Common Languages used in Scientific programming

What is the best language to learn?

That depends on what you want to do.

Most common for scientific programming (in no particular order)

Fortran
C
C++
Matlab
Pearl

High School/Jr.High

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

Prints “HELLO WORLD”

First year in College

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

Prints "HELLO WORLD"

Senior year in College

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

Prints "HELLO WORLD"

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"

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"

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(ResultFromScode(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 <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "pshlo.h"
#include "shlo.hxx"
#include "clsid.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
    wcscpy(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”

FORTRAN

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

FORTRAN (FORmula TRANslator) is a high-level language.

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 compiler and yields an executable specific to that platform

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

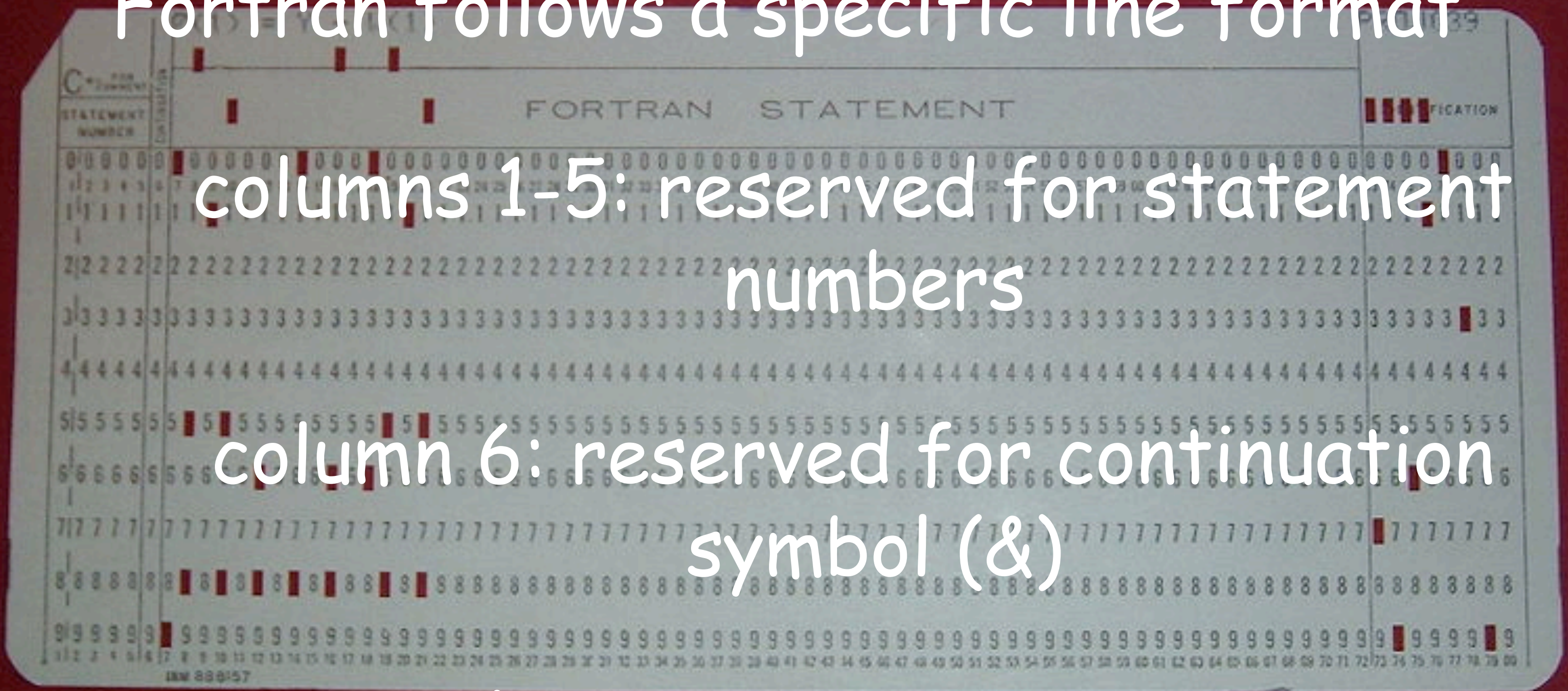
Basics of Fortran

Simple programs have the following structure -

Comments
Common block inclusions
Variable declarations
Program

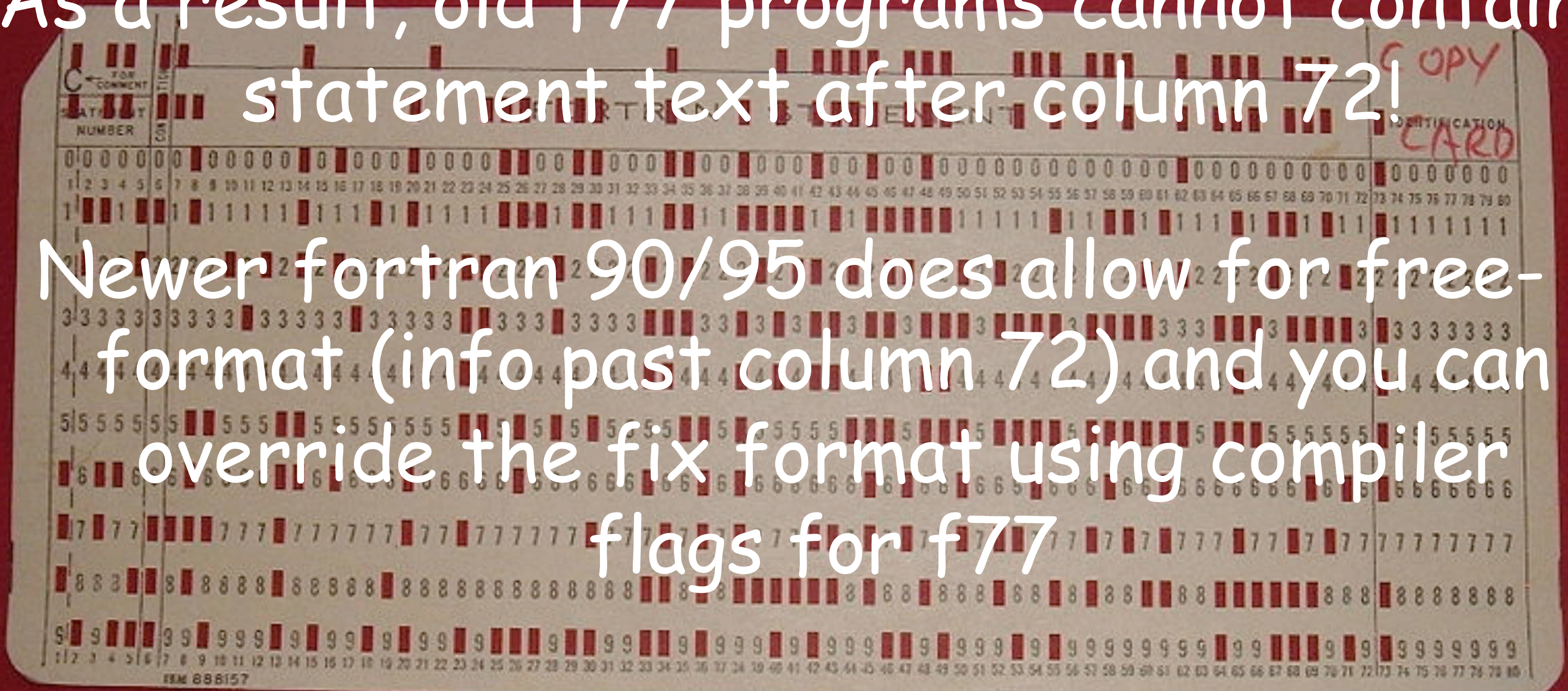
Column formatting

Fortran follows a specific line format



