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 `fmkmk.pl` tool. To create a make file, you just run `fmkmk.pl` on the main file of your program; it will figure out what other files are needed, and any dependencies that exist.

```
fmkmk.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 = 1.5000012345678912345
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 1.0×10^{-5} of one another. (`1.0e-5` means 1.0×10^{-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

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.

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.