

Data Analysis in Geophysics

ESCI 7205

Class 23

Bob Smalley

Short intro FORTRAN.

Common Languages used in Scientific programming

What is the best language to learn?

That depends on what you want to do.

Most common languages used for scientific
programming
(in no particular order – but looks chronological)

Fortran

C

C++

Matlab

Pearl

Evolution of a programmer

High School/Jr.High

```
10 PRINT "HELLO WORLD"  
20 END
```

Prints "HELLO WORLD" to screen.
(don't really need the second line).

First year in College

```
program Hello(input, output)
begin
  writeln('Hello World')
end.
```

Prints “Hello World” to the screen.
Twice as “long”.

Senior year in College

```
(defun hello  
  (print  
    (cons 'Hello (list 'World))))
```

Prints “Hello World” to screen in LISP
which stands for (Lots of ((Irritating (Spurious) (Parenthesis)))

New professional

```
#include <stdio.h>
void main(void)
{
    char *message[] = {"Hello ", "World"};
    int i;

    for(i = 0; i < 2; ++i)
        printf("%s", message[i]);
    printf("\n");
}
```

Prints “Hello World” to screen.
In C

Seasoned professional

```
#include <iostream.h>
#include <string.h>
class string
{
private:
    int size;
    char *ptr;
string() : size(0), ptr(new char[1]) { ptr[0] = 0; }
    string(const string &s) : size(s.size)
    {
        ptr = new char[size + 1];
        strcpy(ptr, s.ptr);
    }
    ~string()
    {
        delete [] ptr;
    }
    friend ostream &operator <<(ostream &, const string &);
    string &operator=(const char *);
};
ostream &operator<<(ostream &stream, const string &s)
{
    return(stream << s.ptr);
}
string &string::operator=(const char *chrs)
{
    if (this != &chrs)
    {
        delete [] ptr;
        size = strlen(chrs);
        ptr = new char[size + 1];
        strcpy(ptr, chrs);
    }
    return(*this);
}
int main()
{
    string str;
    str = "Hello World";
    cout << str << endl;
    return(0);
}
```

Prints “Hello World” to screen

Master Programmer

```
[
  uuid(2573F8F4-CFEE-101A-9A9F-00AA00342820)
]
library LHello
{
  // bring in the master library
  importlib("actimp.tlb");
  importlib("actexp.tlb");
  // bring in my interfaces
  #include "pshlo.idl"
  [
    uuid(2573F8F5-CFEE-101A-9A9F-00AA00342820)
  ]
  cotype THello
{
  interface IHello;
  interface IPersistFile;
  };
};
[
  exe,
  uuid(2573F890-CFEE-101A-9A9F-00AA00342820)
]
module CHelloLib
{
  // some code related header files
  importheader(<windows.h>);
  importheader(<ole2.h>);
  importheader(<except.hxx>);
  importheader("pshlo.h");
  importheader("shlo.hxx");
  importheader("mycls.hxx");
  // needed typelibs
  importlib("actimp.tlb");
  importlib("actexp.tlb");
  importlib("thlo.tlb");

  [
    uuid(2573F891-CFEE-101A-9A9F-00AA00342820),
    aggregatable
  ]
  coclass CHello
  {
    cotype THello;
  };
};
#include "ipfix.hxx"
extern HANDLE hEvent;
class CHello : public CHelloBase
{
public:
  IPFIX(CLSID_Chello);
  CHello(IUnknown *pUnk);
  ~CHello();
  HRESULT __stdcall PrintSz(LPWSTR pwszString);
private:
  static int cObjRef;
};

#include <windows.h>
#include <ole2.h>
#include <stdio.h>
#include <stdlib.h>

#include "thlo.h"
#include "pshlo.h"
#include "shlo.hxx"
#include "mycls.hxx"
int CHello::cObjRef = 0;
CHello::CHello(IUnknown *pUnk) : CHelloBase(pUnk)
{
  cObjRef++;
  return;
}
HRESULT __stdcall CHello::PrintSz(LPWSTR pwszString)
{
  printf("%ws", pwszString);
  return(ResultFromCode(S_OK));
}
CHello::~CHello(void)
{
  // when the object count goes to zero, stop the server
  cObjRef--;
  if( cObjRef == 0 )
    PulseEvent(hEvent);
  return;
}
#include <windows.h>
#include <ole2.h>
#include "pshlo.h"
#include "shlo.hxx"
#include "mycls.hxx"
HANDLE hEvent;
int _cdecl main(
  int argc,
  char * argv[]
) {
  ULONG ulRef;
  DWORD dwRegistration;
  CHelloCF *pCF = new CHelloCF();
  hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  // Initialize the OLE libraries
  CoInitializeEx(NULL, COINIT_MULTITHREADED);
  CoRegisterClassObject(CLSID_Chello, pCF, CLSCTX_LOCAL_SERVER,
    REGCLS_MULTIPLEUSE, &dwRegistration);
  // wait on an event to stop
  WaitForSingleObject(hEvent, INFINITE);
  // revoke and release the class object
  CoRevokeClassObject(dwRegistration);
  ulRef = pCF->Release();
  // Tell OLE we are going away.
  CoUninitialize();
  return(0); }
extern CLSID CLSID_Chello;
extern UUID LIBID_ChelloLib;
CLSID CLSID_Chello = { /* 2573F891-CFEE-101A-9A9F-00AA00342820
*/
  0x2573F891,
  0xCFEE,
  0x101A,
  { 0x9A, 0x9F, 0x00, 0xAA, 0x00, 0x34, 0x28, 0x20 }
};
UUID LIBID_ChelloLib = { /* 2573F890-
CFEE-101A-9A9F-00AA00342820 */
  0x2573F890,
  0xCFEE,
  0x101A,
  { 0x9A, 0x9F, 0x00, 0xAA, 0x00, 0x34, 0x28, 0x20 }
};
};
#include <windows.h>
#include <ole2.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int _cdecl main(
  int argc,
  char * argv[]
) {
  HRESULT hRslt;
  IHello *pHello;
  ULONG ulCnt;
  IMoniker * pmk;
  WCHAR wcsT[_MAX_PATH];
  WCHAR wcsPath[2 * _MAX_PATH];
  // get object path
  wcsPath[0] = '\\0';
  wcsT[0] = '\\0';
  if( argc > 1) {
    mbstowcs(wcsPath, argv[1], strlen(argv[1]) + 1);
    wcsupr(wcsPath);
  }
  else {
    fprintf(stderr, "Object path must be specified\n");
    return(1);
  }
  // get print string
  if(argc > 2)
    mbstowcs(wcsT, argv[2], strlen(argv[2]) + 1);
  else
    wcsncpy(wcsT, L"Hello World");
  printf("Linking to object %ws\n", wcsPath);
  printf("Text String %ws\n", wcsT);
  // Initialize the OLE libraries
  hRslt = CoInitializeEx(NULL, COINIT_MULTITHREADED);
  if(SUCCEEDED(hRslt)) {
    hRslt = CreateFileMoniker(wcsPath, &pmk);
    if(SUCCEEDED(hRslt))
      hRslt = BindMoniker(pmk, 0, IID_IHello, (void **)&pHello);
    if(SUCCEEDED(hRslt)) {
      // print a string out
      pHello->PrintSz(wcsT);

      Sleep(2000);
      ulCnt = pHello->Release();
    }
    else
      printf("Failure to connect, status: %lx", hRslt);
    // Tell OLE we are going away.
    CoUninitialize();
  }
  return(0);
}
```

