THE UNIVERSITY *of* LIVERPOOL

Fortran 90 Programming
(5 Day Course)

Dr. A C Marshall (funded by JISC/NTI)
with acknowledgements to Steve Morgan and Lawrie Schonfelder.

Guide for slide: 0

Lecture 1:
Introduction

2: Course Philosophy

The course:

- assumes a familiarity with a high level language;
- stresses modern scientific programming syntax, for example, array language, modules, defined types, recursion and overloaded operators;
- gives many examples;

Guide for slide: 2

The course:

- assumes familiarity with language such as high level languages such as Basic, ADA, C, Pascal, Algol, FORTRAN 77, APL.
- highlights new Fortran 90 features which make language more robust; for example, of those mentioned on the slide, many are object-based facilities which make it more difficult for the user to make mistakes or do daft things. Fortran 90 has traces of ADA in the sense that there are many many restrictions placed in the standard document meaning that mistakes, which in other languages (e.g., 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, if explicit interfaces are used then its 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.
- see “Programming in Fortran 90” by Morgan and Schonfelder (Alfred Waller Ltd, 1993, ISBN 1-872474-06-3) too. (Recommended text.)

3: Fortran Evolution

History:

- ☐ FORmula TRANslation.
- ☐ first compiler: 1957.
- ☐ first official standard 1972: 'FORTRAN 66'.
- ☐ updated in 1980 to FORTRAN 77.
- ☐ updated further in 1991 to Fortran 90.
- ☐ next upgrade due in 1996 - remove obsolescent features, correct mistakes and add limited basket of new facilities such as `ELEMENTAL` and `PURE` user-defined procedures and the `FORALL` statement.
- ☐ Fortran is now an ISO/IEC and ANSI standard.

Guide for slide: 3

- Fortran stood for *IBM Mathematical FORmula TRANslation System* but this has been abbreviated to *FORmula TRANslation*;
- Fortran was and has always been intended for scientific applications;
- Fortran is the oldest established high level language;
- the first compiler became available in 1957 and was written by IBM. More companies wrote compilers in the 60's but this lead to a number of dialects (by 1963 there were 40 different compilers), so there was a need to standardise the language;
- FORTRAN 77 finally standardised in 1972 (!) (—it is quite common for the Fortran version number to be out of step with the standardisation year).
- owing to standardisation, programs could be moved from platform to platform with little or sometimes no rewriting. This was one of the main reasons why the use of Fortran grew and grew.
- as compiler technology and user demands grew a new standard was required — this was completed in the late '70's and was known as FORTRAN 77 (ANSI X3.9-1978);
- FORTRAN 77 adopted as international standard by ISO in 1980;
- FORTRAN 77 is currently the most widely used "version" of Fortran. Compilers, which usually support a small number of extensions have, over the years, become *very* efficient. The technology which has been developed whilst implementing these compilers is not to be wasted and can still be applied to *all* Fortran 90 programs;
- almost as soon as it became standardised, FORTRAN 77 was outmoded compared to C, Algol, Pascal, APL, ADA etc. 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 meant that all arrays had to have a fixed size which could 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 77 (or Fortran 90) for efficiency.
- as Fortran 90 standard is very large and complex there are a small number of ambiguities / conflicts / grey areas. These anomalies generally only come to light when compliers are developed. Since standardisation many compilers have been under development and a number of conflicts / grey areas / missing / desirable features have been identified. A new draft of Fortran, Fortran 95, will recitify the faults and extend the language in the appropriate areas;
- in Fortran 90 some features of FORTRAN 77 have been replaced by better, safer and more efficient features, some of these are to be removed from Fortran 95 although it is expected that many compilers will still support them as 'extensions'. Tools exist to effect automatic removal of such obsolescent features, examples of these are (Pacific Sierra Research's) VAST90 and (NA Software's) LOFT90. These tools also perform differing amounts of vectorisation (transliteration of serial structures to equivalent parallel structures.)

- Fortran 90 was developed jointly between the ANSI X3 technical subcommittee X3J3 and the ISO/IEC JTC1/SC22/WG5 committee.
- in the last few years the Fortran 90 based de-facto language standard, 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 all of the new syntactic features from HPF.
- 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 procedures (PURE procedures)) which have been adopted by Fortran 95.
- HPF (apart from some features, for example, storage and sequence association) will be a superset of Fortran 95.

4: Drawbacks of Fortran 77

Fortran 77 was limited in the following areas,

1. awkward 'punched card' or 'fixed form' source format;
2. inability to represent intrinsically parallel operations;
3. lack of dynamic storage;
4. non-portability;
5. no user-defined data types;
6. lack of explicit recursion;
7. reliance on unsafe storage and sequence association features.

Guide for slide: 4

Fortran 77 is outmoded — many other languages have been developed which allow greater expressiveness and ease of programming.

1. Fortran 77 awkward 'punched card' source format: each line corresponds to 1 punched card with 72 columns thus each line can only be 72 characters long, this causes problems because text in cols 73 onwards is simply ignored (line numbers used to be placed in these columns). In this day and age this source format 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. 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 and this has been done, to a certain extent, by the introduction of parallel 'array syntax'.

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

4. 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. in Fortran 77 intuitive (user-defined) data types are 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. Fortran 77 lacked 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. 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.

   In Fortran 77 a user could alias arrays using the `EQUIVALENCE` statement which is equally as open to abuse as `COMMON` blocks. A great deal of errors stem for mistakes in these two areas which were generally regarded as unsafe but the was no real alternative in Fortran 77.

5: Fortran 90 New features

Fortran 90 supports,

1. free source form;
2. array syntax and many more (array) intrinsics;
3. dynamic storage and pointers;
4. portable data types (`KIND`s);
5. derived data types and operators;
6. recursion;
7. `MODULE`s

- procedure interfaces;
- enhanced control structures;
- user defined generic procedures;
- enhanced I/O.

Guide for slide: 5

This slide 'answers' the previous one,

1. Fortran 90 defined new code format — much more like every other language;
   - upto 132 columns per line;
   - more than one statement per line;
   - in-line comments allowed — 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 chars);
   - names can be punctuated by underscores making then more readable.
2. parallelism can be expressed using whole array operations including extensive slicing and sectioning facilities, and arithmetic on arrays and array sections; Operations in an array assignment statement are conceptually performed in parallel. This is adapted from APL.

   The masked (parallel) assignment (`WHERE`) statement and construct has also been introduced.
3. 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 `MAXVAL` (scan all elements in an array and return one value — the biggest). This concept comes again from APL.

   The provision of an enhanced set of intrinsics has increased efficiency of the language. Common operations have been identified and new intrinsics such as bit manipulation functions, precision and numeric representation inquiry functions, reshaping and retyping functions and array construction functions, have been specified.

   Many of the FORTRAN 77 intrinsics have been extended to accept whole array arguments.
4. the introduction of dynamic (heap) storage means that pointers can be implemented. Pointers can be used for aliasing arrays and creating dynamic data structures such as linked lists and trees. Temporary arrays can also be created on-the-fly.
5. the `KIND` facility has been introduced. This allows user to specify desired precision in a portable way. The precision of a particular type in the program can be changed by altering the value of one constant. Appropriate intrinsic inquiry functions are defined to query the precision of given objects.
6. user-defined types which are constructed from existing types can now be defined, 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.

   It is possible to define 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 fashion 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).

   An ancillary standard, 'The varying strings module', has been defined and is implemented in a module. When using the module, the user can simply declare an object of the appropriate type (`VARYING_STRING`) which can be manipulated in exactly the same way 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.

7. 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.

8. 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).

   `MODULE`s also provide object oriented 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.

   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.

- procedure interfaces can be declared. The interface specifies to the calling program unit, all pertinent details about the procedure and its arguments, this means:
  - programs 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 across procedure boundaries;
  - 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 — useful for code readability;
  - `SELECT CASE` control block. Present in many other languages and is more succinct, elegant and efficient than an `IF` .... `ELSEIF` ... `ELSEIF` .... `ENDIF` block.
- non-advancing I/O has also been introduced.

6: Advantages of Additions

Fortran 90 is:

- □ more natural;
- □ greater flexibility;
- □ enhanced safety;
- □ parallel execution;
- □ separate compilation;
- □ greater portability;

but is

- □ larger;
- □ more complex;

Guide for slide: 6

This is really a summary slide.

Many of the advantages have been mentioned in the context of the previous slide.

- more natural — for example, we can define data structures to mirror real life objects. The language as a whole is more expressive.
- greater flexibility — semantic extension
- enhanced safety — many 'dangerous' features of FORTRAN 77 have been superseeded by new features of Fortran 90:
  - `COMMON` blocks — use a module;
  - `EQUIVALENCE` — use `TRANSFER` intrinsic; (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;
  - the retyping of arrays across procedure boundaries — use `TRANSFER`;
  - the reliance on labels has been decreased — use new control constructs;
  - fixed form source — use free form;
  - implicit typing — `IMPLICIT NONE` statement introduced which, if present, 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 given at all, if the variable was used then a totally random value (the incumbent value of the memory location where the variable resides) would be used sometimes with disastrous consequences.)
  - internal procedures (procedures within procedures with local scope) alleviate the need for the awkward `ENTRY` statements.
- parallel execution — can take advantage of new advances in hardware and can prototype on sequential systems before moving to a parallel machine;
- separate compilation — makes development easier;
- greater portability — no more rewriting of codes when a port needs to be done;

7: Language Obsolescence

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;

Guide for slide: 7

- All of FORTRAN 77 is included in Fortran 90 but as the language evolves certain features become redundant.
- FORTRAN 77 provided better ways to program certain actions so these actions are to be removed from Fortran.
- They were kept in FORTRAN 77 owing to the existence of dusty deck codes
- It is as well to let programmers know what not to use so they don't!!
- automatic translation of these features is possible — use VAST90 or loft90.

8: Obsolescent Features

The following features are labelled as obsolescent and will be removed from the next revision of Fortran, Fortran 95,

- the arithmetic IF statement;
- ASSIGN statement;
- ASSIGNed GOTO statements;
- ASSIGNed FORMAT statements;
- Hollerith format strings;
- the PAUSE statement;
- REAL and DOUBLE PRECISION DO-loop control expressions and index variables;
- shared DO-loop termination;
- alternate RETURN;
- branching to an ENDIF from outside the IF block;

Guide for slide: 8

As these features will be removed from Fortran they should not be used in programs:

- the arithmetic `IF` statement — use `IF` statement instead;

  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 value is zero, transfer is to the statement labelled < *label2* >, and if it is positive execution transfers 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.

- `ASSIGN` statement;

  used to assign a statement label to an `INTEGER` variable (the label cannot be interpreted as an integer). Used in a `GOTO` or `FORMAT` statement.

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

- `ASSIGN`ed `GOTO` statements;

  Historically this was used to simulate a procedure call before Fortran had such procedures.

- `ASSIGN`ed `FORMAT` statements;

  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 through use of character variables to hold `FORMAT` specifications.

- Hollerith format strings were used to represent stings in a format statement like this

```
      WRITE(*,100)
 100  FORMAT(16HTITLE OF PROGRAM)
```

  use `CHARACTER` strings with single or double quotes now,

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

- the `PAUSE` statement - use a `PRINT` statement and a `READ` statement which waits for input;

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

  `PAUSE` < *stop code* >

  The < *stop code* > is written to the standard output at the `PAUSE` statement.

- REAL and DOUBLE PRECISION DO-loop control expressions and index variables - use integral variables and construct real variable within the loop;

  A loop with real valued DO-loop control expressions could easily loop a different number of times on different machines - a loop with control expression 1.0,2.0,1.0 may loop 1 or 2 times - 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.

- shared DO-loop termination - use separate END DO statements;

  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.

```
    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
```

  Use END DO instead.

- alternate RETURN - use a return code and a GOTO statement or some equivalent control structure.

  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 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
```

- branching to an END IF from outside its block — branch to the next statement instead or else add a CONTINUE statement.

9: Undesirable Features

- fixed source form layout - use free form;
- implicit declaration of variables - use `IMPLICIT NONE`;
- `COMMON` blocks - use `MODULE`;
- assumed size arrays - use assumed shape;
- `EQUIVALENCE` statements;
- `ENTRY` statements;
- the computed `GOTO` statement - use `IF` statement;

Guide for slide: 9

- fixed source form layout - use free form;

  mentioned before - free form is less error prone and more intuitive
- implicit declaration of variables - use `IMPLICIT NONE`;

  mentioned before - saves undeclared variables slipping through
- `COMMON` blocks - use `MODULE`;

  use the more safer `MODULE` program unit - see later.
- assumed size arrays - use assumed shape;

  assumed shape more reliable - see later. Use `TRANSFER` function for changing the type of an object, use `RESHAPE` for changing the shape of an array.
- `EQUIVALENCE` statements;

  this was used to rename and retype an area of storage — use `POINTER` variables for aliasing, use `TRANSFER` function for changing the type of an object
- `ENTRY` statements;

  this was used to 'jump in' to a specified procedure at a specified point in the executable code. Can use internal procedures for this now.
- the computed `GOTO` statement — use `IF` statement instead;

10: Object Oriented Facilities

Fortran 90 has some 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* — user can program his / her own polymorphism by generic overloading;
- *reusability* — Modules;

Guide for slide: 10

FORTRAN 77 had virtually no Object Oriented features, 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.

The object oriented (or object based) facilities available are:

- *data abstraction*
  - user derived types provide a certain degree of abstraction,
  - no enumerated types,
  - a lack of parameterised derived types. 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 KIND. 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 provide data abstraction. 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.
- *data hiding*
  - advanced data hiding, can tag any entity in a module with a visibility attribute,
  - PRIVATE and PUBLIC statements are available but are not necessarily sufficient.
- *encapsulation*
  - very closely aligned with data hiding
  - available through MODULEs / MODULE PROCEDUREs and USE statements and backed up by the renames and ONLY facilities,
- *inheritance* and *extensibility*
  - Fortran 90 supports supertypes — user-defined types can include other defined types,
  - new MODULEs may inherit functionality from previously written MODULEs by using them, however, entities 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 the module under certain circumstances this would allow for much grater extensibility.

- ◇ Fortran 90 does not have subtypes so functional and object inheritance cannot be transmitted this way, this is the main drawback towards OOness, (mind you ADA has subtypes but is still not OO language,
- ◇ being able to define subtypes which inherit functions and the like which are valid for the parent type is a very important aspect of OOness - Fortran 90 does not have this.

□ *polymorphism*

- ◇ the generic capabilities are very advanced, however, polymorphism must be programmed by the user, i.e., specific procedures must be user-written to support the generic interface,
- ◇ Fortran 90 does not support dynamic binding, i.e., the resolution of generic calls at run-time,
- ◇ cannot pass generic procedures as actual procedure arguments — a separate procedure must be exist for each different dummy procedure, (different result type, number of arguments etc).
- ◇ the lack of subtypes rules out many opportunities for polymorphism

□ *reusability*

- ◇ `MODULE`s can be used as libraries of types, object declarations and functionality and can be used in other modules.

There are, however, many shortfalls in its capabilities such as the lack of parameterised derived types

Lecture 2:
Elements of Fortran 90

12: Example

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
```

Guide for slide: 12

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.
- `IMPLICIT NONE`

  `IMPLICIT NONE` — makes declaration of variables compulsory throughout the module. It applies globally.
- `CONTAINS` — specifies that the rest of the `MODULE` contains procedure definitions.
- `FUNCTION Area(x,y,z)`

  this declares a named function and the number and name of its arguments.
- `REAL :: Area ! function type`

  `REAL Area` — `FUNCTION` procedures return a result in a 'variable' which has the same name as the function. 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 should be the same type as the actual arguments.

  `INTENT` — this says how the arguments are to be used.

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

  Specifying the `INTENT` is not compulsory but is good practise as it can highlight errors.
- `REAL :: theta, height`

  final `REAL` statement declares local variables for use in the `FUNCTION` — they cannot be accessed in the calling program,
- `theta = ACOS((x**2+y**2-z**2)/(2.0*x*y))`

  assignment 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 obtain 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`

  Two statements on one line. The `;` indicates that a new statement follows on the same line. Normally there is only one per line.

  The function variable `Area` must be assigned to or else the function will not return a result.

- the `MODULE` program unit terminates with a `END MODULE` statement.

- `PROGRAM Triangle`

  `PROGRAM` statement — don't strictly need one but it is good practice. One per program.

- `USE Triangle_Operations` — attaches the `MODULE` to the program and allows the `FUNCTION` to be used by the program.

- `IMPLICIT NONE`

  `IMPLICIT NONE` — makes declaration of variables compulsory;

- `REAL :: a, b, c, Area`

  `REAL` — declaration of real valued objects. `a`, `b` and `c` are variables, `Area` is a function procedure. This function must be given a type because its name contains a value.

- `PRINT *, 'Welcome, please enter the &` ...

  `PRINT` statement outputs the string in quotes to standard output channel (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 continue the previous line at the character following the `&`. If this symbol was at the start of the line, or if there were no `&` on the second line, the string would have a large gap in it because the indentation would be considered as part of the string.

- `READ *, a, b, c`

  `READ` — 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)`

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

  The `''` are transformed, on output, into a single `'`, a single quote in their place would not be output because the compiler would think it was the end of the string and would flag the character `s` as an error. The string could be enclosed by `"`s which would allow a single `'` to be used within the string. The same delimiter must be used at each end of the string.

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

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

- `END PROGRAM Triangle`

  The `END PROGRAM` statement terminates a program.

13: Coding Style

It is recommended that the following coding convention is adopted:

- *always* use `IMPLICIT NONE`.
- Fortran 90 keywords, intrinsic functions and user defined entities should be in upper case,
- other 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,

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

Guide for slide: 13

- Fortran 90 keywords are things such as `PROGRAM`, `INTEGER` or `WHERE`.
- user entities are variables and constants,
- 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,
- although not required here, the `ONLY` clause will give the compiler the option of only linking the code associated with the objects in the `ONLY` clause instead of the whole module.

14: Source Form

Free source form:

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

Example,

```
PRINT*, "This line is continued &
        &On the next line"; END ! of program
```

Guide for slide: 14

Can be mentioned with respect to the Triangle Example slide.

- the character set now includes amongst others: `&`, `$` and `;`.
- Fortran 90 has two 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.
- if a '`!`' appears on a line (but not within a string) then any following text on the same line is treated as a comment.
- the old "character in column 6" method of line continuation is replaced by an `&` at the end of a line.

  `&` is ineffective within a comment and cannot be on a line on its own.
- the '`;`' allows two statements (or more) to occupy the same line.
- in the new free-format blanks are significant. They cannot be embedded into names or keywords but there can be a blank between and `END` statement and the construct name. In general using common sense will suffice.

15: Character Set

The following are valid in a Fortran 90 program:

- alphanumeric:

  `a-z`, `A-Z`, `0-9`, and _ (the underscore)

- symbolic:

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		

Guide for slide: 15

- the _ (the underscore) is useful for punctuating object names.
- although both upper and lower case letters can be used there is no difference and the language is not case sensitive. `integer` has exactly the same meaning as `INTEGER`.

  It is good practice to use upper case for keywords and lower case for user defined names. Some may advise that matrix names be in upper case!
- In general the alphanumeric characters can be used in object names and the other symbols are used as delimiters or in mathematical expressions.
- `$` and `?` have no real meaning in the language.
- all characters can be embedded within a string.

16: Significance of Blanks

In *free form* source code blanks must not appear:

- within keywords

```
INTEGER :: wizzy       ! is a valid keyword
INT EGER :: wizzy      ! is not
```

- within names

```
REAL :: running_total  ! is a valid name
REAL :: running total  ! 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 some keywords mainly '`END` <*construct*>' and a few others; *if in doubt add a blank* (it looks better too).

Guide for slide: 16

- Blanks were not significant to FORTRAN 77 fixed format - they are to Fortran 90 free format.
- Consecutive spaces/blanks have the same meaning as one blank.
- cannot place them
  - within names, e.g., `INT EGER`
  - within keywords, e.g., `running total`
- must place them
  - between 2 keywords so the compiler can tell where one finishes and the other starts
  - between keywords and names for the same reason
- 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 in other situations. Adding a blank between two valid English words is generally a good idea and will be valid.

17: Comments

It is good practise to use lots of comments, for example,

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

- everything after the `!` is a comment;
- the `!` in a character context does **not** begin a comment, for example,

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

Guide for slide: 17

- In free format the ! comment initiator is the only way of introducing a comment the C in column 1 is meaningless because column numbers are not counted.
- comments, however brief, should be heartily encouraged!
- everything on a line after the ! is ignored.
- comments cannot be continued onto the next line - must re-initiate them.
- the ! in a character context is teated like any other character and has no special meaning.
- as example indicates, some compilers interpret and act upon structured comments - this is clever as other compilers will ignore them so they don't change the meaning of the program. The compiler manual will describe this form of commentary which is not officially part of the language.

18: Names

In Fortran 90 variable names (and procedure names etc.)

- must start with a letter

```
REAL :: a1 ! valid name
REAL :: 1a ! not valid name
```

- may use only letters, digits and the underscore

```
CHARACTER :: atoz ! valid name
CHARACTER :: a-z  ! not valid name
CHARACTER :: a_z  ! OK
```

- underscore should be used to separate words in long names

```
CHARACTER(LEN=8) :: user_name ! valid name
CHARACTER(LEN=8) :: username  ! different name
```

- may not be longer than 31 characters

Guide for slide: 18

- Fortran 90 defines a number of names or keywords. Unlike most languages these keywords may also be used as variable or procedure names although this practise is clearly a terrible idea!
- names must start with `a-z` or `A-Z`. Remember, case is NOT significant. Some compilers allow a $ or an _ to be used but this is `not` standard conforming.
- as bullet says some compilers allow a $ or an _ to be used but this is `not` standard conforming.
- This kind of punctuation (_) should be encouraged for readability
- user defined names can be up to 31 chars.
- With the new long names facility in Fortran 90 — symbolic names should be significant and descriptive. Its worth spending a minute or so choosing a good name which other programmers will be able to understand.

19: Statement Ordering

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>

Guide for slide: 19

The table (or one very like it) can be found in the Fortran 90 standard.

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 header statement must come first, (`PROGRAM`, `FUNCTION` or `SUBROUTINE`). `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 end with an `END` statement.

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.

- `PROGRAM`, `FUNCTION`, `SUBROUTINE`, `MODULE` or `BLOCK DATA` statement:

  These describe a *program unit*.

  - There can be only 1 main `PROGRAM`,
  - there may be many uniquely named `FUNCTION`s and `SUBROUTINE`s program units (procedures).
  - there may be any number of uniquely names `MODULE`s 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 be come *use-associated* with the program unit. When a module is used its 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.

- `IMPLICIT NONE` - should be after `USE` statement - its use is implored.
- `FORMAT` and `ENTRY` - 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 typing 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. Dangerous feature. Cannot have an IMPLICIT statement if there is an IMPLICIT NONE line.)

  PARAMETER statements - its is suggested that the attributed form of PARAMETER declaration be used - i.e., where the PARAMETER descriptor is placed in the object definition statement.

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

- Declarations - obviously any object that is to be used in a declaration must itself have already been defined.

  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.

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

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

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

20: Intrinsic Types

Fortran 90 has three broad classes of object type,

- character;
- boolean;
- numeric.

these give rise to six simple intrinsic types, known as default types,

```
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
```

Guide for slide: 20

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 classes cannot be intermixed without explicit coercion being performed.
- NOTE that `REAL` and `DOUBLE PRECISION` objects are approximate. `DOUBLE PRECISION` should not really 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.
- the above are default types, non-default types are types of different kinds.

21: Literal Constants

A literal constant is an entity with a fixed value:

```
12345      ! INTEGER
1.0        ! REAL
-6.6E-06   ! REAL: -6.6*10**(-6)
.FALSE.    ! LOGICAL
.TRUE.     ! LOGICAL
"Mau'dib"  ! CHARACTER
'Mau''dib' ! CHARACTER
```

Note,

- there are only two `LOGICAL` values;
- `REAL`s contain a decimal point, `INTEGER`s do not,
- `REAL`s have an exponential form
- character literals delimited by `"` and `'`;
- two occurrences of the delimiter inside a string produce one occurrence on output;
- there is only a finite range of values that numeric literals can take.

Guide for slide: 21

- 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 — it will be different for each compiler.
- constants may also include a kind specifier but this is described later.

22: Implicit Typing

Undeclared variables have an implicit type,

- if first letter is `I, J, K, L, M` or `N` then type is `INTEGER`;
- any other letter then type is `REAL`s.

Implicit typing is potentially very dangerous and should **always** be turned off by adding:

```
    IMPLICIT NONE
```

as the first line after any `USE` statements.
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!

Guide for slide: 22

- if the first letter of an undeclared variable is `I, J, K, L, M` or `N` the variable is taken to be an `INTEGER` other wise it is `REAL`.
- `IMPLICIT NONE` should be used as a matter of course. Not using it is asking for trouble as undeclared variables slip through.
- the type mapping of the alphabetic letters can be changed by using the `IMPLICIT` statement.
- the example sighted is reputed to have caused the crash of the American Space Shuttle. An expensive missing comma!

23: Numeric and Logical Declarations

With `IMPLICIT NONE` variables must be declared. A simplified syntax follows,

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

The following are all valid declarations,

```
REAL                    :: x
INTEGER                 :: i, j
LOGICAL, POINTER        :: ptr
REAL, DIMENSION(10,10)  :: y, z
INTEGER                 :: k = 4
```

The `DIMENSION` attribute declares an array ($10 \times\ 10$).

Guide for slide: 23

- in Fortran 90 objects can be attributed at the same time as being declared.
- < *attribute*-list > includes `PARAMETER`, `SAVE`, `INTENT`, `POINTER`, `TARGET`, `DIMENSION`, (for arrays) or visibility attributes — all described later.
- the `::` is actually optional, however, it does no harm to use it!
- If < *attribute*-list > is present then so must be `::`.
- an object may be given more than one attribute per declaration but some cannot be mixed (such as `PARAMETER` and `POINTER`).
- variables may be initialised.

24: Character Declarations

Character variables are declared in a similar way to numeric types. 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.

The following are all valid declarations,

```
CHARACTER(LEN=10)  :: name
CHARACTER          :: sex
CHARACTER(LEN=32)  :: str
CHARACTER(LEN=10), DIMENSION(10,10) :: Harray
CHARACTER(LEN=32), POINTER :: Pstr
```

Guide for slide: 24

- Default `CHARACTER` s have 1 character.
- `CHARACTER(LEN= )` is the same as `CHARACTER*` format
- `CHARACTER` declarations may be attributed, for example, the `POINTER` and `DIMENSION` attributes.
- there is a notable difference between a character variable of 10 characters (`CHARACTER(LEN=10)` or `CHARACTER*10`) and a character array of 10 elements. The first is scalar, the second is non-scalar.

25: Constants (Parameters)

Symbolic constants, oddly known as *parameters* in Fortran, can easily be set up either in an attributed declaration or parameter statement,

```
REAL, PARAMETER :: pi = 3.14159
CHARACTER(LEN=*), PARAMETER :: &
                  son = 'bart', dad = "Homer"
```

`CHARACTER` constants can assume their length from the associated literal (`LEN=*`).
Parameters should be used:

- ☐ if it is known that a variable will only take one value;
- ☐ for legibility where a 'magic value' occurs in a program such as π;
- ☐ for maintainability when a 'constant' value could feasibly be changed in the future.

Guide for slide: 25

- it is recommended that the attributed form is used
- `CHARACTER` parameters can assume their length. Can retrieve this length using `LEN` intrinsic function.

Parameters should be used:

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

  forces it 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 in the future

  Allows it to be adjusted throughout the program by changing 1 line.

26: Initialisation

Variables can be given initial values:

- can use *initialisation expressions*,
- may only contain `PARAMETER`s or literals.

