

# Data Analysis in Geophysics

## ESCI 7205

Class 24

Bob Smalley

Short intro C.

C is a higher-level language that is designed to be independent of computational platform

(as are Fortran, COBOL, ALGOL, PL/I, APL, Matlab, C++, ...  
- 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 Fortran, COBOL, ALGOL, PL/I, APL, C++, ... ).

This is done via compiler and yields an executable program 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).



The standard reference for C is

The C Programming Language

By

Kernighan and Ritchie

Commonly known as "K&R"

Now in its 2<sup>nd</sup> edition covering ANSI C (1988)  
(first edition 1978)

# Basics of C

Simple C programs have the following structure

- Comments (can really be anywhere)
- Library inclusions (have to be defined before using)
- Defines
- Functions (have to be defined before using, so in source code file should come before main program that uses them)
  - Main Program

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

(if you don't like that use the power of UNIX to write your own OS and compiler.)

# Comments

Done with a pair of delimiters `/*` to start, and `*/` to end, comment is enclosed within them. They match up across lines.

```
/*  
* File: hello.c  
* This program prints the message "Hello, world."  
*/
```

To make turning comment on/off easily use

Commented out

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

and not commented out

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

# Libraries

Libraries are collections of tools (subroutines – known as functions in C) that perform specific operations.

(they are also available for all high level languages, HLLs, not just C.)  
(C is most often compared to BASIC, where it "wins" hands-down, rather than "real" languages where the fight is more fair.)

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

As part of the UNIX "lean and mean" 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).

But before moving on to

- Library inclusions
  - Defines
  - Functions
- Main Program

We have to deal with the earlier remark

"(have to be defined before using)"

# Declarations

Variables and functions must be declared in C and C++ !!!

numeric variable types include:

integers

int: integers (usually 4 byte now)

short: short integers (2 byte)

long: long integers (8 byte, more memory)

```
int x;
```



# Declarations

Variables and functions must be declared in C and C++ !!!

numeric variable types include:

## Floating point

`float`: single-precision real floating point number (4 byte)

`double`: double-precision real floating point (8 byte, more precision but also more memory)

# Declarations

Variables and functions must be declared in C and C++ !!!

string variable types

**char**: character variable (1 byte)

"nothing"

**void**: nothing but a name (not really a variable as it does not refer to something stored in memory referred to with that name, but something that needs to be defined because all names have to be defined – used for functions).

# Declarations

Variables and functions must be declared in C and C++ !!!

`struct`: structures

(Blocks of variables that don't all have to be the same type.)

later

Back to the discussion of libraries.

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

```
#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*/
```

You usually have to declare at least the `stdlib.h` for a program to compile.

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

```
#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 next two libraries that you almost always need are

the I/O library, **stdio.h** (how often do you write programs with absolutely no input or output?), 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, `hrdfavorites.h`, is something you (actually somebody whose initials are "hrd") wrote.

Notice the filenames all end in `.h`

Also notice the ones that come with C are within angle brackets `<>`, while ones you write (or are "local") are in quotes `" "`.

```
#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
```

Actually, these statements in your C program do not include the library routines/code in your program, that happens in the compile/link command where you have to specify them (and where they live - i.e. their path) **again** (the linker gets pre-compiled versions of these routines from the library/archive file).



```
#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
```

All these "h files" (as they are called) actually do is define all the names and types of the functions and variables associated with the libraries.

(C requires everything to be defined)

Functions come next since they have to be defined before they are used in the main program (or other functions – so the order of defining functions is important

(although this rule seems to be commonly broken and you can put the functions in any order in your source code – depends on how many "trips" the compiler makes through the code when compiling – typically one or two. The one trip compilers need things in the "proper" order, the two trip ones get to fix things up the second time through).

(This has implications for building libraries if you ever have to do that. And why random libraries were developed.)

Simple programs don't have functions, which are defined the same way the main program is, so we'll just skip ahead to that.

The main Program comes next

Main Program is in a "block" defined by the braces and contains the program itself

```
void main(int argc, char *argv[])  
{  
    printf("Hello.\n");  
}
```

Officially, we are defining a function called `main`, that returns nothing (declared as `void`), has some input arguments, and has a body contained in the `{ }`.

As Yogi Berra said - "it's déjà vu all over again"

Just as the shell is "just another program"!

The main C program is "just another function"!  
(C was developed by same group of guys that developed UNIX, can't you tell!).

So you have to do all the definitions, etc., you would have to do for any other function.

Under the declare everything rules - even things that are never explicitly "called" such as the `main` program - you have to say what each "thing" is in terms of its "result" (in memory).

The function `main` does not produce a result that is stored in memory.

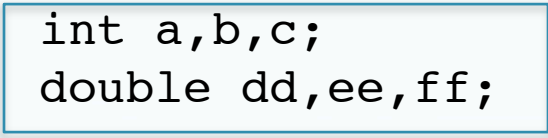
We (have to) tell the compiler that using the `void` type declaration.

Other possibilities for function types are `int`, `float`, `double`, `char`, `struct`, ...and pointers to all of them.

# Declaring variables and functions in a function

Here are some examples of variable and function declarations

