PH502: Scientific Programming Concepts

Irish Centre for High End Computing (ICHEC)

January 12, 2018

Overview

ICHEC
Irish Centre for High-End Computing

- Global variables and modules.
- Casting.
- Introduction to pointers.
- Argument Passing.
- Structs and Types.
- Makefiles, version control and code markup.
- C with Fortran.
- More pointers.

Global Variables

- In C, a variable that are defined outside any programming unit is called a **global variable**.
- These global variables are defined over the whole set of programming units.

```
#define MAXSIZE 100000
double array1[MAXSIZE]; float  array2[MAXSIZE];
float root2 = 1.4142136;
float function1(float x);
double function2(int i, int j);

int main(void) {
    ....
}
float function1(float x) {
   ....
}
double function2(int i, int j) {
   ....
}
```

Fortran Modules

- FORTRAN modules are special programming units that contain definitions, prototypes and even function/subroutines.
- More than one can be generated and named. Modules are then included in each programming unit which requires them.

```
module mod1
  real (kind=4), parameter :: root2 = 1.4142136
  integer (kind=4), parameter :: maxdim = 100000
end module mod1

module mod2
   interface
     function xxx(x)
        real (kind=4) :: xxx,x
     end function xxx
     subroutine yyy(a,b,c)
        real (kind=8), dimension(5,5) :: a,b,c
     end subroutine yyy
   end interface
end module mod2
```

Modules (Use)

```
program MyProg
   use mod1
   use mod2
!  Stuff in program unit
end program MyProg

subroutine yyy(a,b,c)
   use mod1
! Stuff in subroutine
end subroutine yyy

real (kind=4) :: function xxx(x)
! Cannot have "use mod2" here
! because contains prototype
!    Stuff in function
end function xxx
```

Modules (Use): without 'interface'

ICHEC
Irish Centre for High-End Computing

```
program MyProg
   use mod1
   use mod2
!  Stuff in program unit
end program MyProg
module mod1
  real (kind=4), parameter :: root2 = 1.4142136
  integer (kind=4), parameter :: maxdim = 100000
end module mod1
module mod2
     contains
     function xxx(x)
        real (kind=4) :: xxx,x
        ! stuff in function
     end function xxx
     subroutine yyy(a,b,c)
        real (kind=8), dimension(5,5) :: a,b,c
        ! stuff in subroutine
     end subroutine yyy
end module mod2
```

Changing Variable Type

Casting

- Sometimes it is necessary to change the variable type.
- Lets consider the expression $x = i/j$ where i, j are integers and x is real.

```
i = 2; j = 3;
x = i/j; /* Result 0.0 */
x = (float) i/(float) j;
/* Result 0.66666 */
```

```
i = 2; j = 3;
x = i/j ! Result 0.0
x = real(i)/real(j)
! Result 0.66666
```

- In changing the integer variables to floats the expression is calculated as a real expression hence the correct answer is returned.

Casting in C

- In C you can change one variable into any other variable type, *e.g.* $(int)x$, $(double)x$ or $(char)x$.
- However the limitations of each variable type need to be taken into account.
- Also when converting from a character to integer (say '1') you will get the ASCII table value and not one.
- When converting from double or float to integer, representation errors should be considered. The second line of code returns the number excluding digits after the dot. The third line returns the nearest integer value.

```
int i; double x; char c='1';
i = (int) x;
i = (int) rint(x);
i = (int) c - 48;
```

Casting in Fortran

- There are intrinsic functions that perform the conversions, *int*(x), *real*(x) and *dble*(x) to convert x to integer (kind=4), real (kind=4) and real (kind=8). Like C, numeric ranges need to be considered.

```
integer (kind=4) :: i
real (kind=4) :: x
real (kind=8) :: dx
i = int(dx); dx = dble(x); i = nint(x);
dx = dble(1.0)
```

- The function *nint*(x) returns the closest integer to x.
- To convert to and from character variables internal read and write statements are used.

Internal Read and Write

Fortran

- Previously we have used "write(6,*)" to print information to the screen.
- Internal write statements allow conversion of strings to numeric variables.

```
character (len=24) :: line
character (len=9) :: s
integer (kind=4) :: i
real (kind=4) :: x
line = ' 1234 1234.456 etc etc.'
!       1   5   10   15   20  24
read(line,'(i5,f9.3,a9)') i,x,s
! now i=1234, x=1234.456, s=" etc etc."
write(line,'(i5,f9.3,a9)') i,x,s
```

- The read statement allows conversion of a character string into numerics and write is the opposite.

Pointers in C

- Pointers are variables that point to a location in memory.
- Here *px* is a pointer to an integer variable. *px* is the $l-value$ of the variable and $*px$ the $r-value$.

```
int *px, x;
// Set l-value to that of variable x, r-value set to 10
x = 10; px = &x;
// Set r-value to 1
*px = 1;
// Add one to r-value
(*px)++;
// ++ is a higher precedence than *
// Print l-value and r-value
printf(" l and r values %p, %d\n",px,*px);

// Equivalently
x= 2;
printf(" l and r values %p, %d\n",&x,x);
```

Pointers in Fortran

- Pointers are not as necessary in FORTRAN but we include them for completeness.

```
integer (kind=4), pointer :: px
integer (kind=4), target :: x

x = 10; px => x;
px = px + 1          ! x = 11
write(6,*) ' x and address ',x,loc(x)
```

- A FORTRAN pointer can only be associated with the same type of variable with a "TARGET" attribute.
- The $l-value$ cannot be accessed so there is no equivalent of $px++$ in FORTRAN, *i.e.* pointer arthmetic is not permitted.

Subroutines

- Subroutines are a FORTRAN construction. They differ from functions in that information can be passed in an out via arguments. Also it has no return value.
- The equivalent in C is a function that returns "void" or nothing. Even main() in C is a function. Normally it passes 0 if the program executes normally and something else if there is an error.
- Why choose a subroutine rather than a function? In reality there is no real reason for one or the other. In C there is no real distinction. In Fortran it is normal to use subroutines except for very simple operations. A function can only return one variable type but a subroutine can return many.
- To pass arguments in and out of C functions we need to pass $l-values$ rather than $r-values$.