```
REAL             :: x, y =1.0D5
INTEGER          :: i = 5, j = 100
CHARACTER(LEN=5) :: light = 'Amber'
CHARACTER(LEN=9) :: gumboot = 'Wellie'
LOGICAL  :: on = .TRUE., off = .FALSE.
REAL, PARAMETER :: pi = 3.141592
REAL, PARAMETER :: radius = 3.5
REAL :: circum = 2 * pi * radius
```

`gumboot` will be padded, to the right, with blanks.
In general, intrinsic functions *cannot* be used in initialisation expressions, the following can be: `REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRANSFER`, `TRIM`, `LBOUND`, `UBOUND`, `SHAPE`, `SIZE`, `KIND`, `LEN`, `BIT_SIZE` and numeric inquiry intrinsics, for, example, `HUGE`, `TINY`, `EPSILON`.

Guide for slide: 26

- some compilers zero all objects when the program is loaded; most just allow the variables to contain jibberish, it is good practise to initialise variables.
- variables can be initialised in a number of ways
  - `PARAMETER` statements
  - `DATA` statements
  - type declaration statements (with an `=`< *expression* >) clause.
- can be initialised with constants (literals or `PARAMETER`s)
- `CHARACTER` s – 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.
- can do many initialisations per line.
- constants on LHS of assignments must be able to be represented by the data type on the RHS
- if `PARAMETER` attribute occurs then objects must be initialised
- dummy args cannot be initialised.
- `::` must appear if `=` does.
- many attributes cannot appear with `=`, e.g., `ALLOCATABLE`.
- all objects that are initialised have a static storage class, it is as if they had the `SAVE` attribute,

Initialisation expressions,

- all information used in an initialisation expression must be available - if you can't see it then neither can the compiler.
- arrays may be initialised by specifying (conformable) scalar value or by using an array constructor.

27: Expressions

Each of the three broad type classes has its own set of intrinsic (in-built) operators, for example, `+`, `//` and `.AND.`,
The following are valid expressions,

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

Expressions can be used in many contexts and can be of any intrinsic type.

Guide for slide: 27

- Expressions are made from one operator (e.g., +, -, *, /, // and **) and at least one operand.
- Operands are expressions.
- expressions have types - one of the intrinsic types or a user defined types.
- operators can be monadic and dyadic (take 1 or 2 operands)
- intrinsic operators can be overloaded to apply in more situations.

Intrinsic operators are defined for intrinsic types,

- numeric intrinsic operators are
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `**` exponentiation
- `CHARACTER` has only `//` concatenation operator
- `LOGICAL` has all expected boolean operators

User defined operators can be defined to do virtually anything (with 1 or 2 operands) - can be applied to any type or combination of types.

They can be very powerful when used in conjunction with derived types and modules as a package of objects and operators.

28: Assignment

Assignment is defined between all expressions of the same type:
Examples,

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

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

Guide for slide: 28

Assignment:

- `=` is a dyadic operator
- is defined between all numeric types. The two operands of `=` do not have to be the same intrinsic type.
- is defined between two objects of the same type and for all other types (including user types).
- assignment can be overloaded by writing a procedure with two arguments of the desired type which will define what assignment means between two object of these types. Assignment overloading is discussed later.
- can use with overloded operators in a module to provide semantic extension, e.g., varying strings module.

NOTE: `SIN` works in radians,

29: Intrinsic Numeric Operations

The following operators are valid for numeric expressions,

- `**` exponentiation, dyadic 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, monadic and dyadic operators, for example, `10+7-4` and `-3`;

Can be applied to literals, constants, scalar and array objects. The only restriction is that the RHS of `**` must be scalar.
Example,

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

Guide for slide: 29

Numeric:

- numeric operators behave exactly like you would expect - must take into account the type of the object where the result will be stored (See later.)
- `**` is only operator that is evaluated from right to left, i.e., the exponent is worked out first, all others are evaluated from left to right.
- cannot divide by zero (as usual).
- + and - take 1 or 2 operands
- LHS of `**` can be a scalar, whole array or a subsection of an array - see later.
- operators have a predefined precedence, i.e., what order they are evaluated in, see later.

The example demonstrates,

- `a = b-c` the `-` operator applied to two objects (arrays or scalar),
- `f = -3*6/5` the use of a monadic `-` operator. verb f is either array or scalar (can't tell).

30: Relational Operators

The following *relational operators* deliver a LOGICAL result when combined with numeric operands,

.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,

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

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.
```

Guide for slide: 30

Relational operators

- compare the values of two operands
- deliver a logical result
- can be applied to numeric operands (restrictions on `COMPLEX` — 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. Them operators refer to ASCII order.
- cannot be applied to `LOGICAL` objects,

The example demonstrates,

- simple logical assignment using a relational operator,
- an `IF` statement using a relational operator,

31: Intrinsic Logical Operations

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

- `.NOT.` — `.TRUE.` if operand is `.FALSE.`.
- `.AND.` — `.TRUE.` if both operands are `.TRUE.`;
- `.OR.` — `.TRUE.` if at least one operand is `.TRUE.`;
- `.EQV.` — `.TRUE.` if both operands are the same;
- `.NEQV.` — `.TRUE.` if both operands are different.

For example, if `T` is `.TRUE.` and `F` is `.FALSE.`

- `.NOT. T` is `.FALSE.`, `.NOT. F` is `.TRUE.`.
- `T .AND. F` is `.FALSE.`, `T .AND. T` is `.TRUE.`.
- `T .OR. F` is `.TRUE.`, `F .OR. F` is `.FALSE.`.
- `T .EQV. F` is `.FALSE.`, `F .EQV. F` is `.TRUE.`.
- `T .NEQV. F` is `.TRUE.`, `F .NEQV. F` is `.FALSE.`.

Guide for slide: 31

- logical operators can only be used between `LOGICAL` expressions.
- have a precedence (see later),
- can be used in any `LOGICAL` context for example, in an `IF` statement.

32: Intrinsic Character Operations

Consider,

```
CHARACTER(LEN=*), PARAMETER :: str1 = "abcdef"
CHARACTER(LEN=*), PARAMETER :: str2 = "xyz"
```

substrings can be taken,

- str1 is 'abcdef'
- str1(1:1) is 'a' (**not** str1(1) — illegal)
- str1(2:4) is 'bcd'

The concatenation operator, //, is used to join two strings.

```
PRINT*, str1//str2
PRINT*, str1(4:5)//str2(1:2)
```

would produce

```
abcdefxyz
dexy
```

Guide for slide: 32

Substrings are dealt with in more detail later, the example selects

- the whole string
- the first character
- the 2nd, 3rd and 4th characters
- the substring must be specified *from* a position *to* a position, a single subscript is no good.

There is only one intrinsic `CHARACTER` operation `//` which concatenates two strings.

- `//` cannot be mixed with other types
- the object `name` *must* be a `PARAMETER` in order to appear in an initialisation expression.
- both operands must be of the same kind (see later).

33: Operator Precedence

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

Note:

- 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 **.

Guide for slide: 33

- Intrinsically defined order cannot be changed.
- User defined operators have a fixed precedence - cannot change this.
- the order is fairly intuitive,
- we say that the operator with the highest precedence has the *tightest binding*.
- evaluation order can be forced by using parenthesis - expressions in the brackets are evaluated first,
- two operators cannot be adjacent. one times minus one is written `1*(-1)` and not `1*-1`
- in contexts of equal precedence left to right evaluation is performed except for `**` where its is right to left. This is important when teetering around the limits of the machines representation. (See later.) 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.
- have to be careful with the division operator - it is good practice to put numerator and denominator in parentheses.

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

  is not the same as

  ```
  A+B/C
  ```

  but

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

  is equivalent to

  ```
  A*B/C
  ```

34: Precedence Example

The following expression,

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

is equivalent to

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

as ** is highest precedence. This is equivalent to

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

as / and * are next highest. The remaining operators' precedences are equal, so we evaluate from left to right.

Guide for slide: 34

The precedence is worked out as follows

1. in an expression find the operator(s) with the tightest binding
2. if there is 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 tightest binding operator and follow the same procedure.

First example,

- tightest binding operator is `**` — `c**d` is first executed subexpression - surround with brackets,
- `/` and `*` are the second: `b/5.0` is evaluated next followed by `1*e` - put these is brackets
- `+` and `-` are next in line - they are executed from left to right.
- at last the assignment is done.

35: Precision Errors

Since real numbers are stored approximately:

- every operation yields a slight loss of accuracy,
- after many operations such 'round-off' errors build up,
- catastrophic accuracy loss can arise when values that are almost equal are subtracted, leading digits are cancelled and a rounding errors become visible.

Consider,

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

May produce:

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

which is 30% in error.
A whole branch of Numerical Analysis is dedicated to minimising this class of errors in algorithms. Be careful!

Guide for slide: 35

- 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 stored as `.123457` and `.123445` respectively, whereas the true difference is `.100000E-4` the representation may give `.130000E-4`, a 30% error.
- A whole branch of Numerical Analysis is dedicated to minimising this class of errors in algorithms.

Lecture 3:
Control Constructs, Intrinsics and Basic I/O

37: Control Flow

Control constructs allow the normal sequential order of execution to be changed.
Fortran 90 supports:

- conditional execution statements and constructs, (`IF` ... and `IF` ... `THEN` ... `ELSE` ... `END IF`);
- loops, (`DO` ... `END DO`);
- multi-way choice construct, (`SELECT CASE`);

Guide for slide: 37

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.

- conditional execution constructs, (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 END IF 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 (possibly) 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.

  END IF terminates an IF block.

  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);

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

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

  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 were more efficient.

There are also two more minor cases:

- unconditional jump statements, (GOTO);

  Direct jump to a labelled line - very powerful, 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, should the end of a file be encountered or should there be an unexpected end of record.

It is always good practice to use at least the `ERR=` qualifier in I/O statements and does not harm things to comprehensively guard against any other such errors. See later for more about I/O.

38: IF Statement

Example,

```
IF (bool_val) A = 3
```

The basic syntax is,

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

If < *logical-expression* > evaluates to `.TRUE.` then execute < *exec-stmt* > otherwise do not.
For example,

```
IF (x .GT. y) Maxi = x
```

means 'if `x` is greater than `y` then set `Maxi` to be equal to the value of `x`'.
More examples,

```
IF (a*b+c <= 47) Boolie = .TRUE.
IF (i .NE. 0 .AND. j .NE. 0) k = 1/(i*j)
IF (i /= 0 .AND. j /= 0) k = 1/(i*j) ! same
```

Guide for slide: 38

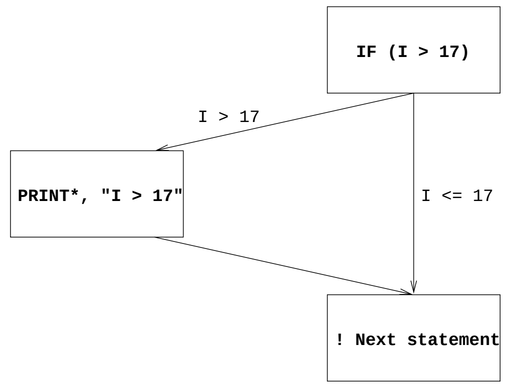
- The basic scenario is if the < *logical-expression* > evaluates to `.TRUE.` then the < *exec-stmt* > is executed, otherwise it isn't!
- The < *exec-stmt* > cannot be an `IF` statement or a statement that ends a program unit.
- This is the most basic form of conditional execution in that there is only an option to execute one statement or not — the general `IF` construct allows a block of code to be executed.
- 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.
- The < *logical-expression* > is any expression that evaluates to either `.TRUE.` or `.FALSE.`. This has been described earlier. Relational operators are very often used here.
- 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 (using the `EPSILON` function — see later).

39: Visualisation of the IF Statement

Consider the IF statement

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

this maps onto the following control flow structure,



Guide for slide: 39

40: IF ... THEN ... ELSE Construct

The block-IF is a more flexible version of the single line IF. A simple example,

```
    IF (i .EQ. 0) THEN
     PRINT*, "I is Zero"
    ELSE
     PRINT*, "I is NOT Zero"
    ENDIF
```

note the how indentation helps.
Can also have one or more ELSEIF branches:

```
    IF (i .EQ. 0) THEN
     PRINT*, "I is Zero"
    ELSE IF (i .GT. 0) THEN
     PRINT*, "I is greater than Zero"
    ELSE
     PRINT*, "I must be less than Zero"
    ENDIF
```

Both ELSE and ELSEIF are optional.

Guide for slide: 40

The first example is a 2-way choice. If `x` is zero then the first `PRINT` is executed otherwise the second is executed instead.

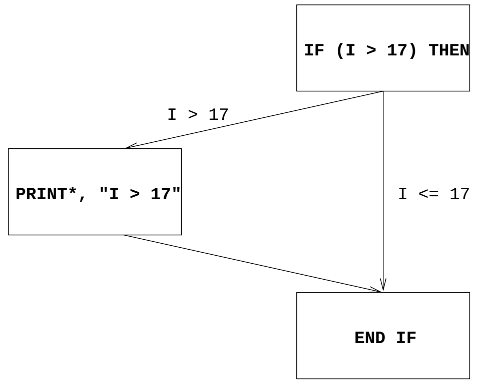
The second example demonstrates `ELSEIF` clauses. If `i .EQ. 0` is not true we must look at whether `i .GT. 0` is. If this is not true we jump to the `ELSE` clause.

41: Visualisation of the IF ... THEN Construct

Consider the IF ... THEN construct

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

this maps onto the following control flow structure,



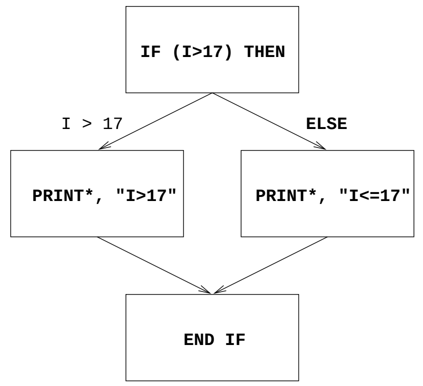
Guide for slide: 41

42: Visualisation of the IF ... THEN ... ELSE Construct

Consider the IF ... THEN ... ELSE construct

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

this maps onto the following control flow structure,



Guide for slide: 42

43: IF ... THEN .... ELSEIF Construct

The `IF` construct has the following syntax,

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

The first branch to have a true < *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
     A = B*C-D
    ELSEIF (x .EQ. 2) THEN
     A = B*B
    ELSE
     IF (y .NE. 0) A=B
    ENDIF
```

`IF` blocks may also be nested.

Guide for slide: 43

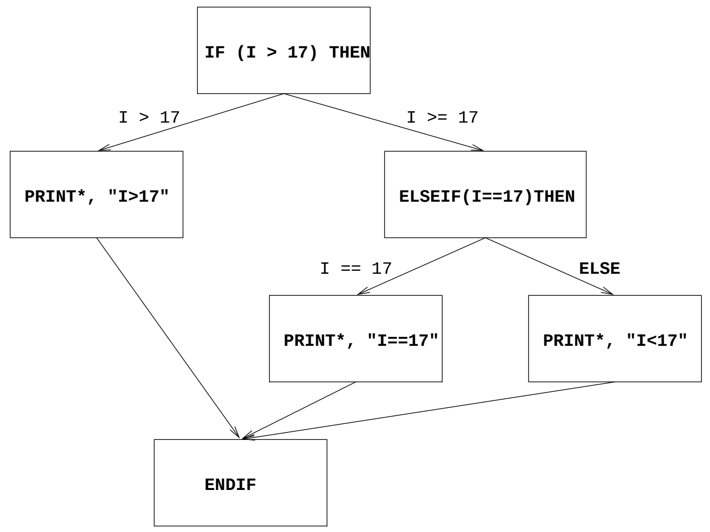
- The block-`IF` is more flexible than the single statement `IF` as there can be a number of alternative mutually exclusive branches guarded by (different) predicates. The control flow is a lot more structured than if a single statement `IF` plus `GOTO` statements were used. The scenario is that the 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.
- 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.
- A further `IF` construct may appear in the < *then-block* >, the < *else-block* > or the < *elseif-block* >. This is now a nested `IF` structure.
- Please note that both the < *else-block* > and the < *elseif-block* > are optional.
- 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* >.

44: Visualisation of IF ... THEN .. ELSEIF Construct

Consider the IF ... THEN ... ELSEIF construct

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

this maps onto the following control flow structure,



Guide for slide: 44

45: Nested and Named IF Constructs

All control constructs can be both named and nested.

```
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
```

The names may only be used *once* per program unit and are only intended to make the code clearer.

Guide for slide: 45

- 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).
- The names are only intended to make the code clearer and serve no real purpose (*cf* `DO`-loop names which do).
- The name is declared implicitly by naming a construct, names cannot hold any values, be assigned to or be used in any other context.
- 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.
- cannot have a statement that ends a program unit within an `IF` block.

46: Conditional Exit Loops

Can set up a DO loop which is terminated by simply jumping out of it. 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
```

The EXIT statement tells control to jump out of the current DO loop.

Guide for slide: 46

- 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`.
- the `EXIT` statement tells control to jump out of the **current** `DO` loop, i.e., to the statement after the next `END DO`.
- 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.
- 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.

47: Conditional Cycle Loops

Can set up a DO loop which, on some iterations, only executes a subset of its statements. Consider,

```
  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
```

CYCLE forces control to the **innermost** active DO statement and the loop begins a new iteration.

Guide for slide: 47

- 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.
- If executed, the statement:

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

  transfers control to the innermost `DO` statement which begins a new iteration.
- a `CYCLE` or `EXIT` statement which is not within a loop is an error.

48: Named and Nested Loops

Loops can be given names and an `EXIT` or `CYCLE` statement can be made to refer to a particular loop.

```
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.
Loop names can only be used once per program unit.

Guide for slide: 48

- It is possible to jump out of more than the innermost loop. If the loops are named then it can be specified which loop it is wished to exit.
- In the example the loops are named in the same way as `IF` statements. If the `DO` statement has a name then so must the `END DO`.
- Unlike `IF` construct names the `DO`-loop names do serve a syntactic purpose, they can be used to modify control flow.
- `IF (a.GT.b) EXIT outa` causes control to immediately jump to the last statement shown, the `END DO`.
- `IF (c.GT.d) EXIT inna` jumps to the statement after `END DO inna`.
- if the name is missing then the directive applied, as usual, to the next outermost loop.

49: 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
```

Guide for slide: 49

- This is an alternative wording for

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

  and is in the language because it was a common language extension to FORTRAN 77 which was extensively implemented and used.

- It is a matter of taste which version is used and there is unlikely to be a difference in implementation speed.
- `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.

50: Indexed DO Loops

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

```
DO i1 = 1, 100, 1
  ... ! i is 1,2,3,...,100
  ... ! 100 iterations
END DO
```

The formal syntax is as follows,

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

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

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

If this is zero or negative then the loop is not executed.
If < *expr3* > is absent it is assumed to be equal to 1.

Guide for slide: 50

- 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 loop can be named and the < *exec-stmts* > could contain `EXIT` or `CYCLE` statements.
- the loop runs from < *expr1* > to < *expr2* > in steps of < *expr3* >.
- an indexed loop **cannot** have a `WHILE` clause, however, this can be simulated with an `EXIT` statement if desired.
- Note: Morgan and Schonfelder book is wrong with iteration count.
- if the < *expr3* > (stride) is missing it is assumed to be 1 this is the only index specifier that is optional.
- it seem to be a common misconception that Fortran loops always have to be executed once; this came from FORTRAN 66 and is now incorrect. Zero executed loops are useful for programming degenerate loops.
- the iteration count is worked out as follows (adapted from the standard, [**?**]):

  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 on the slide)

- then 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>
  4. control passes to the top of the loop again and the cycle begins again

- note how the `DO` variable can be greater than < *expr2* >.
- the `DO` variable cannot be assigned to within the loop.

51: Examples of Loop Counts

A few examples of different loops,

1. upper bound not exact,

```
loopy: DO i = 1, 30, 2
 ... ! i is 1,3,5,7,...,29
 ... ! 15 iterations
END DO loopy
```

2. negative stride,

```
DO j = 30, 1, -2
 ... ! j is 30,28,26,...,2
 ... ! 15 iterations
END DO
```

3. a zero-trip loop,

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

4. missing stride — assume it is 1,

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

Guide for slide: 51

1. upper bound not exact,

   According to the rules (given with the previous slide) 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,

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

3. a zero-trip loop,

   This is a false example in the sense that the loop bounds are literals and (it is hoped) there would be no point in coding a loop of this fashion as it would never ever be executed! However, if symbolic bounds were to evaluate to these values, then, the loop would not be executed. 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,

   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.

52: Scope of DO Variables

1. I is recalculated at the top of the loop and then compared with < *expr2* >,
2. if the loop has finished, execution jumps to the statement after the corresponding END DO,
3. I retains 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
```

An index variable may not have its value changed in a loop.

Guide for slide: 52

Fortran 90 is not block structured, all `DO` variables are visible after the loop and have a specific value. The example here demonstrates this well, the loop is executed three times and `I` is assigned to 4 times.

`I` can be freely used elsewhere in the program and used in other loops if desired.

53: SELECT CASE Construct I

Simple example

```
    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
```

An IF .. ENDIF construct could have been used but a SELECT CASE is neater and more efficient. Another example,

```
    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 >= 10 .AND. num <= 65) .OR. ...
        PRINT*, "Bow wow"
      CASE DEFAULT
!     ELSE
        PRINT*, "Meeeoow"
     END SELECT
!    ENDIF
```

Guide for slide: 53

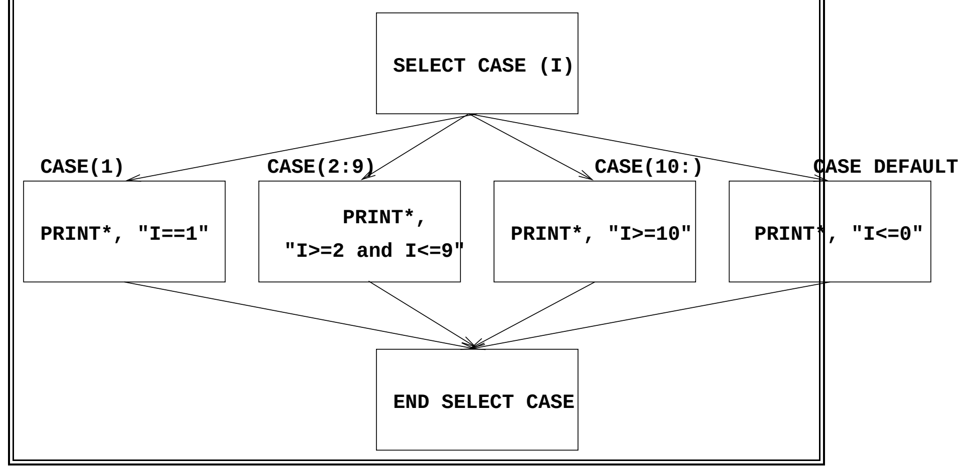
- the < *case-expr* > in this case is `num`.
- the first < *case-selector* >, `(6,9,99,66)` means "if num is equal to either 6, 9, 66 or 99 then",
- the second < *case-selector* >, `(10:65,67:98)` means "if num is between 10 and 65 (inclusive) or 67 and 98 (inclusive) then",
- if a case branch has been executed then 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. `(100:)` specifies the range of greater than or equal to one hundred. If `num` satisfies this then the computer barks(!)
- a case structure can also operate on a scalar `CHARACTER` < *case-expr* > which could be matched to a list of possible character values or, indeed, a list of possible ranges.

54: Visualisation of SELECT CASE

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,



Guide for slide: 54

55: SELECT CASE Construct II

This is useful if one of several paths must be chosen based on the value of a single expression.
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;
- the < *case-selector* > is a parenthesised single value or range, for example, `(.TRUE.)`, `(1)` or `(99:101)`;
- there can only be one `CASE DEFAULT` branch;
- control cannot jump into a `CASE` construct.

Guide for slide: 55

- 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 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.
- 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.
- the < *case-expr* > is evaluated and compared with the < *case-selector* >s in turn to see which branch to take. The < *case-expr* > must be either scalar INTEGER, CHARACTER or LOGICAL valued.
- the < *case-selector* >s can specify a single value, a list or a bounded or unbounded range or a list of ranges. The first one which is encountered which contains the (scalar) < *case-expr* > is the branch which is chosen.
- there can be any number of < *case-selector* >s.
- the < *case-selector* > is a single value or range specifier surrounded by parenthesis.
- 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.
- if no branches are chosen then the CASE DEFAULT is executed (if present).
- there can only be one CASE DEFAULT block.
- 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.
- any number of CASE constructs can be nested.

56: Mixed Type Numeric Expressions

In the CPU calculations must be performed between objects of the *same* type, so if an expression mixes type some objects must change type. Default types have an implied ordering:

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

The result of an expression is always of the highest type, 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`.

The actual operator is unimportant.

Guide for slide: 56

- Must take care with mixed mode arithmetic, (expressions containing objects of different types).
- Numeric and non-numeric types cannot be mixed using intrinsic operators.
- Cannot mix `LOGICAL` and `CHARACTER`.
- In general, for a numeric subexpression (of one operator and two operands) if the types are different then the lower type is promoted or coerced to a higher type. The types have the following ordering:

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

  Thus for a subexpression mixing an `INTEGER` and a `REAL` the `INTEGER` is first converted to a `REAL` and then the expression is evaluated.

57: Mixed Type Assignment

Problems often occur with mixed-type arithmetic; the rules for type conversion are given below.

- `INTEGER` = `REAL` (or `DOUBLE PRECISION`)

  The RHS is evaluated, truncated (all the decimal places lopped off) then assigned to the LHS.

- `REAL` (or `DOUBLE PRECISION`) = `INTEGER`

  The RHS is promoted to be `REAL` and stored (approximately) in the LHS.

For example,

```
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
```

Note: as `a` and `b` stored approximately, the value of `k` is uncertain.

Guide for slide: 57

When the whole RHS expression of a mixed type assignment statement has been evaluated it has a 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, RHS = `INTEGER`, LHS = `REAL` the LHS needs promoting to be a `REAL` value; RHS = `REAL` and LHS = `INTEGER`, the `REAL` needs relegating to be `INTEGER`.

- LHS `INTEGER`; RHS `REAL`— the right hand side is evaluated and then the value is truncated (all the decimal places lopped off) then assigned to the LHS.
- LHS `REAL`; RHS `INTEGER`— the right hand side expression is simply stored (approximately) in the LHS. The `INTEGER` is promoted as if it were an actual-argument of the `REAL` intrinsic function.

Note: 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
```

- `j` would truncated to `0` because numbers are aways truncated *towards* zero.
- the result of `a - b` would be close to `1.0` (it could be `1.0000001` or it could be `0.999999999`) this means that after truncation `k` could contain either `0` or `1`,

58: Integer Division

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             ! LHS is 1
b = -1999/1000            ! LHS is -1
c = (1999+1)/1000         ! LHS is 2
d = 1999.0/1000           ! LHS is 1.999
e = 1999/1000.0           ! LHS is 1.999
```

- `a` is (about) `1.000`
- `b` is (about) `-1.000`
- `c` is (about) `2.000`
- `d` is (about) `1.999`
- `e` is (about) `1.999`

Great care must be taken when using mixed type arithmetic.

Guide for slide: 58

- If one integer divides another in a subexpression then the type of that subexpression is `INTEGER` even if the context is a simple assignment statement with only one operator and two operands (like in the slide).
- in other words the following happens:

  1. the RHS is evaluated
  2. the integral result of the RHS is converted to a `REAL` value before assignment to the LHS.

- The integer expression `1999/1000` is evaluated and then truncated towards zero to produce an integral value. Its says in the Fortran 90 standard, 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."
- likewise `-1999/1000`.
- `c` is 2.0 because 2000/1000 is calculated.
- `d` and `e` are both `1.999` because both RHS's are evaluated to be real numbers, in `1999.0/1000` `1999/1000.0` the integers are promoted to real numbers before the division.

59: Intrinsic Procedures

Fortran 90 has 113 in-built or *intrinsic* procedures to perform common tasks efficiently, they belong to a number of classes:

- elemental such as:
  - mathematical, for example, `SIN` or `LOG`.
  - numeric, for example, `SUM` or `CEILING`;
  - character, for example, `INDEX` and `TRIM`;
  - bit, for example, `IAND` and `IOR`;
- inquiry, for example, `ALLOCATED` and `SIZE`;
- transformational, for example, `REAL` and `TRANSPOSE`;
- miscellaneous (non-elemental `SUBROUTINE`s), for example, `SYSTEM_CLOCK` and `DATE_AND_TIME`.

Note all intrinsics which take `REAL` valued arguments also accept `DOUBLE PRECISION` arguments.

Guide for slide: 59

These 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.

- elemental - this means they 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.
  - numeric, for example, `SUM` or `SIN`;
  - character, for example, `INDEX` and `TRIM`;
    For example, string manipulation, relational comparisons.
  - bit, for example, `IAND` and `IOR`;
    A collection of intrinsics for manipulating and setting up bit variables,
- inquiry, for example, `ALLOCATED` and `SIZE`;
  Can find out 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, `RESHAPE` and `TRANSPOSE`. (Mainly arrays)
  - `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 changing, `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.)
  - `PRODUCT` and `DOT_PRODUCT` (arrays),
- miscellaneous (non-elemental) including timing routines, for example, `SYSTEM_CLOCK`, `DATE_AND_TIME`, random number seeding

60: Type Conversion Functions

It is easy to transform the type of an entity,

- ☐ `REAL(i)` converts `i` to a real approximation,
- ☐ `INT(x)` truncates `x` to the integer equivalent,
- ☐ `DBLE(a)` converts `a` to `DOUBLE PRECISION`,
- ☐ `IACHAR(c)` returns the position of `CHARACTER c` in the ASCII collating sequence,
- ☐ `ACHAR(i)` returns the i^{th} character in the ASCII collating sequence.

All above are intrinsic functions. For example,

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

are equal to

```
1.000000 1 0
67 C
```

Guide for slide: 60

- argument to REAL can be INTEGER, DOUBLE PRECISION or COMPLEX.
- argument to INT can be REAL, DOUBLE PRECISION or COMPLEX.
- argument to DBLE can be INTEGER, REAL or COMPLEX.
- there are also other functions to get integers from non-integer values:
  - CEILING — smallest integer greater or equal to the argument,
  - FLOOR — largest integer less or equal to the argument,
  - NINT — nearest integer
- CMPLX converts to a complex value.
- argument IACHAR must be a single CHARACTER.
- argument ACHAR must be a single INTEGER.

61: 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

Guide for slide: 61

The slide details the function names and, when necessary, details about the required arguments.

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.

- `ASIN`, `ACOS` — arcsin and arccos.

  The argument to each must be real and $\leq |1|$, for example, `ASIN(0.84147)` has value `1.0` (radians).

- `ATAN` — arctan.

  The argument must be real valued, for example, `ATAN(1.0)` is $\frac{\pi}{4}$, `ATAN(1.5574)` has value `1.0`.

- `ATAN2` — arctan; the principle value of the nonzero complex number `(X,Y)`, for example, `ATAN2(1.5574,1.0)` 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 trig functions.

  The actual arguments must be `REAL` valued, for example, `COSH(1.0)` is `1.5431`.

- `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). Note that the argument to `SQRT` cannot be `INTEGER`, for example, `EXP(1.0)` is `2.7182`.

Note all angles are expressed in radians.

62: 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))`

Guide for slide: 62

The slide gives details regarding the names and arguments of these intrinsics.

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 actual argument can be `INTEGER`, `REAL` or `COMPLEX`, the result is 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.

  Argument is `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`. The result 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`. The result is $a \bmod b$, for example, `MODULO(9,5)` is `4`, `MODULO(-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 te 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.

63: 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

Guide for slide: 63

- `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 a valid character and returns its position of a character in the compilers collating sequence. (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 functions.

  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`.

64: PRINT Statement

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

```
PROGRAM Owt
 IMPLICIT NONE
  CHARACTER(LEN=*), PARAMETER :: &
     long_name = "Llanfair...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 Owt
```

produces on the screen,

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

Guide for slide: 64

- The simplest form output is the PRINT statement.
- The output goes to the standard output channel.
- the * takes the place of a format statement number - this point should be glossed over at the present time.
- the PRINT statement takes a comma separated list of things to print!
- each PRINT statement begins a new line.
- the 'arguments' to the print statements can be (virtually) anything.
  - literal (numeric, character, boolean).
  - variables or expressions (numeric, character, boolean).
  - scalar or array valued.

  Cannot be a derived type with a POINTER or private component (see later).
- PRINT*, performs no (controllable) formatting - Fortran 90 supports a great wealth of output (and input) formatting which is not described here!
- PRINT*, long_name simply prints out variable contents!

