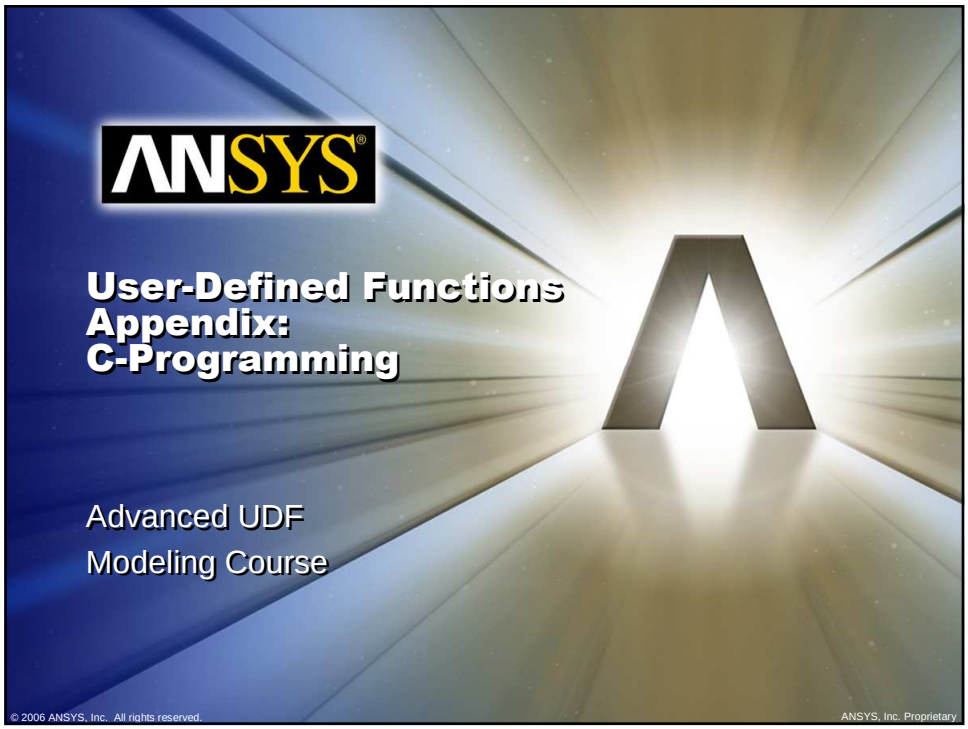

C and UDF

- Many UDFs can be written and used with some limited knowledge of ‘C’
- This presentation will introduce only essential syntax and aspects
- In general, Macros (hook-ups) are available for accessing various locations in the code during the iterations
- FLUENT has a large number of internal macro-s and variables that are not accessible from UDFs (Primarily due to copyright reasons)
- It is advisable to check with your support engineer about the general concept/task that you want to model using UDF
- More elaborate knowledge of ‘C’ helps reducing UDF development and debugging time but need not necessarily provide any extended capabilities

A Brief Introduction to 'C'

- C functions
- C data types
- Pointers and arrays
- Expressions and statements
- C arithmetic and logical operators
- Control flow
- C preprocessor directives

The Basic Form of a C Function

```
/*A simple C function*/          /* Comments are delineated by the character sequence */
                                  /* comments can be placed anywhere in a C listing
                                     use comments liberally to document your UDFs */
#include "udf.h"                  /* A preprocessor directive for including files */
#define PI 3.14159                /* A preprocessor directive for macro substitution */
real a = 1.2345;                  /* Global scope: variables defined outside the function body for use
                                     by all functions which follow the definition */
int my-function(int x)            /* Function declaration (integer type) */
{                                 /* C functions are enclosed by curly braces ({...}) */
                                  /* All C statements must end with a semicolon (;) */
 int y,z;                         /* Local scope: Declare data type for variables y, z variables
                                     defined within the function body are local to the function */
   y = 11;                        /* Set y = 11 */
   z = a*(x+y)*PI;                /* Compute z */
   printf("z = %d",z);            /* Print output to screen */
   return z;                      /* Return integer value */
}                                 /* Right curly brace closes body of function */
```

