Scientific Software Development with Fortran

Dr. Drew A. McCormack

ISBN 978-1-4452-5445-6

1. Preliminaries

This section outlines the objectives of this course, and the preparation you should do before starting.

1.1. Fortran in Practice

Most programming courses are academic in nature, but in this course we are going to focus more on the practicalities of software development in modern Fortran. In addition to the usual programming exercises, you will get the opportunity to work in a real scientific application. You will find out how software is built in Fortran, rather than just learning the rules of the language, as well as how to adapt to existing source code, which is how most of us end up working in the real world.

You are not expected to have any programming experience before beginning this course, but you should be comfortable working with UNIX. In order to develop a Fortran program, you need to be able to edit text files, and run command-line tools to build the application. Without having used a UNIX command line interface before, this would be very difficult.

1.2. Confusion Monte Carlo

The Fortran program you will be working on is called 'Confusion Monte Carlo' (CMC). It implements the Diffusion Monte Carlo (DMC) method for finding the ground state of a one-dimensional quantum system. The DMC method is explained in the paper 'Introduction to the Diffusion Monte Carlo Method' by I. Kosztin *et al* (Am. J. Phys. **64** [5], 633), which can be downloaded online using this URL: http://arxiv.org/abs/physics/9702023 If you would like to know all the details of the DMC method, you can find them in this paper, but you do not have to read it if you don't want to: it is not necessary to fully understand the theory of the method in order to implement it in a program.

In the appendices you will find notes that give an overview of the DMC algorithm used in the CMC program, and the design of the program. These appendices provide you with the level of knowledge necessary for you to work with the source code; you should read through them before getting started on the course proper.

The source code itself is available for download from http://www.mentalfaculty.com/mentalfaculty/Downloads.html

2. Basics

You have to walk before you can run. Everyone has to learn the basics of a programming language before they can start to think about how you write programs with it. This section covers the most fundamental aspects of Fortran programming.

2.1. Introducing Fortran

Fortran is a very old programming language...in fact, you could say it's the oldest. Fortran, or FORTRAN as it was spelt in the beginning, was developed by IBM in the 1950's to be the first high-level programming language. Rather than writing programs in assembly language, which is more suited for reading by machines than humans, FORTRAN was legible to people.

If Fortran is so old, why are we still using it? That's a good question, and we will try to answer it shortly, but first it is important to realize that Fortran has undergone many changes besides its name since it was first devised. Programs written in the FORTRAN of yesteryear can look very different to those developed in the latest iterations of the language. You might even say they are completely different languages.

To give you a feel for the scope of the changes in the language, we'll take a look at two short pieces of source code. The first is written in old FORTRAN.

```
      subroutine phipdep
C
C     Subroutine for plotting the phip-dependence of the PES
C     for given Zp,r,xp,yp,thetap.
C
      implicit real*8(a-h,o-z)
C
      common/ctrans/pi,costht,sintht,dsq3,dsq2,cosith,sinith,
     +              tantht,tanith,dsq6
      common/surf/delta,gx,gy
C
      DIMENSION RAB(6)
      dimension vint(100000),dvdphip(100000),phip(100000),
     +          dvdphipl(100000),dvdphiph(100000)
C
C     read in the Zp, r, xp, yp and thetap and the range of phip values
C     to be plotted.
C
      open(1,file='phipdep.inp',status='old')
      read(1,*) zp,r,xpin,ypin,thetapin
      read(1,*) phipin1,phipin2,nphi
      close(1)
      xp = xpin*delta
      yp = ypin*dsq3/6.0d0*delta
      thetap = thetapin / 180.0d0 * pi
      phip1 = phipin1 / 180.0d0 * pi
      phip2 = phipin2 / 180.0d0 * pi
      delphip = (phip2-phip1) / (nphi-1.0d0)
      open(2,file='phipdep.xpypTthp.Zp=zp.r=r.vpot.out',
     +       status='unknown')
      open(3,file='phipdep.xpypTthp.Zp=zp.r=r.dvdphip.out',
     +       status='unknown')
c      write(2,9001) zp,r,xp/delta,yp*6.0d0/(dsq3*delta),
c     +              thetap/pi*180.0d0
```

```
c      write(3,9001) zp,r,xp/delta,yp*6.0d0/(dsq3*delta),
c     +              thetap/pi*180.0d0
      RAB(1) = xp
      RAB(2) = yp
      RAB(3) = zp
      RAB(4) = r
      RAB(5) = thetap
C
      do i = 1, nphi
         RAB(6) = phip1 + (i-1.0d0)*delphip
         phip(i)=RAB(6)
         CALL VH2PT211(RAB,v,dum1,dum2,dum3,dum4,dum5,DVDFIP,
     +                     dum7,dum8,dum9,dum10,dum11,dum12)
         vint(i)=v
         dvdphip(i)=DVDFIP
         write(2,9002) phip(i),v
      enddo
      do i = 2, nphi-1
         dvdphipl(i)=(vint(i)-vint(i-1))/(phip(i)-phip(i-1))
         dvdphiph(i)=(vint(i+1)-vint(i))/(phip(i+1)-phip(i))
         if((dvdphip(i).gt.dvdphipl(i).and.
     +       dvdphip(i).lt.dvdphiph(i)).or.
     +      (dvdphip(i).lt.dvdphipl(i).and.
     +       dvdphip(i).gt.dvdphiph(i))) then
            write(3,9004)
         else
            write(3,9003) phip(i),vint(i),dvdphip(i),
     +                    dvdphipl(i),dvdphiph(i)
         endif
      enddo
      close(2)
      close(3)
C
 9001 format(5x,'Zp (in bohr) =',t45,f8.4/,
     +       5x,'r  (in bohr) =',t45,f8.4/,
     +       5x,'xp (in units of the lattice constant) =',t45,f8.4/,
     +       5x,'yp (in units of the lattice constant) =',t45,f8.4/,
     +       5x,'thetap (in degrees) =',t45,f8.4/)
 9002 format(2f18.12)
 9003 format(5f18.12)
 9004 format(5x,'ok')
C
      return
      end
```

And here is something written in a more recent style.

```
subroutine OutputResults(self, writer)
   use HistogramBuilderModule
   type (EnsembleWalker), intent(in), target         :: self
   type (OutputWriter), intent(inout)                :: writer

   type (HistogramBuilder)                           :: histoBuilder
   type (Particle)                                   :: part
   real(KREAL)                                       :: lowerBound, upperBound
   real(KREAL), allocatable                          :: positions(:), wavefunc(:)
   integer(KINT)                                     :: numberOfBins, i, n

   if ( self%isConverged ) then
      call Write(writer, 'Calculation Convergence', 'Succeeded')
   else
      call Write(writer, 'Calculation Convergence', 'Failed')
   endif

   call Write(writer, 'Propagation Steps', self%numStepsTaken)
   call Write(writer, 'Energy', self%summedReferenceEnergy / self%numStepsForAverage)
   call Write(writer, 'Number of Particles', GetNumberOfParticles(self%ensemble))

   ! Create histogram for the particle wavefunction. Print out the wavefunction.
   n = GetNumberOfParticles(self%ensemble)
   allocate(positions(n))
   do i = 1, n
      part = GetParticleAtIndex(self%ensemble, i)
      positions(i) = part%position
   enddo
   lowerBound = minval(positions)
   upperBound = maxval(positions)
   call New(histoBuilder, lowerBound, upperBound, self%properties%numberOfEnergyBins)
   allocate( wavefunc(self%properties%numberOfEnergyBins) )
   wavefunc = CreateHistogram(histoBuilder, positions)
   wavefunc = wavefunc / max(1.0_KREAL, sum(wavefunc))
   wavefunc = wavefunc / GetBinWidth(histoBuilder)
   wavefunc = sqrt(wavefunc)
   call Write(writer, 'Wavefunction', GetBinCenters(histoBuilder), wavefunc )
   call Delete(histoBuilder)
   deallocate(positions, wavefunc)

end subroutine
```

You are not expected to understand either of these pieces of code — at least, not yet — but it should be apparent that they are quite different, and that the Fortran language has undergone considerable evolution over the years.

After smaller updates to the language early on, Fortran received a major overhaul in 1978, with the release of the FORTRAN 77 standard, and then again in 1992, with Fortran 90. Since then, we have seen a minor update called Fortran 95, and a major revision called Fortran 2003. In practice, Fortran programs are still written in either FORTRAN 77 or Fortran 90/95, because tools are not yet widely available for the new Fortran 2003 standard.

Although there have been many different versions of Fortran over the years, it is important to realize that they are all backwards-compatible. That is, you can use tools designed for Fortran 90 to develop a program written in FORTRAN 77 — each iteration of the language is a superset of the previous one.

Backwards-compatibility makes it easier for programmers to use the latest language advances in their old programs, but it also complicates a programmer's job. Old Fortran programs are often a mix of different programming styles, representing the evolution of the language, and this inconsistency can make understanding Fortran source code difficult at times. Programs written in another era are called *legacy programs*, and there are many scientific applications that fall into this category. As a scientific software developer, it's something you have to learn to live with.

There are a couple of reasons people still program in Fortran, even a half century after its birth. Legacy is the main one: there are thousands of applications written in old Fortran, many of which are still vitally important to scientific research. Discarding these applications is not an option, and they are often far too big to rewrite in a more modern style or in a different language altogether.

But there are other reasons to use Fortran. Fortran has been designed from the ground up for high-performance numerical computing. Fortran is a poor choice if you are developing a web site, but an excellent choice if you have to do some linear algebra, or need to leverage a massively-parallel supercomputer. The latest versions of the language also include very powerful operations for working with arrays of data — these are not available in other languages. So even if you don't have a legacy problem, Fortran may still be a good choice.

2.2. Building Fortran Programs

Before we start to discuss how you write a Fortran program, we need to consider how you *build* one. Fortran is a *compiled language*, which means that you need a tool called a *compiler* that takes the source code that you write and translates it into something a computer can execute.

There are two phases to building an executable program: *compiling* and *linking*. In the compilation phase, the compiler goes through each of the source code files that you have written and converts them into *object files*, which contain a version of the program written in an intermediate language called *assembler*. Assembler is comprised of instructions that can run on the computer's *Central Processing Unit (CPU)*. In the linking stage, the *linker* takes all the object files and connects them up to form a single executable application. This *executable* contains binary code that runs directly on the CPU: *machine code*.

There are other aspects of Fortran that complicate the building process somewhat. In particular, some source files need to be compiled before other source files — files can have dependencies. What this means is that you cannot simply compile your source files in a random order, but you need a tool to make sure they are compiled in an order consistent with the implicit dependencies in the source code.

So how does it work in practice? To compile a single Fortran source file, we can use the gfortran tool like this

```
gfortran -c some_file.f90
```

This command takes a Fortran source file called 'some_file.f90', and will produce an object file called 'some_file.o'. Depending on the content of some_file.f90, secondary product files may be produced with the extension .mod; these are used when compiling other source files that depend on some_file.f90. We'll discuss how these dependencies arise later in the course.

If you have compiled all of the source files — in the correct order — you can link the resulting object files together to form an executable program.

```
gfortran -o some_app some_file.o another_file.o
```

Here we have linked together two object files, some_file.o and another_file.o, but you should enter the names of all of the object files in the program on the line. The -o option tells gfortran the name of the executable that it should create, in this case 'some_app'.

We mentioned earlier that you need to compile your source files in an order consistent with their interdependencies. The *make* tool can help you do this: make reads a file called a *make file*, which tells it the dependencies between the various source files, and then proceeds to build the program. Here is an example make file, which by default is called 'makefile'.

```
# -------------------Macro-Defs---------------------
```

```
F90=gfortran

# -------------------End-macro-Defs---------------------------

# Here is the link step
cmc:NumberKinds.o OutputWriter.o Logger.o InputReader.o Potential.o
ParticleEnsemble.o GaussianDistributor.o HistogramBuilder.o EnsembleWalker.o
DMCCalculation.o cmc.o
	$(F90) -o cmc NumberKinds.o OutputWriter.o Logger.o InputReader.o Potential.o
ParticleEnsemble.o GaussianDistributor.o HistogramBuilder.o EnsembleWalker.o
DMCCalculation.o cmc.o

# Here are the compile steps

NumberKinds.o:./NumberKinds.f90
	$(F90) -c ./NumberKinds.f90

OutputWriter.o:./OutputWriter.f90 NumberKinds.o
	$(F90) -c ./OutputWriter.f90

Logger.o:./Logger.f90 NumberKinds.o OutputWriter.o
	$(F90) -c ./Logger.f90

InputReader.o:./InputReader.f90 NumberKinds.o Logger.o
	$(F90) -c ./InputReader.f90

Potential.o:./Potential.f90 NumberKinds.o OutputWriter.o Logger.o
	$(F90) -c ./Potential.f90

ParticleEnsemble.o:./ParticleEnsemble.f90 NumberKinds.o OutputWriter.o Logger.o
	$(F90) -c ./ParticleEnsemble.f90

GaussianDistributor.o:./GaussianDistributor.f90 NumberKinds.o OutputWriter.o Logger.o
	$(F90) -c ./GaussianDistributor.f90

HistogramBuilder.o:./HistogramBuilder.f90 NumberKinds.o OutputWriter.o Logger.o
	$(F90) -c ./HistogramBuilder.f90

EnsembleWalker.o:./EnsembleWalker.f90 NumberKinds.o OutputWriter.o Logger.o
Potential.o ParticleEnsemble.o GaussianDistributor.o HistogramBuilder.o
	$(F90) -c ./EnsembleWalker.f90

DMCCalculation.o:./DMCCalculation.f90 NumberKinds.o OutputWriter.o InputReader.o
Logger.o EnsembleWalker.o Potential.o
	$(F90) -c ./DMCCalculation.f90

cmc.o:cmc.f90 DMCCalculation.o
	$(F90) -c cmc.f90
# This entry allows you to type " make clean " to get rid of
# all object and module files
clean:
      rm -f -r f_{files,modd}* *.o *.mod *.M *.d V*.inc *.vo \
      V*.f *.dbg album F.err
```

It is not necessary for you to be able to write a make file for this course, but to give you some idea how it works, each line above tells `make` how to build a particular *target*, and what the dependencies of that target are. Take the object file ‘HistogramBuilder.o’, for example:

```
HistogramBuilder.o:./HistogramBuilder.f90 NumberKinds.o OutputWriter.o Logger.o
```

```
       $(F90) -c ./HistogramBuilder.f90
```

The target is on the left, and is followed by a colon. After the colon are the files that need to be up-to-date *before* HistogramBuilder.o can be built. If, after they are brought up-to-date, any of the files on the right of the colon were changed more recently than HistogramBuilder.o, HistogramBuilder.o needs to be built.

In this case, `make` checks the last change date of HistogramBuilder.f90 — the original source code file — and the listed object files, and, if any were changed more recently than HistogramBuilder.o, the command on the second line is carried out. This command is simply a variation on the compilation command we saw earlier.

Once you have a make file, using the make tool is easy.

```
make cmc
```

This command uses the `make` command to build the program 'cmc', which should be a target in the make file.

You can write make files manually, and keep them up-to-date with changes in your program, but this can be quite error prone. You are likely to forget to add a dependency, and this can lead to strange problems when building. It's better to let a script scan your program, and create a make file for you. In this course, we are going to use the `fmkmf.pl` tool. To create a make file, you just run `fmkmf.pl` on the main file of your program; it will figure out what other files are needed, and any dependencies that exist. The default fortran compiler is f90, you can change this to your own preferred compiler by setting the environment variable FMKMF_F90, e.g. export FMKMF_F90=ifort (to select the Intel compiler).

```
fmkmf.pl cmc.f90 > makefile
```

This will produce a make file to build an executable called 'cmc'. To build the program, you just use the make command above.

2.3. Running a Fortran Program

Once you have an executable, you can run it. Depending on the program, you may need to supply options on the command line, write input files, or redirect standard input and output.

To see how this works, we will now see how to run the CMC executable. After building, the file `cmc` is the program executable, and `cmcinput.txt` is the input file. To run it, enter the following command.

```
./cmc
```

You should see output appear on your screen as the program runs. If you would like to have the output in a file instead of on screen, redirect it.

```
./cmc > cmc_output.txt
```

If it is taking too long, and you want to kill it, just enter Ctrl-C. You can also try changing some values in `cmcinput.txt`, and rerunning the program, to see how results are affected. We will cover what the various input parameters are, and how CMC works internally, as the course proceeds.

Exercise: Building and Running Confusion Monte Carlo

Use the fmkmk.pl tool to create a make file for the cmctests.f90 file, run make, and then execute the resulting binary program.

2.4. Fixed Form and Free Form

With the preliminaries behind us, we can now start to look at Fortran itself. When Fortran was first invented, there were no keyboards or 30 inch LCD displays to help you write your applications. Programs were coded on punchcards, and fed into a punchcard reader on the computer.

This legacy is still seen today in Fortran code. Fortran can be written in one of two forms: *fixed* and *free*. Fixed form is a relic of the punchcard days: 6 columns are left mostly empty, with program instructions beginning in the 7th column, and instructions may not be entered beyond the 72nd column. Free form is used in more modern Fortran programs, and does away with these restrictions. In this course, we will focus on free form, but you should be aware of fixed form, because you are bound to come across it at some point in the future.

In fixed form, you add comments to your code by entering a C in the first column, and you continue a long line by entering a character in the 6th column.

```
C     Subroutine for plotting the phip-dependence of the PES
C     for given Zp,r,xp,yp,thetap.
C
      implicit real*8(a-h,o-z)
C
      common/ctrans/pi,costht,sintht,dsq3,dsq2,cosith,sinith,
     +              tantht,tanith,dsq6
```

The + is in the 6th column, and indicates that the line above continues, as if it were written as one long line. The comments delineated by C’s in the first column are ignored by the compiler, and are intended purely as an aid to the programmer. All other commands begin in the 7th column.

Free form imposes no column-based restrictions. A comment in free form source code begins with an exclamation mark (!), and a line is continued by entering an ampersand (&) at the end. The fixed form code above could be rewritten in free form like this.

```
! Subroutine for plotting the phip-dependence of the PES
! for given Zp,r,xp,yp,thetap.
implicit real*8(a-h,o-z)
common/ctrans/pi,costht,sintht,dsq3,dsq2,cosith,sinith, &
   tantht,tanith,dsq6   ! Comment at end of line
```

Comments can also be entered at the end of a line, as shown; any text after the exclamation mark is ignored by the compiler.

2.5. Variables and Numerical Types

To be able to calculate things, we need to be able to represent, store, and operate on numbers. Fortran has support for several different numerical types. The most common are *integers*, which represent whole numbers, and *reals*, which represent decimal numbers.

Integers include numbers like 1, 5, and -150844. They are whole numbers, and you can enter them directly into your source code. For example, here is a short piece of Fortran that prints out the result of a simple calculation.

```
print *, '5 plus 14345 is ', 5 + 14345
end
```

If you could only program using *literal numbers* like this, it would be quite restrictive. So Fortran allows you to store numbers in *variables*. Here is an example that declares an integer variable, and then stores the result of the calculation in it.

```
integer i
i = 5 + 14345
print *, '5 plus 14345 is ', i
end
```

In this case, instead of printing out the result of the calculation directly, we stored it in the variable 'i', and then retrieved the result from i to print out later.

You can think of variables as pigeon holes in the memory of the computer, each with a label. The *declaration*

```
integer i
```

tells the compiler to set aside enough space to store an integer in memory, and to label that space with the name 'i'. Thereafter, whenever 'i' is encountered the compiler knows how to get its value from memory, or store a new value.

The name of a variable can include letters and numbers, but cannot begin with a number, and must be 31 characters or less. You can also use underscores, but no other punctuation marks are allowed.

There are a few different ways you can declare a variable, but the most useful are as shown above, and as follows

```
integer, parameter :: i = 5
```

In this form, a double colon is used to separate keywords that define the variable's type from it's name and initial value. In this particular instance, the keyword 'parameter' indicates that the variable `i` is constant, *i.e.*, may not be changed anywhere in the program source code.

Unlike integers, real numbers can have a decimal component; like integers, you can enter reals literally, or create real variables.

```
real r
r = 5.0 + 43352.25456
print *, '5.0 plus 43352.25464 is ', r
end
```

There are a couple more numerical types that you will encounter in Fortran programs. The first is *complex*, which is used to represent complex numbers, and the other is *logical*, which represents boolean values that are either true or false. Here is some code that uses these types.

```
complex c1, c2
logical areEqual
areEqual = .false.
```

```
c1 = cmplx(1.0, 2.0)      ! 1.0 + 2.0 i
c2 = cmplx(-1.0, -2.0)    ! -1.0 - 2.0 i
areEqual = ( c1 == c2 )
print *, 'Are the complex numbers equal? ', areEqual
end
```

Complex literals are represented as shown, using ‘cmplx’, with the first argument the real part of the number, and the second the imaginary component. Logical literals can also be entered as either ‘.true.’ or ‘.false.’.

Exercise: Compiling a Simple Program

Copy or type one of the simple programs above into a file called 'simple.f90'. Compile the program with the command `gfortran -o simple simple.f90`, and run the executable that results.

2.6. Operators

We've already seen operators like + and == in action above. Operators are what enable you to perform calculations on numbers. Fortran has operators for all the standard arithmetic and logical operations. Here is a table of the most important operators that you will encounter.

Operator	Description	Example
+	Add two numbers	1 + 534.0
–	Subtract second number from first	5 – 2
*	Multiply two numbers together	25 * 6.023
/	Divide first number by the second	5
**	Raise first number to second	25.0**2
.or.	Logical OR	.true. .or. .false.
.and.	Logical AND	isGreen .and. isBig
.not.	Logical NOT	.not. .true.
.eq. or ==	Compare two numbers for equality	12 == 12
.ne. or /=	Compare two numbers for inequality	12 /= 13
.lt. or <	True if first number is less than the second	4 < 5
.le. or <=	True if first is less than or equal second	5 <= 5
.gt. or >	True if first is greater than the second	45325.2345 > 1.3455
.ge. or >=	True if first is greater than or equal second	–5.5 >= –5.5
=	Assign a variable to a value	x = 5

Logical operators like > and == can take one of two forms: the mathematical symbol can be used, or you can use the dot notation (*eg.* `.gt.`), which derives from older versions of the language.

The assignment operator (=) takes the value of an expression on the right-hand side, and stores it in the variable on the left hand side. Be careful not to confuse assignment with the == operator, which compares two numbers for equality.

One question that arises in this is what happens when the numbers in an expression are of different types. For example, what happens when you evaluate this expression:

```
5.0 * 5
```

Is the result an integer or a real? The answer is that the numbers in the expression are first converted to the most general type, in this case real. So the integer 5 will be converted to a real (5.0), and then the two real numbers will be multiplied together giving 25.0, a real number.

You need to be particularly careful with the division operator, because it may give you a result you weren't expecting. Take this expression, for example.

```
real x
x = 5 / 10
```

You may be thinking that `x` will end up with the value 0.5, but you would be wrong. 5 and 10 are both integers, so the result of the division is also an integer. In Fortran, this means that any decimal component of the result is simply discarded. So you may expect 0.5, but 5/10 actually gives 0. Which means the expression is equivalent to this

```
real x
x = 0
```

meaning that `x` will end up with the real value 0.0.

If you do actually want the result to be 0.5, you need to convert one or both integers to a real number first, like this

```
real x
x = 5.0 / 10
```

or this

```
real x
x = real(5) / 10
```

Something else to be wary of is comparing real numbers for equality. For example, take this simple program.

```
real r
r = 1.50000123456789123 45
print *, r == 15000.012345678912345 / 10000.0
end
```

It might seem that this should print out 'T' for 'true', but it actually prints 'F' for 'false'. The reason for this is *numerical roundoff*. A computer has limited memory in which to store a real number, and only a fixed number of discrete possibilities can be formed with the bits and bytes it has to work with. So even though a real number seems to belong to a continuous spectrum, it must actually take one of a discrete number of possible values.

This can be a problem if you compare real values for equality, because if the values are calculated in different ways, the rounding off that the computer must do to store real numbers in a finite memory slot can lead to unexpected results. In general, it is safer not to directly compare real numbers for equality, but to check if they are almost equal, like this

```
real r
real, parameter :: tolerance = 1.0e-5
r = 1.5000012345678912345
print *, abs(r - 15000.012345678912345 / 10000.0) < tolerance
end
```

This does print out 'T'. Here, the numbers are considered equal if they fall within a tolerance of 1.0x10^{-5} of one another. (`1.0e-5` means 1.0x10^{-5} in Fortran.) The numbers are subtracted, the absolute value taken (using `abs`), and the resulting non-negative number compared to a tolerance value.

Using this approach, you are not a slave to the precision of the computer that you happen to be using to run your program; instead, you decide what constitutes 'equal' numbers in the context of your problem domain.

Finally, it is worth mentioning that you can use parentheses in Fortran expressions, just as in standard arithmetic, to change the order of evaluation. For example,

```
integer x
x = 5 * 3 + 4
```

will result in `x` having the value 19. But

```
integer x
x = 5 * (3 + 4)
```

will give 35.

2.7. Number Kinds and Precision

We already touched upon the issue of numerical precision above. A computer has a finite memory, and this limits the set of values that can be represented. But you do have some control over what the limitations are.

On most computers, an integer is stored in a 4 byte slot in memory. A byte is made up of 8 bits, so there are 32 bits in which to store the integer. Each bit can take one of two values — 0 or 1 — so there are 2^{32} possible permutations. Usually this would give integers in the range –2147483647 to 2147483648.

Real numbers are a bit more tricky, because they are stored in exponential form, with a *sign*, *coefficient*, and *exponent*. The number –1.034534e–45 has negative sign, coefficient 1.034534, and exponent –45.

Suppose that one bit is used by the computer to store whether the real number is positive or negative, *ie*, it's sign, and that 8 bits are used to store the exponent (giving 2^8 possible exponents, *eg*, -127 to 128). That leaves 24 bits to store the coefficient, giving 2^{23} = 8388608 possibilities. (The latter leads to around 7 significant digits for the coefficient.)

So there are limitations to machine precision; computers can't represent every number — not by a long shot. But you can choose to use numerical types with higher precision. Most scientific applications use double precision reals, which fit into 8 bytes or 64 bits of memory. With double precision, you can typically achieve about 16 significant digits of precision for the coefficient, with a maximum exponent of 1024.

A double precision literal can be included in Fortran code by substituting a 'd' for the 'e' in exponential form. For example, 12.543553d0 is a double precision number, and 12.54d1 is a 64-bit version of the decimal number 125.4.

Double precision variables can be declared and initialized like this

```
double precision r
r = 1.25543d0
```

You can also declare double precision reals in other ways, such as

```
real(8) r1
real(kind=8) r2
real*8 r3
```

The number in parentheses is called the *kind*. The kind of a number defines the precision of a number for a given the platform, but a *single value of the kind need not give the same precision on different platforms*.

If you want your program to be more flexible, with the possibility of running it in single or double precision, you should declare your real variables something like this

```
real(KREAL) r
```

`KREAL` is a parameter initialized somewhere in the program. Using this approach, you can change `KREAL` whenever you move to a new platform, or simply decide that you could use more or less precision.

To initialize `KREAL`, you can use a *function* called 'kind'. For example, if you want your real numbers to be double precision, you could set `KREAL` like this

```
integer, parameter :: KREAL = kind(0.d0)
```

This sets `KREAL` to the kind of the number 0.d0, which corresponds to double precision. This would cause all real variables declared in the manner above to be double precision.

We have seen how you use the kind parameter to define a variable's precision, but you also need a means of defining the precision of literal constants. For example, how do you write the number 5.0 so that it is of kind 8? The answer is that you append an underscore followed by the kind, like this

```
real(8) :: r
r = 5.0_8
```

This will also work when you have defined a kind parameter in your program. For example, if you are using `KREAL` as the parameter for the kind of all real variables, you could write this

```
real(KREAL) :: r
r = 5.0_KREAL
```

Exercise: Playing with Precision

In this exercise, we are going to seek out the limits of the machine. Begin by entering the following program in a file called 'precision.f90'. Compile it using gfortran, and run it.

```
real r1, r2
r1 = 1.0000000e10
r2 = 0.0000001e10
print *, r1 == r1 + r2
end
```

It should produce 'F', or false. This is what you would expect, because we are adding the value of `r2` onto `r1`, and you would expect the result to be different to what `r1` was to begin with.

Now insert an extra zero after the decimal point in the initialization of `r1` and `r2`. Compile again, and run. What do you find?

Finally, replace the 'real' declaration with 'double precision'. Recompile and run. What do you find now? How do you explain it?

Exercise: Precision in Practice

Take a look through the CMC source files supplied with this course, and observe how real numbers are declared, and how the kind parameter is used in particular. Look in the file NumberKinds.f90 to see where and how the precision is initialized.

2.8. Intrinsic Routines

Fortran includes a whole library of built-in routines called *intrinisic routines*. A *routine* or *procedure* is a sub-program that you can call upon to perform a calculation or carry out an action. You typically pass a routine some inputs, which may get modified when the routine is finished its work. A routine can also return a result. In Fortran, a routine that has no result is called a *subroutine*, and a routine that returns a value is called a *function*.

We will meet routines in detail later in this course, but for now we will introduce a few useful intrinsic routines for working with numbers. The following table contains *intrinsics* that you will use often in your Fortran programs.

Intrinsic	Description	Example
abs	Find absolute value of a number	abs(-5.0) gives 5.0
ceiling	Round number up toward positive infinity	ceiling(–5.5) gives –5.0
cmplx	Convert reals to complex	cmplx(5.0, –5.0) is 5.0–5.0i
conjg	Take complex conjugate	conjg(c)
cos	Cosine of angle in radians	cos(3.14159) is around -1.0
exp	Exponential of number	exp(1.0)

Intrinsic	Description	Example
floor	Round number down toward negative infinity	floor(–5.5) is –6.0
huge	The biggest number possible of the number kind passed	huge(0.0d0) is the biggest double precision number
int	Convert to integer by truncating	int(4.99) is 4
max	Find maximum of two or more numbers	max(–4,–5) is –4
min	Find minimum of two or more numbers	min(–4,–5,–6) is –6
mod	Remainder after first number is divided by the second	mod(8,5) is 3
nint	Round real to nearest integer	nint(5.5) is 6
random_number	Return a random real between 0 and 1	call random_number(x) puts random value into variable x
real	Convert integer to real	real(5) is 5.0
sign	Applies sign of second number to the first number	sign(–5,1) is 5
sin	Sine of angle in radians	sin(3.14159) is around 0.0
sqrt	Square root of number	sqrt(4.0) is 2.0
tan	Tangent of angle in radians	tan(pi/4.0) is around 1.0

There are many more too, including intrinsics to query every aspect of machine precision (*eg*, digits in coefficient, maximum exponent), but the ones listed above are amongst the most commonly used.