Prints “Hello World” to screen

Fortran

(FORmula TRANslator)

We are going to assume you have already mastered programming ideas like `if/then/else`, various kinds of loops, `i/o` formatting, etc.

So this is going to be a firehose presentation.

You will come across two versions of FORTRAN,
77 and 90/95

FORTRAN is a high-level language designed for
number crunching.

Unlike MATLAB, it is not interactive. It must be
translated into the low-level machine language as
a separate step in order to run.

This is done via a compiler and yields an
executable program specific to that "platform"

<http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/intro.html>

Basics of Fortran

Simple programs have the following
"structure" (used loosely – Fortran is not a
"structured" programming language) –

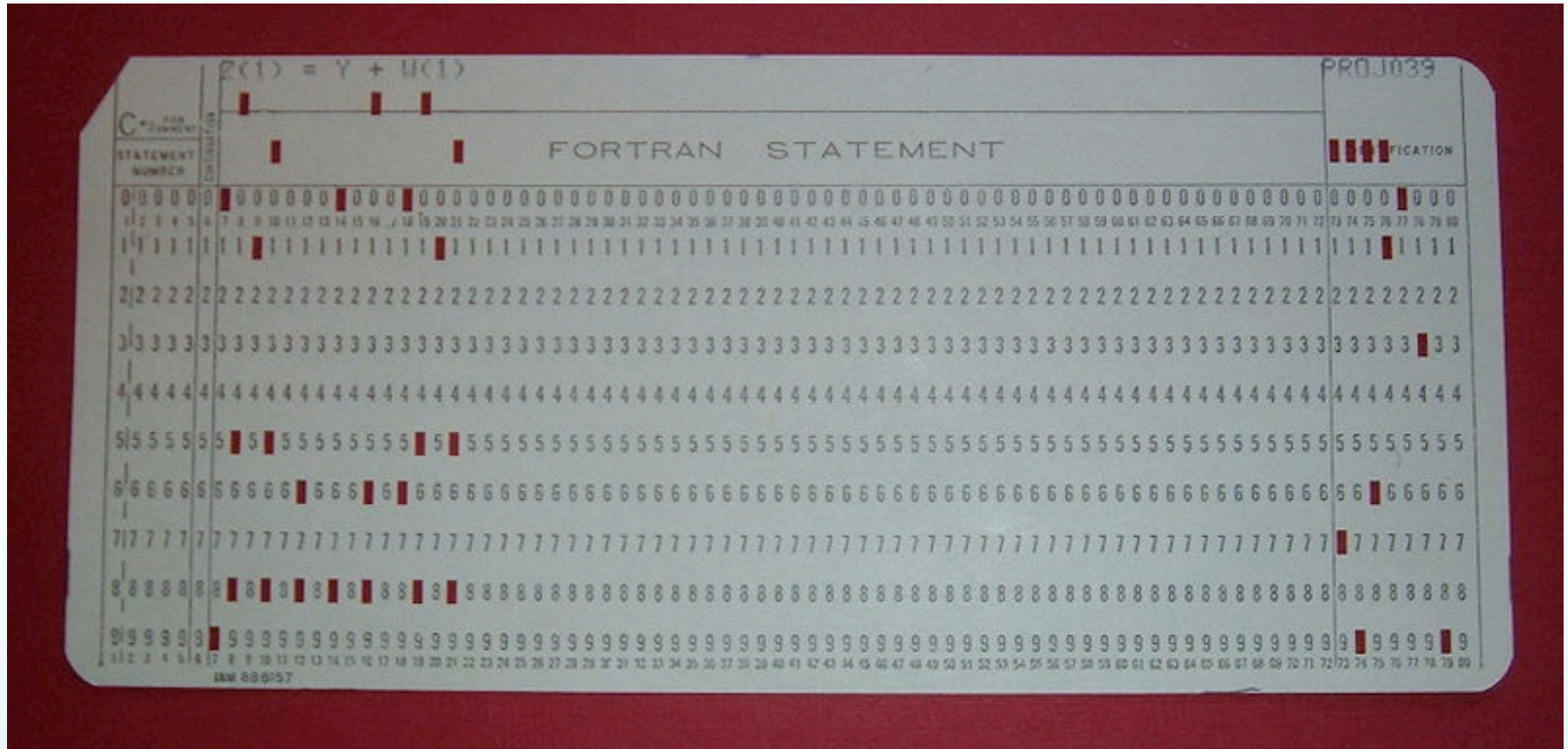
Comments
Common block inclusions
Variable declarations
Program

The power of UNIX

The file name for all source files input to the Fortran compiler must end in `.f`

If you don't like this use the power of UNIX to write your own OS with your own rules.

