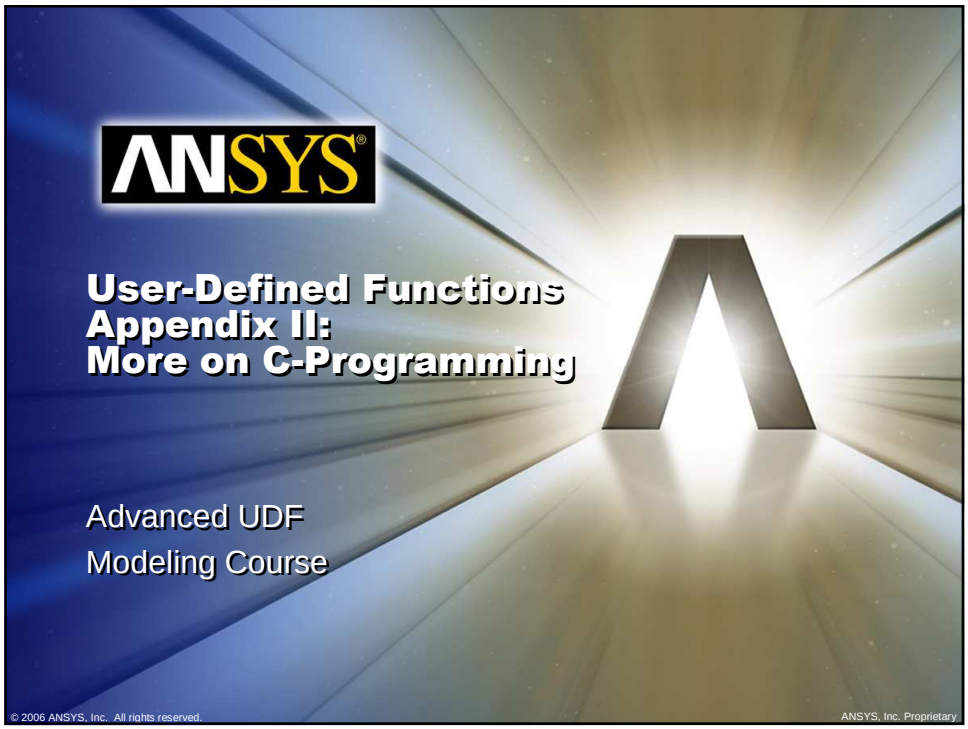


Introduction to C

- Why write in C?
- Topics covered in this brief introduction
  - C functions
  - C data types
  - Pointers, arrays & structures
  - Expressions and statements
  - C arithmetic and logical operators
  - Flow control
  - File I/O
  - C preprocessor

Why C?

- The FLUENT solver is written in C
- C is a versatile language with many versatile features
- Current UDF internal compiler supports only a subset of ANSI C

C Functions (1)

- The basic form of a C function:

```
/* A simple C function */          A comment line
#include "udf.h"                   A preprocessor directive for including files
#define PI 3.14159                 A preprocessor directive for macro substitution
float a = 1.2345;                  A variable with "global" scope, outside of {}
float myfunction(int x)            Function declaration (returns a float type)
{                                  Left curly brace opens body of function
   int y;                          Variable declarations
   float z;
   y = 11;                         Set y = 11
   z = a*(x+y)*PI;                 Compute z
   printf("Value is %f",z);        Print z to screen
   return z;                       Return float value
}                                  Right curly brace closes body of function
```

C Functions (2)

- All C statements must end with a semicolon (;)
- Comments are delineated by the character sequence
  `/* ... */`
  - comments can be placed anywhere in a C listing
  - use comments liberally to document your UDFs
- Groups of C statements are enclosed by curly braces ({ })

C Functions (3)

- Variables defined within a { } body are local to that group (local scope)
- Variables defined outside the function body can be used by all functions which follow the definition (global scope)
- If a function is defined with a specific type, it must return a value of the same type (using the return statement). If it doesn't return a value, it must be declared void

C Functions (4)