```
void main(int argc, char *argv[])  
{  
    int a,b,c;  
    double dd,ee,ff;  
    ...  
}
```



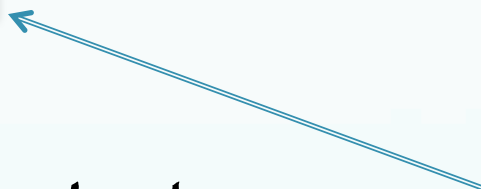
Variables must be declared at the beginning of your program/function block.

Lines of code within blocks have to be terminated by ";"

# Scope of variables

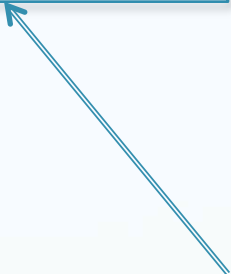
(where they are 'known')

```
void main(int argc, char *argv[])  
{  
    int a,b,c;  
    double dd,ee,ff;  
    ...  
}
```



These variables are only 'known' in this block, i.e. between the {}.

```
void main(int argc, char *argv[])  
{  
    int a,b,c;  
    double dd,ee,ff;  
}
```



You have to declare the variables in the argument list of the function call.

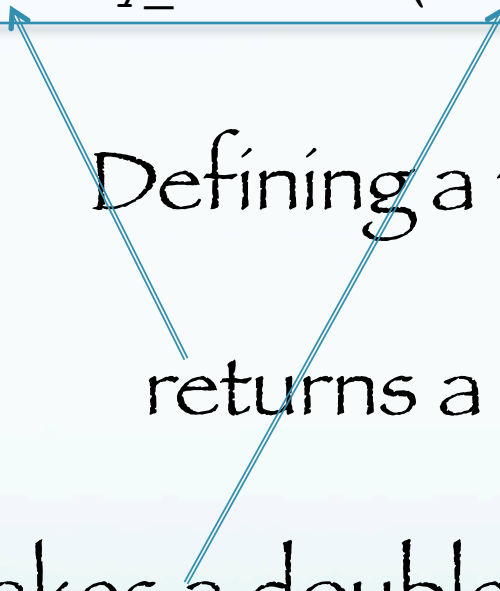
These are inputs to (not outputs from) the function.

These variables are also only 'known' in this block, i.e. between the {}.



```
void main(int argc, char *argv[])  
{  
    int a,b,c;  
    double dd,ee,ff;  
    double my_function(double);  
}
```

Defining a function that  
returns a double and  
takes a double input argument.

The diagram consists of two blue arrows. One arrow originates from the word 'returns' in the text and points to the 'double' return type in the function declaration 'double my\_function(double);'. The second arrow originates from the phrase 'takes a double input argument' and points to the 'double' parameter in the same function declaration.

# Prototyping

Declaration on steroids.

Defining what each function returns and its list of arguments is called prototyping.

`void` – returns nothing

`int` – returns integer

`float` – returns float

`char` – returns character

`struct` – returns structure

`*something` – returns pointer to "something"

If you forget to "type" a function, `int` is assumed and the compiler will complain.

# Declaring variables/functions

```
void main(int argc, char *argv[])  
{  
    int a,b,c;  
    double dd,ee,ff;  
}
```

declare the "type" of the function definition  
(notice you don't need the word function).

Since the main function does not "return"  
anything (put something in memory), its type is  
**void**

(you can get away with `void main()` if you are not processing/expecting any input arguments and some compilers will even let you get away with just `main()`.)

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



The two arguments are an integer and a pointer (coming up next) to a character array (which is itself a pointer). These two things come from the shell (the calling routine).

The integer has the number of command line arguments, and the second argument is a pointer to an array of character pointers, each pointing to the address of the beginning of the character string for each argument.

All this extra typing is supposed to help make sure your code is consistent and protect you from yourself (very un-UNIX like – trying to help the user become a better typer).

You also have to initialize all variables before you use them to avoid getting whatever happens to be sitting in that location in memory.

(i.e. before using a variable on the RHS it should show up being set to something on the LHS.)

This is important when doing things like `x++;`

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 loop index */
    }
}
```

# Structures.

we can declare a block of data containing different data types by means of a structure declaration.

The type is struct

```
struct tag {  
char lname[20]; /* last name */  
char fname[20]; /* first name */  
int age; /* age */  
float rate; /* e.g. 12.75/hour */  
};
```



# Structures.

This defines a new type of variable named tag.

