=10, STATUS='OLD', RECL=iexp, &
        ACCESS='DIRECT',ACTION='READ')
```

Here a file is opened for input only on unit 14. The file is directly accessed and (clearly) already exists.

27.2 READ Statement

Syntax (not all the specifiers can be used at the same time,)

READ([UNIT=]< *unit* >, [FMT=]< *format* >, IOSTAT=< *int-variable* >, ERR=< *label* >, &
END=< *label* >, EOR=< *label* >, ADVANCE=< *advance-mode* >, REC=< *int-expr* >, &
SIZE=< *num-chars* >) < *output-list* >

where

- UNIT=< *unit* > is a valid logical unit number or * for the default (standard) output, if it is the first field then the specifier is optional.
- FMT=< *format* > is a string of formatting characters, a valid FORMAT statement label or a * for free format. If this is the second field then the specifier is optional.
- IOSTAT=< *int-variable* > specifies an integer variable to hold a return code, as usual, zero means no error.
- < *label* > in ERR= is a valid label to where control jumps if there is a read error.
- < *label* > in END= is a valid label to where control jumps if an end-of-file is encountered, this can only be present in a READ statement.
- < *advance-mode* > specifies whether each READ should start a new record or not, setting the specifier to 'NO' initiates non-advancing I/O. The default is 'YES'. If non-advancing I/O is used then the file must be connected for sequential access and the format must be explicitly stated.
- EOR will jump to the specified label if an end-of-record is encountered. This can only be present in a READ statement and only if ADVANCE='NO' is also present.
- REC=< *int-expr* > is the record number for direct access.
- the SIZE specifier is used in conjunction with an integer variable, in this case nch. The variable will hold the number of characters read. This can only be present in a READ statement and only if ADVANCE='NO' is also present.

Consider,

```
READ(14,FMT='(3(F10.7,1x))',REC=iexp) a,b,c
```

here, the record specified by the integer iexp is read — this record should contain 3 real numbers separated by a space with 10 column and 7 decimal places. The values will be placed in a, b and c.

```
READ(*,'(A)',ADVANCE='NO',EOR=12,SIZE=nch) str
```

Since non-advancing output has been specified, the cursor will remain on the same line as the string is read. Under normal circumstances (default advancing output) the cursor would be positioned at the start of the next line. Note the explicit format descriptor. SIZE reurns the length of the string and EOR= specifies an destination if an end-of-record is encountered.

27.3 WRITE Statement

READ and WRITE can be interchanged in the following, (not all the specifiers can be used at the same time,)

WRITE([UNIT=]<*unit*>, [FMT=]<*format*>, IOSTAT=<*int-variable*>, ERR=<*label*>, &
ADVANCE=<*advance-mode*>, REC=<*int-expr*>) <*output-list*>

where

- ☐ UNIT=<*unit*> is a valid logical unit number or * for the default (standard) output. If it is the first field then the specifier is optional.
- ☐ FMT=<*format*> is a string of formatting characters, a valid FORMAT statement label or a * for free format. If this is the second field then the specifier is optional.
- ☐ IOSTAT=<*int-variable*> specifies an integer variable to hold a return code, as usual, zero means no error.
- ☐ <*label*> in ERR= is a valid label to where control jumps if there is an WRITE error.
- ☐ ADVANCE=<*advance-mode*> specifies whether each WRITE should start a new record or not. setting the specifier, (<*advance-mode*>,) to 'NO' initiates non-advancing I/O. The default is 'YES'. If non-advancing I/O is used then the file must be connected for sequential access and the format must be explicitly stated.
- ☐ REC=<*int-expr*> is the record number for direct access.

Consider,

```
WRITE(17,FMT='(I4)',IOSTAT=istat,ERR=10, END=11,EOR=12) ival
```

here, '(I4)' means INTEGER with 4 digits. the labels 10, 11 and 12 are statements where control can jump on an I / O exception. ival is an INTEGER variable whose value is written out.

```
WRITE(*,'(A)',ADVANCE='NO') 'Input : '
```

Since non-advancing output has been specified, the cursor will remain on the same line as the string 'Input : ', under normal circumstances (default advancing output) the cursor would be positioned at the start of the next line. Note the explicit format descriptor.

See later for a more detailed explanation of the format edit descriptors.

27.4 FORMAT Statement / FMT= Specifier

The FMT= specifier in a READ or WRITE statement can give either a line number of a FORMAT statement, an actual format string or a *.

Consider,

```
WRITE(17,FMT='(2X,2I4,1X,''name '',A7)')i,j,str
```

This writes out to the file `output.dat` the three variables `i`, `j` and `str` according to the specified format, *"2 spaces, (`2X`), 2 integer valued objects of 4 digits with no gaps, (`2I4`), one space, (`1X`), the string `'name '` and then 7 letters of a character object, (`A7`)"*. The variables are taken from the I/O list at the end of the line. Note the double single quotes (escaped quote) around `name`. A single " could have been used here instead.

Consider,

```
READ(14,*) x,y
```

This reads two values from `input.dat` using free format and assigns the values to `x` and `y`.

Further consider,

```
   WRITE(*,FMT=10) a,b
10 FORMAT('vals',2(F15.6,2X))
```

The `WRITE` statement uses the format specified in statement `10` to write to the standard output channel. The format is *"2 instances of: a real valued object spanning `15` columns with an accuracy of `6`, decimal places, (`F15.6`), followed by two spaces (`2X`)"*.

Given,

```
   WRITE(*,FMT=10) -1.05133, 333356.0
   WRITE(17,FMT='(2X,2I4,1X,''name '',A7)')11, -195, 'Philip'
10 FORMAT('vals',2(F15.6,2X))
```

the following is written,

```
11-195 name Philip
  -1.051330    333356.000000
```

27.5 Edit Descriptors

Fortran contains a large number of output edit descriptors which means that very complex I/O patterns can be specified, it is only intended that a summary be presented here. Any good textbook will elaborate:

Editor	**Meaning**
`I`*w*	*w* chars of integer data,
`F`*w.d*	*w* chars of real data (*d* dec. pl.),
`E`*w.d*	*w* chars of real data (*d* dec. pl.),
`L`*w*	*w* chars of logical data,
`A`[*w*]	[*w* chars of] `CHARACTER` data,
n`X`	skip *n* chars (*n* spaces),

Where,

- *w* determines the number of the total number of characters (column *w*idth) that the data spans, (on output if there are insufficient columns for the data then numeric data will not (cannot) be printed, `CHARACTER` data will be truncated).

- *d* specifies the number of decimal places that the data contains and is contained in the total number of characters. The number will be *rounded* (not truncated) to the desired accuracy.

and,

- I specifies that the data is of INTEGER type.
- E writes REAL using exponent form, 1.000E+00.
- F uses 'decimal point' form, 1.000.
- for F and E the sign is included in the total number of characters, by default plus signs are not output.
- L specifies LOGICAL data. .TRUE. and .FALSE. are output as a single character T or F. If *w* is greater that one them the single character is padded to the left with blanks.
- A specifies CHARACTER data. If *w* is specified strings which are too long (more than *w* characters) will be truncated. If *w* is not specified any length of string is appropriate.
- X skips the specified number of characters (*n* blanks are output).

Descriptors or groups of descriptors can be repeated by prefixing or parenthesesing and prefixing with the number of repeats, for example, I4 can be repeated twice by specifying 2I4 and I4,1X can be repeated twice by specifying 2(I4,1X).

Many of the above edit descriptors can be demonstrated:

```
WRITE(*,FMT='(2X,2(I4,1X),''name '',A4,F13.5,1X,E13.5)') &
      77778,3,'ABCDEFGHI',14.45,14.5666666
```

gives