- If a function is defined with a specific type, it should return a value of the same type (using the return statement)

Compilers

- C compilers include a library of standard math, I/O, and utility functions which can be used in your C code
- Common I/O functions
  - **`scanf (...)`**- formatted read (like FORTRAN READ)
  - **`printf(...)`**- formatted print (like FORTRAN WRITE)
- Common math functions
  - **`sin (x)`** - sine function
  - **`cos (x)`** - cosine function
  - **`exp (x)`** - exponential function
  - **`sqrt(x)`** - square root function
- For the UDF compiler, all of the standard functions are defined in the file udf.h
  - NOTE: *you do not need a copy of udf.h when you compile your UDF; the solver gets this from the Fluent.Inc/fluent6.x/src/ directory*

Comparison with FORTRAN

- C functions are similar to FORTRAN function subroutines

```c
/*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
```

C Data Types (1)

- The UDF interpreter supports standard C data types
  - **int,long** - integer data types
  - **float,double, real** - floating point (real) data types
  - **char** - character data type
- Functions which do not return values are given the type **void**

```
void myfunction(int x) {...} /*No return needed*/
```

- You can convert from one type to another by "casting"
  - a cast is denoted by **(type)** where the type is int, float, etc.

```
int x = 1;
float y = 3.14159;
int z = x+((int) y); /*z = 4*/
```


C Data Types (2)

- C also allows you to create "user-defined" types using **typedef**

```
typedef struct list   {   int a;
                          float b;
                          int c;};
typedef struct list mylist; /* mylist is of type structure list*/
mylist x,y,z; /* x,y,z are "struct list" type */
```

Pointers

- A pointer is a variable which contains the address of another variable
- Pointers are defined using the * notation

```
int *ip; /* a pointer to an integer variable */
```

- We can make a pointer variable point to the address of predefined variable as follows

```
int a=1;
int *ip;
ip = &a; /* &a returns the address of variable a */
printf("content of address pointed to by ip = %d\n", *ip);
```

```
content of address pointed to by ip = 1
```

- Pointers can also point to the beginning of an array (and thus pointers are strongly connected to arrays in C)

Arrays

- Arrays of variables can be defined using the notation **name[size]** where **name** is the variable name and **size** is an integer which defines the number of elements in the array
- Some examples

```
int a[10];
float radii[5];
a[0] = 1;
radii[4] = 3.14159265;
```

- Notes about C arrays
  - The index of C arrays start at 0

Expressions and Statements

- Arithmetic expressions in C look much like FORTRAN

```
a = 1+(b-c)*d/4;
pi = 3.141592654;
area = pi*radius*radius;
```

- Functions which return values can be used in assignment statements

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

- Functions can also be called without assignments

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


Common C Operators

- 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
```

Control Flow - "If "and "If-else" Statements

- '**if**' and '**if-else**' statements

```
if (logical-expression)
  {statements}

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

- Example

```
/*  C code      */
if (q != 1) {a = 0; b = 1;}
if (x < 0.)
  y = x/50.;
else
  y = x/25.;
```

```
C Equivalent FORTRAN code
      IF (X.LT.0.) THEN
        Y = X/50.
      ELSE
        Y = X/25.
      ENDIF
```


Control Flow - "For" Loops

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

where:

begin = *expression, executed at beginning of loop*

end = *logical expression to test for loop termination*

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

Example:

```
/* C code:
   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 = 10
  DO I = 1,10
  J = I*I
  WRITE (*,*) I,J
  ENDDO
```

The C Preprocessor

- The UDF interpreter supports **C** preprocessor directives
- Macro substitutions using: **`#define name replacement`**

```
#define RAD 1.2345
#define Area_Rectangle(x,y) x*y
```

  - The preprocessor simply substitutes the characters of **`name`** with those of **`replacement`**
- File inclusion using the directive **`#include`**

```
#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)

Exploring C Further

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

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