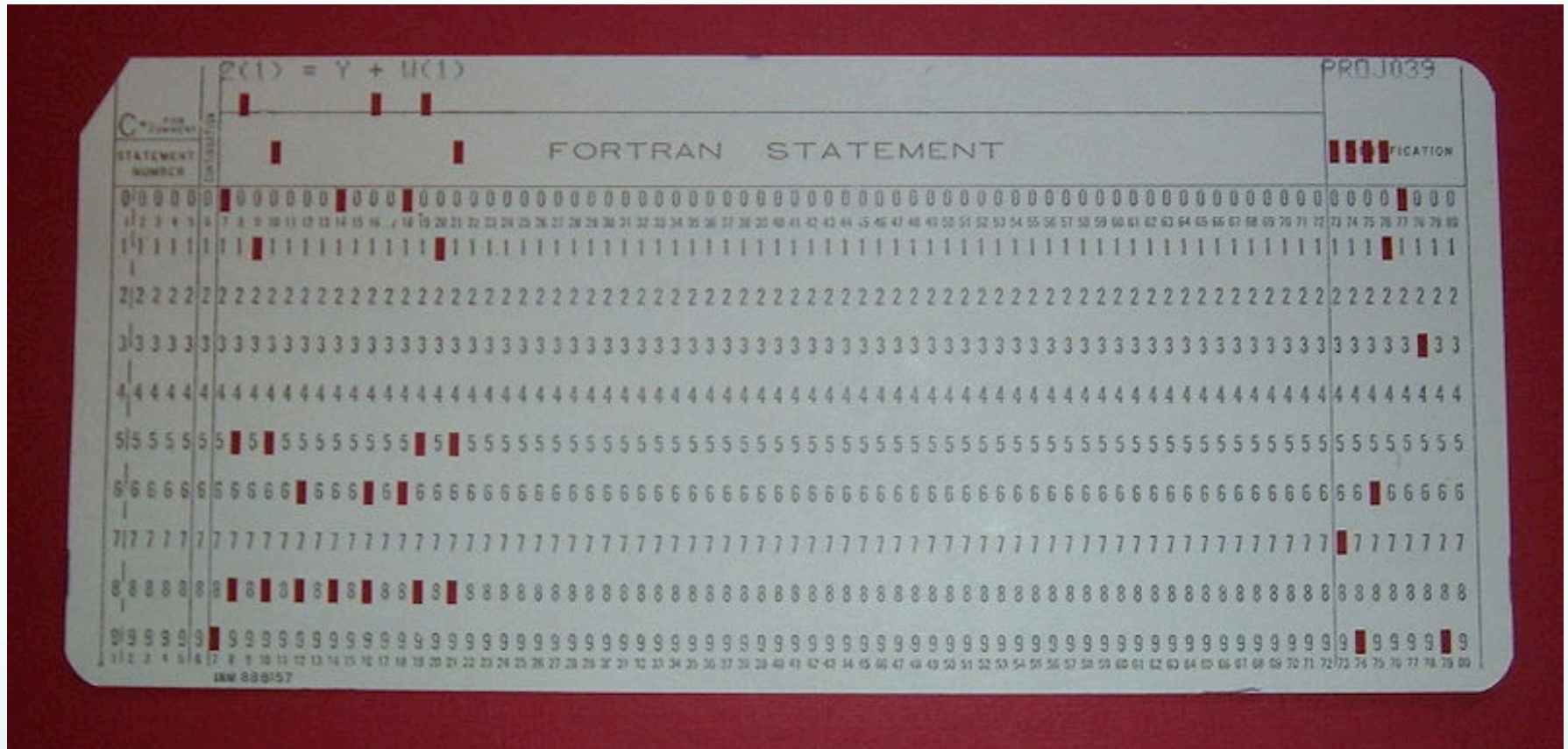
Column formatting – designed for "cards"



Fortran follows a specific line format.

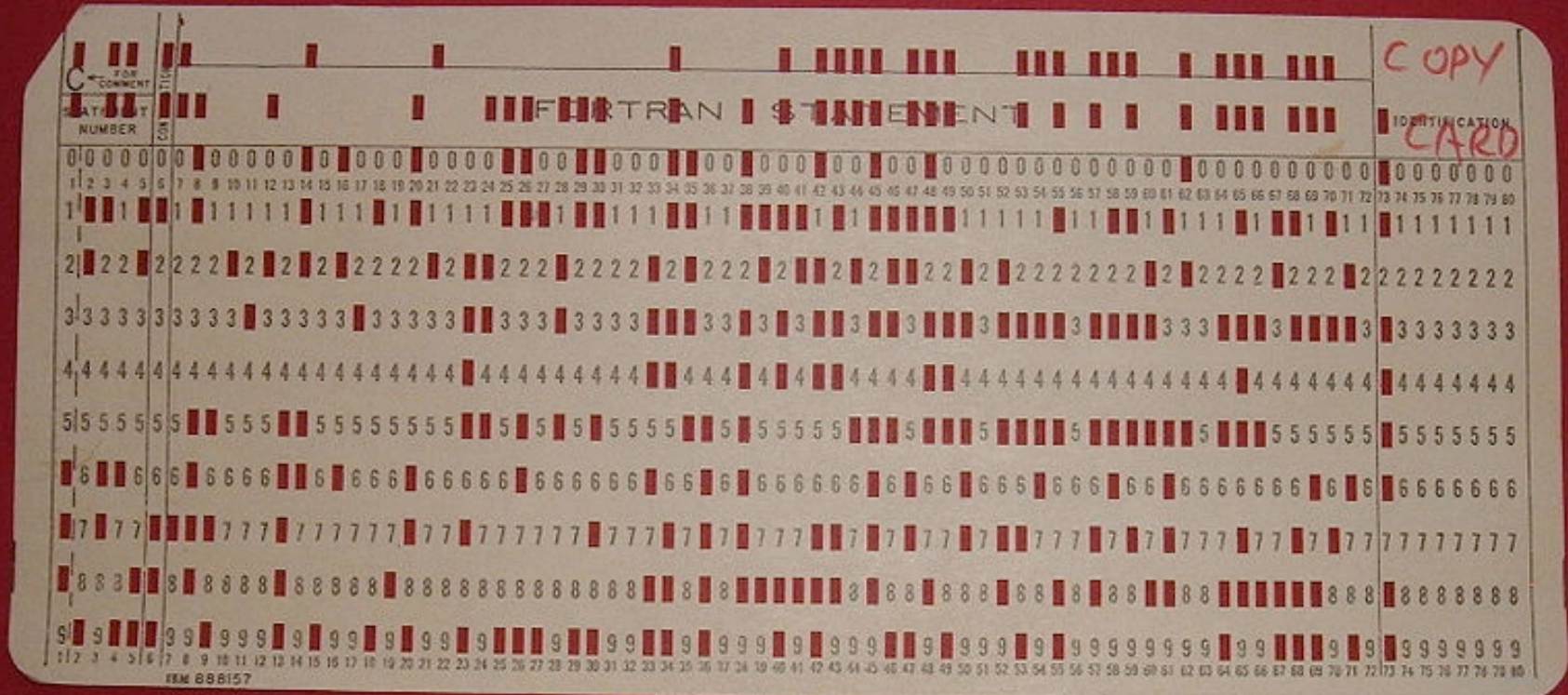
It assumes input is coming from one of these.