This variable will be a block of memory with 20 bytes for a character array/string lname, 20 bytes for a character array/string fname, 4 bytes for an integer and 4 bytes for a float.

```
struct tag {  
char lname[20]; /* last name */  
char fname[20]; /* first name */  
int age; /* age */  
float rate; /* e.g. 12.75/hour */  
};
```

# Structures.

To use this structure we define a structure variable of type tag

```
struct tag my_struct; /* declare the  
structure my_struct */
```

and to assign/reference the elements of the structure

```
strcpy(my_struct.lname, "Jensen");  
strcpy(my_struct.fname, "Ted");  
printf("\n%s", my_struct.fname);  
printf("%s\n", my_struct.lname);
```

# Global Constants - `define`

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 traditionally declared after the `#include` calls

# Arithmetic Operators

+	plus
-	minus
*	multiply
/	divide
%	modulo divide
++	increment
--	decrement
<<	bitwise left shift
>>	bitwise right shift
&	bitwise and
	bitwise or
^	bitwise xor

# Arithmetic Operators

Notice the useful operations – bitwise shifts and bitwise logical operations.

(no exponentiation! C is mean-and-lean!)

(only has math operations that correspond to cpu arithmetic unit register instructions – that's why you get the bitwise shifts and logicals)

# "Assignment by" Operators

`+=`      `sum`

`-=`      `difference`

`*=`      `product`

`/=`      `quotient`

`%=`      `remainder`

`<<=` `bitwise left shift`

`>>=` `bitwise right shift`

`&=`    `bitwise and`

`|=`    `bitwise or`

`^=`    `bitwise xor`

# Examples of C "shortcut" operators

<code>x++;</code>	use <code>x</code> , then increment by 1
<code>++x;</code>	increment by 1, then use <code>x</code>
<code>x+=5;</code>	add 5 to <code>x</code>
<code>x+=y;</code>	add <code>y</code> to <code>x</code>
<code>x*=y;</code>	multiply <code>x</code> by <code>y</code> , store in <code>x</code>
<code>x/=y;</code>	divide <code>x</code> by <code>y</code> , store in <code>x</code>
<code>x%=y;</code>	divide <code>x</code> by <code>y</code> , store remainder in <code>x</code>
<code>x=y&lt;&lt;2</code>	shift <code>y</code> left by 2, store in <code>x</code> ( <code>y</code> unchanged)
<code>x&lt;&lt;=2;</code>	shift <code>x</code> left by 2, store in <code>x</code>
<code>x&amp;=y;</code>	logical bitwise AND of <code>x</code> with <code>y</code> , store in <code>x</code>

# Conditional Operators

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

== equal to

!= not equal to

< less than

<= less than or equal to

> greater than

>= greater than or equal to



# Boolean operators

"Regular" (use in comparison tests)

&&	and
	or
!	logical not (compares variable)

Bitwise (operate on each bit – can't use in comparison tests)

&	and
	or
^	nor
~	not

## Type combinations

floats and doubles divided by floats and doubles are relatively easy to use

but problems tend to occur when performing division of other types.

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  
type cast

```
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.
```

# Loops

C is the original looping language (even though all previous HLL's have loops) ...love it or hate it

Statement blocks, or sequences of statements,  
are encased using

{ }

(this is general, not just for loops).

Statements in a block are executed in sequence  
from first to last by default

(statements in C are terminated by “;”. Otherwise C wraps lines, unlike fortran.).

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

# while

**while:** continues to loop as long as condition tests successful

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

There is no `print` command.

To print you use the commands `printf` (print to file) and `prints` (print to string) from the `stdio` library.

## do-while

Similar to `while` loop except test occurs at end of loop body.

Guarantees the loop is executed at least once before continuing.

```
do
{
printf("Enter 1 for yes, 0 for no :");
scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

# 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 stuff in block (defined by `{ }` ).

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



# if/else if/else

If expression is true, then run the first block (blocks are defined by the pair of braces { }) of commands. Else, if a second expression is true, run the second block of commands. Else, if neither is true, run last block of commands.

```
if ( a > b)
```

```
{  
    statements;  
}
```

```
else if (a == b)
```

```
{  
    statements;  
}
```

```
else
```

```
{  
    printf "%d is less than %d.\n", a, b;  
}
```

# switch looks like

```
switch ( expression )  
{
```

```
    declarations
```

```
    .  
    .  
    .
```

this part can repeat

```
case constant-expression :
```

```
    statements executed if the expression equals the  
    value of this constant-expression
```

```
    .  
    .  
    .
```

```
    break;
```

```
default :
```

```
    statements executed if expression does not equal  
    any case constant-expression
```

```
}
```

# switch

Control passes to the statement inside the switch block whose case constant-expression matches the value of `switch ( expression )`.

No blocks, `{ }`, for the "cases" in the switch block.

The switch statement can include any number of case instances, but no two case constants within the same switch statement can have the same value.

## switch

Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a **break** statement transfers control out of the body.

This construct is particularly useful in handling input variables.

# switch

So what does this do?

```
c='a';  
switch( c )  
{  
case 'A':  
    capa++;  
case 'a':  
    lettera++;  
default :  
    otherletters++; }
```

# switch

So what does this do?

```
c='a';  
switch( c )  
{  
case 'A':  
    capa++;  
case 'a':  
    lettera++;  
case 'b':  
    letterb++;  
default :  
    otherletters++;  
}
```

matches here, so starts  
executing here,  
increments lettera by 1

but what does it do next,  
where does it go/stop  
executing (in the block)?

# switch

So what does this do?