```
bb****bbbb3bnamebABCDbbbbb14.45000bbb0.14567E+02
```

where the b signifies a blank! In the above example, the first INTEGER is unable to be written as the number is too long and the last REAL number is rounded to fit into the spaces available. The string is truncated to fit into the space available. A READ statement could use the same format editor.

Type coercion is not performed so INTEGERs cannot be written out as REALs.

27.6 Other I/O Statements

- CLOSE unattaches the unit number specified in the statement. This should always be used to add an end of file mark at the closure point. It is an error to close a file that is not open.
- REWIND simply puts the file pointer back to the start.
- BACKSPACE moves the file pointer is moved back one record, however, it often puts the file pointer back to the start, and then fast forwards.
- ENDFILE forces an end-of-file to be written into the file but the file remains open.

The above statements have other specifiers such as IOSTAT,

For example,

```
REWIND (UNIT=14)
BACKSPACE (UNIT=17)
ENDFILE (17)
CLOSE (17, IOSTAT=ival)
```

Question 63: File IO

Write a program to open a new sequential file on unit 4 called marks.dat. The program should then read a student's name followed by his/her marks for four exams from the keyboard (the default unit) and write these to the file on a single line. Repeat the read/write process until a student with the name END is entered. Close the output file.

Question 64: Formatted IO

Given the statement:

```
READ(*,'(F10.3,A2,L10)') A,C,L
```

what would the variables A (a real), C (a character of length 2), L (a logical) contain when given the following input? (Note: b signified a blank space.)

1. bbb5.34bbbNOb.TRUE.
2. 5.34bbbbbbYbbFbbbbb
3. b6bbbbbb3211bbbbbbT
4. bbbbbbbbbbbbbbbbbbF

Question 65: Formatted IO

Give the formats which would be suitable for use with the statements,

```
REAL      :: A
CHARACTER(LEN=12) :: B
INTEGER   :: C
READ(*,FORM) A,B,C
```

if the record to be read is, (b is blank),

```
bbb354bbINTRODUCTORYb1993
```

and values to be assigned to A, B and C are given in the following table:

A	B	C
35.4	INTRODUCTORY	1993
3.54	DUCT	93
.354	TROD	19
354.0	TORY	99

Question 66: More Formatted IO

Given that `A` is an array declared with the statement,

```
INTEGER, DIMENSION(1:8) :: A
```

give a formatted input statement which will read in the values `1` to `A(1)`, `2` to `A(2)`, etc. from the following records:

```
12
34
56
78
```

Question 67: Formatted File IO

A file `personnel.dat` contains records of people's name (up to 15 characters), age (3 digit integer), height (in metres to the nearest centimetre), telephone number (4 digit integer). Write a program to read the file and print out the details in the following format:

```
                          Height
     Name          Age   (metres)    Tel. No.
     ----          ---    ------     --------
  Bloggs J. G.      45     1.80        3456
  Clinton P. J.     47     1.75        6783
  etc.
```

Question 68: More File IO

Write a program to read the file produced by the exam marks program and to print a list of students and their average marks.

28 External Procedures

Fortran 90 allows a class of procedure that is not contained within a `PROGRAM` or a `MODULE` — an `EXTERNAL` procedure.

This is the old FORTRAN 77-style of programming and is more clumsy than the Fortran 90 way.

Differences:

- they may be compiled separately,
- may need an explicit `INTERFACE` to be supplied to the calling program,
- can be used as arguments (in addition to intrinsics),
- should contain the `IMPLICIT NONE` specifier.

28.1 External Subroutine Syntax

Syntax of a (non-recursive) subroutine declaration:

`SUBROUTINE` <*procname*>[(<*dummy args*>)]
    <*declaration of dummy args*>
        ...
    <*declaration of local objects*>
        ...
    <*executable stmts*>
        ...
[`CONTAINS`
    <*internal procedure definitions*>]
`END` [`SUBROUTINE` [<*procname*>]]

`SUBROUTINE`s may contain internal procedures but only if they themselves are *not* already internal.

```
SUBROUTINE Ext_1(...)
 ! ...
CONTAINS ! Internal Procs
   SUBROUTINE Int_1(...)
    ! Executable stmts
   END SUBROUTINE Int_1
    ! etc.
   FUNCTION Int_n(...)
    ! Executable stmts
   END FUNCTION Int_n
END SUBROUTINE Ext_1
```

```
SUBROUTINE Ext_2(...)
 ! etc
END SUBROUTINE Ext_2
```

Figure 42: Schematic Diagram of a Subroutine

The structure is similar to that of the main `PROGRAM` unit except a `SUBROUTINE` can be parameterised (with arguments) and the type and kind of these must be specified in the declarations section. A `SUBROUTINE` may include calls to other program units either internal, external or visible by `USE` association (defined in a module and `USE`d in the procedure.

28.1.1 External Subroutine Example

An external procedure may invoke a further external procedure,

```
SUBROUTINE sub1(a,b,c)
 IMPLICIT NONE
 EXTERNAL sum_sq
 REAL :: a, b, c, s
   ...
  CALL sum_sq(a,b,c,s)
  ...
END SUBROUTINE sub1
```

calls,

```
SUBROUTINE sum_sq(aa,bb,cc,ss)
 IMPLICIT NONE
  REAL, INTENT(IN) :: aa, bb, cc
  REAL, INTENT(OUT) :: ss
  ss = aa*aa + bb*bb + cc*cc
END SUBROUTINE sum_sq
```

The principle is the same as for calling an internal procedure except that:

1. whereas an internal procedure has access to the host's declarations (and so inherits, amongst other things, the `IMPLICIT NONE`) external procedures do not. An `IMPLICIT NONE` is needed in every external procedure.
2. the external procedure should be declared in an `EXTERNAL` statement. (This is optional but is good practise.)

28.2 External Function Syntax

Syntax of a (non-recursive) function:

[<*prefix*>] `FUNCTION` <*procname*>([<*dummy args*>])
    <*declaration of dummy args*>
    <*declaration of local objects*>
        ...
    <*executable stmts, assignment of result*>
[`CONTAINS`
    <*internal procedure definitions*>]
`END` [`FUNCTION` [<*procname*>]]

here <*prefix*>, specifies the result type. or,

```
FUNCTION <procname>( [<dummy args>])
        <declaration of dummy args>
        <declaration of procname type>
        <declaration of local objects>
                ...
        <executable stmts, assignment of result>
 [ CONTAINS
        <internal procedure definitions> ]
END [ FUNCTION [ <procname> ] ]
```

Functions are very similar to subroutines except that a function should have a type specifier which defines the type of the result. (The type of the function result can either be given as a prefix to the function name or alternatively be given in the declarations area of the code by preceding the function name (with no arguments or parentheses) by a type specifier. It is a matter of personal taste which method is adopted.) For example,

```
INTEGER FUNCTION largest(i,j,k)
 IMPLICIT NONE
```

is equivalent to,

```
FUNCTION largest(i,j,k)
 IMPLICIT NONE
 INTEGER largest
```

The function name, <*procname*>, is the result variable so a function must contain a statement which assigns a value to this name; a routine without one is an error.

The type of an external function must be given in the calling program unit. It is good practise to attribute this declaration with the `EXTERNAL` attribute. A valid declaration in the calling program unit might be,

```
INTEGER, EXTERNAL :: largest
```

28.2.1 Function Example

A function is invoked by its appearance in an expression at the place where its result value is needed,

```
total = total + largest(a,b,c)
```

The function is defined as follows,

```
INTEGER FUNCTION largest(i,j,k)
 INTEGER :: i, j, k
 largest = i
 IF (j .GT. largest) largest = j
 IF (k .GT. largest) largest = k
END FUNCTION largest
```

or equivalently as,

```
FUNCTION largest(i,j,k)
 INTEGER :: i, j, k
 INTEGER :: largest
  ...
END FUNCTION largest
```

The largest value of `i, j` and `k` will be substituted at the place where the function call is made. As with subroutines, the dummy and actual arguments must match in type kind and rank.

More than one function (or the same function more than once) may be invoked in a single statement. For example,