- 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 fix format using compiler flags for f77

The first line of segment of fortran source code (a program) (in a file) indicates what it is

```
 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 "&" indicates a line continuation

The last line of the segment (a program) needs to indicate the segment (program) is finished

```
end
```

Variable typing - Implicit

```
IMPLICIT NONE
```

not standard (so you should not use it!), but very useful

(all rules are made to be broken!!).

Gives the "Pascal convention" that all variables have to be specified.

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

Variable typing - Implicit

IMPLICIT - the default is

```
IMPLICIT REAL(A-H,O-Z),INTEGER(I-N)
```

And you can specify whatever you want

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

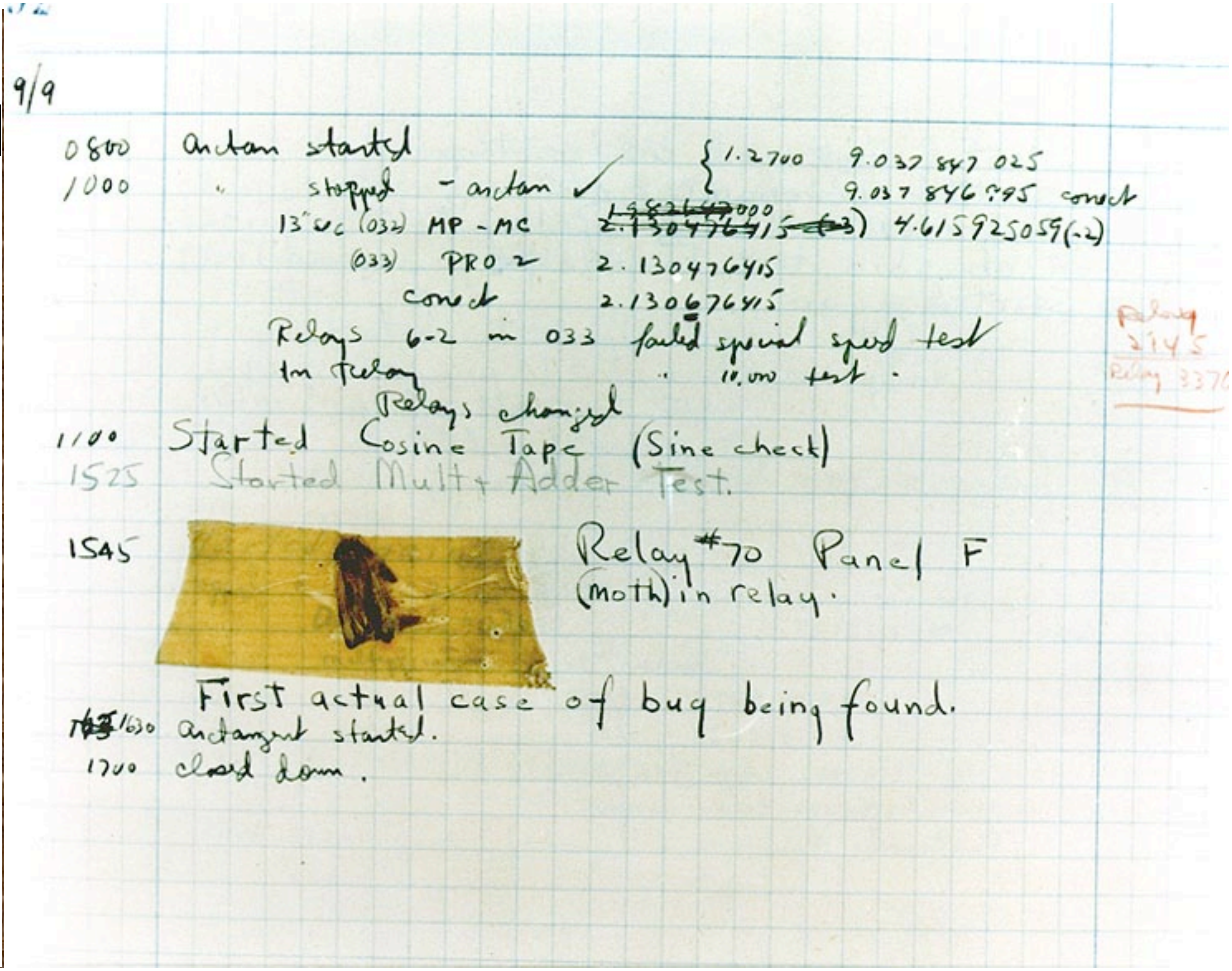
The huge benefit of IMPLICIT NONE is that it will catch most of your typing errors.

Without it, new variables are created as they show up in your source code.

So a typo makes a new variable.

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 ✓ {1.2700 9.037 847 025
9.037 846 995 correct
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ ~~(033)~~) 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test.
Relays changed
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~~ 1630 Antangent started.
1700 closed down.

Relay 2145
Relay 3370

Comments

A "C" or "c" (fortran is case insensetive!) in column 1 is used to indicate the "line"/statement 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
```