Exercise: Intrinsically Interesting

Below is the skeleton of a short program. For each print statement, fill in an appropriate intrinsic routine, and compile and run the program.

```
real r1, r2
r1 = 5.02
r2 = -10.50
print *, 'The value of r1, with the sign of r2 is ', ! Replace this comment
print *, 'The maximum of r1 and r2 is ', ! Replace this comment
print *, 'The closest integer to r1 is ', ! Replace this comment
print *, 'The value of r1 without fractional component is ', ! Replace
print *, 'The value of r2 rounded up is ', ! Replace
end
```

2.9. Implicit Typing

We have already discussed how you declare a variable in Fortran. In all the source code you have seen so far, each variable has been declared explicitly, but Fortran gives you a means of declaring your variables *implicitly*. If you have a line like the following in your program or routine, the type of variables will be determined by the first letter in their name:

```
implicit real*8 (a-h,o-z)
```

This line says that any variable beginning with the letters a through h, or o through z, should have the type real*8. (real*8 is a real number that fits in 8 bytes of memory.) Names starting with i through n are treated as integers, unless explicitly declared.

This may seem like a time-saving idea, and you will see many old Fortran programs using it, but it tends to lead to many bugs, and doesn't play well with modern forms of programming. For these reasons, *you are discouraged from using implicit typing*. Instead, you should add the line

```
implicit none
```

to your programs and routines. This tells the compiler that all variables must be declared explicitly, and it is an error if they are not. This way, you will know you when you forget to declare the type of a variable, and your code should have less bugs.

Exercise: Implicit vs Explicit

Consider this short program.

```
implicit real (a-h,o-z)
bab = 1.0
aba = 2.2
print *, bab * aba
end
```

What would be printed by this program? If you are not sure, type it into a file (*eg*, test.f90), compile it with gfortran, and run the executable.

Now change 'aba' in the print statement to just 'ab'. What will be printed now? Again, if you don't know, compile and run the code.

What does this tell you about the potential for bugs in code with implicit typing?

Finally, change the first line of the program to 'implicit none'. Try to compile, and see what happens. Now define the variables explicitly and recompile. Try to change 'aba' to 'ab' as before. What happens when you try to compile?

2.10. Character Strings

To this point, we have only dealt with variables of numerical types, but Fortran also has support for character strings. Character strings behave somewhat differently than other types, and have their own set of operators and intrinsic routines.

You can declare and assign a character string variable like this

```
character(20) a
a = 'Here is a string'
```

```
end
```

The number '20' is the total number of characters in the string. Literal strings, like the short sentence on the right-hand side of the assignment, should be enclosed in double or single quotation marks. If you use single quotation marks, you can freely include double quotation marks in the string itself, and *vice versa*. So this is legal Fortran

```
character(50) a
a = 'The word "hello" is used as a greeting'
print *, a
end
```

If you want to have the same type of quotation mark in the string, just include two of them, like this

```
a = 'The word ''hello'' is used as a greeting'
```

In addition to the assignment operator, =, which can be used to assign a string variable to another string as above, there is another useful string operator: the concatenation operator //.

```
character(20) a_string
a_string = 'Hi '
a_string = a_string // 'there'
```

This will result in `a_string` containing 'Hi there'. The concatenation operator is used to join — or *concatenate* — two strings together.

Just as for numerical types, Fortran offers a number of intrinsic routines for working with strings. Here is a table with some of the more useful ones.

Intrinsic	Description	Example
index	Find location of a substring	index('hi there', 'there') gives 4
len	The total length of a string	len('hi there') gives 8
len_trim	The length with whitespace removed from the front and back	len_trim(' hi there ') gives 8
repeat	Concatenate multiple copies of a string together	repeat('-', 4) gives '----'
scan	Find first location a set of characters in a string	scan('hello', 'oe') gives 2
trim	Trim whitespace from the front and back of a string	trim(' hi ') gives 'hi'

The intrinsics you will probably use the most are `trim` and `len`. You often need these functions because string variables have a fixed size, and will usually contain a lot of whitespace. For example, take this code

```
character(32) :: line
line = 'Is this the real life?'
end
```

You may think that the variable `line` would be equal to the string 'Is this the real life?', but this is only partially true: It is actually equal to 'Is this the real life?   ', that is to say, whitespace is added to the end to make up the total 32 character length. If you print out `line`, this whitespace will also be printed, and if you use the `len` intrinsic, it will return 32. You can use `len_trim` to get the length with the whitespace removed, and `trim` if you want to get back the original string.

```
character(32) :: line
line = 'Is this the real life?'
print *, 'The length with whitespace is ', len(line)
print *, 'The length without whitespace is ', len_trim(line)
print *, 'The line with whitespace is "', line, '"'
print *, 'The line is "', trim(line), '"'
end
```

This discussion of whitespace in Fortran strings leads to another interesting question: When are two strings considered equal? In the examples above, we assign a string variable to the string 'Is this the real life?', but we also saw that extra whitespace was added to *pad* the string variable to its correct length. So is the variable still considered equal to the string 'Is this the real life?'?

The answer is: yes. In Fortran, any whitespace at the end of a string is ignored when the comparison operator (== or .eq.) is used. So even if two strings are of different lengths, they may still be considered equal in a Fortran program.

The intrinsic routines above, together with the concatenation operator, allow you to do some basic string manipulation, but what about when you want to extract part of a string? For that, you can use *slicing*, which — as we will see later in the course — is also an important feature of Fortran for working with arrays of data.

Slicing involves stipulating the range of indexes of characters in a string variable that you would like to extract. For example, take this code

```
character(20) :: string
string = 'hi there'
print *, string(4:8)
end
```

The print statement will only print the characters 4 through 8 of the `string` variable, *ie*, the string 'there'. The slide notation includes the index of the first character, followed by a colon, and the index of the last character. You can also exclude either index, in which case the slice will begin at the start, if the lower bound is excluded, or go to the end, if the upper bound is excluded. The following example demonstrates this:

```
character(20) :: string
string = 'hi there'
print *, string(:)   ! This prints the whole string
print *, string(:2)  ! This prints 'hi'
print *, string(4:)  ! This prints 'there' with extra whitespace
end
```

You can also use slicing to change a substring, using the assignment operator, =. This example replaces 'hi' with 'my'

```
character(20) :: string
string = 'hi there'
string(:2) = 'my'
```

```
print *, trim(string)   ! Prints 'my there'
end
```

The string you assign the slice to should fit in the slice, or it will be truncated. Fortran will not make room for the substring by moving characters that are come after the slice.

Exercise: Working with Whitespace

Write a short program that stores the string 'The quick brown fox' in a character string variable with length 32. Print out the string with and without whitespace at the end, and the length of the string with and without whitespace.

Exercise: Equality of Strings

In this exercise, we are going to test when Fortran considers two strings to be equal. Begin by entering this short program.

```
print *, 'hi' .eq. 'hi'
print *, ' hi ' == 'hi'
print *, 'hi ' == 'hi'
print *, 'hi  ' == 'hi        '
end
```

Try to predict what each print statement will output. Then compile and run the program.

What does this tell you about how Fortran treats whitespace at the end of a string in comparisons? What about at the beginning of a string?

Exercise: Slicing Strings

Consider the following program

```
character(20) string
string = 'a stitch in time saves nine'
end
```

Add a print statement to this program that prints out the words from the string variable in a different order, using slices. For example, print out 'time saves a stitch in nine'.

2.11. Control Flow

The programs we have encountered to this point in the course have been simple and sequential. 'Simple' because they are very short, and 'sequential' because statements get executed in the order they appear. However, programs in the real world are very different: execution may jump from one section to another, *loop* back on itself, or choose between two or more different *branches*. Programs typically have many non-sequential jumps, and the coming sections are all about how you control where a program will go next.

The term that summarizes all of this is *control flow*. This term covers the various ways you can influence the flow of execution of a program. There are two basic ways to do this. The first is that you can test a condition and choose what action to take next based on the result. To use a real world analogy, you might test whether a traffic light is red or green by looking at it, stopping if it is red, and going if it is green. The second basic form of control flow is looping: repeating some action multiple times. For example, you may want to add up 1000 different numbers; you could write 1000 lines in your program, each with an addition, or you could use a loop construction to repeat one line 1000 times.

The next two sections discuss these basic forms of controlling the flow of execution.

2.12. Conditionals

Fortran supports a few different constructs for choosing a code branch based on some condition. The most widely used is the if/then/else construct. In its simplest form, it provides a means for skipping a block of code if conditions are not right.

```
if ( x < 0 ) then
   x = 2 * x
endif
```

This tests if the variable `x` is less than zero, and executes the statements in the block if it is, which results in `x` being multiplied by 2. If `x` is greater or equal to zero, the block is not executed, and `x` does not get multiplied by 2. The if-block is closed by the keyword `endif`.

The whitespace used in Fortran code is irrelevant to the compiler, so the spacing you use, and the indentation, is up to you. However, it is a good idea to be consistent, and to write your code so that it is easily read by other programmers. For example, spaces are used in the code above to make the expressions more readable, and the code inside the if-block is indented by 3 spaces, again to make it clear to any programmer reading it what expressions are affected by the if statement.

It is also possible for simple conditions, such as the one above, to by written as a single line if-statement, which excludes the `then` and `endif` keywords.

```
if ( x < 0 ) x = 2 * x
```

if-statements can have multiple branches, each with a separate condition attached. These branches are tested in order until a condition evaluates to true, at which point the statements in the branch are executed, and the rest of the if-block skipped. Take this example.

```
if ( x < 0 ) then
   x = 2 * x
elseif ( x == 0 ) then
   x = 3 * x
   y = y + 1
elseif ( (x > 0) .and. (x <= 5) ) then
   x = -2 * x
else
   x = 0
endif
```

Each line containing `elseif` begins a new branch. If any of the conditions are met, the statements in the branch are executed, and then execution jumps to the end of the if-block. If the `if` and `elseif` conditions are all false, the `else` branch will be executed. The `else` is optional; if it is excluded, execution will just continue as usual, with all of the branches skipped.

It is perfectly reasonable to *nest* blocks of if statements. For example, you could do this:

```
if ( x < 0 ) then
   if ( x > -10 ) then
      x = 2 * x
   elseif ( x < -15 ) then
      x = 5 * x
   else
      print *, 'x is between -15 and -10'
   endif
else
   print *, 'x is zero or positive'
endif
```

In this case, one if-block is nested in another. The nested block will only be executed if the condition `x < 0` evaluates to true first.

Often, there are many ways to structure the same set of conditional branches. For example, the code above could be *refactored* to

```
if ( x < 0 .and. x > -10 ) then
   x = 2 * x
elseif ( x < -15 ) then
   x =  5 * x
elseif ( x < 0 ) then
   print *, 'x is between -15 and -10'
else
   print *, 'x is zero or positive'
endif
```

Often 'flatter' constructions like this are a bit easier to follow, but not always. You should try to use constructions that make the logic as clear as possible for other programmers.

Fortran includes a second type of conditional branching construct called a *select case* block. The select case construct takes an expression that evaluates to a string, integer, or logical, and chooses the branch that matches the value. Here is an example.

```
integer month
month = 5

select case (month)
case (5)
   print *,'It''s May'
case (12,1)
   print *,'Merry X-mas and a Happy New Year'
case (6:8)
   print *,'Beach Time!'
case default
   print *,'Business as Usual'
end select

end
```

Each case can have one or more indexes or ranges, separated by commas. The first case that matches is the one that gets executed; all others are skipped. If none match, the `case default` branch is executed, if it is included — it is optional.

In the example above, the first case simply tests if the `month` variable is equal to 5, and if it is, prints a message before jumping to the end of the select case block. The second tests will be executed if `month` is either 12 or 1. The third case has a range, so will be executed if `month` is in the range [6, 8]. Lastly, the default case will be used if no other case matches.

Exercise: Conditional Branching with if-blocks

Take a look at the following code.

```
real x
...
if ( x < 0.0 ) then
   if ( x > 1.0 ) then
      print *, 'Birch'
   endif
else
   if ( x > 5.5 ) then
      if ( x < 10.0 ) then
         print *, 'Pine'
      endif
   elseif ( x > 10.0 .and. x < 13.0 ) then
      print *, 'Gum'
   else
      print *, 'Redwood'
   endif
endif
end
```

If `x` is initialized to 10.5, what will get printed by this program? What about if `x` is 7.0? Under what circumstances will the word 'Birch' get printed?

Type or copy this code into a file. Replace the ellipsis with a line to assign the value of `x`. Now compile and run the program, and change the value of `x` to test your conclusions above, recompiling each time. Were your conclusions correct? If not, why not?

Exercise: Select Case Construct

Write a short program that takes the index of the current month, stored as an integer, and uses a select case statement to print out the name of the current season. Print an error message if the index is not in the range 1 to 12. Compile and run the program.

2.13. Loops

The second basic form of control flow is looping. Looping allows you to execute a series of statements many times, without having to duplicate them many times.

The most basic loop in Fortran is the do loop, which you use to perform a set number of *iterations*. Here is do loop that prints out some text 100 times.

```
integer i
do i = 1, 100
   print *, 'Some text'
enddo
```

A do loop begins with the keyword `do`, and ends with `enddo`. Anything in between is executed each iteration of the loop. The first line of the loop includes a *loop variable*, in this case `i`. This variable is first set to the first number in the range on the right hand side of the assignment, and is incremented by one each iteration until it is equal to the second number. After the loop has been executed with the variable equal to the maximum value in the range, it exits, and execution continues on after the `enddo`.

You can use the loop variable inside the loop, and it is very common to do so. For example, you could do this

```
integer i, n
n = 5
do i = 1, n
   if ( i > 3 ) then
      print *, 'i is greater than 3'
   else
      print *, 'i is less than or equal to 3'
   endif
enddo
```

This code would print the following output

```
i is less than or equal to 3
i is less than or equal to 3
i is less than or equal to 3
i is greater than 3
i is greater than 3
```

which correspond to the 5 values taken by `i`: 1, 2, 3, 4, and 5.

This is by far the most common form of a do loop, but there are other variations. For example, you can choose not only different lower and upper bounds on the range, but also the increment used, which can even be negative.

```
do i = 5, 2, -1
   print *, i
enddo

do i = 1, 100, 2
   print *, i
enddo
```

The first loop will print out the numbers 5, 4, 3, and 2, that is, from 5 to 2 in increments of -1. The second loop will print out 1 to 99, in steps of 2. Note that it is never actually exactly equal to the upper bound, 100, but stops when the index exceeds the upper bound.

A do loop usually runs through all its iterations before exiting, but if the program should exit the loop prematurely, for whatever reason, an `exit` statement can be used, which causes execution to jump outside the loop.

```
do i = 1, n
   x = 2.56 * i + x
   if ( x > 1000.0 ) exit
enddo
```

In this example, if during any iteration of the loop the value of the variable `x` exceeds 1000.0, the loop will exit before all `n` iterations have been completed.

You can even have loops with no loop variable, that continue forever: *infinite loops*. You may think this is not very useful, but they can come in handy when used with an `exit` statement. For example, you could have this

```
do
   x = x + 1
   if ( x > 1000 ) exit
enddo
```

When you write loops like this, that can potentially run forever, it is important that you make sure that the exit conditions will prevent an infinite loop. In general, it is better to use a loop variable, because this will always put a maximum bound on the number of iterations.

As with any of the other constructs we have seen so far, you can nest do loops. For example, you can create a *double loop* like this

```
integer i, j
real x
do i = 1, 10
   x = i * 2
   do j = 1, 2
      x = x + j * 3
      print *, x
   enddo
enddo
```

You use an exit statement when you want to terminate a loop altogether, but there is a keyword that can be used to skip the rest of the current iteration, and continue on with the next one: *cycle*.

```
integer i, j
iloop : do i = 2, 1000
   do j = 2, i-1
      if ( mod(i, j) == 0 ) cycle iloop
   enddo
   print *, i
enddo iloop
end
```

This little program actually prints out all prime numbers less than or equal to 1000. The cycle statement can be used much as the exit statement was earlier, in which case it applies to the immediate enclosing loop. But both the `exit` and `cycle` statements can also be used with *labels* to exit or cycle any enclosing loop. In this case, a label (`iloop`) is added to the outer loop, and the `cycle` applied to that.

The way it works is this: The outer loop iterates over all numbers from 2 to 1000. For a given value of `i`, a second, inner loop iterates over numbers from 2 up to 1 less than `i`. It tests each value to see if it divides exactly into `i`, using the modulus function, and if one is found the number is known not to be prime, and `cycle` is used to skip to the next number in the outer loop. Only if the inner loop completes entirely without finding a divisor will the print statement be executed.

There are other loop constructs available in Fortran, which tend to be used less frequently. One such construct is the do-while loop, which continues until a particular condition is no longer met. For example,

```
real r
r = 1.1
```

```
do while ( r < 1000.0 )
   r = r**2
enddo
print *, r
```

The loop in this example continues until the variable `r` is greater than or equal to 1000.0; in each iteration, `r` is raised to the power 2. In this case, it is obvious that the loop will eventually exit, but you need to be careful with do-while loops, because they can potentially become infinite loops.

More recent versions of Fortran support loops that are designed to work with arrays of data. In an upcoming chapter we will introduce arrays, and these loops will be discussed there.

Exercise: Summing Up

Write a short program that uses a do loop to add up the numbers between 19 and 37. You will need two variables, one for the loop, and the other to hold the sum of values.

Exercise: Loops that Don't Loop

One thing you might be wondering is what happens when a loop has no iterations. In this exercise, we are going to find out.

Enter this program in a file.

```
integer i
do i = 1, 0
   print *, 'In loop'
enddo
end
```

Compile and run it. What does it tell you?

Now change the upper bound to 1, and recompile and run. How many times does the print statement get executed?

3. Program Units

We have already encountered a few intrinsic (*ie*, built-in) routines in this course, but now we are going to see how you can write your own. *Routines*, or *procedures*, are blocks of code that you can jump to from some other place in your program, and jump back again upon completion. In that sense, procedures are a form of control flow.

3.1. Main Program

Before we look at procedures, we should consider a special part of any Fortran program: the *main program*. This is the piece of code that is run when the program starts up. In most of the short examples we have looked at so far, the code you compiled was the main program.

The main program begins with the optional *program* keyword followed by a program name, and is concluded by an *end* or *end program* statement.

```
program Main
   print *, 'Hello World'
end program
```

The `program` keyword and name are optional, so you can also write simply

```
print *, 'Hello World'
end
```

In the examples to date, we have been working with the latter form, but as our programs get bigger, you will start to see the former form being used more often.

3.2. Subroutines

Procedures fall into two categories in Fortran: *subroutines* and *functions*. Subroutines do not return a value, and functions do. Here is a simple subroutine by way of demonstration

```
subroutine AddAndPrintNumbers(a, b)
   real :: a, b
   print *, a + b
end subroutine
```

A subroutine begins with the keyword subroutine, followed by the subroutines name (`AddAndPrintNumbers`), and then a set of comma-delimited *argument* variables in parentheses. The subroutine is terminated by an either `end`, `end subroutine`, or `end subroutine` followed by the subroutine's name.

The arguments are defined at the beginning of the subroutine. In the example they are both real numbers. In addition to the arguments, *local variables* — which are only visible inside the subroutine — can also be used. For example, the code above could be rewritten as

```
subroutine AddAndPrintNumbers(a, b)
   real :: a, b
   real :: c
   c = a + b
   print *, c
end subroutine
```

where `c` is a local variable.

To make use of a subroutine, you need to *call* it from somewhere else in your program, like so

```
real :: a1, b1
a1 = 2.0
b1 = 3.0
call AddAndPrintNumbers(a1, b1)
end
```

Note that there is no correlation between the names of the variables passed to the subroutine in the call, and the variables as they are used inside the subroutine, though the values of the variables passed in are transferred to the variable arguments inside the subroutine. How this happens will be discussed in more detail a bit later.

The benefit of subroutines, and procedures in general, is that they allow you to repeatedly make use of a piece of code without having to duplicate it explicitly. For example, the simple subroutine `AddAndPrintNumbers` can be used many times over, with different combinations of numbers. Take a look at this program

```
integer :: i
real :: a1, b1
do i = 1, 100
   a1 = 2.0 * i
   b1 = 3.0 * i
   call AddAndPrintNumbers(a1, b1)
enddo
end
```

The *subroutine call* has been inserted into a do loop; Each iteration of the loop, a new set of inputs are calculated, and passed to the subroutine to be added up and printed out. This is repeated 100 times.

If a particular argument is only used for input to a subroutine, you can pass in literal values, or even the results of an expression evaluation. For example, you could do this

```
integer :: i
do i = 1, 100
   call AddAndPrintNumbers(2.0*i, 3.0)
enddo
end
```

This variation on the previous example uses an expression (`2.0*i`) for the first argument. This expression will be evaluated *before* the subroutine is called and the result passed in as the argument. The second argument is the literal real number `3.0`.

When you are passing a literal or expression as an argument, you need to be very careful that it has the right type — including the right kind — and that it doesn't get changed inside the procedure. For example, you should not pass a `real(4)` number to a subroutine that is expecting a `real(8)` number. If you do this, the results you get may be unexpected, or your program may crash.

Exercise: Square Subroutine

Write a short subroutine, `Square`, that takes a real number as argument, and squares it. In the same file, after the subroutine, add a program to call the `Square` routine, like this

```
    real x
    x = 5.0
    call Square(x)
    print *, x
    end
```

Compile and run.

What value does x have after the subroutine is called? What does this tell about the effects of changes made inside a subroutine for variables in the calling code?

3.3. Functions

In theory, you could write all of your procedures as subroutines, because they allow you to input *and* output data. Functions are similar, but they have a return value, which is like an extra output. You *could* do without functions altogether, but they can make your code a bit more readable.

The `function` keyword is used to begin a function, and they are concluded — just as for subroutines — with `end`, `end function`, or `end function` with the function name. Here's an example:

```
real function Double(value)
   real :: value
   Double = 2.0 * value
end function Double
```

In this case, the type of the function's output, or *return value*, is declared just before the `function` keyword. The return value can be set in the function using the functions name as if it were a variable. So, in this example, the variable `Double` is assigned to be twice `value`.

Unlike for subroutines, you don't use the `call` keyword to invoke a function; instead, you just use the function in an expression.

```
real Double
real x
x = Double(5.0)
print *, x
end
```

The return type of the Double function needs to be defined in the calling code. (You can avoid this by using something called an *explicit interface*, which will be discussed later in the course.)

There is some flexibility in how you write functions. For example, you can name the return variable something other than the function name by using a `result` clause.

```
function Double(value) result(doubleValue)
   real value, doubleValue
   doubleValue = 2.0 * value
end function
```

Using this approach, the return variable is declared after the arguments list in the `result` clause, and its type is declared in the main body of the function. The variable can be assigned as before, and its value will be returned to the caller.

Exercise: Square Function

Rewrite the subroutine from the previous exercise as a function, which takes a real number as argument, and returns the square of that number. Add the following program to test the function:

```
real x
x = 5.0
print *, Square(x)
print *, x
end
```

Compile and run.

What do you notice about the value of `x`, which is the second number printed? How does this differ from the way `x` was affected in the subroutine?

3.4. Argument Passing

We have seen a few times now that if you modify an argument in a procedure, the corresponding variable in the calling code will be similarly modified. This is stipulated in the Fortran standard, and it is different to some other languages, such as C.

The Fortran standard does not say how this behavior should be achieved, just that it must be so. Often, this behavior is referred to as *pass-by-reference*, but pass-by-reference is simply one way of producing the correct behavior. When pass-by-reference is used, the memory address of the variable is passed from the caller to the callee (*ie*, subroutine or function). The argument variable in the procedure shares the same memory address as the original variable, so when you modify the argument, it also modifies the variable passed in.

This approach is used in many instances by a Fortran compiler to achieve the expected behavior, but it is not the only possibility. In some instances, the compiler may choose to use something called *copy-in-copy-out*. In this case, when the procedure is called, the compiler copies the value of the variable passed in into a new memory location, and copies it back to the original location when the procedure exits. The net effect is the same as with pass-by-reference: the variable in the calling code is updated when the corresponding argument in the procedure gets updated.

3.5. Argument Intent

So far we have seen that you can pass values into and out of a procedure via its argument list, but sometimes you might want more control over how each argument is used — whether it is for passing a value in, passing a value out, or both. You can do this by indicating the *intent* of an argument in its declaration within the procedure.

Take a function that returns the square of an real number. The input argument is not intended to be changed, so it can be marked as *intent in*.

```
real function Squared(x)
   real, intent(in) :: x
   Squared = x**2
end
```

This tells the compiler — and any other programmers — that the argument may not be changed inside the procedure. Any attempt to do so should result in a compilation error.

The same applies to variables that are only intended to be used to pass values out, such as this subroutine equivalent of the function above:

```
subroutine Squared(x, xSquared)
   real, intent(in)  :: x
   real, intent(out) :: xSquared
   xSquared = x**2
end
```

The default intent is actually *inout*. This means a variable can be used to pass a value in and/or used to return a value. If you do not include any intent explicitly in the declaration, `inout` will be assumed.

Here is the subroutine `Squared` rewritten to work with only one argument, which is used to both pass in the value to be squared, and return the result.

```
subroutine Squared(x)
   real, intent(inout)  :: x
   x = x**2
end
```

3.6. Returning Early

When control reaches the end of a procedure, it returns to the calling code, but you can also exit a function or subroutine prematurely if you choose. The *return* keyword is used for this purpose:

```
real function SafeSqrt(x)
   real, intent(in) :: x
   if ( x < 0.0 ) then
      print *, 'Negative value in SafeSqrt function'
      SafeSqrt = 0.0
      return
   endif
   SafeSqrt = sqrt(x)
end function
```

This routine, which is designed to perform a square root safely by first checking for negative arguments, uses `return` to exit the function in case of an error. If the argument `x` is less than zero, a message is printed, the result set to zero, and the function returns. If `x` is greater than or equal to zero, the square root operation is carried out, and the function returns when it reaches the end.

3.7. Naming Conventions

In the early days, FORTRAN imposed some pretty severe restrictions on programmers. For example, *identifiers* — the names of variables, functions, subroutines, etc. — could only have up to 6 characters. This made writing large applications particularly challenging.

Modern Fortran imposes far fewer constraints, with identifiers now only limited to 31 characters. There is nothing in the standard that stipulates how you should name your identifiers, but in real world programming, it is good to have a convention, and to stick to it.

There are two standard conventions in widespread use today: *mixed-case* and *underscored*. The mixed case convention involves beginning each new word with a capital letter, and leaving all other characters lowercase. The case of the first letter is sometimes lowercase, and sometimes uppercase. In this course, we will use lowercase letters for variables (*eg*, `latentMunicipalPressure`), and uppercase letters for everything else (*eg*, `CalcLatentMunicipalPressure`).

The underscored convention uses only lowercase characters, with spaces between words replaced by underscores (*eg*, `latent_municipal_pressure`). We *will* only use the underscored notation for one particular class of variables in this course: parameters, *ie* constant variables, will be written in underscored notation using only uppercase characters (*eg*. `NUM_THREADS`).

Exercise: Factorials

Write a function that calculates n!, the factorial of n, where n is an argument to the function. Use the mixed-case naming convention, and make the intent of any arguments explicit. Add tests to handle any illegal values of n, and report an error and return if such an argument is passed in. Write a short program to call this function, and print out 0!, 1!, 2!, ... , 10!. Compile and run your program.

3.8. Modules

Modules are a language construct for grouping variables and procedures together, and controlling access to them from the rest of the program. Here is an example

```
module Stuff
   implicit none

   save

   integer                     :: numThings = 0
   integer, parameter, private :: MAX_THINGS = 100

   private                     :: PrintNumThings

contains

   subroutine IncrementThings
      if ( numThings == MAX_THINGS ) stop 'Too many things'
      numThings = numThings + 1
   end subroutine

   subroutine PrintNumThings
      print *, numThings
   end subroutine

end module
```

There is a lot to cover here. Firstly, a module is broken into two sections: before the *contains* keyword, where you can declare variables, and after `contains`, where procedures can be included. The procedures have access to all of the variables declared in the module's data declaration section.

As we have already seen, it is good practice to use `implicit none` to prevent bugs caused by accidentally forgetting to declare variables; by adding an `implicit none` to the beginning of the module, it automatically applies to all variables and procedures in the module.

It is also good practice to enter the *save* keyword in your modules. This effectively allows the variables declared there to be shared between different parts of a program, such that when a module variable is set to a particular value in one spot, the value can be used from another. If you don't add the `save` attribute, `nosave` is assumed, which doesn't guarantee that this sharing of data can take place.

Modules need to be *used* before the variables and procedures they contain are accessible. A *use* statement achieves this:

```
program TestStuff
   use Stuff
   call IncrementThings
end program
```

`use` statements must appear right at the beginning of a program unit. Any variables or procedures declared in a module that are *public* can be accessed from a program unit that is using a module. Variables and subroutines are usually public by default, but as the example above shows, can be made *private* by either adding an attribute in a variable declaration (*eg.*, `MAX_THINGS`), or using the `private` keyword (*eg*, `PrintNumThings`) to restrict access to a particular procedure. A private member of a module is only accessible from inside the module itself.

As stated, the default *accessibility* is `public`, but you can change this default for any given module by simply entering the keyword `private` along on a line in the data declaration section. For example, the module above could be rewritten like this

```
module Stuff
   implicit none

   save
   private

   integer, public              :: numThings = 0
   integer, parameter           :: MAX_THINGS = 100

   public                       :: IncrementThings

contains

   subroutine IncrementThings
      if ( numThings == MAX_THINGS ) stop 'Too many things'
      numThings = numThings + 1
   end subroutine

   subroutine PrintNumThings
      print *, numThings
   end subroutine

end module
```

With the default set to `private`, it is necessary to explicitly indicate which members of the module should be accessible using the `public` keyword. The practice of setting the default accessibility to `private` is actually a good one, because — as we will see later in the course — *hiding* data from outside use can lead to more robust and easy to understand software.

3.9. Interfaces

Modules are useful for grouping together related variables and procedures, but if that was all they were for, they wouldn't be *that* useful. Modules actually perform a second important function in Fortran: they define *interfaces*.

An *interface* is all of the information that is needed to call a procedure, such as its name, type (*ie*, function or subroutine), and arguments, including all associated properties (*eg*, type, kind, intent).

Originally, Fortran only supported *implicit interfaces*. What this basically means is that the information discussed above was not available to the compiler to check. So if you called a subroutine and accidentally passed a real argument instead of an integer, the compiler would not see the error, and your program would contain a bug.

Fortran 90 introduced *explicit interfaces*, which allow the programmer to tell the compiler all of the details of a procedure's interface. This means the compiler can check that the routine is being called in the right way, and also allows extra information to be transferred to a subroutine. (The latter will be discussed in more detail in the section on arrays.)

The easiest way to make a procedure's interface explicit is to include the procedure in a module. The compiler will then automatically extract its interface, and usually store it in a file with the extension .mod. The .mod file will be read whenever the module is used, allowing the compiler to check the interface against the calling code, and flag any errors.

For this reason, it's good practice — where possible — to add all procedures to modules. That way, your interfaces will be explicit, and you can be sure to avoid certain bugs. In addition, extra information will be passed into the procedure, which — as will be discussed later — can be particularly useful when working with arrays of data.