```
rezzy = funky1(a,b,c) + funky2(a,b,c)
```

Care must be taken that the order of execution will not alter the result; aside from operator precedence, the order of evaluation of a statement in a Fortran 90 program is undefined. It is not specified whether `funky1` or `funky2` is evaluated first; they could even be executed in parallel!

If a function invocation has side-effects and it is called twice in the same statement then problems may occur. To safeguard against this a function invocation should not assign to its arguments, modify any global variables or perform I/O! A function which obeys these restrictions is termed `PURE`. The `PURE` specifier will be part of the Fortran 95 language.

28.3 Procedure Interfaces

Fortran 90 has introduced a new feature whereby it is possible, often essential and wholly desirable to provide an *explicit interface* for an external procedure. Such an interface provides the compiler with all the information it needs to allow it to it make consistency checks and ensure that enough information is communicated to procedures at run-time.

Consider the following procedure,

```
SUBROUTINE expsum( n, k, x, sum )    ! in interface
 USE KIND_VALS:ONLY long
 IMPLICIT NONE
 INTEGER, INTENT(IN) :: n            ! in interface
 REAL(long), INTENT(IN)  :: k,x            ! in interface
 REAL(long), INTENT(OUT)  :: sum           ! in interface
 REAL(long) :: cool_time
 sum = 0.0
 DO i = 1, n
  sum = sum + exp(-i*k*x)
 END DO
END SUBROUTINE expsum                 ! in interface
```

The explicit `INTERFACE` for this routine is,

```
INTERFACE
 SUBROUTINE expsum( n, k, x, sum )
```

```
      USE KIND_VALS:ONLY long
      INTEGER :: n
      REAL(long), INTENT(IN)    :: k,x
      REAL(long), INTENT(OUT)    :: sum
     END SUBROUTINE expsum
    END INTERFACE
```

An interface declaration, which is initiated with the INTERFACE statement and terminated by END INTERFACE, gives the characteristics (attributes) of both the dummy arguments (for example, the name, type, kind and rank and the procedure (for example, name, class and type for functions). (The INTENT attribute should also be given — this has not been covered yet see Section 17.7 for details.) The USE statement is necessary so that the meaning of long can be established. An interface cannot be used for a procedure specified in an EXTERNAL statement (and vice-versa).

The declaration can be thought of as being the whole procedure without the local declarations (for example, cool_time,) and executable code! If this interface is included in the declarations part of a program unit which calls expsum, then the interface is said to be *explicit*. Clearly, an interface declaration must match the procedure that it refers to.

It is is generally a good idea to make all interfaces explicit.

An interface is only relevant for external procedures; the interface to an internal procedure is always visible as it is already contained within the host. Interfaces to intrinsic procedures are also explicit within the language.

An interface only contains:

- the SUBROUTINE or FUNCTION header,
- (if not included in the header) the FUNCTION type,
- declarations of the dummy arguments (including attributes),
- the END SUBROUTINE or END FUNCTION statement

Interfaces are only ever needed for EXTERNAL procedures.

Question 69: Interfaces

What is the interface of the following procedure?

```
SUBROUTINE SGETRI_F90(A, IPIV, INFO )
   USE LA_PRECISION, ONLY:WP
   IMPLICIT NONE
   REAL(KIND=WP), INTENT(INOUT), DIMENSION(:,:) :: A
   REAL(KIND=WP), ALLOCATABLE, DIMENSION(:,:)   :: WORK
   INTEGER, INTENT(IN), DIMENSION(:)            :: IPIV
   INTEGER, INTENT(OUT)                         :: INFO
   INTEGER                                      :: N
   INTEGER                                      :: ILAENV
   EXTERNAL  ILAENV
   INTRINSIC MIN, MATMUL
   INFO = 0
```

```
   N = SIZE(A,1)
   IF (SIZE(A,2) /= N) THEN
      INFO = -1
   ELSE IF (SIZE(IPIV,1) /= N) THEN
      INFO = -2
   ENDIF
   IF( N.EQ.0 ) RETURN
     ....
END SUBROUTINE SGETRI_F90
```

28.3.1 Interface Example

The following program includes an explicit interface,

```
PROGRAM interface_example
 IMPLICIT NONE
 INTERFACE
  SUBROUTINE expsum(N,K,X,sum)
   INTEGER, INTENT(IN) :: N
   REAL, INTENT(IN)  :: K,X
   REAL, INTENT(OUT)  :: sum
  END SUBROUTINE expsum
 END INTERFACE
 REAL :: sum
  ...
 CALL expsum(10,0.5,0.1,sum)
  ...
END PROGRAM interface_example
```

The above interface includes information about the number, type, kind and rank of the dummy arguments of the procedure `expsum`.

Using an `INTERFACE` provides for better optimisation and type checking, and allows separate compilation to be performed.

28.4 Required Interfaces

Explicit interfaces are mandatory if an `EXTERNAL` procedure:

- has dummy arguments that are assumed-shape arrays, pointers or targets.

  This is so the compiler can figure out what information needs to be passed to the procedure, for example, the rank, type and bounds of an array whose corresponding dummy argument is an assumed-shape array, (see Section 18.2), or the types and attributes of pointers or targets.

- has `OPTIONAL` arguments.

  The compiler needs to knows the names of the arguments so it can figure out the correct association when any of the optional arguments are missing.

- is an array or pointer valued result (functions).

  The compiler needs to know to pass back the function result in a different form from usual.

- ☐ contains an inherited `LEN=*` length specifier (character functions).

  The compiler needs to know to pass string length information to and from the procedure.

and when the reference:

- ☐ has a keyword argument.

  Same reasons as optional case above.
- ☐ is a defined assignment.

  Extra information is required.
- ☐ is a call to the generic name.

  Extra information is required.
- ☐ is a call to a defined operator (functions).

  Extra information is required.

Recall that a procedure cannot appear both in an `INTERFACE` block and in an `EXTERNAL` statement in the same scoping unit.

Question 70: Simple External Procedure

Write a program that fills an array of a user specified size with random real numbers (in the range 0 - 1). This program should then call an `EXTERNAL` procedure which reports on the frequency of numbers less than 0.5 in this array. Use assumed-shape arrays in the procedure. [You may find the `COUNT` intrinsic useful in this question.]

28.5 Procedure Arguments

If an external procedure is to be used as an argument it needs to be declared at the call site with an `INTRINSIC` or `EXTERNAL` attribute.

- ☐ `INTRINSIC` attribute — for in-built external procedures.
- ☐ `EXTERNAL` attribute — for external or dummy procedures.

Internal procedures are **forbidden** to appear as arguments.

28.5.1 The `INTRINSIC` Attribute

A name with the intrinsic attribute represents an intrinsic procedure and allows it to be used as an actual argument. (Note: the *specific*, not generic, procedure name should be used. Many intrinsic functions can be referred to be two different names, the *specific* and the *generic* names. The Fortran 90 intrinsic functions are multiply defined one for each argument of a different intrinsic type, (for example, `REAL`, `INTEGER` and `COMPLEX`). Each of these different functions has a different name, (for example, `ABS`, `IABS` and `CABS`), corresponding to the different argument types. All of these *specific* names also belongs to a *generic* name class (in this case, `ABS`) and can be accessed by using the generic name instead of the specific name. The compiler is able to look at the type of the argument and decide which specific function should be referenced — this is called generic resolution. If a procedure is to be passed as an actual procedure argument the specific name must be used because the procedure reference will not have any arguments at the point where it is passed so the generic reference will not be able to be resolved.) A procedure may be attributed as part of a type declaration or in an `INTRINSIC` statement.

For example,

```
INTRINSIC MVBITS
REAL, INTRINSIC :: ASIN
```

declares `ASIN` to be an intrinsic function and will allow it to be used as an actual argument. (`INTRINSIC` and `EXTERNAL` statements **cannot** currently contain the `::` separator. This is a language anomily and will be removed in Fortran 95.)

The following procedure call would be valid,