Variables

Variables do not need to be declared in Fortran

But should be unless you like debugging.

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)

But "the process of writing programs and getting them to work was excruicating difficult." (Sci. Am., Dec. 2009)

Wilkes
Memoirs

"the realization came over me with full force that a goo part of the remainder of my life was going to be spent finding errors in my own programs" 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) !integer vector declaration
double    at_idx(MAXDATA) !double precision vector
real      acond  !single precision scalar declaration
character dt_sta(MAXDATA)*7 !string with length
```

or

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

Variables must be declared at the beginning of your program.

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 character variables “A” is not equal to “a”.

There is no special syntax ($, @, etc.) for using 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).

```
      minwght= 0.00001
      rms_ccold= 0
      rms_ctold= 0
      rms_cc0old= 0
      rms_ct0old=  0
c--- get input parameter file name:
nargument s = iargc()  !similar to argc in C, counts number of
   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
```

You can put blank lines, tabs, spaces as you like for readability (except at beginning – first 5 characters for statement number, 6th for continuation --- can use tab with digit 1-9 for continuation immediately after the tab.).

Global Variables/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 edit out the comma from the previous line also.)

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 sub1 (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.

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 sub1 (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/"

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

Operators

Type	Operator	Associativity
Arithmetic		
	**	right to left
	* & /	left to right
	+ & -	left to right
Relational		
	lt (<) le (<=)	none
	gt (>) ge (>=) !	
	"()" indicate 90/95 convention	
	eq (==) ne (/=)	
	! is negation	

Intrinsic Fortran Functions

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

All trig functions use radian or degrees.

sin, sind, etc.

abs (absolute value) will also accept INTEGERs.

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

if/else if/else/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
```

goto/go to

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.

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
```

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 or do/continue aka the “do loop”

Two forms

1 - block form (do-enddo)

```
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.

2 - statement number form

(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
```

What is the value of loop counter (l in this case) when I leave the loop? (can I depend on its 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 l is undefined (we cannot be sure its value is j1). Solution save l into another variable.

```
      . . .
      goto 23188
23186 continue
      . . .
```

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

```
      . . .
23188 continue
```

Arrays

Arrays of any type can be formed in Fortran.

The syntax is simple:

```
type name(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(l) = (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), and 1 to 5 in steps of 1 (second).

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
```

Input example is unformatted (second *).

If the the variables `Title, Height, Length, Area` are declared as numbers, it reads 4 numbers in any format (1 1.1 1.3e2 . 1) , separated by spaces, commas, or tabs into them.

I/O from file

To read in from standard input

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

Input example is <u>unformatted</u>.

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 and fourth” fifth).

Formatted I/O

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

Output example is formatted.

It prints out the string 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 printf with a different syntax.

Get same results from.

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

Can also specify format in its own statement

(useful when more than one write statement uses same format).

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

and similar results from unformatted version.

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

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(log,file='hypoDD.log',status='unknown')
      str1= 'starting hypoDD (v1.0 - 03/2001)...'
      call datetime(dattim)    !calls a subroutine
      write(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 do reads and writes from that file.

```
c--- open log file for writing:

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

unit1 associated with file somewhere else (previously) in code.

```
read(unit1,*)  i, a  !free format for integers and reals
```

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

Checking for file existance.

```
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
      . . .
      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
```

Do while loop.

Predefined units

0 and 102 – standard error
5 and 100 – teletype (standard in)
6 and 101 – line printer!! (standard out)

n without an open looks for file “fort.n”

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
```

C & C++

C and C++ are higher-level languages that are designed to be independent of computational platform (as is Fortran, COBOL, ALGOL, PL/1, APL,... - and all pretty much dismal failures at it.).

Higher-level languages must be translated into the low-level machine language in order to run (same as is Fortran, COBOL, ALGOL, PL/1, APL,...).

This is done via compiler and yields an executable specific to that platform.

Differences between C & C++

C++ grew out of C and is mostly a superset of the latter, but it is considered a different language

They are not developed to be cross-compatible and C++ does not supersede the use of C

Differences between C & C++

C++ introduces many features that are not available in C and in practice almost all code written in C++ is not valid C code

There are many C syntaxes which are invalid or behave differently in C++

This is all we are going to say about C++

(see the master programmer example for why).

Basics of C

Simple C programs have the following structure

Comments
Library inclusions
Main Program

C program source file names MUST end in .c (.cpp for C++)

Lecture based largely on : http://www.physics.drexel.edu/students/courses/Comp_Phys/General/C_basics/