```
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.
- note the double " in the string the first one *escapes* the second and allows it to be printed.
- notice how the outout 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.
- LOGICAL values are always represented a T or F

65: PRINT Statement

Note,

- each PRINT statement begins a new line;
- the PRINT statement can transfer any object of intrinsic type to the standard output;
- strings may be delimited by the double or single quote symbols, " and ';
- two occurrences of the string delimiter inside a string produce one occurrence on output;

Guide for slide: 65

- non-advancing I/O is available - have to specify it (see later).
- the `PRINT` statement can transfer any object of intrinsic type to the standard output - can also do the non-default kinds (which of course are intrinsic types).
- a string cannot contain a single unescaped delimiter of the same type that is used to delimit it. In order to output a " in the middle of a string which is delimited by " characters a double " must be used. Like wise for the ' delimiter. If the alternative delimiter is used then a single instance is OK. For example

```
"Man from Delmonte, he say 'Yes'"
'Man from Delmonte, he say "Yes"'
```

66: READ Statement

READ accepts 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
```

accepts

```
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;

Guide for slide: 66

- the * format specifier the `READ` statement is comparable to the functionality of `PRINT` i.e., unformatted data can be read.
- actually this is not strictly true, formatted data can be read but the format cannot be specified.
- can have more than one object per `READ`.
- each separate input entity must be blank separated.

this is for file I/O and is not strictly relevant at this stage — each `READ` statement reads from a newline - so if you have 3 things on one line and the `READ` statement only reads one (the first one) the last two are simply ignored.

- any intrinsic type can be read including non-default kinds. (No default kind values will look very much like normal values except they may not fit into the representation of default kinds.

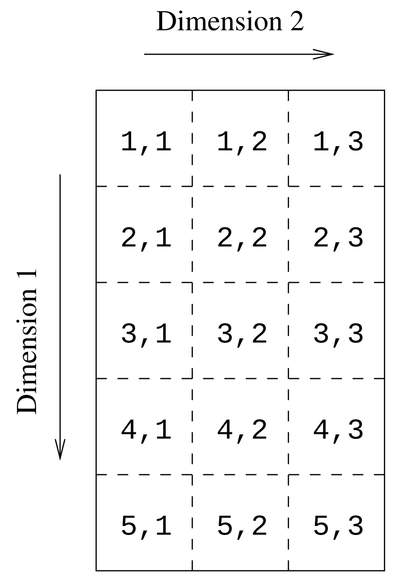
Lecture 4: Arrays

68: 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:

1	2	3	....	13	14	15

And a 5 × 3 array as:



Every array has a type and each element holds a value of that type.

Guide for slide: 68

69: Array Terminology

Examples of declarations:

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

The above are *explicit-shape* arrays.
Terminology:

- **rank** — number of dimensions.

  Rank of X is 1; rank of Y and Z is 2.

- **bounds** — upper and lower limits of indices.

  Bounds of X are 1 and 15; Bound of Y and Z are 1 and 5 and 1 and 3.

- **extent** — number of elements in dimension;

  Extent of X is 15; extents of Y and Z are 5 and 3.

- **size** — total number of elements.

  Size of X, Y and Z is 15.

- **shape** — rank and extents;

  Shape of X is 15; shape of Y and Z is 5,3.

- **conformable** — same shape.

  Y and Z are conformable.

Guide for slide: 69

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

- *explicit-shape arrays* can have symbolic bounds so long as they are initialisation expressions — evaluatable at compile time.
- *rank* — up to and including 7 dimensions.
- *extent* — can be zero
- size — can specify size for a particular dimension.
- shape — shape is expressed as an array of extents
- *conformable* — for operations between two arrays the shapes (of the sections) must (generally) conform (just like in mathematics).
- there is no *storage association* for Fortran 90 arrays.

70: Declarations

Literals and constants can be used in array declarations,

```
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
```

- default lower bound is 1,
- bounds can begin and end anywhere,
- arrays can be zero-sized (if `lda = 0`),

Guide for slide: 70

Declared using the `DIMENSION` attribute

- `REAL, DIMENSION(-10:-1) :: X` demonstrates that bounds can begin below zero,
- if the lower bound is not specified then the default lower bound is 1,
- initialisation expressions are allowed in bound specifications expressions,
- zero sized arrays are useful because 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 generally ignored. If `lda` were set to be zero,

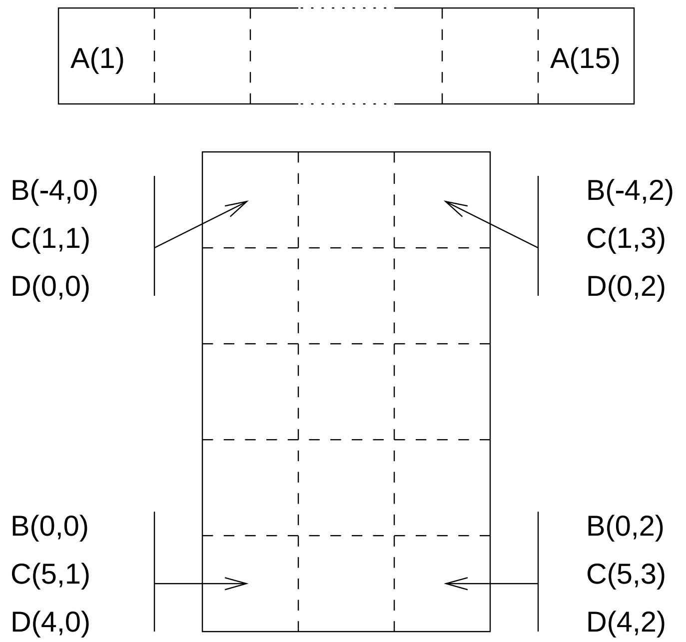
  ```
  INTEGER, PARAMETER :: lda = 0
  ```

  then the array `Y` would be zero sized.
- there are different types of arrays - these will be covered later.

71: Visualisation of Arrays

```
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.



Guide for slide: 71

This slide allows visualisation of the arrays which were declared on the previous slide.

- The top matrix (`A`) is a 1D 15 element vector. The default lower bound is 1.
- The lower matrix is a 5×3 2D array, it is the same shape as B, C and `D` and the corresponding elements of each matrix are shown.

72: Array Conformance

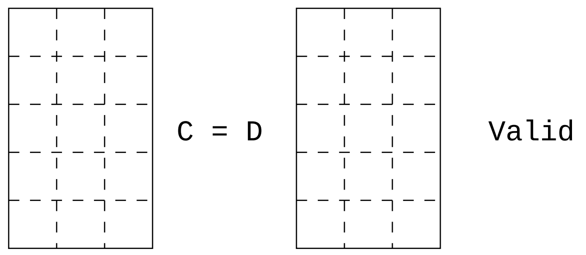
Arrays or sub-arrays must conform with all other objects in an expression:

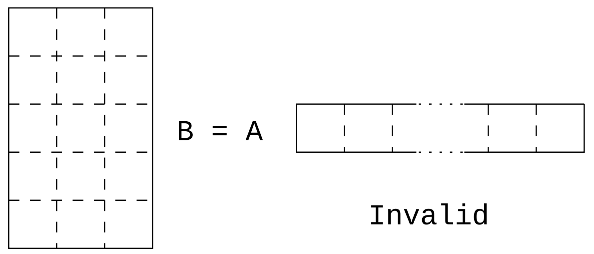
- a scalar conforms to an array of any shape with the same value for every element:

```
C = 1.0   ! is valid
```

- two array references must conform in their shape.

  Using the declarations from before:





`A` and `B` have the same size but have different shapes so cannot be directly equated.

Guide for slide: 72

- `C = 1.0`: the value 1.0 is assigned to all elements of `C`.
- C and D are the same shape so conform.
- `A = B`: even though A and B have the same number of elements (15) the two arrays do not conform - conformable means 'same shape'.
- two arrays of different types can be used in the same expression and will have the relevant type coercion performed just like scalars.
- there is a difference between a scalar and an array with one element.
- see later for intrinsics that are able to change the shape of arrays.

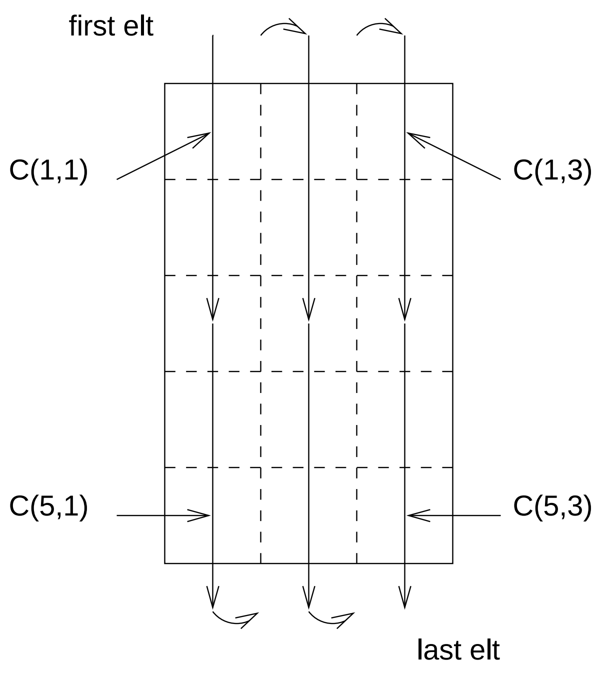
73: Array Element Ordering

Organisation in memory:

- Fortran 90 does not specify anything about how arrays should be located in memory. **It has no storage association.**

- Fortran 90 does define an array element ordering for certain situations which is of column major form,

The array is conceptually ordered as:




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

Guide for slide: 73

- 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, on distributed memory computers an array may be stored over 100 processors with each processor owning only a small section of the whole array.
- Fortran 77 did have storage association. It specified that array elements were stored in column major form, (i.e. with the indices of the lowest dimension varying most rapidly).
- 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.

74: Array Syntax

Can reference:

- whole arrays
  - `A = 0.0`
    sets whole array `A` to zero.
  - `B = C + D`
    adds `C` and `D` then assigns result to `B`.
- elements
  - `A(1) = 0.0`
    sets one element to zero,
  - `B(0,0) = A(3) + C(5,1)`
    sets an element of `B` to the sum of two other elements.
- array sections
  - `A(2:4) = 0.0`
    sets `A(2)`, `A(3)` and `A(4)` to zero,
  - `B(-1:0,1:2) = C(1:2,2:3) + 1.0`
    adds one to the subsection of `C` and assigns to the subsection of `B`.

Guide for slide: 74

- 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.
- A whole array can be referenced by using only its name with no subscripts nor parentheses.
- A whole array can also be referenced by subscripting it with a single colon this is shorthand for `lower_bound:upper_bound`.
- B and C must be same shape.
- `A(1)` is a Scalar variable.
- Scalars (literals and variables) conform to arrays.
- Parts of arrays can be referenced, these are 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) thus the section specification is denoted by a *subscript-triplet* which is a linear function.
- 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.
- All the elemental procedures mentioned before (`SIN`, `LOG`, `IBTEST`, etc etc) work with array arguments.

75: Whole Array Expressions

Arrays can be treated like a single variable in that:

- can use intrinsic operators between conformable arrays (or sections),

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

  this is equivalent to concurrent execution of:

```
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 ||
```

- elemental intrinsic functions can be used,

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

  the function is applied element by element.

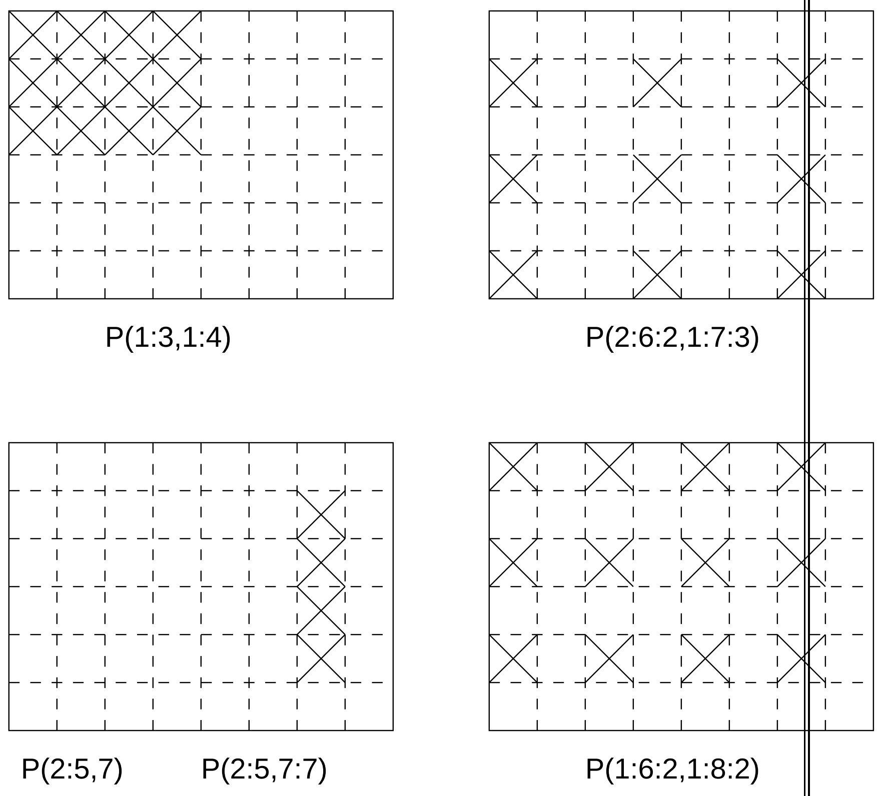
Guide for slide: 75

- 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.
- The assignment `B = C * D - B**2` is an elemental assignment and is performed between each corresponding element of each array.
- Clearly `C*D` is not 'traditional' matrix multiplication. For this effect `MATMUL(C,D)` should be used instead.
- Recall that the RHS operand of the `**` operator must be scalar.
- Operators have the same precedence as for scalars and behave in exactly the same way with all their operations being performed in parallel.
- Many of the intrinsics are elemental including all numeric, mathematical, bit, character and logical intrinsics.

76: Array Sections — Visualisation

Given,

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



Consider the following assignments,

- `P(1:3,1:4) = P(1:6:2,1:8:2)` and `P(1:3,1:4) = 1.0` are valid.
- `P(2:8:2,1:7:3) = P(1:3,1:4)` and `P(2:6:2,1:7:3) = P(2:5,7)` are not.
- `P(2:5,7)` is a 1D section (scalar in dimension 2) whereas `P(2:5,7:7)` is a 2D section.

Guide for slide: 76

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)` is a 3×3 section.
- `P(2:5,7)` is a 1D array with 4 elements.
- `P(2:5,7:7)` is a 4×1 section.
- `P(1:6:2,1:8:2)` is a 3×4 section.

Conformance:

- `P(1:3,1:4) = P(1:6:2,1:8:2)` is valid — both LHS and RHS are 3×4 sections.
- `P(1:3,1:4) = 1.0` is valid as 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 — trying to equate a 3×3 section with a 3×4 section, the arrays do not conform.
- `P(2:6:2,1:7:3) = P(2:5,7)` is not trying to equate a 3×3 section with a 4 element 1D array.

Also note

- `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.

77: Array Sections

subscript-triplets specify sub-arrays. The general form 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 sections.

Guide for slide: 77

- the stride specification follows the same principle as for a `DO` loop — if the upper bound is not a combination of the lower bound plus multiples of the stride then the actual upper bound is different from that stated.
- negative strides are permitted.
- zero sections are permitted
- so long as the section specifiers are scalar integers then any expression (even involving arrays and reduction functions) can be used.
- if < *bound1* > or < *bound2* > are absent then the lower or upper bound of the dimension is implied. Thus `A(:)` refers to the whole array.
- if the < *stride* > is absent then it is taken to be 1.
- `A(m:n:k)` - from `m` to `n` in strides of 1.
- `A(m:n:k)` - from `m` to `n` in strides of `k`.
- `A(8:3:-1)` - from 8 to 3 in steps of `-1`
- `A(8:3)` - from 8 to 3 in steps of 1, i.e., this is a zero sized section
- `A(m:)` - from `m` to the upper bound in steps of 1
- `A(:n)` - from the default LWB to `n` in steps of 1
- `A(::2)` from the lower bound to the upper bound in strides of 2.
- `A(m:m)` one element array. Different from `A(m)`.

78: Array I/O

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 the order:

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

```
READ*, A
```

would assign to the elements in the above order.
This order could be changed by using intrinsic functions such as `RESHAPE`, `TRANSPOSE` or `CSHIFT`.

Guide for slide: 78

This is the same order as FORTRAN 77 defined for arrays.

- 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.
- when reading in an array any number of carriage returns may be inserted amongst the data — they are simply ignored.

79: Array I/O Example

Consider the matrix A:

1	4	7
2	5	8
3	6	9

The following PRINT statements

```
  ...
   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 PROGRAM 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
```

Guide for slide: 79

- Arrays are simply output in ”array element order”
- Note that the default format is used and it may not look like this!
- Any functions are evaluated before output.

80: Array Inquiry Intrinsics

These are often useful in procedures, consider the declaration:

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

- `LBOUND(SOURCE[,DIM])` — lower bounds of an array (or bound in an optionally specified dimension).
  - `LBOUND(A)` is `(/-10,1,14/)` (array);
  - `LBOUND(A,1)` is `-10` (scalar).
- `UBOUND(SOURCE[,DIM])` — upper bounds of an array (or bound in an optionally specified dimension).
- `SHAPE(SOURCE)` — shape of an array,
  - `SHAPE(A)` is `(/21,23,15/)` (array);
  - `SHAPE((/4/))` is `(/1/)` (array).
- `SIZE(SOURCE[,DIM])` — total number of array elements (in an optionally specified dimension),
  - `SIZE(A,1)` is `21`;
  - `SIZE(A)` is `7245`.
- `ALLOCATED(SOURCE)` — array allocation status;

Guide for slide: 80

These intrinsics allow the user to quiz arrays about their attributes and status and are most often most used on Dummy arguments in procedures.

- `LBOUND`, `UBOUND`, give the bounds of the array a viewed in the current scoping unit. The result is a one dimensional array containing the bounds of an array or, if a dimension is specified, a scalar containing the bound in that dimension.

  When an array is an assumed shape dummy argument, its bounds always begin from 1 - this is not necessarily the case for the actual argument. The result is returned
- if the `DIM=` argument is present then the `LBOUND`, `UBOUND`, result is always scalar,
- for `SHAPE`, the result is always array valued unless the argument is scalar,
- for the `SIZE` intrinsic, `SOURCE` must always be an array and the result is *always* scalar,
- `ALLOCATED` returns `.TRUE.` or `.FALSE.` depending on array status (allocated or not).
- all these intrinsics are useful in procedures - see later!

81: Array Constructors

Used to give arrays or sections of arrays specific values. For example,

```
IMPLICIT NONE
INTEGER                          :: i
INTEGER, DIMENSION(10)           :: ints
CHARACTER(len=5), DIMENSION(3)   :: colours
REAL, DIMENSION(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 /)
 ...
```

- constructors and array sections must conform.
- must be 1D.
- for higher rank arrays use `RESHAPE` intrinsic.
- `(i, i=1,8)` is an *implied* `DO` and is 1,2,..,8, it is possible to specify a stride.

Guide for slide: 81

- the `(/` and `/)` delimit the values.
- the values are placed into the array in array element order.
- anything within the delimiters must be a scalar expression.
- `heights(1:4)` is filled in order with the values `5.10, 5.6, 4.0, 3.6`.
- for `CHARACTER` constructors the string within the constructor must be the correct length for the variable, this is why `'RED  '` is padded with blanks.
- `ints` will contain the values `(/ 100, 1, 2, 3, 4, 5, 6, 7, 8, 100 /)`.
- an implied `DO` may be used in the constructor 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.
- type conversions are performed as in regular assignment.
- only 1D constructors are permitted - see later for `RESHAPE`.
- the constructor must be of the correct length for the array section - they must conform.

82: The RESHAPE Intrinsic Function

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

```
RESHAPE(SOURCE, SHAPE)
```

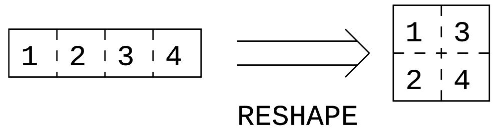
For example,

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

A is filled in array element order and looks like:

```
1  3
2  4
```

Visualisation,



Guide for slide: 82

- ☐ `RESHAPE` 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,
- ☐ in the example a 2×2 array is being constructed,
- ☐ clearly the result of `RESHAPE` must conform to the array object on the LHS of the `=`,

83: Array Constructors in Initialisation Statements

Arrays can be initialised

```
INTEGER, DIMENSION(4) :: solution = (/2,3,4,5/)
CHARACTER(LEN=*), DIMENSION(3) :: &
         lights = (/'RED  ','BLUE ','GREEN'/)
```

In the second statement all strings must be same length.
Named array constants may also be created:

```
INTEGER, DIMENSION(3), PARAMETER :: &
         Unit_vec = (/1,1,1/)
REAL, DIMENSION(3,3), PARAMETER :: &
         unit_matrix = &
            RESHAPE((/1,0,0,0,1,0,0,0,1/),(/3,3/))
```

Guide for slide: 83

- PARAMETERs declared as expected.
- RESHAPE can be used in initialisation statements as long as all components of it can be evaluated at compile time - (just like in initialisation expressions).
- note how in a PARAMETER initialisation, the length of the string lights can be assumed from the length of the constructor values. The strings in the constructor *must* all be the same length.
- because unit_matrix is a PARAMETER it may also be used in initialisation expressions.

84: Allocatable Arrays

Fortran 90 allows arrays to be created on-the-fly; these are known as *deferred-shape* arrays:

- Declaration:

```
INTEGER, DIMENSION(:), ALLOCATABLE :: ages    ! 1D
REAL, DIMENSION(:,:), ALLOCATABLE :: speed    ! 2D
```

  Note ALLOCATABLE attribute and fixed rank.

- Allocation:

```
READ*, isize
ALLOCATE(ages(isize), STAT=ierr)
IF (ierr /= 0) PRINT*, "ages : Allocation failed"

ALLOCATE(speed(0:isize-1,10),STAT=ierr)
IF (ierr /= 0) PRINT*, "speed : Allocation failed"
```

- the optional STAT= field reports on the success of the storage request. If the INTEGER variable ierr is zero the request was successful otherwise it failed.

Guide for slide: 84

- Fortran 90 has dynamic storage this means memory can be grabbed (from a part of memory called the *heap store*), used and then put back at any point in the program.
- This allows the creation of "temporary" arrays which can be created used and discarded
- there is a certain overhead in managing dynamic or `ALLOCATABLE` arrays - explicit-shape are cheaper and should be used if the size is known and the arrays are persistent (i.e., 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. There will almost certainly be an option of the compiler to specify / increase the size of heap storage.
- *deferred-shape* arrays have deferred bound specification - hence their name.
- *deferred-shape* arrays *must* be given the `ALLOCATABLE` attribute.
- the `ALLOCATE` statement grabs the heap storage and sets up an array. with specified bounds.
- they follow the same general rules as "normal" arrays, i.e., bounds can begin anywhere, zero sized arrays are allowed, etc.
- cannot pass *unallocated* allocatables as actual arguments.
- the `STAT=`reports on success / failure of storage request. If it is supplied then the keyword must be used to distinguish it from an array that needs allocating.
- `ierr` is `INTEGER` and should always be used.
- a list of objects can be allocated together but ideally only one array per `ALLOCATE` statement should be given. If there is more than one and the allocation fails it is not possible to easily tell which allocation was responsible.

85: Deallocating Arrays

Heap storage can be reclaimed using the DEALLOCATE statement:

```
IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)
```

- it is an error to deallocate an array without the ALLOCATE attribute or one that has not been previously allocated space,
- there is an intrinsic function, ALLOCATED, which returns a scalar LOGICAL values reporting on the status of an array,
- the STAT= field is optional but its use is recommended,
- if a procedure containing an allocatable array which does not have the SAVE attribute is exited without the array being DEALLOCATEd then this storage becomes inaccessible.

Guide for slide: 85

- as a matter of course the intrinsic inquiry function, `ALLOCATED`, should be used to check on the status of the array before attempting to `DEALLOCATE`,
- the `STAT=` field is optional but its use is recommended, it reports on the success / failure of the `DEALLOCATE` request in an analogous way to the `ALLOCATE` statement.
- see later for procedures and the `SAVE` attribute - always deallocate arrays as soon as they are finished with.

86: Masked Array Assignment — Where Statement

This is achieved using `WHERE`:

```
WHERE (I .NE. 0) A = B/I
```

the LHS of the assignment must be array valued and the mask, (the logical expression,) and the RHS of the assignment must all conform; 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.

Guide for slide: 86

- the `WHERE` statement is used when an array assignment is to be performed on a subset of the LHS array elements.
- cannot have a scalar on the LHS in a `WHERE` statement / construct.
- in the first example, if I is a 2D array, for all values of `j` and `k` where the mask, `(I(j,k) .NE. 0)`, is `.TRUE.` the assignment `A(j,k) = B(j,k)/I(j,k)` is performed for the cases where the mask is `.FALSE.` no action is taken.
- conformable array sections may be used in place of the whole arrays.
- every `WHERE` statement must contain an array assignment statement.

87: Where Construct

- there is a block form of masked assignment:

```
WHERE(A > 0.0)
 B = LOG(A)
 C = SQRT(A)
ELSEWHERE
 B = 0.0 ! C is NOT changed
ENDWHERE
```

- the mask must conform to the RHS of each assignment; `A, B` and `C` must conform;
- `WHERE` ... `END WHERE` is *not* a control construct and cannot currently be nested;
- 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.

Guide for slide: 87

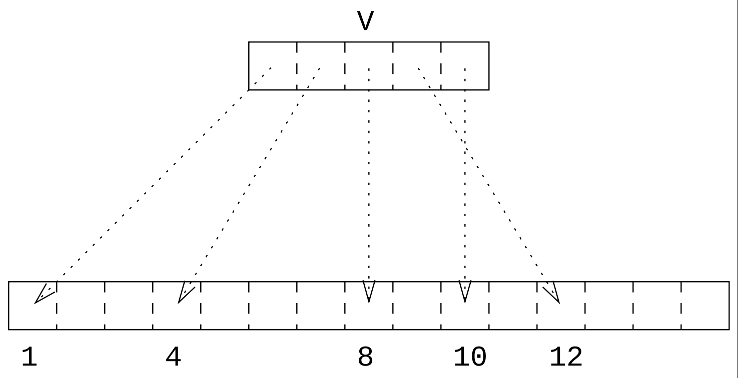
- all statements in a `WHERE` construct must be array assignments.
- wherever `(A(j,k) > 0.0)` is `.TRUE.`, the assignments `B(j,k) = LOG(A(j,k))` and `C(j,k) = SQRT(A(j,k)` are made, otherwise the assignments in the `ELSEWHERE` block are made instead.
- note that the two branches are executed sequentially.
- the separate assignment statements are executed sequentially but the individual elemental assignments within each statement are (conceptually) executed in parallel.
- conformable array sections may be used in place of the whole arrays.
- all statements within a `WHERE` must be array assignments.

88: Vector-valued Subscripts

A 1D array can be used to subscript an array in a dimension. Consider:

```
INTEGER, DIMENSION(5) :: V = (/1,4,8,12,10/)
INTEGER, DIMENSION(3) :: W = (/1,2,2/)
```

- A(V) is A(1), A(4), A(8), A(12), and A(10).



- the following are valid assignments:

```
A(V) = 3.5
C(1:3,1) = A(W)
```

- it would be invalid to assign values to A(W) as A(2) is referred to twice.
- only 1D vector subscripts are allowed, for example,

```
A(1) = SUM(C(V,W))
```

Guide for slide: 88

- Indexing indirection can be introduced by using vector-valued subscripts, this is where a 1D vector is substituted for the subscript-triplet specifier.
- This means that an array section can be specified where the order of the elements do not follow a linear pattern.
- This from of subscripting is **very inefficient** and should not be used unless absolutely necessary.
- Operations conceptually done in parallel.
- Vector-valued Subscripts can be used on either side of the assignment operator.
- Can refer to the same element twice in the same statement on the RHS.
- In order to preserve the integrity of parallel array operations we cannot assign to the same element twice in the same statement, i.e., subscripts on the LHS of the = must be unique. Fortran 90 conceptually assigns to all array elements referenced in a single statement at the same time, clearly this is not possible when the same elements are assigned to twice.
- 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,

```
A(1) = SUM(C(V,W))
```

is valid

89: Random Number Intrinsic

- RANDOM_NUMBER(HARVEST) will return a scalar (or array of) pseudorandom number(s) in the range $0 \leq x < 1$.

  For example,

```
REAL                    :: HARVEST
REAL, DIMENSION(10,10) :: HARVEYS
CALL RANDOM_NUMBER(HARVEST)
CALL RANDOM_NUMBER(HARVEYS)
```

- RANDOM_SEED([SIZE=< *int* >]) finds the size of the seed.

- RANDOM_SEED([PUT=< *array* >]) seeds the random number generator.