```
CALL subbo(MVBITS, ASIN)
```

where the definition of `subbo` is as follows,

```
SUBROUTINE subbo(sub,fun)
 IMPLICIT NONE
 EXTERNAL sub
 REAL, EXTERNAL :: fun
   ...
  PRINT*, fun(0.4)
  CALL sub
   ...
END SUBROUTINE subbo
```

It can be seen that the dummy procedure arguments can be invoked in the same way as regular procedures. Note how dummy procedure argument are always declared as `EXTERNAL` even if they are intrinsic functions. This is because, in the above case, `fun` is not the name of an intrinsic procedure.

28.5.2 The EXTERNAL Attribute

A name with the external attribute represents an external procedure or a dummy procedure and allows it to be used as an actual argument. (Note: the *specific*, not generic, procedure name should be used.) As Fortran 90 allows `EXTERNAL` (user defined) procedures to have the same names as `INTRINSIC` procedures it is often necessary to be able to differentiate between two references. If an intrinsic procedure name is used in an `EXTERNAL` statement then only the external procedure is visible in that scope; the intrinsic becomes unavailable. A procedure may be attributed as part of a type declaration or in an `EXTERNAL` statement.

For example,

```
EXTERNAL My_Subby
INTEGER, EXTERNAL :: My_Funky
INTEGER, EXTERNAL :: IABS
```

My_Subby and My_Funky are declared to be a user written external subroutine and integer function respectively. Both these procedures can now be used as actual arguments in procedure invocations. IABS is also declared to be a user-written external function and will also be allowed to appear an actual argument. The intrinsic function IABS will become unavailable in the current scoping unit. (If an intrinsic procedure name is used in an EXTERNAL statement then only the external procedure of that name is visible in that scope; the intrinsic becomes unavailable.)

28.5.3 Procedure Arguments Example

The following example demonstrates the use of procedures as arguments:

```
PROGRAM main
 IMPLICIT NONE
 INTRINSIC ASIN
 REAL, EXTERNAL :: my_sin
 EXTERNAL diffo1
 CALL subby(ASIN,my_sin,diffo1,SIN(0.5))
END PROGRAM

SUBROUTINE subby(fun1,fun2,sub1,x)
 IMPLICIT NONE
 REAL, INTENT(IN)  :: x
 REAL, EXTERNAL :: fun2, fun1
 EXTERNAL sub1
  PRINT*, fun1(x), fun2(x)
  CALL sub1(fun2(x),fun1,x)
END SUBROUTINE subby

SUBROUTINE diffo1(y,f,x)
 IMPLICIT NONE
 REAL, INTENT(IN)  :: x,y
 REAL, EXTERNAL :: f
 print*, "Diffo1 = ",y-f(x)
END SUBROUTINE diffo1

REAL FUNCTION my_sin(x)
 ...
END FUNCTION my_sin
```

It can be seen that when a procedure is passed as an actual argument it is merely passed by referencing its name, if it is a function which is supplied with arguments, for example, SIN(0.5), then it will be evaluated before the call is made.

The above example raises a few points:

- INTRINSIC ASIN specifies that ASIN refers to the Fortran 90 intrinsic function and allows it to be used as an actual argument which corresponds to a dummy procedure, ASIN is the specific name of the generic function SIN,
- REAL, EXTERNAL :: my_sin declares my_sin to be a user-defined real valued function which can be used as an actual argument which corresponds to a dummy procedure,

- EXTERNAL diffo1 declares diffo1 to be an external subroutine and allows it to be used as an actual argument which corresponds to a dummy procedure,
- diffo1 is argument associated with sub1, when sub1 is invoked it is really diffo1 that is being invoked.
- when sub1 / diffo1 is called the first argument is evaluated (fun2(x)) and the second and third arguments are passed across.
- dummy and actual procedure arguments must match in type, kind, rank and number of arguments,
- some intrinsic functions cannot be used as dummy procedure arguments, these include LLT and type conversion functions such as REAL, (see the Fortran 90 standard for a full list,)
- must always use the **specific** name of any intrinsic procedure, i.e., must use ASIN and not SIN.

Question 71: Functions as dummy arguments

Write a subroutine that accepts a function with one real argument (user defined or intrinsic), a start value $i1$, end value $i2$ and stride $i3$ and prints out the value of the function at each point defined by the sequence, $i1, i1 + i3, i1 + 2i3, ...$ Demonstrate its functionality by producing tables for two intrinsic and two user defined functions.

29 Object Initialisation

29.1 DATA Statement

In Fortran 90 the main use of the DATA statement is to allow initialisation of sections of arrays where the facilities of array constructors and assignment in the array declaration make this difficult. A DATA statement can be placed anywhere in the relevant program unit but its is common practice to place it amongst the declarations. It is 'executed' *once* (conceptually in parallel) on the first invocation of the procedure just before the first executable statement.

DATA statements are good for initialising odd shaped sections of arrays.

The syntax is as follows,

DATA < *var1*-list >/< *data1*-list >/, ...< *varn*-list >/< *datan*-list >/

The number of constants in each < *data*-list > must be equal to the number of variables / array elements in the corresponding < *var*-list >. Each < *data*-list > can only contain constants or structure (user-defined type) constructors, plus implied-do loop specifiers and repetition specifiers.

Any object initialised by the DATA statement has the SAVE attribute

29.1.1 DATA Statement Example

As an example, consider initialising a 1000×1000 array with all the edge values equal to 1 and with the rest of the array zero. This is very hard to do in an initialisation statement, it can be done using a

RESHAPE and a series of implied-do loops, but the whole array must be initialised in one go. The DATA statement allows this operation to be spread over a number of lines.

```
REAL :: matrix(100,100)
DATA matrix(1,    1:100) / 100*1.0  / ! top row
DATA matrix(100,  1:100) / 100*1.0  / ! bot row
DATA matrix(2:99, 1)     / 98*1.0   / ! left col
DATA matrix(2:99, 100)   / 98*1.0   / ! right col
DATA matrix(2:99, 2:99)  / 9604*0.0 / ! interior
```

The expression 100*1.0 means "100 occurrences of 1.0", the * is the repetition specifier and the number of repeats must be a scalar integer literal (not a variable). In this context it cannot be confused with the multiplication operator because such operators are not allowed in a DATA statement. A list of comma separated initialisations is also allowed, for example,

```
DATA matrix(1,    1:100) / 50*1.0, 50*2.0  / ! top row
```

A further example shows how initialisation in an assignment statement when it can be done is neater than a DATA statement,

```
INTEGER :: count,I,J
REAL    :: inc, max, min
CHARACTER(LEN=5) :: light
LOGICAL :: red, blue, green
DATA count/0/, I/5/, J/100/
DATA inc, max, min/1.0E-05, 10.0E+05, -10.0E+05/
DATA light/'Amber'/
DATA red/.TRUE./, blue, green/.FALSE.,.FALSE./
```

is the same as

```
INTEGER :: count=0, I=5, J=100
REAL    :: inc=1.0E-05, max=10.0E+05, min=-10.0E+05
CHARACTER(LEN=5) :: light='Amber'
LOGICAL :: red=.TRUE., blue=.FALSE., green=.FALSE.
```

29.2 Data Statement — Implied DO Loop

In a DATA statement the < *var*-list > may be specified by means of an implied-DO. Initialising a matrix to have a given constant value, say, 5.0 on the diagonal and zero everywhere else is simple to do using this method.

The object section can be specified by a tight loop which is more expressive than array syntax would allow:

```
REAL :: diag(100,100)
DATA (diag(i,i), i=1,100) / 100*5.0 / ! sets diagonal elements
DATA ((diag(i,j),diag(j,i),j=i+1,100),i=1,100) / 9900*0.0 /
                          ! sets the upper and lower triangles
```

The first DATA statement containing the following implied-DO, (diag(i,i), i=1,100), has the same effect as if diag(i,i)=5.0 were nested in a DO i=1,100 DO-loop. The second contains a nested implied-DO and behaves as if the second loop expression i= is the outermost DO of a set of two nested loops, in other words, its is like,