```
Comment blocks
/*   …. */    : Used to enclose comments
/*
* File: hello.c
* ---------------
* This program prints the message “Hello, world.”
*/
```

To make turning comment on/off easily use

Commented out

```
/* i++; /* */
```

not commented out

```
i++; /* */
```

Libraries

Libraries are collections of tools (subroutines/functions) that perform specific operations.

They are not part of the basic language. (they may even be written in another language).

As part of the UNIX philosophy (remember the power of unix) C does not include

I/O (basic or otherwise)

math (beyond what is in the CPU as an instruction: +, -, *, /, and, or, ex-or, not, shift).

(and they got away with it!)

Writing I/O routines, math (exponentiation for example) are left to the user to write as they see fit/need.

Lucky for us - somebody has developed some of these things

(but we are now relinquishing the power of unix to them).

Since C is so stripped down - libraries are much more important to C than previous languages we have seen/used.

You have to declare at least the stdlib.h for a program to compile (not really, but it is a good idea).

```
#include  <stdlib.h>      the standard general purpose library
#include  <stdio.h>     the standard input/output library
#include <math.h>       the standard math library
#include “hrdfavorites.h”     a personal extended library
```

The other two libraries above you almost always need are the I/O library, stdio.h, and the math library, math.h.

```
#include  <stdlib.h>      the standard general purpose library
#include  <stdio.h>      the standard input/output library
#include <math.h>        the standard math library
#include “hrdfavorites.h”     a personal extended library
```

The final library is some thing you wrote.

Notice the filenames all end in .h

Notice the ones that come with C are in <>,
while local ones are in “”.

Main Program
This block contains the program itself

```
void main()
{
        printf("Hello.\n");
}
```

Officially, we are defining a function called main with the body of the function contained in {}

Variables

Variables need to be declared in C/C++ !!!

numeric variable types include:

int: integers

short: short integers

long: long integers (more memory)

float: single-precision real floating point number

double: double-precision real floating point (more precision but also more memory)

string variable types include

Declaring variables
Here are some examples of variable declarations

```
main()
{
    int a,b,c;
    double dd,ee,ff;

}
```

Variables mustbe declared at the beginning of your program/function.

Prototyping

Declaration on steroids.

Not only do we have to define all the variables in C, we must also define what each function returns and its list of arguments.

```
void – returns nothing
int – returns integer
float – returns float
etc.
```

If you forget to type the funcitons, int is assumed and the compiler will complain.

```
void main(int argc, char *argv[])
```

Main does not return anything and takes two input/calling arguments, an integer and a pointer to a character array.

One has to look up what the input/calling arguments are (the integer has the number of command line arguments, and the pointer to the character array has the address of the beginning of the character string for each argument).

In a function you write, you decide what to pass in and out.

All this extra typing is supposed to help the compiler make sure your code is consistent (very un-unix like - trying to help the user).

There is no special syntax ($, @) for using a variable once it has been declared.

```
#include < stdio.h>
#include < math.h>
main()
{
        int     angle_degree;
        double angle_radian, pi, value;
        printf ("\nCompute a table of the sine function\n\n");
   /* obtain pi once for all */
   /* or just use pi = M_PI, where   M_PI is defined in math.h          */
   pi = 4.0*atan(1.0);
   printf ( " Value of PI = %f \n\n", pi );
        printf ( " angle      Sine \n" );
        angle_degree=0;                          /* initial angle value */

        while (  angle_degree <= 360 ) {   /* loop until angle_degree > 360 */
                  angle_radian = pi * angle_degree/180.0 ;
                  value = sin(angle_radian);
                 printf ( " %3d       %f \n ", angle_degree, value );
                  angle_degree = angle_degree + 10;       /* increment the loop
   index          */
         }
}
```

Floats/doubles are relatively easy to use but problems tend to occur when performing division.

An int divided by an int returns an int.
An int divided by a float returns a float.
A float divided by an int returns a float.
A float divided by a float returns a float.

As an example, 3 is considered as an int, but 3.0 is considered as a float.

If you want to store the result of a division as a floating-point (decimal) number, make sure you store it in a float declared variable.

Explicit conversion
you can specify explicit conversion by using a <u>type cast</u>

```
int num, den;
double quotient;

quotient = num / (double) den;  /*this recasts den as a
double so the value of an int/double is a double.
```

Global Constants

You can define constants of any type by using the #define compiler directive. Its syntax is simple--for instance

```
#define ANGLE_MIN 0
#define ANGLE_MAX 360
```

C distinguishes between lowercase and uppercase letters in variable names. It is customary to use capital letters in defining global constants.

These are traditional declared after the #include calls

Loops
C is the original looping language…love it or hate it

Statement blocks, or a sequences of statements, are encased using { }.

Statements are executed in sequence from first to last by default

(have not mentioned so far, but, statements in C are terminated by ";". They wrap lines, unlike fortran.).

```
{
    first_statement;
    last_statement;
}
```

While

while: continues to loop as long as condition exited successfully

```
count = 0;
 while (count < 10) {
        count += 2;
        printf ("count is now %d\n”,count);
  }
```

There is no print, there is printf (print to file) and prints (print to string).

You have to initialize numeric variables to 0 to avoid getting whatever happens to be sitting in that location in memory.

if/else if/else

If expression is true, then run the first set of commands. Else if a second expression is true, run the second set of commands. Else if neither is true, run a third set of commands. End the if command

```
if ( a > b) {
        *statement*
 } else if (a == b) {
     statement
 } else {
        printf "%d is less than %d.\n", a, b;
 }
```

Conditional Operators

Conditionals are logical operations involving comparison of quantities (of the same type) using the conditional operators:

<	greater than
<=	greater than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

Conditional Operators

and the boolean operators

&&	and
||	or
!	not

For

one of the most common loop structures is the for loop, which iterates over an array of objects

for i values in array, do this

```
for  (i=0; i<=10; i++ ) {
    for (j=0; j<=10; j++) {
        H[i][j]=0;
    }
}
```

Switch

The appropriate block of statements is executed according to the value of the expression, compared with the constant expressions in the case statement.

This construct is particularly useful in handling input variables.

