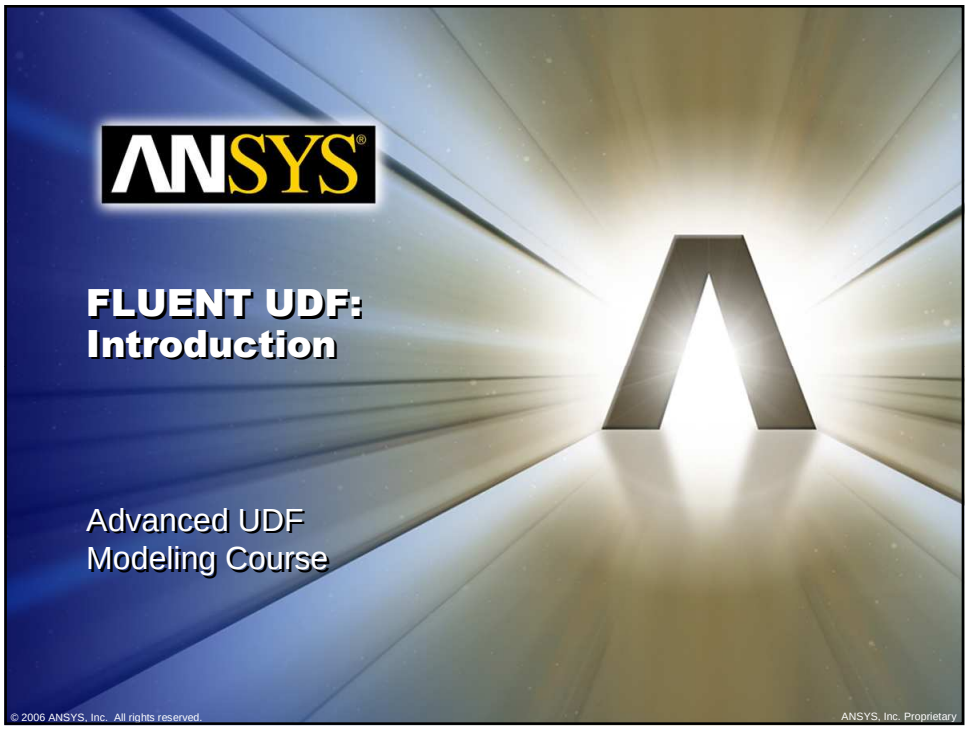


Welcome to Fluent Europe

- Introducing your trainer….
- Domestic issues:
  - Toilets – *all in entrance lobby near reception*
  - Tea, Coffee and Water – *help yourself, in customer dining room*
  - Fire Alarm and Escape Routes *(note alarms are tested at 09:15 Tuesday)*
  - Visitors Badge – *Leave on front reception desk if you go out at lunchtime, and when you leave for the evening*.
  - Smoking – *Outside only.*
  - Taxis – *Please let reception know by lunchtime if you need a taxi for the evening.*

Agenda

09:15 – 09:30	General Introduction to User Defined Functions
09:30 – 10:00	Fluent Data Structure and Macros
10:00 – 10:15	Break
10:15 – 10:45	Interpreted / Compiled UDF
10:45 – 11:45	UDF Hooks - 'DEFINE' Macros
11:45 – 12:30	Tutorial Session
12:30 – 13:30	Lunch
13:30 – 14:00	User Defined Scalars and Memories
14:00 – 14:30	UDF for Discrete Phase Model
14:30 – 15:00	UDF for Multiphase Flows
15:00 – 15:15	Break
15:15 – 16:00	Tutorial-session-2
16:00 – 16:30	UDF for Parallel FLUENT
16:30 – 17:00	Miscellaneous Functions / Macros