Subroutine Example

ICHEC
Irish Centre for High-End Computing

```
module consts
  real (kind=4) :: pi
end module
program fexample
  use consts
  interface
    subroutine degtorad(d,r)
      real (kind=4), intent(in) :: d
      real (kind=4), intent(out) :: r
    end subroutine degtorad
  end interface
  pi = atan(1.0)*4.0;    deg = 10.0;
  call degtorad(deg,rad)
  write(6,*) ' Degrees and radians ',deg,rad; stop;
end program fexample
subroutine degtorad(d,r)
  use consts
  real (kind=4), intent(in) :: d
  real (kind=4), intent(out) :: r
  r = (d * pi)/ 180.0; return;
end subroutine degtorad
```

Passing Arguments Out

C

- To pass arguments out of a function they must be passed by $l-value$.
- Below is an example where the $r-value$ of two variables are swapped. The $l-values$ of a and b are passed to "swap".

```c
#include <stdio.h>
void swap(int *px, int *py);
int main(void) {
  int a,b;
  a = 1; b = 2;
  swap(&a,&b);
  printf("a=%d and b=%d with l-values %p %p\n",a,b,&a,&b);
  return 0;
}
void swap(int *px,int *py) {
   int temp;

   temp = *px; *px = *py;
   *py = temp;
   printf(" In swap %p, %p\n",px,py);
/* return incorrect as void fn */
}
```

Passing Arguments

Fortran vs C

Fortran:

- rules for arguments are complicated and compiler dependent
- scope is controlled by: intent(in), intent(out), intent(inout)

C:

- non-array arguments
  - are passed by $r-value$ by default
  - use pointers to pass by $l-value$ and out of function
- arrays are passed by $l-value$

Generic Subroutines

FORTRAN

- A generic function or subroutine is one that works on different variable types *e.g.* integer and real variables.
- In fact most of the intrinsic functions are generic.
- The cot function is constructed below.

```
interface cot
  function fcot(x)
     real (kind=4) :: fcot,x
  end function fcot
  function dcot(x)
    real (kind=8) :: dcot,x
  end function dcot
  function icot(x)
    integer (kind=4) :: x
    real (kind=4) :: icot
  end function icot
end interface cot
```

Cot Function

- Here is the rest of the program.

```
program fexample
   integer (kind=4) :: i
   real (kind=4) :: x
   real (kind=8) :: dx

   i = 1; x = 1.0; dx = dble(1.0);
   write(6,*) cot(i),cot(x),cot(dx)
end program fexample

real (kind=4) function icot(x)
   integer (kind=4) :: x
   icot = 1.0/tan(real(x))
end function icot
real (kind=4) function fcot(x)
   real (kind=4) :: x
   fcot = 1.0/tan(x)
end function fcot
real (kind=8) function dcot(x)
   real (kind=8) :: x
   dcot = dble(1.0)/dtan(x)
end function dcot
```

Generic Functions in C

- Below is a similar example to the previous *cot* function.
- As we do not know the variable type the argument's $l-value$ is passed as "void *".
- Within the function the variable type is reestablished, with the help of the other argument "type".
- The tan function takes a double as input and returns a double. Thus the variable y (within the function) has its $r-value$ set to that of the input argument.
- The function then returns a $r-value$ of type double.

```
double cot(void *px, int type)
int main(void)
  int a; double dx,cotval;
  a =1; dx = 1.0;
  cotval = cot(&a,1);
  cotval = cot(&dx,3);
  return 0;
}
double cot(void *px, int type) {
   double y;
   switch (type) {
   case (1):
      y = *(int *) px;
      break;
   case (2):
      y =  *(float *) px;
      break;
   case (3):
      y = *(double *) px;
      break;
   }
   return 1.0/tan(y);
}
```

User Input

C

- So far we have looked at printing information to the screen, "printf".
- Programs can also accept user input from the screen, "scanf".

```c
#include <stdio.h>
int main(void) {
  int i; double a;
// Enter information from user
  printf("Enter an int and double\n");
  while (scanf("%d %lf",&i,&a) != 2) {
    printf("Problem with user input\n");
    fpurge(stdin);
  }
}
```

- The first thing to notice is that for "scanf" the addresses of the variables are passed.
- The formatting strings are the same as those for "printf".
- The "scanf" function returns the number of values it has read in, so it is recommended that this is tested.
- It is good practice to print the request for information.

User Input

FORTRAN

- The equivalent function in FORTRAN is below.

```
program uinput
   integer (kind=4) :: i,ierr=0
   real (kind=8) :: a

   write(6,*) ' Enter an int and double '
   do while (ierr .ne. 0)
     read(5,*,iostat=ierr) i,a
   end do
   if (ierr .ne. 0) write(6,*) ' Problem with input '
end program uinput
```

- In fortran read and write statements are assigned to a unit. Unit 5 is input from screen and 6 output to screen.
- The error is returned via an argument.
- Formatting is the same as that for write but the "*" format will accept anything.

Principal Minor

$$\begin{pmatrix} 0 & 1 & 2 & 3 & \cdots & n \\ 1 & 2 & 3 & 4 & \cdots & n+1 \\ 2 & 3 & 4 & 5 & \cdots & n+2 \\ 3 & 4 & 5 & 6 & \cdots & n+3 \\ \vdots & & & & & \vdots \end{pmatrix}$$

Structs and Types

- Supposing you wanted to classify the 4x4 principal minors of a matrix.
- For each class for a particular order you want to store:
  1. Number (integer)
  2. Determinant (float)
  3. Example matrix (array[4][4])
- Representing the above data structure can be achieved using structs.

```c
struct pmclass {
  int num;
  float det;
  float exam[4][4];
};
```

```
type pmclass
  integer (kind=4) :: num
  real (kind=4) :: det
  real (kind=4) :: exam(4,4)
end type
```

- Structs/Types allow explicit associations between variables which can make the program easier to interpret. It can reduce the number of different variables needed or reduce memory usage. (Consider the data structure of different orders).