Column formatting – designed for "cards"



- columns 1-5: reserved for statement numbers
- column 6: reserved for continuation symbol (&)
- columns 7-72: statement
- Columns 73-80: line/card numbers

Column formatting



As a result, old f77 programs cannot contain
statement text after column 72!

Newer fortran 90/95 does allow for free-format (info past column 72) and you can override the fixed format using compiler flags for f77

The first line of a segment of Fortran source code (a program) (in a file) indicates what it is (this declaration for the main program is actually optional)

```
program [name of program]
```

```
subroutine cluster1(log, nev, ndt  
& , idata, minobs_cc, minobs_ct  
& , dt_c1, dt_c2, ev_cusp  
& , clust, noclust, nclust)
```

*Note, the indented “&” – which is in, and has to be in column 6, indicates a line continuation

Most modern Fortran (77 or later) will let you start lines with a tab (which is the equivalent of 8 spaces)

If the first character after the tab is one of the digits 1–9 it interprets it as a continuation (there is no significance to the digit you put there – you can use the same one over and over, etc. It just has to be a non-zero digit.)

```
subroutine cluster1(log, nev, ndt  
1, idata, minobs_cc, minobs_ct  
4, dt_c1, dt_c2, ev_cusp  
1, clust, noclust, nclust)
```

The last line of the segment (a program) needs to indicate the segment (program) is finished (in this case, again, is optional)

end

Variables

Variables do not need to be declared in Fortran

But should be (and enforced with **IMPLICIT NONE**) unless you like debugging.

Variable typing – Fortran has implicit typing of variables (but depending on this will produce many hours of debugging fun).

the default is

```
IMPLICIT REAL(A-H,O-Z)
```

```
INTEGER(I-N)
```

All variables beginning with A-H and O-Z are real, and those beginning with I-N are integer.

And you can specify new implicit variable definitions

```
IMPLICIT REAL(A-H)
IMPLICIT DOUBLE(O-Z)
IMPLICIT LOGICAL(K)
IMPLICIT INTEGER(I-J,L-N)
```

But are still in the dangerous world of implicit definitions.

Variable typing – turning off implicit variable naming

IMPLICIT NONE

Unfortunately this is not standard in F-77 (so you should not use it!), but very useful (all rules are made to be broken!!).

Gives the "Pascal (also C) convention" that all variables have to be specified – turns off all implicit definitions.

On the Sun the same effect can be obtained with the switch `-u` in the compilation command

The huge benefit of `IMPLICIT NONE` is that it will catch most of *your* typing errors (unless, unlike me, you are a perfect typer).

Without using `IMPLICIT NONE`, new variables are created (by typos) as they show up in your source code.

A typo makes a new variable!
You now have a bug in your program and have to "debug" it.

The First Computer Bug

Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1947.

The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found".

They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program".

9/9

0800 Antan started
 1000 " stopped - antan ✓
 1300 (032) MP - MC ~~1.982147000~~
 (033) PRO 2 2.130476415
 convd 2.130676415
 Relays 6-2 in 033 failed special speed test
 in relay " 10.000 test.

Relay
 2145
 Relay 3370

1100 Started Cosine Tape (Sine check)
 1525 Started Mult+ Adder Test.

1545



Relay #70 Panel F
 (moth) in relay.

First actual case of bug being found.
 1630 Antan started.
 1700 closed down.

Comments

A “C” or “c” (Fortran is case insensitive!) in column 1 is used to indicate the “line”/statement/card is a comment

```
c Version 1.0 - 03/2001
```

```
c Author: Felix Waldhauser, felix@andreas.wr.usgs.gov
```

```
C
```

```
c started 03/1999
```

A “!” after a Fortran statement, indicates a comment at the end of a statement (it may also be placed at the beginning of the line)

```
integer log! Log-file identifier
```

Newmann and Goldstine

Series of reports:
Planning and Coding Problems for an Electronic
Computing Instrument

Published “dozens of routines for mathematical computation with the expectation that some lowly “coder” would be able to convert them into working programs.” (Sci. Am., Dec 2009)

Newmann and Goldstine

But people quickly found that “the process of writing programs and getting them to work was excruciating difficult.” (Sci. Am., Dec. 2009)

And Wilkes observed in his Memoirs

“the realization came over me with full force that a good part of the remainder of my life was going to be spent finding errors in my own programs”

Fortran numeric variable types include:

integer: integers (short, regular, long, quad)

real: floating point number (single, double, quad)

complex: complex number (single, double, quad)

logical: logical value (i.e., true or false).

string variable types include

character: character string of a certain length
(≤ 256 long).

Declaring variables

Here are some examples of variable declaration

```
integer  dt_idx(MAXDATA) !default length integer array
                                !declaration using parameter
integer*2 dt_idy          !explicitly define 2 byte integer
integer*4 dt_idz          !explicitly define 4 byte integer
integer*4 dt_idz          !explicitly define 4 byte integer
real      acond           !single precision scalar declaration
real*4    bcond           !double precision scalar declaration
double    at_idx(1000)    !double precision vector declaration
character dt_sta(MAXDATA)*7 !array of strings with length 7
```

or

```
INTEGER :: ZIP, Mean, Total (90/95 only)
```

Variables must be declared at the beginning of
your program.

Have to specify size of arrays upon definition
(this amount of memory becomes part of the
program – does not allocate dynamically – like
Matlab or C)

Can hard-code or use "parameter" syntax.

```
integer MAXDATA  
parameter (MAXDATA=1000)
```

```
integer  dt_idx(1000) !default length integer array  
                                !declaration  
double   at_idx(MAXDATA) !double precision vector  
character dt_sta(MAXDATA)*7 !string with length
```

MAXDATA is regular integer variable that you can
use like any other variable, plus its value is known
to the compiler

Can do simple arithmetic in parameter statements.

Variable defined can be used by compiler and is also regular variable.