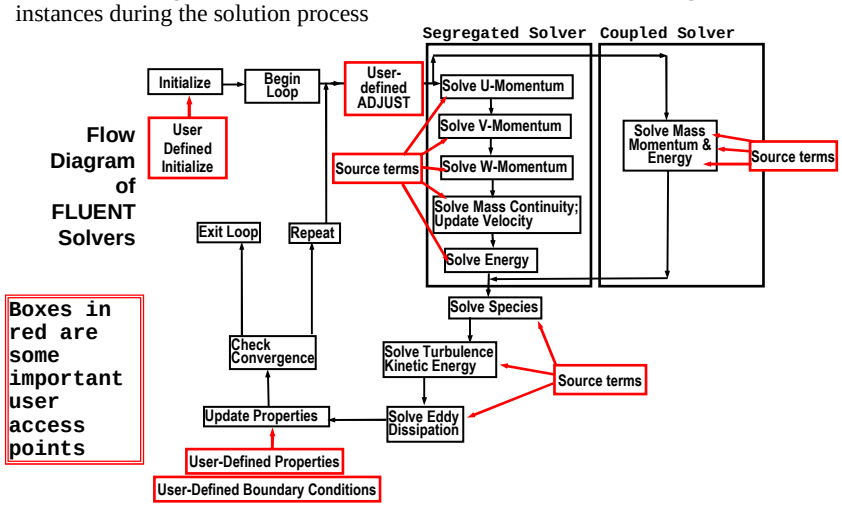
Why Build UDFs?

- Standard interface can not be programmed to anticipate all needs
- Customization of boundary conditions, source terms, reaction rates (volume and surface), properties
- Solution initialization
- Adjust functions (once per iteration)
- Solve for user defined scalars
- Modify model specific parameters
- Many more...

- Limitations
  - Not all solution variables or solver models can be accessed by UDFs
    - Example: Cannot change specific heat (would require additional solver capabilities)

User Access Points to the Solver

- Fluent is so designed that the user can access the solver at some strategic instances during the solution process



User Defined Functions in FLUENT

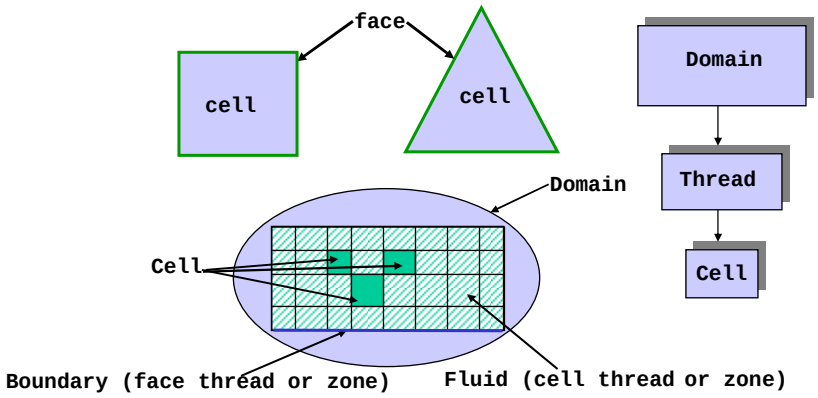
- User Defined Functions are not just any C-functions:
  - User access needs specific "Type" of function calls
  - These Function types or macros are defined in the header file (e.g., udf.h)

- UDF's in FLUENT are available for:
  - **`Profiles`** (Boundary Conditions)
    velocity, temperature, turbulence, species, scalars
  - **`Source terms`** (Fluid and solid zones)
    mass, momentum, energy, species, turbulence, scalars
  - **`Properties`**
    viscosity, conductivity, density, scattering_phase_function (except specific heat)
  - **`Initialization`**
    zone and variable specific initialization
  - **`Global Functions`**
    adjust, read, write, execute_on_demand
  - **`Scalar Functions`**
    unsteady term, flux vector, diffusivity
  - **`Model Specific Functions`**
    reaction rates, discrete phase model, turbulent viscosity