Using Structs/Types

C

- Below are examples of using elements of a struct or type.

```
int i,j,k,Nclasses;
struct pmclass pmc[Nclasses];

for (i=0; i<Nclasses; i++) {
   pmc[i].num = 0;
   pmc[i].det = 0.0;
   for (j=0; j<4; j++) {
     for (k=0; k<4; k++) {
       pmc[i].exam[j][k] = 0.0;
     }
   }
}
```

Using Structs/Types

FORTRAN

```
integer (kind=4) :: i,j,k
type (pmclass) :: pmc(Nclasses)

forall (i=1:Nclasses)
   pmc(i)%num = 0
   pmc(i)%det = 0.0
   pmc(i)%exam = 0.0
end forall
```

Makefiles

- Large software packages can be installed semi-automatically.
- A typical process would be “configure”, “make” and “make install”.
- The configure step generates the “Makefile”s that are appropriate for your system. The make step build all the executable programs which are then installed into a publicly accessible directory.
- Here we shall focus on the “make” phase. Below is an example.

```
VARS = something

section: dependencies
<tab>        statement1; \
<tab>        statement2
```

- When a user types “make section” the dependencies are checked. Then the statement in that section are executed within a shell.

Example

- Below is an example Makefile. Variables are defined at the top. There are several sections which are interdependent. Prior dependencies are executed first.

```
CC = icc
CCFLAGS = -g -ansi
LDFLAGS = -lm
ROOTDIR = $(shell pwd)
CTARGS = prog1 prog2

install: prog
    cd /usr/bin; cp $(ROOTDIR)/prog .

prog: prog1.o prog2.o
    echo "Linking"; \
    $(CC) -o $@ prog1.o prog2.o $(LDFLAGS)

$(CTARGS):
    echo "Compiling"; $(CC) -c $(CFLAGS) $@.c
```

Version Control

Git

- Version control is especially important when software is being maintained by a group.
- Each authour has a local set of files and a snap shot of the files contained in a repository. If changes are made to the files then they differ from the repository snap shot. The repsitory can be updated giving a new snap shot. A record of modifications are maintained within the repository enabling a return to previous versions.
- There are a number of version control packages: cvs, svn, git, mercurial, bazaar, monotone and libresource. At ICHEC we use git.
- Here we will focus on the command lime interface to "git". A git repository can be created within a directory. All files and subdirectories that are part of the repository will be version controlled.

Git

- Below is a table of some git commands. These can be used to maintain a local git repository.

Command	Function
git init	create repository
git add/rm	add remove files in repo
git commit	commit the modifications to the repo
git tag	tag the current committed version
git checkout	take file from repo
git log	check the committed changes to the repo
git diff	see changes from the repo version
git status	shows file changes from repo

Remote Repository

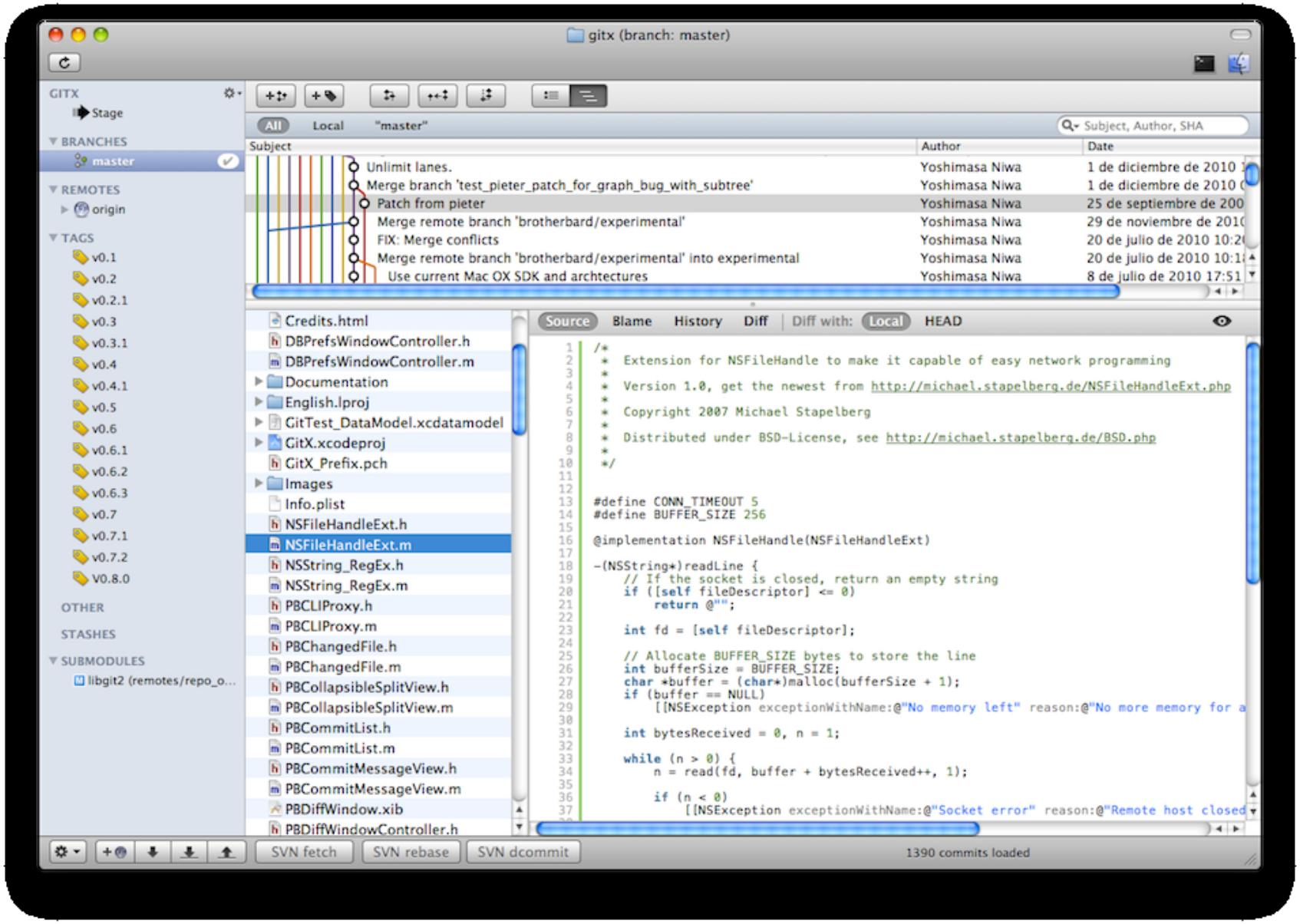
- It is possible to have the repository in a different location. In this case both the local repository and remote one need to be aligned. Additional commands are:

Command	Function
git clone	clone central repo locally
git pull	get lastest version from central repo
git push	synchronize local repo with centre

- An example of using git

```
# Make sure up to date with remote repo
git pull
# Edit prog.c and make but changes cause error
# Return to the original
git checkout prog.c
# Edit and make, this time successful
# Commit changes and push to remote repo
git commit -m "Fixed bug" prog.c
git push origin master
```

Git Gui


gitx (branch: master)
GITX
Stage
BRANCHES
master
REMOTES
origin
TAGS
v0.1
v0.2
v0.2.1
v0.3
v0.3.1
v0.4
v0.4.1
v0.5
v0.6
v0.6.1
v0.6.2
v0.6.3
v0.7
v0.7.1
v0.7.2
V0.8.0
OTHER
STASHES
SUBMODULES
libgit2 (remotes/repo_o...
All
Local
"master"
Subject, Author, SHA
Subject
Author
Date
Unlimit lanes.
Merge branch 'test_pieter_patch_for_graph_bug_with_subtree'
Patch from pieter
Merge remote branch 'brotherbard/experimental'
FIX: Merge conflicts
Merge remote branch 'brotherbard/experimental' into experimental
Use current Mac OX SDK and archtectures
Yoshimasa Niwa
1 de diciembre de 2010
25 de septiembre de 200
29 de noviembre de 201
20 de julio de 2010
8 de julio de 2010 17:51
Credits.html
DBPrefsWindowController.h
DBPrefsWindowController.m
Documentation
English.lproj
GitTest_DataModel.xcdatamodel
GitX.xcodeproj
GitX_Prefix.pch
Images
Info.plist
NSFileHandleExt.h
NSFileHandleExt.m
NSString_RegEx.h
NSString_RegEx.m
PBCLIProxy.h
PBCLIProxy.m
PBChangedFile.h
PBChangedFile.m
PBCollapsibleSplitView.h
PBCollapsibleSplitView.m
PBCommitList.h
PBCommitList.m
PBCommitMessageView.h
PBCommitMessageView.m
PBDiffWindow.xib
PBDiffWindowController.h
Source
Blame
History
Diff
Diff with:
Local
HEAD
/*
* Extension for NSFileHandle to make it capable of easy network programming
*
* Version 1.0, get the newest from http://michael.stapelberg.de/NSFileHandleExt.php
*
* Copyright 2007 Michael Stapelberg
*
* Distributed under BSD-License, see http://michael.stapelberg.de/BSD.php
*
*/
#define CONN_TIMEOUT 5
#define BUFFER_SIZE 256
@implementation NSFileHandle(NSFileHandleExt)
-(NSString*)readLine {
// If the socket is closed, return an empty string
if ([self fileDescriptor] <= 0)
return @"";
int fd = [self fileDescriptor];
// Allocate BUFFER_SIZE bytes to store the line
int bufferSize = BUFFER_SIZE;
char *buffer = (char*)malloc(bufferSize + 1);
if (buffer == NULL)
[[NSException exceptionWithName:@"No memory left" reason:@"No more memory for a
int bytesReceived = 0, n = 1;
while (n > 0) {
n = read(fd, buffer + bytesReceived++, 1);
if (n < 0)
[[NSException exceptionWithName:@"Socket error" reason:@"Remote host closed
SVN fetch
SVN rebase
SVN dcommit
1390 commits loaded

Git Resources

ICHEC
Irish Centre for High-End Computing

- on the command line: 'git –help' or 'man git'
- Git documentation: git-scm.com/documentation
- Online courses at Udacity, Coursera
- GitLab: free private git repository host

Comments

- Comments in code can improve its interpretabilty, along with appropriate variable names and formatted code (indentation).
- Example comments are below

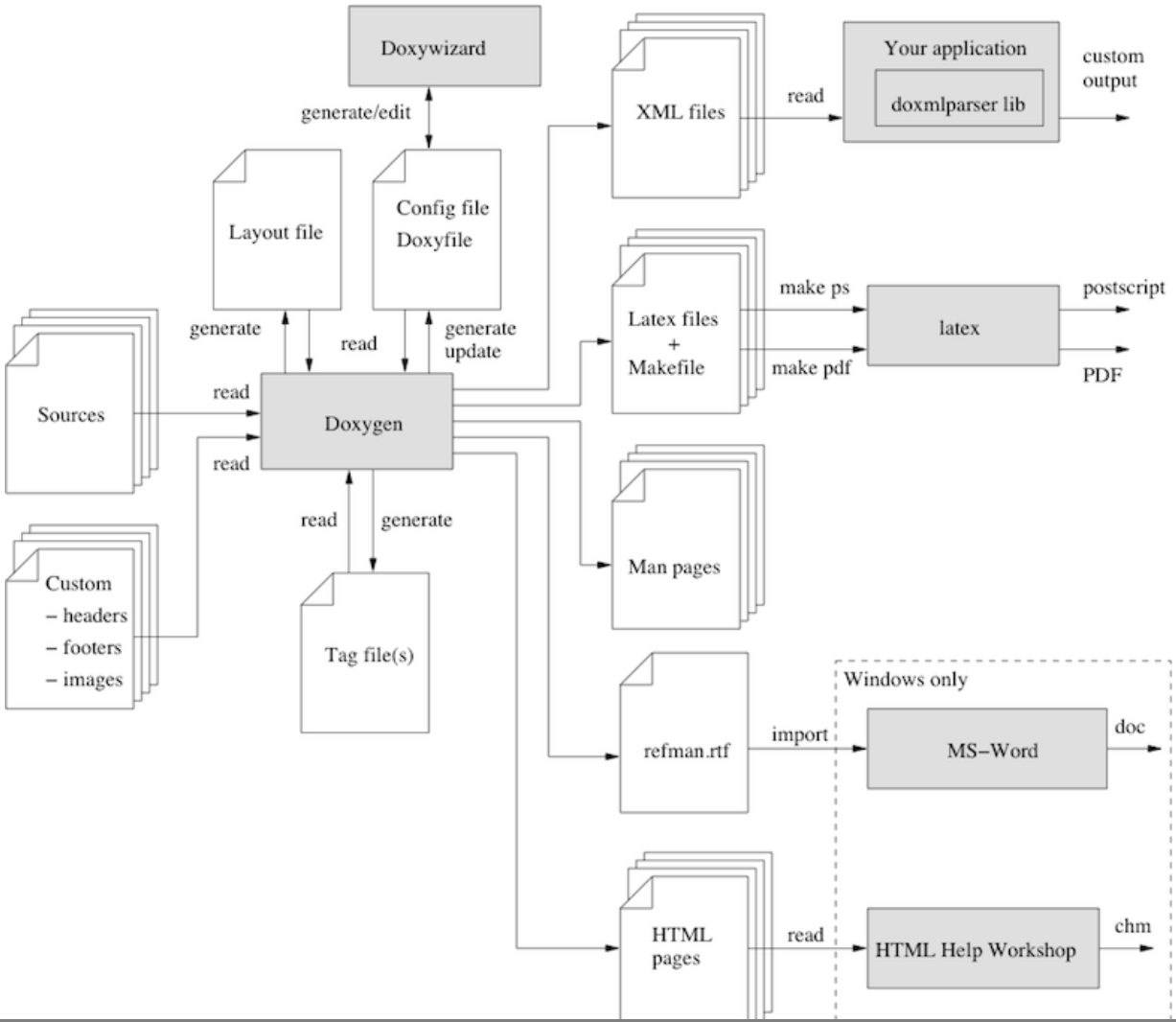
```
a=1; // Comment
// Another comment
/*
   Block
   Comment
*/
```

```
a=1    ! Comment
!
! Multi line
! Comment
```