```
c='a';  
switch( c )  
{  
case 'A':  
    capa++;  
case 'a':  
    lettera++;  
case 'b':  
    letterb++;  
default :  
    otherletters++;  
}
```

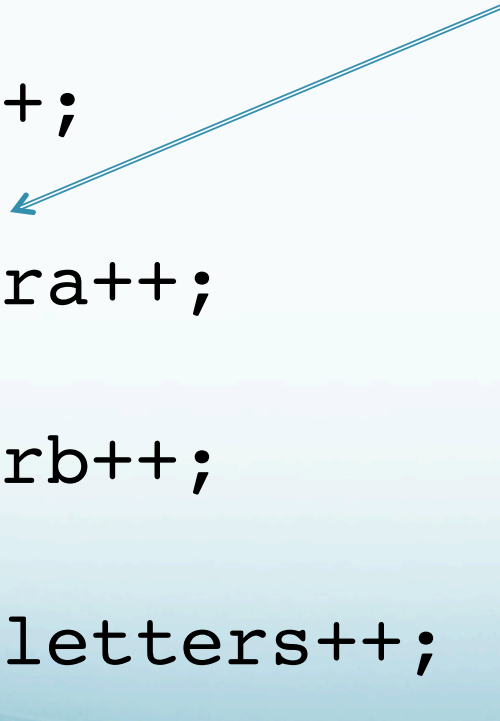
reread the description  
two slides ago.

It continues executing till the  
end of the block, or it  
encounters a **break** statement.

# switch

So what does this do?

```
c='a';  
switch( c )  
{  
case 'A':  
    capa++;  
case 'a':  
    lettera++;  
case 'b':  
    letterb++;  
default :  
    otherletters++;  
}
```



But this is probably not what we want to do here, as it will execute to the end and increment the number of lettera, letterb and otherletters.

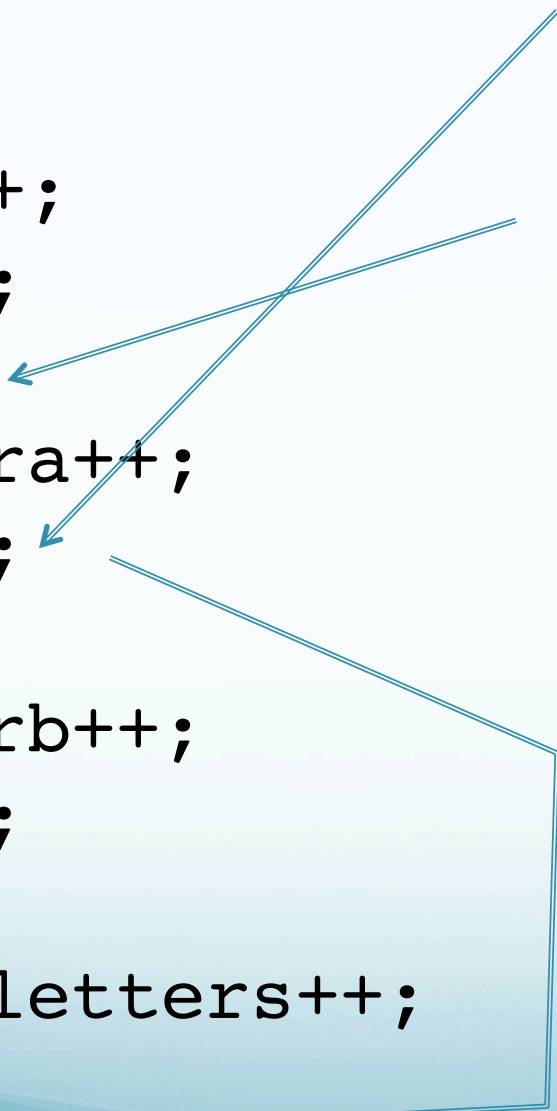


break command to the rescue

allows you to break-out of a for or while loop,  
or a switch block.

```
c='a';  
switch( c )  
{  
case 'A':  
    capa++;  
    break;  
case 'a':  
    lettera++;  
    break;  
case 'b':  
    letterb++;  
    break;  
default :  
    otherletters++;  
}
```

now it stops executing  
the block after  
matching and doing the  
statements between  
the match and the  
break.



In loops, the default behavior is the break out of  
the enclosing loop block  
(not the block associated with the `if`)

```
for ( a=0; a<20; a++ )  
{  
    if ( a > 10)  
    {  
        break;  
    }  
}  
## break or terminating loop here ##
```

But *you* may want to do it without the **breaks**

```
switch( c ){  
    case 'A':  
        capa++;  
    case 'a':  
        lettera++;  
    default :  
        total++;}
```

This default behavior in C is called "fall-through".  
When fall-through is the default action, **switch/case** statements are a frequent source of bugs among even experienced programmers, given that, in practice, the "break" is almost always the desired path, not the default fall-through behavior.

switch example with integer that counts number of times *i* is -1, 0 or 1.