- C compilers include a library of standard math, I/O, and utility functions which can be used in your C code
- Some common I/O functions
  - **`scanf(...)`** - formatted read (like FORTRAN READ)
  - **`printf(...)`** - formatted print (like FORTRAN WRITE)
- Some common mathematical functions
  - **`sin(x)`** - sine function
  - **`cos(x)`** - cosine function
  - **`exp(x)`** - exponential function
  - **`sqrt(x)`** - square root function

Comparison with FORTRAN

- C functions are similar to FORTRAN function subroutines

```
/* A simple C function */
int myfunction(int x)
{
   int y,z;
   y = 11;
   z = x+y;
   printf("z = %d",z);
   return z;

}
```

```
C An equivalent FORTRAN function
      INTEGER FUNCTION MYFUNCTION(X)

      INTEGER X,Y,Z
      Y = 11
      Z = X+Y
      WRITE (*,100) Z
      MYFUNCTION = Z
100   FORMAT("Z = ",I5)
      END
```

The main() function

- You won't see it much with UDFs but there is a wrapper function called **`main()`**
- Generally a portal in the same was **`PROGRAM`** was in FORTRAN

```
#include <stdio.h>
int main(void )
{
   printf("Hello, world\n");
   return 0;
}
```


Exercise: Hello, world

- Start up the editor **`gedit`** or **`emacs`**
- Type in the program from the previous slide
- Save the file as **`hello.c`**
- Compile the program
  - **`cc hello.c –o hello`**
- Run the program
  - **`./hello`**

C Data Types (1)

- The UDF compiler supports standard C data types
  - **`int,long`** - integer data types
  - **`float,double`** - floating point data types (Usually use **`real`** in UDFs)
  - **`char`** - character data type
- Functions which do not return values are given the type void
  - **`void myfunction(int x) { ... }  /* No return needed */`**

C Data Types (2)

- You can convert from one type to another by “casting”

```
int z,x = 10;
float y = 3.14159;
z = (int)(x*y);    /* z = 31 */
```

- C also allows you to create “user-defined” types using **`typedef`**

```
typedef int mytype; /* define mytype to be integer type */
mytype a,b,c;       /* equivalent to int a,b,c */
typedef float real; /* or double depending on version*/
```

Pointers (1)

- A pointer is a variable which contains the address of another variable
- Possibly the greatest leap of faith required for the FORTRAN77 programmer
- When we declare a variable
  - `int k;`

  on seeing **`int`** the compiler sets aside 4 bytes of memory to hold the value of the integer
- In C, **`k`** is called an object. Later if we write
  - `k = 2;`

  the value 2 will be placed at the memory location associated with the object **`k`**

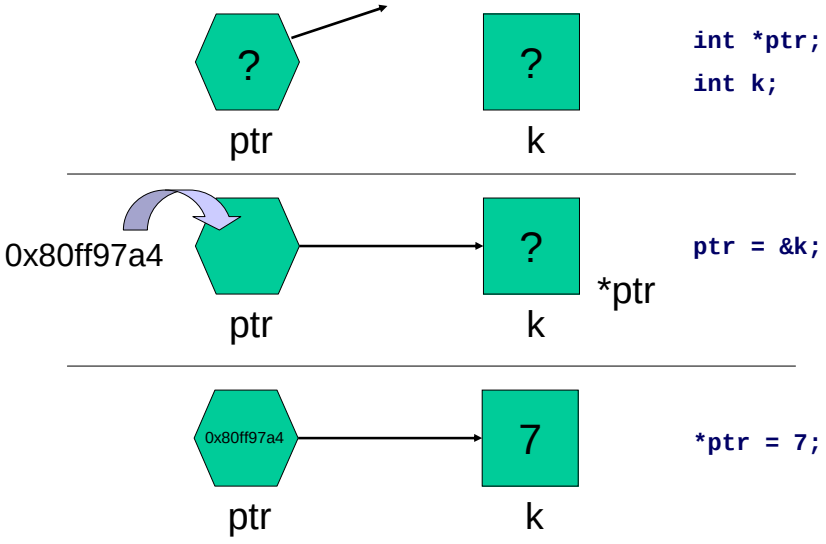
Pointers (2)