3.10. Overloading Procedures

It can sometimes be useful to be able to refer to different procedures by the same name. The procedures may do basically the same thing, but take different arguments. If your subroutines have explicit interfaces, you can use a technique known as *overloading* to do this. Here's what it looks like.

```fortran
module NumbersModule
   implicit none
   save
   private
   public AddNumber, PrintNumbers

   interface AddNumber
      module procedure AddInteger
      module procedure AddReal
   end interface

   integer :: i = 0
   real    :: r = 0.0

contains

   subroutine AddInteger(arg)
      integer, intent(in) :: arg
      i = i + arg
   end subroutine

   subroutine AddReal(arg)
      real, intent(in) :: arg
      r = r + arg
   end subroutine

   subroutine PrintNumbers()
      print *, 'Real is ', r
      print *, 'Integer is ', i
   end subroutine

end module

program TestProg
   use NumbersModule
   call PrintNumbers()
   call AddNumber(5)
   call AddNumber(7.0)
   call PrintNumbers()
end program
```

If you run this program, you get this output

```
 Real is    0.0000000
 Integer is            0
 Real is    7.0000000
 Integer is            5
```

The part of the module responsible for the procedure overloading is this *interface block*

```
   interface AddNumber
      module procedure AddInteger
      module procedure AddReal
   end interface
```

The interface block, which must appear in the data declaration part of the module (*ie*, before `contains`), tells the compiler that the two subroutines `AddInteger` and `AddReal` can also be called using the name `AddNumber`. If you call `AddNumber`, as in the main program, the compiler will look at the arguments that you pass to determine which of the two specific subroutines you want to execute. This means that the overloaded subroutines must have different argument types, otherwise the compiler will not be able to *resolve* the call, and will give an error.

When you call an overloaded procedure, you can use either its specific name or its generic name (*ie*, overloaded name); however, it is good practice to force users of your modules to use the generic name by making the specific names (*eg*, `AddReal`, `AddInteger`) private to the module. That is what has been done in the example above: only the generic name `AddNumber` is public.

There is some flexibility in how you write the interface block. You can put all of the names on one line, delimited by commas

```
   interface AddNumber
      module procedure AddInteger, AddReal
   end interface
```

and you can even spread the subroutines and interface blocks over several modules. So you could split the module above like this

```
module IntegerNumbersModule
   implicit none
   save
   private
   public AddNumber

   interface AddNumber
      module procedure AddInteger
   end interface

   integer :: i = 0

contains

   subroutine AddInteger(arg)
      integer, intent(in) :: arg
      i = i + arg
   end subroutine

end module


module RealNumbersModule
   implicit none
   save
   private
   public AddNumber

   interface AddNumber
      module procedure AddReal
   end interface

   real :: r = 0.0

contains

   subroutine AddReal(arg)
      real, intent(in) :: arg
      r = r + arg
   end subroutine

end module
```

If a program uses both of these modules, and makes a call to `AddNumber`, the compiler will search through each interface to determine which specific subroutine should be called.

To finish off this section, it should be noted that you can also overload built-in operators like +, –, *, /, ==, and even =. To overload an operator, you use an interface block like this

```
interface operator(+)
   module procedure AddThings
end interface
```

The `AddThings` routine would need to be a function that takes two arguments, and returns the result of adding them together.

The assignment operator, =, can similarly be overloaded, but for that you use this form of interface block

```
interface assignment(=)
   module procedure AssignThings
end interface
```

`AssignThings` would need to be a subroutine that takes two arguments, and assigns the first to the second. We'll see how you write such a subroutine in the next section.

Exercise: Modules and Overloading

Write a module that contains two subroutines, one called `PrintInteger`, and the other called `PrintReal`. `PrintInteger` should take an integer argument, and `PrintReal` should take a real, and each should simply print the value passed with a message like 'The real number is ...'. Write a simple main program to use the module and call each of the subroutines with some arbitrary numbers.

Now use overloading to make each subroutine callable by the name `PrintNumber`. Make the specific names of the subroutines private, and adapt the main program to use the new `PrintNumber` call. Compile and run.

3.11. Optional Arguments

If a procedure has an explicit interface, it is possible to make some of its arguments *optional*. When an argument is optional, the caller can choose to exclude it.

```
module Mod
contains
   subroutine RoutineWithOpt(nonOpt, opt)
      integer :: nonOpt
      integer, optional :: opt
      if ( present(opt) ) then
         print *, 'optional argument was present'
      else
         print *, 'optional argument was not present'
      endif
   end subroutine
end module

program Main
   use Mod
   call RoutineWithOpt(1,1)
   call RoutineWithOpt(1)
end program
```

The keyword `optional` is used to indicate an optional argument, and the intrinsic function `present` can be used to determine whether or not a particular optional argument was passed in. The output of the example above is

```
 optional argument was present
 optional argument was not present
```

The first call to `RoutineWithOpt` includes the optional argument, and generates the first line of output; the second call does not, and results in the second line.

You can have more than one optional argument, and they can be mixed with non-optional arguments, but the caller of the procedure must use labels for all arguments after the first omitted argument, like this

```
module Mod
contains
   subroutine RoutineWithOpt(nonOpt1, opt, nonOpt2)
      integer :: nonOpt1, nonOpt2
      integer, optional :: opt
      if ( present(opt) ) then
         print *, 'optional argument was present'
      else
         print *, 'optional argument was not present'
      endif
   end subroutine
end module

program Main
   use Mod
   call RoutineWithOpt(1,1,1)
   call RoutineWithOpt(1,nonOpt2=1)
end program
```

In the second call to `RoutineWithOpt`, the label `nonOpt2` must be used to make clear that `opt` has been omitted. In the first call, no labels are needed, because all arguments are passed in.

3.12. User-Defined Types

In addition to the standard numerical types and character strings built into Fortran, you can also create your own *user-defined types.* User-defined types provide a mechanism for grouping together related variables, similar to the way a subroutine groups together related operations, or a module groups together related procedures.

You'll usually declare a user-defined type in a module, so that it can be shared between different parts of a program. Here's how you do that

```
module MonkeyModule
   implicit none
   save
   private
   public Monkey

   type Monkey
      character(32) :: name
      real          :: height
      logical       :: isMale
   end type

end module
```

The user-defined type `Monkey` contains a number of variables that are attributes of a monkey, like its name and height. Whenever you create a `Monkey` variable, a process known as *instantiation*, it will automatically get all of the variables declared in the `Monkey` type.

```
program MonkeyTest
   use MonkeyModule
   implicit none
   type (Monkey) :: m
   m%name = 'Bo Bo'
   m%height = 1.25
   m%isMale = .true.
end program
```

To access the variables, you use the % operator. In this example, the variable `m` is a `Monkey`, and you can access its name with `m%name`; `m%name` can be used just like any other character string variable.

Variables of user-defined types are not really any different to variables of built-in types. You can pass them to subroutines, add them to modules, and so on. The only catch is that the type declaration must be accessible wherever a variable of that type appears. The easiest way to ensure this is the case is to declare your types in modules.

Here's an extended version of the module above, with a subroutine that makes use of the `Monkey` type.

```
module MonkeyModule
   implicit none
   save
   private
   public Monkey, Grow

   type Monkey
      character(32) :: name
      real          :: height
      logical       :: isMale
   end type

contains

   subroutine Grow(theMonkey)
      type (Monkey), intent(inout)  :: theMonkey
      theMonkey%height = theMonkey%height * 2
   end subroutine

end module
```

The `Grow` subroutine simply doubles the `height` variable of the `Monkey` passed in via the argument.

You may be surprised to learn that you can directly assign two variables of particular user-defined type:

```
program MonkeyTest
   use MonkeyModule
   implicit none
   type (Monkey) :: m1, m2
   m1%name = 'Bo Bo'
   m1%height = 1.25
   m1%isMale = .true.
   m2 = m1
   print *, 'The height of m2 is ', m2%height
end program
```

When run, this program prints out

```
 The height of m2 is    1.2500000
```

All of the variables in the `Monkey` variable `m1` are copied into the `m2` variable by the line

```
   m2 = m1
```

If you need to do something special during the assignment, you can overload the assignment operator, as discussed in the previous section.

```
module MonkeyModule
   implicit none
   save
   private
   public Monkey, assignment(=)

   type Monkey
      character(32) :: name
      real          :: height
      logical       :: isMale
   end type

   interface assignment(=)
      module procedure AssignMonkeys
   end interface

contains

   subroutine AssignMonkeys(lhs, rhs)
      type (Monkey), intent(inout)  :: lhs
      type (Monkey), intent(in)     :: rhs
      print *, 'Assigning monkeys'
      lhs%name = rhs%name
      lhs%height = rhs%height
      lhs%isMale = rhs%isMale
   end subroutine

end module
```

In this rather contrived example, a message is printed every time a `Monkey` is assigned to another `Monkey`, in addition to all variables being explicitly copied.

One question that remains open is whether it is possible to embed literal values of user defined types in a program, in the same way that you can simply enter 5.0 to represent a real number with the value 5.0. It turns out you can:

```
program MonkeyTest
   use MonkeyModule
   implicit none
   type (Monkey) :: m1, m2
   m1 = Monkey('Bo Bo', 1.25, .true.)
   m2 = m1
   print *, 'The height of m2 is ', m2%height
end program
```

In this program, rather than assigning each variable in the type separately, a *constructor* has been used for the `Monkey` type. The values that are assigned to each variable in the `Monkey` are passed in order to the constructor, which looks like a function call, but is not. Using a constructor like this can save some space, and will also allow the compiler to warn you if you forget to initialize one of the variables.

Exercise: User-Defined Types

Write a module containing a user-defined type called `Person`. A `Person` should have a name, height, and weight. Add a function to the module called `IsOverweight` that performs a simple calculation to determine if a given `Person` — passed in as the only argument — is overweight. (The actual calculation is not so important. One option would be to divide height in centimeters by weight in kilograms, and if the ratio is less than 2 the person is considered overweight.)

Write a main program to initialize a person using a constructor, and print out if they are overweight. Compile and run.

4. Working with Arrays

We have already dealt with numerical types and character strings in Fortran, but a lot of scientific software involves operations on whole blocks of similarly typed data. These blocks are known as *arrays*. Fortran has some of the most powerful array programming facilities of any programming language — it's one of the language's great strengths. In this section you will be introduced to arrays, and the numerous things you can do with them in Fortran.

4.1. Declaring Array Variables

There are many different ways of declaring array variables in Fortran. We will only cover the most conventional approaches as used in modern programs.

The first is to include the *array bounds*, which is the range of indexes in the array, in parentheses after the variable name, like so

```
real a(10)
```

This line declares an array variable called 'a', with space for 10 real numbers. You access the *elements* in an array using an index; indexes begin by default at 1. For example, to set the third element of `a` to 5.0, you could write this

```
a(3) = 5.0
```

Arrays don't have to be one dimensional, as above — they can be *multidimensional*. Multidimensional arrays have more than one index.

```
integer b(5,3)
b(1,1) = 9
b(5,3) = 2
b(4,3) = b(1,1) + b(5,3)
```

In an array declaration, each dimension has *lower* and *upper bounds*. The lower bound defaults to 1, but you can override this to impose your own set of indexes. For example, you could have the first index begin at 5 and end at 10, like this

```
integer c(5:10, 3)
```

The first index of this array can take the values 5, 6, 7, 8, 9, and 10; the second index can take the values 1, 2, and 3. The first colon separates the lower and upper bounds on the index range.

If you have several arrays with the same bounds and *rank* — the number of dimensions — you may find it useful to use the `dimension` keyword to avoid having to repeat the shape declaration. Here's how it works:

```
real, dimension(4:9, 2:3)  :: a, b, c
```

In this case, all three arrays will end up with the index ranges declared in parentheses after the `dimension` keyword.

4.2. Initializing Arrays

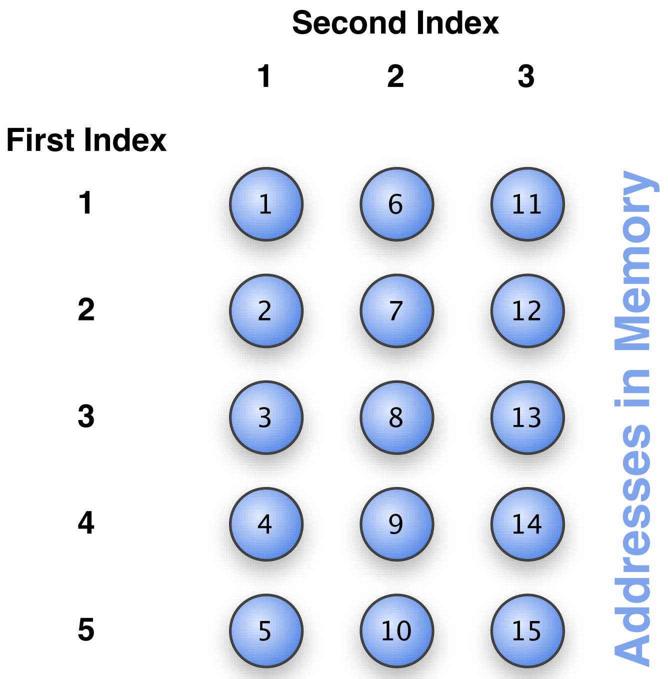
You can initialize arrays using one or more loops. For example, we could initialize all elements of an array to zero like this

```
real r(10,5)
integer i, j
do i = 1, 5
   do j = 1, 10
      r(j,i) = 0.0
   enddo
enddo
```

You may be wondering why the loops are ordered the way that they are. Is that choice subjective, or is there some rationale behind it?

You can choose any order you like, but one particular order will usually make for better performance of your program, and this applies not only to initialization of arrays, but any operations on arrays. *You should try to have the first index of your array correspond to the inner loop*, with the second index corresponding to the next loop out, and so on, such that the last index corresponds to the outer loop.

Why is loop order of any importance? This has to do with the order in which array elements are stored in the memory of a computer. In Fortran — unlike the popular C programming language — elements are stored in *column major order*. This means that elements along the columns of a two-dimensional array (*ie*, matrix) are consecutive in memory. To be more concrete, in the example above, the element `r(5,4)` is just before `r(6,4)`, but is not next to element `r(5,5)`. The accompanying figure may help you visualize this.



Memory addresses of the elements of a 5 by 3 array. Elements along each column are in consecutive memory addresses.

When you loop over an array, you should try to do it in small steps through memory, because this allows for the most optimal use of the computer's memory architecture. A modern computer brings chunks of memory into its *cache* for the CPU to operate on; if the array elements in a series of operations are close together in memory, the CPU can perform many operations before it has to write the results back out to main memory.

If, on the other hand, the operations require big strides through memory, the computer will bring in a chunk of memory — technically called a *cache line* — but will only be able to operate on one or a few elements before having to write the chunk back out and get a new one. Retrieving and storing cache lines is a relatively time consuming operation on a modern computer.

Returning to the subject at hand, you can also initialize arrays by assigning them to *literal arrays*, but this is only really convenient for smaller arrays.

```
real a(5)
a = (/1, 5, 2, 6, 7/)
```

The `(/` operator delineates the beginning of the array, and `/)` the end.

If you have more than just a few elements to initialize, you can use a loop, or an *implied do loop*. These are do loops that can be written very compactly, like this

```
real b(6)
b = (/ (2.0 * i, i=1,6) /)
print *, b
```

The implied do loop consists of contents of parentheses inside the literal array operators: the expression `2.0 * i` is evaluated for each value of i, with `i` looping from 1 to 6.

Exercise: Column-Major Order

Write a program that declares a two-dimensional integer array, `a`, with dimensions 5 by 5. To initialize the elements, write a double do loop, and set each element to its column number, *ie*, the second array index. Add a print statement that prints out the whole array at the end, like this

```
print *, a
```

Compile and run. What does the output of the program tell you about the order of elements in the computer's memory?

Exercise: Nested Loops

You have been given the following piece of source code.

```
integer i,j
real a(1000,1000)
...
do i = 2,1000
   do j = 2,1000
      a(i,j) = a(i-1,j-1) * a(i,j)
   enddo
enddo
```

What is a simple change that you could make to this code that may make it run faster?

Exercise: Initializing Arrays

Declare two one-dimensional real arrays of size 5. Use an implied do loop to initialize all the elements of the first one to 0.1. Use an assignment to a literal array to initialize the elements of the other array to the values 10, 20, 30, 40, and 50. Write a do loop to multiply the corresponding elements of each array together, and print out the resulting values. Compile and run.

Exercise: Finding the Maximum Element of an Array

Declare a real array, `a`, of size 100, and initialize it using the intrinsic subroutine `random_number`, like this

```
call random_number(a)
```

Now, write a do loop to find the maximum element of the array. Print out the maximum element, and its index, after the loop. Compile and run.

Hint: To do this, you will need several extra variables: one to store the current maximum element, and one for the index of the maximum element. Use an if statement inside the do loop to update these variables each iteration as necessary.

4.3. Element-wise Operations

More recent versions of Fortran support array operations, which can eliminate many loops. For example, you can assign an array to a constant or another array of the same shape, and all elements will be assigned.

```
real, dimension(10) :: a, b
a = 0.1
b = a
```

Assigning an array to *scalar*, as in the second line above, causes all elements of the array to be set to that scalar value. It is similar to writing this loop

```
do i = 1, 10
   a(i) = 0.1
enddo
```

When you assign an array to another array of the same *shape*, each element of one is assigned to its corresponding element in the other array, similar to this loop

```
do i = 1, 10
   b(i) = a(i)
enddo
```

This works for both one-dimensional and multi-dimensional arrays, so you can also do this

```
real a(2,2), b(5:6, 5:6)
call random_number(a)
b = a
```

In this example, the arrays a and b have the same shape, but different indexes. In *element-wise operations* the indexes are actually not important — the positions of the elements in the array are the important thing. In this case, both arrays are 2 by 2 in size, so you can use an array assignment. The end result will be equivalent to the following set of scalar operations:

```
a(1,1) = b(5,5)
a(1,2) = b(5,6)
a(2,1) = b(6,5)
```

```
a(2,2) = b(6,6)
```

Element-wise operations are not restricted to assignments — any arithmetic operator will work. For example, to add up corresponding elements of two arrays, and put the result in a third, you could do this

```
real, dimension(10) :: a, b, c
a = 0.1
call random_number(b)
c = a + b
```

And mathematical intrinsic routines will also work, like `sqrt`:

```
real :: a(10,10), b(10,10)
call random_number(a)
call random_number(b)
a = b**2 - 2.0 * sqrt(a)
```

Note that you can also use scalars in the expressions; when a scalar appears, it behaves as if it were an array with all elements equal to the scalar value.

Exercise: Element-wise Ops

Write a short program that declares two one-dimensional real arrays, one with indexes ranging from 1 to 50, and the other with indexes ranging from 51 to 100. Initialize the elements of the first array, using an implicit loop, to the numbers 0.1, 0.2, 0.3, ... , 5.0. Now use an element-wise operation to assign each element of the second array to the square root of the corresponding element in the first array, multiplied by 3. Print the second array. Compile and run the program.

4.4. Array Slices

When looking at character strings earlier in the course, we discussed how they could be *sliced*. Slicing involves extracting a sub-string that covers a particular range of indexes. Arrays have this same capability; you can use a range of indexes to extract a portion of an array. For example, you could extract a row from a matrix, or the first half of a vector. You can even use a stride to extract every second element of an array.

The syntax for array slices is similar to how you declare index ranges.

```
real :: r(10)
r = 0.6
print *, r(2:6)
end
```

This short program initializes the array `r`, and then prints out the elements from indexes 2 to 6, using an array slice.

Array slices are just like any other array, and can be used in expressions.

```
integer, dimension(3) :: a, b
a = (/1,0,0/)
b = (/0,1,0/)
a(2:3) = 2 * b(::2)
print *, a
end
```

This rather convoluted little program actually leaves the array a unchanged. Why? The expression

```
a(2:3) = 2 * b(::2)
```

sets the second and third elements of a to twice the first and third elements of b, which are zero. The slice ::2 is the same as 1:3:2 — when an index is excluded, it takes its default value. The default value of the first slice index is the lower bound of the array, and the second slice index defaults to the upper bound.

Array slices work equally well with multi-dimensional arrays.

```
real :: a(10,10), b(10), c(5,5)
call random_number(a)
b = a(3,:)
c = a(3:7,3:7)
a(:,1) = 0.5
end
```

In the three assignments above, the first assigns a rank one array (b) to the third row of a rank two array (a); the second assignment assigns a 5 by 5 array (c) to a sub-matrix of a; and the final assignment sets all elements in the first column of a to 0.5.

Fortran offers another means of deriving a new array from an existing one. You can use an index array — array of integers representing indexes — to extract elements. For example, take the case of reordering some elements of an array.

```
real    :: a(5) = (/1.,2.,3.,4.,5./)
real    :: b(5)
integer :: ia(5) = (/2,1,3,5,4/)
b = a(ia)
a( (/1,3,5/) ) = 10.0
print *, 'a is ', a
print *, 'b is ', b
end
```

This shows how you can use an array of indexes to reference a part of an array, and even reorder the elements. The array b is first set to a permutation of the array a, and then three elements of a are set to 10.0. The output is

```
 a is   10.0000000       2.0000000      10.0000000       4.0000000      10.0000000
 b is    2.0000000      1.00000000       3.0000000       5.0000000       4.0000000
```

You can also use an index array like this with multi-dimensional arrays. For example, if you wanted to swap the first two columns of a matrix, you might do it like this:

```
real, dimension(3,3) :: reordered, matrix
...
reordered(:,:) = matrix(:,(/2,1,3/))
```

Before concluding this section on slicing, there is one advanced feature that can come in very handy: slices can be used with an array of a user-defined type to access members of the type. Consider this

```
type Person
   real :: height
end type
```

```
type (Person) :: people(10)
people(:)%height = 10.0

end
```

This program sets the `height` of all the `Person`'s stored in the array to 10.0, without using a loop. This technique can be quite useful when working with user-defined types.

Exercise: Slicing and Dicing Arrays

Write a program that declares a real array with dimensions 5 by 3.

Use array slicing to set the last two rows of this matrix to 0, and the first 3 rows to 1. Also use array slicing to take the square root of all elements in the third column, and store the result back in the matrix.

Lastly, declare a new one-dimensional array of length 3, and set it equal to 10 times the second row of the first array. Print out the one-dimensional array.

Compile and run. Are the values printed what you had expected?

Exercise: Slices and User-Defined Types

Write a short program that defines two user-defined types, `Source` and `Sink`. Include a real variable in each one called `value`. Declare an array of 50 `Source` objects, and another with 100 `Sink` objects. Initialize the value of the `Source` objects using the `random_number` intrinsic routine. Copy the values of the `Source` objects into the first 50 `Sink` objects. Do all of this without using any explicit loops (*ie*, use slices and array operations).

4.5. Allocatable Arrays

On many occasions, you will not know how big a particular array should be when you declare it. It may depend on conditions at *run time*, such as input parameters or other data. For example, you may need an array to store the points on a grid, but don't know how big the grid needs to be until you calculate its dimensions from the user's input.

Allocatable arrays allow you to postpone the *allocation* of memory until the correct dimensions have been determined. You declare the array as usual, but no memory will be set aside for the array until an *allocate* statement is used.

```
real, allocatable :: a(:)
integer n
n = 50
allocate(a(n))
...
deallocate(a)
```

The attribute *allocatable* is included with the array declaration, and colons are used in place of absolute dimensions. The declaration above indicates that the array `a` is real, allocatable, and is one-dimensional.

At some later point, before the elements of `a` are used, and after the dimensions of the array are already known, the array gets allocated. The dimensions of the array are provided in an `allocate` statement.

When an allocatable array is no longer needed, its memory must be freed up using a *deallocate* statement. In general, if you don't do this, your program will *leak memory*. This may result in the program crashing, or the memory of the computer filling up, causing it to slow dramatically. To avoid this situation, it is good practice to insert a `deallocate` statement whenever you insert an `allocate` statement, to ensure that they remain balanced.

Allocatable arrays are not restricted to a single dimension. You can have multi-dimensional allocatable arrays, and you can also specify index ranges in an allocate statement, like this

```
real, allocatable :: r(:,:)
allocate( r(5:10,5:6) )
r(6,6) = 2.0
print *, r(6,6)
deallocate(r)
end
```

If you ever need to test whether an allocatable array has already been allocated, you can use the *allocated* intrinsic function.

```
real, allocatable :: r(:)
print *, 'Is r allocated? ', allocated(r)
allocate(r(10))
print *, 'Is r allocated? ', allocated(r)
deallocate(r)
print *, 'Is r allocated? ', allocated(r)
end
```

This program gives the following output:

```
 Is r allocated?  F
 Is r allocated?  T
 Is r allocated?  F
```

A common question that new Fortran developers have is whether you can pass an *unallocated* allocatable array to a procedure, allocate it in the procedure, and then use it upon return.

```
real, allocatable :: r
call AllocArray(r)
deallocate(r)
end

subroutine AllocArray(r)
   real, allocatable :: r
   allocate(r(10))
end subroutine
```

This is not allowed in Fortran 90 and 95, but has added in Fortran 2003. Unfortunately, the Fortran 2003 standard is not yet widely used, so in practice you cannot pass an allocatable array to a procedure for allocation. But you can achieve a similar effect using array pointers, which are discussed in the next section.

Exercise: Allocating Arrays

Write a short program that declares an integer parameter n, and initializes it to 50. Declare a real allocatable array, and allocate it to hold n numbers. Use an implicit loop to fill it with the numbers 30.0, 30.1, 30.2, *etc*. Print the array, and deallocate it. Compile and run the program.

Now change the value of n to 100, and see if the program still works. If not, fix it so that there are no assumptions about how big n is.

4.6. Pointers

Pointer variables are variables that hold references to other variables or data. To declare a pointer variable, you simply add the attribute *pointer* to its declaration.

```
real, pointer :: r
```

In this example, `r` is not a real number, but a pointer or reference to a real number. So you can't simply use `r` without first pointing it at something. If you try to do this

```
real, pointer :: r
r = 5.0
end
```

your program will likely crash, because r has not been associated with any space in memory, and so trying to put the real number 5.0 into it will have undefined consequences.

There are two ways that you can associate memory with a pointer variable. The first is to use a *pointer assignment* to another pointer variable or target variable. A target variable is a variable with the *target* attribute.

```
real, pointer :: a
real, target :: b
b = 5.0
a => b
print *, a
end
```

The operator => is the pointer assignment operator; it associates the pointer variable on the left with the target or pointer variable on the right. After a pointer has been associated in this way, it points to the same memory as the other variable.

In this case, `a` and `b` will share the same memory location. You can see this because we never actually assigned the value of `a`, and yet the program will print out 5.0. That's because we assigned `b` the value of 5.0, and then made `a` point to the same location in memory, which contained 5.0.

The second way that you can associate a pointer variable with some memory is to use an allocate statement.

```
real, pointer :: r
allocate(r)
r = 5.0
print *, r
deallocate(r)
end
```

Using allocate on a pointer variable sets aside enough memory to hold the variable, and then associates the variable with the memory. *Be sure to deallocate the memory when you are finished with it, or you will get a memory leak.* Something to be aware of is that the deallocate statement will operate on the memory that is currently associated with the pointer; you should be sure that this memory was originally allocated with an allocate statement, and does not, for example, correspond to a target variable.

You can also use the pointer attribute with arrays. Array Pointers can be used much like allocatable arrays, in that you can allocate and deallocate memory for them, but they have the added advantage that you can associate them with other pointers or targets. You can even associate them with sub-arrays.

```
real, pointer :: a(:,:)
real, pointer :: b(:)

allocate(a(3,3))
a = 0.1

b => a(1,:)
b = 0.2

print *, 'a(1,1) is ', a(1,1)
print *, 'a(2,1) is ', a(2,1)

deallocate(a)
end
```

When you run this code you get the following output:

```
 a(1,1) is   0.20000000
 a(2,1) is   0.10000000
```

This shows that `b` is pointing to the first row of `a`, because assigning `b` to 0.2 also changes `a`.

There are some subtleties you should be aware of when working with array pointers. Firstly, the pointer being associated must conform in rank with what it is being associated with. For example, you cannot associate a rank 2 array pointer with a rank 1 array or sub-array.

Secondly, the indexes of the array pointer do not necessarily correspond to the indexes of the target. To demonstrate this, consider the following

```
real, pointer :: a(:,:)
real, pointer :: b(:)

allocate(a(3:5,3))
a = 0.1
b => a(:,1)
b(1) = 0.2

print *, 'a(3,1) is ', a(3,1)

deallocate(a)
end
```

In this example, `a` is allocated with a lower bound for the first index of 3. `b` is then assigned to the first column of `a`, and the first element of `b` is set to 0.2. When the value of `a(3,1)` is printed, it has changed 0.2. In other words, `b(1)` corresponds to `a(3,1)`. So when you associate an array pointer, the array pointer does not inherit the indexes of the pointee.

Pointers allow you to perform some useful tricks with variables, without having to copy the data they contain. For example, in the following code, imagine you wanted to have the array `a` end up holding the data from `b`, and `b` hold the data from `a`. Rather than copying it, you could swap the pointers.

```
real, pointer, dimension(:) :: a, b, c

allocate(a(5), b(10))
a = 0.1
b = 0.2

! Swap a and b, using temporary c
c => a
a => b
b => c

print *, 'a is ', a
print *, 'b is ', b

deallocate(a,b)
end
```

If you run this program, you should see that `a` holds 10 elements equal to 0.2, and `b` holds 5 elements equal to 0.1. The 3 lines in the middle effect this swap. Note that an extra pointer, `c`, is needed to temporarily store the location of the data originally pointed to by `a`; if you didn't use `c`, as soon as you associated `a` with `b`, the location of that memory would be lost.

To test if a pointer is pointing at something, you can use the *associated* intrinsic function, like so

```
if ( associated(p) ) then
   print *, 'Pointer is associated'
endif
```

but this will only work if the pointer is initially *nullified.* Nullifying a pointer variable is basically the same as pointing it at nothing. There are two ways to do this: you can either call the *nullify* intrinsic function, or associate the variable with the return value of the *null* function. (The latter only works in Fortran 95 and later.)

```
real, pointer :: a
real, pointer :: b => null()
nullify(a)
allocate(b)
print *, 'a is associated: ', associated(a)
print *, 'b is associated: ', associated(b)
deallocate(b)
end
```

In the previous section, we mentioned that you cannot pass an allocatable array to a procedure to allocate it. You can do this with array pointers, but only under certain conditions. In particular, you need to have an explicit interface. Here is some code to show you how you can go about passing an array pointer to a subroutine for allocation.

```
module AllocMod
contains
```

```
   subroutine Alloc(a, n)
      real, pointer       :: a(:)
      integer, intent(in) :: n
      allocate(a(n))
   end subroutine
end module

program AllocTest
   use AllocMod
   real, pointer :: r(:)
   call Alloc(r, 10)
   r = 100.0
   print *, r
   deallocate(r)
end program
```

A *module* is used here to make the interface of the subroutine explicit. If you try this without an explicit interface, the code will compile, but will crash when you try to run it.

Exercise: Working with Array Pointers

Write a short program that declares a real array pointer, `a`, to represent a matrix. Allocate memory for `a` to take the dimensions 5 by 5, and initialize all elements to 0.0.

Add a second array pointer, `b`, and associate it with the bottom right corner of `a`, with both indexes ranging from 3 to 5. Set all elements of `b` to 1.0.

Lastly, add a rank 1 array pointer, `c`, and associate it with the third row of `a`. Print out `c`.