```
switch( i )  
{  
    case -1:  
        n++;  
        break;  
    case 0 :  
        z++;  
        break;  
    case 1 :  
        p++;  
        break;  
}
```

the test cases have to be constants/static and test for equality

(not  $<j$  for example)

Take home message:

The `switch` statement is not the same thing as a  
big `if-if else-else` statement.

Each case must be unique, evaluated statically,  
and it has fall-through.

## Arrays

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

```
type name[dim];
```

```
double name[100];
```

```
/*you have to already know how big the array/  
vector will be in this case!*/
```

In C, arrays starts at position 0.

The array above is 100 elements long.

The elements of the array occupy adjacent locations in memory.

One way to access array elements is

```
name[ 0 ]
```

```
name[ 1 ]
```

```
name[ n ]
```

where the index gives the array element number,  
this is also the offset into the array from the

beginning (that's why C indices start at zero, the first element is at the address of the array plus zero - which is one of the secrets for efficiently using C arrays)

C does not do bounds checking, if  $n$  is greater than or equal to the array size (remember first element is 0) C will happily go there.



# Pointers

(The key to using arrays in C)

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

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

C does this with variables called pointers that store the address of other variables.

# Pointers

If you are going to use C  
you are going to use pointers!

# Pointers

Pointers are a type of variable and you therefore have to declare them.

The pointer itself is an integer, but it is a special type of integer.

You need to associate a pointer with the type of element to which it points.

# Pointers

You define a pointer by putting the asterisk modifier, \*, before the name.

```
int *p;    /*declared that p is a pointer  
to an integer variable */  
double *q; /*declared that q is a  
pointer to a doublevariable */
```

C will interpret the contents of p and q as addresses (which are integers).

We can do two things with addresses

- manipulate them
- get the contents of memory at that address

# Pointers

Notice that there are two parts to the definition, first that it is a pointer (the `*`) and what type of variable it points to (`int`, `float`, `char`, ...).

The variable type is important because this sets up the "step size" when you increment/decrement the pointer variable to get the "next/previous" value of that type of variable from memory.

Next we need a way to get the address of something to put it in the pointer variable.

We use the "address-of" operator (unary &).

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

```
p=&x;    /* the value of p contains the  
address of x */
```

We can then use the pointer variable to access the memory contents at that address.

But we need a way to indicate we want the data in the memory address pointed to by `p` rather than the value (address) stored in `p`.

This is done using the asterisk modifier, \*, before the variable name (as in the definition).

```
x=17;  
p=&x; /* the value of p contains the  
address of x */  
y=*p; /* put the contents of memory  
found at address p in y */
```

This is called "dereferencing" (\* is the dereferencing operator) and puts 17, the contents at the address pointed to by p, into the variable y.

y=\*p; produces the same result as  
y=x;



One can also set the value of `x` directly using the pointer to `x`'s address

```
p=&x; /* p has the address of x */  
*p=17; /* same result as x=17 */  
*p=y; /* same result as x=y */
```

You can see the power (and confusion) that pointers offer.

I can now set `p` to some other address and store something in this new address.

It is often difficult to figure out one is doing with pointers when reading the code.

A pointer is "like an integer".

You can do arithmetic with `p`, but this arithmetic "knows about" the size of the object `p` points to,

so adding 1 to a pointer makes it point to the next object (which is generally not one byte long ~ so the value in the "integer" `p` will jump by the size of the object it points to).

To refer to what is in the address pointed to by the pointer you use `*p`, on either the right or left sides of expressions.

## Back to arrays (or more on pointers)

The other secret to using arrays in C is that the array variable is effectively a pointer (even though the `*` was not used when the array variable was defined, the `[ ]` substitute for the `*`).

An array consists of multiple elements, accessed by `arrayname[n]`, so it "makes sense" that the variable `arrayname` works like a pointer.

```
int my_array[] = {1,23,17,4,-5,100};  
int *ptr;
```

```
void main() {  
    ...
```

```
    ptr = &my_array[0];  
    ptr = my_array;
```

The two lines do the same thing. So `my_array` all by itself acts like a pointer (if it was not a pointer, and of the proper type, C would complain when we tried to store it in the pointer variable `ptr`. C is protecting us from ourselves here.)

We cannot do this however

```
my_array=ptr;
```

as `my_array` is a constant value that cannot be changed once `my_array` is defined (either by the compiler or dynamically).

It is the address of `my_array` which is fixed/constant.

arrays and pointers are interchangeable

the expression  $a[i]$  is semantically equivalent to  
 $*(a+i)$

**Array subscripts vs. pointer arithmetic**

Element	First	Second	Third	<i>n</i> th
Array subscript	<code>array[0]</code>	<code>array[1]</code>	<code>array[2]</code>	<code>array[n - 1]</code>
Dereferenced pointer	<code>*array</code>	<code>*(array + 1)</code>	<code>*(array + 2)</code>	<code>*(array + n - 1)</code>

### Array subscripts vs. pointer arithmetic

Element	First	Second	Third	<i>n</i> th
Array subscript	array[0]	array[1]	array[2]	array[n - 1]
Dereferenced pointer	*array	*(array + 1)	*(array + 2)	*(array + n - 1)