```
CALL RANDOM_SEED(SIZE=isze)
CALL RANDOM_SEED(PUT=IArr(1:isze))
```

Guide for slide: 89

- useful when developing and testing code for setting up random valued objects.
- `RANDOM_SEED` can be used to modify the behaviour of `RANDOM_NUMBER`. By supplying different parameters to this intrinsic subroutine the random number generated my be supplied with a specific array valued seed, report on which seed is in use and report on how big an array is needed to hold a seed. Not necessary to know any more at this stage.
- Using the same seed on separate executions will generate the same sequence of random numbers.

90: Vector and Matrix Multiply Intrinsics

There are two types of intrinsic matrix multiplication:

- `DOT_PRODUCT(VEC1, VEC2)` — inner (dot) product of two rank 1 arrays.

  For example,

  ```
  DP = DOT_PRODUCT(A,B)
  ```

  is equivalent to:

  ```
  DP = A(1)*B(1) + A(2)*B(2) + ...
  ```

  For `LOGICAL` arrays, the corresponding operation is a logical `.AND.`.

  ```
  DP = LA(1) .AND. LB(1) .OR. &
       LA(2) .AND. LB(2) .OR. ...
  ```

- `MATMUL(MAT1, MAT2)` — 'traditional' matrix-matrix multiplication:

  - 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);

  For `LOGICAL` arrays, the corresponding operation is a logical `.AND.`.

Guide for slide: 90

DOT_PRODUCT:

- is equivalent to

```
DP = SUM(A*B)
```

  or for COMPLEX arguments

```
DP = SUM(CONJG(A)*B)
```

- is the mathematical dot product of two vectors,
- VEC1, VEC2 must conform in size (and must be 1D),
- don't get this intrinsic confused with DPROD the DOUBLE PRECISION product function or PRODUCT the intra-matrix product (see later).
- For LOGICAL arrays, the operation is a logical .AND.,

```
DP = LA(1).AND.LB(1) .OR. LA(2).AND.LB(2) .OR. ...
```

  and is equivalent to

```
DP = ANY(LA .AND. LB)
```

MATMUL:

- is the mathematical matrix multiply intrinsic, it is not MAT1*MAT2, it is equivalent to setting element (i,j) of the result to:

```
SUM(MAT1(i,:)*MAT2(:,j))
```

- arrays must match in the specified dimensions (as you would expect); they do not have to conform,
- arrays can only be of rank 1 or rank 2. There must be one of each. i.e., if MAT1 is 1D then MAT2 must be 2D, and vice-versa.
- 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);
- for LOGICAL arrays, the operation is a logical .AND.: and is equivalent to (i,j) of the result to:

```
ANY(MAT1(i,:).AND.MAT2(:,j))
```

91: Maximum and Minimum Intrinsics

There are two intrinsics in this class:

- `MAX(SOURCE1,SOURCE2[,SOURCE3[,...]])`— maximum values over all source objects
- `MIN(SOURCE1,SOURCE2[,SOURCE3[,...]])`— minimum values over all source objects

Scan from left to right, choose **first** occurrence if there are duplicates

MAX(X)

7	9	-2	4	8	10	2	7	10	2	1

- `MAX(1,2,3)` is 3
- `MIN((/1,2/),(/-3,4/))` is `(/-3,2/)`
- `MAX((/1,2/),(/-3,4/))` is `(/1,4/)`

Guide for slide: 91

- the result of these intrinsics is either the maximum (or minimum) value from the argument list or a conformable array filled with the maximum (or minimum) values from the correspond locations of the source arrays, there can be any number of source objects (arrays or scalars).

92: Array Location Intrinsics

There are two intrinsics in this class:

- MINLOC(SOURCE[,MASK])— Location of a minimum value in an array under an optional mask.
- MAXLOC(SOURCE[,MASK])— Location of a maximum value in an array under an optional mask.

A 1D example,

MAXLOC(X) = (/6/)

7	9	-2	4	8	10	2	7	10	2	1

A 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

- MINLOC(Array) is (/3,4/)
- MAXLOC(Array,Array.LE.7) is (/1,4/)
- MAXLOC(MAXLOC(Array,Array.LE.7)) is (/2/) (array valued).

Guide for slide: 92

- the result of these intrinsics is a 1D array of the coordinates of the maximum or minimum value,
- if the `MASK` is present the only elements considered are those corresponding to where the mask is `.TRUE.`,
- the result is **always** array valued even if it is only one number,
- for the first example, `MAXLOC(ARRAY,ARRAY.LE.7)`, only the following elements are considered

```
ARRAY =  0  -1  1  6  -4
         1  -2  5  4  -3
         3   .  3 -7   0
```

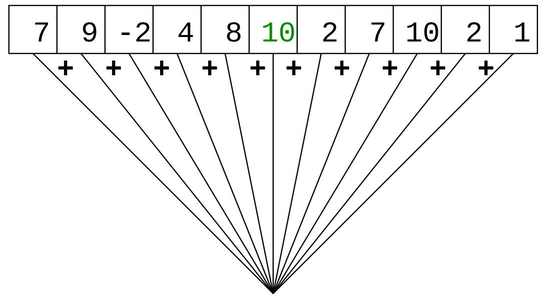
  The largest is element `A(1,4)`
- second example `-7` is smallest value which is element `A(3,4)`,
- the third example is finding the location which holds the largest element in the array `(/1,4/)`.

93: Array Reduction Intrinsics

- `PRODUCT(SOURCE[,DIM][,MASK])`— product of array elements (in an optionally specified dimension under an optional mask);
- `SUM(SOURCE[,DIM][,MASK])`— sum of array elements (in an optionally specified dimension under an optional mask).

The following 1D example demonstrates how the 11 values are reduced to just one by the `SUM` reduction:

SUM(W) = 58



= 58

Consider this 2D example, if

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

- `PRODUCT(A)` is 720
- `PRODUCT(A,DIM=1)` is `(/2, 12, 30/)`
- `PRODUCT(A,DIM=2)` is `(/15, 48/)`

Guide for slide: 93

94: Array Reduction Intrinsics (Cont'd)

These functions operate on arrays and produce a result with less dimensions that the source object:

- `ALL(MASK[,DIM])` — `.TRUE.` if *all* values are `.TRUE.`, (in an optionally specified dimension);
- `ANY(MASK[,DIM])` — `.TRUE.` if *any* values are `.TRUE.`, (in an optionally specified dimension);
- `COUNT(MASK[,DIM])` — number of `.TRUE.` elements in an array, (in an optionally specified dimension);
- `MAXVAL(SOURCE[,DIM][,MASK])` — maximum Value in an array (in an optionally specified dimension under an optional mask);
- `MINVAL(SOURCE[,DIM][,MASK])` — minimum value in an array (in an optionally specified dimension under an optional mask);

If `DIM` is absent or the source array is of rank 1 then the result is scalar, otherwise the result is of rank $n-1$.

Guide for slide: 94

Reduction functions are aptly named because an array is operated upon and a result obtained which has a smaller rank than the original array. If DIM is absent or the source array is of rank 1 then the result is scalar, otherwise the result is of rank $n-1$.

- ALL(MASK[,DIM])
  - .TRUE. if *all* values of the mask are .TRUE. along dimension DIM, the result is an array of rank $n-1$. 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.
  - .TRUE. if array is zero sized,
  - if DIM is absent then the whole array is considered.
- ANY(MASK[,DIM])
  - .TRUE. if *any* values are .TRUE. along dimension DIM, the result is an array of rank $n-1$
  - .FALSE. if array is zero sized,
  - if DIM is absent then the whole array is considered.
- COUNT(MASK[,DIM])
  - number of .TRUE. elements in an array along dimension DIM, the result is an array of rank $n-1$,
  - zero if array is zero sized,
  - if DIM is absent then the whole array is considered.
- MAXVAL(SOURCE[,DIM][,MASK])
  - maximum Value in an array
  - if DIM is specified the result is an array of rank $n-1$ of maximum values in other dimensions
  - if MASK is present then the survey is only performed on elements of SOURCE which correspond to .TRUE. elements of MASK
  - if DIM is absent the whole array is considered and the result is a single scalar.
  - the largest negative number of the appropriate kind is returned if the array is zero sized.
- MINVAL(SOURCE[,DIM][,MASK])
  - minimum value in an array (in an optionally specified dimension under an optional mask);
  - if DIM is specified the result is an array of rank $n-1$ of minimum values in other dimensions
  - if MASK is present then the survey is only performed on elements of SOURCE which correspond to .TRUE. elements of MASK
  - if DIM is absent the whole array is considered and the result is a single scalar.

- ◇ the smallest positive number of the appropriate kind is returned if the array is zero sized.

- □ `PRODUCT(SOURCE[,DIM][,MASK])`
  - ◇ product of array elements
  - ◇ if `DIM` is specified the result is an array of rank $n-1$ of the product
  - ◇ if `MASK` is present then the product only involves elements of `SOURCE` which correspond to `.TRUE.` elements of `MASK`
  - ◇ if `DIM` is absent the whole array is considered and the result is a single scalar.
  - ◇ if the array is zero sized then the product is 1

- □ `SUM(SOURCE[,DIM][,MASK])`
  - ◇ sum of array elements
  - ◇ if `DIM` is specified the result is an array of rank $n-1$ of the sum
  - ◇ if `MASK` is present then the sum only involves elements of `SOURCE` which correspond to `.TRUE.` elements of `MASK`
  - ◇ if `DIM` is absent the whole array is considered and the result is a single scalar.
  - ◇ if the array is zero sized then the sum is 0

Lecture 5:
Procedures

96: 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. May contain procedures.

- `MODULE`.

  a program unit which can contain procedures and declarations. It is intended to be attached to any other program unit where the entities defined within it become accessible.

There are two types of procedures:

- `SUBROUTINE`,

  a parameterised named sequence of code which performs a specific task and can be invoked from within other program units.

- `FUNCTION`,

  as a `SUBROUTINE` but returns a result in the function name (of any specified type and kind).

Guide for slide: 96

- execution always begins in the main `PROGRAM`.
- `SUBROUTINE` and `FUNCTION` calls may be made in the main program. A call transfers control temporarily to a parameterised named sequence of code which performs a specific task — a procedure. When this procedure has finished control is passed back to the line following the invocation.
- `MODULE` program units are new to Fortran 90 and provide the functionality to replace the need for things like `COMMON`, `INCLUDE`, `BLOCK DATA` and add a (limited) 'object oriented' aspect to the language.
- Modules are probably the most important new feature of Fortran and it will be shown why their use is so important.
- `SUBROUTINE` and `FUNCTION` are also known as procedures.
- procedures should be as flexible as possible to allow for software reuse.

97: Main Program Syntax

```
PROGRAM Main
 ! ...
CONTAINS ! Internal Procs
 SUBROUTINE Sub1(..)
  ! Executable stmts
 END SUBROUTINE Sub1
  ! etc.
 FUNCTION Funkyn(...)
  ! Executable stmts
 END FUNCTION Funkyn
 END PROGRAM Main
```

[PROGRAM [< *main program name* >]]
    < *declaration of local objects* >
        ...
    < *executable stmts* >
        ...
[CONTAINS
    < *internal procedure definitions* >]
END [PROGRAM [< *main program name* >]]

Guide for slide: 97

- This is the only compulsory program unit.
- the `PROGRAM` statement is optional however it is good policy to always use one. It can be any valid Fortran 90 name.
- the < *main program name* > is optional however it is good policy to always use one,
- the main program contains declarations and executable statements and may also contain internal procedures. The procedures are separated from the rest of the main program by a `CONTAINS` statement. These procedures may only be called from within the surrounding program unit — they automatically have access to all the host program unit's declarations but may also override them. These 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` indicating that the may be any number of internal procedures (subroutines or functions). They are wholly contained within the main program.
- the main program may also contain calls to `EXTERNAL` procedures. The diagram represents two external procedures, `ExtSub1` and `ExtFunn` indicating that the may be any number of external procedures (subroutines or functions). External procedures are totally outside of the main program. This will be discussed later.
- the main program must contain, as its last statement, an `END`. For neatness sake this should really be suffixed by `PROGRAM` (ie so it reads `END PROGRAM`) and should also have the name of the program unit attached too. Using as descriptive as possible `END` statements helps to reduce confusion.

98: Program Example

```
PROGRAM Main
 IMPLICIT NONE
 REAL :: x
  READ*, x
  PRINT*, FLOOR(x) ! Intrinsic
  PRINT*, Negative(x)
CONTAINS
 REAL FUNCTION Negative(a)
  REAL, INTENT(IN) :: a
   Negative = -a
 END FUNCTION Negative
END PROGRAM Main
```

Guide for slide: 98

- The example demonstrates a main program which calls an intrinsic function, (`FLOOR`), and an internal procedure, (`Negative`)
- 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.

99: Introduction to Procedures

The first question should be: "Do we really need to write a procedure?" Functionality often exists,

- intrinsics, Fortran 90 has 113,
- libraries, for example, NAg `fl90` Numerical Library has 300+, BLAS, IMSL, LaPACK, Uniras
- modules, number growing, many free! See WWW.

Library routines are usually *very fast*, sometimes even faster than Intrinsics!

Guide for slide: 99

- a procedure is a block of 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.
- 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, do not rely too heavily on global data or host-association (see later).
- try to give procedures useful names and initial descriptive comments,
- there is absolutely NO POINT in reinventing the wheel - if a procedure or collection of procedures already exist then it is invariably a good idea to use them. (There are over 100 intrinsic procedures and a very very large amount of existing libraries,)
  - the NAg library deals with solving numerical problems and is ideal for engineers and scientists. The NAg Fortran 90 Mk I library `f190` has just been released as a successor to the well respected and popular FORTRAN 77 library.
  - GKS is an international graphics standard - an implementation exists in FORTRAN 77 which should generally work with Fortran 90 programs (the depends on the compiler / linker).
  - other libraries include:
    - BLAS, Basic Linear Algebra Subroutines, for doing vector, matrix-vector and matrix-matrix calculations. Should *always* use these.
    - IMSL, akin to NAg Library
    - LaPACK, linear algebra package
    - Uniras, graphics routines, very comprehensive
  - there is an auxiliary Fortran 90 standard known as the "Varying String" module. This is to be added to the Fortran 90 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 (see later for operator overloading) so that `VARYING_STRING` objects can be used in more or less the same way as other intrinsic types.
  - It is envisaged that many modules will be implemented which will encompass a large functionality. Due to the portable nature of Fortran 90 these modules and their associated procedures will be widely available and will result in grater software reuse. (One of the goals of Object Oriented programming.)

100: Subroutines

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  CALL OutputFigures(NumberSet)
 .....
CONTAINS
 SUBROUTINE OutputFigures(Numbers)
  REAL, DIMENSION(:), INTENT(IN) :: Numbers
   PRINT*, "Here are the figures", Numbers
 END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

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* >]]

Note that, in the example, the `IMPLICIT NONE` statement applies to the whole program including the `SUBROUTINE`.

Guide for slide: 100

The subroutine here simply prints out its argument. Internal procedures can 'see' all variables declared in the main program. 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.

101: 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 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 declarations area instead of in the header.

Guide for slide: 101

Functions operate on the same principle to 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 $\sqrt{a^2+b^2}$.

102: Argument Association

Recall, on the SUBROUTINE slide we had an invocation:

```
CALL OutputFigures(NumberSet)
```

and a declaration,

```
SUBROUTINE OutputFigures(Numbers)
```

NumberSet is an *actual argument* and is *argument associated* with the *dummy argument* Numbers.
For the above call, in OutputFigures, the name Numbers is an **alias** for NumberSet. Likewise, consider,

```
PRINT*, F(a,b)
```

and

```
REAL FUNCTION F(x,y)
```

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.

Guide for slide: 102

- The parenthesised names following a function or subroutine name are call arguments. Procedures may have any number of such arguments.
- arguments at a call site are actual arguments (they use the actual name) and arguments in the procedure declaration are dummy arguments as their name is a substitute for the real 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.
- arguments must correspond in 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.]

103: 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 know as *local objects*. 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.

x will probably have a different size and shape on each call.
The space usually comes from the programs stack.

Guide for slide: 103

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.

104: Argument Intent

Hints to the compiler can be given as to whether a dummy argument will:

- only be referenced — `INTENT(IN)`;
- be assigned to before use — `INTENT(OUT)`;
- be referenced and assigned to — `INTENT(INOUT)`;

```
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
```

The use of `INTENT` attributes is recommended as it:

- allows good compilers to check for coding errors,
- facilitates efficient compilation and optimisation.

Note: if an actual argument is ever a literal, then the corresponding dummy must be `INTENT(IN)`.

Guide for slide: 104

New to Fortran 90,

- `INTENT(IN)`- value on entry equals value on exit - variable is not assigned to, (cf. constant).
- `INTENT(OUT)`- value on entry is undefined (or not used until assigned to). Must be assigned to.
- `INTENT(INOUT)`- object can be both used and assigned to.

The example,

- `arg1` has `INTENT(IN)` so it is simply used and not assigned to within the procedure.
- `arg2` has `INTENT(OUT)` so it is assigned to at least once inside the procedure.
- `arg3` has `INTENT(INOUT)` so it is first used and then reassigned to inside the procedure.

- `INTENT` not essential but it aids optimisation - can pass `INTENT(IN)` objects by value.
- also aids safety - if an `INTENT(IN)` object is assigned to an error will be generated. If an `INTENT(OUT)` object is not assigned to an error will also be generated.

105: Scoping Rules

Fortran 90 is *not* a traditional block-structured language:

- the *scope* of an entity is the range of program unit where it is visible and accessible;
- internal procedures can inherit entities by *host association*.
- objects declared in modules can be made visible by *use-association* (the `USE` statement) — useful for global data;

Guide for slide: 105

- an example of a block structured language is Pascal. Here the scope of a variable can be limited to a block construct (such as a loop) by making the declaration inside the block.
- a program unit extends for its inception to termination by a corresponding `END` statement;
- internal procedures can access all the objects of their containing outer program units. Internal procedures may have parameters and any actual parameters must not be assigned to. The internal procedures may declare new (local) objects with the same names as objects from the calling program unit - in this case the outer objects will become inaccessible from within the internal procedure.
- `USE` association attaches a module to the program unit and allows all objects with public visibility to be accessed.
- global data should be provided by a module. See later.

106: Host Association — Global Data

Consider,

```
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
```

Here, `NumberCalcsDone` is a *global* variable. It is available in all procedures by *host association*.

Guide for slide: 106

Internal procedures are new to Fortran 90:

- `PrintPay` and `NewTax` have access to the declaration of `NumberCalcsDone` by *host association*. Variables passed to procedures as arguments are available by argument association.
- `NewTax` cannot access any of the declarations of `PrintPay` and *vice-versa*.
- `PrintPay` and `NewTax` could, if desired, invoke other internal procedures which are contained by the same program.
- Upon return from either procedure the value of `NumberCalcsDone` will have increased by one owing to the last line of the procedure.

107: Scope of Names

Consider the following example,

```
PROGRAM Proggie
 IMPLICIT NONE
 REAL ::  A, B, C
 CALL sub(A)
CONTAINS
 SUBROUTINE Sub(D)
  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
END PROGRAM Proggie
```

In Sub, as A is argument associated it may not be have its value changed but may be referenced.
C in Sub is totally separate from C in Proggie, changing its value in Sub does **not** alter the value of C in Proggie.

Guide for slide: 107

- if an internal procedure declares an object with the same name as one in the outer (containing) procedure then this new variable supersedes the one from the outer scope for the duration of the procedure, for example, `C` in `Sub`.
- `Sub` accesses `A` (known locally as `D`) by argument association.
- `Sub` accesses `B` (and `A`) from `Proggie`. `C` is entirely local and supersedes `C` from `Proggie` for the duration of the call. When `Sub` is exited and control is returned to `Proggie`, the value that `C` had before the call to the subroutine is restored.
- `Sub` is forbidden to change the value or availability of `A` because it `A` associated with a dummy argument (P180 of Fortran 90 standard).

108: SAVE Attribute and the SAVE Statement

SAVE attribute can be:

- applied to a specified variable. NumInvocations is initialised on **first** call and retains its new value between calls,

```
SUBROUTINE Barmy(arg1,arg2)
 INTEGER, SAVE :: NumInvocations = 0
  NumInvocations = NumInvocations + 1
```

- applied to the whole procedure, and applies to *all* local objects.

```
SUBROUTINE Mad(arg1,arg2)
 REAL :: saved
 SAVE
 REAL :: saved_an_all
```

Variables with the SAVE attribute are *static* objects.
Clearly, SAVE has no meaning in the main program.

Guide for slide: 108

- The initialisation `NumInvocations = 0` is performed **once** conceptually at program start-up.
- the value of `NumInvocations` is retained between invocations of the procedure. The variable is said to be *static*
- objects initialised in a procedure implicitly have the `SAVE` attribute, in the first example, the following is exactly equivalent:

```
INTEGER :: NumInvocations = 0
```

- giving the whole procedure the `SAVE` attribute means every local object (not dummy arguments) is placed in the static storage class.
- clearly, the `SAVE` attribute has no meaning in the main program

109: Keyword Arguments

Can supply dummy arguments in any order using *keyword arguments*, for example, given

```
SUBROUTINE axis(x0,y0,l,min,max,i)
 REAL, INTENT(IN) :: x0,y0,l,min,max
 INTEGER, INTENT(IN) :: i
  .....
END SUBROUTINE axis
```

can invoke procedure:

- 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)
```

The names are from the *dummy arguments*.

Guide for slide: 109

- Normal argument correspondence is performed by position, the first actual corresponds to the first dummy and so on.
- using keywords allows this order to be changed.
- the actual argument list can specify which dummy argument a particular actual argument corresponds to.
- this is done by giving the *dummy* argument name followed by the actual argument (variable or literal).
- as in the example, once keyword arguments are used in a calling sequence all the remaining arguments (to the right of the first keyword argument) must also be keyword arguments.
- this is invaluable for optional arguments.
- if keyword arguments are to be used with external procedures then the `INTERFACE` must be explicit within the procedure in which it is invoked.

110: Keyword Arguments

Keyword arguments:

- allow arguments to be specified in any order.
- makes it easy to add an extra argument — no need to modify any calls.
- helps improve the readability of the program.
- are used when a procedure has optional arguments.

Note: once a keyword is used all subsequent arguments must be keyword arguments.

Guide for slide: 110

New to Fortran 90,

- allows arguments to be specified in any order.
- helps improve the readability of the program.
- makes it easy to add an extra argument without the need to modify users code (except for the interface declaration), this is an important point.
- easier to check that argument association is correct.
- this technique is often used in conjunction with optional arguments.

111: Optional Arguments

Optional arguments:

- allow defaults to be used for missing arguments;
- make some procedures easier to use.

Also:

- once an argument has been omitted all subsequent arguments must be keyword arguments,
- the `PRESENT` intrinsic can be used to check for missing arguments.

Guide for slide: 111

New to Fortran 90,

- dummy arguments with the `OPTIONAL` attribute are allowed to be missing from argument lists.
- optional arguments are specified in a procedure definition.
- default values may be used in place of missing arguments which will make some procedures easier to use.
- the `PRESENT` intrinsic, which should be used to test the status of optional arguments, returns a scalar `LOGICAL` result.
- if an extra argument is added it can be appended to the end of the argument list and given the optional attribute which will mean that a user does not have to modify his or her program to use the new routine. Only the interface declaration will need to be changed. (If the modified procedure is a `MODULE PROCEDURE` then even this will not be necessary.) The code will of course need recompiling.
- once an argument has been omitted all subsequent arguments must be keyword arguments, if this were not the case the compiler would often not be able to match the actual arguments with their corresponding dummies.
- 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. Intrinsic function interfaces are always visible to a calling program.

112: Optional Arguments Example

For example, consider the internal procedure,

```
SUBROUTINE SEE(a,b)
 REAL, INTENT(IN), OPTIONAL    :: a
 INTEGER, INTENT(IN), OPTIONAL :: b
 REAL    :: ay; INTEGER :: bee
 ay = 1.0; bee = 1 ! defaults
 IF(PRESENT(a)) ay  = a
 IF(PRESENT(b)) bee = b
  ...
```

both a and b have the optional attribute so 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)
```

Guide for slide: 112

- $\boxed{\text{stress}}$ that the keyword names come from the dummy arguments.
- the first call `CALL SEE()` uses both defaults.
- the second call uses both arguments so no defaults are used.
- the third call `CALL SEE(b=1,a=1.0)` again passes both arguments but they are specified by their keywords becauses they are supplied in the reverse order, it has the same effect as `CALL SEE(1.0,1)`.
- the forth and fifth calls both have on argument missing so will use the default for one of their arguments, in both cases the effect is identical.
- the final case is missing argument `a`.
- it is illegal to assign to a missing optional argument.
- for external procedures an `INTERFACE` must be present so the compiler knows the dummy argument names.

Lecture 6:
More Procedures

114: Dummy Array Arguments

There are two main types of dummy array argument:

- *explicit-shape* — all bounds specified;

  ```
  REAL, DIMENSION(8,8), INTENT(IN) :: expl_shape
  ```

  The actual argument that becomes associated with an explicit-shape dummy must conform in size and shape.

- *assumed-shape* — no bounds specified, all inherited from the actual argument;

  ```
  REAL, DIMENSION(:,:), INTENT(IN) :: ass_shape
  ```

  An explicit interface *must* be provided.

- dummy arguments cannot be (unallocated) `ALLOCATABLE` arrays.

Guide for slide: 114

- *explicit-shape* — all bounds specified. The actual argument that becomes associated with an explicit-shape dummy must conform in size and shape. No explicit `INTERFACE` is required. This approach is very inflexible very inflexible as only arrays of one fixed size can be passed. This form of declaration is best suited to non-dummy arguments.
- *assumed-shape* — no bounds specified, all inherited from the actual argument. An explicit interface must be provided. This approach should be used in preference to *assumed-size*. Arrays must match in type, kind and rank.
- an actual argument can be an `ALLOCATABLE` but a dummy argument cannot be - this means effectively that an `ALLOCATABLE` must be allocated before being used as an argument.

115: Explicit-shape Arrays

A dummy argument that is an explicit-shape array must conform in size and shape to the associated actual argument

```
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)                ! ERROR
  CALL subby(A3(::2,::4))       ! OK
  CALL subby(RESHAPE(A2,(/8,8/)) ! OK
  ...
CONTAINS
 SUBROUTINE subby(expl_shape)
  INTEGER, DIMENSION(8,8) :: expl_shape
  ...
 END SUBROUTINE subby
END PROGRAM Main
```

Guide for slide: 115

- `subby` can “accept any argument as long as it is an 8×8 default `INTEGER` array”! (Joke - get the class laughing with you not at you!)
- this is an inflexible approach which should not be used unless a routine will always be supplied with a fixed size array.

116: Assumed-shape Arrays

Should declare dummy arrays as assumed-shape arrays:

```
PROGRAM Main
 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
```

Note:

- the actual arguments cannot be a vector subscripted array,
- the actual argument cannot be an assumed-size array.
- in the procedure, bounds begin at 1.

Guide for slide: 116

New to Fortran 90 — this is the recommended way to pass arrays as arguments to procedures:

- only the rank and type (and kind) need to be specified in the procedure, the exact shape is assumed.
- sections can be passed so long as they are regular, i.e., not defined by vector subscripts. This is to do with efficiency, as a vector subscripted section will be non-trivial to find in the memory.
- if an actual argument were an assumed-size array then the bound / extent information of the last dimension would not be known so this information could not be passed to a further procedure.
- the second section in the third call `Y(2:4,4)` has one dimension, the dummy only has 2.
- if the procedure is external then an `INTERFACE` block is needed to transfer type, kind, bound / extent information into the procedure.

117: Automatic Arrays

Other arrays can depend on dummy arguments, these are called *automatic* arrays and:

- their size is determined by dummy arguments,
- they cannot have the SAVE attribute (or be initialised);

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)                 ! auto
  REAL :: A2(SIZE(A,1),SIZE(A,2)) ! auto
   ...
 END SUBROUTINE
END PROGRAM
```

The SIZE intrinsic or dummy arguments can be used to declare automatic arrays. A1 and A2 may have different sizes for different calls.