- There is software available that can take the comment lines of a program and construct a user manual. The advantage of this approach is that th manual can be kept up to date with the code.

Doxygen

- Doxygen will automatically document a set of programs given a set of comment lines and directives. It works with C an FORTRAN code.
- Below is a flow diagram of Doxygen.



Using Doxygen

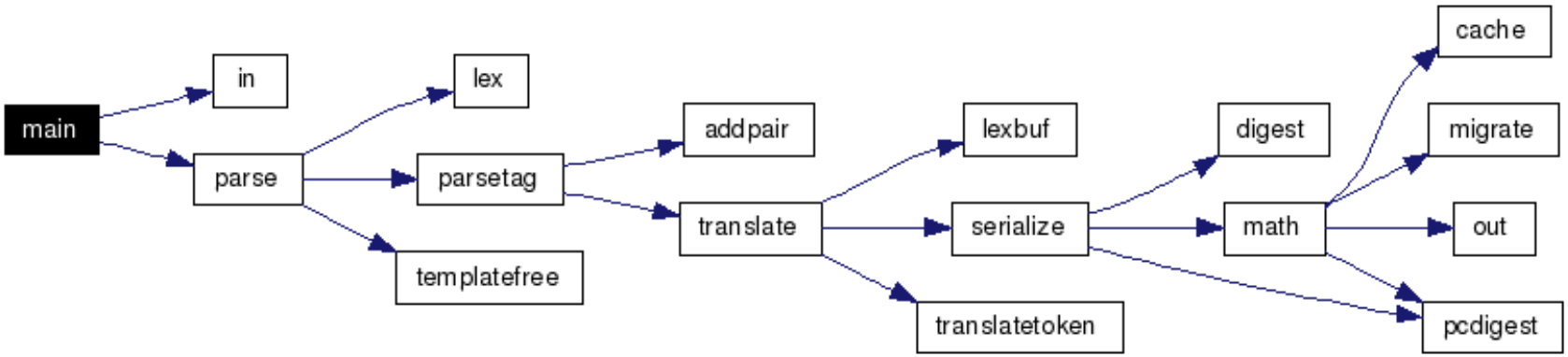
- To initialise the configure file type "doxygen -g Doxy.config". It can then be editted (or use the wizard) to customize it for the job at hand.
- Doxygen interprets modified comment blocks, typically these blocks are just before the program unit to which they refer.

```
/*! @brief "text".
*
* More detailed text.
*/
```

```
!----------------------
!> @brief "text".
!>
!> More detailed text.
!----------------------
```

- Doxygen has many commands or directives and we shall only mention a few of them.
- The basic HTML layout has a mainpage, "namespace/module" tab and a "file" tab.
- A call graph can also be generated, showing links between the program units. An example is on the next slide.

Call Graph


main
in
parse
lex
parsetag
templatefree
addpair
translate
lexbuf
translatetoken
serialize
digest
math
cache
migrate
out
pcdigest

Doxygen Commands

- The doxygen commands can be preceeded by a \ or @. Below is a a selection.

Command	Descrption
@mainpage	Main page title
@section	Subtitle and text for main page
@file	File documentation
@author	Program author
@brief	Description of coding unit
@param[in,out]	Argument description

- To generate the documentation type “doxygen Doxy.config”.
- Examples of marking up code, **from terminal**.

ICHEC
Irish Centre for High-End Computing

Break. Resume at 11:30

ISO_C_BINDINGS

- It is possible to link FORTRAN and C code together into one executable.
- However there are a number of differences between the way FORTRAN and C handle: variables, passing arguments, etc.
- To make things easier there is an ISO_C_BINDING module for FORTRAN.
- Here is a table of some of the conversions.