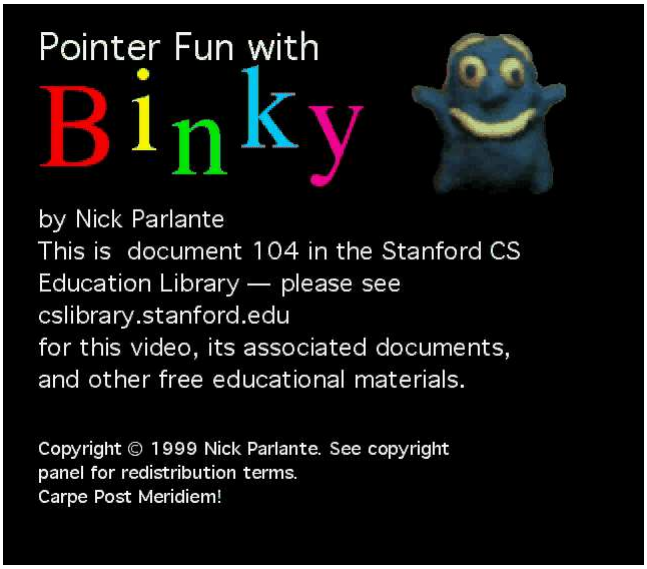
- Suppose we want a variable that holds a memory location (or *address*)
- Such a variable is called a *pointer*
- Consider the declaration
  - `int *ptr;`
- The **`*`** informs the compiler we wish to set aside enough memory for an address
- The **`int`** informs the compiler we wish to store the address of an integer

Pointers (3)

- Suppose we store the in **ptr** the address of our integer **k**
  - **ptr = &k;**
- Now **ptr** is said to *point to* **k**
- Suppose we want to copy 7 to the address pointed to by **ptr**
  - ***ptr = 7;/* Contents of ptr = 7 */**
- The * is the *dereferencing operator*
  - It allows access to the value stored at the address **ptr**
- Since **ptr** points to **k**, we have also set the value of **k** to 7

Pointers (4)





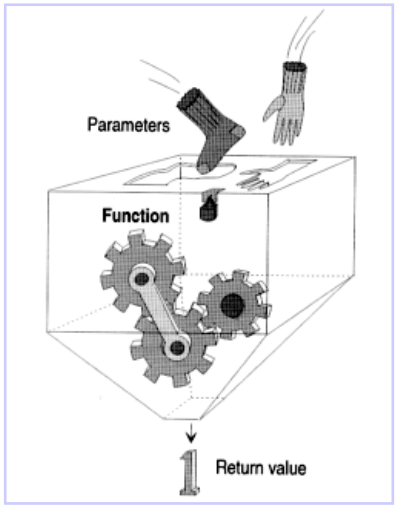
Exercise: Pointer1

- Save as pointer1.c, compile and execute it

```
#include <stdio.h>
int j, k;
int *ptr;
int main(void)
{
  j = 1;
  k = 2;
  ptr = &k;
  printf("\n");
  printf("j has the value %d and is stored at %p\n", j, (void *)&j);
  printf("k has the value %d and is stored at %p\n", k, (void *)&k);
  printf("ptr has the value %p and is stored at %p\n", ptr, (void *)&ptr);
  printf("The value of the integer pointed to by ptr is %d\n",*ptr);
  return 0;
}
```

Pointers (5)

- In C, function parameters are passed by value
- They only go one way
- You cannot alter the value of a parameter within a function and expect the calling function to see the change
  - Complete opposite of F77
- Only one value is returned by the function
- Classic opportunity to use pointers!!!!



Exercise: By value

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

int main(void)
{
  double x[3] = {1.0, 1.0, 2.0};
  double mag;

  double unit_vector(double *v); /* Function prototype */

  printf("Initial vector: (%9.2e%9.2e%9.2e )\n",x[0],x[1],x[2]);

  mag = unit_vector(x);

  printf("Magnitude of vector: %9.2e\n",mag);

  printf("Unit vector: (%9.2e%9.2e%9.2e )\n",x[0],x[1],x[2]);

}
```

Exercise: By value (cont.)