Guide for slide: 117

- automatic arrays have local scope and a limited lifespan. They cannot have the `SAVE` attribute as they are created and destroyed with each invocation of the procedure. They are traditionally used for workspace.
- they take their size from dummy arguments:
  - either from scalar dummy arguments which are passed as arguments, A1,
  - from characteristics of other dummy arguments, A2,
  - or a combination of the above.

118: 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))
```

As `X` has the `SAVE` attribute it will retain its allocation status between calls otherwise it would disappear.
As `Y` depends on a dummy argument it cannot be given `SAVE` attribute.

Guide for slide: 118

- SAVE gives X the static storage class which means that it persists between procedure calls.
- objects in COMMON cannot be given the SAVE attribute they already are static objects.

119: Explicit Length Character Dummy Arguments

An *explicit-length* dummy CHARACTER argument **must** match the actual in kind and rank (like explicit length arrays), however, it can be shorter than the actual.

```
PROGRAM Mian
 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), :: wird
  CHARACTER(LEN=10), :: werds(:)
  ...
 END SUBROUTINE char_example
END PROGRAM Mian
```

Guide for slide: 119

- *explicit-length* `CHARACTER` arguments are comparable to explicit-length arrays.
- 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.
- using explicit length objects is very inflexible because only fixed length strings can be passed.
- should really use other methods such as inheriting the length from the actual argument.
- can still use assumed-shape arrays for dummy array arguments.

120: Assumed Length Character Dummy Arguments

- `CHARACTER` dummys can inherit their length specifier (`LEN=`) from the actual argument;
- the kind and rank must still match;
- such an argument is an *assumed-length* character dummy;

```
PROGRAM Main
 IMPLICIT NONE
  ...
  CHARACTER(LEN=10) :: vari1
  CHARACTER(LEN=20) :: vari2
  CALL char_lady(vari1)
  CALL char_lady(vari2)
CONTAINS
 SUBROUTINE char_lady(word)
  CHARACTER(LEN=*), INTENT(IN) :: word
   ...
   PRINT*, "Length of arg is", LEN(word)
 END SUBROUTINE char_lady
END PROGRAM Main
```

will produce,

```
Length of arg is 10
Length of arg is 20
```

Guide for slide: 120

- assumed length character arguments are analogous to assumed-shape arrays, the length information is 'inherited' from the actual argument.
- much more flexible than explicit shape as any length of string can be passed as an actual argument.
- recommended method of passing strings.
- the actual length value can be acquired by using character inquiry intrinsics (for example, `LEN`) comparable to the `SIZE` intrinsic for array bounds.

121: Array-valued Functions

Functions can return arrays, for example,

```
PROGRAM Maian
 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, DIMENSION(SIZE(ima,1),SIZE(ima,2)) &
                   :: funnie
  funnie(:,:) = ima(:,:)*scal
 END FUNCTION funnie
END PROGRAM
```

Note how the `DIMENSION` attribute **cannot** appear in the function header.

Guide for slide: 121

- the size of the result of the function is determined in a fashion which is similar to the way that automatic arrays are declared,
- the bounds of `funnie` are inherited from the actual argument and then used to determine the size of the result arrays.
- a fixed sized result could be returned by declaring the result to be an explicit-shape array,
- by default, functions are assumed to return a scalar result - an explicit interface is needed if they return anything else.
- the result could be determined from any of the other dummy arguments (just like automatic arrays).
- if the function is external then an interface is mandatory.

122: Character-valued Functions

Can define a function that returns a CHARACTER string of a given length:

```
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
```

Here the length of the function result is determined by the dummy argument.

Guide for slide: 122

- the length of the result is taken from the length of the argument by using the `LEN` intrinsic.
- could also create automatic `CHARACTER` variables using `LEN`.
- could also declare the function to return a fixed length string each time — but this approach would be inapplicable in many cases.
- explicit interface is mandatory `LEN=*` is used.

123: Side Effect Functions

If, in

```
rezzy = funky1(a,b,c) + funky2(a,b,c)
```

`funky1` and `funky2` modify the value of, a then the order of execution could be important. Consider:

```
INTEGER FUNCTION funky1(a,b,c)
 ...
a = a*a
funky1 = a/b
```

and

```
INTEGER FUNCTION funky2(a,b,c)
 ...
a = a*2
funky2 = a/c
```

With `a=4`, `b=2` and `c=4` the following happens:

- if `funky1` executed first then `rezzy=8+8=16`
- if `funky2` executed first then `rezzy=2+32=34`

A properly constructed function should *not* change its arguments or any global entities.

Guide for slide: 123

The salient points are,

- both functions modify `a`
- if execution order is `funky1` followed by `funky2` then the following happens:
  - 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 execution order is `funky2` followed by `funky1` then the following happens:
  - 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

  So be careful!!!
- 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.
- 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`.

124: Recursive Procedures

In Fortran 90 recursion is supported as a feature.

- recursive procedures call themselves (either directly or indirectly),
- recursion is a neat technique
- recursion may incur certain efficiency overheads,
- recursive procedures must be explicitly declared
- recursive functions declarations must contain a `RESULT` keyword, and one type declaration refers to both the function name and the result variable.

Guide for slide: 124

- recursive procedures can call themselves or can call another procedure which then calls the first procedure again and so on. Any procedural call chain with a circular component exhibits recursion.
- recursive procedures must be explicitly declared using the `RECURSIVE` keyword. A recursive function declaration has a slightly different syntax to a regular function.
- the `RESULT` keyword, which must be used with recursive functions, specifies a variable name where the result of the function can be stored.
- the type of a recursive function can be specified on the function header:

  ```
  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 in the body of the function:

  ```
  RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
  INTEGER :: N_Fact
  ```

  In the last case `INTEGER :: N_Fact` implicitly gives a type to `fact`, in fact it is illegal to ALSO specify a type for `fact`.

125: Recursive Function Example

The following example calculates the factorial of a number 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
```

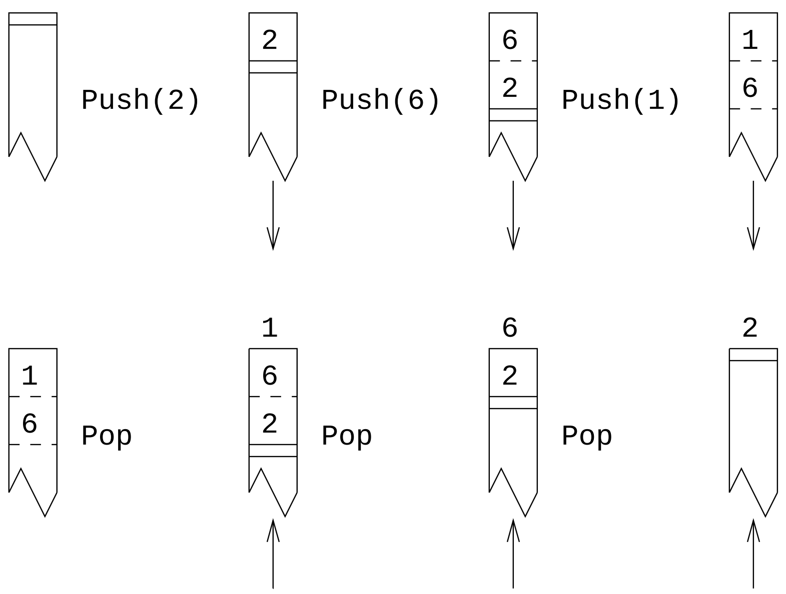
To calculate $4!$,

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.

Guide for slide: 125

- the `RESULT` keyword is necessary because there could easily be confusion in array valued recursive functions. A function reference and an array reference look very similar. (For example, 2D recursive function with 2 integer arguments that calls itself is ambiguous without the `RESULT`.)
- The recursive function repeatedly calls itself (as detailed on the slide). On each call, the value of the argument is reduced by one and the function called again. Recursion continues until the argument is zero. When this happens, the `IF` statement will dictate that the recursion begins to unwind and the result is calculated.

126: Stack Simulation

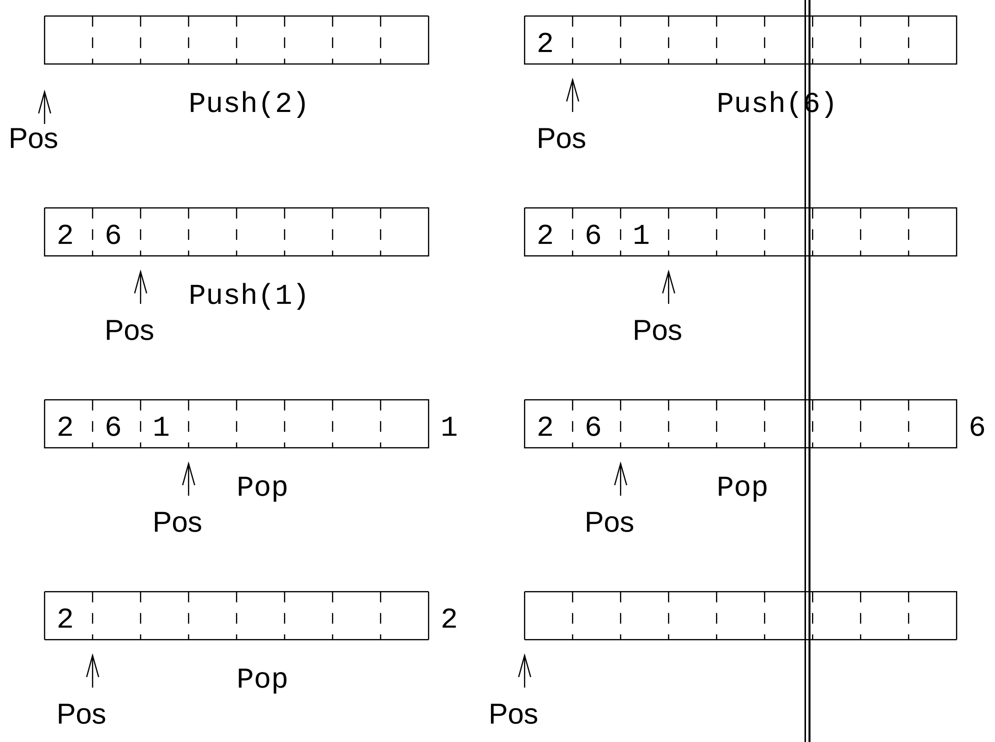


The important thing to remember is that one may only add and remove items from the top of the stack.

Guide for slide: 126

- 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 stuff is piled on the top the previous top elements get pushed down, as things are popped off the top the lower stack entries move towards the top.

127: Implementation of a Stack



The top of the stack is marked by an arrow, data is added to the right of this arrow and removed from the left.

Guide for slide: 127

- The 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. This we need three things to simulate a stack
  - a current position marker,
  - the maximum size of the stack (array)
  - the array to hold the data
- This data is needed by both `Push` and `Pop`, it should be global.

128: Stack Example Program

For example, the following defines a very simple 100 element integer stack and two access functions,

```
PROGRAM stack
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos=0
  ....
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
```

The main program can now call push and pop which simulate a 100 element INTEGER stack.

Guide for slide: 128

- if `store` and `pos` do not really need the `SAVE` attribute
- the `PARAMETER` allows the stack size to be easily modified and does not need the `SAVE` attribute (because its a constant),
- the `push` and `pop` procedures can be called with one argument — this argument will either be the element to put on or take off the stack.
- the current state of the stack, i.e., the number of elements on the sack and the values of each location, are stored as global data,
- if the stack limit is exceeded the program will terminate with an error. You would probably not do this in practise!
- this is **not** the best way of simulating a stack but simply demonstrates global data. (See later for an improvement.)

129: Reusability - Modules

A stack is a useful data structure. To allow it to be used elsewhere convert to a MODULE. This 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
```

Stack can now be accessed by other programs. The USE statement attaches it to a program:

```
PROGRAM StackUser
 USE stack
 IMPLICIT NONE
  ....
 CALL Push(14); CALL Push(21);
 CALL Pop(i); CALL Pop(j)
  ....
END PROGRAM StackUser
```

It is as if the code had been included in StackUser.

Guide for slide: 129

The stack and its access functions have been encapsulated into a module. The module contains the data structure and all the functions that operate upon this structure.

The `SAVE` statement behaves as usual, it is used so that `store` and `pos` retain their values between uses of the module.

The `USE` statement allows the module to be visible in `PROGRAM StackUser` and, by host association, will be visible in any procedures that are contained in the program.

130: Reusability - Modules II

Points raised on the previous slide:

- within a module, functions and subroutines are called *module procedures* ,
- module procedures may contain internal procedures (like PROGRAMs),
- module objects which retain their values should be given the SAVE attribute,
- modules can also be USEd by procedures and other modules,
- modules can be compiled separately. They should be compiled **before** the program unit that uses them.

Guide for slide: 130

- module procedures can only contain one level of internal procedures (same as `PROGRAM`s).
- 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.
- 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 `USE` the module,) it will go out of scope and any non-static objects (non-`SAVE`d objects) will disappear. A new instance will be created the next time the module is brought into scope, however, any values held by non-static variables defined at the top of a module will be lost.

131: Restricting Visibility

In stack example, the main program has access to store and pos and so could modify the values. This is dangerous.
Can prevent this by assigning visibility attributes:

```
PRIVATE :: pos, store, stack_size  ! hidden
PUBLIC  :: pop, push               ! not hidden
```

store, stack_size and pos are hidden, pop and push are not.
This makes the module **much** safer and allows internals of module to be changed without modifying the users program: only allow the user to access what he needs.
Alternatively, use statements or attributes;

```
PUBLIC                 ! set default visibility
INTEGER, PRIVATE, SAVE :: store(stack_size), pos
INTEGER, PRIVATE, PARAMETER :: stack_size = 100
```

so, in the main PROGRAM:

```
CALL Push(21); CALL Pop(i)  ! OK
pos = 22; store(pos) = 99   ! Forbidden
```

Guide for slide: 131

- data hiding has been introduced as a safety feature, it will stop anybody who uses a module from doing anything too daft such as changing module objects that are used for 'housekeeping'. It also allows the module to be updated / modified without the users knowledge. If only the private variables are modified the user should be unaware of any difference.
- the default accessibility is `PUBLIC` which means that if accessibility is not specificed the user will have access to all entities in a module.
- if a module with private entities is used in another module then the first modules' private entities cannot be seen in the second; there is no way of allowing entities to be public to a module and private to a different program unit.

132: The USE Renames Facility

The USE statement names a module whose public definitions are to be made accessible.
Syntax:

USE < *module-name* > &
    [,< *new-name* > => < *use-name* >...]

module entities can be renamed,

```
USE Stack, IntegerPop => Pop
```

The module object Pop is renamed to IntegerPop when used locally.

Guide for slide: 132

- 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,
- as many renames as desired can be made in one `USE` statement, they are just supplied as a comma separated list.
- can only rename the same object once per program unit.

133: USE ONLY Statement

Another way to avoid name clashes is to only use those objects which are necessary. It has the following form:

`USE` < *module-name* > [`ONLY:` < *only*-list >...]

The < *only*-list > can also contain renames (=>).
For example,

```
    USE Stack, ONLY:pos, &
           IntegerPop => Pop
```

Only `pos` and `Pop` are made accessible. `Pop` is renamed to `IntegerPop`. The `ONLY` statement gives the compiler the option of including only those entities specifically named.

Guide for slide: 133

- can mix renaming and use-only in one USE statement.
- it is possible to restrict the availability of objects declared in a module made visible use association by using the ONLY clause of the USE statement,
- this could be used as an alternative to renaming module entities, 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.
- using the ONLY clause gives the compiler the option of just including the object code associated with the entities specified in the ONLY clause instead of the code for the whole module. This has the potential to make the executable code smaller and faster.
- the USE statement with its renaming and restrictive access facilities gives great control enabling two modules with similar names to be accessed by the same program with only minor inconvenience.

134: Module - General Form

```
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
```

MODULE < *module name* >
    < *declarations and specifications statements* >
[CONTAINS
    < *definitions of module procedures* >]
END [MODULE [< *module name* >]]

Guide for slide: 134

- < *module name* > is the name that appears in the `USE` statement, it is not (but can be) the filename.
- partial-inheritance is possible. Basically this means that one module can inherit the environment of another module either fully or partially by using a `USE` or `USE .. ONLY` clause.
- declarations and specifications include:
  - user-defined type definitions,
  - `USE` statements,
  - object definitions, (anything that could appear in a main program can be found here),
  - object initialisations
  - accessibility statements (see later),
  - interface declarations (of external procedures);
- the `CONTAINS` block houses module procedures:
  - these procedures are accessible when the module is used,
  - each module procedure may also contain internal procedures,
  - the module procedure are written in exactly the same way as regular (external) procedures.
- Entities of other modules can be accessed by a `USE` statement, *use association*, as the first statement in the specification pa rt.
- Non-circular dependent definition chains are allowed (one module `USE`s an other which `USE`s another and so on) providing a partial-inheritance mechanism.

135: Modules — An Overview

The `MODULE` program unit provides the following facilities:

- global object declaration;
- procedure declaration (includes operator definition);
- semantic extension;
- ability to control accessibility of above to different programs and program units;
- ability to package together whole sets of facilities;

Guide for slide: 135

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 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 declaration;

  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 using the module. The objects can be given a static storage class so will retain their values between uses. (Derived types should be declared in a module — see later.)

  Sets of variable declarations which are not globally visible can also be made in a module and used at various places in the code which achieves the functionality of the `INCLUDE` statement.

- interface declaration;

  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. Only really useful when module procedures cannot be used.

- procedure declaration;

  Can put any procedures in a module which will be visible to any program unit which uses the module. This has the advantage that all the interfaces are already explicit within the module, so there is no need for any `INTERFACE` blocks to be written.

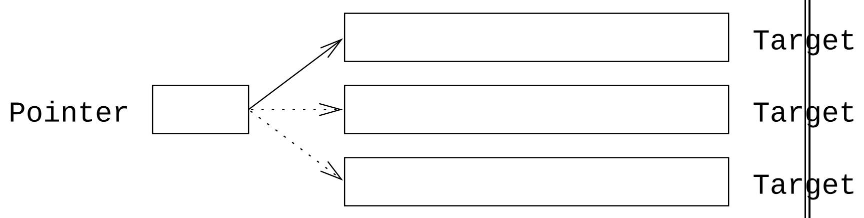
We only discuss the above here, the following topics will be discussed later:

- semantic extension,
- controlled object accessibility,
- packaging derived types,
- defining and overloading operators and procedures,
- defining generic interfaces to provide object oriented capabilities.

Lecture 7:
Pointers and Derived Types

137: 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.



- The pointer often uses less space than the target.
- A reference to the pointer will in general be a reference to the target, (pointers are automatically dereferenced).

Guide for slide: 137

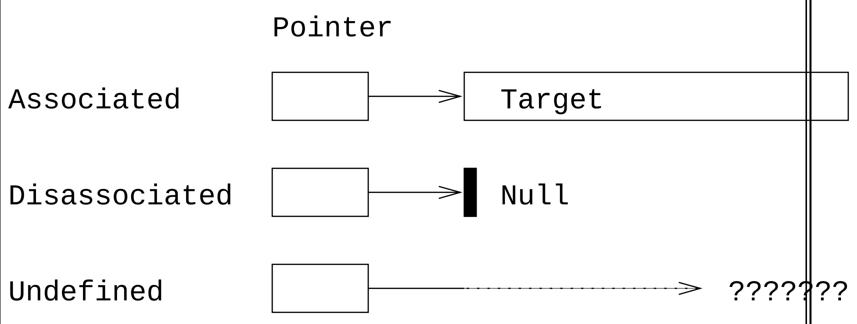
- The diagram shows a pointer variable pointing to an area of storage (its target).
- The area of storage is shown to be larger than the pointer to indicate that retargeting the arrow (changing the memory location stored in the pointer variable) could very well be computationally cheaper than moving the target (copying the numbers stored in the target locations).
- The target of the pointer can be any of the indicated areas of storage, but a pointer can only have one target at any one time.
- pointers are very useful as they can be used to access variables indirectly (aliasing a variable) and provide a more flexible alternative to `ALLOCATABLE` arrays. In conjunction with user-defined types they can also be used to create dynamic and recursive data structures such as linked lists and binary trees.
- pointers are useful in situations where data is being created and destroyed dynamically and it cannot be predicted in advance how this will occur.
- The use of pointers often has a detrimental effect on the performance of code but they can sometimes can be used to 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) which can be retargeted 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 using them in certain cases will conserve memory, for example, moving large arrays about in memory will involve a large and costly copy operation whereas retargeting a couple of pointers is a far simpler solution.
- a pointer can be reassigned just like a normal (non-pointer) variable.
- a target may be the target of more than one pointer.
- 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 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). Pointers cannot point to literals.
- pointer variables do not hold data, they point to scalar or array variables which themselves may contain data,
- A reference to the pointer will in general be a reference to the target, (pointers are automatically dereferenced). (Dereferencing means following the pointer to its target and using the target value at the place where the pointer is referenced.) On most occasions, when the pointer is referenced it is the value of the space that the pointer references that is used.

138: Terminology

A pointer has 3 possible states:

- if a pointer has a particular target then the pointer is said to be *associated* with that target,
- a pointer can be made to have no target — the pointer is *disassociated*,
- the initial status of a pointer is *undefined*,

Visualisation,



Use `ASSOCIATED` intrinsic to get the (association) status of a pointer.

Guide for slide: 138

- pointer variables have 3 states:
  - associated — the pointer has a target,
  - disassociated — the pointer does not have a target and is null valued.
  - undefined — this is the state after definition. The dotted line and question marks indicate the undefined status.
- anything that is to be pointed at must have the `TARGET` attribute (either explicitly or implicitly).

Other relevant facts,

- there is an scalar `LOGICAL` intrinsic function `ASSOCIATED` which returns the (association) status of a pointer — `.TRUE.` if the pointer is associated with a target and `.FALSE.` if it is disassociated. The result is undefined if the pointer status is itself undefined.
- as good practise one should *always* nullify pointers before use (with the `NULLIFY` statement). This will give the pointer disassociated status. As an unassociated pointer has an undefined status, it should not be used at all — not even in an `ASSOCIATED` function call. The two compilers available to me (at the time of writing) gave different results when `ASSOCIATED` was called with an undefined argument.
- Fortran 95 will allow pointers to be nullified on declaration.

139: Pointer Declaration

A `POINTER`:

- ☐ is a variable with the `POINTER` attribute;
- ☐ has static type, kind and rank determined by its declaration, for example,

```
REAL, POINTER :: Ptor
REAL, DIMENSION(:,:), POINTER :: Ptoa
```

  - ◇ `Ptor` is a pointer to a scalar real target,
  - ◇ `Ptoa` is a pointer to a 2-D array of reals.

So,

- ☐ the declaration fixes the type, kind and rank of the target;
- ☐ pointers to arrays are declared with deferred-shape array specifications;
- ☐ the rank of a target is fixed but the shape may vary.

Guide for slide: 139

- the attributed statement is exactly the same as other attributed declarations.
- The first declaration specifies that the name `Ptor` is a pointer to scalar a `REAL` target, and the second specifies that `Ptoa` is a pointer to a rank 2 array of reals. Pointers cannot point to literals.
- the targets of pointers to arrays are allocated just like allocatable arrays (using the `ALLOCATE` statement) - there are subtle differences between pointers to arrays and `ALLOCATABLE` arrays.
- as with `ALLOCATABLE` arrays it is not possible to extend the size of an allocated object; if a larger object is required then a new object must be declared afresh.

140: Target Declaration

Targets of a pointer must have 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 the previous slide):

- `x` or `y` may become associated with `Ptor`;
- `a, b` or `c` may become associated with `Ptoa`.

Guide for slide: 140

- pointers can very easily have a detrimental effect on the performance of a program, it was felt that Fortran is always associated with efficiency and so pointers should be as efficient as possible. To this end the `TARGET` attribute was introduced.
- a pointer may not point at a variable that does not have this attribute.
- The attribute is implicit for pointer variables.
- a target may have any number of things pointing to it.
- a target of a 2D pointer can have any shape but must have two dimensions. A subsection of an array can be a target but only if it is defined using subscript triplets.
- from the Fortran 90 standard:

  "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 in Fortran 95. The word *solely* is to replaced by *primarily* in Fortran 95 because the former is felt to be misleading. The `TARGET` attributed is also used at procedure boundaries to say that one of the arguments is a `TARGET`. Thus the `TARGET` attribute has uses other than for optimisation.

141: Pointer Manipulation

The following operators manipulate pointers:

- =>, *pointer assignment* — alias a pointer with a given target;
- =, *'normal' assignment* — assign a value to the space pointed at by the pointer.

Pointer assignment makes the pointer and the variable reference the same space while the normal assignment alters the value contained in that space.

Guide for slide: 141

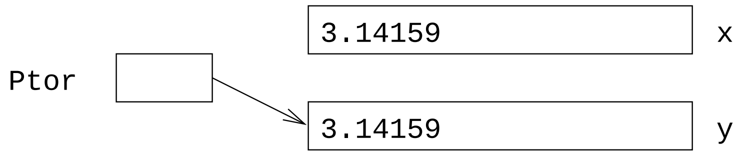
Pointers have two separate forms of assignment — they should not be confused as they have very different results.

- *pointer assignment*
  - can take place between a pointer variable and a target variable, or between two pointer variables.
  - is a form of `aliasing` (referring to an object by a different name), the pointer and its target refer to the same space.
  - there is no implicit or explicit type coercion performed here.
- *'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* have 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; it is not being pointed to.) 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. The aliased object can be assigned to in the normal way, therefore, so can its alias.
  - 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.
  - normal Fortran 90 type coercion is performed.

142: Pointer Assignment

Consider,

```
x = 3.14159
Ptor => y
Ptor = x
```



- x and Ptor have the same value.
- Ptor is an alias for y so the last statement sets y = 3.14159.
- if the value of x is subsequently changed, the value of Ptor and y do not.

Coding,

```
Ptor = 5.0
```

sets y to 5.0.

Guide for slide: 142

- `x = 3.14159` is a regular assignment, gives the variable `x` a value.
- pointers cannot point to literals, only variables.
- `Ptor => y` alias's `y` to be `Ptor`, `y` at this stage `y` has not been assigned a value.
- `Ptor = x` sets the space pointed at by `Ptor`, (i.e., `y`) to `3.14159`.
- `x` can be changed and will never alter the value of `y`.
- any assignments / references to `Ptor` are also assignments / references to `y`.