```
int *i
```

```
...
```

```
i=0;
```

the expression `a[i++]` returns the value at `a[i]` and then adds `n` to the value of pointer `i` (the `++` follows the `i`), where `n` represents the size of one instance for the type of variable the pointer points to (4 for `int`, 4 for `float`, 8 for `double`, 1 for `char`,...)



### Array subscripts vs. pointer arithmetic

Element	First	Second	Third	<i>n</i> th
Array subscript	array[0]	array[1]	array[2]	array[n - 1]
Dereferenced pointer	*array	*(array + 1)	*(array + 2)	*(array + n - 1)

We could also do

```
i=a;
```

And then use

```
*i++;
```

returns the value at the address pointed to by *i*  
and then increments *i* by "one" to the next  
element of the array.



Using pointers is the most common method of  
accessing arrays

`*(a+i)`

rather than

`a[i]`

also, if `p` is a pointer and

`p=a;`

I can step through the array in a loop using

`b=p++;`

in addition to pointing to variables and arrays, pointers can point to pointers (which can make for very interesting debugging) and structures (more on them later) and be elements of arrays.

```
int* arr[8]; // An array of 8 int pointers  
(arr is effectively a pointer).
```

```
int *(arr[8]); // same as line above
```

```
int (*arr)[8]; // A pointer to an array of 8  
integers
```

This is how C handles character strings.  
Character strings are arrays with one character  
per element.

(In Fortran character strings are their own data  
type, like integers or reals. In C character strings  
are just arrays of bytes.)

# 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 continue until a a null [zero] byte is  
encountered).

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

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

Example of how not to build string array  
(although is legal), but shows how they work.

```
char my_string[40];  
my_string[0] = 'T';  
my_string[1] = 'e';  
my_string[2] = 'd':  
my_string[3] = '\0';
```

In C strings are "zero terminated".  
The string continues until the zero byte is  
encountered.

This means the only way to get the length of a  
string is to count it (slow).

Another example of how not to build string array  
(although is legal)

```
char my_string[40]=  
    {'T','e','d','\0',};
```

Usual way

```
char my_string[40] = "Ted";
```

Sets aside 40 bytes, but does not remember this information. If you write past `my_string+40` you are clobbering something.

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.

Pointers are used to pass arrays to functions.

C always passes arguments to functions by value  
[a copy], except when it does not [arrays].  
We will see the implications of "pass by value"  
next.

Fortran passes by address (pointer, but you  
don't have to deal with things as pointers in  
Fortran).

You can also pass pointers to functions (you just  
have to define everything properly).



C passes arguments to subroutines by value (a copy).

If you want results from the subroutine returned to the calling program there are two ways to do it.

You can have the function return a single "thing" (int, float, etc.) back using the

type function\_name(argument list)

format and then use as

```
int p,q;  
int function_name(int);  
...  
p=function_name(q);
```

C passes arguments to subroutines by value (a copy).

If you want to pass something back to the calling program through the argument list you have to pass something you can state by value.

If you try

```
void my_sqrt(float val_in, float val_out){  
val_out=sqrt(val_in);}
```

It will not do what you want/think since `val_in` and `val_out` in your function are copies of the two variables that your calling program gave to `my_sqrt`

```
my_sqrt(in_val,out_val)
```

so the calling program does not know what the subroutine did.

The value in `out_val` has not changed.

The solution to this is to use pointers.

```
void my_sqrt(float val_in, float *val_out){  
    *val_out=sqrt(val_in);}
```

Now I pass a copy of the value of the pointer to the address of val\_out to the function.

The function uses the pointer to change what is stored at the address in the pointer.

In the function I can use this pointer to store the result in the memory location pointed to by the pointer.

```
void my_sqrt(float val_in, float  
*val_out){  
*val_out=sqrt(val_in);}
```

Here's the call.

```
float out_val;  
my_sqrt(in_val, &out_val);
```

# Short example using both ways to pass variables

```
$ cat fnex.c
#include <stdlib.h>
#include <stdio.h>
void my_sqrt_1(float val_in, float *val_out)
{
double sqrt(double);
*val_out=sqrt(val_in);
}
float my_sqrt_2(float val_in)
{
double sqrt(double);
double val_out;
val_out=sqrt(val_in);
return val_out;
}
int main()
{
float x;
float y;
x=2;
my_sqrt_1(x,&y);
printf("%f\n",y);
y=my_sqrt_2(x);
printf("%f\n",y);
}
```

## Structures and pointers.

We can also use pointers with structures

```
struct tag *st_ptr; /* a pointer to  
                    a structure */  
  
st_ptr = &my_struct; /* point the  
                    pointer to my_struct */
```

# Structures and pointers