You can also reassign the value of the variable in your program.

```
integer MAXDATA  
integer ANOTHERMAX  
parameter (MAXDATA=1000)  
parameter (ANOTHERMAX=2*MAXDATA)  
real pi  
parameter (pi=atan(1))
```


Except for the content of strings, Fortran is not case sensitive (A is the same as a) .

So as a variable “DENS”, “dens”, “Dens” are all the same.

In a comparison of the contents of character variables, however, “A” is not equal to “a”.

Assigning values to variables

```
real X  
integer I  
character*2 plate
```

```
X=2.3
```

```
I=4
```

```
plate='sa'
```

There is no special syntax (\$, @, etc.) for accessing the value of a variable.

You don't have to end statements with a ";"

You should initialize your variables to be sure they start at 0

(or where you want them to start – else they may have whatever was sitting that spot in memory – depends on how person who wrote the Fortran compiler wrote the code as what do to in this case is not defined in the language definition.).

```
minwght= 0.00001  
rms_ccold= 0  
rms_ctold= 0  
rms_cc0old= 0  
rms_ct0old= 0
```

```
c--- get input parameter file name:
```

```
c narguments = iargc() !similar to argc in C, counts number of c  
c command line input parameters
```

you can initialize a variable when specifying the type (F90/95)

```
REAL :: Offset = 0.1, Length = 10.0, tolerance = 1.E-7
```

In Fortran you can put blank lines, tabs, and spaces as you like for readability (between col 7 and 72 –the first 5 characters are for statement numbers, 6th for continuation).

Parameters

You can define constants of any type by using the parameter call

```
INTEGER, PARAMETER :: Limit = 30, Max_Count = 100
```

or

```
integer*4 MAXEVE, MAXDATA, MAXCL  
parameter(MAXEVE= 13000  
&          , MAXDATA= 1300000  
&          , MAXCL=    50)
```

(I usually put the comma separating variables at the beginning of the continuation line, rather than at the end of the line being continued. If I have to comment out that line for some reason ~ it saves me from having to fix the previous line by editing out the comma.)

Global Variables - Common blocks
collections of variables that can be shared
between different parts of the program (main,
subroutines).

This is a way to specify that certain variables
should be shared among different subroutines.

In general, those that give advice about
programming suggest that, the use of common
blocks should be minimized.

Common blocks

```
program main  
  real alpha, beta  
  common /coeff/ alpha, beta  
  . . . Statements . . .  
  stop  
end
```

```
subroutine subl (some arguments – but not alpha or beta)  
  real alpha, beta  
  common /coeff/ alpha, beta  
  . . . Statements . . .  
  return  
end
```

The main program and subroutine will physically share the memory in the common block.
(they already share the memory of the variables passed to the subroutine)

Since memory is physically shared, we don't have to use the same names or even the same types in the different instances of the “named” common block. (can be handy, and very dangerous)

```
program main
real*4 alpha, beta
common /coeff/ alpha, beta
. . . Statements . . .
stop
end
```

```
subroutine subl (some arguments – but not alpha or beta)
Integer*4 delta, gamma
common /coeff/ delta, gamma
. . . Statements . . .
return
end
```

Common blocks can also be “unnamed” (just leave out the “/name/” – there is only “one” of these)

INCLUDE statements

INCLUDE statements insert the entire contents of a separate text file into the source code (ex: “`include mydefs.inc`”, include files normally have “.inc” as their “extension”).

This feature can be particularly useful when the same set of statements has to be used in several different program units.

INCLUDE statements

Such is often the case when defining a set of constants using **PARAMETER** statements, or when declaring common blocks with a set of **COMMON** statements (without the common below, the variables would be local to each subroutine).

```
include 'hypoDD.inc' !in the main program hypoDD.f
```

contents of file hypoDD.inc

```
integer*4 MAXEVE, MAXDATA, MAXSTA
c params for medium size problems (e.g.: SUN ULTRA-2,768 MB RAM)
parameter(MAXEVE= 13000
&          , MAXDATA= 1300000
&          , MAXSTA= 2000)
common /mycommon/MAXEVE, MAXDATA, MAXSTA
```

Fortran Operators

<u>Type</u>	<u>Operator</u>	<u>Associativity</u>
Arithmetic		
Mult, Div	*, /	left to right
Add, Sub	+, -	left to right
Exponentiation	**	right to left

Fortran Operators

<u>Type</u>	<u>Operator</u>	<u>Associativity</u>
Relational		
Less than, less than or equal	.lt. (<), .le. (<=)	none
Greater than, greater than or equal	.gt. (>), .ge. (>=)	
“()” indicates 90/95 convention		
Equal, not equal	.eq. (==), .ne. (/=)	
	! is negation	

Fortran Operators

<u>Type</u>	<u>Operator</u>	<u>Associativity</u>
-------------	-----------------	----------------------

Logical

.not.	right to left
.and.	left to right
.or.	left to right
.eq. .ne.	left to right

Logical variables take on the values `.true.`
and `.false.`

if/endif
if/else/endif
if/elseif/endif

```
if (iflrai(no,neit).eq.1) then      ! note the testing syntax
    ttime= temps
else if (iflrai(no,neit).eq.2) then
    ttime = atim
    if(iheter1.eq.3) then
        if(isp.eq.0) then
            secp(no,neit)=seco(neit)+pdl(ji)+ttime
        else
            secp(no,neit)=seco(neit)+sdl(ji)+ttime
        endif
    endif
endif
endif
```

goto/go to

(remember spacing not important in Fortran)

One of the best features of Fortran is the ability to quickly jump to (almost) anywhere in the code.

One of the worst features of Fortran is the ability to quickly jump to (almost) anywhere in the code.

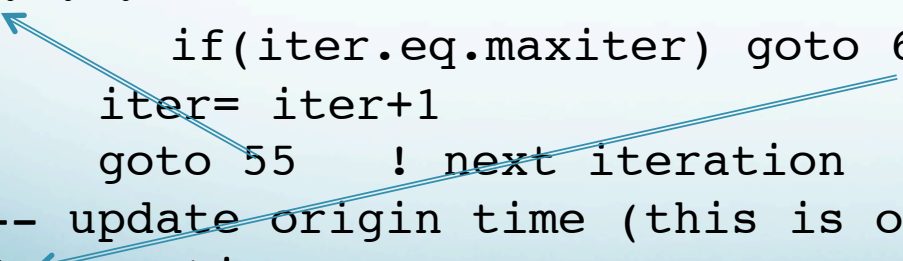
goto/go to

Any command or block may be labeled using a numeric number.