143: Association with Arrays

An array pointer may be associated a *regular section* of a target of the correct rank (and type and kind). It cannot, however, be associated with a vector subscripted array section.
Assuming the same declarations as the previous slide:

- these are valid (2D sections):

```
Ptoa => a(3:5,::2)
Ptoa => a(1:1,2:2)
```

- these are invalid (wrong rank):

```
Ptoa => a(1:1,2)
Ptoa => a(1,2)
Ptoa => a(1,2:2)
```

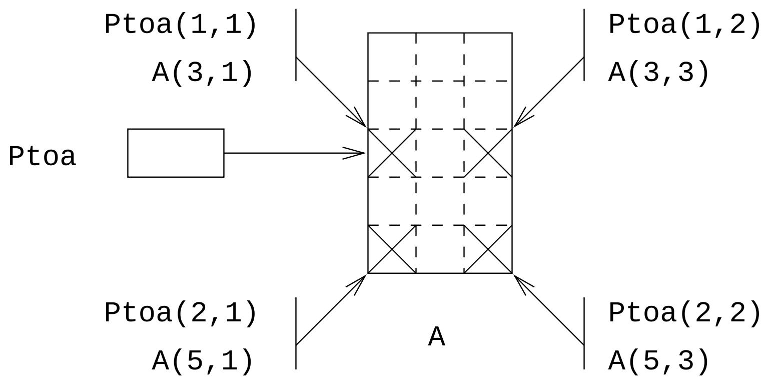
- this is invalid (vector subscript):

```
v = (/2,3,1,2/)
Ptoa => a(v,v)
```

Guide for slide: 143

- *regular section* is a section that can be defined by a linear function; a *subscript-triplet* defines a regular section.
- an array dimension can be collapsed by specifying a scalar value in one of the dimensions — this reduces the rank of the particular array reference.
- `Ptoa => a(3:5,::2)` alias's this particular section with the name `Ptoa`, `Ptoa(1,1)` refers to element `a(3,1)`; `Ptoa(3,2)` refers to to element `a(5,3)` and so on. `SIZE(Ptoa)` gives `6` and `SHAPE(Ptoa)` gives `(/ 3, 2 /)`.
- `Ptoa => a(1:1,2:2)` alias's `Ptoa` to a 1×1 2D array. Note that the section specifiers uphold the rank even though they only specify one element.
- `Ptoa => a(1:1,2)` etc., are invalid because the rank of the array reference is not 2.
- the last example is invalid because the RHS array reference is not a regular section. (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.)
- as seen earlier `Ptoa` could be associated with the whole of `a`.

144: Visualisation of Pointers to Sections




```
Ptoa => a(3::2,::2)
```

Here the top left subsection element can be referred to through TWO names

- `A(3,1)`
- `Ptoa(1,1)`

Guide for slide: 144

The diagram shows how element `Ptoa(1,1)` corresponds to element `A(3,1)` and so on.

`Ptoa` is an alias for a subsection of `A`.

145: Dynamic Targets

Targets can also be created dynamically by allocation. `ALLOCATE` can make space become the target of a pointer.

```
ALLOCATE(Ptor,STAT=ierr)
ALLOCATE(Ptoa(n*n,2*k-1),STAT=ierr)
```

- the first statement allocates a single real as the target of `Ptor`.
- the second allocates a rank 2 real array as the target of `Ptoa`.

It is not an error to allocate an array pointer that is already associated.

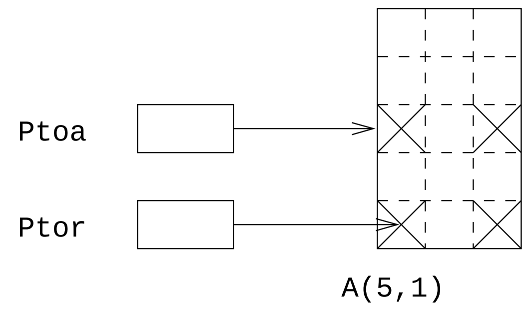
Guide for slide: 145

- can allocate a target for a pointer variable; 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.
- pointers to arrays can be used much like `ALLOCATABLE` arrays in the sense that they can be allocated and deallocated.
- the status specifier `STAT=` should always be used in allocate statements; a zero value means that the allocate request was successful, 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.
- 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.

146: Automatic Attributing

Pointer variables implicitly have the `TARGET` attribute.

```
Ptoa => A(3::2,1::2)
Ptor => Ptoa(2,1)
```



Element `(2,1)` of `Ptoa` is associated with `Ptor`.
Here the bottom left subsection element can be referred to through THREE names:

- `Ptor`
- `A(5,1)`
- `Ptoa(2,1)`

When a new association for a pointer is established any previous association with a target is broken.

Guide for slide: 146

- Pointers automatically have the `TARGET` attribute,
- NOTE: allocatable arrays **do not** have the `TARGET` attribute.
- Uses of this type of aliasing are common, for example, if a particular array element was being used frequently within a nested loop, the calculation of the particular element need only be done once. Subsequent references to the element can then be made through the scalar pointer.
- Note: when a new association for a pointer is established either by allocation or by pointer assignment, any previous association with a target is broken. Care must be taken not to 'lose' this space.

147: Association Status

The status of a defined pointer may be tested by an 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 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.`.

Guide for slide: 147

- ASSOCIATED is a scalar LOGICAL intrinsic function,
- if the pointer is undefined then so is the result. Edinburgh and NAg compilers give opposite results if ASSOCIATED is given an undefined pointer argument—neither give an error.
- ASSOCIATED is useful in situations analogous to linked list processing.
- ASSOCIATED is useful to avoid deallocation errors:

```
IF(ASSOCIATED(Ptoa)) DEALLOCATE(Ptoa, STAT=ierr)
```

148: Pointer Disassociation

Pointers can be disassociated with their targets by:

- nullification

  ```
  NULLIFY(Ptor)
  ```

  - breaks the connection of its pointer argument with its target (if any),
  - disassociates the pointer.

  Note: it is good practise to nullify *all* pointers before use.

- deallocation

  ```
  DEALLOCATE(Ptoa, STAT=ierr)
  ```

  - breaks the connection between the pointer and its target,
  - deallocates the target.

Guide for slide: 148

- NULLIFY should be used when a pointer is to be disassociated from its target and not immediately associated with any other target.
- NULLIFY disassociates the pointer with its target but **does not** deallocate the targets storage. This space will be inaccessible until the program terminates unless pointed to by at least one more pointer.
- a nullified pointer has ASSOCIATED(Ptor) as .FALSE..
- immediately after declaration, 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 used **anywhere**! (Note: not all compilers report this as an error!!)
- a list of pointers to nullify may be specified.
- DEALLOCATE returns space to the heap storage for re-use and also disassociates the pointer — it should be used when the storage is finished with, for example, at the end of a procedure.
- deallocating a disassociated pointer is an error.
- STAT= should always be used to report on the success / failure of the deallocation request; as usual zero indicates success, a positive value means failure.
- can deallocate ALLOCATABLE arrays and POINTER variables in the same statement.
- it is a good idea to only deallocate one object per DEALLOCATE statement in case there is an error; it will then be possible to tell which object is inducing the error.
- it is an error to deallocate anything that has not been first allocated
- it is an error to deallocate an object that has more than one pointer associated with it. The other pointers should be nullified before deallocation is performed.

149: Pointers to Arrays vs. Allocatable Arrays

There are a number of subtle differences between `ALLOCATABLE` and `POINTER` arrays:

- `ALLOCATABLE` arrays are more efficient,
- `TARGET` arrays may be referred to by a number of different names,
- unallocated `ALLOCATABLE` arrays cannot be actual arguments but unassociated (unallocated) `POINTER` arrays can be.

In summary,

- `ALLOCATABLE` arrays are more efficient,
- `POINTER`s are more flexible.

Guide for slide: 149

- ALLOCATABLE arrays are more efficient.
- POINTERs are more flexible.

150: Practical Example

Pointers can be of great use in iterative problems. Iterative methods:

- make guess at required solution;
- use guess as input to an equation to produce better approximation;
- use new approximation to obtain better approximation;
- repeat until degree of accuracy is obtained;

```
REAL, DIMENSION(100,100), TARGET :: app1, app2
REAL, DIMENSION(:,:), POINTER :: prev_app, &
                                 next_app, swap
prev_app => app1
next_app => app2
prev_app = initial_app(.....)
DO
 next_app = iteration_function_of(prev_app)
 IF(ABS(MAXVAL(next_app-prev_app))<0.0001)EXIT
 swap => prev_app
 prev_app => next_app
 next_app => swap
END DO
```

Using pointers here avoids having to copy the large matrices.

Guide for slide: 150

- declare two arrays which can be targets of pointers

```
REAL, DIMENSION(100,100), TARGET :: app1, app2
```

- declare 3 array pointers (can point to `app1` and `app2`).

```
REAL, DIMENSION(:,:), POINTER :: prev_app, &
                                 next_app, swap
```

- set up initial approximations, `initial_app` is a function that guesses at the initial approximation.

```
prev_app => app1
next_app => app2
prev_app = initial_app(.....)
```

- the `DO` loop contains the iteration which gradually moves towards the solution,
- `iteration_function_of` is a function that (somehow) moves towards the solution; it uses the values on `prev_app` and improves them to be closer to the solution;

```
next_app = iteration_function_of(prev_app)
```

- the difference in the two most recent approximations is examined. 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 `0.0001` then a further iteration is performed. The supposition is that when values remain 'constant' between iterations this means they have reached their optimum value. When all array elements have stabilised the solution has been found.

```
IF(ABS(MAXVAL(next_app-prev_app))<0.0001) EXIT
```

- this shuffles the arrays around (turns the current approximation in to the previous one for the next iteration). `next_app` ends up pointing to the old `prev_app` which has outlived its usefulness — on the next iteration this array can be used to hold the next set of approximations.

```
swap => prev_app
prev_app => next_app
next_app => swap
```

  Doing this form of manipulation replaces two potentially expensive copy operations with 3 relevantly inexpensive pointer reassociation operations.

151: Pointers and Procedures

A actual pointer argument can be interpreted in two ways:

- immediately dereference and pass the target — the corresponding dummy argument does **not** have the `POINTER` attribute;
- pass the pointer so it can be manipulated as a pointer in the procedure — the corresponding dummy argument must 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
```

Note: a `POINTER` dummy argument **cannot** have the `INTENT` attribute. An explicit interface is required if a pointer is used as an actual argument to a procedure.

Guide for slide: 151

- an unallocated `ALLOCATABLE` array cannot be passed as an actual argument to a procedure, an unassociated (unallocated) `POINTER` array can!
- 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.
- pointer arguments must match in type, kind and rank. (Just like all other arguments.)
- a `POINTER` dummy argument **cannot** have the `INTENT` attribute.
- when passing `POINTER` variables to procedures it is recommended that error checking, using the `STAT=` specifier and the `ASSOCIATED` function should be used whenever appropriate. It is very easy to get confused about association status when the code to manipulate the pointer is spread over a number of different program units.

152: Pointer Valued Functions

A function may be pointer valued,

```
PROGRAM main
 IMPLICIT NONE
 INTEGER, TARGET :: a, b
 INTEGER, POINTER :: largest
 a = ...; b = ...
 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
```

Guide for slide: 152

- `ptr` return a pointer to the largest out of `a` and `b`.
- the Ellis et al book says that, with pointer valued functions "the result keyword must be used", this is incorrect.
- the interface of a pointer valued function must be explicit.
- cannot specify that the function is pointer valued in the `FUNCTION` statement, it must be done in the declarations part.
- assignment to the function result variable is by pointer assignment.

153: Pointer I/O

Pointers which appear in input/output lists:

- are always dereferenced;
- must not be disassociated as dereferencing would make no sense;

```
PROGRAM Main
 IMPLICIT NONE
  REAL, DIMENSION(3), TARGET :: arr=(/1.,2.,3./)
  REAL, DIMENSION(:), POINTER :: p, q
   p => arr
   PRINT*, p
   ALLOCATE(q(5))
   READ*,  q
   PRINT*, q
   DEALLOCATE(q)
   PRINT*, q ! invalid, no valid target
END PROGRAM Main
```

Guide for slide: 153

- pointers are immediately dereferenced in I/O statements and so behave just as if their targets were referenced instead.
- the same principles apply for for `READ` statements too.
- in the example the `STAT` specifier has been omitted to conserve space.
- in the example, `p` is dereferenced and the three real numbers are printed out,
- once `q` has been allocated it can be used; the `READ` statement expects 5 real numbers to be entered on the standard input channel. `q` is immediately dereferenced in both the `READ` and `PRINT` statements.
- it is not possible to print out the value of a pointer.

154: Derived Types

It is often advantageous to express some objects in terms of aggregate structures, for example: coordinates, (x, y, z).
Fortran 90 allows compound entities or *derived types* to be defined:

```
TYPE COORDS_3D
 REAL :: x, y, z
END TYPE COORDS_3D
TYPE(COORDS_3D) :: pt1, pt2
```

Derived types definitions should be placed in a MODULE.

Guide for slide: 154

- derived types are also known as structures or records in other languages.
- it is useful from a programming point of view to treat a collection of objects as one entitly, for example, a set of coordinates (2 or 3 real numbers) or an address (some lines of text and numbers).
- 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.
- a derived type is made up from a collection of existing types so every derived type can be resolved into components of the Fortran 90 intrinsic types.
- new types are declared in a *derived-type-statement*.
- 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.)
- the declaration of `centre` inside the `SPHERE` type declaration block is an example of derived type object declaration, note the similarity to intrinsic object declarations,

155: Supertypes

Previously defined types can be used as components of other derived types,

```
TYPE SPHERE
 TYPE (COORDS_3D) :: centre
 REAL             :: radius
END TYPE SPHERE
```

Objects of type SPHERE can be declared:

```
TYPE (SPHERE) :: bubble, ball
```

Guide for slide: 155

- a 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. (See later - this is how recursive data structures are formed.)
- a sphere (like a circle) 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
```

- an object of type `SPHERE` is declared in a very similar way to a regular type declaration. The type declaration can be attributed in the same way as for regular type declarations, for example, can have `DIMENSION`, or `ALLOCATABLE` attributes and, subject to a small proviso mentioned later, can be used as dummy or actual arguments and so can have the `INTENT` or `OPTIONAL` attributes. The `PARAMETER` attribute cannot be used.
- cannot initialise a component in a derived type declaration statement.
- There is no sequence association for derived types, i.e., there is no reason that objects of type `COORDS_3D` should occupy 3 contiguous `REAL` storage locations.

156: Derived Type Assignment

Values can be assigned to derived types in two ways:

- component by component;
- as an object.

An individual component may be selected, using the % operator:

```
pt1%x = 1.0
bubble%radius = 3.0
bubble%centre%x = 1.0
```

The whole object may be selected and assigned to using a constructor:

```
pt1 = COORDS_3D(1.,2.,3.)
bubble%centre = COORDS_3D(1.,2.,3.)
bubble = SPHERE(bubble%centre,10.)
bubble = SPHERE(COORDS_3D(1.,2.,3.),10.)
```

The derived type component of SPHERE must also be assigned to using a constructor.
Assignment between two objects of the same derived type is intrinsically defined,

```
ball = bubble
```

Guide for slide: 156

Objects of a derived type can have values assigned in one of two ways,

- component by component:
  - the `%` operator can be used to select a single specific component of a derived type object; in this way the object can be given values in a piecewise fashion. In order to select a component it is necessary to know how the derived type was initially declared, the names of the components need to be known.
  - the object `bubble` is of type `SPHERE` which we know is composed of `centre` and a `radius`, however, `centre` is a derived type component so the individual components must be selected from this type too. In order to assign a value (an intrinsic typed variable or constant) to a derived type component, it must be ensured that the component is fully resolved, i.e., the component selected is itself of intrinsic type.
  - In the example, `bubble%radius` is of type `REAL` but `bubble%centre` is of type `COORDS_3D` and must be resolved further in order to perform assignment of a single intrinsically typed object.
  - `bubble%centre%x` (or `..%y` or `..%z`) is of intrinsic type and can therefore be given a value using the intrinsic assignment operator `=`. (Note we could overload the `=` operator to make assignment simpler — see later.)
  - normal type coercion rules still apply, i.e., an `INTEGER` could be assigned to a `REAL` component of a derived type — the integer would simply be promoted.
- as an object:
  - a derived type constructor can also be used to assign values. This structure constructor takes the form of the type name followed by a parenthesised list of values for each of the components.
  - `bubble%centre` is of type `COORDS_3D` so we can use the automatically defined derived type constructor (`COORDS_3D`) — this constructor must be supplied with 3 numeric components.
  - 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`.
  - `COORDS_3D(1.,2.,3.)` could be used in place of `bubble%centre` but the presense of the derived type name to force type coercion is absolutely essential. You could not 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. This means that the components must be listed in order with any coercion to derived types being explicit.
- assignment between two derived type objects of the *same* type is intrinsically defined. `ball` and `bubble` can be equated.

157: Arrays and Derived Types

It is possible to define derived type objects which contain non-ALLOCATABLE arrays and arrays of derived type objects.
Consider,

```
TYPE FLOBBLE
 CHARACTER(LEN=6)           :: nom
 INTEGER, DIMENSION(10,10) :: harry
END TYPE FLOBBLE
 ...
TYPE (FLOBBLE)                :: bill
TYPE (FLOBBLE), DIMENSION(10) :: ben
```

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)
```

but not a collection of elements or subsections of the array component:

```
ben(9:9)%harry(:,:) ! invalid
ben(:)%harry(:,::2) ! invalid
```

Guide for slide: 157

- most objects can be used as components of a derived type, i.e., all the intrinsic types, other derived types and arrays of intrinsic or derived types. `ALLOCATABLE` arrays however cannot a pointer to an array must be used.
- the example defines a derived type which contains an array.
- can also declare an array of objects where each element of the array also contains an array, however, cannot refer to an array of arrays as one object.
- the five expressions in the first example all define some sort of array section. There is a scalar index on at least one side of the `%` operator. In a derived type that has more levels of derived type nesting there can still only be sectioning of at most one component level.
- in the second example of invalid object selection, the first expression selects a one element subsection of `ben` and the whole of the `harry` component; remember that `ben(9)` is scalar and `ben(9:9)` is non scalar.
- the second expression is more obviously incorrect, a section is taken from both components which is not legal.

158: Derived Type I/O

Derived type objects which do not contain pointers (or private) components may be input or output using 'normal' methods:

```
    PRINT*, bubble
```

is exactly equivalent to

```
    PRINT*, bubble%centre%x, bubble%centre%y, &
            bubble%centre%z, bubble%radius
```

Derived types are handled on a component by component basis.

Guide for slide: 158

- 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 nor must the visibility of the components be restricted (see later). (For the first case this is because the pointer reference may never be resolved, for example, if a circular list structure were to appear in an output statement the program would never terminate.)
- a derived type object is written out as if all the components were explicitly inserted into the output statement in the same order in which they were defined.
- a derived type object is read in as if all the components were explicitly inserted into the input statement in the same order in which they were defined.
- derived types containing arrays are handled in the expected way; individual components are input / output one at a time and then arrays are dealt with in array element order.

159: Derived Types and Procedures

Derived types definitions should be packaged 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.

Guide for slide: 159

- two identical textual descriptions of a type definition statement in two separate program units will not have the desired effect of making the same type visible in two separate places. These two specifications, by definition, do not represent the same `TYPE` (unless the type has the `SEQUENCE` attribute). To overcome this the type definition *must* be placed in a module.
- this is because of optimisation — the compiler can represent the type however it sees fit.
- derived type arguments behave like intrinsically typed arguments, they can be given attributes (`OPTIONAL`, `INTENT`, `DIMENSION`, `SAVE`, `ALLOCATABLE`, etc), and interfaces are required in the same cases as for intrinsic types.
- a type definition can only become accessible by host or use association, this means a VERY GOOD place for type definitions is in a `MODULE`. Cannot use an `INCLUDE` file!
- the renaming facility can be used to change the name of the type if desired.

160: 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 definitions of `VecDef` must be made available by use or host association.

Guide for slide: 160

This is exactly the same principle as for intrinsic typed functions except that the interface must be visible in the program unit that is doing the calling.

Lecture 8:
Modules and Object-based Programming

162: POINTER Components of Derived Types

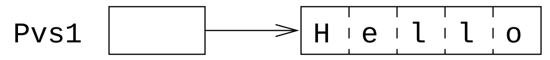
- ALLOCATABLE arrays cannot be used as components in a derived type,
- POINTERs can be,

Dynamically sized structures can be created and manipulated, for example,

```
TYPE VSTRING
 CHARACTER, DIMENSION(:), POINTER :: chars
END TYPE VSTRING
```

this has a component which is a pointer to a 1-D array of characters.

```
TYPE(VSTRING) :: Pvs1
 ...
ALLOCATE(Pvs1%chars(5))
Pvs1%chars = (/"H","e","l","l","o"/)
```



Guide for slide: 162

- complex data structures can be defined by including pointer variables as components of derived types.
- pointer variables can point to intrinsic types or derived types.
- using pointers in derived types is the only way to create derived types with dynamic components (`ALLOCATABLE` arrays are not allowed).
- The example sets up a data type for storing strings of different lengths,.
- in the example, the assignment operator is overloaded for objects of type `VSTRING` — the overloading also implements the allocation of the array to hold the characters.
- a derived type may have any number of pointer components.
- I/O restriction — The normal component by component expansion followed by the dereferencing rules cannot be applied, since it would not, in general, be possible for this process to terminate.
- a composite derived type object with pointer components cannot be output — for example, in the example,

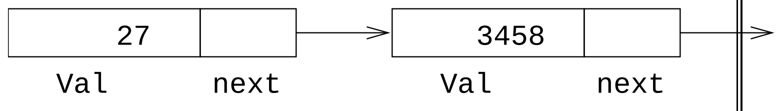
```
TYPE(VSTRING) :: nom
 ...
print*, nom                    ! Illegal
print*, nom%chars              ! fine
```

  the composite object `nom` cannot be output, whereas `nom%chars` is a pointer to a `CHARACTER` array so can be written.

163: Pointers and Recursive Data Structures

- Derived types which include pointer components provide support for recursive data structures such as linked lists.

```
TYPE CELL
 INTEGER :: val
 TYPE (CELL), POINTER :: next
END TYPE CELL
```



- Assignment between structures containing pointer components is subtlely different from normal,

```
TYPE(CELL) :: A
TYPE(CELL), TARGET :: B
A = B
```

is equivalent to:

```
A%val = B%val
A%next => B%next
```

Guide for slide: 163

- derived types can contain pointers to any other type including the type being currently defined. In the example `CELL` contains a pointer to another object of type `CELL`.
- can build up recursive data structures such as singly or doubly linked lists or n-ary trees.
- these structures are very useful owing to their dynamic size.
- cannot point to a type which has yet to be defined, the target type must be defined or be being defined.
- the 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 `NULLIFY`). So long as something is pointing to the head of the structure the whole of the list can be traversed.

Other points

- `=` is defined for assignment between two objects of the same derived type,
- `=` is intrinsically defined to "behave sensibly" when assigning between structures containing pointer components, it does what the user would want it to do
- the meaning of `=` between to objects of type `CELL` could, if desired, be redefined by the program (see later).

164: Practical Example of Linked Lists

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
```

Guide for slide: 164

Note: the example use a form of the read statement which is not covered until musch later.

A line-by-line dissection of the example,

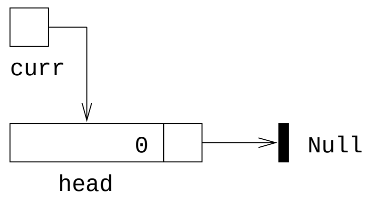
- `TYPE (CELL), TARGET :: head` — this is a cell which anchors the list so long as something points to this the list can always be found and traversed.
- `TYPE (CELL), POINTER :: curr, temp` — two cells, `curr` points to the current position in the list (this will tell us where the next list cell is 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; this puts it in a state from which it can be used,
- `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 a value,
- `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 to make it ready for general use.
- `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.

165: Example (Continued)

The statements,

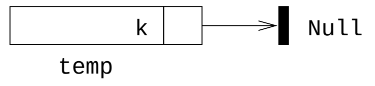
```
head%val = 0; NULLIFY(head%next); curr => head
```

give,



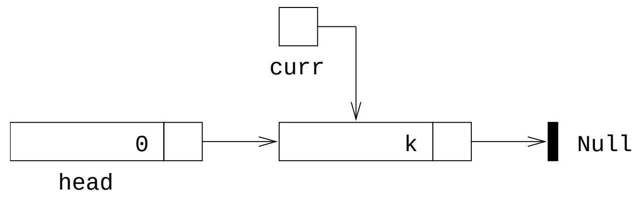

```
ALLOCATE(temp); temp%val = k; NULLIFY(temp%next)
```

give,

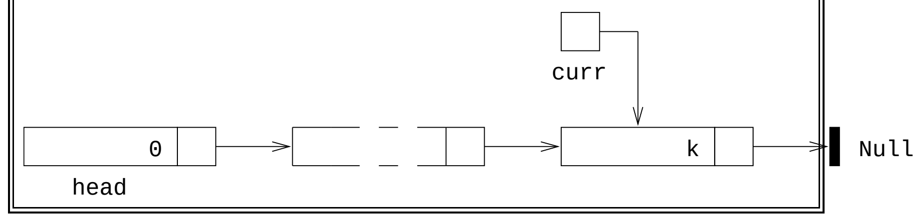



```
curr%next => temp; curr => temp
```

give,



The final list structure is,



Guide for slide: 165

- The first set of statements take a `TARGET` cell `head` (the head of the list) and initialise it. In this case 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.
- 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`).
- 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.
- When the list is completed it will have a structure similar to that given in the forth diagram. The list can be traversed by starting at the `head` cell and by following the pointer links.

166: Example (Continued)

A "walk-through' the previous linked list could be written as follows:

```
curr => head
DO
 PRINT*, curr%val
 IF(.NOT.ASSOCIATED(curr%next)) EXIT
 curr => curr%next
ENDDO
```

All sorts of multiply linked lists can be created and manipulated in analogous ways.

Guide for slide: 166

The previous slide showed how to set up a linked list, here we show how to navigate such a list.

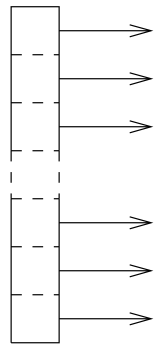
- `curr => head` — The variable `curr` is used as a place-marker. Since we do not need to know where the end of the list is there is no need to keep `curr` pointing at it. (We can always find the end by starting at the head and traversing the list until we reach a null pointer.) Thus `curr` is being reused.
- `DO ... ENDDO` — the list traversal is performed with this loop. The criterion for exiting is that the pointer component of the current cell is disassociated (null).
- `PRINT*, curr%val` prints 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 tough 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.)

167: Arrays of Pointers

It is possible to create what are in effect arrays of pointers:

```
TYPE iPTR
 INTEGER, POINTER :: compon
END TYPE iPTR
TYPE(iPTR), DIMENSION(100) :: ints
```

Visualisation,



ints

Cannot refer to a whole array of pointers,

```
ints(10)%compon ! valid
ints(:)%compon  ! not valid
```

If desired `ints` could have been `ALLOCATABLE`.

Guide for slide: 167

- the example declares a 100 element array of type `iPTR`, which is a 100 element array of pointers to integers.
- note the difference between the slide and

```
INTEGER, POINTER :: arr(:)
```

  which is a pointer to an integer array,
- When the left part of `%` is array valued the component must not be a pointer. Dereferencing can only be performed on a scalar pointer.
- an array of pointers to arrays can have different sized arrays for each individual target, the only restriction is that they must all be of the same rank. The same is true if the target storage is allocated from the heap storage

168: Modules — Object Based Programming

We can write a module that allows a derived type to behave in the same way as an intrinsic type. The module can contain:

- the type definitions,
- constructors,
- overloaded intrinsics,
- overload set of operators,
- other related procedures

An example of such a module is the varying string module which is to be an ancillary standard.

Guide for slide: 168

- 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,
- recall that derived type definitions must be placed in a module so that their structure is visible by use association in non-`MODULE` program units.
- this means that the module containing the type definition must be `USE`d in all procedures that required the type definition. To save messing about we may as well make these procedures into `MODULE PROCEDURE`s where the type definition would already be visible!
- the interfaces for generic procedures, user defined operators, operator overloads and assignment overloads must all be explicit which is another good reason for placing them in a module,
- following the same logic it makes excellent sense to put overloaded intrinsic procedures and intrinsic operators in this module too. The module, when `USE`d, would provide an almost transparent *semantic extension* to Fortran 90.
- 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 is the basis of object oriented programming,
- a module to implement varying length strings has been written at Liverpool. All the relevant intrinsics, e.g., `LEN`, `INDEX`, and operators, e.g., `//`, `LGE`, `=`, have been defined for this new type. Objects of type `VARYING_STRING` can be used (almost) transparently with objects of the same and intrinsic type.

169: Derived Type Constructors

Derived types have in-built constructors, however, it is better to write a specific routine instead.