```
double unit_vector(double *v)
{
  double magnitude;

  magnitude = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);

  v[0] = v[0]/magnitude;
  v[1] = v[1]/magnitude;
  v[2] = v[2]/magnitude;

  return (magnitude);
}
```

- Type this in and compile using **`cc by_value.c -lm -o by_value`**
- Look at the output and convince yourself that the by reference route works

Arrays (1)

- Arrays are defined using the notation:
  - **`type name[size];`**

  where type is **`int`**, **`float`**, etc.; **`name`** is self-explanatory; and **`size`** is the number of elements in the array
- Examples:
  - **`int a[10];`**
  - **`float radii[5];`**
- In C, arrays start with index 0
  - **`a[0] = 1;`** to **`a[9] = 44;`**

Arrays (2)

- An alternative way of declaring and initialising an array in one go:
  - `int array[] = { 1, 2, 5, 7, 11, 13};`

  will create an array with six elements

- The six integers are located contiguously in memory
  - There is an interesting (and useful) relationship between arrays and pointers

Arrays and Pointers (1)

- We can access the elements of **`array`** using pointers

```
int *ptr;
ptr = &array[0];
```

- **`ptr`** is set to the address of the zeroth element in the array
  - More simply done by **`ptr = array;`**

- We can access the i^{th} element of the array as
  - **`*(ptr+i)`**

Exercise: Pointer2

- Save as pointer2.c, compile and execute it

```
#include <stdio.h>
int array[] = {1, 23, 17, 4, -5, 100};
int *ptr;
int main(void)
{
  int i;
  ptr = &array[0];  /* Pointer points to first element of array */

  printf("\n\n");
  for (i=0; i<6; i++)
    {
      printf("array[%d] = %3d   ", i, array[i]);
      printf("ptr + %d = %3d\n", i, *(ptr+i));
    }
  return 0;
}
```


Exercise: Pointer 2 (cont.)

- Modify the program by changing

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

to

```
ptr = array;
```

and verify that the results are the same

Structures (1)

- A structure is a user-defined data type
- It is a combination of a number of previous declared types
- Usually appears near the start of a program

```
typedef struct
  {
        double real;
        double imag;
  } Complex; /* types usually capitalised */

Complex c1, c2;
```


Structures (2)

- The individual elements of the structure are accessed as follows:

```
double x, y;
x = c1.real – c2.imag;
y = c1.imag + c2.real;
```

- You can define a pointer to a structure in the usual way
  - `complex *c_ptr;`
- Referencing the elements of a structure when using a pointer is achieved thus:
  - `c_ptr->real;`

  which is equivalent to
  - `(*c_ptr).real;`

  …but much easier to use!
- Passing pointers to structures to functions is a good way of passing data to and fro
  - Careful of big structures though!

Exercise: Structure1

```
#include <stdio.h>

int main(void)
{
Struct
   {
     char initial;    /* last name initial      */
     int age;         /* childs age             */
     int grade;       /* childs grade in school */
   } boy,girl;

   boy.initial = 'R';   boy.age = 15;   boy.grade = 75;

   girl.age = boy.age - 1;  girl.grade = 82;  girl.initial = 'H';

   printf("%c is %d years old and got a grade of %d\n",
          girl.initial, girl.age, girl.grade);
   printf("%c is %d years old and got a grade of %d\n",
          boy.initial, boy.age, boy.grade);
}
```


Expressions and Statements

- Arithmetic expressions in C look like F77

```
a = 1.0+(b-c)*d/4.0;      /* Note decimal points for floats.*/
pi = 3.141592654;         /* All statements end with a semicolon. */
area = pi*radius*radius;
```

- Functions which return values can be used in assignments

```
b = myfunc(a);  /* The function myfunc() is defined elsewhere */
x = pow(y,2);   /* pow(a,b) returns x raised to power y */
```

- Functions can also be used without assignments

```
do_stuff();            /* Function do_stuff() takes no arguments */
printf("x = %f\n",x);  /* printf(..) is a standard C library function */
```

Operators (1)

- Arithmetic operators
  - `=` assignment
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `%` modulo
  - `++` increment
  - `--` decrement
- Logical operators
  - `<` less than
  - `<=` less than or equal to
  - `>` greater than
  - `>=` greater than or equal to
  - `==` equal to
  - `!=` not equal to

Operators (2)

- There are some shortcuts in C
  - `i++;` is the same as `i=i+1;`
  - `i++2;` is the same as `i=i+2;`
  - Similarly for `--` (`**` and `//` do NOT exist)
  - `a +=b;` is the same as `a = a+b;`
  - Similarly for `-= *=` and `/=`

Control of Flow (1)

- **if** statements

```
if (logical-expression)
  {statements}
else if (logical-expression)
  statement;
else
  {statements}
```

Note

A single statement can be used or multiple statements enclosed in a {} block.