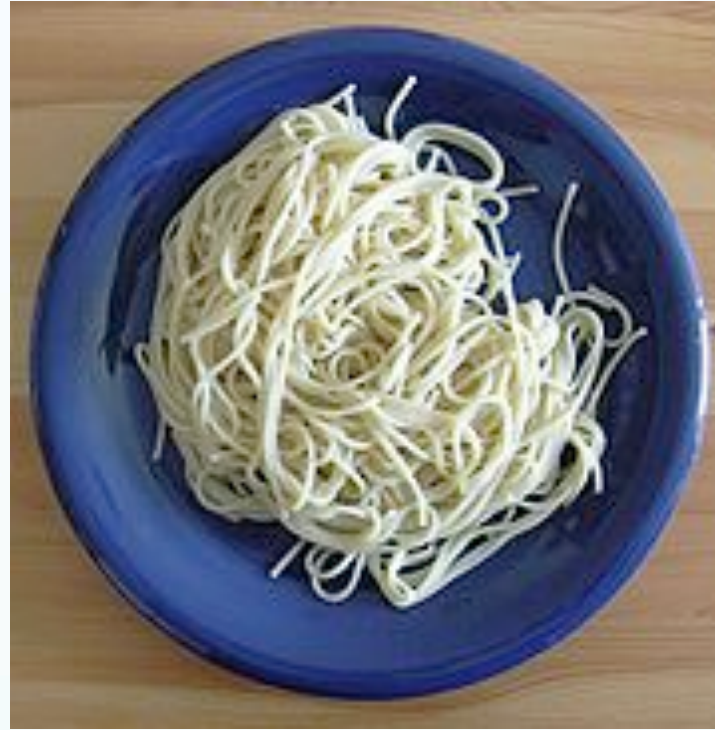
Then you can use the goto command to jump to that line (label).

Labels must be unique.

```
55 . . .  
56     if(iter.eq.maxiter) goto 600           ! all iterations done.  
      iter= iter+1  
      goto 55    ! next iteration  
c--- update origin time (this is only done for final output!!)  
600 ← continue
```

A diagram with blue arrows illustrating the jumps in the code. One arrow points from the 'goto 600' statement on line 56 to the label '600' at the bottom. Another arrow points from the 'goto 55' statement on line 56 to the label '55' at the top left.

Problem with indiscriminant use of “go to”s is spaghetti code.



Disorganized structure of code makes validation
(making sure code does what you want it to), debugging and
maintenance difficult to impossible.
(program flow tends to look like a bowl of spaghetti, i.e. twisted and tangled.
[Wikipedia])

See also

Ravioli code (good)

Lasagna code (good)

Spaghetti and meatballs code (bad ravioli code)

do/endo
do # / # continue
aka the “do loop”

Two forms

1st form - block form (do-endo)

```
mbad= 0
k= 1
do i= 1,nsrc
    if(src_dep(i) + (src_dz(i)/1000).lt.0) then
        amcusp(k)= ev_cusp(i)
        k=k+1
    endif
enddo
mbad= k-1      ! number of neg depth events
```

Indenting to make it more readable, maintainable.

2nd form - statement number form

(labelled statement at end of do loop can be executable statement, eg. $x=x+1$, or non-executable - continue)

```
do 23184 l=1,j1
  if (.not.(v(l).gt.vlmax)) goto 23186
  lmax = l
  tklmax = thk(l)
  vlmax = v(l)
```

```
23186      continue
```

```
23184 continue
```

Numbered line can be executable or not (continue) and be shared between multiple do loops.

(labelled statement at end of do loop can be executable statement, eg. $x=x+1$, or non-executable – continue)

```
do 23184 l=1,j1
  do 23184 m=1,j2
    if (.not.(v(l).gt.vlmax)) goto 23186
    lmax = l
    tkxmax = thk(l)
    vlmax = v(l)
```

```
23186      continue
```

```
23184 continue
```

do/while-loops

```
while (logical expr) do  
...   statements ...  
enddo
```

Or

```
do while (logical expr)  
...   statements ...  
enddo
```


semi-infinite loop

```
do while (.true.)
```

```
...   statements ...
```

```
    test with go to
```

```
...   statements ...
```

```
enddo
```

What is the value of loop counter (1 in this case) when I leave the loop? (can I depend on 1's value and use it for something?)

It depends on how the loop “terminates”

```
do 23184 l=1,j1
    if (.not.(v(l).gt.vlmax)) goto 23186
    lmax = l
    tklmax = thk(l)
```

```
23184 continue
```

• • •

If I'm here the loop ran to "completion" and 1 is undefined (we cannot be sure its value is j1). Solution save 1 into another variable.

• • •

```
goto 23188
```

```
23186 continue
```

• • •

If I'm here I branched out of the loop and 1 keeps its value.

• • •

```
23188 continue
```

Arrays

Arrays of any type can be formed in Fortran.

The syntax is simple:

```
type name(dim, dim, ...)
```

/you have to know how big the array/vector will be when you define the array (write the program)!/

(Static, not dynamic, memory allocation. But - F90/95 allow dynamic memory allocation.)

```
real          sta_rmsn(MAXSTA)
real          tmp_ttp(MAXSTA,MAXEVE)
example usages:
dt_dt(1) = (tmp_ttp(i,j)-tmp_ttp(i,k))
```

Arrays

Array indices are integers, increment by 1.

No restriction on range of indices.

```
Real X(100)
```

Indices range from 1 to 100 in steps of 1.

```
Real Y(-100:100)
```

Indices range from -100 to 100 in steps of 1.

```
Real Z(-10:10,5)
```

Indices range from -10 to 10 in steps of 1 (first index), and 1 to 5 in steps of 1 (second index).

This is a very powerful feature of Fortran.

It allows one to “map” real coordinates easily into the array.

Say I have a seismogram that goes from 1 second to 12 seconds, sampled at 100 sps (0.01 sec).