Compile and run the program. What do you notice about the elements printed? What does this tell you about the memory associated with `a` and `b`?

4.7. Advanced Loops

We have already seen how you can use do loops to work with arrays, and how you can avoid loops altogether by using element-wise expressions, but Fortran has a few other types of loops that may come in handy when working with arrays.

The where loop is use to update the elements of an array that meet a certain condition.

```
real a(10)
call random_number(a)
where ( a > 0.5 ) a = sqrt(a)
end
```

In this single-line version of the loop, any element in a that is greater than 0.5 is updated to the square root of the element. Note that the where loop doesn't use explicit indexes — all operations are performed on each element, with the indexes implicit.

There is also a multi-line version of the where loop.

```
real a(10), b(10)
call random_number(a)
b = 0
where ( a > 0.5 )
   a = sqrt(a)
   b = 2 * a
```

```
elsewhere
   b = 1.0
endwhere
end
```

In this example, `a` is updated as before, and each element of `b` is set to twice the corresponding element of `a`, but only where the element of `a` is greater than 0.5 to begin with. This loop also includes an `elsewhere` block; `elsewhere` is optional and only gets executed whenever the condition is false. In this particular example, an element of b is set to 1.0 whenever the corresponding element of a is *not* greater than 0.5.

If you wrote this out as a do loop, it might look like this

```
real a(10), b(10)
integer i
call random_number(a)
b = 0
do i = 1, 10
   if ( a(i) > 0.5 ) then
      a(i) = sqrt(a(i))
      b(i) = 2 * a(i)
   else
      b(i) = 1.0
   endif
enddo
end
```

As you can see, the where loop is more compact and readable.

Fortran also includes another loop called `forall`, which performs operations over a range of indexes.

```
real a(10,10)
integer i
a = 0.0
forall ( i = 1:10 ) a(i,i) = 1.0
end
```

This program sets up the array `a` as an identity matrix, with 1's on the diagonal, and 0's elsewhere. Unlike the `where` loop, `forall` has explicit indexes, but they are given in slice notation, rather than the comma separated ranges of the `do` loop. And whereas the iterations of a `do` loop are performed sequentially, this is not a requirement on the `forall` loop — iterations need not be executed in order.

`forall` also has a multiline block form, which you terminate with `endforall`.

Exercise: Where art thou?

Write a where loop that ensures that any values in an array with an absolute value less than 1.e-6 are given an absolute value of 1.e-6 with sign unchanged. (*Hint*: *the intrinsic routine* `abs` *can be used to calculate the absolute value of a number, and the intrinsic* `sign` *function can supplant the sign of one number with that of another.)*

4.8. Array Intrinsics

In modern Fortran code, you can often get away with operating on arrays without having to write any loops at all. We have already seen that you can perform element-wise operations, including applying mathematic operators such as `sqrt`. But there are many other array intrinsics, and the most useful ones are presented in the table below.

For the purpose of the examples in this table, we make the following array declarations:

```
real    :: vec1(3) = (/1,2,3/)
real    :: vec2(3), matrix(3,3), multi(3,4:8,5)
integer :: i
real    :: r
```

Intrinsic	Description	Example
dot_product	Dot product of two arrays	`r = dot_product(vec1, vec2)`

lbound	Lower bound of an array dimension	`i = lbound(multi, 2)  ! gives 4`
		`i = lbound(vec1)      ! gives 1`

minloc	Location of the minimum value in an array, returned as a one-dimensional index array	`integer indexes(3)`
		`indexes = minloc(multi)`

minval	The minimum value in an array	`r = minval(multi)`
matmul	Matrix multiplication of two arrays	`vec2 = matmul(matrix, vec1)`

maxloc	Location of the maximum value in an array, returned as a one-dimensional index array	`integer indexes(3)`
		`indexes = maxloc(multi)`

maxval	The maximum value in an array	`r = minval(multi)`

reshape	Reshape an array to have the dimensions given in the second argument array	`real long(9)`
		`long = reshape( matrix, (/9/) )`

size	Return the number of elements in a whole array, or the size of a particular dimension	`i = size(matrix)    ! Gives 9`
		`i = size(matrix, 2) ! Gives 3`
		`i = size(multi, 2)  ! Gives 5`

spread	Increase the rank of an array by duplicating elements along a particular dimension (second argument) an allotted number of times (third argument)	`matrix = spread(vec1, 1, 3)`
		`! Gives`
		`! 1  2  3`
		`! 1  2  3`
		`! 1  2  3`
		
		`matrix = spread(vec1, 2, 3)`
		`! Gives`
		`! 1  1  1`
		`! 2  2  2`
		`! 3  3  3`

Intrinsic	Description	Example

sum	Sum the elements of an array, or sum elements along a particular dimension	`r = sum(matrix)`
		`vec1 = sum(matrix, 2)`

transpose	Transpose of a matrix	`real trans(3,3)`
		`trans = transpose(matrix)`

ubound	Upper bound of an array dimension	`i = ubound(multi, 2)  ! gives 8`
		`i = ubound(vec1)      ! gives 3`

There are a lot of useful routines in the table above, but it won't be immediately obvious to you how some of them can be used. To get the ball rolling, we will now give a few practical examples of some of the array intrinsics.

It should be fairly obvious that some of the matrix operations are useful, but they become really powerful when you start to combine them in expressions. For example, imagine you need to perform a similarity transform; perhaps you have diagonalized a matrix, and want to transform some coefficients from the original space to the eigenspace. Here's how you might do it.

```
real :: coeffs(10), eigenspaceCoeffs(10), eigenvectors(10,10)
eigenspaceCoeffs = matmul(transpose(eigenvectors), coeffs)
```

Transforming a matrix from one space to another is not much more difficult:

```
real, dimension(10,10) :: matrix, eigenspaceMatrix, eigenvectors
eigenspaceMatrix = matmul( transpose(eigenvectors), matmul(matrix, eigenvectors) )
```

These operations would usually require many lines of nested loops, but with the array intrinsics, they are literally reduced to single line operations.

The `dot_product` and `sum` intrinsics are very useful for operations such as determining the length (norm) of a vector, or the angle between two vectors. To find the length of a vector, you can do it like this

```
real :: vec(3)
print *, 'Length is ', sqrt(dot_product(vec,vec))
```

or like this

```
real :: vec(3)
print *, 'Length is ', sqrt(sum(vec**2))
```

Calculating a unit vector is equally trivial:

```
real :: unit(3), vec(3)
unit = vec / sqrt(sum(vec**2))
```

`size` is one of the most widely used array intrinsics. It is particularly useful when you don't know exactly how long an array is, such as is the case for *assumed-shape arrays*, which are discussed below.

```
real    :: a(:)
integer :: i
do i = 1, size(a)
   a(i) = i
enddo
```

`size` also works great with multi-dimensional arrays.

```
real    :: a(:,:)
integer :: i,j
do i = 1, size(a,2)
   do j = 1, size(a,1)
      a(j,i) = i+j
   enddo
enddo
```

The last intrinsic we will discuss in detail is included not because it is widely used, but because it is under used. The `spread` intrinsic takes some getting used to, but it can help to create array expressions which would otherwise not be possible. For example, take this fairly common problem: You have a matrix and an array of weights; you want to multiply each column of the matrix by the weights. Here is a solution using a loop.

```
real :: matrix(30,50), weights(30)
integer i
do i = 1, size(matrix,2)
   matrix(:,i) = matrix(:,i) * weights
enddo
```

That's not too bad, but with `spread` it can be made even more compact:

```
matrix = matrix * spread(weights, 2, size(matrix,2))
```

`spread` turns the `weights` vector into a matrix where each column is equal to the original array. The only thing left to do is an element-wise multiplication of the result with `matrix`.

Another example of the usefulness of `spread` is taking the outer product of two one-dimensional arrays. Usually you would do it like this:

```
real :: vec1(10), vec2(10), outer(10,10)
integer :: i,j
do i = 1, 10
   do j = 1, 10
      outer(i,j) = vec1(i) * vec2(j)
   enddo
enddo
```

but with spread, it can again be reduced to a single line:

```
outer = matmul( spread(vec1,2,1), spread(vec2,1,1) )
```

The first call to `spread` turns `vec1` into a 10 by 1 matrix (*ie*, a column vector), and the second converts `vec2` into a 1 by 10 matrix (*ie*, a row vector). The matrix product then gives the outer product.

Exercise: Sizing Things Up

Take a look at this code

```
real, dimension(10,5) :: a
integer               :: i,j
do j = 1,5
   do i = 1,10
      a(i,j) = i+j
   enddo
enddo
end
```

Remove any explicit references to the array dimensions from the do loops, and replace them with expressions using the `size` intrinsic. Ensure the code compiles and runs.

Exercise: Reshaping

Write a short program that initializes a real one-dimensional array to the numbers 1, 2, 3, ... 9, and then uses the `reshape` intrinsic to copy these values into a matrix, such that the first column becomes 1, 2, and 3, and the second column 4, 5, and 6 etc.

Exercise: Finding Angles

Consider these vectors

```
   real :: vec1(3) = (/0, 2, 0/)
   real :: vec2(3) = (/2, 0, 0/)
```

Use array intrinsics to determine the angle between the two vectors in radians. *(Hint: You need to use the definition of a dot product.)*

Exercise: Spreading Integrals

Suppose you have a matrix with rows corresponding to grid points, and columns corresponding to a set of functions. Each grid point has a weight, such that you can calculate the integral of a function by multiplying each value on each grid point by its weight, and summing up. Write code that uses the `spread` function to integrate all the functions stored in the matrix. Assume the weights are stored in a one-dimensional array.

4.9. Arrays in Procedures

When you use explicit interfaces for your procedures, by including them in a module, for example, the compiler is able to pass extra information about an array, such as its dimensions. This can save you having to pass this information via the argument list.

```
module TestMod
   implicit none
   save
   private
   public PrintRows
contains
   subroutine PrintRows(array)
      real :: array(:,:)
      integer i
      do i = 1, size(array,1)
         print *, 'Row ', i
         print *, array(i,:)
      enddo
   end subroutine
end module

program TestProg
   use TestMod
   real :: a(3:5,3:5)
   a = 5.0
   call PrintRows(a)
end program
```

The argument `array` in the `PrintRows` subroutine is known as an *assumed-shape array*, because its dimensions are 'assumed' from the array that is passed in. `PrintRows` can accept any rank two real array. The size of each dimension is also passed implicitly by the compiler, so using the `size` intrinsic works inside the subroutine to determine the array's dimensions.

Note that there is a one-to-one correspondence between the elements of the array passed in, `a`, and those of the array in the subroutine, `array`, but that the indexes do not necessary correspond. In the calling code, `a` has indexes ranging from 3 to 5, but these map to 1 to 3 in the `PrintRows` subroutine.

You can also use slicing to pass sub-arrays to procedures. For example, imagine you only wanted to print the upper-left 2 by 2 section of the array `a`. You could change the main program to the following

```
program TestProg
   use TestMod
   real :: a(3:5,3:5)
   a = 5.0
   call PrintRows(a(:4,:4))
end program
```

or you could even pass in a sub-array holding the four corners of `a`, like this

```
program TestProg
   use TestMod
   real :: a(3:5,3:5)
   a = 5.0
   call PrintRows(a(3:5:2,3:5:2))
end program
```

Fortran provides another way of avoiding having to declare the exact dimensions of an array argument in a procedure: *assumed-size arrays*. An array is assumed size when you use an asterisk for the last dimension. If you do this, the compiler will do no checking of the array dimensions — it is up to you to make sure that the dimensions are correct.

```
subroutine AssumedSizeTest(a)
   real :: a(10,*)
   print *, a(:,:10)
end subroutine
program Main
   real :: a(100)
   a = 0.1
   call AssumedSizeTest(a)
end program
```

In the procedure `AssumedSizeTest`, the array `a` is assumed size. The compiler will do no checking of the dimensions against the argument passed in, so the assumed-size array can even have a different shape to the array passed in. In this case, the assumed-size array is rank 2, and the array passed in is rank 1.

The print statement in `AssumedSizeTest` is somewhat unusual. Notice that an upper bound has been given for the second dimension of the array `a`. That's because the compiler cannot determine the upper bound of the last dimension of an assumed-size array, or its total size for that matter, so you need to provide it explicitly. When an assumed-*shape* array is used, the compiler knows the exact dimensions of the array, and it is not necessary to supply the upper bound.

Assumed-shape arrays are superior to assumed-size arrays because they provide the compiler with more information for checking. You should only use assumed-size arrays on rare occasions, perhaps when you need to change the rank of an array (though, the `reshape` intrinsic is probably a better way to handle this).

Earlier we saw that you could use allocatable arrays when you do not know how big an array has to be before a program starts running. Fortran provides a second mechanism for sizing arrays at run time: *automatic objects*. Automatic objects are local variables that are allotted memory when a procedure begins to execute, and that memory is cleaned up when the procedure exits.

```
subroutine AutoObject(n)
   integer, intent(in) :: n
   real :: a(n)
   a = 0.1
   print *, a
end subroutine

program main
   call AutoObject(10)
end program
```

This program will print out the number 0.1 ten times. The array `a` is an automatic object: it is created when the subroutine `AutoObject` is called, and deleted when the subroutine exits. The size of the array is given by the argument `n`.

A common application of automatic objects is to make a local array the same shape as an array argument. You can do this using the `size` intrinsic.

```
module SimilarArraysModule
contains
   subroutine DoubleAndPrint(a)
      real, intent(in) :: a(:)
      real             :: copy(size(a))
      copy = 2 * a
      print *, copy
   end subroutine
end module

program Test
   use SimilarArraysModule
   real :: a(10)
   a = 0.1
   call DoubleAndPrint(a)
end program
```

The automatic object `copy` is given the same size as the array argument `a`. It is then used to temporarily store some values, and will be automatically deleted when the routine `DoubleAndPrint` returns.

Exercise: Assumptions About Arrays

Write a subroutine that prints the first 10 elements of any one-dimensional array it is passed. Put the subroutine in a module, and make the array argument an assumed-shape array. Make sure you test the size of the array passed in to make sure it is big enough.

Add an automatic object to the subroutine, an array that holds as many elements as the array passed in. Use this local array to store the values of the array passed in, raised to the power of 2. Print out the sum of all the elements in the local array.

Write a program to test the subroutine, and compile and run it.

5. Input/Output (IO)

Thus far, we have dealt with small programs that keep all of their data stored in variables in the main memory of the computer, but real applications need to have the capability of storing data on file, and reading it back in again. This is known in programming as *Input/Output (IO)*.

An important part of IO is *persistence*, which is having data persist even when the program stops running or the computer is turned off. You typically achieve this by writing information to a file on the hard disk, such that it continues to exist even after the program exits.

In this section, we are going to learn the basics of IO in Fortran — how you write data, and how you read it back in.

5.1. Standard Input and Standard Output

Command-line programs, like the ones we are writing in this course, need a means of passing in input parameters and printing out results. We have already used the `print` statement in many of the examples to print out the value of a variable, or the contents of an array. When you use `print`, you are sending data to a special channel called *standard output*, which usually results in the data appearing in the users terminal.

None of the examples provided to date have read any input, but this is handled in an analogous manner to writing output. Data is read in from *standard input*. This data would usually come from a redirected file, or be entered directly by the user at a prompt.

In the coming sections, you will see how you can read and write data to/from standard input and output, respectively. After that, we will move on the more general forms of IO, such as reading and writing files on disk.

5.2. Writing

Using the `print` statement, output always goes to standard output, but there is a more general statement that can be used with files too: `write`. In order to use write to output some data, you need to supply it with the *unit number*. In Fortran, the unit number of standard output is always 6.

```
print *,'This is with print'
write(6,*)'This is with write'
end
```

Optionally, you can use an asterisk for the unit number as well, and it will default to standard output.

```
write(*,*)'This is with an asterisk'
```

The two output lines above look and behave much the same. The `write` statement looks like a function, but isn't, and must be supplied a unit number, whereas `print` always assumes the unit number is 6. The asterisk in each case is for the *format*, which will be discussed in the next section.

An interesting aside — and a useful one — is that you can use write to enter data into a character string. When used in this way, the string is known as an *internal file*.

```
character(64) :: string
```

```
integer       :: n = 1000
write(string, *)'There are ', n, ' elements'
```

When this code runs, the variable `string` will end up with the characters ' There are 1000 elements'. Internal files can thus be used to convert data into character format, and dynamically build up strings. The `read` statement, which is discussed below, can achieve the reverse operation, reading a character string and converting it into a numerical type, for example.

5.3. Formatting

The asterisk in the `print` and `write` statements above indicate to the compiler that you do not mind too much how the output looks, but — if you wish — you can control the appearance and accuracy of the output using a *format string*.

```
real(8) :: r
r = 12.4354873498543
print *, r
print '(f14.10)', r
print '(f12.6)', r
end
```

When run, this program outputs

```
   12.435487747192383
 12.4354877472
   12.435488
```

A format string begins and ends with parentheses, and includes a comma-separated list of format codes. Below is a table detailing some of the most useful format codes.

Code	Meaning	Example

f*a.b*	A real number occupying *a* characters in total, with *b* numbers after the decimal point	`! Following prints out '-213.153'`
		`print '(f8.3)', -213.153452325`

e*a.b*	A real number in exponential format, with *a* the total characters, and *b* the number of decimal places	`! Following prints out '  -0.213E+03'`
		`print '(e12.3)', -213.153452325`

es*a.b*	A real number in scientific format, with *a* the total characters, and *b* the number of decimal places	`! Following prints out '  -2.132E+02'`
		`print '(es12.3)', -213.153452325`

i*n*	An integer occupying *n* characters in total	`! Following prints out '  -213'`
		`print '(i6)', -213`

a*n*	Print a string of length *n*; if *n* is excluded, the whole string will be printed	`print '(a)', 'Hello'   ! Prints 'Hello'`
		`print '(a3)', 'Hello'  ! Prints 'Hel'`
		`print '(a7)', 'Hello'  ! Prints '  Hello'`

Code	Meaning	Example
t*n*	Insert a tab to the *n*-th column	`print '(a,t10,i1)','label',3 ! Prints 'label     3'`
x	Print a space	`print '(i1,x,i1)',5,3  ! Prints '5 3'`
/	Print a new line character	`print '(i1,/,i1)',5,3  ! Prints on two lines`

A number before any code indicates that it is repeated that number of times. So, in order to print 5 integers, you could do this

```
print '(5i3)',1,2,3,4,5
```

This would print out each integer in 3 characters, giving two spaces between each.

The repeat rule also works with groups of format codes, which can be formed using parentheses, like this

```
print '(3(i3,x,f5.3))',1,1.2,2,2.4,3,5.3
```

This prints out

```
  1 1.200  2 2.400  3 5.300
```

The factor of three at the start means that the group of codes in parentheses is repeated three times.

When there are more values to print than format codes, a new line is inserted, and the format codes repeat. To print an array with 10 elements, for example, you could do this

```
real a(10)
a = 0.1
print '(5f12.6)',a
end
```

which prints out two lines of five entries, like this

```
    0.100000    0.100000    0.100000    0.100000    0.100000
    0.100000    0.100000    0.100000    0.100000    0.100000
```

Complex numbers are just treated as two real numbers, one for the real part, and one for the imaginary part. To print them, you just need to double up on the format codes.

```
complex a(3)
a = cmplx(0.1,0.2)
print '(3(2f12.6,2x))',a
end
```

which prints out

```
    0.100000    0.200000      0.100000    0.200000      0.100000    0.200000
```

We have already seen that you can print a string using the a format code, but you can also just insert literal strings into a format code, provided you are careful not to mix quotation marks.

```
integer :: n = 454
print '("The number is",x,i5)',n
end
```

All of the examples above utilize `print` statements, but they work equally well with `write`. Here's the previous example using `write` instead of `print`:

```
integer :: n = 454
write(*,'("The number is",x,i5)') n
end
```

There is a second, and more traditional way of including format strings: the *format statement*. The format statement was very common in early Fortran code, and is still in common use today.

When you use a format statement, you label it with a number, and then provide that number to the `print` or `write` statement.

```
real a,b,c
a = 0.1
b = 0.2
c = 0.3
write(6,9000)a,b,c
9000 format(3f12.6)
end
```

In modern Fortran, it's generally better to include the format string in the `write` or `print` statement, rather than in a separate format statement. If you find the `write`/`print` statement becomes difficult to read, or you need the same format string for multiple statements, you can always create a character string variable to use for the format.

```
character(32) :: form = '("The number is",x,i5)'
write(*, form) 32
write(*, form) 64
end
```

5.4. Reading

Reading data in Fortran is very similar to writing it. Reads can be *unformatted* or *formatted*. By default, you read from standard input, which has the unit number 5. Here is a simple program that prompts the user to enter a number, and then prints the number back out:

```
program Main
   integer i
   print *,'Enter a number'
   read *,i
   print *,'You entered ', i
end program
```

Analogous to the `print` statements, when you use an asterisk with `read` as above, it performs an unformatted read from standard input. When you are running a program interactively, standard input comes from the keyboard.

It will not come as a surprise that you can include as many variables as you like in a `read` statement.

```
program Main
   real r1,r2,r3
   print *,'Enter three real numbers'
   read *,r1,r2,r3
   print *,'You entered ',r1,r2,r3
end program
```

A second form of the read statement is used for formatted reads.

```
program Main
   real r1,r2,r3
   print *,'Enter three real numbers'
   read(*,'(3f5.1)')r1,r2,r3
   print *,'You entered ',r1,r2,r3
end program
```

What is a formatted read? When you use a format string in the read statement, the program assumes that is the exact format that will be supplied to it. If the format of the input is different, unexpected results may arise. For example, in the program above, if you enter '34.1 ' for three times, which fits the format, you get the following output.

```
You entered    34.099998       34.099998       34.099998
```

Allowing for round-off error, that is good enough. But if you enter the following numbers

```
45324.324545 345543245.345 2434.32
```

which do not fit the format, you get this output

```
You entered    4532.3999      0.32449999       453.39999
```

You can see that the program interpreted the input based on the format, and ignored the whitespace giving unexpected results.

Exercise: Interactive Input

Try compiling and running the first program in this section. What happens when you enter an integer? What about when you enter a non-integer string at the prompt?

Exercise: Formatted Output

Write a program that outputs two columns of data. The first column should contain integers between 1 and 100. The second column should contain random numbers between 0.0 and 50.0. Use a `write` statement with a format string to format the columns. Compile and test the program. Pipe the output to a file, like this

```
program > programoutput.dat
```

Exercise: Formatted Input

Write a program that reads in the data printed out in the previous exercise, using a formatted read. Compile and run the program, redirecting standard input in order to read the output file from the previous exercise, like this

```
program < programoutput.dat
```

5.5. Opening and Closing Files

Being able to read and write standard input and output is useful, but real-world programs usually need to work with many more files. Fortran allows you to open files in various modes, read and/or write them, and close them. To open a file and write to it, you can do this

```
open(7,file='progoutput.txt')
write(7,*)'This is output'
close(7)
end
```

You pass a unit number in the `open` statement — which should be greater than 6 — and use this thereafter when referencing the file, such as in the `write` statement, and the `close` statement. The `open` statement takes an argument for the name of the file to open.

There are many other optional arguments for the `open` statement, which give you more control over how the file can be accessed, whether it should already exist prior to opening, and so forth, but we will not cover these details here.

Opening a file for reading is very similar, except that the file must exist prior to the `open` statement being executed.

```
real r1,r2,r3
open(32, file='proginput.txt')
read(32,*)r1,r2,r3
close(32)
end
```

This program will try to open a file called 'proginput.txt', and read in real numbers from it.

Exercise: Formatted Output to File

Modify the programs you wrote in the previous section to read from and write to a file, rather than standard input and output. Compile and run.

6. Structured Programming

We've now covered the basics of Fortran programming. From here on the course will deal with how you use the language to write robust software. A programming language is not dissimilar to a spoken language: You have now learned the words, but you need to be able to combine them into sentences, paragraphs, and chapters before you can write a novel.

6.1. Programming Paradigms

Structured programming is a term that can have different meanings to different people, but here we will use it to mean building up a program from well-structured data types and procedures. These building blocks should be relatively independent of one another — *loosely coupled*.

Traditional Fortran programming is better described as *procedural*, because subroutines and functions are used to break-up the functionality and flow of a program, but the data (*eg*, variables) are left relatively unstructured. Often variables will be *global* — shared by all parts of a program — rather than localized to a particular subroutine or module.

Fortran 90 introduced new language constructs, like user-defined types and modules, that facilitate structured programming. Using this paradigm makes larger programs easier to understand and modify than programs written in the procedural style. Even more recently, a third paradigm — object-oriented programming — has been introduced to Fortran (in 2003), but this style is not yet widely-used, partially due to the lack of compilers that support it.

6.2. Good Design

Before moving on to structured programming in practice, we will spend quite a bit of time discussing concepts of software design. This will help you better grasp the reasoning behind the techniques you will learn later in the section.

It's quite difficult to teach someone 'good design'. How do explain how you paint a beautiful oil painting, or design a classic piece of architecture? Creative processes are not easy to teach. You can tell someone how to bring the paint onto the canvas, but if it were easy to teach someone to be a great artist, everyone would be doing it.

Luckily, software design is not as inaccessible as art, though it does involve a creative process: If you take two different software designers, they will often come up with two different solutions to the same problem. And some will come up with better designs than others — software design is far from being a mechanical process. A given problem does not have a unique solution in software, and a creative designer will generally come up with a better solution than someone that approaches design mechanically.

That said, there are guidelines to what constitutes good design. It is possible to ask 'What makes a design good?', and to give reasonable answers. The following sections are about exploring the essence of good software design.

6.3. Duplication

Creating software is not much more than information management. You have a problem to solve, and you have to express that in a form that can be mechanically performed by a computer. The programming language and paradigm you use is largely irrelevant to the computer: ultimately it must perform a set of operations that lead to the solution, independent of how you initially represent those operations.

Having duplicated information in a program is no problem for a computer, it may even be advantageous in some cases. A computer can perform the same operation billions and billions of times, and never make a mistake. A human being is a whole other story. Duplication of information — whether it be code, documentation or data — is bad for developers, because it creates artificial dependencies in software that need to be maintained in the future.

Take a simple case: You 'copy and paste' 30 lines of source code. It seems harmless enough, but later you realize there is a bug in that code. You then have to fix the bug in two places, and if you forget you copied that code, you will still be left with a bug.

The two pieces of code express the same information — a given set of operations — and are effectively dependent upon one other as a result of that shared functionality. Should you ever need to change one copy, you would almost certainly need to change the other. This is not only more work, it also increases the code complexity: You will have to remember that you copied the code, and that the dependency exists for the life of the software. Furthermore, if you stop developing the software, you will have to explain that dependency, and any others, either in documentation, or directly to another developer. If not, there is a very high risk of bugs being introduced, and *code rot* setting in.

The simplest solution to this scenario is to avoid it. *Avoid it at all costs*. Be vigilant when it comes to duplication. Feel repulsed by the sight of it. It should keep you awake at night.

Some software development companies take duplication very seriously. One company organized to have a jar installed, and anyone caught even thinking about using copy and paste was required to contribute a dollar to the jar. (The money in the jar funded Friday evening's entertainment for the group at the local cafe.) This may be going to extremes, but it shows you that experienced software developers treat duplication as a serious issue.

Direct duplication is easy to spot and avoid. If you find yourself reaching for the Edit menu, think twice. But duplication can be much more subtle. Maybe two pieces of code look similar, but there are a few small differences. There is still considerable information that is duplicated, but there is also some unique information in each section.

One solution to this is to break the code into smaller grains, adding structure by placing each in a procedure. The differences between the two sections of code can then be represented by variations in the procedure call. For example, the code

```
integer, parameter :: n = 100
integer, dimension(n) :: x, y
integer :: i

do i = 1, n
   x(i) = 2
enddo

do i = 1, n
   y = 3
enddo
```

```
do i = 1, n
   x(i) = x(i) * y(i)
enddo
```

has clear similarities to

```
integer, parameter :: n = 50
integer, dimension(n) :: a, b
integer :: i

do i = 1, n
   a(i) = 2
enddo

do i = 1, n
   b(i) = 3
enddo

do i = 1, n
   a(i) = a(i) * b(i)
enddo
```

The code is not directly duplicated, but duplicated information is unquestionably present.

A *factorization* that reduces this duplication, making the software easier to maintain, is to introduce a procedure.

```
subroutine PerformArrayOps(a, b)
   integer :: a(:), b(:)
   integer :: i, n

   n = size(a)
   do i = 1, n
      a(i) = 2
   enddo

   do i = 1, n
      b(i) = 3
   enddo

   do i = 1, n
      a(i) = a(i) * b(i)
   enddo
end subroutine
```

Now, rather than explicitly entering the loops into the code at each point, a call is made to the `PerformArrayOps` procedure. The loops that perform the operations are only found once in the program.

This new code, while not necessarily being shorter, has a number of advantages: The shared information is now clearly delineated, because it has been shifted into a procedure. Looking at the examples before restructuring, you had to think a bit to see the similarities. It was a bit like a 'Spot the Differences' quiz. In the restructured examples, the differences are quite clear: they are the arguments to the subroutine.

Information duplication has been reduced considerably in the restructured code, and this is a big advantage. Each time you call a procedure, you are avoiding duplication of the algorithmic information that it contains. If there is a bug in the `PerformArrayOps` subroutine, only the code in that subroutine needs to be changed. In the unstructured case, each instance of the duplicated section of code would need to be checked. In real-world software, these sections could be widely-spread throughout the code base, making it a non-trivial exercise.

A user-defined type is also a means of avoiding duplication: duplication of data declaration. Just as a procedure allows you to reuse a set of operations, a user-defined type allows you to reuse a particular data structure.

Take this unstructured example

```
subroutine Apply(first, second, third)
   integer :: first, second, third(:)
   ...
end subroutine

...

integer :: x, y, z(10)
integer :: a, b, c(10)
integer :: e, f, g(10)

call Apply(x, y, z)
call Apply(a, b, c)
call Apply(e, f, g)
```

There is lots of declarative information duplicated here, because structural information has not been included in the code. That information must be explicitly stated each time a variable is declared.

If we add structure to the data, we can reuse that blueprint every time we need a new instance.

```
type Data
   integer :: first, second, third(10)
end type

...

subroutine Apply(d)
   type (Data) :: d
   ...
end subroutine

...

type (Data) :: x, a, e

call Apply(x)
call Apply(a)
call Apply(e)
```

By only including each piece of information once, we have again made the code more robust. The code is also more readable, because the relationships between the variables have been explicated, with closely-related data being grouped together.

Changes need only be made at one point should they be required, *i.e.*, the `Data` type. If you decide to add a fourth variable to the type, the rest of the program, including the subroutine calls, is unaffected. This is not the case in the unstructured code: every subroutine call would need to be updated to include the new variable, and declarations of the new variable would be required at every point that the other variables have been declared.

6.4. Indirection

Each of the solutions to duplication discussed above have something in common: they introduce *indirection*. A function or subroutine allows you to access a set of operations indirectly, and a user-defined type is used to indirectly reference data. Indirection is as old as the art of programming itself, and embedded deep in its core.