```
 switch (n) {
         case 1:  printf("Ace\n"); break;
         case 11: {          /*there is some flexibility in
syntax*/
                    printf("Jack\n");
                           break;
         }
          …..
         default: printf ("%d\n",n); break;
   }
```

break

<u>break</u>: allows you to break out of a for, do, while, or switch loop

Default behavior is the break out of the enclosing loop

```
for ( a=0; a<20; a++ ) {
      if ( a > 10) {
         break;
      }
}
    ## last comes here ##
```

Arrays

Arrays of any type can be formed in C. The syntax is simple:

```
    type name[dim];
     double name[100][50];
/*you have to already know how big the array/vector
will be!*/
```

In C, arrays starts at position 0.

The elements of the array occupy adjacent locations in memory.

Pointers

The C language allows the programmer to ``peek and poke'' directly into memory locations.

This gives great flexibility and power to the language, but it also one of the great hurdles that the beginner must overcome in using the language.

variables called pointers store the address of other variables.

Pointers

Have to declare them, they are a special kind of integer.

```
int *p;   /*declared that p is a pointer*/
```

&x returns address of x, which can be stored in a variable.

If that variable is a pointer, we can then use it to access the memory contents at that address.

```
p=&x;   /*p is the address of x*/
```

Pointers

Value of pointer is the address in memory. Value of what is in that address obtained using *.

```
x=17;

p=&x;   /*p is the address of x*/
*p = 17;   /* same as setting x = 17 */

y=x;
y=*p;
```

Pointers are used to pass arrays to functions. (C always passes arguments to functions by value [a copy], except when it does not [arrays]. Fortran passes by address)

Strings

You have to think of strings as character vectors (much like matlab)

Strings are manipulated either via pointers or via special routines available from the standard string library string.h

(basic C does almost nothing!).

C strings are null terminated (start at address of string and to till encounter a null [zero] byte).

```
#include <string.h>  to work efficiently with strings

      char  string[20];
      char  message[] = "Hello, world.";
```

```
main()
{
     char text_2[100];
     char *ta, *tb;
     int i;

/* set message to be an arrray  of characters; initialize it
*to the constant string "...” and let the compiler decide its size by using []
*/
     char message[] = "Hello, I am a string; what are you?”;

     printf("Original message: %s\n", message);

     /* use explicit pointer arithmetic to copy the original message to text_2
   */
     ta=message;
     tb=text_2;
     while ( ( *tb++ = *ta++ ) != '\0' ) { ; }   /*set the pointers equal at
   each element until FALSE (aka ! 0) */
     printf("Text_2: %s\n", text_2);

}
```

Higher-Level I/O
To read in from external files

```
main(int argc, char *argv) {
              const char *progname = argv[0];
       if (argc==5) {   /*argc = number command line files
       listed*/
                 sscanf(argv[1], "%s", cfile);  /*argv stores
       the files/values*/
                 sscanf(argv[2], "%s", sfile);
                 sscanf(argv[3], "%d", &winlen);
                 sscanf(argv[4], "%f", &thresh);
       }

       fl=fopen("outdesc","w");
       fc=fopen(cfile,"r");
```

Here, fl and fc are file handles. If you include stdio.h, you would declare them as

FILE *fl, *fc;

The if block is an example of reading the command line input parameters (not a file). Uses sscanf (read from string) rather than fscanf (read from file) [fortran also does this - by simply placing the character string you want to read into the read statement in place of the unit number in the read statement. It is known as an “internal” read.].

```
main(int argc, char *argv) {
              const char *progname = argv[0];
       if (argc==5) {   /*argc = number command line files
       listed*/
                 sscanf(argv[1], "%s", cfile);   /*argv stores
       the files/values*/
                 sscanf(argv[2], "%s", sfile);
                 sscanf(argv[3], "%d", &winlen);
                 sscanf(argv[4], "%f", &thresh);
       }

       fl=fopen("outdesc","w");
       fc=fopen(cfile,"r");
```

```
#include < stdio.h>

void main()
{
    FILE *fp;
    int i;

    fp = fopen("foo.dat", "w");           /* open foo.dat for
  writing */

    fprintf(fp, "\nSample Code\n\n");   /* write some info
  */
    for (i = 1; i <= 10 ; i++)
  fprintf(fp, "i = %d\n", i);

    fclose(fp);                          /* close the file
  */
}
```

Subroutines (called functions in C) [fortran has both subroutines and functions - the difference being that a function returns a value "y=sin(x)" for example, versus "call sin(angle,value)"]

A function has the following layout:

```
return-type function-name ( argument-list-if-necessary )
{
    ...local-declarations…
    ...statements…
    return return-value;
}
```

If return-type is omitted, C defaults to int.