```
(*st_ptr).age = 63;
```

replace that within the parenthesis with that which `st_ptr` points to, which is the structure `my_struct`. Thus, this breaks down to the same as `my_struct.age`.

However, this is a fairly often used expression and the designers of C have created an alternate syntax with the same meaning which is:

```
st_ptr->age = 63;
```



```
#include <stdio.h>
#include <conio.h>
int main() {
```

```
    struct st {
        int id;
        char *name;
        char *address;
    };
```

Define structure

```
    struct st employee, *stptr;
```

Define instance and pointer to  
st structure

```
    stptr = &employee;
```

Set pointer to address of structure

```
    stptr->id = 1;
    stptr->name = "Angelina";
    stptr->address = "Rohini, Delhi";
    printf("Employee Info: id=%d\n%s\n%s\n", stptr->id,
        stptr->name, stptr->address);
```

```
    return 0;
```

Use pointer->element to  
access elements in structure.

```
}
```

Pointers can also be made to point to functions (so now you don't know what function is being called when you write the program – it gets determined during execution of the program based on the data being processed).

Pointers are also used for dynamic allocation of memory.

## Pointers to functions

Their declaration is easy: write the declaration as it would be for the function, say

```
int func(int a, float b);
```

# Pointers to functions

simply put brackets around the name and a \* in front of it: that declares the pointer.

Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* func returning pointer to int*/  
int *func(int a, float b);
```

```
/* pointer to func returning int */  
int (*func)(int a, float b);
```

# Pointers to functions

Once you've got the pointer, you can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression.

You can call the function using one of two forms:

```
(*func)(1,2);  
/* or */  
func(1,2);
```

# Pointers to functions - example

```
#include <stdio.h>
#include <stdlib.h>
void func(int);
main() {
    void (*fp)(int);
    fp = func;
    (*fp)(1);
    fp(2);
    exit(EXIT_SUCCESS); }

void func(int arg) {
    printf("%d\n", arg); }
```

# Structure arrays

You can also make arrays of structures (pointers to structures)

define

```
struct personal_data my_struct_array[100];
```

Use

```
my_struct_array[3].year_of_birth = 1974;
```

C does not do multidimensional arrays - but they can be simulated by arrays or arrays (another case of pointers to pointers).

```
#define ROWS 5
#define COLS 10
int multi[ROWS][COLS];
/* we can access individual elements of
the array multi using either of the
following */

multi[row][col];

*(*(multi + row) + col);
```

see <http://pwl.netcom.com/~tjensen/ptr/ch7x.htm> from "A tutorial on Pointers and arrays in C" at <http://pwl.netcom.com/~tjensen/ptr/pointers.htm>



# Higher-Level I/O

## To read in from external files

```
main(int argc, char *argv) {  
    const char *programe = 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;
```

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");
```

# File I/O example

Open file, write to it, close file.