In the beginning, there were computers the size of rooms, with punch card readers instead of keyboards, and human batch systems, feeding in input, and extracting output. Back in those days, programs were written in machine code or assembler. You gave instructions directly to the machine, referencing memory directly. A program written for one type of computer, had no chance of ever running on another.

High-level languages, like C and FORTRAN, introduced a level of indirection: Instead of making direct reference to memory addresses, explicitly loading and storing the data they contained, it was possible to use variables. Variables allowed a programmer to reference memory indirectly, with a compiler translating instructions for the machine on which it would run. As well as being more readable, this was much more flexible than coding directly in assembler or machine code. A single program could be made to run on different types of computers, by creating a different compiler for each architecture.

This was a big step forward, but large programs were still difficult to write and maintain. In the early days, GOTO statements were used to control program flow, and this soon led to *spaghetti code*. You could repeat a set of operations by using a GOTO to jump to the beginning of a block of code, and another to return, but basically programs were large blocks of unstructured code, with global data.

Procedures were the solution to this problem, and procedural programming was born. Procedures added a new form of indirection; they made it possible to have a program within a program, a set of operations acting on data either passed in via an argument list, or existing purely within the scope of the procedure. It became possible to perform a set of operations by simply making a procedure call. The code no longer needed to be explicitly included at every place it was used, or accessed via a GOTO, it could by indirectly triggered. Any changes made to the procedure's content immediately influenced the operation of all code making a call to that procedure.

User-defined types represented yet another form of indirection. Instead of declaring every variable used, at every point in the code it was needed, it was possible to group variables, and reference the group by a single identifier. Again, indirection helped increase program flexibility: Now you could add a variable to a type without changing every declaration and procedure call involving the type.

Hopefully you are convinced by now that indirection is everywhere. But why is that? What does indirection achieve, and how can you use it to make your software better? Indirection is about separating elements with an interface. For simple variables, the interface is the variable name: the variable name is used to refer to the memory represented by the variable, and the compiler maps this name to a particular memory address. This allows the compiler to decide where that memory should be, without influencing program operation. For a procedure, the interface is its argument list. The calling code, and the called code, each abide by this interface.

The elements on either side of an interface do not need to have any knowledge of one other, they only need to conform to the interface. The interface effectively decouples the elements; they are free to vary independently, as long as the interface is not changed. This is why indirection works: it reduces coupling, and increases flexibility. Changing any one aspect of the code does not necessarily require wholesale global changes. Changes can be localized, safe in the knowledge that they are shielded from the rest of the code by fixed interfaces.

Indirection is actually not specific to software development, it's everywhere. Take anatomy. Imagine going through life without an elbow. You couldn't move your hand independent of your shoulder. Your body would lose some of its flexibility, and many tasks would become more difficult or impossible. In this example, your elbow is an interface between your hand and shoulder. Your hand does not need to know that the shoulder exists; it simply needs to conform to the elbow's movement. The same applies to the shoulder. By introducing an interface between the two, your body becomes much more flexible, and you can perform actions such as moving your hand upwards, while your upper arm moves downwards, which would be impossible without an elbow.

So indirection is a good thing. It reduces coupling, increasing flexibility, making code maintenance easier, and enhancing the prospects of code reuse. Good software design is largely about building in clearly defined interfaces, decoupling elements from one another.

As with anything in life, you can have too much of a good thing. Indirection makes your code more flexible, building in the possibility of future change, but it comes at a price: performance. Generally, when you do something less directly, it takes longer, and that is just as true in software as any other field of endeavor.

When considering whether or not to introduce a form of indirection, unless it is blindingly obvious that it is a performance bottleneck, you should just do it. Your software will be better for it. Later, you can time your program, and if it turns out that the indirection in question consumes excessive resources, you can change the design to remove it. This is far more desirable than avoiding forms of indirection because you think they *may* impose unacceptable performance penalties.

Another reason you should not avoid indirection on performance grounds is that often it will lead to better performance. This seems to contradict what was said above, but it doesn't. Indirection in itself is usually slower, but the design improvements that result from indirection often make it easier to make performance-enhancing changes. This could actually make your software faster, as well as more flexible.

Here's an example to demonstrate this point: Imagine that we have two programmers, Mr. Hare and Ms. Tortoise, each of which must write a piece of software making heavy use of Linear Algebra. Mr. Hare abhors indirection of any form; he avoids writing subroutines, and programs every operation as a series of low-level, hand-optimized loops. His program screams! Ms. Tortoise builds lots of structure into her program; she wants it to be flexible for future change. Her program initially has the performance of molasses flowing uphill ... against a strong head wind.

Then someone mentions to each of them that there exists a library of routines called the 'Basic Linear Algebra Subprograms (BLAS)', which have been hand-optimized for virtually every computing architecture on the planet. Now Mr. Hare has a problem: much of his code needs to be rewritten, with each of those low-level loops being replaced by calls to BLAS. In the end, he decides it's too much work, and sticks with the solution he has. Tortoise, on the other hand, concentrated all of her Linear Algebra in a couple of subroutines, avoiding duplication of the Linear Algebra loop structures throughout her source code. Now she just has to modify those subroutines to make use of BLAS, and her whole program will run significantly faster. We all know how the race ends.

This story leads to a few conclusions:

- Indirection, and good design in general, often lead to better performance, because they make performance enhancements easier to implement.
- Performance of software developers may be just as important as performance of the software itself. Poor design can cause the cost of making changes or introducing new functionality to become excessive. If this means a piece of functionality does not get implemented at all, the performance of that particular functionality is effectively nil. The performance of functionality not implemented as the result of poor design should also be considered when measuring overall performance of a software package.
- Hares really should re-evaluate their strategy. They're never going to win a race the way things are going!

Use your common sense when it comes to applying indirection, and your programs will be all the better for it. If it is clear that a particular instance of indirection will be a death blow to your application's performance, avoid it, but only if you are 100% sure. Often you may think something will have a high cost, but it doesn't. And even if it does represent a relatively large portion of some operation, if the total time required to perform the operation is small, there is no point optimizing it.

6.5. Cohesion

Writing good software is about implementing a solution in a form easily consumed by human beings. Humans are perfectly capable of digesting small sections of code with many dependencies, but large, intra-dependent programs are another story. Such programs do not present any problem to a computer, but they quickly become gobbledegook to a Software Developer.

To achieve the objective of a human-readable solution, we need to break our large problem into many smaller pieces. The pieces themselves should be clusters of closely-related information, because that will help the human mind to digest them — randomly grouping information will not make the system any more understandable to humans. Each piece will need to work with a number of other pieces, together forming a net representing the solution to the problem.

Computer Scientists have assigned words to these simple ideas. The measure of how closely pieces of information are related is known as *cohesion*. Cohesion is about whether or not things belong together. Grouping together closely-related information increases cohesion, and grouping unrelated or weakly-related information decreases it.

Cohesion applies equally to procedures as user-defined types. A well written subroutine should be highly cohesive; it should contain operations and data that are very closely related, and strongly interdependent. Creating a subroutine that embodies more than one set of operations tends to decrease cohesion.

Take a simple example, from some imaginary Zoo administration software. The software in question may include user-defined types such as `Elephant` and `Zebra`. This is the design that most programmers would adopt, but a less-experienced programmer may be tempted to coalesce these two types to reduce the total number, creating an `ElephantOrZebra` type.

This is clearly a dumb idea, but why? With our new found vocabulary we can proffer an answer: cohesion. Grouping two weakly-related concepts together reduces the cohesion of the types involved. The `trunkLength` variable would be equally meaningless for a zebra as the `numberOfStripes` variable would be for an elephant. Half of the data would have no relation to the other half. This is a trivial example, and it won't always be as obvious which variables 'belong together', but hopefully it makes the concept of cohesion clear.

It is also important to consider cohesion of procedures. A procedure should encapsulate closely-related functionality, not a series of unrelated tasks. Rather than having one subroutine called `DoHousework`, why not break it into a series of more cohesive procedures such as `WashDishes`, `WashClothes`, and `DryClothes`. You may still want to keep the `DoHousework` subroutine — as a convenience — but its implementation would simply call the other three routines in turn.

6.6. Coupling

In Mathematics, two quantities are orthogonal if you can change one independent of the other. When you write software, you should be trying to achieve this in your design. It will never be fully achievable, but your goal should be to come up with a design that allows each type or procedure to vary as independently as possible from all others.

Orthogonality is closely tied to the concept of *coupling*. Two entities are said to be coupled if changing one requires the other to accommodate the change. When a Mathematician looks for a solution to a complex system, they will often try to find the *eigenvectors*. The eigenvectors are a representation of the system for which coupling is minimal. If you work with the eigenvectors, the problem usually becomes significantly simpler, because it reduces to a number of smaller, independent problems, rather than a large, highly-coupled one.

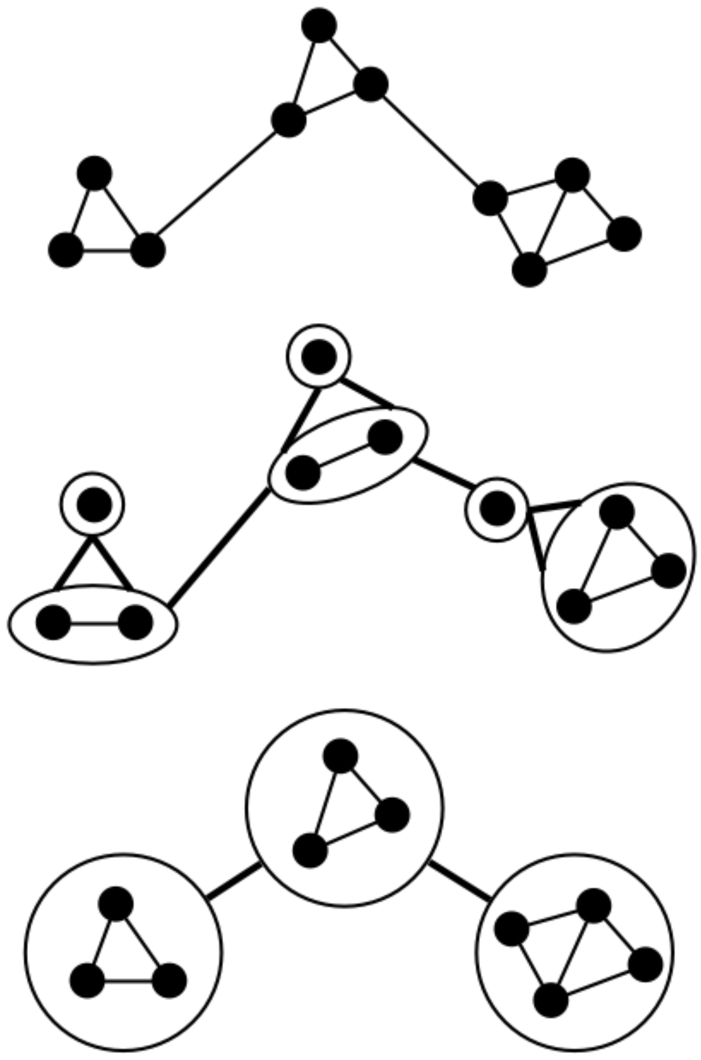
Your task as a programmer is to find the eigenvectors of your software. By reducing coupling between program entities, you make your software much more flexible, and easier to understand. With loosely-coupled types and procedures, changing one will not require you to change any others, and reusing a piece of code elsewhere will also become simpler.

So the flip side of cohesion is coupling. Increase cohesion in your programs, and you will generally be decreasing coupling at the same time. Finding a good solution is about finding the eigenvectors, which minimize coupling by definition. Aim to group closely-related data and operations together in a single procedure, and minimize the dependencies between user-defined types, and you will go a long way to writing robust and reusable software.

6.7. Granularity of Design

Cohesion and Coupling are closely related to another important concept in software design: granularity. How big should the units in a program be? Should you use lots of small types, or a few large ones? How long should procedures be? Strive for high cohesion and low coupling, and you should do quite well.

To elucidate the roles of cohesion and coupling, consider the figure below, which represents some dependencies in a piece of software.



Diagrammatical representation of the dependencies between entities in a program. It shows how a well-structured program (bottom) groups together highly interdependent program entities, and leaves only weak coupling between the groups.

The figure represents an unstructured program (top), a poorly-structured program (middle), and a well-structured program (bottom). The number of program elements, and the dependencies between them, are the same in each case. What the well-structured program manages to achieve is to group highly-interdependent elements together. The resulting groups are loosely coupled: the number of dependencies between them is minimized.

Achieving high cohesion (loose coupling) means *neither* creating many small types and short procedures, *nor* creating a few large types and long procedures. The right solution lies somewhere in between, and becoming a good software developer is largely about learning to strike the right balance.

7. Abstract Data Types

Structured programming involves identifying which programmatic entities belong together, and grouping them. At the lowest levels, this grouping is into user-defined types and procedures, but you can also group at a higher level, gathering together types and procedures into *abstract data types (ADTs)*.

An ADT is an entity that has both *state* and *behavior*. State corresponds to the data (variables) stored in the ADT, and behavior relates to its associated procedures. There is no formal language construct for an ADT in Fortran; instead, you can form an ADT by combining a module, a user-defined type, and some module procedures. In the coming sections we will see how this is done.

7.1. Attributes, Methods, and Messaging

The variables of an ADT, which are stored in a user-defined type, are called *attributes*. The attributes of an ADT define its state at any point in time.

The behavior of an ADT arises from its current state in combination with the procedures that it defines. The procedures of an ADT are called *methods*, and calling or *invoking* them is sometimes referred to as *messaging*, because it is similar to sending a message to the ADT.

Here is a simple ADT to make these definitions more concrete.

```
module LazyModule
   implicit none
   save
   private
   public LazyType, New, Delete
   public GetSleepTime, SetSleepTime, Sleep

   type LazyType
      private
      real :: sleepTime
   end type

   interface New
      module procedure NewPrivate
   end interface

  interface Delete
      module procedure DeletePrivate
   end interface

contains

   subroutine NewPrivate(self, time)
      type (LazyType), intent(out)   :: self
      real, intent(in)               :: time
      call SetSleepTime(self, time)
   end subroutine

   subroutine DeletePrivate(self)
      type (LazyType), intent(inout) :: self
   end subroutine

   real function GetSleepTime(self) result (time)
```

```
      type (LazyType), intent(in)    :: self
      time = self%sleepTime
   end function

   subroutine SetSleepTime(self, time)
      type (LazyType), intent(in)    :: self
      real, intent(in)               :: time
      self%sleepTime = time
   end function

   subroutine Sleep(self)
      type (LazyType), intent(inout) :: self
      print *, 'Sleeping ', self%sleepTime, ' minutes'
   end subroutine

end module
```

The only attribute of this ADT is `sleepTime`. The publicly accessible methods are `New`, `Delete`, `GetSleepTime`, `SetSleepTime`, and `Sleep`.

A common aspect of methods is that the first argument is a variable of the ADT's type, and is called 'self' by convention. The ADT methods operate on an *instance* of the ADT, the attributes of which are passed in via this first argument.

Other aspects of the ADT above will be discussed more in the coming sections.

7.2. Data Hiding

An important quality of an ADT is that it makes a clear distinction between *interface* and *implementation*. The interface of an ADT is anything that is directly accessible from outside the ADT's module. The implementation is the internal part of the ADT, which should be hidden from the outside, and should only be accessible indirectly via the interface. In general, the interface of an ADT should only consist of public procedures — data (attributes) should not be publicly accessible.

The practice of preventing direct access to an ADT's attributes is known as *data hiding* or *encapsulation*. In Fortran, data hiding involves declaring the contents of the ADT's user-defined type to be private, as shown earlier.

```
   type LazyType
      private
      real :: sleepTime
   end type
```

The `private` keyword here applies to the variables declared inside the type, preventing them from being accessed outside of the module. The type itself, `LazyType`, can be used outside the module, because it was declared `public`, but no direct access to its contents is possible.

The purpose of data hiding is to reduce dependencies and couplings in a program. The implementation of an ADT is free to vary independent of its interface. If you, as developer, know that the attributes of an ADT are hidden, then it is much easier to change them or work with them, without having to worry about side-effects outside of the module.

7.3. Accessors

Of course, an ADT wouldn't be very useful if it didn't provide at least some indirect access to its internal state. There needs to be a means to access relevant data without violating the encapsulation of the ADT. The solution is to introduce procedures whose sole purpose is to provide indirect access to data stored in the ADT. Such procedures are known as *accessors* or *accessor methods*.

Accessors are generally very simple, and can be separated into two classes: *getters* and *setters*. A getter is an accessor that retrieves the value of an attribute from the ADT instance and returns it to the caller. A setter is a procedure that allows the value of an attribute to be changed.

In the example above, the method `GetSleepTime` was a getter, returning the value of the `sleepTime` attribute.

```
   real function GetSleepTime(self) result (time)
      type (LazyType), intent(in)    :: self
      time = self%sleepTime
   end function
```

The setter in the example was `SetSleepTime`.

```
   subroutine SetSleepTime(self, time)
      type (LazyType), intent(in)    :: self
      real, intent(in)               :: time
      self%sleepTime = time
   end function
```

You're probably wondering what the point of such methods is if all that they do is set or get a value. For instance, why don't we just make `sleepTime` public and retrieve its value directly from the type? The answer is that it is not what accessor methods do in the present that is so important, as much as what they could do in the future. Accessors build in a degree of indirection, which makes it possible to change the implementation of an ADT without altering code outside the module.

Take a simple example like changing the name of the `sleepTime` variable. With the variable private, and the accessors in place, this is a simple operation localized in one module — it may even be possible to achieve it with one find-and-replace command. But if `sleepTime` were to be public, and no accessor methods were in place, the task could be considerably more difficult: You may be required to search through the whole program, perhaps many files, locating references to `sleepTime` and changing them all.

That might not sound like a big problem, but that is only the tip of the iceberg. Take a more extreme case: you want to print out a message every time the `sleepTime` is retrieved, perhaps for the purpose of debugging. With a getter in place, this is trivial:

```
   real function GetSleepTime(self) result (time)
      type (LazyType), intent(in)    :: self
      print *,'Getting sleepTime'
      time = self%sleepTime
   end function
```

But if `sleepTime` is public, you must again track down all of the occasions that it is used, and at each point insert a `print` statement. When you are finished debugging, you have to remove all of the `print` statements again. Aside from the time this takes, the probability of making a mistake — and introducing a bug — is quite high.

7.4. Mutability

People new to structured programming have a tendency to write accessor methods for all of the attributes in a particular ADT. In fact, it is much better to write as few as possible — only the ones that are absolutely necessary to allow the ADT to be used. The reason for this is that it again improves data hiding. The smaller the interface of an ADT, to more loosely coupled it is, and the more flexible it becomes.

Many attributes will need to be accessed from outside the ADT, and for these it is appropriate to include a getter method. But just because there is a getter, does not mean you should include a setter. Including a setter is often unnecessary, and can complicate the implementation of an ADT, because the programmer has to take into consideration that attributes may change at unexpected times. When there is no setter, the programmer *knows* the attribute can only change when it is changed inside the ADT.

Whether or not you include a setter methods relates to the ADTs *mutability*. Mutability is about whether attributes can change after they have been initialized. As discussed above, making an ADT *immutable* — unchanging — often makes its implementation easier to program. A *mutable* ADT, on the other hand, can be more challenging. *When you have the choice, make your ADTs immutable by leaving out setters.*

Just because there are no setter methods does not mean that an attribute can never be set; it just means it has to be set when the ADT instance is initialized. *Initialization* of an ADT, which is also known as *construction*, will be discussed in the next section.

Exercise: Setters, Getters, and Mutability

Look in the cmc program that accompanies this course and locate the file HistogramBuilder.f90. Open it in a text editor, and try to locate its accessor methods. Is the HistogramBuilder ADT mutable or immutable?

Take a closer look at the getters GetBinCenters and GetBinWidth. How do these differ from the other getters? What does this tell you about the correspondence between getters and variables in an ADT?

Modify the HistogramBuilder ADT so that bin width is stored as a variable, rather than calculated on-the-fly. Compile cmc with your changes.

How many changes did you need to make outside of the HistogramBuilder ADT to get the new implementation working? What does this tell you about the benefits of accessor methods?

7.5. Construction and Destruction

When you create a new ADT variable — an *instance* of the ADT — you need to initialize it so that it is in a valid state. Similarly, when the instance is no longer needed, any resources it has, such as open files or allocated arrays, must be cleaned up. Because these aspects are common to all ADTs, a conventional set of methods known as *constructors* and *destructors* are used to initialize and delete instances.

A constructor is usually called `New` or `Init`. In Fortran, we can overload the procedure name so that all ADT instances are initialized by the same procedure call. The Lazy ADT shown earlier includes a constructor.

```
module LazyModule

   ...

   interface New
      module procedure NewPrivate
   end interface

   ...

contains

   subroutine NewPrivate(self, time)
      type (LazyType), intent(out)   :: self
      real, intent(in)               :: time
      call SetSleepTime(self, time)
   end subroutine

...
```

In this particular case, the constructor takes a single argument, but a constructor can take any number of arguments. It is even possible to have more than one constructor for a single ADT, with different arguments for each.

The role of the constructor is to initialize the ADT user-defined type so that it is in a valid state. This could involve setting variables, opening files, and allocating arrays.

The destructor should do the opposite, closing any open files and deallocating memory. The destructor of the Lazy ADT actually does nothing.

```
   subroutine DeletePrivate(self)
      type (LazyType), intent(inout) :: self
   end subroutine
```

Why include a destructor if it doesn't do anything? A destructor should always be supplied, even if it is empty, because at some point in the future it may be needed. If no destructor is provided, and it doesn't get called, then it will be necessary to add many calls to the destructor later if one is introduced. It is better to provide an empty destructor and make the appropriate calls to it from the beginning.

Here is a small program that demonstrates how an ADT instance is created, used, and disposed using constructors and destructors.

```
use LazyModule
type (LazyType) :: lazy
call New(lazy, 5.0)   ! Sleeps for five minutes
call Sleep(lazy)
call Delete(lazy)
end
```

To make use of an ADT, you first need to use its module. Then you can define variables (instances) of the ADT. The instance needs to be constructed by calling `New` before it can do anything else. After construction, methods like `Sleep` can be called, and when the instance is no longer needed, `Delete` should be called. (Note that, as previously discussed, each call passes the ADT instance as the first argument.)

7.6. Copy Constructors

For many ADTs, it is useful to be able to make new copies of instances. A *copy constructor* makes this easy. Consider the following ADT:

```
module StoreModule

   ! An ADT that stores an array of real numbers

   implicit none
   save
   private
   public StoreType, New, Delete, GetData, SetData

   type StoreType
      private
      real, pointer :: data(:)
   end type

   interface New
      module procedure NewPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

contains

   subroutine NewPrivate(self, dataLength)
      type (StoreType), intent(out)   :: self
      integer, intent(in)             :: dataLength
      allocate(self%data(dataLength))
      self%data = 0.0
   end subroutine

   subroutine DeletePrivate(self)
      type (StoreType), intent(inout) :: self
      deallocate(self%data)
   end subroutine

   subroutine GetData(self, data)
      type (StoreType), intent(in)    :: self
      real, intent(out)               :: data(:)
      if ( size(data) < size(self%data) ) stop 'Array argument too small in GetData'
      data(:size(self%data)) = self%data
      data(size(self%data)+1:) = 0.0   ! Zero any excess
   end subroutine

   subroutine SetData(self, data)
      type (StoreType), intent(in)    :: self
      real, intent(in)                :: data(:)
      if ( size(data) > size(self%data) ) stop 'Array argument too big in SetData'
      self%data(:size(data)) = data
      self%data(size(data)+1:) = 0.0   ! Zero any excess
   end subroutine
end module
```

The Store ADT does not have a copy constructor yet, so let's add one.

```
module StoreModule

   ...

   interface New
      module procedure NewPrivate
      module procedure NewCopyPrivate
   end interface

   ...

contains

   ...

   subroutine NewCopyPrivate(self, other)
      type (StoreType), intent(out)   :: self
      type (StoreType), intent(in)    :: other
      integer                         :: n
      n = size(other%data)
      allocate(self%data(n))
      self%data = other%data
   end subroutine

   ...

end module
```

A copy constructor takes two arguments: the first — as always — is the instance being initialized; the second is the instance being copied. Generally, the implementation of a copy constructor involves copying the attributes of the second argument into the first argument. If there are pointer arrays, as above, you first need to allocate them to be the same size as their counterparts in the second instance.

A copy constructor is called in the same way as other constructors, and any instance created with a copy constructor also needs to be deleted. Here is a program that demonstrates how to use the Store ADT copy constructor.

```
use StoreModule
type (StoreType) :: a,b
real             :: temp(3)

! Initialize a
call New(a, 3)
call SetData(a, (/1.0, 2.0, 3.0/))

! Initialize b as copy of a
call New(b, a)

! Get data from b, and print out
call GetData(b, temp)
print *, 'Data in b is ', temp

! Delete a and b
```

```
call Delete(b)
call Delete(a)

end
```

7.7. Assignment

A copy constructor is one way to copy the information in one ADT into another one. Another way is to use the assignment operator.

If you have a simple ADT, you may not even need to do anything to get the assignment operator to work. For example, the Lazy ADT that was introduced above only includes a single real attribute, so you can assign instances of the ADT without having to add any special code.

```
use LazyModule
type (LazyType) :: lazy1, lazy2
call New(lazy1, 5.0)
call New(lazy2, 10.0)
lazy2 = lazy1
call Sleep(lazy2)   ! Will sleep for 5 minutes
call Delete(lazy1)
call Delete(lazy2)
end
```

Here, the assignment resulted in the `lazy2` variable copying across the sleep time of `lazy1`.

If an ADT includes pointers, you need to be more careful, and will have to add a special assignment procedure. Let's do that for the Store ADT introduced earlier.

```
module StoreModule

   ...

   public assignment(=)

   interface assignment(=)
      module procedure AssignPrivate
   end interface

   ...

contains

   ...

   subroutine AssignPrivate(self, other)
      type (StoreType), intent(inout) :: self
      type (StoreType), intent(in)    :: other
      call Delete(self)
      call New(self, other)
   end subroutine

   ...

end module
```

The difference between an assignment and a copy construction is that in an assignment it is assumed that both arguments are already initialized. That means that before you can copy the contents of the second instance into the first, you have to delete anything in the first instance. In the example above, this is achieved by calling the `Delete` method, though you could also do it explicitly by deallocating the `data` array pointer. Once the attributes of the first instance have been 'cleaned up', you can begin initializing it again with the contents of the second instance. In the example, this is achieved by simply calling the copy constructor.

With a custom assignment operator routine in place, assigning two Store instances is very simple, even though behind the scenes this involves deallocation, allocation, and assignment of arrays.

```
use StoreModule
type (StoreType) :: a,b
call New(a, 3)
call New(b, 10)
call SetData(a, (/1.0, 2.0, 3.0/))
b = a
call Delete(a)
call Delete(b)
end
```

Exercise: A Matrix ADT

Imagine you are charged with writing an ADT to represent a square matrix type called ... wait for it ... 'SquareMatrix'. It should store real data in a two-dimensional array, with the dimensions passed to the constructor. It does not need to be mutable, so the size of the matrix never changes.

Write a module for the SquareMatrix ADT, including any appropriate constructors, destructors, and accessor methods. Include a copy constructor and make sure the ADT will work properly when used with the assignment operator.

Write a short program to test the ADT.

7.8. Relationships

ADTs in isolation, like the ones we have seen to date, are not very useful. ADTs become much more useful when you build up relationships between them. This section is about the types of relationships you can have, and how you implement them.

The most basic type of relationship is *containment*. This is when one ADT includes another ADT as one of its variables. The following ADT, StoreManager, contains an array of instances of the Store ADT introduced above.

```
module StoreManagerModule
   use StoreModule
   implicit none
   save
   private
   public New, Delete, SetDataInStore, GetDataInStore

   type StoreManagerType
      private
      type (StoreType), pointer :: stores(:)
   end type
   interface New
      module procedure NewPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

contains

   subroutine NewPrivate(self, numStores, dataLength)
      type (StoreManagerType), intent(out)   :: self
      integer, intent(in)                    :: numStores, dataLength
      integer                                :: i
      allocate(self%stores(numStores))
      do i = 1, size(self%stores)
         call New(self%stores(i), dataLength)
      enddo
   end subroutine

   subroutine DeletePrivate(self)
      type (StoreManagerType), intent(inout) :: self
      integer                                :: i
      do i = 1, size(self%stores)
         call Delete(self%stores(i))
      enddo
      deallocate(self%stores)
   end subroutine

   subroutine GetDataInStore(self, storeIndex, data)
      type (StoreManagerType), intent(in)    :: self
      integer, intent(in)                    :: storeIndex
      real, intent(out)                      :: data(:)
      call GetData(self%stores(storeIndex), data)
   end subroutine

   subroutine SetDataInStore(self, storeIndex, data)
      type (StoreManagerType), intent(in)    :: self
      integer, intent(in)                    :: storeIndex
      real, intent(in)                       :: data(:)
      call SetData(self%stores(storeIndex), data)
   end subroutine

end module
```

A containment relationship can be *one-to-one* or *one-to-many*. In this example, *one* StoreManager has *many* Store instances, meaning the relationship is one-to-many. In a containment relationship, the contained ADT instance(s) 'belongs to' the container instance.

The container is responsible for constructing and destructing the contained instances. This usually occurs in the container ADT's own constructor/destructor. In the example above, the array of Store instances was allocated, and each Store initialized, in the StoreManager constructor; each Store was deleted, and the array deallocated, in the destructor.

Containment is appropriate when one object takes complete control of another object, including its creation and destruction. But often an object has to be shared between several other objects. In that instance, containment is no longer appropriate, and the more general *association* relationship should be used. This is implemented by including pointer variables in each ADT.

```
module ChemistryModule

   type ChemicalElementType
      character(3) :: symbol
      real         :: atomicMass
   end type

   type AtomType
      type (ChemicalElementType), pointer :: element
      real                                :: position(3)
   end type

end module
```

In this example, the `ChemicalElementType` represents an element in the periodic system (*eg*, Carbon, Oxygen, Hydrogen). Many atoms can be associated with a single element, so it is not appropriate to say that an element belongs to any one atom. For this reason, an association relationship — implemented as a pointer variable — is the best approach.

When you are setting up association relationships, you need to be careful how you create the instances in the relationship. In particular, any instance that may be the target of a pointer should initially be declared as a pointer variable, like this

```
program Chemistry
   use ChemistryModule
   type (ChemicalElementType), pointer :: oxygen
   type (AtomType)                     :: oxygenAtom

   ! Initialize element
   allocate(oxygen)
   oxygen%symbol = 'O'
   oxygen%atomicMass = 15.9994

   ! Initialize atom
   oxygenAtom%element => oxygen
   oxygenAtom%position = (/0.0, 0.0, 0.0/)

   ! Print out details of the atom's element
   print *, 'Atoms''s element is ', oxygenAtom%element%symbol

   ! Clean up
   deallocate(oxygen)
end
```

For the sake of brevity, all methods have been excluded from this example, but they should be included in any real-world program. The example shows that the `ChemicalElementType` instance `oxygen` is actually declared as a pointer variable, and thus needs to be allocated before use. The reason it's a pointer is that it later becomes the target of the pointer variable `element` in the `oxygenAtom` instance; in Fortran, a pointer variable can only be associated with another pointer, or a variable with the `target` attribute.