Fluent Data Structure and Macros

Advanced UDF Modeling Course

Data structures in FLUENT



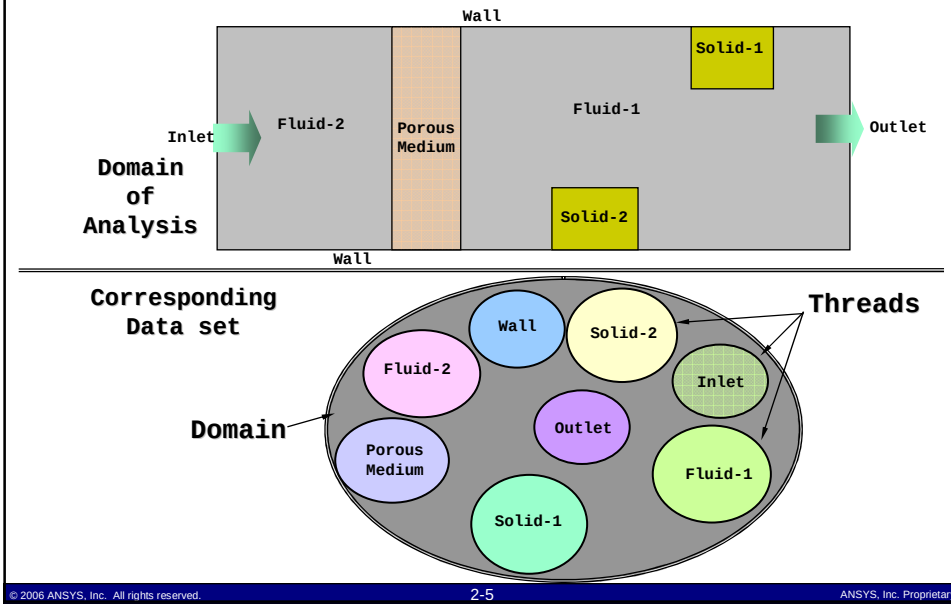
The Domain

- "Domain" is the set of connectivity and hierarchy info for the entire data structure in a given problem. It includes:
  - » all fluid zones ('fluid threads')
  - » all solid zones ('solid threads')
  - » all boundary zones ('boundary threads')
- Cell/face - Computational unit, face is one side. Conservation equations are solved over a cell
- Thread - is the collection of cells or faces; defines a fluid/solid/boundary zone
- FLUENT6 introduces the concept of multi-"domain" for multiphase simulations (singlephase simulations use single domain only)
  - Each phase has its own "Domain-structure"
  - Geometric and common property information are shared among 'sub-domains'
  - Multiphase UDF will be discussed later

The Threads

- A '**Thread**' is a sub-set of the '**Domain**' structure
- Individual '**fluid**', '**solid**' and each '**boundary**' zones are identified as '**zones**' and their datatype is maintained as '**Thread**'
- '**Zone**' and '**Thread**' terms are often used interchangeably
- But **Zone/Thread ID** and **Thread-datatype** are different:
  - Zones are identified at mesh level with an 'integer' **ID** in the **Define → Boundary Condition** panel
  - **Threads,** a Fluent-specific datatype, that store structured information about the mesh, connectivity, models, property, etc. all in one place
  - Users identify zones through the **ID**'s
  - **Zone/Thread-ID** and **Thread**s are correlated through UDF macro's

Domain and Threads



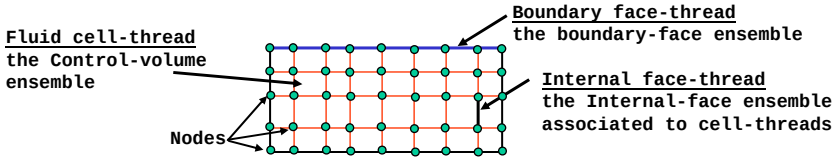
Cell and Face Datatypes

- Control volumes (equivalent of 'FEM:Elements') of fluid and solid zones are called '**`cell`**s' in FLUENT
  - The data structure for the cell zones is typed as '**`cell_t`**' (the cell thread)
  - The data structure for the cell faces is typed as '**`face_t`**' (the face thread)
- A fluid or solid zone is called a cell zone, which can be accessed by using cell threads
- Boundary or internal faces can be accessed by using face threads

Some additional info on Faces

- Each Control volume will have a finite number of faces (4 for tets, 6 for hex and 5 for pyramids, and wedges)
  - Faces on the boundary are also typed '**face_t**'; their ensemble are listed as boundary **face-threads** with the fluid & solid cell-threads under **Define-Boundary_Condition** panel
  - Those faces which are inside the flow-domain and do not share any external boundary are not accessible from GUI (because you do not need them)
  - They can still be accessed from User-Defined-Functions