```
DO i=1,100
 DO j=i+1,100
  diag(i,j) = 0.0
  diag(j,i) = 0.0
 END DO
END DO
```

so the j loop varies the quickest meaning that the following elements are selected in the given order,

```
diag(1,2), diag(2,1), diag(1,3), diag(3,1), ...,
diag(2,3), diag(3,2), ...
```

30 Handling Exceptions

30.1 GOTO Statement

The GOTO statement:

- is a powerful but undisciplined branching statement;
- can be used to create jumps to almost anywhere in a program unit;
- can be dangerous;
- can lead to unstructured code (logical spaghetti).
- is very useful in exceptional circumstances such as jumping out of heavily nested structures.

The basic syntax is

GOTO < *numeric-label* >

The label must exist, be in the same scoping unit as the statement and be executable. This label cannot be a construct name.

GOTO should not be used for simulating other available control structure such as loops — use the purpose-designed syntax.

30.1.1 GOTO Statement Example

Consider the following example of an atrocious use of the GOTO statement,

```
   GOTO 10   ! jump forward
 23 CONTINUE
   i = i - 1
   IF (i .eq. 0) GOTO 99
 10 PRINT*, "Line 10"
 69 j = j - 1 ! loop
    ...
   IF (j .ne. 0) GOTO 69
   GOTO 23   ! jump back
099 CONTINUE
```

The code fragment demonstrates forward and backward transfer of control and the simulation of a loop. The whole example could be be rewritten in a far neater and structured way.

The best use of GOTO statements is in jumping out of a heavily nested structure when an unexpected event occurs such as a potential divide by zero.

30.2 RETURN and STOP Statements

The RETURN and STOP statements can be used to program exceptions in procedures.

RETURN transfers control to the last line of the procedure (and then back to the calling program unit) and serves as a quick exit from the middle of a procedure. It is especially useful when things have gone wrong and the procedure execution needs to be aborted.

STOP causes the program to terminate immediately. A string or number may be written to the standard output when this happens. STOP is, again, useful for exceptions, however, this statement could cause great problems on a distributed memory machine if some processors just stop without cleaning up! STOP is often used as status report which comments on the success or failure of a program execution.

For example, the following will exit from the subroutine if there is insufficient space to allocate the array A,

```
SUBROUTINE sub(ierror)
 INTEGER, INTENT(OUT) :: ierror
 ...
 ALLOCATE(A(100),STAT=ierror)
 IF (ierror>0) THEN
  PRINT*, 'memory fault'
  RETURN
 END IF
     ...
END SUBROUTINE
```

The same effect as that of the RETURN statement could be obtained by putting a label on the END statement and using a GOTO statement to pass control directly to this label.

In the above example STOP could be used instead of RETURN:

```
STOP 'stopped in sub'
```

the string is optional or can be a literal integer constant of up to 5 digits. It is output upon execution of STOP at which time execution of the program terminates.

31 Fortran 95

Fortran 95 will be the new Fortran Standard. It will include the following:

- FORALL statement and construct