This example shows how you can implement a *to-one* relationship using a pointer variable, but how do you represent *to-many* relationships? To-Many relationships are implemented as arrays of pointers, but there are some complications which arise due to few Fortran idiosyncrasies.

To demonstrate a to-many relationship, imagine that we wish to implement the inverse relationship of the atom–element relationship above. In other words, we want to have a relationship that gives us all of the atoms associated with a particular chemical element. Here's how that could be implemented.

```
module ChemistryModule

   type AtomType
      type (ChemicalElementType), pointer :: element
      real                                :: position(3)
   end type

   type AtomPointerType
      type (AtomType), pointer        :: atom
   end type

   type ChemicalElementType
      character(3)                    :: symbol
      real                            :: atomicMass
      type (AtomPointerType), pointer :: atomPointers(:)
   end type

end module
```

An auxiliary type has been introduced called `AtomPointerType`; all this does is store a pointer to an `AtomType`. This apparently futile exercise is necessary because of limitations in Fortran's syntax: there is no direct way to declare a variable as a pointer array containing `AtomType` pointers. If you could, it might look something like this

```
type (AtomType), pointer, pointer :: atoms(:)
```

Hopefully you can see the problem. The way around it is to make a new type that contains only a single pointer variable, and to form the array from this new (non-pointer) type.

With this new to-many relationship, the earlier program could evolve into this:

```
program Chemistry
   use ChemistryModule
   type (ChemicalElementType), pointer :: oxygen
   type (AtomType), pointer            :: oxygenAtom1, oxygenAtom2

   ! Initialize element
   allocate(oxygen)
   oxygen%symbol = 'O'
   oxygen%atomicMass = 15.9994

   ! Initialize atoms
   allocate(oxygenAtom1, oxygenAtom2)
   oxygenAtom1%position = (/0.0, 0.0, 0.0/)
   oxygenAtom2%position = (/0.0, 1.0, 0.0/)

   ! Setup relationships
   oxygenAtom1%element => oxygen
   oxygenAtom2%element => oxygen
   allocate(oxygen%atomPointers(2))
   oxygen%atomPointers(1)%atom => oxygenAtom1
   oxygen%atomPointers(2)%atom => oxygenAtom2
   ! Print out some details of atoms and elements
   print *, 'Atoms 1 has element ', oxygenAtom1%element%symbol
   print '(a,x,i2,x,a)', 'There are', size(oxygen%atomPointers), 'oxygen atoms'

   ! Clean up
   deallocate(oxygenAtom1, oxygenAtom2)
   deallocate(oxygen%atomPointers)
   deallocate(oxygen)
end
```

(Again, this example is very oversimplified for the sake of brevity: in any real-world application, many of the statements condensed into this program would be moved to constructors, destructors, and accessor methods.)

We now see two oxygen atoms being created, and the to-many relationship from element to atom storing each of these. Note also that an extra allocate–deallocate pair is needed, to allocate memory for the relationship's pointer array. Remember too that the array pointer, atomPointers, must be deallocated *before* the element variable `oxygen`, otherwise a crash or memory leak may arise.

One final issue needs to be addressed before this section concludes, and that is *ownership*. When ADT instances partake in a containment relationship, it is obvious which is the owner and which the owned, but in general association relationships, this may not be clear. It is important that you decide who 'owns' a particular instance, and document this in your program's comments. The owner is responsible for destructing the instance when it is finished with.

Sometimes you will create an instance in one ADT, and delete it in another. This involves a transfer of ownership, and should be made very clear in the comments, to prevent both ADTs deleting the same instance by mistake, or confusing a programmer that is new to the code. If you can, it is better to delete an instance in the same part of the program that you create it, but this will not always be practical.

Exercise: One for All and All for One

Look in the cmc program that accompanies this course and locate the file ParticleEnsemble.f90. The ADT in this file includes a to-many containment relationship: each `ParticleEnsemble` contains many `Particle` instances. Look through the code and try to understand how the `ParticleEnsemble` ADT handles this relationship. Some quite advanced techniques are used to handle the fact that the number of `Particles` can grow in time. What are they?

7.9. Design

We've now covered how you structure ADTs, and relate them to one another. Something that has not been covered is how you identify them: Give a particular problem, what should the ADTs be?

This falls under the heading of 'Software Design'. There are a few simple tips that can help you design your programs. The first is simply to keep in mind the concepts that were introduced in the earlier on, concepts like cohesion, indirection, and coupling.

The second is more concrete: write out a one or two paragraph description of the program, what it does, and how it works. Having done this, go through and underline all of the nouns — the nouns are good candidates to be ADTs. The verbs are often good candidates for ADT procedures.

Here's a simple example of applying this technique. Consider the following description of a Density Functional Theory (DFT) program:

> DeFuncT is a program that solves the Kohn-Sham equations for electrons moving in one-dimension. Electronic orbitals are represented in a basis set of Slater functions, and move under the influence of a positively charged nucleus.
>
> The Kohn-Sham equations include the following operators: the kinetic energy operator, the external potential due to the nucleus, the electron-electron Coulomb potential, and the exchange-correlation potential. These are evaluated for a given density, with corresponding single-particle potential, on a numerical grid. The KS equations are solved self-consistently via an iterative process, until the density converges.

The underlined terms are nouns that would translate well into ADTs. Describing software in this way naturally conglomerates entities that belong together. Because a major software design principle is to ensure that ADTs contain elements that exhibit high cohesion, this technique should give you a reasonable first-up design almost every time. You will inevitably need to change and expand things as you go, but you will at least have a basis from which to work.

Exercise: Confusion Monte Carlo Design

Read through the design specifications of the Confusion Monte Carlo program, which are given in an appendix. Try to understand how the description of the numerical algorithm maps to the ADTs in the final program. Which ADTs are obvious from the description, and which are more obscure?

8. Advanced Fortran

In this part of the course, we'll cover a few very advanced techniques in Fortran programming. It is intended for those that are comfortable with the preceding sections, and would like to go a bit further.

8.1. Callbacks

In this section, we'll take a look at an important *design pattern*. Design patterns are particular ways of combining program elements that recur over and over. When you learn to recognize design patterns, it can help you more quickly understand how a program works, and think up better solutions to common problems.

The design pattern we are going to look at is the *callback*. Callbacks are useful when a procedure needs its caller to do something. You are probably used to thinking the other way around — the caller needs the procedure to do something — but sometimes they need each other: The caller calls the procedure to do something, but the procedure itself needs something from the caller to proceed. How can you get around this? With a callback, of course.

A callback is a procedure that gets passed as an argument to another procedure. Fortran allows you to achieve this using an interface block.

```
module PrintModule
contains
   subroutine PrintValues(ValueFunc, coords)
      real, intent(in) :: coords(:)
      integer          :: i
      interface
         real function ValueFunc(x)
            real, intent(in) :: x
         end function
      end interface

      do i = 1, size(coords)
         print *, ValueFunc(coords(i))
      enddo

   end subroutine
end module

module MathFuncsModule
contains
   real function QuadraticFunc(x)
      real, intent(in) :: x
      QuadraticFunc = x*x
   end function
end module

program Values
   use PrintModule
   use MathFuncsModule
   call PrintValues(QuadraticFunc, (/0.0, 1.0, 2.0, 3.0/))
end program
```

The `PrintValues` subroutine takes two arguments: one is an array of coordinates (`coords`), and the other is a function (`ValueFunc`). To tell the compiler what the interface of `ValueFunc` is —

what it's arguments are, whether it is a function or subroutine etc. — an interface block is used. This looks a bit like the interface blocks that you use in a module to overload a procedure name, but here the interface block is used to declare the procedure's interface.

```
      interface
         real function ValueFunc(x)
            real, intent(in) :: x
         end function
      end interface
```

The interface declaration is simply a procedure with no content, except that which is necessary to declare its arguments. This could, and often does, require the interface to use modules, in order to make a complete declaration of the interface.

Once the procedure argument has been declared, it can be used just like any other procedure. In this example, `PrintValues` loops over coordinates, calling `ValueFunc` once for each, and printing the result.

Passing a procedure as an argument is just as trivial: you just give its name in the call. The only catch is that all interfaces must be explicit. Usually that means ensuring that the procedure being passed, and the procedure being passed to, are both in modules.

In the example, a function called `QuadraticFunc` — which has exactly the same interface as `ValueFunc` — is passed to `PrintValues`. The net result is `PrintValues` prints out the quadratic function over a range of coordinates.

The callback design pattern can be very powerful. The simple example above shows how you could write a generic printing subroutine that would work with any one-dimensional mathematical function. And one particularly attractive aspect of the callback is that it encourages loose coupling, because the procedure that is passed the callback need know nothing more about it than its interface. For example, if you wrote a `SineFunc` function, you could pass it to `PrintValues`, and it would work without modification.

Exercise: Trigonometric Callback

Add a SineFunc function to the example above, which calculates the value of sine at the coordinate passed in. Use PrintValues to print the value of sine at 50 values between 0 and π. (Hint: You can calculate the value of π as `acos(-1.0)`.)

Did you have to make any changes to `PrintValues`? What does this tell you about the coupling between `PrintValues` and `SineFunc`?

Callbacks won't always be as simple as shown above. For example, sometimes you will need to pass some extra data to the callback function for it to perform its calculation. But the data passed may depend on which procedure is passed as the callback argument.

By way of example, consider this variation on the `MathFuncsModule` above.

```
module MathFuncsModule
contains

   real function QuadraticFunc(x, forceConst)
      real, intent(in) :: x
      real, intent(in) :: forceConst
      QuadraticFunc = 0.5 * forceConst * x**2
   end function

   real function CosineFunc(x, wavenumber, amplitude)
      real, intent(in) :: x
      real, intent(in) :: wavenumber, amplitude
      CosineFunc = amplitude * cos(wavenumber * x)
   end function

end module
```

The problem here is that the two functions no longer have the same interface — they each have a different set of arguments. How can you use functions with differing interfaces in a callback, which must have a fixed interface? The answer — as is often the case — is to introduce some indirection.

The first thing we'll do is add an extra argument to the callback. We will use this argument to store any extra data that is needed by the callback to perform its calculation.

```
module GenericDataModule
   type DataType
      character(1) :: notused
   end type
   type (DataType), parameter :: DataPrototype(1) = DataType('')
end module

module PrintModule
   use GenericDataModule
contains
   subroutine PrintValues(ValueFunc, coords, data)
      real, intent(in)            :: coords(:)
      integer                     :: i
      type (DataType), intent(in) :: data(:)
      interface
         real function ValueFunc(x, data)
            use GenericDataModule
            real, intent(in)            :: x
            type (DataType), intent(in) :: data(:)
         end function
      end interface

      do i = 1, size(coords)
         print *, ValueFunc(coords(i), data)
      enddo

   end subroutine
end module
```

The extra argument is designed to hold *generic* data, that is any data that the callback needs, regardless of its type. The argument itself is an array of the type `DataType`, and `DataType` is declared as a simple type containing a one character string variable. The specific type used for this generic data is actually not important at all; the only thing that is important is that it is an array. We'll find out why shortly.

The implementation of `PrintValues` now also includes the same generic data argument. `PrintValues` doesn't know — or care — what is in this argument, because it never accesses it directly itself. Instead, it just passes it on to the callback in the function call.

Because the interface of the callback function has been changed, we need to introduce new *wrapper* functions that fit the interface required by `PrintValues`, and know how to call the original math functions.

```
module MathFuncsModule
   use GenericDataModule

   type CosineArgsType
      real :: wavenumber, amplitude
   end type

contains
   real function QuadraticFunc(x, forceConst)
      real, intent(in)           :: x
      real, intent(in)           :: forceConst
      QuadraticFunc = 0.5 * forceConst * x**2
   end function

   real function GenericQuadraticFunc(x, data)
      real, intent(in)           :: x
      type(DataType), intent(in) :: data(:)
      real                       :: forceConst
      forceConst = transfer(data, forceConst)
      GenericQuadraticFunc = QuadraticFunc(x,forceConst)
   end function

   real function CosineFunc(x, wavenumber, amplitude)
      real, intent(in)           :: x
      real, intent(in)           :: wavenumber, amplitude
      CosineFunc = amplitude * cos(wavenumber * x)
   end function

   real function GenericCosineFunc(x, data)
      real, intent(in)           :: x
      type(DataType), intent(in) :: data(:)
      type(CosineArgsType)       :: args
      args = transfer(data, args)
      GenericCosineFunc = CosineFunc(x, args%wavenumber, args%amplitude)
   end function
end module
```

The wrapper functions are called `GenericQuadraticFunc` and `GenericCosineFunc`. They're role is to act as the callback function, and use the original math functions to calculate a result. In order to do this, each must take the generic data argument and extract from it the arguments that are needed by the corresponding math function. For example, `QuadraticFunc` takes a force constant argument, so `GenericQuadraticFunc` must extract this real number from the generic data. But how?

Fortran supplies an intrinsic function that can be used to convert data between two different types: `transfer`. `transfer` converts its first argument to the type of the second argument, and returns the result. (The value of the second argument is never used, only its type.) Importantly, if the second argument is an array, `transfer` will return an array that is big enough to hold the data of the first argument.

To demonstrate how `transfer` works, consider this very short program, which copies the data of a real array into an integer array, and then back again.

```
real                 :: a(10)
integer, allocatable :: b(:)
integer              :: s
a = 0.1
s = size(transfer(a,b))
allocate(b(s))
b = transfer(a,b)
a = transfer(b,a)
deallocate(b)
print *,a
end
```

After initializing the real array, we need to determine how big an integer array needs to be to store the real data from the array `a`. To do this, we embed the call `transfer(a,b)`, which forms an array of type `b` that is big enough to hold the data in `a`, in a `size` function that calculates how many elements are in the returned array. The size calculated is then used to allocate the array `b`.

With `b` the right size, `transfer` can be used again, with the result this time stored in array `b`. Used a third time, with the arguments reversed, `transfer` will convert the data in `b` back into real data in `a`, which can then be printed. This shows that `transfer` provides a means of converting data to a generic form, and then back again, which makes it very useful for callbacks.

Returning to the main example, consider the `GenericQuadraticFunc` function:

```
   real function GenericQuadraticFunc(x, data)
      real, intent(in)           :: x
      type(DataType), intent(in) :: data(:)
      real                       :: forceConst
      forceConst = transfer(data, forceConst)
      GenericQuadraticFunc = QuadraticFunc(x,forceConst)
   end function
```

As you can see, `transfer` is used here to extract the force constant from the generic data array passed in.

In `GenericCosineFunc`, multiple values are extracted; to store multiple values in a single argument, they are packed into a user-defined type.

```
   real function GenericCosineFunc(x, data)
      real, intent(in)           :: x
      type(DataType), intent(in) :: data(:)
      type(CosineArgsType)       :: args
      args = transfer(data, args)
      GenericCosineFunc = CosineFunc(x, args%wavenumber, args%amplitude)
   end function
```

All of this shows how the callbacks extract arguments from generic data and use it to calculate a result, but how is the generic data formed in the first place? The main program is responsible for that.

```
program Values
   use PrintModule
   use MathFuncsModule
   use GenericDataModule
   real, parameter  :: pi = acos(-1.)
   type(DataType), allocatable :: data(:)
   type(CosineArgsType)        :: args
   integer                     :: lengthData
   lengthData = size(transfer(args, DataPrototype))
   allocate(data(lengthData))
   args = CosineArgsType(1.0,1.0)
   data = transfer(args, DataPrototype)
   call PrintValues(GenericCosineFunc, pi * (/ (0.25 * i, i = 0,4) /), data )
   deallocate(data)
end program
```

This main program shows how `PrintValues` can be used to print out values of cos(x). The wavenumber and amplitude of the cosine function are packed into an instance of the type `CosineArgsType`, and `transfer` is used to convert this into generic data. This data is then passed to `PrintValues`, which passes it back to the callback function `GenericCosineFunc`, and the cycle is complete.

This example is quite complex, and is not really necessary in such a simple program. However, when programs get larger, the small amount of extra code that you have to write in order to program the callback design pattern is well worth the trouble. Larger programs can quickly become a mess, making it very difficult to navigate; using callbacks, you can reduce couplings, and make you programs more modular and flexible.

To finish off this section, we should consider one more application of the `transfer` function to callbacks. Often, you will want to pass more than just a few numerical arguments to a callback function. You may want to pass a very large type, or even an ADT instance that is shared. In such cases, it may make more sense to pass a pointer variable to the callback.

The trouble is, when you use the `transfer` function with a pointer variable, it won't convert the pointer itself, but the thing that it is pointing at. It's a bit like the problem of storing pointer variables in an array, which was discussed earlier in the course.

Just as for an array of pointers, the solution is to create a simple user-defined type to store the pointer in question, and to pass that to `transfer`. For example, imagine you have a shared instance of the type `BigType`, and want to pass it via a pointer to a callback. You could declare the pointer container type as follows:

```
type BigType
   ...
end type

type BigPointerType
   type (BigType), pointer :: p
end type
```

The pointer container type would then be used to store the pointer to the `BigType` instance, and could be converted to generic data by `transfer`.

```
type(BigType), pointer      :: big
type(BigPointerType)        :: bigPointer
type(DataType), allocatable :: data(:)
integer s
bigPointer%p => big
s = size(transfer(bigPointer, data))
allocate(data(s))
data = transfer(bigPointer, data)
call DoSomethingWithCallback(BigTypeCallback, data)
```

```
deallocate(data)
end
```

The `BigType` pointer can be retrieved in the callback function (`BigTypeCallback`) like this

```
subroutine BigTypeCallback(data)
   real, intent(in)       :: data(:)
   type(BigType), pointer :: big
   type(BigPointerType)   :: bigPointer

   ! Extract pointer from generic data argument
   bigPointer = transfer(data, bigPointer)
   bigType => bigPointer%p

   ! Use bigType
   ...
end subroutine
```

Exercise: Finite-Differencing with Callbacks

A finite-difference scheme is a simple way to numerically approximate the derivative of a function. The first derivative is given by f'(x) = [f(x+dx) – f(x-dx)] / (2 dx), where dx is a small step.

Write a Fortran procedure that calculates the first derivative of a callback function passed to it via the argument list. In addition to the standard coordinate (x) arguments, include an extra 'generic' argument that can be used to pass additional information to the callback function.

Test you differentiator by writing a callback that calculates the expression 'a.sin(bx)', with a and b being passed via the generic data argument. Write a main program to calculate and print the derivative of 2sin(x) at x = π / 3. Compile and run the program. Was the result correct?

8.2. Protocols

A *protocol* is a programming construction intended to reduce coupling by introducing a well-defined procedural interface. It is a bit like a generalized callback: In the callback pattern, any procedure can be used that fits the callback interface declaration; in the protocol pattern, any ADT can be used that includes all of the procedure interfaces in the protocol.

There is no direct support for protocols in Fortran 90/95, though they have been added in Fortran 2003 under the guise of 'abstract interfaces'. Even without direct support, you can use tools to mimic protocols, and still reap the benefits. One such tool is Forpedo, which has been supplied with this course, and will be used in the examples that follow.

Let's begin by defining our objectives. When we introduced callbacks in the previous section, we were able to write subroutines that were decoupled from the procedures they were calling to perform their calculations. For example, a procedure to print out function values at certain coordinates did not need to know anything about the function it was calling — only its interface was needed.

We now want to perform the same feat with ADTs. We want to be able to write code that makes use of an ADT that it knows nothing about, other than the procedures it declares. To elaborate, imagine we have two ADTs representing domestic cats and dogs.

```
module DogModule
   implicit none
   save
   private
   public DogType, MakeSound, IncreaseAge

   type DogType
      integer ageInYears
   end type

   interface MakeSound
      module procedure MakeSoundDog
   end interface

   interface IncreaseAge
      module procedure IncreaseAgeDog
   end interface

contains

   subroutine MakeSoundDog(self)
      type (DogType)                                    :: self
      print *,'Woof!'
   end subroutine

   subroutine IncreaseAgeDog(self, dogYearsIncrease)
      type (DogType)                                    :: self
      integer, intent(in)                               :: dogYearsIncrease
      ! Assume dog year is 7 human years
      self%ageInYears = self%ageInYears + dogYearsIncrease * 7
   end subroutine

end module

module CatModule
   implicit none
   save
   private
   public CatType, MakeSound, IncreaseAge

   type CatType
      integer ageInYears
   end type

   interface MakeSound
      module procedure MakeSoundPrivate
   end interface

   interface IncreaseAge
      module procedure IncreaseAgePrivate
   end interface

contains

   subroutine MakeSoundPrivate(self)
      type (CatType)                                    :: self
      print *,'Meeow!'
   end subroutine

   subroutine IncreaseAgePrivate(self, catYearsIncrease)
```

```
      type (CatType)                                    :: self
      integer, intent(in)                               :: catYearsIncrease
      ! Assume cat year is 6 human years
      self%ageInYears = self%ageInYears + catYearsIncrease * 6
   end subroutine

end module
```

Now imagine that we want to write a procedure that can be used with either of these ADTs. We could easily write a procedure that works only with `CatType` instances

```
subroutine WorkWithCat(c)
   use CatModule
   type (CatType) :: c
   call MakeSound(c)
   call IncreaseAgeInAnimalYears(c, 2)
end subroutine
```

or `DogType` instances

```
subroutine WorkWithDog(d)
   use DogModule
   type (DogType) :: d
   call MakeSound(d)
   call IncreaseAgeInAnimalYears(d, 2)
end subroutine
```

but if we write a different routine for each type, there will be a lot of duplication. What we really want is a single procedure that will work with cats and dogs, and any other animal that might be introduced to the program in future. We want something like this

```
module AnimalWorkModule
   use AnimalProtocolModule
contains
   subroutine WorkWithAnimal(a)
      type (AnimalProtocol) :: a
      call MakeSound(a)
      call IncreaseAge(a, 2)
   end subroutine
end module
```

This looks similar to the other two procedures, but replaces the *concrete types* `CatType` and `DogType` with `AnimalProtocol`.

For this to work, the AnimalProtocol ADT must know how to pass on the procedure calls to the appropriate animal. To generate the code to do this, we need a tool like Forpedo. The rest of this section will be about how you can get Forpedo to generate the `AnimalProtocol` ADT, so that the `WorkWithAnimal` procedure will work with both cats and dogs, without having any direct knowledge of either.

In Forpedo, the `protocol` directive is used to declare which procedures are required in order for an ADT to *conform* to the protocol in question. Here is the Forpedo declaration of `AnimalProtocol`:

```
#protocol AnimalProtocol AnimalProtocolModule

#method MakeSound
type(AnimalProtocol), intent(in) :: self
```

```
#endmethod

#method IncreaseAge increase
type(AnimalProtocol), intent(inout) :: self
integer, intent(in)                 :: increase
#endmethod

#conformingtype DogType DogModule
#conformingtype CatType CatModule

#endprotocol
```

The `method/funcmethod/endmethod` directives declare the interfaces of procedures that conforming ADTs must implement. In this case, the conforming types must have a `MakeSound` and `IncreaseAge` subroutine.

The arguments list for each routine is given on the line after the method name. This list should not include the first argument, which is assumed to be the instance `self`. Note that the declaration of `self` is included in the `method/funcmethod/endmethod` block, so that you can assign attributes to it (*eg.*, `intent(in)`).

The types that conform to the protocol are given explicitly in the protocol block, using the `conformingtype` directive. This directive must include the Fortran user-defined type that conforms to the protocol, and the module that declares the type.

The protocol above, having been run through Forpedo, can be used like this:

```
program Main
   use AnimalWorkModule
   use AnimalProtocolModule
   use DogModule
   use CatModule
   type (DogType), pointer   :: d
   type (CatType), pointer   :: c
   type (AnimalProtocol)     :: p

   allocate(d,c)

   ! Assign protocol to Dog, and call WorkWithAnimal
   p = d
   call WorkWithAnimal(p)

   ! Repeat for Cat. Results will be different, though subroutine call is the same.
   p = c
   call WorkWithAnimal(p)

   deallocate(d,c)

end program
```

When you run this program, it produces the following output

```
Woof!
Meeow!
```

which shows that even though each `WorkWithAnimal` call looks exactly the same, different subroutines end up getting executed for different animals. The subroutine `WorkWithAnimal` is able to indirectly call subroutines belonging to the Dog and Cat ADTs without having any direct knowledge of those types. Information about the concrete type is stored by the protocol, and the correct subroutine invoked via the `AnimalProtocol` type.

All branching required to select the correct subroutine is encapsulated in the protocol, and generated by Forpedo, making the code easier to read and extend. Adding a new conforming type to the protocol only requires a single line to be added, and changes often do not need to be made to any existing code. For example, adding a type `Tiger` to the program, and making it conform to the protocol, would not require any changes to `WorkWithAnimal`.

Exercise: Using Forpedo

Copy the source code of the program discussed in this section into files. Add the `CatModule` to cat.f90, `DogModule` to dog.f90, `AnimalWorkModule` to animalwork.f90, main program to main.f90, and `AnimalProtocol` to animalprotocol.f90t.

Now run Forpedo to generate `AnimalProtocolModule`, like this

```
forpedo.py < animalprotocol.f90t > animalprotocol.f90
```

Build and run the program:

```
gfortran -o catsanddogs dog.f90 cat.f90 animalprotocol.f90 \
   animalwork.f90 main.f90
./catsanddogs
```

Take a look in animalprotocol.f90 at the code produced by Forpedo. Try to understand how it works.

9. Final Assignment

The evaluation for this course takes the form of a programming assignment. You are expected to write some short pieces of Fortran code. The source code that you produce must be submitted and will be used for evaluation purposes.

1. (40 points) Create a Fortran module called `MatrixMultiplyModule`. Add three subroutines to it called `LoopMatrixMultiply`, `IntrinsicMatrixMultiply`, and `MixMatrixMultiply`. Each routine should take two real matrices as argument, perform a matrix multiplication, and return the result via a third argument. `LoopMatrixMultiply` should be written entirely with do loops, and no array operations or intrinsic procedures; `IntrinsicMatrixMultiply` should be written utilizing the `matmul` intrinsic function; and `MixMatrixMultiply` should be written using some do loops and the intrinsic function `dot_product`.

Write a small program to test the performance of these three different ways of performing the matrix multiplication for different sizes of matrices. The program should call each routine with matrices from 200 x 200 up to a dimensions of 1600 x 1600 and print out the efficiency of the multiplication measured in Millions of Floating Point Operations / second (MFlop/s). Here we may use the fact that a matrix multiplication of two rectangular matrices, with dimensions l and m, and m and n should take 2 x l x m x n floating point operations (m times one add plus one multiply per matrix element of the l x n result matrix).

Fill the matrices with random numbers before passing them to the routines. Use the intrinsic function system_clock to time how long each routine takes. Also compile the program with the extra option `-O3` to optimize performance, and run it. Discuss your results: which approach to multiplication is fastest in each case?

Hint: The system_clock intrinsic can be used like this to time a routine

```
integer :: count, rate
real :: timeAtStart, timeAtEnd

call system_clock(count = count, count_rate = rate)
timeAtStart = count / real(rate)  ! Time in seconds

call ProcedureToMeasure()

call system_clock(count = count, count_rate = rate)
timeAtEnd = count / real(rate)

print *, 'Time taken by ProcedureToMeasure is ', timeAtEnd - timeAtStart
```

Output: A piece of the possible output of the program could look like

```
Benchmark of matrix multiplication (measured in MFlops) :

Dim_1 Dim_2 Dim_3    Intrinsic      Mixed   Explicit
 400   200   400        4096.0     2048.0     1365.3
 800   200   400        5461.3     1365.3     1489.5
 400   400   400        5461.3     1489.5     1489.5
 800   400   400        4681.1      528.5      537.2
 400   800   400        4681.1      697.2      682.7
 800   800   400        5041.2      520.1      516.0
```

2. (30 points) Selection sort is an algorithm for sorting a one-dimensional array. It works as follows: You begin by looking for a number to use as the first element of the array. You loop through all the elements, and choose the smallest one. You swap this element with the one that is initially the first element. Then you move onto the second position. You search the whole array — except the first position — for the smallest element, and swap this with the element that is currently in the second position. This process is repeated until the whole array is sorted. Each time, you search the unsorted segment of the array, and swap it into the lowest position in that segment.

Write a module called `SortingModule`, and add a subroutine to it called `SelectionSort`. `SelectionSort` should take a single argument — the real array to be sorted. Implement the selection sort algorithm described above to sort the array passed in. Try to use intrinsic procedures in place of loops where possible. (Hint: Particularly useful intrinsics could be `minloc` and `minval`.)

Write a program to test your implementation. Make sure you test it thoroughly by including tests for arrays that are already sorted, and arrays that include multiple elements with the same value.

3. (30 points) This question consists of three parts. *You only have to choose one of the three. You DO NOT have to complete all three.*

(i) A *jagged array* is a two dimensional array in which each column has a different length. Fortran does not directly support jagged arrays, but you can write an ADT to mimic their behavior.

Write a JaggedArray ADT. It should include the following public methods: `New`, `Delete`, `AddColumn`, `GetColumn`, and `GetElement`. (`GetColumn` should return a pointer to an array of real data, and `GetElement` should take two indexes as arguments, the row index and column index, and return a single real element at that location.) Also include a copy constructor, and overload the assignment operator to make sure assignment works properly.

Read Appendix A covering Testing and Logging. Look in the CMC program to see how tests and logging work in practice. Add logging statements to the JaggedArray ADT, and write some tests using the `Testing` module from CMC. Add a module like `TestRunner`, and a program similar to the file `cmctests.f90` that runs the tests you have written.

(ii) Finite-differencing can be used to calculate the curvature matrix for a given function, for use in normal mode analysis, for example. Write an ADT called CurvatureCalculator that calculates the curvature matrix in a subroutine called `Calculate`; the function should be passed as an argument to the `Calculate` routine (see chapter on callbacks).

The ADT should include the following methods: `New`, `Delete`, `GetDelta`, and `Calculate`. (`delta` is the step used in the finite differencing; see below.) The constructor, `New`, should take the `delta` parameter as an argument.

The mathematical formula for second order finite differencing is as follows:

$$\frac{d^2 f(\mathbf{x})}{dx_i dx_j} \approx \left(\frac{f(\mathbf{x}+\Delta\mathbf{x}_i+\Delta\mathbf{x}_j) + f(\mathbf{x}-\Delta\mathbf{x}_i-\Delta\mathbf{x}_j) - f(\mathbf{x}-\Delta\mathbf{x}_i+\Delta\mathbf{x}_j) - f(\mathbf{x}+\Delta\mathbf{x}_i-\Delta\mathbf{x}_j)}{4\Delta^2} \right)$$

where

Δ is the finite differencing step, and

$\Delta\mathbf{x}_i$ is a vector with all components zero except the *i*th, which is Δ

Read Appendix A covering Testing and Logging. Look in the CMC program to see how tests and logging work in practice. Add logging statements to the CurvatureCalculator ADT, and write some tests using the `Testing` module from CMC. Add a module like `TestRunner`, and a program similar to the file `cmctests.f90` that runs the tests you have written.