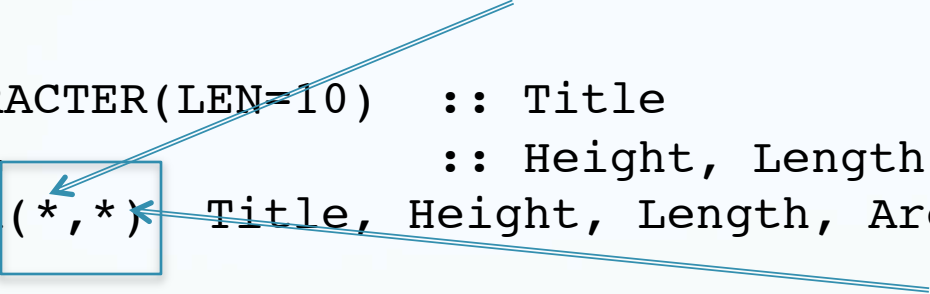
I have 1101 samples. I can define my seismogram array to go from 100 to 1200 and map the index directly into time by multiplying the index value by 0.01 and vice versa.

(in Matlab or C it would be something more complicated.)

Standard I/O

To read in from standard input (first *)

```
CHARACTER(LEN=10)  :: Title  
REAL               :: Height, Length, Area  
read(*,*) Title, Height, Length, Area
```



A blue box highlights the first asterisk in the `read(*,*)` statement. Two blue arrows originate from this box: one points to the `Title` variable in the list of variables, and the other points to the remaining variables `Height, Length, Area`.

Input example is unformatted list processing (second *, tries to put variable type and what finds on input line together "reasonably", works for reads and writes).

Reads a character string and 3 numbers in any format (`ceri 2 3e4, -.02`), separated by whitespace (spaces, tabs) or commas into 4 variables.

I/O from file

To read in from standard input

```
CHARACTER(LEN=10)  :: Title, Height, Length, Area  
...  
read(*,*) Title, Height, Length, Area
```

Input example is unformatted list processing.

If the the variables `Title`, `Height`, `Length`, `Area` are declared as character strings – it reads groups of characters separated by spaces or enclosed in quotes (`first second "third-i third-ii " fourth`).

Formatted I/O with "inline" format statement

```
write (*, '( "# lines = ",i7," in file ",a)') ncts, filename
```

Output example is formatted.

It prints out the string in variable `filename` in double quotes then a 7 character integer (no decimal point) whose value comes from `ncts`, and the `filename` (uses the length of the character string, first byte of Fortran character string has length)

The single quotes define the complete format specification.

think of `write` as `nawk/C printf` with a different syntax.

Can also specify format in its own statement,
identified with line number

This is useful as it allows more than one write
statement to use the same format.

```
write (*,8) "# lines = ",ncts, " in file ",filename  
format(a,i7,a,a)
```

Finally – you can use the unformatted version (Fortran figures out what each variable is – during compile – and handles them with some “intelligent” default or standard).

```
write (*,*) "# lines = ",ncts, " in file ",filename
```

Format codes for Fortran

- a ~ text string
- d ~ double precision numbers, exponent notation
- e ~ real numbers, exponent notation
- f ~ real numbers, fixed point format
- g ~ "reasonably" selects between e and f.
- i ~ integer
- x ~ horizontal skip (space)
- / ~ vertical skip (newline)

Specifying format

The syntax is: `[n]FC[width-1].[width-2]`

`n` - optional repeat count specifying number times format code should be processed. If not specified, a repeat count of 1 is used.

`FC` - is the format code. One of `a, d, e, f, g, i, x, /`.

Specifying format

The syntax is: `[n]FC[width-1].[width-2]`

`width-1, -2` – optional width specifications, default values are format-code specific, look in documentation for the format code.

If output does not fit in `width-1`, prints `width-1` number of `*`.

Specifying format

The syntax is: `[n]FC[width-1].[width-2]`

`width-1` is total number characters (including signs, decimal points, exponential symbol – e.

`width-2` is number digits after decimal point for floating point numbers or number digits to display, with zero rather than blank padding, for integers. Not applicable for `a`, `x`, `/`.

Dynamic format specification

For compilers that support DEC extensions to Fortran (Sun, Absoft on Mac).

$f\langle m \rangle . \langle n \rangle$

Where m and n are variables.

```
f=1.234567
```

```
m=5
```

```
n=3
```

```
write(*,8)f
```

```
format(f<m>.<n>)
```

8

Prints out

1.234

I/O to other than standard I/O

Use unit numbers (or modern name - file handles)
to work with external files

c--- open log file for writing:

```
call freeunit(log) !sets file handle (gets free unit #)
open(unit=log,file='hypoDD.log',status='unknown')
str1= 'starting hypoDD (v1.0 - 03/2001)...'
call datetime(dattim) !calls a subroutine
write(unit=log,'(a45,a)') str1, dattim !formatted i/o
```

Assigns some unused number to variable “log”
associated with a file specified in the open
statement.

Use “log” to identify file from which to do
reads and writes.

c--- open log file for writing:

```
      call freeunit(unit=log) !sets file handle (gets free, as in
c                               unused, unit #)
      open(unit=log,file='hypoDD.log',status='unknown')
      str1= 'starting hypoDD (v1.0 - 03/2001)...'
      call datetime(dattim)    !calls a subroutine
      write(unit=log,'(a45,a)') str1, dattim !formatted i/o
```

See Fortran documentation for other parameters
in open statement.

Since UNIX only supports flat files, most of the
options for the open statement are not applicable
under UNIX.

Predefined units

0 and 102 – standard error

5 and 100 – teletype (standard in)

6 and 101 – line printer!! (standard out)

n without an open (that defines a file name) looks
for file “fort.n”

Be careful with, and while mixing, free format character input with numeric input.

Obtaining length (in characters) of input line

```
read(*, '(q,a)', end=1) len, buf
```

q format specifier returns length of input line, in this case going into variable len.

Internal read

```
read(*, '(q,a)', end=1) len, buf  
read(buf, *) lat2, lon2
```

First read reads from the standard in into a variable `buf`. Second read, reads from the variable `buf` instead of a file.

This allows you to read in an input line without knowing what kinds of information/variables it has and process it depending on what you find.

Internal read

```
character*150 inbuf
```

```
...
```

```
  read(*, '(q,a)', end=1) len, inbuf  
  read(inbuf(11:14), '(a4)') code  
(cnt)
```

Reads 4 characters (a4) from variable `inbuf`, columns/positions 11 through 14, instead of starting at the beginning of the character string.

This way of accessing columns/positions in a character string is general for using character strings anywhere in a program.