  ```
  FORALL(i=1:n:2,j=1:m:2)
   A(i,j) = i*j
  END FORALL
  ```

  simultaneously assigns i*j to each array element A(i,j).

- nested WHERE constructs,
- user defined ELEMENTAL procedures, this will allow the ELEMENTAL keyword to be added to procedure definitions.
- PURE procedures, this will allow the PURE keyword to be added to procedure definitions and states that the procedure is "side-effect free". ELEMENTAL procedures are PURE.
- user-defined functions in initialisation expressions.
- automatic deallocation of arrays, no need to deallocate arrays before leaving a procedure.
- improved object initialisation, pointers can be initialised to be disassociated.
- remove conflicts with IEC 559 (IEEE 754/854) (floating point arithmetic standard).
- more obsolescent features, for example, fixed source form, assumed sized arrays, CHARACTER*< *len*> declarations, statement functions.
- language tidy-ups and ambiguities (mistakes).

The full content of Fortran 95 is not yet finalised and may change.

31.1 Rationale (by Craig Dedo)

The reasons for the changes are laid out below.

31.1.1 FORALL

The FORALL statement and construct and PURE procedures were added to Fortran 95 to allow Fortran 95 programs to execute efficiently in parallel on multi-processor systems. These features allow the majority of programs coded in High Performance Fortran (HPF) to run on a standard conforming Fortran 95 processor with little change. Adding these features to Fortran 95 does not imply that a particular Fortran 95 processor is multi-processor.

The purpose of the FORALL statement and construct is to provide a convenient syntax for simultaneous assignments to large groups of array elements. The multiple assignment functionality it provides is very similar to that provided by the array assignment statement and the WHERE construct in Fortran 90. FORALL differs from these constructs in its syntax, which is intended to be more suggestive of local operations on each element of an array, and in its generality, which allows a larger class of array sections to be specified. In addition, a FORALL may invoke user-defined functions on the elements of an array, simulating Fortran 90 elemental function invocation (albeit with a different syntax).

FORALL is not intended to be a completely general parallel construct; for example, it does not express pipelined computations or multiple-instruction multiple-data (MIMD) computation well. This was an explicit design decision made in order to simplify the construct and promote agreement on the statement's semantics.

31.1.2 Nested WHERE Construct

The WHERE construct was extended in order to provide for syntactic regularity with the FORALL construct. The FORALL construct allows for nested FORALL constructs.

Early implementation of some High Performance Fortran (HPF) processors included the nested WHERE construct. Use of these processors showed that the feature provided real value to customers.

31.1.3 PURE Procedures

The purpose of PURE procedures is to allow a processor to know when it is safe to implement a section of code as a parallel operation. The freedom from side effects of a pure function assists the efficient implementation of concurrent execution and allows the function to be invoked concurrently in a FORALL without such undesirable consequences as nondeterminism. It also forces some error checking on functions used in a FORALL construct.

31.1.4 Elemental Procedures

ELEMENTAL procedures provide the programmer with more powerful expressive capabilities and the processor with additional opportunities for efficient parallelisation.

Extending the concept of elemental procedures from intrinsic to user-defined procedures is very much analogous to, but simpler than, extending the concept of generic procedures from intrinsic to user-defined procedures. Generic procedures were introduced for intrinsic procedures in FORTRAN 77 and extended to user-defined procedures in Fortran 90. Elemental procedures were introduced for intrinsic procedures in Fortran 90 and, because of their usefulness in parallel processing, it is quite natural that they be extended to user-defined procedures in Fortran 95. ELEMENTAL procedures are PURE.

31.1.5 Improved Initialisations

A significant number of useful applications will be facilitated by the ability to perform more complicated calculations when specifying data objects. Allowing a restricted class of nonintrinsic functions in certain specification expressions achieves this goal.

31.1.6 Automatic Deallocation

Automatic deallocation of allocatable arrays provides a more convenient and less error prone method of avoiding memory leaks by removing the burden of manual storage deallocation for allocatable arrays.

The "undefined" allocation status of Fortran 90 meant that an allocatable array could easily get into a state where it could not be further used in any way whatsoever. It could not be allocated, deallocated, referenced, or defined. The result was undefined if the array was used as the argument to the `ALLOCATED` function. Removal of this status provides the user with a safe way of handling unSAVEd allocatable arrays and permits use of the `ALLOCATED` intrinsic function to discover the current allocation status of the array at any time.

31.1.7 New Initialisation Features

The prime motivation for adding a means to specify default initialisation for objects of derived type is a desire to eliminate memory leakage, i.e., situations in which allocated memory becomes inaccessible. This can occur in applications which manipulate objects of derived type with pointer components. Most memory leakage can be avoided if it is possible to specify that pointers be created with an initial status of disassociated.

This standard provides a new intrinsic function, `NULL`, with a single optional argument. `NULL` allows the initial status of a pointer to be specified as disassociated in declarations, structure constructors, or type definitions.

Several alternatives were considered and rejected for a variety of reasons. Most of these reasons involve the problem of disambiguating references to generic procedures; i.e., when a program invokes a generic procedure, which specific procedure is supposed to be invoked? Completely defining a pointer object before using it does not help with the disambiguation problem. Creating a `.NULL.` constant does not provide any way to specify the type and type parameters of the pointer that is returned. A `NULL` function with a `MOLD` argument provides these needed capabilities.

This language extension does not completely solve the memory leakage problem; for that, an automatic destructor is needed which would be invoked for local pointers and structures with pointer components when the procedure in which they are created terminates. Such a facility is not included in this standard; it could be provided automatically by a processor that strove to conserve allocatable memory.

Automatic pointer destructors were not included in this standard because the consequences of inaccessible pointers are not as serious as they are with allocatable arrays which become inaccessible. If a local, unSAVEd pointer is not deallocated and the program exits the procedure in which it is defined, the storage is not recoverable but the pointer is reusable.

For reasons of determinacy and portability, an object for which default initialisation is specified is not allowed to appear in a `DATA` statement. In traditional implementations, a compiler passes initialisation information for nondynamic variables to a loader, which is frequently designed to handle object code from several different languages. It would be difficult to guarantee that initialisation in a `DATA` statement would override the default initialisation specified in a type definition.

31.1.8 Remove Conflicts With IEC 559

Edits to sections 4.3.1 and 7.1.7 removed conflicts in Fortran 90 with the IEC 559 (IEEE 754/854) standard. Previous Fortran standards appeared to prohibit certain numeric values and numeric operations which were valid when using IEC 559 arithmetic. Fortran 90 requires that the value of a positive zero be the same as that of a negative zero, whereas IEC 559 requires that they be different. Fortran 90 prohibits any mathematically invalid operation, whereas IEC 559 requires such operations to produce `+Inf`, `-Inf` or `NaN`s.

These conflicts were removed to allow processors the ability to implement more of the IEC 559 arithmetic model without violating the Fortran standard.

31.1.9 Minimum Width Editing

Certain edit descriptors have been extended to allow formatted output of numeric values, while minimising the amount of "white spac"e produced. This extension permits more information to be presented on a limited width display device (e.g. terminal), without concern about overflowing a user specified field width. Allowing a zero field width to be specified for the `I, B, O, Z` and `F` edit descriptors provides this functionality. A field width specification of zero always specifies a minimum field width; it never specifies suppression of data output or an actual field width of zero.

31.1.10 Namelist

Although not described here, namelist input has been extended to allow comments to follow one or more object name / value pairs on a line. This allows users to document how a namelist input file is structured, which values are valid, and the meaning of certain values. Each line in the namelist input may contain information about name / value pairs. This additional information may improve the documentation of the input file.

31.1.11 `CPU_TIME` Intrinsic Subroutine

A new intrinsic subroutine `CPU_TIME` has been added. This feature is provided to allow the assessment of which processor resources a piece of code consumes during execution. This could be the execution of the whole program or only a small part of it. Additional purposes could be comparing different algorithms on the same computer or trying to discover which parts of a calculation on a computer are most expensive.

31.1.12 `MAXLOC` and `MINLOC` Intrinsics

Fortran 90 specified the `MAXLOC` and `MINLOC` intrinsics with only `ARRAY` and `MASK` arguments. The High Performance Fortran (HPF) group added the `DIM` argument between the original two arguments for consistency with the `MAXVAL` and `MINVAL` intrinsics. An incompatibility between Fortran 90 and HPF results unless `MAXLOC` and `MINLOC` are specified as generic interfaces, each with two specific interfaces: the first matching Fortran 90 and the second adding a non-optional `DIM` argument as the second argument. The Fortran Standards Committee decided that Fortran 95 should allow the `DIM` and `MASK` arguments to be specified in either order. For consistency, this provision for `DIM` and `MASK` to be specified in either order was extended to the other intrinsics which have the same arguments. These intrinsics are `MAXVAL`, `MINVAL`, `PRODUCT`, and `SUM`.

31.1.13 Deleted Features

Any deleted features will appear in an Annex to the standard. The current list of deleted features is:

- real and double precision `DO` control variables,
- branching to an `END IF` from outside its block,
- PAUSE statement,
- `ASSIGN`, assigned `GOTO`, assigned `FORMAT` specifiers,

- cH edit descriptor,
- CHARACTER*<*len*> style declarations.

31.1.14 New Obsolescent Features

There are some new obsolescent features to be added to those brought over from Fortran 90:

- computed GOTO,
- statement functions,
- DATA statements amongst executables,
- assumed-length character functions,
- fixed form source,
- assumed-size arrays (i.e. with '*' in the last dimension).

31.1.15 Language Tidy-ups

Language tidy-ups are mainly concerned with ironing out ambiguities in the Fortran 90 standard and adding extra text to make matters clearer.

32 High Performance Fortran

[Some of this section has been taken from the High Performace Fortran Forum Draft Specification v1.0. Using sections of this report is encouraged by the HPF so long as the following notice is published:

> ©1993 Rice University, Houston Texas. Permission to copy without fee all or part of this material is granted, provided the Rice University copyright notice and the title of this document appear, and notice is given that copying is by permission of Rice University.

]

It is widely recognised that parallel computing represents the only plausible way to continue to increase the processor power that is available to programmers and engineers. In order that this power be embraced by the whole scientific community, the method by which the power is harnessed must be made much more accessible.

On current parallel distributed memory architectures, the language systems provided require the programmer to insert explicit message passing between communicating programs each of which knows only about a subset of the data, even if this data represents a single logical entity such as an array. This approach has often been compared to programming early digital computers in the days before high level languages were developed — it is awkward, time consuming and error prone. Put succinctly:

> Message Passing is the assembler language of parallel programming.

High Performance Fortran (HPF) was developed to help bring distributed memory systems into the general scientific community. Using HPF, programmers can write data parallel programs in much the same way as they write 'normal' sequential high level programs.

Loosely, HPF contains by data distribution directives which allow the user to specify to the compiler how to distribute a single object across the available processors. This distribution is done by the compiler transparently to the programmer, who can manipulate distributed objects as a whole even though they may be partitioned between, say, 100 processors. Thus, HPF programs are much easier to write than are message-passing programs.

HPF is an extension of Fortran 90. The array calculation and dynamic storage allocation features of Fortran 90 make it a natural base for HPF. The new HPF language features fall into four categories with respect to Fortran 90:

- New directives:

  The new directives are structured comments that suggest implementation strategies or assert facts about a program to the compiler. They may affect the efficiency of the computation performed, but do not change the value computed by the program. The form of the HPF directives has been chosen so that a future Fortran standard may choose to include these features as full statements in the language by deleting the initial comment header.

- New language syntax:

  Some new language features, for example, `FORALL`, `WHERE`, new intrinsics, `PURE` and `ELEMENTAL` procedures and the intrinsic functions, which were made first-class language constructs rather than comments because they affect the interpretation of a program.

- Library routines:

  The HPF library of computational functions defines a standard interface to routines that have proven valuable for high performance computing including additional reduction functions, combining scatter functions, prefix and suffix functions, and sorting functions.

- Language restrictions:

  The language has been restricted in a very small number of areas.

  Storage and sequence association is immensely complicated when arrays are stored non-continuously and even on totally separate processors. (If an HPF program is being run over a wide area network (WAN) then rather than occupying two adjacent storage locations in memory, two elements of an array may be separated by 1,000's of miles!) Full support of Fortran sequence and storage association is simply not possible with the data distribution features of HPF. Some restrictions on the use of sequence and storage association are defined. These restrictions may in turn require insertion of HPF directives into standard Fortran 90 programs in order to preserve correct semantics.

32.1 Compiler Directives

HPF includes features which describe the collocation of data (`ALIGN`) and the partitioning of data among memory regions or abstract processors (`DISTRIBUTE`). Compilers may interpret these annotations to improve storage allocation for data, subject to the constraint that semantically every data object has a single value at any point in the program. In all cases, users should expect the compiler to arrange the computation to minimise communication while retaining parallelism.

The model is that there is a two-level mapping of data objects to memory regions, referred to as "abstract processors". Data objects (typically array elements) are first *aligned* relative to one another;

this group of arrays is then *distributed* onto a rectilinear arrangement of abstract processors. (The implementation then uses the same number, or perhaps some smaller number, of physical processors to implement these abstract processors. This mapping of abstract processors to physical processors is language-processor dependent.)

Arrays are aligned with a template, that is to say mapped onto a meshed object. Any number of arrays can be mapped to a template. The idea is to align array elements which are used in the same assignment statement, this will minimise communication between processors. Once all the arrays have been aligned, the template is distributed amongst the processors (declared in a `PROCESSORS` directive). In each dimension the distribution can be:

- □ `BLOCK` or `BLOCK(`$<m>$`)` — if the template has t items and these are to be distributed amongst p processors then each processor receives $\lceil t\rceil/\lceil p\rceil$ elements. It is also possible to specify how many elements each processor received by supplying the optional $<m>$ parameter.

- □ `CYCLIC` or `CYCLIC`LBR$<m>$) — the template items are distributed to the processor either one element at a time or $<m>$ elements at a time in a round robin fashion. Thus the first processor gets the first ($<m>$) element(s) of the template. If any array elements are aligned with these particular template items then they will be assigned to the processor. The second processor will get the next ($<m>$) element(s) of the template and so on. When the last processor has been given template items the whole process wraps around and the first processor receives more template items until the template is exhausted. `CYCLIC` distribution is often used to ensure load balancing.

32.2 Visualisation of Data Directives

Consider the following data layout directives:

```
!HPF$ PROCESSORS P(5,7)
!HPF$ TEMPLATE T(20,20)
      INTEGER, DIMENSION(6,10) :: A
!HPF$ ALIGN A(J,K) WITH T(J*3,K*2)
!HPF$ DISTRIBUTE T(CYCLIC(2),BLOCK(3)) ONTO P
```

These directives will cause the data distribution depicted in Figure 43. It can be seen that the array, `A`, is aligned to the template with a stride, 3 in the first dimension and 2 in the second; every processor owns part of the array although it can be seen that, owing to the `CYCLIC(2)` distribution in dimension 1, the set of array items that a particular processor owns **cannot** be described by a linear function or a *subscript-triplet*:

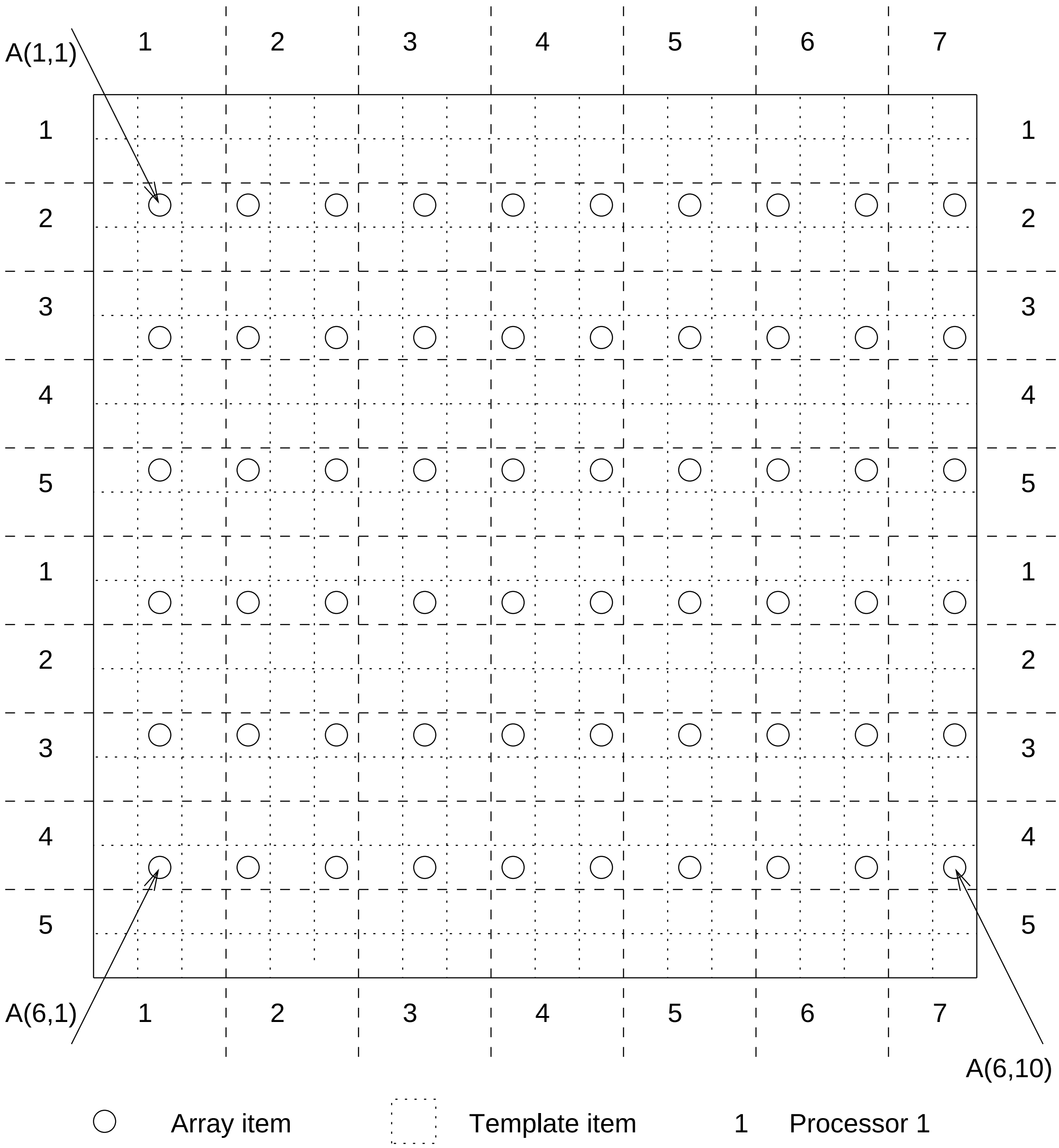


Figure 43: Alignment of a 2-D Array with a 2-D Template and Distribution onto a 2-D Processor Grid.

```
P(1,1)  owns  A(4,1)
P(2,1)  owns  A(1,1)
P(3,1)  owns  A(2,1) and A(5,1)
   :       :       :
P(1,2)  owns  A(4,2) and A(4,3)
P(2,2)  owns  A(1,2) and A(1,3)
P(3,2)  owns  A(2,2), A(2,3), A(5,2) and A(5,3)
   :       :       :
```

It is for this reason that `CYCLIC(M)` distribution results in an inefficient implementation — the compiler must describe the elements that a particular processor owns by a union of subscript triplets instead of the single subscript triplet which can be used for all the other distributions. The target code will have an extra nested loop for each for each dimension with a CYCLIC(M) distribution.

The statement,

```
    A(2,1) = A(5,1)
```

causes no communication because both elements are owned by the same processor but

```
    % A(2,1) = A(2,2)
```

does generate a communication along a row of processors (between P1 and P2 on row 2).

The directives:

```
!HPF$ PROCESSORS P(7)
!HPF$ TEMPLATE T(20,20)
      INTEGER, DIMENSION(6,10) :: A
!HPF$ ALIGN A(J,K) WITH T(J*3,K*2)
!HPF$ DISTRIBUTE T(*,BLOCK(3)) ONTO P
```

demonstrate the collapsing of a template dimension so that a 2 dimensional template can be distributed onto a 1 dimensional processor, see Figure 44.

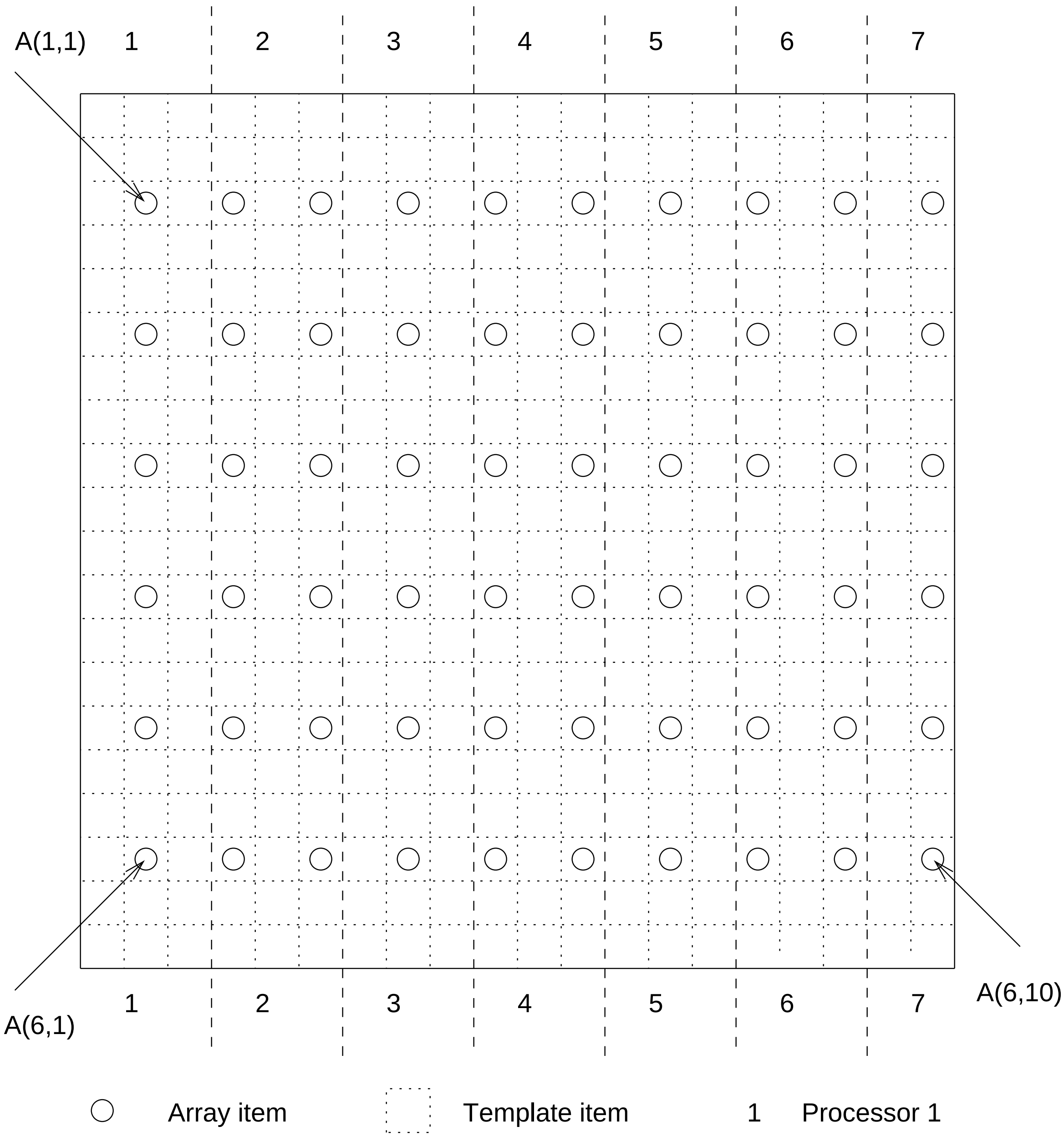


Figure 44: Alignment of a 2-D Array with a 2-D Template and Distribution onto a 1-D Processor Chain.

`P(1)`	owns	`A(:,1)`
`P(2)`	owns	`A(:,2)` and `A(:,3)`
:	:	:
`P(6)`	owns	`A(:,8)` and `A(:,9)`
`P(7)`	owns	`A(:,10)`

It is also possible to collapse one or more dimensions of an array onto a template, see Figure 45:

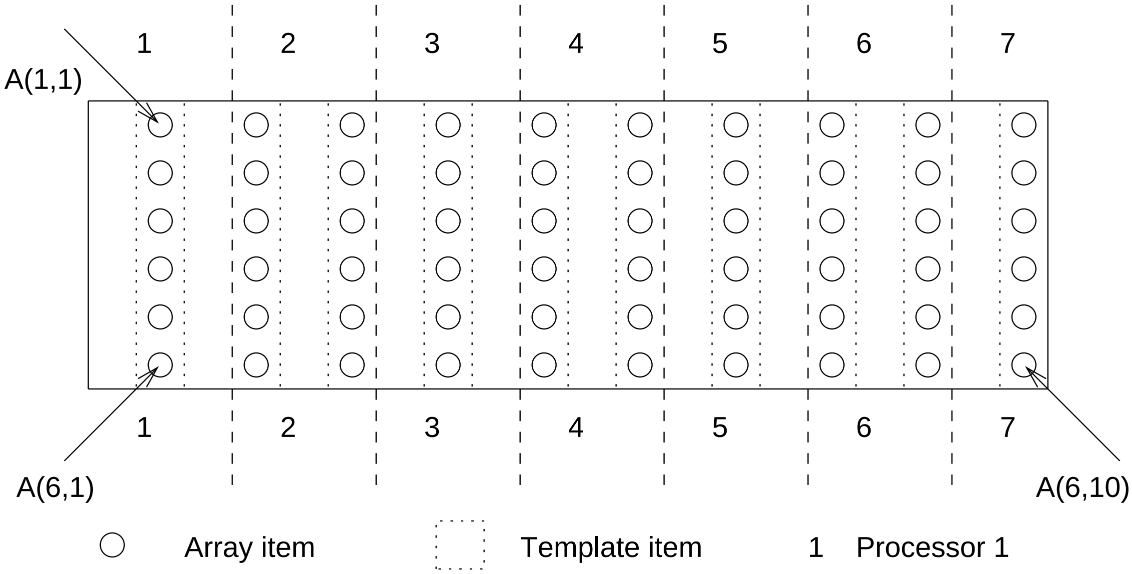


Figure 45: Alignment of a 2-D Array with a 1-D Template and Distribution onto a 1-D Processor Chain.

```
!HPF$ PROCESSORS P(7)
!HPF$ TEMPLATE T(20)
      INTEGER, DIMENSION(6,10) :: A
!HPF$ ALIGN A(*,K) WITH T(K*2)
!HPF$ DISTRIBUTE T(*,BLOCK(3)) ONTO P
```

This data distribution results in the same array elements being present on the same processors as the previous mapping.

33 ASCII Collating Sequence

The following table represents the ASCII character set. At the top of the table are hexadecimal digits (0–7), and to the left of the table are the digits (0–F). To determine the hexadecimal value of a given ASCII character, use the hexadecimal value that corresponds to the row in the "units" position and the digits that corresponds to the column in the "16's" position. For example, the value of the character representing the tilde is 7E.

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

ASCII Collating Sequence

where,

NUL	Null	DLE	Data link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledgement
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tab	EM	End of medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tab	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space	DEL	Delete

References

[1] Anonymous. *Fortran*. ISO/IEC 1539, 1991.

[2] CSEP. Fortran 90 and Computational Science. Technical report, Oak Ridge National Laboratory, 1994.

[3] S. Davis. *C++ Programmer's Companion*. Addison-Wesley, 1993.

[4] High Performance Forum. High Performance Fortran language specification, version 1.1. Technical report, Rice University, May 1993.