Fortran variable	C variable
integer(C_INT)	int
integer(C_SIZE_T)	size_t
real(C_FLOAT)	float
real(C_DOUBLE)	double
CHARACTER(C_CHAR)	char
TYPE(C_PTR)	void *

Passing Arguments

- When passing arguments they can be passed as $r-values$ or $l-values$.
- In the example below the integers are passed by $l-value$ and the floats as $r-value$.

```
interface
  function inc(i,j,x,y)
    integer(C_INT) :: i,j,inc
    real(C_FLOAT), value :: x,y
  end function
end interface
```

```
/* C function */
int inc(int *i, int *j, float x, float y);
```

ISO_C_BINDINGS

Pointers

- The FORTRAN function "C_LOC(X)" returns the address of the variable "X" as a "void *". The variable must have a "TARGET" attribute.
- There is a subroutine that converts from C to FORTRAN pointers "C_F_POINTER(cptr,fptr,shape)". "SHAPE" defines the size of the array to which the pointer points.

```
real (C_FLOAT), pointer :: fptr(:,:)
real (C_FLOAT), target :: array1(100,100)
type (C_PTR) :: cptr
! More code
fptr => array1
call c_f_pointer(cptr,fptr,[100,100])
```

Calling C from Fortran

FORTRAN Code

- Here we have the cot function implimented in C but called from FORTRAN.

```
program test1
  use, intrinsic :: iso_c_binding
  implicit none
  interface
    function ccot(x)
      import      ! include iso_c_binding in prototype
      real (C_DOUBLE) :: ccot
      REAL (C_FLOAT), value :: x
    end function
  end interface
  REAL (C_FLOAT) :: a
  REAL (C_DOUBLE) :: b
  a = 1;  b = 2; b = ccot(a)

  write(6,*) a,b
  stop
end program test1
```

Calling C from FORTRAN

C Code

- Here is the C function.

```c
#include <stdio.h>
#include <math.h>

double ccot(float x) {
    double y;
    y = x;
    return 1.0/tan(y);
}
```

Calling C from FORTRAN

Makefile

- Here is the associated Makefile.

```
CC = cc
FC = gfortran
LDFLAGS = -lm
CFLAGS = -c
FFLAGS = -c -fno-underscoring

test1: test1.o subs.o
    echo $(FC) -o test1 test1.o subs.o $(LDFLAGS); \
    $(FC) -o test1 test1.o subs.o $(LDFLAGS)


test1.o: test1.f90
    $(FC) $(FFLAGS) test1.f90

subs.o: subs.c
    $(CC) $(CFLAGS) subs.c
```

Call FORTRAN from C

- The same cot example but this time the function is in FORTRAN.

```c
#include <stdio.h>

double fcot(double x);

int main(void) {
   double a, b;
   a = 1.0;
   b = fcot(a);
   printf("a=%f, b=%f\n",a,b);
}
```

```
real (C_DOUBLE) function fcot(a)
   use, intrinsic :: iso_c_binding
   real (C_DOUBLE), value :: a
   real (kind=8) :: b
   b = dble(a)
   fcot = real(1.0/dtan(b),kind=C_DOUBLE)
   return
end function fcot
```

Pointers

- A pointer is a variable type that stores a memory address.
- This address can be used to read or modify the memory content in this particular location.
- The * and & operators allow us to manipulate pointers:
  - Used in a variable declaration, * marks a variable type as a pointer.
  - Used anywhere else, * is the *dereferencing* operator that accesses the value stored at the address stored by the pointer variable.
  - & is the pointer generating operator, used to generate a pointer to a variable's value.

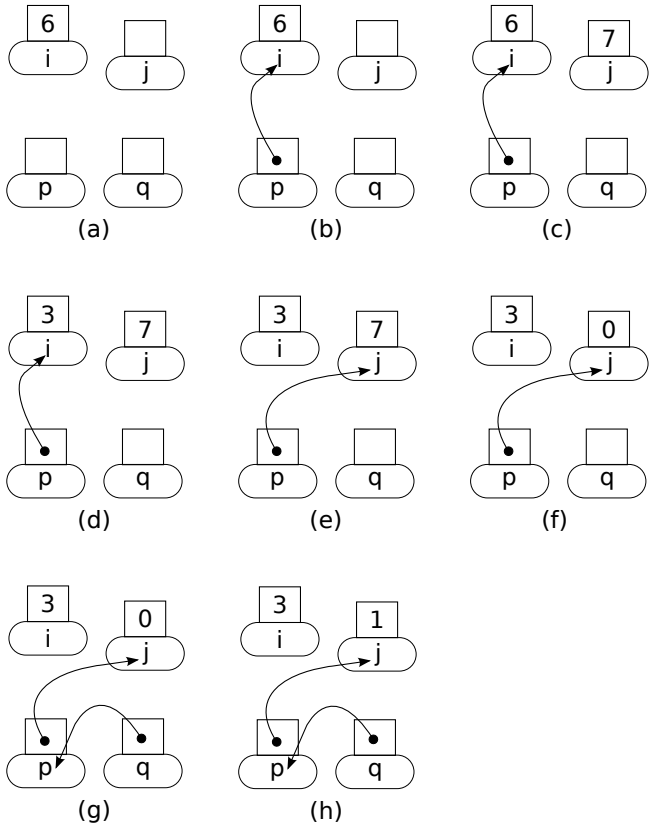
```
int i = 3;
int   *ip; // ip is a pointer to an integer
float *fp; // fp is a pointer to a float

ip = &i;   // ip points to the content of i
i == *ip;  // true
fp = &i;   // type error
```

Pointers (examples)

```
int *p, **q;
int i, j;

i   = 6;        // (a)
p   = &i;       // (b)
j   = *p + 1;   // (c) j==7
*p  = 3;        // (d) i==3
p   = &j;       // (e)
*p  = 0;        // (f) j==0
q   = &p;       // (g)
**q = *p + 1;   // (h) j==1
```



Pointers (examples)

- When declaring a pointer, whitespace between the star marker `*` and the variable name is ignored.
- That said, *it doesn't give a line-wide scope to the star marker*.
- Easy way to avoid any issues: favour use of the `type *var` syntax rather than `type* var`.

```
int *p;          // p is a pointer to an int
int* q;          // q is a pointer to an int
int* r, s;       // r is a pointer to an int
                 //     but s is just an int
int* *t, u, *v; // what's t? u? v?
```

Pointers and Arrays

- Each cell of a C array has its own address in memory.
- A pointer doesn't have to point at a variable, therefore can be used to reference a particular array cell.

```c
int arr[10];
int *p;
p = &arr[2];
```

```fortran
integer (kind=4), target :: arr(10)
integer (kind=4), pointer :: p
p => arr(3)
```

- The pointer `p` in the example above points to the third cell of the integer array `arr`.