Checking for file existence.
Look up `inquire` to see other things you can
ask the OS.

```
inquire(FILE= fn_inp,exist=ex)  
if(.not. ex) stop' >>>ERROR OPENING INPUT FILE.'
```

```
c  read input control parameters
    open(unit=01,file='CNTL',status='old',form='formatted',read
only)
```

```
    call input1      !this subroutine actually reads the file
```

```
    subroutine input1
    implicit none
    integer countrecords
    . . .
```

C this routine reads in control parameters, number of eq's
C and also counts them

Do while loop.

```
    . . .
    countrecords=0
    do while (.true.)
    read(1,*,err=999,end=998) neqs,nsht,nbls,wtsht,kout
    countrecords=countrecords+1
    read(1,*) nitloc,wtsp,eigtol,rmscut,zmin,dxmax,rderr
    read(1,*) hitct,dvpmx,dvsmx,idmp,(vdamp(j),j=1,3),stepl
    end do
```

998 continue processing

999 handle error

```
    ...
    return !alternately you can end using stop or exit
```

Uses `end=` and `err=` features of `read`.

Use `end=` together with semi-infinite loop to read file of unknown length, on EOF go to line #.

Can "handle" error using `err=`, instead of letting program fall over go to line #.

```
...  
countrecords=0  
do while (.true.)  
  read(1,*,err=999,end=998) neqs,nsht,nbls,wtsh,kout  
  countrecords=countrecords+1  
  read(1,*) nitloc,wtsp,eigtol,rmscut,zmin,dxmax,rderr  
  read(1,*) hitct,dvpmx,dvsmx,idmp,(vdamp(j),j=1,3),step1  
end do
```

The diagram illustrates a loop structure. A blue line starts from the left of line 998, goes up, then right, then down to line 999, indicating a jump. Another blue line starts from the right of line 998, goes up, then left, then down to line 999, indicating a jump. A blue box highlights the condition `(.true.)` in the `do while` statement. A blue box highlights the `err=999,end=998` part of the `read` statement. A blue box highlights the `end do` statement.

998 continue processing

999 handle error

...

return !alternately you can end using `stop` or `exit`

Subroutines – little programs, but not independent. Use for stuff you do lots and for organization.

```
subroutine latlon(x,y,lat,xlat,lon,xlon)
c  convert from Cartesian coord to lat and long.
c  Takes x,y and returns lat,xlat,lon, and xlon
      common /shortd/ xltkm,xlnkm,rota,nzco,xlt,xln,snr,csr
      rad=1.7453292e-2
      rlt=9.9330647e-1
      fy=csr*y-snr*x
      fx=snr*y+csr*x
      fy=fy/xltkm
      plt=xlt+fy
      xlt1=atan(rlt*tan(rad*(plt+xlt)/120.))
      fx=fx/(xlnkm*cos(xlt1))
      pln=xln+fx
      lat=plt/60.
      xlat=plt-lat*60.
      lon=pln/60.
      xlon=pln-lon*60.
      return
end
```


When defining arrays in a subroutine you will not in general know how big they are when you are writing the program. Subroutine can get input parameters from anywhere.

```
function findsr  
  instr, instrlen  
  1, seastr, seastrlen)  
  implicit none  
  character *(*) instr, seastr
```

use (*) to tell Fortran that it is an array.

A word on array usage

Fortran needs to know how big the arrays are to assign memory, but after that it forgets this information (unless your compiler has a flag that does array bounds checking –but this slows the program down).

You are responsible for making sure your array indexing does not go outside the array limits. (this is true for all but "training-wheels" languages designed to teach programming and computer science approved programming techniques.)

A word on array usage

So if your array is 100 long and you say

```
myarray(1000)=10
```

Fortran will happily put 10 in the 1000th memory position from the start of the array.

If this position in memory is some other variable/
data you will clobber it.

If it is "code" of your program it will cause the program to fall over (OS may notice this and die gracefully with an error message)

A word on array usage

So if your array is 100 long and you say

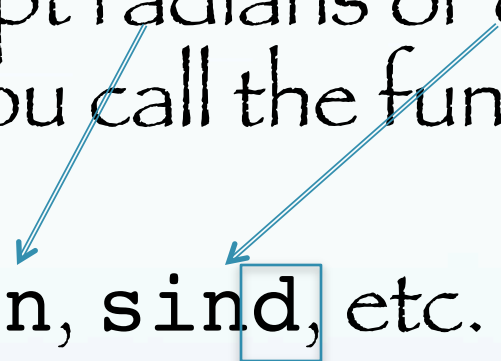
```
myarray(1000)=10
```

If this position in memory is outside your program (and therefore in somebody else's area) the OS will notice this and die gracefully with an error message

Intrinsic Fortran Functions

Mathematical functions (`sqrt`, `sin`, `cos`, `tan`, etc) accept `REAL` types and return `REAL` types.

Trig functions accept radians or degrees - based on how you call the function -



`sin`, `sind`, etc.

`abs` (absolute value) will also accept `INTEGER`s.

Intrinsic Fortran Functions

Conversion functions (90/95 conventions)

`INT(x)` integer part `x`, REAL2INTEGER

`NINT(x)` nearest integer to `x`, REAL2INTEGER

`FLOOR(x)` greatest integer less than or equal to `x`,
REAL2INTEGER

`FRACTION(x)` the fractional part of `x`,
REAL2REAL

`REAL(x)` convert `x` to REAL, INTEGER2REAL

Functions – little programs, but not independent.
Work like built-in functions sin, cos, etc..

In calling routine have to declare it

```
real mom_mag
```

To use it in calling routine

```
magm=mom_mag(mt)
```


Functions – little programs, but not independent.
Work like built-in functions \sin , \cos , etc..

To define function

```
function mom_mag(mt)
implicit none
real mom_mag
real mt(6)
```

...

```
return
end
```

How to turn your Fortran source files into a program that runs/executes.

```
f77 -W132 -lU77 vel_az_apkim2000.f  
subs.f -o vel_az_apkim2000
```

Output – executable file will be named
vel_az_apkim2000.

If you don't put to fields "-o name" the output
file is named a.out.

How to turn your Fortran source files into a program that runs/executes.

```
f77 -W132 -lU77 vel_az_apkim2000.f  
subs.f -o vel_az_apkim2000
```

The `-W` switch lets your input lines go out to 132 characters.

The `-lU77` switch gives Sun F77 (actually Sun's continuation of DEC extensions).

(These switches are for Absoft Fortran on the Mac. In general they are different for different compilers. Most compilers do not have the DEC extensions.)