Cell- & face-Threads



Type	Example	Details
Domain	*d	pointer to the collection of all threads
Thread	*t	pointer to a thread
cell_t	c	cell identifier
face_t	f	face identifier
Node	*node	pointer to a node

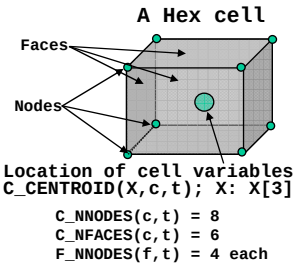
Geometry Macros

The argument **(c,t)** stands for a cell, c of a thread, t

- **C_NNODES(c, t);** Number of nodes in a cell
- **C_NFACES(c, t);** No. of faces in a cell
- **F_NNODES(f, t);** No. of nodes in a face
- **C_CENTROID(x, c, t);** x, y, z-coords of cell centroid
- **F_CENTROID(x, f, t);** x, y, z-coords of face centroid
- **F_AREA(A, f, t);** Area vector of a face;
- **NV_MAG(A);** Area-magnitude
- **C_VOLUME(c, t);** Volume of a cell
- **C_VOLUME_2D(c, t);** Volume of a 2D cell

(Depth is 1m in 2D; 2*π m in axisymmetric)

- **NODE_X(nn);** Node x-coord;
- **NODE_Y(nn);** Node x-coord;
- **NODE_Z(nn);** Node x-coord;



Looping Macros for Geometry

- **thread_loop_c(t, d);** Loop over cell threads
- **thread_loop_f(t, d);** Loop over face threads
- **begin_c_loop(c, t);** } Loop over cells in a cell thread
- **end_c_loop(c, t);** }
- **begin_f_loop** } Loop over faces in a face thread
- **end_f_loop** }
- **f_edge_loop(f, t,en);** Loop over edges in a face thread
- **f_node_loop(f, t,nn);** Loop over nodes in a face thread
- **c_node_loop(c, t,nn);** Loop over nodes in a cell thread
- **c_face_loop(c, t,fn);** Loop over faces in a cell thread

Pointer to a Thread

- Given the integer **ID** of a **thread**, it is possible to retrieve the pointer to that thread -

```
int ID = 1;
Thread *tf = Lookup_Thread(domain, ID);
```

- Conversely, given the pointer to a **thread**, it is possible to retrieve the integer **ID** of that **thread** -

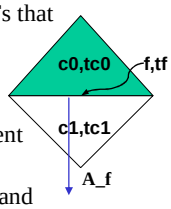
```
int ID = 1;
if (THREAD_ID(tf)==1)...
```

```
int ID = 1;
Thread *tf =
   Lookup_Thread(domain,ID);
begin_f_loop(f, tf)
 {
  F_CENTROID(FC, f, tf);
  printf("x:%f y:%f",FC[0],
  FC[1]);
 }
end_f_loop(f, tf)
```

```
int ID = 1;
thread_loop_f (tf, domain)
 {
  if (THREAD_ID(tf)==1)
   begin_f_loop(f, tf)
    {
     F_CENTROID(FC, f, tf);

     printf("x:%f y:%f",FC[0],FC[1]);
    }
   end_f_loop(f, tf)
 }
```


Cells across a face and Their Threads

- These macros identify the neighboring cells of a face
- This information may be required of some of the more sophisticated UDFs that loop through
  - faces of a boundary thread or
  - a particular cell
- Associated with a given face f, and its thread tf, are potentially two adjacent cells denoted `c0` and `c1` (face normals are always pointing outwardly)
  - If the face is on the boundary of the domain, `c1` is defined as NULL and only `c0` exists
- The following macros return the ID of the cells `c0` and `c1`, as well as the associated threads:




```
 c0 = F_C0(f,tf);        /* returns thread ID for cell c0*/
tc0 = THREAD_T0(tf);     /* returns the cell thread pointer for c0 */
 c1 = F_C1(f,tf);        /*returns thread ID for c1 */
tc1 = THREAD_T1(tf);     /* returns the cell thread pointer for c1 */
```