Pointer Arithmetic

- A pointer being a numerical address in memory, it is possible to apply basic arithmetics on it.

```
int i, *p;

i = 42;
p = &i;
printf("p   points to address: %p.\n", p);
printf("p+1 points to address: %p.\n", p+1);

// sample output:
// p   points to address: 0xbfdf7028.
// p+1 points to address: 0xbfdf702c.
```

- What is the value of `*p`? `*(p+1)`?
- Looking at the example above, `p+1` is actually an increment of 4 over the hexadecimal value of `p`.
- This is because pointer arithmetics is *type-dependent* and a pointer increment will see the address increase by as many bytes required to store the type of variable pointed at. Here, an integer is stored on 4 bytes.

Pointer Arithmetic

- C arrays are stored as a contiguous sequence of bytes in memory.
- Therefore, if one has access to one cell, pointer arithmetics allow you to access all other cells.

```
int a[10], i, *p, *q, *r;

p = &a[2]; // points to 3rd cell
q = p + 3; // points to 6th cell
r = p - 2; // points to 1st cell
i = q - r; // works too. value of i?
```

Pointer Arithmetic

- C offers some convenient equivalences between pointers and arrays.

```c
int a[10], c, *p;

p = &a[0]; // point p to first cell
p = a;     // equivalent to previous line

c = a[3];   // c takes the value of the content
            // of the 3rd cell
c = *(p+3); // pointer arithmetic equivalent
c = p[3];   // quicker equivalent notation
```

- These equivalences are fundamental when passing pointers or arrays as function arguments.

Pointer Arithmetics (exercise)

- What are the values in a and what is p pointing at at the various points in the code?

```
int a[3], i, *p;

for (i=0; i<=2; i++) { a[i] = 0 } // (a)
p = &a[0];                       // (b)
p++;                             // (c)
*(p++);                          // (d)
*p++;                            // (e)
```

The NULL Pointer

- NULL is a special value for a pointer not pointing anywhere.
- Actual value in memory meaning 'NULL' is 0.

```
int *p = NULL;
int *q = 0; // same
```

```
integer (kind=4), pointer :: p
p => NULL()
```

Pointers and Strings

- No native string type in C.
- A string is an array of 'char'.
- The 'nul' character '\0' marks the end of the string (warning: nul $\neq$ NULL).

```
char s[]="Hello world";
char *p = &s[0];

while (*p !='\0'){
  printf("%c", *p);
  p++;
}
printf("\n");
```

- It is beside the point, but you could also have done `printf("%s\n", s);`.

Strings
FORTRAN

- In FORTRAN strings are defined by their length, *e.g.*

```
character (len=40) :: fstring, fstrarr(10)
```

- C and FORTRAN strings are not the same and need to be converted, if passing strings from C to FORTRAN.
- Use of pointers and strings are not required in FORTRAN, below are some useful functions:

```
fstring = ''          ! Null string
len(fstring)          ! String length, 40 in this case
len_trim(fstring)     ! String length, excl. trailing blanks
trim(fstring)         ! String excluding trailing blanks
fstring//fstrarr(1)   ! Concatenation
```

Allocating Memory

- Sometimes, you won't know beforehand how much space you need in an array.
- The memory for the array content needs to be allocated dynamically.
- Load `<stdlib.h>`, meet `malloc()`, and say hello to memory leaks.
- In FORTRAN memory can be allocated dynamically. Any variable must have the ALLOCATABLE attribute.

Allocating Memory

- `malloc()` takes an integer as argument, and returns a pointer to a block of memory that it reserved.
- The size of that memory block is as many bytes as the integer passed as argument.
- For convenience, the `sizeof()` function provides the size in bytes of a given type.

```
char *p, *q;
int  *r;

p = malloc(10);                    // 10 bytes
q = malloc(10 * sizeof(char)); // 1 char==1 byte==10 bytes
r = malloc(10 * sizeof(int));   // 1 int==4 bytes==40 bytes
r[8] == *(r+8);                    // true (both content of the
                                   // 9th cell of the r "array")
```

Allocating Memory

- Sometimes, `malloc()` is unable to allocate the requested memory.
- In such a case it returns `NULL`.

```
int *p;

p = malloc(4000 * sizeof(int));
if (p == NULL){
  printf("Failed memory allocation!\n");
  exit(); // or recover in a nicer way
}
```

Allocating Memory

FORTRAN

- In FORTRAN ALLOCATE() does the job of malloc(). The amount of memory is determined by the array size and the variable type.
- The shape of the array must be determined at declaration.
- If there is a problem with allocation then the program will terminate unless "stat" is used.

```
integer (kind=4) :: ierr
integer (kind=4), allocatable :: arr1d(:), arr2d(:,:)

allocate(arr1d(10),arr2d(10,5),stat=ierr)
if (ierr .ne. 0) write(6,*) ' Allocation error'
```

Freeing Memory

- Consider the following code.

```
int *p;

p = malloc(10 * sizeof(int));
// pretend we do stuff with *p
p = malloc(10 * sizeof(int));
// pretend we do some other stuff with *p
p = malloc(10 * sizeof(int));
```

- Three blocks of memory have been allocated. Yet, only the third is still accessible.
- The first two blocks are allocated, yet unreachable. They are *lost* memory that will only be freed on termination of the program.
- Dynamically allocated memory must be *freed* when no longer needed.

Freeing Memory

- The `free()` function frees memory previously allocated by `malloc()`.
- Unless explicitly freed, dynamically allocated memory remains allocated for the entire program execution.

```
int *p;

p = malloc(10 * sizeof(int));
// [...]
free(p);
p = malloc(10 * sizeof(int));
// [...]
free(p);
p = malloc(10 * sizeof(int));
// [...]
free(p);
```

- Though memory is freed for the system, `p` still points to the same area which is no longer reserved.
- Assigning NULL to a "freed pointer" can avoid doing anything dangerous with the memory.