```
#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 */
}
```

```
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 */
}
```

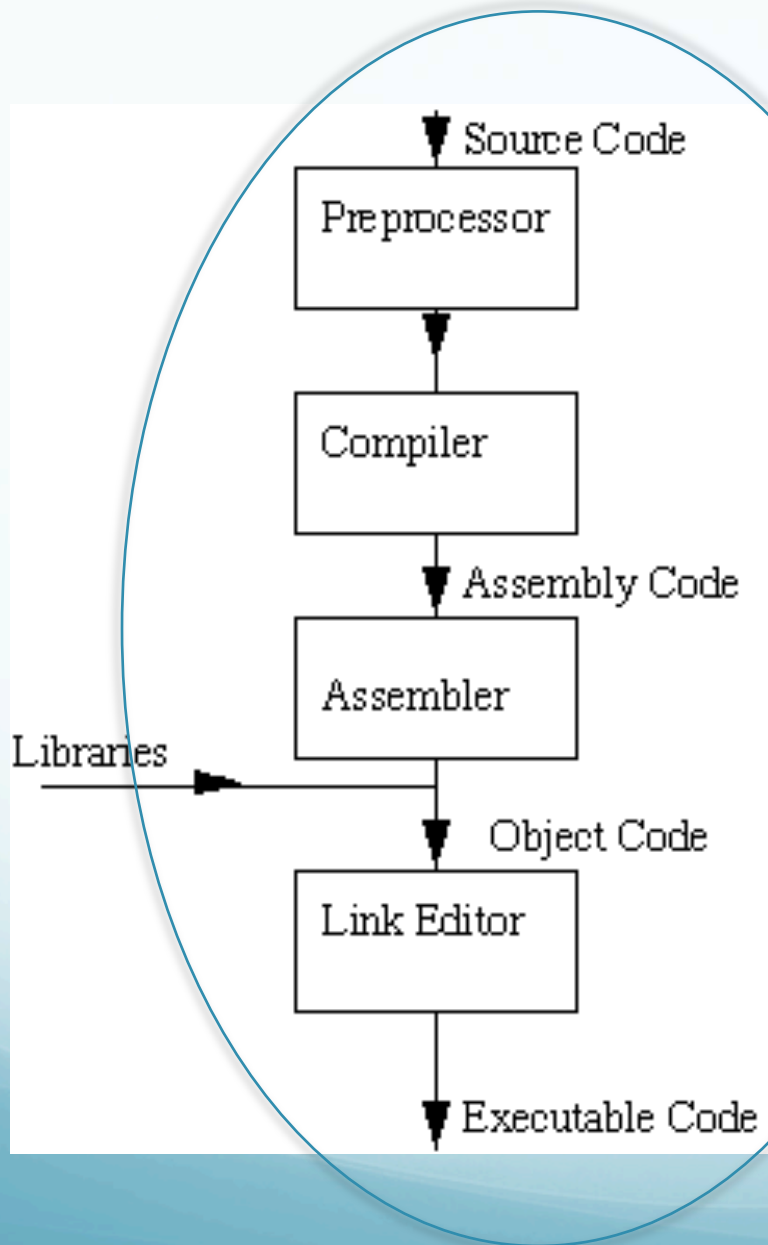
# Compiling

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 UNIX command.

C and Fortran are compiled using different compilers

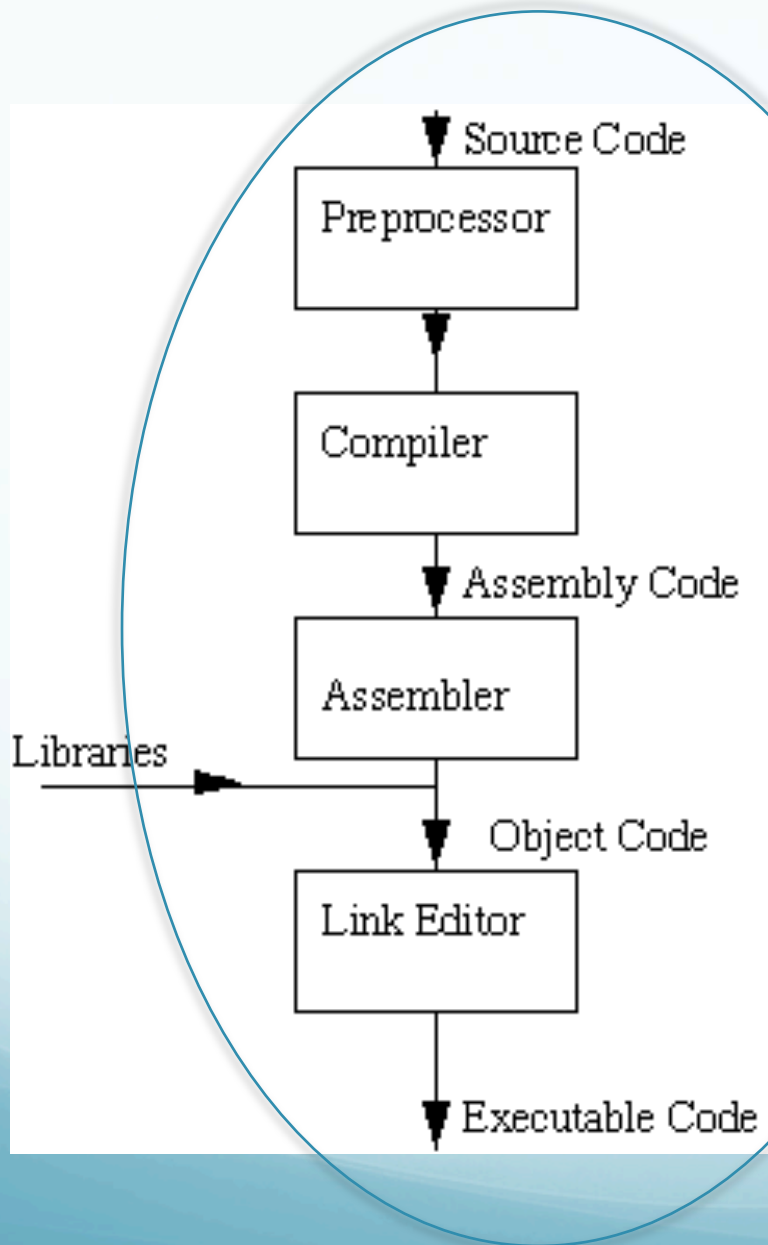
# “the compiler”



The preprocessor accepts source code as input and is responsible for removing comments and 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.

Both the Suns and mac's have `gcc` installed.



Another C compiler available at CERI is the SUN  
distribution `cc`.  
(`cc` on the mac is aliased to `gcc`)

There are differences, beyond the scope of this  
class, but in general `gcc` is a good option.

## 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 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` (object) file for each source file listed.

Object files can be subsequently linked by the `cc` command:

```
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 known as g77.

On the Mac, g77 has some problems with some codes.

Always check for platform dependence.

Another Fortran compiler available at CERl is the  
SUN distribution

/opt/Studio/SUNWsprow/bin/f77

/opt/Studio/SUNWsprow/bin/f90

/opt/Studio/SUNWsprow/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 it 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 $<

```