Cell Variables (1)

- `C_R(c,t)` Density
- `C_P(c,t)` Pressure
- `C_U(c,t)`, `C_V(c,t)`, `C_W(c,t)` Velocity components
- `C_T(c,t)` Temperature
- `C_H(c,t)` Enthalpy
- `C_K(c,t)` Turbulent kinetic energy
- `C_D(c,t)` Turbulent energy dissipation
- `C_YI(c,t,i)` Species mass fraction
- `C_UDSI(c,t,i)` User defined scalar

Cell Variables (2)

- `C_DUDX(c,t)`, `C_DUDY(c,t)`, `C_DUDZ(c,t)`, `C_DVDX(c,t)`, `C_DVDY(c,t)`, `C_DVDZ(c,t)`, `C_DWDX(c,t)`, `C_DWDY(c,t)`, `C_DWDZ(c,t)` Velocity derivatives
- `C_MU_L(c,t)`, `C_MU_T(c,t)`, `C_MU_EFF(c,t)` Viscosities
- `C_DP(c,t)[i]` Pressure derivatives
- `C_D_DENSITY(c,t)[i]` Density derivatives

Cell Variables (3)

- `C_K_L(c,t)`
- `C_K_T(c,t)`
- `C_K_EFF(c,t)`

  } Thermal conductivities

- `C_CP(c,t)` Specific heat
- `C_RGAS(c,t)` Gas constant
- `C_DIFF_L(c,t,i)`
- `C_DIFF_EFF(c,t,i)`

  } Species diffusivity

Face Variables

- `F_P(f,t)` Pressure
- `F_U(f,t)`
- `F_V(f,t)`
- `F_W(f,t)`

  } Velocity components

- `F_T(f,t)` Temperature
- `F_H(f,t)` Enthalpy
- `F_K(f,t)` Turbulent kinetic energy
- `F_D(f,t)` Turbulent energy dissipation
- `F_YI(f,t,i)` Species mass fraction
- `F_UDSI(f,t,i)` User defined scalar
- `F_PROFILE(f,t,i)` Boundary profile storage

UDF Macro-s (Types of UDF)

- UDF's in FLUENT are available for:
  - Boundary conditions : **`Profiles`**
  - Fluid and solid zones : **`Source terms`**
  - Fluid/solid, particle, flow : **`Properties`**
  - UDS unsteady, flux, diffusivity : **`Scalar Functions`**
  - Zone and variable specific initialization : **`Initialization`**
  - Adjust, read/write, execute_on_demand : **`Global Function`**
  - Convective & radiative : **`Wall-heat-flux`**
    (Alternative: profile)
  - Reaction rates, dpm, slip velocity,… : **`Model Specific Functions`**

UDF Macro-s (Types of UDF)