```
if (q != 1)
  {a = 0; b = 1;}
```

```
if (x < 0.)
  y = x/50.;
else
 {y = x/25.; x=-x;}
```

```
IF (X.LT.0.) THEN
        Y = X/50.
      ELSE
        Y = X/25.
        X=-X
      ENDIF
```


Control of Flow (2)

- **for** loops

```
for (begin ; end ; increment)
  {statements}
```

where:

begin; expression which is executed at beginning of loop

end; logical expression which tests for loop termination

increment; expression which is executed at the end of each loop iteration (usually incrementing a counter)

```
/* Print integers 1-10 and
their squares */
int i, j, n = 10;
for (i = 1 ; i <= n ; i++)
{
  j = i*i;
  printf("%d %d\n",i,j);
}
```

C Equivalent FORTRAN code

```
      INTEGER I,J, N
      N = 10
      DO I = 1,10
      J = I*I
      WRITE (*,*) I,J
      ENDDO
```

Exercise: Control

- Write a C program to step through the first 10 integers
- If the integer is a multiple of 3 then print out the number itself
- If the integer is a multiple of 4 then print out the number divided by one less than itself (in floating arithmetic)
- Otherwise add the number to a running total which should be output at the end

File Handling (1)

- **`printf`** writes formatted data to the console/screen
- **`fprintf`** writes to a file instead
- **`scanf`** and **`fscanf`** are similar functions for reading files

```
#include <stdio.h>
FILE *iofile;
iofile = fopen("test.dat", "w");
fprintf(iofile, "Hello, world\n");
fclose(iofile);
```

```
printf("%d\n", i);
```

BUT

```
scanf("%d", &i);
```

Exercise: Write

- Modify your control program to write the data to an output file called **`control.dat`**
- Save this as **`write.c`** in the usual way

The C Preprocessor (1)

- Commands preceded by # are passed through the C preprocessor (ie before compilation)
  - Header file inclusion
  - Macro definitions
- File inclusion using the directive **`#include`**
  - **`#include <stdio.h>`**
  - **`#include "udf.h"`**
  - **`#include "mystuff.h"`**
  - The files named in quotes must reside in your current directory (except for udf.h which is read automatically by the solver as noted earlier)

The C Preprocessor (2)

- Macro substitutions using **#define name** replacement
  - **#define RADIUS 1.2345**
  - **#define DIAM (3.14159*RADIUS)**
- The preprocessor simply substitutes the characters of name with those of replacement

The C Preprocessor (3)

- Macro substitutions can be made more like simple functions:
  - **#define SQR(A)((A)*(A))**
  - **#define DOT_PROD(A,B)(A[0]*B[0]+A[1]*B[1]**
    +A[2]*B[2])
- **SQR(A)** & **DOT_PROD(A,B)** are replaced by everything after the first closing ")".
- The pattern **A** can be any expression. Note that it is in brackets **(A)** on the definition side of **SQR(A).**
- This avoids errors when **A** is a complex mathematical expression.
- Note also that there doesn't have to be a space after the first closing ")".
- The "\ " is a continuation character used to split long #define lines onto multiple lines.

Exploring C Further

- Some topics not discussed here
  - while and do-while control statements
  - unions
  - recursion
  - many details!
- For more information on C programming, you may consult any general text (there are many available)

A very good set of books are published by O'Reilly, (www.oreilly.com) in particular:

Practical C Programming, 3rd Ed.
by Steve Oualline
O'Reilly, 1997

For the more dedicated, the book by the originators of C can be useful:

The C Programming Language, 2nd Ed.
by Brian Kernighan and Dennis Ritchie
Prentice-Hall, 1988