```
int n_char(char string[])
{
    int n;    /* local variable in this function    */

    /* strlen(a) returns the length of  string a        */
    /* defined via the string.h header          */
     n = strlen(string);
     if (n > 50)
    printf("String is longer than 50 characters\n");

     return n;    /* return the value of integer n  */
}
```

COMPILING

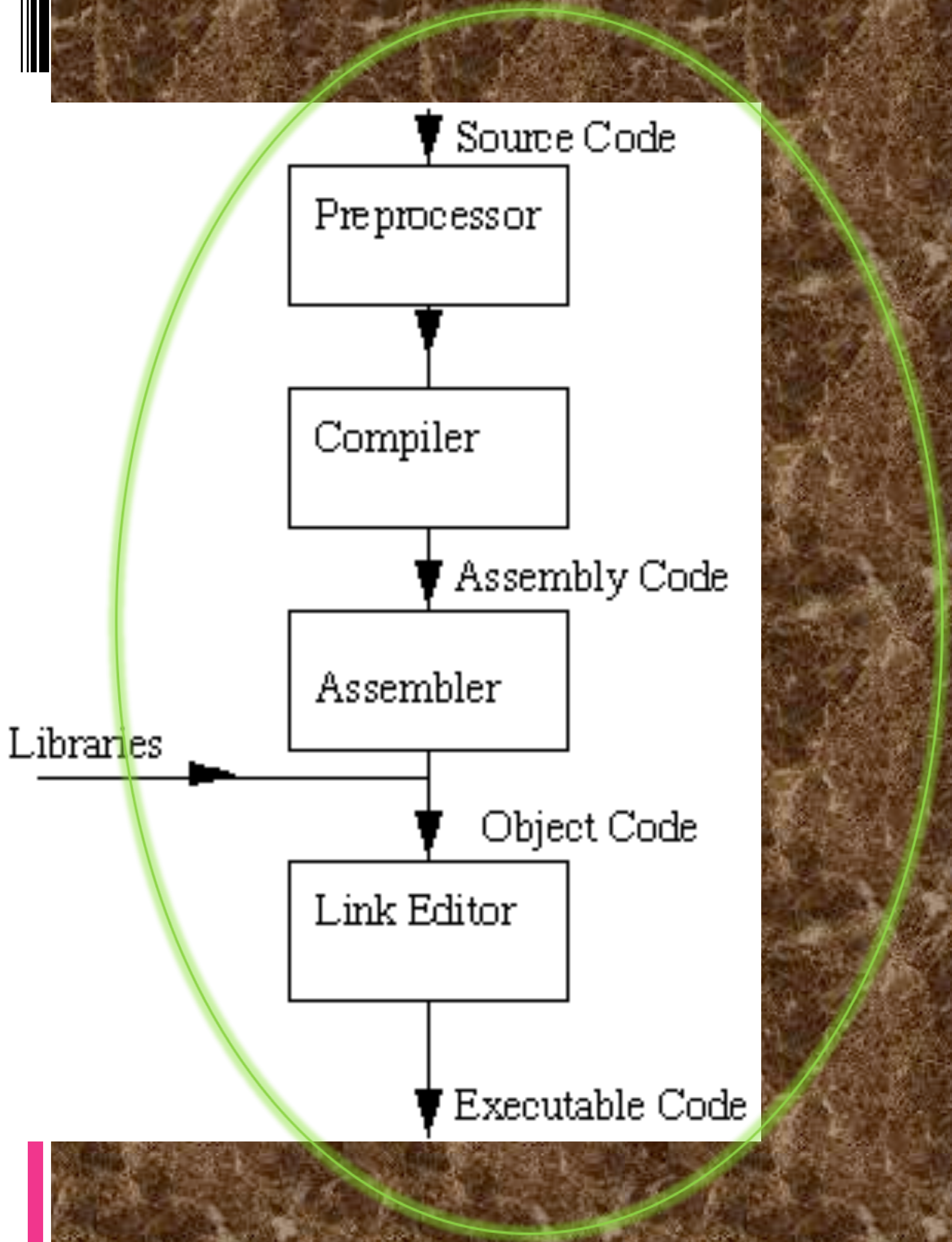
Compiling

Your C or Fortran program won't work unless you compile (and link) it

The compiler will convert your program to machine code and the linker (called automatically) will build your program (connects it to all those i/o, math, etc. library functions) as an executable file (typically in the current directory), which you can then invoke and run just like any other command.

C and Fortran are compiled using different compilers

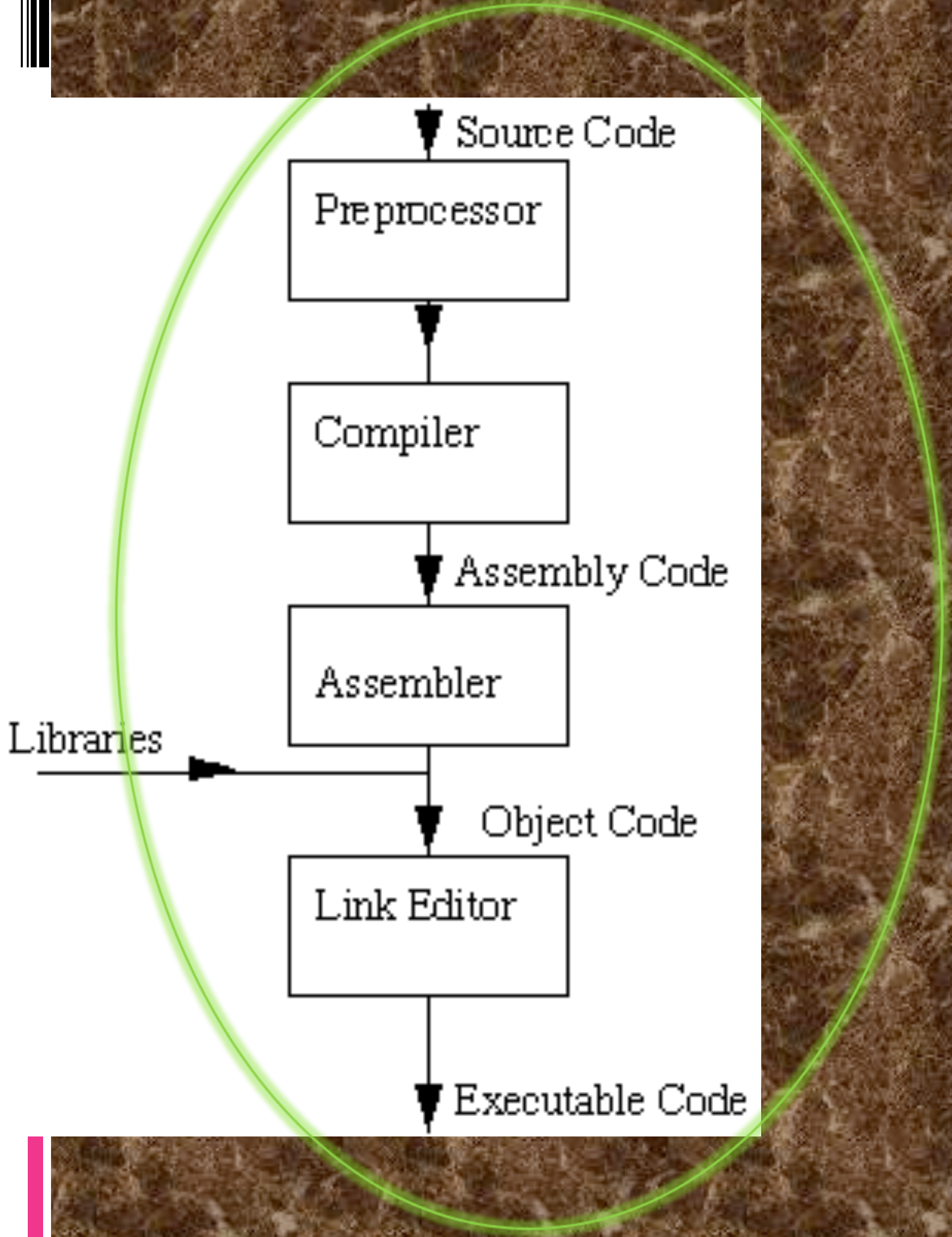
“the compiler”



The preprocessor accepts source code as input and is responsible for removing comments interpreting special preprocessor directives

The compiler translates source to assembly code.

"the compiler"



The assembler creates object code.

If a source file references library functions or functions defined in other source files the link editor combines these functions to create an executable file.

C compilers

One extremely popular Unix compiler, which happens to be of extremely high quality, and also happens to be free, is the Free Software Foundations's gcc, or GNU C Compiler.

at CERI:

```
%which gcc
    /opt/local/bin/gcc

%gcc –v
    gcc version 3.4.2
```

Another C compiler available at CERI is the SUN distribution cc

```
/opt/Studio/SUNWspro/bin/cc
```

There are differences, beyond the scope of this class, but in general gcc is a good option (both come with Mac developer tools)

C++ compilers

The GNU compiler for C++ is g++

The SUN compiler for C++ is CC (versus cc for regular C)

At the level of this class, they will work the same as gcc and cc, but they have a different set of flags.

Simple example

```
%gcc -o hello hello.c
```

hello.c : text file with C program
hello : executable file

The -o hello part says that the output, the executable program which the compiler will build, should be named “hello”

if you leave out the “-o hello” part, the default is usually to leave your executable program in a file named a.out (which will get overwritten the next time you do compile something without the –o part)

Example with math, need math library.

If you're compiling a program which uses any of the math functions declared in the header file <math.h>, you'll typically have to request explicitly that the compiler (actually linker) include the math library:

```
% gcc -o myprogram myprogram.c -lm
```

Notice that the -lm option which requests the math library must be placed after all the source code elements.

```
% gcc myprogram.c -lm-o myprogram
```

Also works.

Finding out library information requires a trip to the local unix wizard.

It is poorly documented.

It is non standard (each power user does their own - the power of unix).

It varies between machines.

Some Useful Compiler Options (switches)

-g : invoke debugging option. This instructs the compiler to produce additional symbol table information that is used by a variety of debugging utilities.

-llibrary : Link with object libraries. This option must follow the source file arguments. The object libraries are archived and can be standard, third party or user created libraries

-c : Suppress the linking process and produce a .o file for each source file listed. Several can be subsequently linked by the cc command, for example:

```
cc file1.o file2.o ...... -o executable
```

-Ipathname : Add pathname to the list of directories in which to search for #include files with relative filenames (not beginning with slash /). By default, the preprocessor first searches for #include files in the directory containing source file, then in directories named with -I options (if any), and finally, in /usr/include.

-Olevel : performs some optimization of the executable and can lead to significant increases in execution speed. Example

```
gcc -o hello hello.c —O2
```

But oftentimes optimization only increases the speed at which it is doing something incorrectly.

Fortran compilers
The GNU project also supplies Fortran compilers

at CERI:

```
%which g77
    /opt/local/bin/g77

%g77  –v

    gcc version 3.4.2
```

!this is not a typo. gcc comes with Fortran 77 compilers. However, on the Mac, g77 has some problems with some codes. Always check for platform dependence.

Another Fortran compiler available at CERI is the SUN distribution

```
/opt/Studio/SUNWspro/bin/f77
/opt/Studio/SUNWspro/bin/f90
/opt/Studio/SUNWspro/bin/f95
```

File names ending in .f90 and .f95 are assumed to be free source form - suitable for Fortran 90/95 compilation.

File names ending in .f and .for are assumed to be assumed fixed form source - compatible with old Fortran 77 compilation.

Simple example

```
%g77 hello.f -o hello

hello.f :  text file with Fortran 77
hello : executable file
```

The -o hello part says that the output, the executable program which the compiler will build, should be named hello

if you leave out the -o hello part, the default is usually to leave your executable program in a file named a.out

Example with include files

The path of include files can be given with the -I option, for example:

```
g77 myprog.f -o myprog -I/home/fred/fortran/inc
```

or

```
g77 myprog.f -o myprog -I$MYINC
```

where the environment variable MYINC is set with:

```
MYINC=/home/hdeshon/fortran/inc/
```

Some Useful Compiler Options

-Olevel : performs some optimization of the executable and can lead to significant increases in execution speed. Example:

```
g77 myprog.f -o myprog -O2
```

-Wlevel : enables most warning messages that can be switched on by the programmer. Such messages are generated at compile-time warning the programmer of, for example, unused or unset variables. Example:

```
g77 myprog.f -o myprog -O2 -Wall
```

Various run-time options can be selected, these options cause extra code to be added to the executable and so can cause significant decreases in execution speed.

However these options can be very useful during program development and debugging.

Example

```
g77 myprog.f90 -o myprog -O2 -fbounds-check
```

This causes the executable to check for "array index out of bounds conditions" (and slows your code way down).

Recommended options

```
g77 myprog.f -o myprog -Wuninitialized -Wimplicit-none -
Wunused-vars -Wunset-vars -fbounds-check
         -ftrace=full -O2
```

If speed of execution is important then the following options will improve speed:

```
g77 myprog.f -o myprog -Wuninitialized -Wimplicit-none -
Wunused-vars -Wunset-vars -O2
```

Compiling subprogram source files.

It is sometimes useful to place sub-programs into separate source files especially if the sub-programs are large or shared with other programs or programmers.

If a Fortran project contains more than one program source file, then to compile all source files to an executable program you can use the following command:

```
g77 main.f sub1.f sub2.f sub3.f -o myprog
```

You can also build your own libraries

(same idea as with subroutines on last example, but compile and build library once, and then link to to library with the -l switch.)

Makefiles

Makefiles are special format files that together with the make unix utility will help you to automatically build and manage your projects.

make utility

If you run make, this program will look for a file named makefile in your directory, and then execute it.

If you have several makefiles, then you can execute them with the command:

```
make -f MyMakefile
```

Example of a simple makefile
The basic makefile is composed of:

```
target: dependencies
[tab] system command

All:
g++ main.cpp hello.cpp
factorial.cpp -o hello
```

Dependencies

Sometimes is useful to use different targets. This is because if you modify a single file in your project, you don't have to recompile everything, only what modified.

```
all: hello

hello: main.o hello.o
    g++ main.o hello.o -o hello

main.o: main.cpp
    g++ -c main.cpp

hello.o: hello.cpp
    g++ -c hello.cpp

clean:
    rm -rf *o hello
```

Typical example

```
# I am a comment, the variable CC will be the compiler to use.
CC=g++
# Hey!, I'm comment number 2. CFLAGS are options for compiler.
CFLAGS=-c -Wall

all: hello

hello: main.o hello.o
   $(CC) main.o hello.o -o hello

main.o: main.cpp
   $(CC) $(CFLAGS) main.cpp

hello.o: hello.cpp
   $(CC) $(CFLAGS) hello.cpp

clean:
   rm -rf *o hello
```

Combining C and Fortran

```
CMD       = hypoDD
CC        = gcc          #Specified the C compiler
FC        = g77          #Specified the Fortran compiler
SRCS      = $(CMD).f \     #List the main program first…in this
case hypoDD.f
            aprod.f cluster1.f covar.f datum.f \
            delaz.f delaz2.f direct1.f dist.f dtres.f exist.f \
            freeunit.f getdata.f getinp.f ifindi.f \
            indexxi.f juliam.f lsfit_lsqr.f lsfit_svd.f \
            lsqr.f matmult1.f matmult2.f matmult3.f mdian1.f \
            normlz.f partials.f ran.f redist.f refract.f \
            resstat.f scopy.f sdc2.f setorg.f skip.f \
            snrm2.f sort.f sorti.f sscal.f \
            svd.f tiddid.f trialsrc.f trimlen.f \
            ttime.f vmodel.f weighting.f
CSRCS     = atoangle_.c atoangle.c datetime_.c hypot_.c rpad_.c
sscanf3_.c
```

#The underscore is added prior to the .c to indicate that these are C programs to the fortran assembler

```
INCLDIR = ../../include
LDFLAGS = -O

# Flags for GNU g77 compiler
FFLAGS  = -O -I$(INCLDIR) -g -fno-silent -ffixed-line-length-none
   —Wall -implicit

#Flags for the GNU gcc compiler
CFLAGS  = -O -g -I$(INCLDIR)

OBJS    = $(SRCS:%.f=%.o) $(CSRCS:%.c=%.o)

all: $(CMD)          #make all makes hypoDD and all dependencies

$(CMD): $(OBJS)             #To make hypoDD, link all OBJS with
   the fortran comp
        $(FC) $(LDFLAGS) $(OBJS) -o $@

#%.o: %.f            #long version of the shortcut under OBJS
#         $(FC) $(FFLAGS) -c $(@F:.o=.f) -o $@
```

```
CC       = g++
FC       = gcc
CFLAGS   = -g -DDEBUG -Wall
FFLAGS   = -Wall
OBJS1    = bcseis.o \
           sacHeader.o sacSeisgram.o distaz.o readSacData.o \
           mathFuncs.o fourier.o complex.o \
           stas.o evData.o seisData.o tmDelay.o calcTravTm.o \
           getMaxShiftLag.o calcTmDelays.o calcCCTmDelay.o calcSubTmDelay.o calcBSTmDelay.o \
    ttime.o direct1.o refract.o vmodel.o tiddid.o   #These are fortran, the others are c
BIN      = ../../bin
PROG     = bcseis

.c.o:
        ${CC} $(CFLAGS) -c $<
.f.o:
        ${FC} $(FFLAGS) -c $<

all:    ${PROG}

bcseis: ${OBJS1}
        ${CC} ${CFLAGS} -lm -o $@ ${OBJS1}
        mv $@ ${BIN}
```

Web page
Excel/spreadsheets