(iii) The Diffusion Monte Carlo algorithm implemented in the CMC program has a shortcoming: After the reference energy stabilizes, it is averaged over many iterations to estimate the ground state energy; however, the wavefunction does not get averaged. A better approach would be to average the wavefunction over the same iterations as the reference energy gets averaged. This should reduce errors and anomalies in the wavefunction histogram printed at the end.

Implement this improvement in the CMC code. Add an input parameter which allows this feature to be turned off and on. Run the program with the wavefunction-averaging feature turned on, and compare the resulting wavefunction histogram with that printed when the feature is turned off (*ie*, plot the wavefunctions in Excel or a similar plotting program). What can you say about the 'averaged' wavefunction?

A. Appendix: Logging and Testing

A major part of programming is debugging and testing. The compiler can help you track down certain types of programming errors, such as syntactical ones, but it is wrong to assume that a program is 'correct' simply because it compiles. You need to test your programs, and when something doesn't work as it should, you need a means of debugging them.

Logging is a practice widely used in large programs. In its simplest form, it involves printing out information which can be used to track down problems. More advanced logging typically includes different levels of detail: you can print out only the highest level messages, or you can print out a lot of detail.

CMC includes a basic logging ADT: Logger. You can look in any number of CMC files to see how it works (*eg*, EnsembleWalker.f90), and — of course — the Logger.f90 file itself.

Basically, you create one Logger instance per module. You declare and initialize it like this:

```
module MyADTModule

   ...

   ! Logging
   type (Logger)              :: log
   integer(KINT), parameter   :: LOGGING_LEVEL = WARNING_LOGGING_LEVEL

contains

   subroutine InitLogger()
      ! This routine is called from the constructor to initialize the Logger.
      logical, save :: loggerInitialized = .false.
      if ( .not. loggerInitialized ) then
         call New(log, LOGGING_LEVEL)
         loggerInitialized = .true.
      end if
   end subroutine

   subroutine NewPrivate()
      ...
      call InitLogger()
   end subroutine

   ...

end module
```

You then add logging calls wherever they might be useful. One place they are useful is when you enter or exit an important routine:

```
   subroutine PropagateEnsemble(self)
      ...   ! Declarations
      call LogMessage(log, TRACE_LOGGING_LEVEL, 'Entered PropagateEnsemble')

      ...

      call LogMessage(log, TRACE_LOGGING_LEVEL, 'Exiting PropagateEnsemble')

   end subroutine
```

Each logging message has a level; the `TRACE_LOGGING_LEVEL` is used for just tracing the path of execution through the program.

The `DEBUG_LOGGING_LEVEL` is for when you want to print out some information useful during debugging. Usually this will be a variable or small array.

```
call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Progation Cycle', i)
call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Number of Particles', numParticles)
call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Reference Energy', &
   self%currentReferenceEnergy)
```

Each message has a label, passed as a string, and a variable to be printed. The variable can be a real or integer value, or a one-dimensional array.

The other levels, `WARNING_LOGGING_LEVEL` and `ERROR_LOGGING_LEVEL`, are for more serious problems that should be reported to the developer.

Generally, the level of a logger is set to something like `WARNING_LOGGING_LEVEL`. This means that only `WARNING_LOGGING_LEVEL` or `ERROR_LOGGING_LEVEL` messages will be printed — others will be skipped. If you are debugging a particular module, you will probably want to set its logger level to `DEBUG_LOGGING_LEVEL`, so that all messages are printed.

Where logging is designed to help you follow what is happening in a program, testing helps you to determine if it is giving the results you expect. Writing *unit tests*, which are low-level tests that probe basic functionality, can be very useful when you are developing new code, and making sure you don't break existing code. If you write tests, and run them regularly, it gives you more confidence to make changes.

In CMC, testing functionality is provided by the `Testing` and `TestRunner` modules. You can run the tests by compiling as usual, but providing `cmctests.f90` as the main program, rather than `cmc.f90`.

You usually write one test module for each module in the program. For example, `HistogramBuilder` has a test module called `HistogramBuilderTests`. Each test module includes a number of tests, and each one makes calls to a routine like `Assert` or `AssertAlmostEqual`, which tests if a particular assertion is true. If not, the test fails.

```
module HistogramBuilderTests
   use NumberKinds
   use HistogramBuilderModule
   use Testing
   implicit none

   private
   public RunHistogramBuilderTests
   save

contains

   subroutine RunHistogramBuilderTests
      call TestBuilder
   end subroutine

   subroutine TestBuilder
      type (HistogramBuilder) :: histoBuilder
      ...

      call AssertAlmostEqual(GetLowerBound(histoBuilder), low, eps, &
         'lowerBound was not correctly initialized')
      call AssertAlmostEqual(GetUpperBound(histoBuilder), up, eps, &
         'upperBound was not correctly initialized')
      call Assert( GetNumberOfBins(histoBuilder) == NUM_BINS, &
         'numberOfBins was not correctly initialized')

      ...

      call Assert( histo(1) == 0, 'Histogram value incorrect' )
      call Assert( histo(2) == 0, 'Histogram value incorrect' )

      ...

   end subroutine

end module
```

The `Assert` subroutine tests whether a logical value is true, and `AssertAlmostEqual` compares two real numbers to see if they lie within a certain tolerance, which is passed as the third argument. If the assertion in either subroutine is false, the string passed as the last argument is printed, stating which assertion failed.

One routine, `RunHistogramBuilderTests` in the example above, is included to call each test procedure in turn. A call to this routine is added to the `TestRunner` module.

```
   subroutine Run(self)
      type (TestRunner), intent(inout)                  :: self

      print *,'Running Tests'
      call RunParticleEnsembleTests()
      call RunPotentialTests()
      call RunHistogramBuilderTests()
      call RunGaussianDistributorTests()
      call RunInputReaderTests()
      print *

   end subroutine
```

It's good practice to run the unit tests regularly, and add new tests as you are writing new code, rather than after the fact.

B. Appendix: Diffusion Monte Carlo Algorithm

Diffusion Monte Carlo is an algorithm for finding the ground state of a quantum system. As implemented in the CMC program, it will find the ground state of a one dimensional system.

The algorithm is particle based. Initially an ensemble of particles is positioned at a single point at which the ground state wavefunction is assumed to be non-zero. The particles are then propagated in imaginary time using a random step (Monte Carlo) approach. The range of possible steps depends on the time step and particle mass, as you might expect. After each step, a particle may duplicate itself, or be annihilated. The probability of these events depends on the potential energy at the particle's current position, and a reference energy, which is varied during the propagation until it stablilizes. Once stable, the reference energy gives the ground state energy, and the distribution of particles matches the distribution of the wavefunction.

There are two phases to each propagation step of the particle ensemble: the first causes each particle to take a random step, given by the following formula (Eq. 2.30 in Kosztin *et al*)

$$x_i(t+\Delta t) = x_i(t) + \sqrt{\hbar \Delta t / m}\,.\rho$$

where

x_i is the position of particle i

t is imaginary time

Δt is a propagation time step

$\hbar$ is Plancks contant divided by 2π

m is the particle mass

ρ is a random number

The random number, ρ, must be chosen from a gaussian distribution with a variance of 1. Also, in CMC all quantities will be in atomic units, for which $\hbar$ is equal to 1.

The second phase of each propagation step is to duplicate or annihilate each particle. The number of particles formed for any given particle is given by the formula (Eq. 2.28 in Kosztin *et al*)

$$n_i = \min\{\text{int}[W(x_i) + u], 3\}$$

where

n_i is the number of particles created

$W(x_i)$ is a weight function

u is a random number between 0 and 1

int[] means to take the integer part of a number

The number of particles produced can vary from 0 to 3; if 0, the particle in question is effectively annihilated.

The weight function is given by (Eq. 2.16 in Kosztin *et al*)

$$W(x_i) = \exp\left(-\frac{[V(x_i) - E_R]\Delta t}{\hbar}\right)$$

where

$V(x_i)$ is the potential energy of particle i

E_R is the reference energy

The propagation involves these two phases each time step. The reference energy, which is initially set to a guess, is adjusted each step in an attempt to cause the total number of particles to stabilize. When the number of particles is approximately constant, and the reference energy no longer changes, the calculation is assumed to be converged to the ground state.

The formula used to adjust the reference energy each time step is as follows (similar to Eq. 2.36 in Kosztin *et al*)

$$E_R(t + \Delta t) = E_R(t) + D\frac{\hbar}{\Delta t}\left(1 - \frac{N(t)}{N(t - \Delta t)}\right)$$

where

D is a damping factor

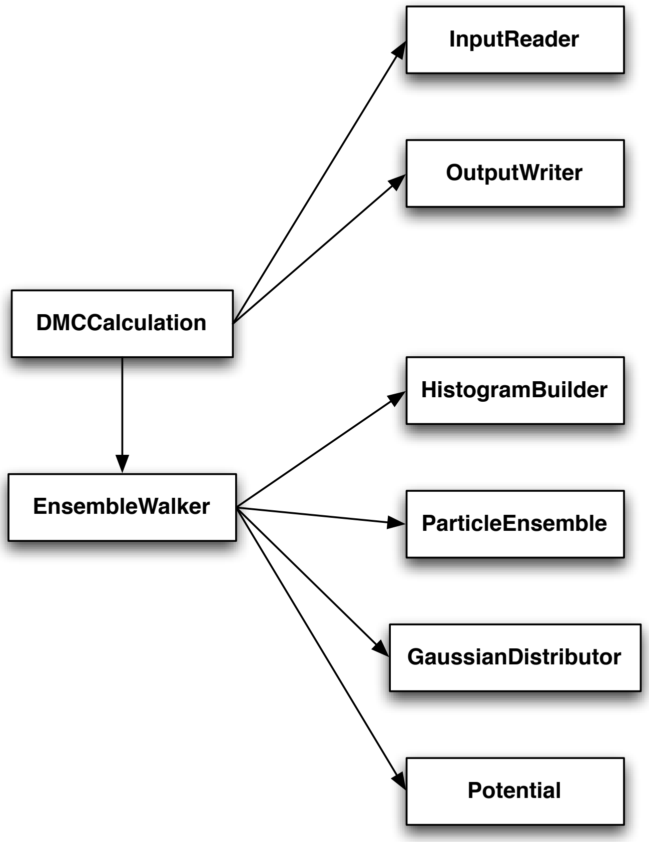
$N(t)$ is the total number of particles at time t

The damping factor is set to 0.1 in the CMC program. This formula will cause the reference energy to rise if the number of particles diminished in the last time step, and decrease if the number of particles rose. If the number of particles remains constant, the reference energy will also be unchanged.

In practice, the reference energy never completely stabilizes. Instead, the whole calculation is broken into two stages: First, the program propagates the system a fixed number of time steps to bypass the initial transitory behavior. Then, in the second stage, the system is propagated further, and the reference energy averaged over time steps until the average changes by less than a given tolerance. At this point the calculation is considered converged, and the time-averaged reference energy is considered to be the ground state energy.

C. Appendix: Design of 'Confusion Monte Carlo'

This appendix describes the various ADTs that make up the CMC program. Hopefully, you will recognize the correspondence between the CMC design and the algorithm described in Appendix B.



The main ADTs of the Confusion Monte Carlo program, and the relationships between them.

Abstract Data Type	DMCCalculation
Description	This is the top level ADT in the program. It is responsible for reading input, performing the calculation, and writing results.
Details	An instance of this ADT is created in the main program, and then the `Run` method is called. The `DMCCalculation` first opens an input file, and reads the data in using an `InputReader`. The data is used to initialize an `EnsembleWalker`, and the system is propagated. When the propagation is finished, methods are called to cause the results to be output.

Abstract Data Type	EnsembleWalker
Description	The `EnsembleWalker` is responsible for propagating an ensemble of particles in imaginary time, until they converge to the ground state wavefunction.
Details	The `EnsembleWalker` creates and repeatedly propagates a `ParticleEnsemble` in time until the reference energy is converged, when the ground state is assumed to have been reached. The propagation occurs in the subroutine `PropagateEnsemble`; this routine loops through time steps, and also checks for convergence. Each time step, `PropagateEnsemble` calls the `StepParticles` method to change the particle positions, and `SpawnParticles` to create and annihilate particles. `EnsembleWalker` also includes subroutines that output the calculation properties and results. The final wavefunction distribution is determined with a `HistogramBuilder`.

Abstract Data Type	ParticleEnsemble
Description	`ParticleEnsemble` is for storing the mass and positions of all of the particles in the system.
Details	This ADT provides methods for adding and retrieving particles in the ensemble. `AddParticle` is used to add new particles, and `GetParticleAtIndex` can be used to retrieve an existing particle. The array used to store the particles is grown as needed by allocating new memory, copying existing particles, and deallocating the old memory.

Abstract Data Type	GaussianDistributor
Description	`GaussianDistributor` is a random number generator that generates numbers according to a Gaussian distribution.
Details	The method `GenerateRandomValue` implements an algorithm to generate the random numbers as follows: First a random coordinate is chosen from a coordinate range that extends over the non-negligible part of the gaussian distribution. (The non-negligible range is chosen to extend a fixed number of standard deviations from the center.) A second random number is then created, and the coordinate is accepted if this second random number is less than the value of the gaussian function at the chosen coordinate. If the coordinate is rejected, a new random coordinate is chosen, and the process repeated until the test succeeds.

Abstract Data Type	HistogramBuilder
Description	`HistrogramBuilder` takes an array of values, and bins them to form a histogram.
Details	This ADT is fairly independent from the rest of the program. It is simply used to take the particle coordinates resulting from the DMC calculation, and create a wavefunction distribution from them. The `CreateHistogram` method makes the histogram. Various parameters passed to the constructor determine the bin widths and range of coordinates covered.

Abstract Data Type	Potential
Description	`Potential` is an ADT that represents the potential energy function of the system.
Details	The `Potential` ADT can implement several different types of one dimensional potentials. The type of potential used is passed to the constructor. Potential types are defined as integer parameters, and potential parameters are simply hard-coded for simplicity. The ADT implements an harmonic oscillator, a morse potential, and a Lennard-Jones potential. The `ValueAtCoord` function is used to evaluate the potential at a given point.

Abstract Data Type	InputReader
Description	`InputReader` reads the program input, which is in a key-value pair form.
Details	The `InputReader` is responsible for reading the input file — a process known as *parsing* — and extracting input parameters. Only real and integer parameters can be parsed. The implementation reads the file each time a request is made, via the `Read` method, and extracts the relevant parameter.

Abstract Data Type	OutputWriter
Description	`OutputWriter` formats and writes program output.
Details	The `OutputWriter` ADT can write various types of data, from simple integers and reals, to arrays of reals. The output is in key-value pairs; the key is a label for the output, and is written as a title. It also supports sections: by calling `BeginSection`, indentation of output will be increased; `EndSection` causes it to decrease again.

Abstract Data Type	Logger
Description	`Logger` is used by the developer to log messages to standard output, which can be helpful for debugging purposes.
Details	The `Logger` ADT allows messages and data to be written to standard output, for the purposes of debugging. A `Logger` has a number of possible levels; by setting the level of a `Logger`, the developer can control how much output is written. The developer inserts logging calls into the code in order to print useful information. An argument is passed for each call indicating the level of the message. If the message level is greater than or equal to the level of the `Logger`, the message is printed.

Abstract Data Type	TestRunner
Description	`TestRunner` is used to run unit tests.
Details	The `TestRunner` ADT is responsible for running all the unit tests in the program. If a unit test is written, it should be added to the `Run` method.

D. Appendix: Source Code of 'Confusion Monte Carlo'

D.1. cmc.f90

```
program CMC
   use DMCCalculationModule
   implicit none
   type (DMCCalculation)                                :: dmcCalc
   call New(dmcCalc)
   call Run(dmcCalc)
   call Delete(dmcCalc)
end program
```

D.2. DMCCalculation.f90

```
module DMCCalculationModule
   use NumberKinds
   use OutputWriterModule
   use InputReaderModule
   use LoggerModule
   use EnsembleWalkerModule
   use PotentialModule
   implicit none

   private
   public DMCCalculation
   public New, Delete, Run
   save

   type DMCCalculation
      private
      type (EnsembleWalkerProperties), pointer          :: walkerProperties
      type (Potential), pointer                         :: potentialFunction
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

contains

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction.
   ! ------------------------
   subroutine NewPrivate(self)
      type (DMCCalculation), intent(out)                :: self

      type (InputReader)                                :: reader
      type (EnsembleWalkerProperties), pointer          :: p
      logical                                           :: didRead
      integer(KINT)                                     :: inputFileUnit, potForm

      ! Initialize ensemble walker properties with defaults
      allocate(self%walkerProperties)
      p => self%walkerProperties
      p%timeStep = 2.0_KREAL
      p%initialReferenceEnergy = 0.0_KREAL
      p%energyConvergenceTolerance = 1.e-8_KREAL
      p%particleMass = 1800.0_KREAL
      p%initialPositionOfParticles = 0.0_KREAL
      p%initialNumberOfParticles = 10000
      p%maxPropagationSteps = 500
      p%numberOfTransientSteps = 150
      p%numberOfEnergyBins = 100

      ! Override defaults from input
      inputFileUnit = 70
      open(unit=inputFileUnit, file='cmcinput.txt', action='read', status='old')
      call New(reader, inputFileUnit)
      didRead = Read(reader, 'timestep', p%timestep)
      didRead = Read(reader, 'referenceenergy', p%initialReferenceEnergy)
      didRead = Read(reader, 'energyconvergence', p%energyConvergenceTolerance)
      didRead = Read(reader, 'particlemass', p%particleMass)
      didRead = Read(reader, 'particleposition', p%initialPositionOfParticles)
      didRead = Read(reader, 'numberofparticles', p%initialNumberOfParticles)
      didRead = Read(reader, 'maximumsteps', p%maxPropagationSteps)
      didRead = Read(reader, 'minimumsteps', p%numberOfTransientSteps)
      didRead = Read(reader, 'numenergybins', p%numberOfEnergyBins)

      ! Initialize potential
      allocate(self%potentialFunction)
      potForm = HARMONIC_POTENTIAL
      didRead = Read(reader, 'potential', potForm)
      print *,potForm
      call New(self%potentialFunction, potForm)

      call Delete(reader)
      close(inputFileUnit)

   end subroutine

   subroutine DeletePrivate(self)
      type (DMCCalculation), intent(inout)             :: self

      call Delete(self%potentialFunction)
      deallocate(self%walkerProperties)
      deallocate(self%potentialFunction)

   end subroutine

   ! -----------------------
   ! Other methods.
   ! -----------------------
   subroutine Run(self)
      type (DMCCalculation), intent(inout)             :: self

      type (EnsembleWalker)                            :: walker
      type (OutputWriter)                              :: writer

      call New(writer, 6)    ! Write to standard output

      call New(walker, self%potentialFunction, self%walkerProperties)
      call OutputProperties(walker, writer)
      call PropagateEnsemble(walker)
      call OutputResults(walker, writer)
      call Delete(walker)
```

```
      call Delete(writer)

   end subroutine

end module
```

D.3. EnsembleWalker.f90

```
module EnsembleWalkerModule
   use NumberKinds
   use OutputWriterModule
   use LoggerModule
   use PotentialModule
   use ParticleEnsembleModule
   implicit none

   private
   public EnsembleWalker, EnsembleWalkerProperties
   public New, Delete, OutputProperties, OutputResults
   public GetProperties, GetPotential, GetParticleEnsemble
   public GetMeanPotential, PropagateEnsemble
   save

   type EnsembleWalkerProperties
      real(KREAL)                                       :: timeStep
      real(KREAL)                                       :: initialReferenceEnergy
      real(KREAL)                                       :: energyConvergenceTolerance
      real(KREAL)                                       :: particleMass
      real(KREAL)                                       :: initialPositionOfParticles
      integer(KINT)                                     :: initialNumberOfParticles
      integer(KINT)                                     :: maxPropagationSteps
      integer(KINT)                                     :: numberOfTransientSteps
      integer(KINT)                                     :: numberOfEnergyBins
   end type

   type EnsembleWalker
      private
      type (EnsembleWalkerProperties)                   :: properties
      type (Potential), pointer                         :: potentialFunction
      type (ParticleEnsemble), pointer                  :: ensemble
      real(KREAL)                                       :: summedReferenceEnergy
      real(KREAL)                                       :: currentReferenceEnergy
      logical                                           :: isConverged
      integer(KINT)                                     :: numStepsTaken
      integer(KINT)                                     :: numStepsForAverage
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

   interface OutputProperties
      module procedure OutputPropertiesPrivate
   end interface
   ! Logging
   type (Logger)            :: log
   integer(KINT), parameter :: LOGGING_LEVEL = WARNING_LOGGING_LEVEL

contains

   ! ------------------------
   ! Logging.
   ! ------------------------
   subroutine InitLogger()
      ! This routine is called from the constructor to initialize the Logger.
      logical, save  :: loggerInitialized = .false.
      if ( .not. loggerInitialized ) then
         call New(log, LOGGING_LEVEL)
         loggerInitialized = .true.
      end if
   end subroutine

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction, Copying, and Assignment.
   ! ------------------------
   subroutine NewPrivate(self, potentialFunction, properties)
      type (EnsembleWalker), intent(out)               :: self
      type (Potential), pointer                        :: potentialFunction
      type (EnsembleWalkerProperties), intent(in)      :: properties

      type (Particle)                                  :: p
      integer(KINT)                                    :: i

      self%potentialFunction => potentialFunction
      self%properties = properties
      self%currentReferenceEnergy = properties%initialReferenceEnergy
      self%isConverged = .false.
      self%numStepsTaken = 0

      ! Create initial ensemble
      allocate(self%ensemble)
      call New(self%ensemble, properties%particleMass)
      p%position = properties%initialPositionOfParticles
      do i = 1, properties%initialNumberOfParticles
         call AddParticle(self%ensemble, p)
      enddo

      call InitLogger

   end subroutine

   subroutine DeletePrivate(self)
      type (EnsembleWalker), intent(inout)             :: self
      call Delete(self%ensemble)
      deallocate(self%ensemble)
   end subroutine

   ! ------------------------
   ! Accessors.
   ! ------------------------
   function GetProperties(self)
      type (EnsembleWalker), intent(in)                :: self
      type (EnsembleWalkerProperties)                  :: GetProperties
      GetProperties = self%properties
```

```
end function

function GetPotential(self)
   type (EnsembleWalker), intent(in)                  :: self
   type (Potential), pointer                          :: GetPotential
   GetPotential => self%potentialFunction
end function

function GetParticleEnsemble(self)
   type (EnsembleWalker), intent(in)                  :: self
   type (ParticleEnsemble), pointer                   :: GetParticleEnsemble
   GetParticleEnsemble => self%ensemble
end function

function GetMeanPotential(self)
   ! Calculates the mean potential on the fly
   type (EnsembleWalker), intent(in)               :: self
   real(KREAL)                                     :: GetMeanPotential

   type (Particle)                                 :: part
   integer(KINT)                                   :: i, n
   real(KREAL)                                     :: pot, potSum

   n = GetNumberOfParticles(self%ensemble)
   if ( n == 0 ) stop 'There were no particles in the ensemble in ' // &
      'GetMeanPotential'

   potSum = 0.0_KREAL
   do i = 1, n
      part = GetParticleAtIndex(self%ensemble, i)
      pot = ValueAtCoord(self%potentialFunction, part%position)
      potSum = potSum + pot
   enddo

   GetMeanPotential =  potSum / n
end function

! -----------------------
! Other methods.
! -----------------------
subroutine PropagateEnsemble(self)
   type (EnsembleWalker), intent(inout)               :: self

   type (ParticleEnsemble), pointer                   :: newEnsemble
   integer(KINT)                                      :: i, numParticlesLast
   integer(KINT)                                      :: numParticles
   real(KREAL)                                        :: lastEnergy, factor, tol
   real(KREAL)                                        :: energy
   logical                                            :: converged
   real(KREAL), parameter                             :: ENERGY_DAMPING = 0.1_KREAL

   call LogMessage(log, TRACE_LOGGING_LEVEL, 'Entered PropagateEnsemble')

   self%summedReferenceEnergy = 0.0_KREAL
   self%numStepsForAverage = 0
   self%isConverged = .false.
   do i = 1, self%properties%maxPropagationSteps
      ! Step each particle, and duplicate or destroy it
      numParticlesLast = GetNumberOfParticles(self%ensemble)
      call StepParticles(self, self%ensemble)
      allocate(newEnsemble)
      call New(newEnsemble, GetParticleMass(self%ensemble))
      call SpawnParticles(self, self%ensemble, newEnsemble)
      call Delete(self%ensemble)
      deallocate(self%ensemble)
      self%ensemble => newEnsemble

      ! Update steps taken
      self%numStepsTaken = self%numStepsTaken + 1

      ! Update reference energy
      numParticles = GetNumberOfParticles(self%ensemble)
      factor = 1.0_KREAL - real(numParticles) / numParticlesLast
      self%currentReferenceEnergy = self%currentReferenceEnergy + &
         ENERGY_DAMPING * factor / self%properties%timeStep

      ! Update reference energy summation
      if ( i >= self%properties%numberOfTransientSteps ) then
         lastEnergy = self%summedReferenceEnergy / max(1, self%numStepsForAverage)
         self%numStepsForAverage = self%numStepsForAverage + 1
         self%summedReferenceEnergy = self%summedReferenceEnergy + &
            self%currentReferenceEnergy
      endif

      ! Log progress
      call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Progation Cycle', i)
      call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Number of Particles', &
         numParticles)
      call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Reference Energy', &
         self%currentReferenceEnergy)

      ! Check convergence
      if ( i > self%properties%numberOfTransientSteps ) then
         tol = self%properties%energyConvergenceTolerance
         energy = self%summedReferenceEnergy / self%numStepsForAverage
         self%isConverged = (abs(energy - lastEnergy) <= tol)
         call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Averaged Ref. Energy', &
            self%summedReferenceEnergy / self%numStepsForAverage)
      endif

      if ( self%isConverged ) exit
   enddo

   call LogMessage(log, TRACE_LOGGING_LEVEL, 'Exiting PropagateEnsemble')

end subroutine

subroutine StepParticles(self, particles)

   ! Propagate each particle in the ensemble one step in time.

   use GaussianDistributorModule
   type (EnsembleWalker), intent(inout)             :: self
   type (ParticleEnsemble), intent(inout)           :: particles

   type (GaussianDistributor)                       :: distributor
   type (Particle)                                  :: p
   real(KREAL)                                      :: step, mass, factor
   integer(KINT)                                    :: i, n
   call New(distributor, 1.0_KREAL)
   mass = GetParticleMass(particles)
   factor = sqrt(self%properties%timeStep / mass)
   n = GetNumberOfParticles(particles)
   do i = 1, n
      p = GetParticleAtIndex(particles, i)
      step = GenerateRandomValue(distributor)
      step = step * factor
      p%position = p%position + step
      call SetParticleAtIndex(particles, i, p)
   enddo
   call Delete(distributor)

end subroutine

subroutine SpawnParticles(self, originalEnsemble, resultEnsemble)

   ! Duplicates particles from originalEnsemble into resultEnsemble. Particles
   ! may spawn more than one new particle, or disappear altogether. Both
   ! ensembles should already be constructed, but resultEnsemble will usually
   ! be empty initially.

   type (EnsembleWalker), intent(inout)             :: self
   type (ParticleEnsemble), intent(in)              :: originalEnsemble
   type (ParticleEnsemble), intent(inout)           :: resultEnsemble

   type (Particle)                                  :: p
   integer(KINT)                                    :: n, i, j, numSpawned
   real(KREAL)                                      :: weight, potValue, rand

   n = GetNumberOfParticles(originalEnsemble)
   do i = 1, n
      p = GetParticleAtIndex(originalEnsemble, i)
      potValue = ValueAtCoord(self%potentialFunction, p%position)
      weight = exp( (self%currentReferenceEnergy - potValue) * &
         self%properties%timeStep )
      call random_number(rand)
      numSpawned = min( floor(weight + rand), 3 )
      do j = 1, numSpawned
         call AddParticle(resultEnsemble, p)
      enddo
   enddo

end subroutine

! ------------------------
! Output properties.
! ------------------------
subroutine OutputPropertiesPrivate(self, writer)
   type (EnsembleWalker), intent(in), target        :: self
   type (OutputWriter), intent(inout)               :: writer
   type (EnsembleWalkerProperties), pointer         :: pr

   pr => self%properties
   call Write(writer, 'Time Step', pr%timeStep)
   call Write(writer, 'Initial Reference Energy', pr%initialReferenceEnergy)
   call Write(writer, 'Energy Convergence', pr%energyConvergenceTolerance)
   call Write(writer, 'Initial Position of Particles', &
      pr%initialPositionOfParticles)
   call Write(writer, 'Initial Number of Particles', pr%initialNumberOfParticles)
```

```
      call StartSection(writer, 'Potential Function Properties', &
         'The potential function used in the calculation.')
      call OutputProperties(self%potentialFunction, writer)
      call EndSection(writer)

   end subroutine

   subroutine OutputResults(self, writer)
      use HistogramBuilderModule
      type (EnsembleWalker), intent(in), target         :: self
      type (OutputWriter), intent(inout)                :: writer

      type (HistogramBuilder)                           :: histoBuilder
      type (Particle)                                   :: part
      real(KREAL)                                       :: lowerBound, upperBound
      real(KREAL), allocatable                          :: positions(:), wavefunc(:)
      integer(KINT)                                     :: numberOfBins, i, n

      if ( self%isConverged ) then
         call Write(writer, 'Calculation Convergence', 'Succeeded')
      else
         call Write(writer, 'Calculation Convergence', 'Failed')
      endif

      call Write(writer, 'Propagation Steps', self%numStepsTaken)
      call Write(writer, 'Energy', self%summedReferenceEnergy / &
         self%numStepsForAverage)
      call Write(writer, 'Number of Particles', GetNumberOfParticles(self%ensemble))

      ! Create histogram for the particle wavefunction. Print out the wavefunction.
      n = GetNumberOfParticles(self%ensemble)
      allocate(positions(n))
      do i = 1, n
         part = GetParticleAtIndex(self%ensemble, i)
         positions(i) = part%position
      enddo
      lowerBound = minval(positions)
      upperBound = maxval(positions)
      call New(histoBuilder, lowerBound, upperBound, &
         self%properties%numberOfEnergyBins)
      allocate( wavefunc(self%properties%numberOfEnergyBins) )
      wavefunc = CreateHistogram(histoBuilder, positions)
      wavefunc = wavefunc / max(1.0_KREAL, sum(wavefunc))
      wavefunc = wavefunc / GetBinWidth(histoBuilder)
      wavefunc = sqrt(wavefunc)
      call Write(writer, 'Wavefunction', GetBinCenters(histoBuilder), wavefunc )
      call Delete(histoBuilder)
      deallocate(positions, wavefunc)

   end subroutine

end module
```