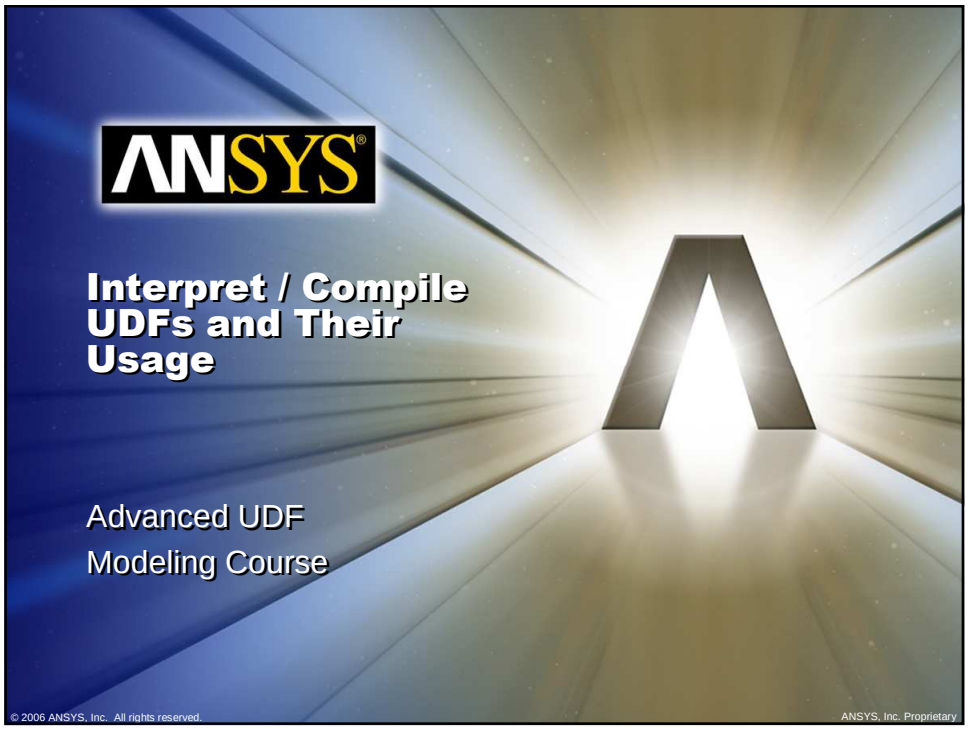
- Available UDF Macro-s :
  - Profiles : **`DEFINE_PROFILE`**
  - Source terms : **`DEFINE_SOURCE`**
  - Properties : **`DEFINE_PROPERTY`**
  - Scalar Functions : **`DEFINE_UNSTEADY`**
    `DEFINE_FLUX`
    `DEFINE_DIFFUSIVITY`
  - Initialization : **`DEFINE_INIT`**
  - Global Functions : **`DEFINE_ADJUST`**
    `DEFINE_ON_DEMAND`
    `DEFINE_RW_FILE`
  - Wall-heat-flux : **`DEFINE_HEAT_FLUX`**
  - Model Specific Functions : **`DEFINE_DPM_…`**
    `DEFINE_SR_RATE`
    `DEFINE_VR_RATE`
    `DEFINE_SCAT_PHASE_FUNC`
    `DEFINE_DRIFT_DIAMETER`
    `DEFINE_SLIP_VELOCITY`

The udf.h File

- The udf-macros are defined in the **'udf.h'** file
- **udf.h** is a fluent header file in the **~/Fluent.Inc/Fluentx.y/src/** directory
- **udf.h** must be included at the top in each and every udf file
  - A file may contain more than one UDF
  - User can use multiple files for UDF
- Any UDF you might write **must** use one of the **'DEFINE_…'** macros from this **udf.h** file

Part of the 'udf.h' file from ~/Fluent.Inc/fluentx.y/src directory

```
#define DEFINE_PROFILE(name, t, i) void name(Thread *t, int i)
#define DEFINE_PROPERTY(name,c,t) real name(cell_t c, Thread *t)
#define DEFINE_SOURCE(name, c, t, dS, i) \
        real name(cell_t c, Thread *t, real dS[], int i)
#define DEFINE_INIT(name, domain) void name(Domain *domain)
#define DEFINE_ADJUST(name, domain) void name(Domain *domain)
#define DEFINE_DIFFUSIVITY(name, c, t, i) \
        real name(cell_t c, Thread *t, int i)
```



How to use the UDF

- First, we need to write and save the C-source file containing the appropriate **`DEFINE_MACRO`** routine(s).
- To use this file, the steps are:
  1: Interpret / Compile the UDF
  2: Start the solver (FLUENT) and read in your case/data files
  3: Assign the UDFs in the BC and/or other panels for the appropriate zones
  4: Set the UDF update frequency in the Iterate panel
  5: Run the calculation as usual
- Note: Values obtained from and returned to the solver by UDFs must be in SI units

Interpreted Vs. Compiled Code

- UDFs can be 'interpreted' on-the-fly using the standard 'GUI'
  - does not need a separate compiler and are architecture-independent
  - It translates the C-source to assembly language
  - Executes the code on line-by-line instantaneously
    - performs slower than compiled UDFs
  - The interpreter resides in the computer's memory
    - involves extra memory usage
- UDFs can be precompiled before invoking in FLUENT
  - Needs a compiler
  - It translates the C-source to machine language (object modules)
  - Needs to follow a standard multi-step procedure (will be discussed later)
  - Creates 'shared libraries' linked with the rest of the solver