Freeing Memory

FORTRAN

- Unlike C, local dynamically allocated memory is freed when returning from a program unit.
- If the array is in a "module" then it will only be freed at the end of the program (by default).
- However it is good practise to deallocate any dynamically allocated arrays, after use.

```
integer (kind=4), allocatable :: iarr(:)
integer (kind=4) :: n,error

n = 1000
allocate(iarr(n),stat=error)  ! Allocate

if (allocated(iarr)) then     ! Test to see if allocated
  deallocate(iarr,stat=error) ! Deallocate
endif
```

Reallocating Memory

C

- Sometimes, you may want to increase the size of a memory block..
- The `realloc()` function will do it for you.
- It will allocate the memory block requested, copy the contents of the old block at the beginning of the new one, free the old one, and return a pointer to the beginning of the new block.

```c
int *p;

p = malloc(100 * sizeof(int));
// do something with p, fill it up, etc
p = realloc(p, 200 * sizeof(int)); // p now has 200 'cells'
```

- Exercise: assume that `realloc()` doesn't exist. Write a `my_realloc()` function that does the same.

Reallocating Memory

FORTRAN

- Sadly there is no equivalent of the realloc() function. Some systems may have extensions that include realloc().
- Below is a function that does the job.

```
MODULE realloc_mod
CONTAINS
  FUNCTION reallocate(p, n)  ! realloc REAL 1D array
    REAL, POINTER, DIMENSION(:) :: p, reallocate
    INTEGER, intent(in) :: n
    INTEGER :: nold, ierr
    ALLOCATE(reallocate(1:n), STAT=ierr)
    IF(ierr .NE. 0) STOP "allocate error"
    IF(.NOT. ASSOCIATED(p)) RETURN
    nold = MIN(SIZE(p), n)
    reallocate(1:nold) = p(1:nold)
    DEALLOCATE(p)
  END FUNCTION REALLOCATE
END MODULE realloc_mod
```

Reallocating Memory

FORTRAN

- Below is an example of the use of the function.
- In FORTRAN the only way of expanding an existing array is to destroy it and allocate it again.
- Thus if there are any associated pointers, they must be reassociated.

```
PROGRAM realloc_test
USE realloc_mod
IMPLICIT NONE
REAL, POINTER, DIMENSION(:) :: p
  ALLOCATE(p(2))
  p => reallocate(p, 10000)     ! note pointer assignment
END PROGRAM realloc_test
```

Pointer and Arrays (again)

- Allocating a multiple of the memory block required to store a particular pointer type effectively allocates... an array of that type!

```
int *a,i;

a = malloc(6 * sizeof(int));
for(i=0; i<=5; i++){
    a[i] = 2 * i;
}
for(i=0; i<=5; i++){
    printf("a[%d] == %d\n", i, a[i]);
}
```

- Such a dynamic array doesn't exactly behave like a static array though...

Pointer and Arrays (final)

- The `sizeof()` function can be used to compute the number of elements of a static array.
- It doesn't work on dynamic arrays allocated through `malloc()`.

```
int *a, b[6];
a = malloc(6 * sizeof(int));
printf("Size of array 'a' is %d\n", sizeof(a)/sizeof(int) );
printf("Size of array 'b' is %d\n", sizeof(b)/sizeof(int) );
```

- Using dynamic arrays requires some bookkeeping to avoid access to out-of-bounds memory.

Pointers and Functions

- Pointers being variables, they can be passed to functions as arguments.
- A function can also return a pointer as its result.
- In fact the FORTRAN reallocate function did just that.

Pointers as function arguments

- Arguments passed by value to a function are *copied*. The function is then free to modify the value of the argument without impacting the original variable. This is a *call by value*.
- Pointers are no exception, C only passes a copy of the pointer's value to the function.
- This value being a memory address, the function is then free to use it to access or modify that memory content.

```
void addonetome (int *me){
  *me = *me + 1;
}

void main(){
  int i = 1;
  int *p = &i;
  addonetome(p);
  printf("i = %d\n",i);
}
```

- The function above modifies the *state* of the program. It is said to have a *side effect*.

Pointers as function arguments

- As mentioned earlier, pointers are passed by value, so modifying a pointer argument does not impact the original variable.

```
void addonetome (int *me){
  *me = *me + 1;
  me = NULL;
}

void main(){
  int i = 1;
  int *p = &i;
  addonetome(p);
  printf("i = %d\n",*p);
}
```

Arrays as function arguments

- As opposed to other simple types, a C array is *not* passed by value.
- When an array is passed as an argument, the function receives a pointer to the first element of the array.
- An array passed as a function argument therefore behaves like a dynamic array: no way to get size automatically.

```
void printarray(int a[]){
    int i;
    for (i=0; i< sizeof(a)/sizeof(int); i++){
        printf("%d\n", a[i]);
    }
}// fails. only prints 1st element

void printarray_n(int a[], int n){
    int i;
    for (i=0; i< n; i++){
        printf("%d\n", a[i]);
    }
}// works, given a valid n
```

Arrays as function arguments

- As an array passed as an argument generates a pointer, you can write your function to accept pointers.
- Notice the difference between the two `int a[]` and `int *a`.

```
void printarray_n(int a[], int n){
    int i;
    for (i=0; i< n; i++){
        printf("%d\n", a[i]);
    }
}
// function above same as:
void printarray_n(int *a, int n){
    int i;
    for (i=0; i< n; i++){
        printf("%d\n", a[i]);
    }
}
```

Pointers as a function return value

- A pointer is a valid function return value if the function is thus defined.

```
int *pointertothatint(int n){
// this useless function returns a pointer to an
// int of a chosen value. Useless, but works as
// intended.
    int *fp;
    fp = malloc(sizeof(int));
    *fp = n;
    return fp;
}

void main(){
    int *mp;
    mp = pointertothatint(14);
    printf("*mp = %d\n",*p);
}
```

Pointers as a function return value

- Warning this will not work because as the argument is passed by value, it is copied to another place in memory.

```
int *anotherpointertothatint(int n){
// same as before, but with a big scope issue
    int *fp;
    fp = &n;
    return fp;
}

void main(){
    int *mp;
    mp = pointertothatint(14);
    printf("*mp = %d\n",*p);
}
```

- When the function exits, the memory for the variable *n* is deallocated, so it may be filled up by something else at any time.

Arrays as a function return value

- Using the pointer/array equivalence, we can return a dynamic array from a function.

```
int *nintarray(int n){
    int *p;
    p = malloc(n * sizeof(int));
    return p;
}

void main(){
    int *p, i, nelements=100;
    p = nintarray(nelements);
    for(i=0; i<nelements; i++){
        p[i] = i+1;
    }
}
```