D.4. GaussianDistributor.f90

```
module GaussianDistributorModule
   use NumberKinds
   use OutputWriterModule
   use LoggerModule
   implicit none
   private
   public GaussianDistributor
   public New, Delete, assignment(=), OutputProperties
   public GetStandardDeviation, GenerateRandomValue
   save

   type GaussianDistributor
      private
      real(KREAL)  :: standardDeviation
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate, NewCopyPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

   interface assignment(=)
      module procedure AssignPrivate
   end interface

   interface OutputProperties
      module procedure OutputPropertiesPrivate
   end interface

   ! Logging
   type (Logger)            :: log
   integer(KINT), parameter :: LOGGING_LEVEL = WARNING_LOGGING_LEVEL

contains

   ! ------------------------
   ! Logging.
   ! ------------------------
   subroutine InitLogger()
      ! This routine is called from the constructor to initialize the Logger.
      logical, save :: loggerInitialized = .false.
      if ( .not. loggerInitialized ) then
         call New(log, LOGGING_LEVEL)
         loggerInitialized = .true.
      end if
   end subroutine

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction, Copying, and Assignment.
   ! ------------------------
   subroutine NewPrivate(self, standardDeviation)
      type (GaussianDistributor), intent(out)           :: self
      real(KREAL), intent(in)                           :: standardDeviation
      self%standardDeviation = standardDeviation
      call InitLogger
   end subroutine

   subroutine NewCopyPrivate(self, other)
      type (GaussianDistributor), intent(out)           :: self
      type (GaussianDistributor), intent(in)            :: other
      self%standardDeviation = other%standardDeviation
   end subroutine

   subroutine DeletePrivate(self)
      type (GaussianDistributor), intent(inout)         :: self
   end subroutine

   subroutine AssignPrivate(self, other)
      type (GaussianDistributor), intent(inout)         :: self
      type (GaussianDistributor), intent(in)            :: other
      self%standardDeviation = other%standardDeviation
   end subroutine

   ! -----------------------
   ! Accessors.
   ! -----------------------
   subroutine SetStandardDeviation(self, standardDeviation)
      type (GaussianDistributor), intent(inout)         :: self
      real(KREAL), intent(in)                           :: standardDeviation
      self%standardDeviation = standardDeviation
   end subroutine

   function GetStandardDeviation(self)
      type (GaussianDistributor), intent(in)            :: self
      real(KREAL)                                       :: GetStandardDeviation
      GetStandardDeviation = self%standardDeviation
   end function

   ! -----------------------
   ! Other methods.
   ! -----------------------
   function GenerateRandomValue(self) result (coord)
      type (GaussianDistributor), intent(inout)         :: self
      real(KREAL)                                       :: coord,random
      real(KREAL)                                       :: gaussVal
      call LogMessage(log, TRACE_LOGGING_LEVEL, 'Entered GenerateRandomValue')

      random = 1.0_KREAL
      gaussVal = 0.0_KREAL
      do while ( random > gaussVal )
         ! Choose random value of coord in a 3 standard deviation range from zero
         call random_number(coord)
         coord = 3.0 * self%standardDeviation * (2.0 * coord - 1.0)

         ! Calculate value of gaussian function (normal curve)
         gaussVal = exp(-1.0_KREAL / (2.0 * self%standardDeviation**2) * coord**2)

         ! Generate a second random number, and test it against the gaussian value
         call random_number(random)
      enddo

      call LogMessage(log, DEBUG_LOGGING_LEVEL, 'The random number is', coord)
      call LogMessage(log, TRACE_LOGGING_LEVEL, 'Exiting GenerateRandomValue')
   end function

   ! -----------------------
   ! Output properties.
   ! -----------------------
   subroutine OutputPropertiesPrivate(self, writer)
      type (GaussianDistributor), intent(in)            :: self
```

```
      type (OutputWriter), intent(inout)                :: writer

      ! Write details of ADT instance to the writer
      call Write(writer, 'Standard Deviation', self%standardDeviation)

   end subroutine

end module
```

D.5. HistogramBuilder.f90

```
module HistogramBuilderModule
   use NumberKinds
   use OutputWriterModule
   use LoggerModule
   implicit none

   private
   public HistogramBuilder, New, Delete, Assignment(=), OutputProperties
   public GetLowerBound, GetUpperBound, GetNumberOfBins, CreateHistogram
   public GetBinCenters, GetBinWidth
   save

   type HistogramBuilder
      private
      real (KREAL)         :: lowerBound,upperBound,delta
      integer (KINT)       :: numberOfBins
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate, NewCopyPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

   interface OutputProperties
      module procedure OutputPropertiesPrivate
   end interface

   ! Logging
   type (Logger)             :: log
   integer(KINT), parameter  :: LOGGING_LEVEL = WARNING_LOGGING_LEVEL

contains

   ! ------------------------
   ! Logging.
   ! ------------------------
   subroutine InitLogger()
      ! This routine is called from the constructor to initialize the Logger.
      logical, save :: loggerInitialized = .false.
      if ( .not. loggerInitialized ) then
         call New(log, LOGGING_LEVEL)
         loggerInitialized = .true.
      end if
   end subroutine

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction, Copying, and Assignment.
   ! ------------------------
   subroutine NewPrivate(self, lowerBound, upperBound, numberOfBins)
      type (HistogramBuilder), intent(out)              :: self
      real (KREAL), intent(in)                          :: lowerBound, upperBound
      integer (KINT), intent(in)                        :: numberOfBins
      if ( numberOfBins <= 0 ) stop 'Too few bins in HistogramBuilder'
      self%lowerBound = lowerBound
      self%upperBound = upperBound
      self%numberOfBins = numberOfBins
      self%delta = (upperBound - lowerBound) / numberOfBins
      call InitLogger
   end subroutine

   subroutine NewCopyPrivate(self, other)
      type (HistogramBuilder), intent(out)              :: self
      type (HistogramBuilder), intent(in)               :: other
      self = other
   end subroutine

   subroutine DeletePrivate(self)
      type (HistogramBuilder), intent(inout)            :: self
   end subroutine

   ! ------------------------
   ! Accessors.
   ! ------------------------
   function GetLowerBound(self)
      type (HistogramBuilder), intent(in)               :: self
      real(KREAL)                                       :: GetLowerBound
      GetLowerBound = self%lowerBound
   end function

   function GetUpperBound(self)
      type (HistogramBuilder), intent(in)               :: self
      real(KREAL)                                       :: GetUpperBound
      GetUpperBound = self%upperBound
   end function

   function GetNumberOfBins(self)
      type (HistogramBuilder), intent(in)               :: self
      integer(KINT)                                     :: GetNumberOfBins
      GetNumberOfBins = self%numberOfBins
   end function

   function GetBinCenters(self) result(centers)
      type (HistogramBuilder), intent(in)               :: self
      real(KREAL)                                       :: centers(self%numberOfBins)
      integer(KINT)                                     :: i
      do i = 1, size(centers)
         centers(i) = self%lowerBound + GetBinWidth(self) * (i - 0.5_KREAL)
      enddo
   end function

   function GetBinWidth(self)
      type (HistogramBuilder), intent(in)               :: self
      real(KREAL)                                       :: GetBinWidth
      GetBinWidth = (self%upperBound - self%lowerBound) / self%numberOfBins
   end function
```

```
   ! ------------------------
   ! Other methods.
   ! ------------------------
   function CreateHistogram(self, values) result(histo)
      type (HistogramBuilder), intent(inout)            :: self
      real(KREAL), intent(in)                           :: values(:)
      integer(KINT)                                     :: i,dummy
      integer(KINT)                                     :: histo(self%numberOfBins)
      call LogMessage(log, TRACE_LOGGING_LEVEL, 'Entered CreateHistogram')

      histo = 0
      do i = 1, size(values)
         dummy = ceiling((values(i) - self%lowerBound) / self%delta)
         if ((dummy > 0) .and. (dummy <= self%numberOfBins)) then
            histo(dummy) = histo(dummy) + 1
         endif
      enddo

      call LogMessage(log, DEBUG_LOGGING_LEVEL, 'The array of values', histo)
      call LogMessage(log, TRACE_LOGGING_LEVEL, 'Exiting CreateHistogram')
   end function

   ! ------------------------
   ! Output properties.
   ! ------------------------
   subroutine OutputPropertiesPrivate(self, writer)
      type (HistogramBuilder), intent(in)               :: self
      type (OutputWriter), intent(inout)                :: writer

      ! Write details of ADT instance to the writer
      call Write(writer, 'Lower Bound', self%lowerBound)
      call Write(writer, 'Upper Bound', self%upperBound)
      call Write(writer, 'Number of Bins', self%numberOfBins)

   end subroutine

end module
```

D.6. InputReader.f90

```
module InputReaderModule
   use NumberKinds
   use LoggerModule
   implicit none

   private
   public InputReader
   public New, Delete, assignment(=)
   public Read
   save

   type InputReader
      private
      integer(KINT) :: fileUnit
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

   interface Read
      module procedure ReadInteger, ReadReal
   end interface

   ! Defines
   integer(KINT), parameter :: MAX_LINE_LENGTH = 256

   ! Logging
   type (Logger)            :: log
   integer(KINT), parameter :: LOGGING_LEVEL = WARNING_LOGGING_LEVEL

contains

   ! ------------------------
   ! Logging.
   ! ------------------------
   subroutine InitLogger()
      ! This routine is called from the constructor to initialize the Logger.
      logical, save                            :: loggerInitialized = .false.
      if ( .not. loggerInitialized ) then
         call New(log, LOGGING_LEVEL)
         loggerInitialized = .true.
      end if
   end subroutine

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction, Copying, and Assignment.
   ! ------------------------
   subroutine NewPrivate(self, fileUnit)
      type (InputReader), intent(out)                :: self
      integer(KINT), intent(in)                      :: fileUnit
      self%fileUnit = fileUnit
      call InitLogger
   end subroutine

   subroutine DeletePrivate(self)
      type (InputReader), intent(inout)              :: self
   end subroutine

   ! ------------------------
   ! Accessors.
   ! ------------------------
   function GetFileUnit(self)
      type (InputReader), intent(in)                 :: self
      integer(KINT)                                  :: GetFileUnit
      GetFileUnit = self%fileUnit
   end function

   ! ------------------------
   ! Other methods.
   ! ------------------------
   subroutine ExtractValueString(self, key, valueString)
      type (InputReader), intent(in)                 :: self
      character(len=*), intent(in)                   :: key
      character(len=*), intent(out)                  :: valueString
      character(len=MAX_LINE_LENGTH)                   :: line
      integer(KINT)                                    :: i, iostat
      logical                                          :: endOfRead

      call LogMessage(log, TRACE_LOGGING_LEVEL, 'Entered ExtractValueString')
      call LogMessage(log, DEBUG_LOGGING_LEVEL, 'key: ' // key)

      ! Locate the line with the right key
      endOfRead = .false.
      rewind(self%fileUnit)
      do while ( .not. LineMatchesKey() .and. .not. endOfRead )
         read(self%fileUnit, '(a)', iostat=iostat) line
         endOfRead = (iostat /= 0)
         call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Line read: ' // trim(line))
      enddo

      ! Read value
      if ( LineMatchesKey() ) then
         i = index(line, ':')      ! Index of colon in line
         valueString = line(i+1:)  ! Copy everything after colon into value
         call LogMessage(log, DEBUG_LOGGING_LEVEL, 'Line matched key with value: '//&
            trim(valueString))
      else
         valueString = ''
         call LogMessage(log, DEBUG_LOGGING_LEVEL, 'No line matched key')
      endif

      call LogMessage(log, TRACE_LOGGING_LEVEL, 'Exiting ExtractValueString')

   contains

      logical function LineMatchesKey()
         integer(KINT) :: s
         s = len(key)
         if ( len(line) < s + 1 ) then
            LineMatchesKey = .false.
         else if ( line(1:s) /= key ) then
            LineMatchesKey = .false.
         else if ( line(s+1:s+1) /= ':' ) then
            LineMatchesKey = .false.
         else
            LineMatchesKey = .true.
         endif
      end function

   end subroutine

   logical function ReadInteger(self, key, value)
      type (InputReader), intent(in)                   :: self
      character(len=*), intent(in)                     :: key
      integer(KINT), intent(inout)                     :: value

      character(len=MAX_LINE_LENGTH)                   :: valueString

      call ExtractValueString(self, key, valueString)
      if ( valueString == '' ) then
         ReadInteger = .false.
         return
      else
         ReadInteger = .true.
```

```
         read(valueString, *) value
      endif

   end function

   logical function ReadReal(self, key, value)
      type (InputReader), intent(in)                    :: self
      character(len=*), intent(in)                      :: key
      real(KREAL), intent(inout)                        :: value

      character(len=MAX_LINE_LENGTH)                    :: valueString

      call ExtractValueString(self, key, valueString)
      if ( valueString == '' ) then
         ReadReal = .false.
         return
      else
         ReadReal = .true.
         read(valueString, *) value
      endif

   end function

end module
```

D.7. Logger.f90

```
module LoggerModule
   use NumberKinds
   use OutputWriterModule
   implicit none

   private
   public Logger, New, Delete, Assignment(=), OutputProperties, LogMessage
   public DEBUG_LOGGING_LEVEL, TRACE_LOGGING_LEVEL, WARNING_LOGGING_LEVEL,
ERROR_LOGGING_LEVEL
   save

   type Logger
      private
      integer(KINT)                                     :: level
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate, NewCopyPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

   interface Assignment(=)
      module procedure AssignPrivate
   end interface

   interface OutputProperties
      module procedure OutputPropertiesPrivate
   end interface

   interface LogMessage
```

```fortran
      module procedure LogMessageReal, LogMessageInteger, LogMessageIntegerArray
      module procedure LogMessageRealArray, LogMessageTextOnly
   end interface

   ! Definitions
   integer(KINT), parameter                             :: DEBUG_LOGGING_LEVEL   = 1
   integer(KINT), parameter                             :: TRACE_LOGGING_LEVEL   = 2
   integer(KINT), parameter                             :: WARNING_LOGGING_LEVEL = 3
   integer(KINT), parameter                             :: ERROR_LOGGING_LEVEL   = 4

contains

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction, Copying, and Assignment.
   ! ------------------------
   subroutine NewPrivate(self,level)
      type (Logger), intent(out)                        :: self
      integer(KINT), intent(in)                         :: level
      self%level = level
   end subroutine

   subroutine NewCopyPrivate(self, other)
      type (Logger), intent(out)                        :: self
      type (Logger), intent(in)                         :: other
      self%level = other%level
   end subroutine

   subroutine DeletePrivate(self)
      type (Logger), intent(inout)                      :: self
   end subroutine

   subroutine AssignPrivate(self, other)
      type (Logger), intent(inout)                      :: self
      type (Logger), intent(in)                         :: other
      self%level = other%level
   end subroutine

   ! ------------------------
   ! Accessors.
   ! ------------------------
   subroutine SetLevel(self, level)
      type (Logger), intent(inout)                      :: self
      integer(KINT), intent(in)                         :: level
      self%level = level
   end subroutine

   function GetLevel(self)
      type (Logger), intent(in)                         :: self
      integer(KINT)                                     :: GetLevel
      GetLevel = self%level
   end function

   ! ------------------------
   ! Other methods.
   ! ------------------------
   subroutine LogMessageTextOnly(self, level, text)
      type (Logger)                                     :: self
      integer(KINT)                                     :: level
      character(len=*)                                  :: text
      if ( level >= self%level ) print *, 'Log: ', text
   end subroutine

   subroutine LogMessageReal(self, level, key, val)
      type (Logger)                                        :: self
      integer(KINT)                                        :: level
      character(len=*)                                     :: key
      real(KREAL)                                          :: val
      if ( level >= self%level ) print *, 'Log: ' // trim(key), val
   end subroutine

   subroutine LogMessageInteger(self, level, key, val)
      type (Logger)                                        :: self
      integer(KINT)                                        :: level
      character(len=*)                                     :: key
      integer(KINT)                                        :: val
      if ( level >= self%level )  print *, 'Log: ' // trim(key), val
   end subroutine

   subroutine LogMessageIntegerArray(self, level, key, vals)
      type (Logger)                                        :: self
      integer(KINT)                                        :: level
      character(len=*)                                     :: key
      integer(KINT)                                        :: vals(:)
      if ( level >= self%level ) then
         print *, 'Log: ' // trim(key)
         print *, vals
      endif
   end subroutine

   subroutine LogMessageRealArray(self, level, key, vals)
      type (Logger)                                        :: self
      integer(KINT)                                        :: level
      character(len=*)                                     :: key
      real(KREAL)                                          :: vals(:)
      if ( level >= self%level ) then
         print *, 'Log: ' // trim(key)
         print *, vals
      endif
   end subroutine

   ! ------------------------
   ! Output properties.
   ! ------------------------
   subroutine OutputPropertiesPrivate(self, writer)
      type (Logger), intent(in)                            :: self
      type (OutputWriter), intent(inout)                   :: writer

      call Write(writer, 'Logging Level', self%level)

   end subroutine

end module
```

D.8. NumberKinds.f90

```
module NumberKinds
   implicit none

   integer, parameter                                      :: KREAL = kind(0.d0)
   integer, parameter                                      :: KINT  = kind(1)
```

```
end module
```

D.9. OutputWriter.f90

```
module OutputWriterModule
   use NumberKinds
   implicit none

   private
   public OutputWriter, New, Delete, Write
   public GetFileUnit, StartSection, EndSection
   save

   type OutputWriter
      private
      integer(KINT)                                     :: fileUnit
      integer(KINT)                                     :: indentLevel
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

   interface Write
      module procedure WriteReal, WriteRealArray, WriteInteger
      module procedure WriteIntegerArray, WriteString
      module procedure WriteRealColumns
   end interface

contains

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction, Copying, and Assignment.
   ! ------------------------
   subroutine NewPrivate(self, fileUnit)
      type (OutputWriter), intent(out)                  :: self
      integer(KINT), intent(in)                         :: fileUnit
      self%fileUnit = fileUnit
      self%indentLevel = 0
   end subroutine

   subroutine DeletePrivate(self)
      type (OutputWriter), intent(inout)                :: self
   end subroutine

   ! ------------------------
   ! Accessors.
   ! ------------------------
   function GetFileUnit(self)
      type (OutputWriter), intent(in)                   :: self
      integer(KINT)                                     :: GetFileUnit
      GetFileUnit = self%fileUnit
   end function

   ! --------------------
   ! Other methods.
   ! --------------------
   function FormatWithIndent(self, formatString)
      type (OutputWriter), intent(in)                   :: self
      character(len=*), intent(in)                      :: formatString
      character(len=len(formatString)+10)               :: FormatWithIndent
      if ( self%indentLevel > 0 ) then
         write(FormatWithIndent, '(a,i2,2a)')'(', 4 * self%indentLevel, 'x,',&
             formatString(2:)
      else
         FormatWithIndent = formatString
      endif
   end function

   subroutine WriteReal(self, key, val)
      type (OutputWriter), intent(in)                   :: self
      character(len=*), intent(in)                      :: key
      real(KREAL), intent(in)                           :: val
      character(len=32)                                 :: form
      form = FormatWithIndent(self, '(a,t40,f18.8)')
      write(self%fileUnit, form) key, val
   end subroutine

   subroutine WriteInteger(self, key, val)
      type (OutputWriter), intent(in)                   :: self
      character(len=*), intent(in)                      :: key
      integer(KINT), intent(in)                         :: val
      character(len=32)                                 :: form
      form = FormatWithIndent(self, '(a,t40,i8)')
      write(self%fileUnit, form) key, val
   end subroutine

   subroutine WriteRealArray(self, key, vals)
      type (OutputWriter), intent(in)                   :: self
      character(len=*), intent(in)                      :: key
      real(KREAL), intent(in)                           :: vals(:)
      character(len=32)                                 :: form
      form = FormatWithIndent(self, '(a)')
      write(self%fileUnit, form) key
      form = FormatWithIndent(self, '(6f12.6)')
      write(self%fileUnit, form) vals
   end subroutine

   subroutine WriteRealColumns(self, key, vals1, vals2)
      type (OutputWriter), intent(in)                   :: self
      character(len=*), intent(in)                      :: key
      real(KREAL), intent(in)                           :: vals1(:), vals2(:)
      integer(KINT)                                     :: i
      character(len=32)                                 :: form
      form = FormatWithIndent(self, '(a)')
      write(self%fileUnit, form) key
      form = FormatWithIndent(self, '(2f12.6)')
      do i = 1, size(vals1)
         write(self%fileUnit, form) vals1(i), vals2(i)
      enddo
   end subroutine

   subroutine WriteIntegerArray(self, key, vals)
      type (OutputWriter), intent(in)                   :: self
      character(len=*), intent(in)                      :: key
      integer(KINT), intent(in)                         :: vals(:)
```

```
      character(len=32)                                 :: form
      form = FormatWithIndent(self, '(a)')
      write(self%fileUnit, form) key
      form = FormatWithIndent(self, '(6i12)')
      write(self%fileUnit, form) vals
   end subroutine

   subroutine WriteString(self, key, str)
      type (OutputWriter), intent(in)                   :: self
      character(len=*), intent(in)                      :: key
      character(len=*), intent(in)                      :: str
      character(len=32)                                 :: form
      form = FormatWithIndent(self,'(a,t40,a)')
      write(self%fileUnit, form) key, str
   end subroutine

   subroutine StartSection(self, sectionName, description)
      type (OutputWriter), intent(inout)                :: self
      character(len=*), intent(in)                      :: sectionName
      character(len=*), intent(in), optional            :: description
      character(len=32)                                 :: form
      form = FormatWithIndent(self,'(a)')
      write(self%fileUnit, form) sectionName
      self%indentLevel = self%indentLevel + 1
      if ( present(description) ) then
         form = FormatWithIndent(self,'(a)')
         write(self%fileUnit, form) description
      endif
   end subroutine

   subroutine EndSection(self)
      type (OutputWriter), intent(inout)                :: self
      self%indentLevel = self%indentLevel - 1
      if ( self%indentLevel < 0 ) &
         stop 'Indentation level dropped below zero in EndSection'
   end subroutine

end module
```

D.10. ParticleEnsemble.f90

```
module ParticleEnsembleModule
   use NumberKinds
   use OutputWriterModule
   use LoggerModule
   implicit none

   private
   public Particle, ParticleEnsemble, New, Delete, Assignment(=), OutputProperties
   public GetParticleMass, GetNumberOfParticles
   public AddParticle, GetParticleAtIndex, SetParticleAtIndex, RemoveAllParticles
   save

   type Particle
      real(KREAL)                                       :: position
   end type

   type ParticleEnsemble
      private
      real(KREAL)                                       :: particleMass
      integer(KINT)                                     :: numParticles
      type(Particle), pointer                           :: particles(:)
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate, NewCopyPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

   interface Assignment(=)
      module procedure AssignPrivate
   end interface

   interface OutputProperties
      module procedure OutputPropertiesPrivate
   end interface

   ! Logging
   type (Logger)              :: log
   integer(KINT), parameter   :: LOGGING_LEVEL = WARNING_LOGGING_LEVEL

contains

   ! ------------------------
   ! Logging.
   ! ------------------------
   subroutine InitLogger()
      ! This routine is called from the constructor to initialize the Logger.
      logical, save :: loggerInitialized = .false.
      if ( .not. loggerInitialized ) then
         call New(log, LOGGING_LEVEL)
         loggerInitialized = .true.
      end if
   end subroutine

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction, Copying, and Assignment.
   ! ------------------------
   subroutine NewPrivate(self, particleMass)
      type (ParticleEnsemble), intent(out)              :: self
      real(KREAL)                                       :: particleMass

      self%particleMass = particleMass
      self%numParticles = 0
      allocate(self%particles(100))   ! Allocate enough space for 100 particles
      call InitLogger
   end subroutine

   subroutine NewCopyPrivate(self, other)
      type (ParticleEnsemble), intent(out)              :: self
      type (ParticleEnsemble), intent(in)               :: other
      integer(KINT)                                     :: i

      self%particleMass = other%particleMass
      self%numParticles = other%numParticles

      allocate(self%particles(other%numParticles))
      self%particles(:self%numParticles) = other%particles(:self%numParticles)

   end subroutine

   subroutine DeletePrivate(self)
      type (ParticleEnsemble), intent(inout)           :: self
      integer(KINT)                                    :: i
      deallocate(self%particles)
   end subroutine

   subroutine AssignPrivate(self, other)
      type (ParticleEnsemble), intent(inout)           :: self
      type (ParticleEnsemble), intent(in)              :: other
      integer(KINT)                                    :: i

      call Delete(self)
      call New(self, other)

   end subroutine

   ! ------------------------
   ! Accessors.
   ! ------------------------
   function GetParticleMass(self)
      type (ParticleEnsemble), intent(in)              :: self
      real(KREAL)                                      :: GetParticleMass
      GetParticleMass = self%particleMass
   end function

   function GetNumberOfParticles(self)
      type (ParticleEnsemble), intent(in)              :: self
      integer(KINT)                                    :: GetNumberOfParticles
      GetNumberOfParticles = self%numParticles
   end function

   function GetParticleAtIndex(self, index) result(p)
      type (ParticleEnsemble), intent(in)              :: self
      integer(KINT), intent(in)                        :: index
      type (Particle)                                  :: p
      p = self%particles(index)
   end function

   subroutine SetParticleAtIndex(self, index, p)
      type (ParticleEnsemble), intent(inout)           :: self
      integer(KINT), intent(in)                        :: index
      type (Particle), intent(in)                      :: p
      call GrowParticlesArray(self, index)
      self%particles(index) = p
   end subroutine

   ! ------------------------
   ! Other methods.
   ! ------------------------
   subroutine GrowParticlesArray(self, minSize)
      type(ParticleEnsemble), intent(inout)            :: self
      integer(KINT), intent(in)                        :: minSize

      integer(KINT)                                    :: newSize
      type(Particle), pointer                          :: tempParticles(:)
```

```
      if ( minSize > size(self%particles) ) then
         ! allocate a new array that is at least twice as big as the old one, copy
         ! across existing particles. Then clean up the old array, and reset pointer.
         newSize = max(size(self%particles), minSize) * 2
         allocate( tempParticles(newSize) )
         tempParticles(:self%numParticles) = self%particles(:self%numParticles)
         deallocate(self%particles)
         self%particles => tempParticles
      end if

   end subroutine

   subroutine AddParticle (self, p)
      type(ParticleEnsemble), intent(inout)             :: self
      type(Particle), intent(in)                        :: p

      call GrowParticlesArray(self, self%numParticles+1)
      self%particles(self%numParticles+1) = p
      self%numParticles = self%numParticles + 1

   end subroutine

   subroutine RemoveAllParticles (self)
      type(ParticleEnsemble), intent(inout)             :: self
      integer(KINT)                                     :: i
      self%numParticles = 0
   end subroutine

   ! ------------------------
   ! Output properties.
   ! ------------------------
   subroutine OutputPropertiesPrivate(self, writer)
      type (ParticleEnsemble), intent(in)               :: self
      type (OutputWriter), intent(inout)                :: writer

      integer(KINT)                                     :: i

      ! Write details of ADT instance to the writer
      call Write(writer, 'Particle Mass', self%particleMass)
      call Write(writer, 'Number of Particles', self%numParticles)

      ! write the particles
      call StartSection(writer, 'Particles in Ensemble')
      do i = 1, self%numParticles
         call Write(writer, 'Particle Index', i)
         call Write(writer, 'Position of Particle', self%particles(i)%position)
      end do
      call EndSection(writer)

   end subroutine

end module
```

D.11. Potential.f90

```
module PotentialModule
   use NumberKinds
   use OutputWriterModule
   use LoggerModule
   implicit none
   private
   public Potential, New, Delete, assignment(=), OutputProperties
   public SetPotentialForm, GetPotentialForm, ValueAtCoord
   public HARMONIC_POTENTIAL, MORSE_POTENTIAL, LJ_POTENTIAL
   save

   type Potential
      private
      integer(KINT) :: potentialForm
   end type

   ! Overloaded procedure interfaces
   interface New
      module procedure NewPrivate
   end interface

   interface Delete
      module procedure DeletePrivate
   end interface

   interface OutputProperties
      module procedure OutputPropertiesPrivate
   end interface

   ! Definitions
   integer(KINT), parameter  :: HARMONIC_POTENTIAL  = 1
   integer(KINT), parameter  :: MORSE_POTENTIAL     = 2
   integer(KINT), parameter  :: LJ_POTENTIAL        = 3

   ! Logging
   type (Logger)             :: log
   integer(KINT), parameter  :: LOGGING_LEVEL = WARNING_LOGGING_LEVEL

contains

   ! ------------------------
   ! Logging.
   ! ------------------------
   subroutine InitLogger()
      ! This routine is called from the constructor to initialize the Logger.
      logical, save :: loggerInitialized = .false.
      if ( .not. loggerInitialized ) then
         call New(log, LOGGING_LEVEL)
         loggerInitialized = .true.
      end if
   end subroutine

   ! ------------------------
   ! Standard ADT Methods. Construction, Destruction, Copying, and Assignment.
   ! ------------------------
   subroutine NewPrivate(self,form)
      type (Potential), intent(out)                    :: self
      integer(KINT), intent(in)                        :: form
      self%potentialForm = form
      call InitLogger
   end subroutine

   subroutine DeletePrivate(self)
      type (Potential), intent(inout)                  :: self
   end subroutine
```

```fortran
! -----------------------
! Accessors.
! -----------------------
subroutine SetPotentialForm(self, form)
   type (Potential), intent(inout)                   :: self
   integer(KINT), intent(in)                         :: form
   self%potentialForm = form
end subroutine

function GetPotentialForm(self)
   type (Potential), intent(in)                      :: self
   integer(KINT)                                     :: GetPotentialForm
   GetPotentialForm = self%potentialForm
end function

! -----------------------
! Other methods.
! -----------------------
function ValueAtCoord(self, coord)
   type (Potential), intent(in)                      :: self
   real(KREAL), intent(in)                           :: coord
   real(KREAL)                                       :: valueAtCoord
   call LogMessage(log, TRACE_LOGGING_LEVEL, 'Entered ValueAtCoord')

   select case (self%potentialForm)
   case (HARMONIC_POTENTIAL)
      valueAtCoord = (coord-1)**2
      call LogMessage(log, DEBUG_LOGGING_LEVEL, &
         'Harmonic potential with value', valueAtCoord)
   case (MORSE_POTENTIAL)
      valueAtCoord = (1-exp(-coord+1))**2
      call LogMessage(log, DEBUG_LOGGING_LEVEL, &
         'Morse potential with value:', valueAtCoord)
   case (LJ_POTENTIAL)
      valueAtCoord = 4*(1/(coord**12) - 1/(coord**6))
      call LogMessage(log, DEBUG_LOGGING_LEVEL, &
        '12/6 Lennard Jones potential with value:', valueAtCoord)
   case default
      stop 'Invalid potential form'
   end select

   call LogMessage(log, TRACE_LOGGING_LEVEL, 'Exiting ValueAtCoord')
end function

! -----------------------
! Output properties.
! -----------------------
subroutine OutputPropertiesPrivate(self, writer)
   type (Potential), intent(in)                      :: self
   type (OutputWriter), intent(inout)                :: writer

   select case (self%potentialForm)
   case (HARMONIC_POTENTIAL)
      call Write(writer, 'Potential Type', 'Harmonic')
   case (MORSE_POTENTIAL)
      call Write(writer, 'Potential Type', 'Morse')
   case (LJ_POTENTIAL)
      call Write(writer, 'Potential Type', 'Lennard-Jones')
   case default
```

```
         stop 'Invalid potential form'
      end select

   end subroutine

end module
```