Purpose written constructors can support default values and will not change if the internal structure of the type is modified. It is also possible to hide the internal details of the type:

```
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.

Guide for slide: 169

In the example, a value can be assigned by use of a user-written constructor but cannot be assigned by the intrinsic constructor `Coords_3D` or on a component by component basis. This approach allows greater flexibility. This constructor can be called like a proper function, that is, with optional and keyword arguments.

Note: that the accessibility statements only apply outside the module, everything is visible within the module.

170: Generic Interfaces

Most intrinsics are generic in that their type is determined by their argument(s). For example, the generic function `ABS(X)` comprises the specific functions:

- `CABS` — called when `X` is `COMPLEX`,
- `ABS` — called when `X` is `REAL`,
- `IABS` — called when `X` is `INTEGER`,

These specific functions are called the *overload set*.
A user may define his own overload set in an `INTERFACE` block:

```
INTERFACE CLEAR
 MODULE PROCEDUE clear_int
 MODULE PROCEDUE clear_real
END INTERFACE ! CLEAR
```

The *generic name*, `CLEAR`, is associated with *specific* names `clear_int` and `clear_real` (the overload set).

Guide for slide: 170

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 would have a general name and would contain a list of specific procedures with similar functionality which are implemented for all data types in a program. The generic procedure name can be used and the compiler is able to tell which specific procedure this corresponds to by looking the types of the actual arguments.

- the method which the generic function name, `ABS`, uses to arrive at its answer is dependent upon the type of the argument(s). It is convenient to have a general (generic) name to describe to the compiler the overall functionality required and let the compiler go away, study the type of the argument and decide which specific method it should use to arrive at the answer.
- intrinsic functions have generic names, the user has a choice of calling the function by the specific name (e.g., `DABS`, `CABS`) or by the generic name `ABS`— the end effect will be the same. (If the procedure is an actual argument then the specific name *must* be used.)
- the user can define his / her own generic names for user procedures. This is done by a generic interface specification. The user may then call the procedure with a generic name and the compiler will determine the type and number of arguments and then match these up with a unique (specific) interface from the generic interface specifications. If no such procedure exists then an error is generated.
- Specific interfaces are related to procedure calls by comparing the *number, type, rank* and *kind* of the non-optional arguments.

  Note that if a procedure is defined for a default intrinsic type then it is also defined for one (unknown) parameterised version of that type. Which parameterised type corresponds to the intrinsic type will vary from compiler to compiler so the resulting program will not be portable. Once parametrised types have been used in generic interfaces all intrinsic types should be converted to parameterised types. This will be explained later.
- the 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.
- `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`.
- cannot have the generic interface name on the `END INTERFACE` statement. This is a language anomaly and is due to be corrected in the next revision of the standard. It is likely that many compilers will accept code with the name present.
- using generic procedure names makes programming much easier as the user does not have to remember the specific name of a procedure.
- clearly 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 them by `USE` association.

171: Generic Interfaces - Example

The full module would be

```
MODULE Schmodule
 IMPLICIT NONE
 INTERFACE CLEAR
  MODULE PROCEDURE clear_int
  MODULE PROCEDURE clear_real
 END INTERFACE CLEAR
CONTAINS
 SUBROUTINE clear_int(a)
  INTEGER, DIMENSION(:), INTENT(INOUT) :: a
   ... ! code to do clearing
 END SUBROUTINE clear_int
 SUBROUTINE clear_real(a)
  REAL, DIMENSION(:), INTENT(INOUT) :: a
   ... ! code to do clearing
 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
```

The first procedure invocation would be resolved with `clear_real` and the second with `clear_int`.

Guide for slide: 171

- `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`.
- the same logic is followed for `counts`, this is integer valued so will correspond to `clear_int`.
- both module procedures are unique with respect to their arguments.
- the module procedures themselves are able to have internal procedures.

172: Generic Interfaces - Commentry

In order for the compiler to be able to resolve the reference, both module procedures must be unique:

- the specific procedure to be used is determined by the *number*, *type*, *kind* or *rank* of the non-optional arguments,
- the overload set of procedures must be unambiguous with respect to their dummy arguments,
- default intrinsic types *should not* be used in generic interfaces, use parameterised types.

Basically, by examining the argument(s), the compiler calculates which specific procedure to invoke.

Guide for slide: 172

- modules are perfect for this form of encapsulation - all the appropriate procedures can be kept in one re-usable module. `USE`ing this module extends the functionality of the language.
- the module procedure statements should be used to save time and to remove the possibility of a mismatch between two interface definitions of the same procedure.
- the overload set must be unambiguous, i.e., the procedures must all be unique in at least one of: type, kind or rank in their non-optional arguments.

173: Derived Type Output Procedures

Derived types with PRIVATE components need special procedures for I/O:

```
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(Coord)
END MODULE ThreeDee
```

Define an INTERFACE called Print and overload to accept arguments of type Coords_3D

Guide for slide: 173

Again this approach allows greater flexibility. Output can be formatted in a standard way.

Types with private components cannot be used in I/O statements, again, special purpose I/O routines must be placed in the module for this purpose.

174: Overloading Intrinsic Procedures

When a new type is added, it is a simple process to add a new overload to any relevant intrinsic procedures.
The following extends the LEN_TRIM intrinsic to return the number of letters in the owners name for 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_trim = 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.

Guide for slide: 174

- to overload intrinsic functions a generic interface should be defined with the same name as the existing generic intrinsic name. Any procedures included in this interface are *added* to the overload set.
- `owner_len_trim` could also be called by its specific or generic names (`owner_len_trim` or `LEN_TRIM`).
- existing intrinsics may 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.
- 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 the same ways as `CHARACTER` objects, i.e., transparently.
- the body of the procedures should be pure in the sense that they should not alter any global data or produce output

175: Overloading Operators

Intrinsic operators, such as `-`, `=` and `*`, can be overloaded to apply to all types in a program:

- specify the generic operator symbol in an `INTERFACE OPERATOR` statement,
- specify the overload set in a generic interface,
- declare the `MODULE PROCEDURE`s (`FUNCTION`s) which define how the operations are implemented.

These functions must have one or two non-optional arguments with `INTENT(IN)` which correspond to monadic and dyadic operators.
Overloads are resolved as normal.

Guide for slide: 175

- operator overloading should be implemented in a module so that all the functionality is encapsulated.
- the generic interface specification is initiated by an `INTERFACE OPERATOR` statement with the symbol which represents the operator being included in parentheses.
- the specification process is implemented by a `FUNCTION` which must have one or two arguments. Arguments must be `INTENT(IN)`.
- for a monadic operator the function only has one argument corresponding to the single operand of the operator.
- 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 a new type, say, a representation of a rational number, there would have to be two functions defined, one for the integer-rational combination and one for for the rational-integer combination.
- the body of the function should be pure in the sense that it should not alter any arguments, global data or produce output.

176: Operator Overloading Example

The '*' operator can be extended to apply to the rational number data type as follows:

```
MODULE rational_arithmetic
 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
    ...
     rat_rat = ...
  FUNCTION int_rat(l,r)      ! int * rat
   INTEGER, INTENT(IN)      :: l
   TYPE(RATNUM), INTENT(IN) :: r
   ...
  FUNCTION rat_int(l,r)      ! rat * int
   TYPE(RATNUM), INTENT(IN) :: l
   INTEGER, INTENT(IN)      :: r
   ...
END MODULE rational_arithmetic
```

The three new procedures are added to the operator overload set allowing them to be used as operators in a normal arithmetic expressions.

Guide for slide: 176

- the operator is specified in an `INTERFACE OPERATOR` statement and the operator is in brackets,
- 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) must be defined in procedures.
- the bodies of the procedures define what is meant by multiplication of these two types.
- it is not possible to modify the operator precedence, when an operator is overloaded it retains its place in the pecking order of operators.

177: Example (Cont'd)

With,

```
    USE rational_arithmetic
    TYPE (RATNUM) :: ra, rb, rc
```

we could write,

```
    rc = rat_rat(int_rat(2,ra),rb)
```

but better:

```
    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
    ....
```

Guide for slide: 177

- the functionality would exist in Fortran 90 even if it were not permitted to overload operators, however, it would have to be implemented using procedures and the semantics are far clearer with overloaded operators.
- can overload all operators given in the table in one of the earlier lectures, i.e., all the intrinsic operators.
- separate procedures must be given for monadic and dyadic `+` and `-`.

178: Defining New Operators

can define new monadic and dyadic operators. They have the form,

.< *name* >.

Note:

- monadic operators have precedence over dyadic.
- names must be 31 letters (no numbers or underscore) or less.
- basic rules same as for overloading procedures.

Guide for slide: 178

- any name (31 letters or less, no numbers, no underscore) can be chosen including intrinsic operator names.
- the functions which define the actions that an operator implies have one or two operands depending on whether the operator is monadic or dyadic. Operands must have `INTENT(IN)`.
- operator names can be the same as object names with no conflict,
- the body of the procedure should be pure in the sense that it should not alter any arguments, global data or produce output.
- a defined monadic has the highest precedence.
- a defined dyadic has the lowest precedence.

179: Defined Operator Example

For example, consider the following 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 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
```

Guide for slide: 179

- The example defines a MODULE that contains the definition of an operator .TWIDDLE. in both monadic or dyadic form. Note the INTENT attribute which is mandatory and also note that the END INTERFACE statement cannot contain the operator name.
- The operator is defined in much the same way as for the extension of intrinsic operators,
- note how the precedence affects the results of .TWIDDLE. (2 .TWIDDLE. 5) and .TWIDDLE. 2 .TWIDDLE. 5. The parentheses modify the execution order.

180: Precedence

- user defined monadic operators are most tightly binding.
- user defined dyadic operators are least tightly binding.

For example,

```
.TWIDDLE.e**j/a.TWIDDLE.b+c.AND.d
```

is equivalent to

```
(((.TWIDDLE.e)**j)/a).TWIDDLE.((b+c).AND.d)
```

Guide for slide: 180

- the precedences were given earlier and cannot be modified.
- monadic are the most tightly binding operators of all operators,
- dyadic are the least tightly binding operators of all operators,
- the example demonstrates how the monadic `.TWIDDLE.` has the highest precedence and is combined withits operand first and the dyadic `.TWIDDLE.` has the lowest precedence and is combined with its operands last.

181: User-defined Assignment

Assignment between two different user defined types must be explicitly programmed; a `SUBROUTINE` with two arguments specifies what to do,

- the first argument is the result variable and must have `INTENT(OUT)`;
- the second is the expression whose value is converted and must have `INTENT(IN)`.

Overloading the assignment operator differs from other operators:

- assignment overload sets do **not** have to produce an unambiguous set of overloads;
- later overloads override earlier ones if there is an ambiguity;

Guide for slide: 181

- assignment overloading should be implemented in a module so that all the functionality is encapsulated.
- the generic interface for assignment is specified by an `INTERFACE ASSIGNMENT` block.
- the assignment process is specified by a `SUBROUTINE` which must always have two arguments. The first argument is result variable (the LHS) and must have `INTENT(OUT)`. The second argument is the RHS expression and must have `INTENT(IN)`.
- overloads are resolved in the normal way.
- unlike operator overloads, assignment overloads do not have to be unique, the one that appears last in the code is the one that is used. It is perfectly legal for there to be more than one accessible generic interface block with the same generic identifier. In these situations the effective generic set is the sum of the individual ones which, except for assignment, must remain unambiguous.
- the body of the subroutine should be pure in the sense that it should not alter any arguments, global data or produce output.

182: Defined Assignment Example

Should put in a module,

```
INTERFACE ASSIGNMENT(=)
 MODULE PROCEDURE rat_ass_int, real_ass_rat
END INTERFACE
PRIVATE :: rat_ass_int, real_ass_rat
```

specify SUBROUTINEs in the CONTAINS block:

```
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 = REAL(exp%num) / REAL(exp%den)
END SUBROUTINE real_ass_rat
```

Wherever the module is used the following is valid:

```
ra = 50
x = rb*rc
```

for real x.

Guide for slide: 182

- For the rational numbers example, defining the mathematical operators is of little use if the assignment operator is not defined. We need to say what happens if a rational number is assigned to an `INTEGER` or a `REAL` and vice versa. We need to state how the types are coerced. (This coercion is intrinsically defined for the intrinsic types - we need to extend the coercion for user-defined types.)
- the < `INTEGER` > `=` < `INTEGER` > and < `RATNUM` > `=` < `RATNUM` > cases are intrinsically defined, we need to extend (overload) the `=` operator to cater for
  - < `REAL` > `=` < `RATNUM` >
  - < `RATNUM` > `=` < `INTEGER` >

  plus any other combinations that are required.
- the generic assignment interface will contain two module procedures — one for each of the above two cases.
- in the subroutine headers, `var` is the LHS and `exp` is the RHS expression, this notation is useful.
- in all cases the first argument is attributed with `INTENT(OUT)` and the second `INTENT(IN)`.
- the body of the procedure *must* contain an assignment to the first argument.
- the first assignment statement uses `rat_ass_int` and the second `real_ass_rat`.

183: Semantic Extension Example

Collect stuff from previous slides:

```
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 (*)
  PRIVATE rat_rat, int_rat, rat_int
   ... ! and so on
```

The type `INTERNAL` is only accessible from within the module.
Should also add:

- constructors for `RATNUM`: `init_RATNUM`
- output procedure for `RATNUM`: `Print`
- overloaded intrinsics: `REAL`, `CEILING`, `FLOOR`, etc

Guide for slide: 183

- accessibility is very useful when applied to other module entities such as operators and procedures,
- it is a particularly good idea to deny access to the module procedures which define what overloading certain operators 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. These should be made private.
- the accessibility statements for procedures must come *after* the `MODULE PROCEDURE` statements.
- it is also possible to create a type which is not visible at all outside the module, such a type would be used internally in the module to facilitate the task in hand. An example is `INTERNAL` given above.

184: Semantic Extension Modules

The real power of the `MODULE` / `USE` facilities appears when coupled with derived types and operator and procedure overloading to provide *semantic extensions* to the language.
Semantic extension modules require:

- ☐ a mechanism for defining new types;
- ☐ a method for defining operations on those types;
- ☐ a method of overloading the operations so user can use them in a natural way;
- ☐ a way of encapsulating all these features in such a way that the user can access them as a combined set;
- ☐ details of underlying data representation in the implementation of the associated operations to be kept hidden (desirable).

This is an Object Oriented approach.

Guide for slide: 184

- semantic extension is so named because by the use of modules it can be made to look like the language contains far more functionality than it actually does. The language is extended.
- Semantic extension allows the user to express a problem in terms of the natural objects and operations relating to that problem. This is so-called Object Oriented Programming.
- Fortran 90 provides many features desirable for OOP but not all of them.
- as well as allowing the language to be extended to more closely relate to real-world objects. Fortran 90 has introduced many safety features to make sure this added functionality cannot be abused, features such as data hiding and argument intents make for improved safety,
- as mentioned previously the `VARYING_STRING` module has been written at Liverpool and this has provided (to all intents and purposes) a new data type which can be used safely in all programs which `USE` the appropriate module. (`VARYING_STRING` module has 96 module procedures which are used to extend 25 generic procedures, 6 intrinsic relational operators and the concatenation operator. Most entities are `PRIVATE` except the type and intrinsic entities such as the extended operators and functions.)

Lecture 9:
Parametrised Intrinsic Types Plus

186: Complex Data

This data type has the same precision as default `REAL`.

- `COMPLEX` object declaration:

```
COMPLEX :: z, j, za(1:100)
```

- symbolic constants are expressed in the same way:

```
COMPLEX, PARAMETER :: i = (0.0,1.0)
```

- complex constants are represented as co-ordinate pairs: `(1.0,1.0)` or `(3.141,1.0E-9)`.
- complex values can be constructed using the `CMPLX` function,

```
z = CMPLX(x,y)
```

- 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)`.

Guide for slide: 186

- complex objects are declared and have exactly the same properties as other intrinsic data types.
- a `COMPLEX` variable is made up from two default real cells.
- cannot use the `CMPLX` intrinsic in initialisation expression.
- can use real-valued literals and symbolic constants, and complex valued literals and symbolic constants a `COMPLEX` object initialisation statement (`PARAMETER` statement etc.). It is not permissible to allow a constructed value containing real-valued symbolic constants as components. For example,

  ```
  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.
- when an non-complex number is assigned to a complex number the imaginary part of the complex number becomes zero (as yout would expect). In other words when a non-complex value is coerced a `0` is placed in the imaginary part of the coordinate pair.
- a complex coordinate pair could contain integers, e.g., (1,1), these would be promoted to `REAL` values before assignment to the `COMPLEX` variable.
- the `CMPLX` intrinsic type conversion routine is not necessary for literal values (it can be used though), however, it is essential for symbolic constants.
- when a mixed type expression contains a `COMPLEX` variable the other variables are promoted to this type before the expression is evaluated.
- extended precision `COMPLEX` is achieved by parameterisation (`KIND`) of the intrinsic `COMPLEX` type, (see later).

187: Complex Intrinsic Functions

`AIMAG(z)`	Imaginary part of a complex number
`REAL(z)`	Real part of a complex number
`CONJG(z)`	Conjugate of a complex number
`ABS(z)`	Absolute value of a complex number
`SIN(z)` etc	All appropriate mathematical functions

Guide for slide: 187

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.

188: Parameterised Data Types

- FORTRAN 77 had a problem with numeric portability, the precision (and exponent range) between processors could differ,
- Fortran 90 implements a portable precision selecting mechanism,
- intrinsic types can be parameterised by a kind value (an integer). For example,

```
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 but procedure arguments must match in type **and** kind.

Guide for slide: 188

- The precision on one machine is not necessarily the same as on another.
- A default `REAL` may be able to support numbers up to (say) 10^{68} on one machine and upto 10^{136} on another.
- Fortran 90 has introduced a mechanism to allow the user to specify the required precision of a data type (so long as that precision exists on the processor). This will allow the hardware to be used in a suitable manner.
- a compiler must support at least one representation for each intrinsic default data type.
- A processor can support different precisions (representations) of `INTEGER`, `REAL` and `COMPLEX`. Different character sets (for `CHARACTER` s) and different ways of representing `LOGICAL` objects.
- different `CHARACTER` kinds can correspond to different character sets, e.g., arabic, musical notation, cryllic.
- `DOUBLE PRECISION` does not have different precisions and its use is not recommended — use a parameterised `REAL` type instead.
- representations will be listed in the compiler manual.
- different representations are denoted by an integer.
- the `KIND=` is optional.
- recall that generic overloads can be resolved by the kind of an actual argument indeed actual and dummy arguments *must* match in kind value.

189: Integer Data Type by Kind

- selecting kind, by an explicit integer is still **not** portable,
- must use the SELECTED_INT_KIND intrinsic function. For example, SELECTED_INT_KIND(2) returns a kind number capable of expressing numbers in the range, $(-10^2, 10^2)$.
- here 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
```

Guide for slide: 189

- the explicit integer will correspond to different precisions on different machines — we need a mechanism to select a kind value on the basis of the desired precision. This will be portable.
- use `SELECTED_INT_KIND` intrinsic,
- For `INTEGER` types the argument to `SELECTED_INT_KIND` specifies the exponent range of the representation or *model* in Fortran 90 terminology,
- for `REAL` or `COMPLEX` the arguments to `SELECTED_REAL_KIND` correspond to
  - the number of significant digits
  - the exponent range
- no equivalent intrinsic for `LOGICAL` type.
- kind parameters could be set up in a `MODULE`.
- the example specifies that precision should be at least
  - $(-10^2, 10^2)$
  - $(-10^4, 10^4)$
  - $(-10^{10}, 10^{10})$
  - $(-10^{100}, 10^{100})$

  if any representation is not available then `SELECTED_INT_KIND` returns `-1` and any declaration using this error value will give an error (at compile time).
- `KIND(0)` (the intrinsic function) gives the default kind,
- the compiler manual will explain the different kinds available on a particular installation.

190: 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.

Guide for slide: 190

- coercion and promotion exists within mixed kind expressions, however, a literal of a particular kind can be specified using the underscore notation.
- use a name not a number — enhances portability,
- any initialisation expression could be used to specify kind.
- be careful because `-` is valid in most situations where `_` is valid.
- the model corresponding to `short` only supports numbers for -100 to 100.

191: Real KIND Selection

Similar principle to INTEGER:

- SELECTED_REAL_KIND(8,9) will support numbers with a precision of 8 digits and decimal exponent range from $(-9, 9)$. 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.

Guide for slide: 191

- exactly the same pitfalls as for integer,
- `COMPLEX` is made from two `REAL` numbers so precision of complex numbers is specified in exactly the same way as for reals

192: Kind Functions

- it is often useful to be able to interrogate an object to see what kind parameter it has.
- `KIND` returns the integer which corresponds to the kind of the argument.
- for example, `KIND(a)` will return the integer parameter which corresponds to the kind of a. `KIND(20)` returns the kind value of the default integer type.
- the intrinsic type conversion functions have an optional argument to specify the kind of the result, for example,

```
print*, INT(1.0,KIND=3), NINT(1.0,KIND=3)
x = x + REAL(j,KIND(x))
```

Guide for slide: 192

- `KIND(a)` returns the kind value of `a`.
- `KIND(20)` returns the kind value of the constant `20`.
- `print*, INT(1.0,KIND=3)` returns integer 1 in the representation of `KIND=3`.
- `x = x + REAL(j,KIND(x))` converts `j` to the representation used for `x` and then evaluates the expression.
- `INT`, `CMPLX`, `AINT`, `REAL`, `ACHAR`, `IACHAR` etc., all have an optional kind argument.

193: Mixed Kind Expression Evaluation

Mixed kind expressions:

- If all operands of an expression have the same type and kind, then the result also has this type and kind.
- If the kinds are different, then operands with lower range are promoted before operations are performed. For example, if

  ```
  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.

Guide for slide: 193

- the rules for normal type coercion still hold, integers are promoted to reals, reals to complex, etc.

194: Logical KIND Selection

- There is no `SELECTED_LOGICAL_KIND` intrinsic, however, the `KIND` intrinsic can be used as normal.

  For example,

```
LOGICAL(KIND=4) :: yorn = .TRUE._4
LOGICAL(KIND=1), DIMENSION(10) :: mask
IF (yorn .EQ. LOGICAL(mask(1),KIND(yorn)))...
```

- `KIND=1` may only use one byte of store per variable,

LOGICAL(KIND=1)	1 byte
LOGICAL(KIND=4)	4 bytes

- Must refer to the compiler manual.

Guide for slide: 194

- exactly the same principle as for numeric data types,
- LOGICAL intrinsic converts between different kind representations.
- no need for SELECTED_LOGICAL_KIND as only 2 states can be represented: (.TRUE. and .FALSE.).
- different LOGICAL kinds will differ in the number of bytes / bits used to store one value.

195: 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
CHARACTER(KIND=2,LEN=25) :: mohammed
```

- 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)
```

  Note `CHAR` gives the character in the given position in the collating sequence.

- Literals can also be specified:

  `greek_"`$\alpha\delta\alpha\mu$`"`

  Notice how the kind is specified first.

Guide for slide: 195

- must use `ICHAR` and `CHAR` when converting between one character set and another - can only do one character at a time.
- compilers only have to support one character set.

196: Kinds and Procedure Arguments

Dummy and actual arguments must match exactly in kind, type and rank, consider,

```
SUBROUTINE subbie(a,b,c)
 USE kind_defs
 REAL(r2), INTENT(IN)  :: a, c
 REAL(r1), INTENT(OUT) :: b
  ...
```

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 every compiler. This is very important with generics.

Guide for slide: 196

- Generic overloads are resolved by kind, type and rank,
- cannot automatically coerce between different kinds across a subroutine boundary.
- 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 cleanly 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!

197: 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 unsigned integer, for example,

```
s-1      3  2  1  0
 0  ..   0  0  0  0    value = 0

s-1      3  2  1  0
 0  ..   0  1  0  1    value = 5

s-1      3  2  1  0
 0       0  0  1  1    value = 3
```

The number of bits in a single variable depends on the compiler

Guide for slide: 197

All variables used as bit arguments must be INTEGER valued. The model for bit representation is that of an unsigned 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 number of bits in a single variable depends on the compiler - parameterised integers should be used to fix the number of bytes. BIT_SIZE gives the number of bits in a variable.

Assume that A has the value 5 (00...000101) and B the value 3 (00...000011) in the following:

- BTEST — bit value.

  .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 bit move 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, this gives B the value 00...000010 (2).

198: Array Construction Intrinsics

There are four intrinsics in this class:

- MERGE(TSOURCE,FSOURCE,MASK)— merge two arrays under a mask,
- SPREAD(SOURCE,DIM,NCOPIES)— replicates an array by adding NCOPIES of a dimension,
- PACK(SOURCE,MASK[,VECTOR])— pack array into a one-dimensional array under a mask.
- UNPACK(VECTOR,MASK,FIELD)— unpack a vector into an array under a mask.

Guide for slide: 198

- MERGE(TSOURCE,FSOURCE,MASK)— merge 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)— 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.
- PACK(SOURCE,MASK[,VECTOR])— pack array into a one-dimensional array under a mask, useful for compressing data.
- UNPACK(VECTOR,MASK,FIELD)— unpack a vector into an array under a mask, useful for uncompressing data.

199: MERGE Intrinsic

`MERGE(TSOURCE,FSOURCE,MASK)`— merge 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.`.
If,

$$\texttt{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

and

$$\texttt{TSOURCE} = \begin{pmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{pmatrix}$$

and

$$\texttt{FSOURCE} = \begin{pmatrix} 0 & 4 & 8 \\ 2 & 6 & 10 \end{pmatrix}$$

we find

$$\texttt{MERGE(TSOURCE, FSOURCE, MASK)} = \begin{pmatrix} \boxed{1} & \boxed{5} & 8 \\ 2 & 6 & \boxed{11} \end{pmatrix}$$

Guide for slide: 199

- `RESHAPE` puts the elements into the LHS arrays in array element order, so

```
TSOURCE = 1 5 9
          3 7 11
```

  and

```
FSOURCE = 0 4 8
          2 6 10
```

- T and F defined as:

```
LOGICAL, PARAMETER :: T = .TRUE.
LOGICAL, PARAMETER :: F = .FALSE.
```

  so

```
MASK = T T F
       F F T
```

  thus we get from `TSOURCE`

```
1 5 .
. . 11
```

  and from `FSOURCE`

```
. . 8
2 6 .
```

200: SPREAD Intrinsic

`SPREAD(SOURCE,DIM,NCOPIES)`— replicates an array by adding `NCOPIES` of in the direction of a stated dimension.
If `A` is `(/5, 7/)`, then

$$\mathtt{SPREAD}(\mathtt{A},2,4) = \begin{pmatrix} 5 & 5 & 5 & 5 \\ 7 & 7 & 7 & 7 \end{pmatrix}$$

and

$$\mathtt{SPREAD}(\mathtt{A},1,4) = \begin{pmatrix} 5 & 7 \\ 5 & 7 \\ 5 & 7 \\ 5 & 7 \end{pmatrix}$$

Guide for slide: 200

- `DIM=` 2 (the first example) — this means we spread `(/ 5, 7/)` along dimension 2 (the direction of a row).
- `DIM=` 1 (the second example) — this means we spread `(/ 5, 7/)` along dimension 1 (the direction of a column).
- `NCOPIES=` 4 — which means we want 4 copies of the specified section,

201: PACK Intrinsic

`PACK(SOURCE,MASK[,VECTOR])` — pack 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`.
If

$$\texttt{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

and

$$\texttt{A} = \begin{pmatrix} \boxed{1} & \boxed{5} & 9 \\ 3 & 7 & \boxed{11} \end{pmatrix}$$

then

- `PACK(A,MASK)` is `(/1, 5, 11/)`;
- `PACK(A,MASK,(/3,4,5,6/))` is `(/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/)`.

Guide for slide: 201

- If `VECTOR` is present, the result size is that of `VECTOR`; otherwise, the result size is the number t of `.TRUE.` elements in `MASK` unless `MASK` is scalar with the value `.TRUE.` in which case the result size is the size of `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 $i = 1, ..., t$.
- If `VECTOR` is present and has size $n > t$, element i of the result has the value `VECTOR(i)`, for $i = t + 1, ..., n$.
- so for the first example, `VECTOR` is absent meaning that size of the result is the number of `.TRUE.` elements in `MASK`, i.e., 3. (This is with the previous `MASK`.)

  The elements of `MASK` that are true are `(1,1), (1,2), (2,3)`

  ```
  1  5  .
  .  . 11
  ```

  giving the result `(/1, 5, 11/)`.
- the second example, `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/)`.
- the third example, the mask is scalar so the first 6 elements are `A` and the rest from the end of `VECTOR`.

202: UNPACK Intrinsic

UNPACK(VECTOR,MASK,FIELD)— unpack 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

$$\texttt{UNPACK}((/6,5,4/),\texttt{MASK},\texttt{FIELD}) = \begin{pmatrix} \boxed{6} & \boxed{5} & 1 \\ 7 & 7 & \boxed{4} \end{pmatrix}$$

and

$$\texttt{UNPACK}((/3,2,1/),\texttt{.NOT.MASK},\texttt{FIELD}) = \begin{pmatrix} 9 & 5 & \boxed{1} \\ \boxed{3} & \boxed{2} & 3 \end{pmatrix}$$

Guide for slide: 202

- The element of the result that correspond to the ith `.TRUE.` element of the mask (in array element order) has the value `VECTOR` for $i = 1, .., t$ where t is the number of `.TRUE.` elements of the mask, each other element has a value equal to the corresponding element of `FIELD`
- for the first example, the portions of the result originating from `VECTOR` and corresponding to the `.TRUE.` values of `MASK` are

```
6  5  .
.  .  4
```

  and the rest of this is filled in by elements of `FIELD`

```
.  .  1
7  7  .
```

- for the second example, the portions of the result corresponding to the `.TRUE.` values of `MASK` are

```
.  .  1
3  2  .
```

  and the rest of this is filled in by elements of field

```
9  5  .
.  .  3
```

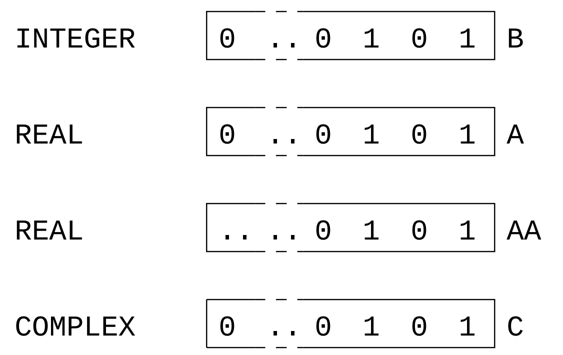
203: TRANSFER Intrinsic

TRANSFER converts (not coerces) physical representation between data types; it is a retyping facility. Syntax:

TRANSFER(SOURCE,MOLD)

- ☐ SOURCE is the object to be retyped,
- ☐ MOLD is an object of the target type.

```
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) /))
 ...
```



Guide for slide: 203

Most languages have a facility to change the type of an area of storage, in Fortran 77 people used `EQUIVALENCE`. Fortran 90 adds different facility.

`TRANSFER` takes the bit pattern of the underlying representation and interprets it as a different type.

- `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 above 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.

Lecture 10:
Odds and Ends

205: Input / Output

Fortran 90 has a wealth of I/O, too much to cover here.
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. Each unit can have many properties, for example,

- file — the name of the file connected to the unit,
- action — read, write, read and write, etc.
- status — old, new, replace, etc.
- access method — sequential, direct,

Guide for slide: 205

Fortran 90 has statements built in to handle I/O. It has `READ`, `WRITE`, `OPEN`, `CLOSE`, `REWIND` and `BACKSPACE` statements.

It also has very good formatting commands for ‘pretty printing’ or formatted output.

The *logical unit* is the “name” by which the file is referred to in the program it must be positive. The number may often limited to be between 1 and 100.

- file — < *filename* >

  this is represented as a string in the program.

- action — read, write, read and write,

  if the file is opened for one sort of action and another is attempted then an error will arise. For example, cannot write to a file opened 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.

206: OPEN Statement

The syntax is,

```
OPEN([UNIT=]<integer>,&
       FILE=<filename>, ERR=<label>, &
       STATUS=<status>, ACCESS=<method>,&
       ACTION=<mode>, RECL=<int-expr>)
```

where,

- <*filename*> is a string,
- <*status*> is 'OLD', 'NEW', 'REPLACE', 'SCRATCH' or 'UNKNOWN',
- <*method*> is 'DIRECT' or 'SEQUENTIAL'
- <*mode*> is 'READ', 'WRITE' or 'READWRITE'
- if the file is direct access the RECL must be specified,

For example,

```
OPEN(17,FILE='output.dat',ERR=10, &
    STATUS='REPLACE', &
    ACCESS='SEQUENTIAL',ACTION='WRITE')
OPEN(14,FILE='input.dat',ERR=10, &
    STATUS='OLD', RECL=iexp, &
    ACCESS='DIRECT',ACTION='READ')
```

Guide for slide: 206

`OPEN` is used to connect a named file to a logical unit

- there are many default settings many of which are processor dependent. Its good practice not to rely on the defaults but to specify what is required.
- there are other less important specifiers which are not covered here.
- the strings can be delimited by single or double quotes.
- `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 be padded with a specific number of blanks.
- `ERR=`< *label* > specifies a numeric statement 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,
  - `'SEQUENTIAL'` — the file is written / read (sequentially) line by line.
  - `'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.
- `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.
- `RECL=`< *int-expr* > — specifies the maximum length of a record in a file opened for direct access.

The example,

- the first example opens a file for writing and connects it to unit 17. The file is accessed on a line by line basis and already exists but is to be replaced,
- the label must exist,
- the second example opens a file for input only on unit 14. The file is directly accessed and (clearly) already exists.

207: READ Statement

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* > is a number or `*`.
- < *format* > is a string or a `FORMAT` statement label.
- < *int-variable* > holds a return code, zero means no error.
- < *label* > is an executable statement label.
- < *advance-mode* > is `'YES'` or `'NO'`.
- < *num-chars* > is the number of characters read.
- < *int-expr* > is a record number (for direct access).

For example

```
READ(14,FMT='(3(F10.7,1x))',REC=iexp) a,b,c
READ(*,'(A)',ADVANCE='NO',EOR=12,SIZE=nch) str
```

Guide for slide: 207

- the `UNIT=` and `FMT=` specifiers are optional if these 2 fields are first and second in the statement.
- $<$ *integer* $>$ in the `UNIT` specifier is a valid unit number or `*` for the default input or output channel, (the keyboard or the screen).
- $<$ *format* $>$ in the `FMT` specifier is a string of formatting characters, a format statement number or a `*` to signify free format.
- $<$ *int-variable* $>$ in the `IOSTAT` specifier nominates an integer variable to hold a return code, if it is zero there is no error, positive is there is a read 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.
- $<$ *int-expr* $>$ in `REC` specifier specifies which record is to be written in direct access mode.
- 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.
- the first example: the record specified by the integer expression `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`.
- the second example: since non-advancing input 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.

208: WRITE Statement

Can interchange `READ` and `WRITE` 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* > is a number or `*`,
- < *format* > is a string or a `FORMAT` statement label,
- < *int-variable* > holds a return code, zero means no error,
- < *label* > is an executable statement label,
- < *advance-mode* > is `'YES'` or `'NO'`,
- < *int-expr* > is a record number (for direct access).

For example

```
WRITE(17,FMT='(I4)',IOSTAT=istat,ERR=10) ival
WRITE(*,'(A)',ADVANCE='NO') 'Yello'
```

Guide for slide: 208

- the `UNIT=` and `FMT=` specifiers are optional if these 2 fields are first and second in the statement.
- < *integer* > in the `UNIT` specifier is a valid unit number or `*` for the default input or output channel, (the keyboard or the screen).
- < *format* > in the `FMT` specifier is a string of formatting characters, a format statement number or a `*` to signify free format.
- < *int-variable* > in the `IOSTAT` specifier nominates an integer variable to hold a return code, if it is zero there is no error, positive is there is a `WRITE` error.
- < *label* > in `ERR` is a valid label to where control jumps if there is a write error.
- < *advance-mode* > specifies whether each `WRITE` 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.
- < *int-expr* > in `REC` specifier specifies which record is to be written in direct access mode.
- in the first example, `'(I4)'` means `INTEGER` with 4 digits. the label 10 is a statement where control can jump on a write exception. `ival` is an integer variable whose value is written out.
- the second example: since non-advancing input has been specified, the cursor will remain on the same line as the string after writing. Under normal circumstances (default advancing output) the cursor would be positioned at the start of the next line. Note the explicit format descriptor.

209: FORMAT Statement / FMT= Specifier

FMT= can specify either a line number of a FORMAT statement, an actual format string or a *.

Fortran 90 has a rich formatting syntax, only the highlights are given here.

An example,

```
    WRITE(17,FMT=&
     '(2X,2I4,1X,''name '',A7)')i,j,str
    READ(14,*) x,y
    WRITE(*,FMT=10) a,b
 10 FORMAT('vals',2(F15.6,2X))
```

The data is formatted using edit descriptors.

The following is written,

```
    11-195 name Philip
vals      -1.051330    333356.000033
```

Guide for slide: 209

- `WRITE(17,FMT='(2X,2I4,1X,''name '',A7)')i,j,str`

  writes out `i,j,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, (*`A 7`*)"*. The variables are taken from the I/O list at the end of the line.

  2 integers of 4 digits with no gaps, (`2I4`), one space, (`1X`), the string `'name '` and then 7 letters, (`A7`)"

  Note the double single quotes (escaped quote) around name. A single " could have been used instead,

- `READ(14,*) x,y`

  reads `x, y` in default format

- `WRITE(*,FMT=10) a,b,c,d`

  writes out `a,b,c,d` according to `FORMAT` statement `10`

- `10 FORMAT('vals',2(F15.6,2X))`

  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`*)"*.

210: Edit Descriptors

Summary,

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),

For example,

```
WRITE(*,FMT='(2X,2(I4,1X),''name '',&
                A4,F13.5,1X,E13.5)') &
 77778,3,'abcdefghi',14.45,14.5666666
```

gives

```
  ****    3 name abcd     14.45000   0.14567E+02
```

Here 77778 will not fit into 4 spaces

Guide for slide: 210

- w determines the number of the total number of characters (columns) 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.
- `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).

The 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.
- (Ignoring the stars) the output data shown could be read in by a `READ` statement using the same format.
- `2(I4,1X)` is *exactly* the same as `I4,1X,I4,1X`,
- `2I4` is *exactly* the same as `I4,I4`,

211: Other I/O Statements

- CLOSE — unattaches the unit number specified in the statement.
- REWIND — puts the file pointer back to the start of the file,
- BACKSPACE — the file pointer is moved back one record.
- ENDFILE — forces an end-of-file to be written into the file.

The above statements have other specifiers such as IOSTAT,
For example,

```
REWIND (UNIT=14)
BACKSPACE (UNIT=17)
ENDFILE (17)
CLOSE (17, IOSTAT=ival)
```

Guide for slide: 211

- 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 often puts the file pointer back to the start, and then fast forwards,
- ENDFILE the end of the file is marked but the file remains open.

212: 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.

Guide for slide: 212

213: Subroutine Syntax

Syntax of a (non-recursive) subroutine declaration:

```
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
```

SUBROUTINE < *procname* >[(< *dummy args* >)]
< *declaration of dummy args* >
< *declaration of local objects* >
...
< *executable stmts* >
[CONTAINS
< *internal procedure definitions* >]
END [SUBROUTINE [< *procname* >]]

Guide for slide: 213

- the structure is similar to that of the main `PROGRAM` unit except a `SUBROUTINE` can be parameterised (with arguments) and these must be declared in the declarations section.
- a `SUBROUTINE` may include calls to other program units either internal, external or visible by `USE` association (i.e., in a module — see later).
- in order to promote optimisation a recursive procedure must be specified as such, i.e., it must have the `RECURSIVE` keyword at the beginning of the subroutine declaration.
- if a `SUBROUTINE` has no arguments (parameters) then the parenthesis in the `SUBROUTINE` statement are optional (unlike a `FUNCTION`).
- as with the main `PROGRAM` the `SUBROUTINE` must terminate with an `END` statement and it is jolly good show to append `SUBROUTINE` and the name of the routine to this line as well.
- `SUBROUTINE`s may contain internal procedures but only if they themselves are *not* already internal.

214: External Subroutine Example

An external procedure may invoke a further external procedure,

```
SUBROUTINE sub1(a,b,c)
 IMPLICIT NONE
 EXTERNAL sum_sq  ! Should declare or use an INTERFACE
 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
```

Guide for slide: 214

The principle is the same as for calling an internal procedure except for two main things

- 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.
- the external procedure should be declared in an `EXTERNAL` statement. (This is optional but is good practise.)

215: 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 result type>
         <declaration of local objects>
                  ...
         <executable stmts, assignment of result>
 [ CONTAINS
         <internal procedure definitions> ]
END [ FUNCTION [ <procname> ] ]
```

here, <*procname*> must be declared.

Guide for slide: 215

- A `FUNCTION` is very similar to a `SUBROUTINE` except that it has a type, <*prefix*>. The type can either be declared in the declarations area of the code or 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, the two syntax specifications detail both cases.
- `RECURSIVE` functions must be explicitly defined.
- owing to the possibility of confusion between a variable name and a function reference the parentheses are **not** optional.
- the function name <*procname*> is the result variable.
- the function must contain a statement which assigns a value to the function name — a routine without one is an error.
- use the full form of the `END` statement.
- functions can return results of any type (including user defined types and pointers), kind and rank.

216: External 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)
```

- external functions should be declared as `EXTERNAL` or else the `INTERFACE` should be given,

```
INTEGER, EXTERNAL :: largest
```

- The function is defined as follows,

```
INTEGER FUNCTION largest(i,j,k)
 IMPLICIT NONE
 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)
 IMPLICIT NONE
 INTEGER :: i, j, k
 INTEGER :: largest
  ...
END FUNCTION largest
```

Guide for slide: 216

- the dummy and actual arguments must correspond in number, type and kind (as with `SUBROUTINE`s).
- can have functions which have arguments that are procedures.
- can have function that have no arguments. An empty set of brackets must be supplied.
- if the same function is called twice in the same statement problems may occur — a function call should not alter its arguments or any global variables.

217: Procedure Interfaces

For `EXTERNAL` procedures it is possible to provide an *explicit interface* for a procedure. Consider:

```
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), SAVE :: 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 given by the statements which appear in the declarations part of any program unit that calls `expsum`:

```
INTERFACE           ! for EXTERNAL procedures
 SUBROUTINE expsum( n, k, x, sum )
  USE KIND_VALS:ONLY long
  INTEGER, INTENT(IN) :: n
  REAL(long), INTENT(IN) :: k,x
  REAL(long), INTENT(OUT)  :: sum
 END SUBROUTINE expsum
END INTERFACE
```

Interfaces replace any `EXTERNAL` statements and are *not* needed for internal (or module) procedures.

Guide for slide: 217

`INTERFACE`s are new to Fortran 90,

- it is possible (often mandatory) to declare the explicit interface of an external procedure that is referenced in a particular program unit.
- it is generally a good idea to make all interfaces explicit.
- an interface declaration is initiated with the `INTERFACE` statement and terminated by `END INTERFACE`.
- the interface gives the characteristics of the dummy arguments, (for example, the name, type, kind, ranks and whether it is `OPTIONAL`; the `INTENT` should also be given but as this has not been covered it is omitted), and the characteristics of the procedure (i.e. name, class and type for functions). An interface cannot be used for a procedure specified in an `EXTERNAL` statement (and vice-versa).
- the `USE` statement is required in order that the kind of the `REAL` valued dummy arguments is known.
- an interface declaration is included once in the declarations part of the calling program unit.
- interfaces must match the procedure that they refer too.
- interfaces are only relevant for external procedures as internal procedures have an implicitly visible interface. Interfaces to intrinsic procedures are also implicit.

218: What Appears in an Interface?

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 and remove the need for any other form of declaration of that procedure.

Guide for slide: 218

219: 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
```

Explicit interfaces permit separate compilation, optimisation and type checking.

Guide for slide: 219

- using `INTERFACE` provides for better optimisation and type checking.
- `INTERFACE`s allow compilers to perform efficient separate compilation.

220: Required Interfaces

Explicit interfaces are mandatory if an EXTERNAL procedure has:

- dummy arguments that are assumed-shape arrays, pointers or targets;
- OPTIONAL arguments;
- an array valued or pointer result (functions);
- a result that has an inherited LEN=* length specifier (character functions);

and when the reference:

- has a keyword argument;
- is a defined assignment;
- is a call to the generic name;
- is a call to a defined operator (functions).

Guide for slide: 220

Explicit interfaces are mandatory if the 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 as assumed-shape array (see later), or the types and attributes of pointers or targets.
- has `OPTIONAL` arguments.

  So the compiler knows the names of the arguments and can figure out the correct association when any of the optional arguments are missing.
- has an array valued or pointer result (functions).

  Needs to know to pass back the function result in a different form from usual.
- has a result that has an inherited `LEN=*` length specifier (character functions).

  The compiler needs to know to pass string length information to and from the procedure.

Explicit interfaces are mandatory if the reference:

- has a keyword argument;
- is a defined assignment;
- is a call to the generic name;
- is a call to a defined operator (functions).

221: Procedure Arguments

If `EXTERNAL` procedures are to be used as arguments they must be declared at the call site as:

- `INTRINSIC` — for in-built external procedures.

```
INTRINSIC MVBITS
REAL, INTRINSIC :: ASIN
```

- `EXTERNAL` — for external or dummy procedures.

```
EXTERNAL My_Subby
INTEGER, EXTERNAL :: My_Funky
```

  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.

In both cases the *specific*, not generic, procedure name must be used as an actual argument.
Internal procedures are **forbidden** to appear as arguments.

Guide for slide: 221

If an external procedure is to be used as an argument it needs to be declared at the call site in an `INTRINSIC` or `EXTERNAL` statement:

- `INTRINSIC` and `EXTERNAL` statements **cannot** contain the `::` separator. This seems to be a language anomily and will **not** be changed in the forseeable future. [No reason for this at all.] Attributed type statements can contain the `::` separator.
- `INTRINSIC` and `EXTERNAL` statements are really only necessary if the procedures are dummy arguments or are to be used as actual arguments, however, there is no harm in using them as a matter of course.
- it is very good practise to use `INTRINSIC` and `EXTERNAL` statements for all referenced procedures regardless of whether they are to be used as arguments or not.
- 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.
- internal procedures **cannot** be used as actual arguments which correspond to dummy procedures; they must be evaluated before the argument association is made. In fact there is no mechanism for making them associated with a dummy procedure name.
- 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.

222: Example of Procedure Arguments

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
```

Guide for slide: 222

- a type declaration which gives a function an `INTRINSIC` or `EXTERNAL` statement cannot contain the `::` separator; attributed statements can!
- `INTRINSIC ASIN` specifies that `ASIN` refers to the Fortran 90 specific name of the intrinsic function and allows it to be used as an actual argument which corresponds to a dummy procedure.
- `REAL, EXTERNAL :: my_sin` declares `my_sin` to be a 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`.
- must always use the **specific** name of any intrinsic procedure, i.e., must use `ASIN` and not `SIN`.

223: The DATA Statement

The `DATA` statement is good for initialising odd shaped sections of arrays. Syntax,

`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 >. Any object initialised by the `DATA` statement has the `SAVE` attribute

Guide for slide: 223

- in most cases one can use the standard `=` form of object initialisation `REAL :: X = 1.0`.
- if the above isn't possible use the `DATA` statement.
- the `DATA` statement is executed *once* on the first invocation of the procedure.
- currently a `DATA` statement can be placed anywhere in the relevant program unit but its is common practice to place them amongst the declarations.
- each < *data*-list > can only contain constants or structure (user-defined type) constructors, plus implied-do loop specifiers and repetition specifiers.
- the number of constants in each < *data*-list > must be equal to the number of variables / array elements in the corresponding < *var*-list >.

224: 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 not as simple to do with an initialisation statement as it is with a DATA statement:

```
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. In this context it cannot be confused with the multiplication operator because such operators are not allowed in a DATA statement.

Guide for slide: 224

- the above example 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 (which is executed once on the first invocation of the program unit) allows this operation to be spread over a number of lines.
- the above DATA statement makes it simple to see what is being initialised,
- the * operator is a repeat specifier, 100*1.0 is equivalent to a comma separated list of 100 occurrences of 1.0, It can be applied to any literal or scalar constant.
- the number of repeats must be an integer scalar literal (a number).
- a list of comma separated initialisations is allowed, e.g.,

```
DATA matrix(1,    1:100) / 50*1.0, 50*2.0  / ! top row
```

225: 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 is 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
```

Guide for slide: 225

Morgan and Schonfelder is most wrong with this example,

- `(diag(i,i), i=1,100)` is as if `diag(i,i)` were nested in a `DO i=1,100` loop, however, the implied do has a different functionality.
- `(diag(i,i), i=1,100)` selects the diagonal (100 elts),
- the nested implied `DO` (second `DATA` statement) 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.
- `DATA ((diag(i,j),diag(j,i),j=i+1,100),i=1,100)` selects the following elements in the following order:

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

226: 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).
- can be very useful!

The basic syntax is

GOTO < *numeric-label* >

The label must exist, be in the same scoping unit as the statement and be executable.

Guide for slide: 226

- GOTO should not be used for simulating other available control structure such as loops, use the purpose-designed syntax.
- is a powerful but undisciplined branching statement.
- GOTO cannot jump out of a program unit.
- can be dangerous.
- can lead to unstructured code (logical spaghetti).
- a construct name may **not** be used instead of a < *numeric-label* >.
- the label must exist.
- should be avoided as they have shown to be one of the biggest causes of bad programming and errors.
- is very useful in exceptional circumstances such as jumping out of heavily nested structures.

227: 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 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.

Guide for slide: 227

Code is appalling use of `GOTO`

```
     GOTO 10
```

- □ forward jump to line 10

```
 69 j = j - 1 ! loop
      ...
      IF (j .ne. 0) GOTO 69
```

- □ simulated loop (as long as `j` is greater than or equal to 1) with a backwards jump.
- □ note the label `99` is equivalent to `099`

The whole example could be be rewritten in a far neater and structured way.

228: RETURN and STOP

Can be used to program exceptions in procedures.
For example,

```
SUBROUTINE sub(ierror)
 INTEGER, INTENT(OUT) :: ierror
 ...
 ALLOCATE(A(100),STAT=ierror)
 IF (ierror>0) THEN
  PRINT*, 'memory fault'
  RETURN
 END IF
     ...
END SUBROUTINE
```

`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 the program finishes.

Guide for slide: 228

If odd things happen in a procedure can use `RETURN` or `STOP`.

- the `RETURN` statement transfers control to the last line of the procedure and then back to the calling program unit. It is a quick exit from the middle of a procedure and is useful when things have gone wrong.

  In the case of `RETURN`, the same effect 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 example, there seems little point in carrying on if the `ALLOCATE` fails - a quick retun is required!

- the `STOP` statement causes the program to terminate immediately. A string or number may be written to the standard output when this happens. It is, again, useful for exceptions, however, it 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.

229: Fortran 95

Fortran 95 will be the new Fortran Standard.

- ☐ FORALL statement and construct

```
FORALL(i=1:n:2,j=1:m:2)
 A(i,j) = i*j
END FORALL
```

- ☐ nested WHERE constructs,
- ☐ ELEMENTAL and PURE procedures,
- ☐ user-defined functions in initialisation expressions,
- ☐ automatic deallocation of arrays,
- ☐ improved object initialisation,
- ☐ remove conflicts with IEC 559 (IEEE 754/854) (floating point arithmetic),
- ☐ deleted features, for example, PAUSE, assigned GOTO, cH edit descriptor,
- ☐ more obsolescent features, for example, fixed source form, assumed sized arrays, CHARACTER*< *len* > declarations, statement functions,
- ☐ language tidy-ups and ambiguities (mistakes),

Guide for slide: 229

The full content of Fortran 95 is not yet finalised but it is thought to be as described on the slide.

Rationale (by Craig Dedo):

- 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.

- 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.

- 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.

  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.

- 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.

- 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 un`SAVE`d allocatable arrays and permits use of the `ALLOCATED` intrinsic function to discover the current allocation status of the array at any time.

- 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, un`SAVE`d 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.

- 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 `NaNs`.

  These conflicts were removed to allow processors the ability to implement more of the IEC 559 arithmetic model without violating the Fortran standard.

- 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.

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.

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.

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`.

- any deleted features will appear in an Annex to the standard. The current list of deleted features (September 2, 1997) 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.
- 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).
- language tidy-ups are mainly concerned with ironing out ambiguities in the Fortran 90 standard and adding extra text to make matters clearer.

230: High Performance Fortran

High Performance Fortran (or HPF) is an ad-hoc standard based on Fortran 90. It contains

- Fortran 90,
- syntax extensions, FORALL, new intrinsics, PURE and ELEMENTAL procedures,
- discussion regarding storage and sequence association,
- compiler 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
```

Guide for slide: 230

- 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.

- compiler directives:

  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.

  For example,

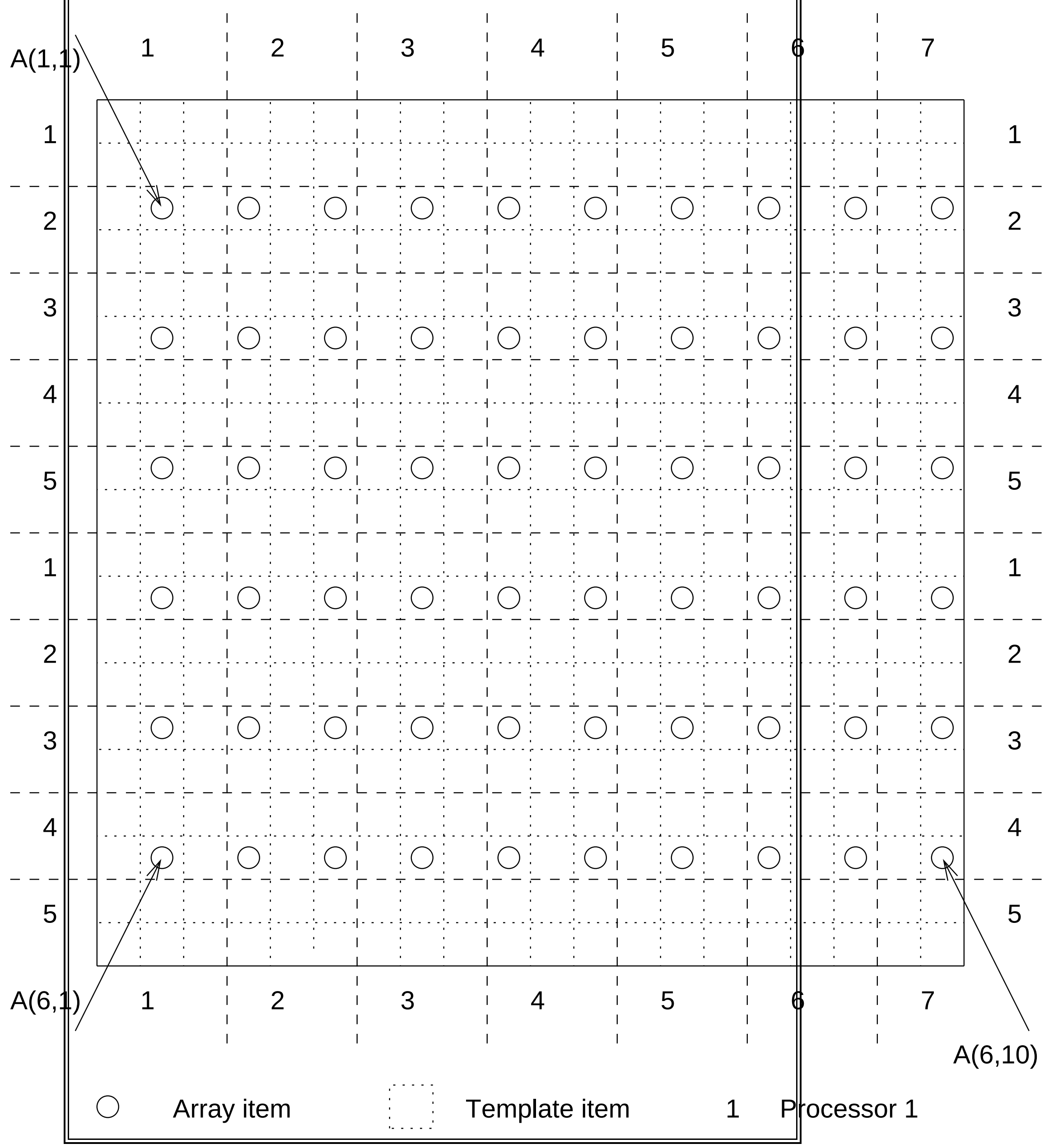
  ```
  A(2,1) = A(5,1)
  ```

  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).

231: Data Alignment



Guide for slide: 231

```
!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 on the slide.

- a 5×7 array of PROCESSORS is defined,
- a 20×20 TEMPLATE is defined,
- the template is distributed onto the processor array.

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. The first array element A(1,1) is aligned to template element T(1*3, 1*2), i.e., element T(3,2) the second is aligned to T(6,2) and so on.

The distribution is different in each dimension:

- in dimension 1 CYCLIC(2) distribution means that each processor (in dimension 1) initially gets two template elements, when each processor has been covered processor 1 will get a further two template elements. This process continues until the template has been fully distributed.
- in dimension 2 BLOCK(3) distribution means that each processor (in dimension 1) gets three template elements until the template has been fully distributed. If all processors have been covered and the template has not been fully used up then the program is in error.

The result is that 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*.

232: The End

Guide for slide: 232