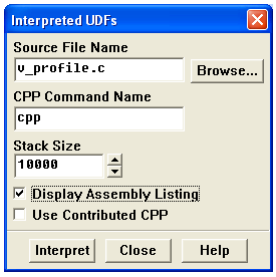
ALL INTERPRETED UDF-S CAN ALSO BE COMPILED
THOUGH THE CONVERSE IS NOT TRUE

Interpreted UDFs

- Interpreter limitations:
  - mixed mode arithmetic,
  - structure references etc.
  - cannot be linked to compiled system or user libraries
  - less powerful than compiled UDFs due to limitations in the C language supported by the interpreter
- In particular, interpreted UDFs cannot contain:
  - non ANSI-C prototypes for syntax
  - declarations of local structures, unions, pointers to functions, and arrays of functions
  - direct structure references
- Interpreted UDFs can indirectly access data stored in a FLUENT structure only via a set of macro-s

Interpreting the UDF (2)

- Define →User Defined Functions →Interpreted…



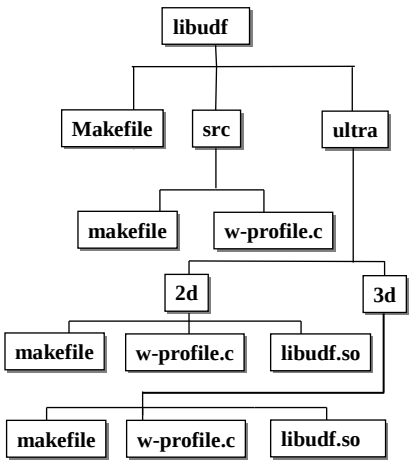
- Click Interpret
- The assembly language code will scroll past window

Listing appearing on Fluent windows:

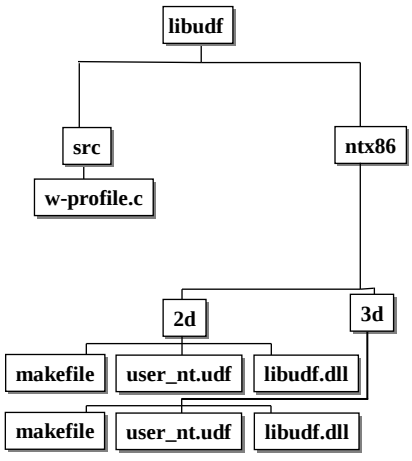
```
w_profile:
    .local.pointer thread (r0)
    .local.int position (r1)
0   .local.end
0    save
    .local.int f (r6)
8    push.int 0
10   save
    .local.int.
    .
    .   Skipping display here
.L1:
132 restore
133 restore
134 ret.v
```


Compiled UDF Directory Structure

Unix Tree

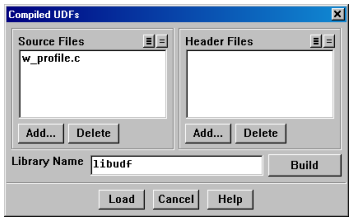


Windows Tree



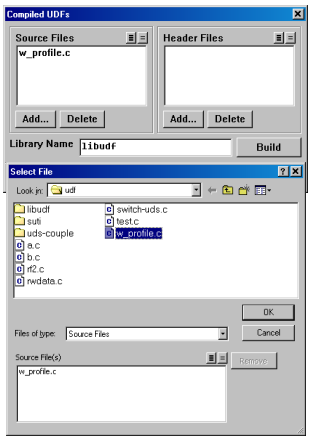
UDF Compilation in F6.2

- To compile UDFs from within Fluent, use:
  - Define→User_Defined→Functions→Compile…
- Placing source routines in your working directory would be sufficient and necessary
- This GUI creates the directory structure below your working directory where you have your case and data files
- This GUI identifies the architecture as well as the version of fluent running and compiles only for the appropriate UDF version (2d/2ddp/3d/3ddp/or any parallel version)



UDF Compilation in F6.2

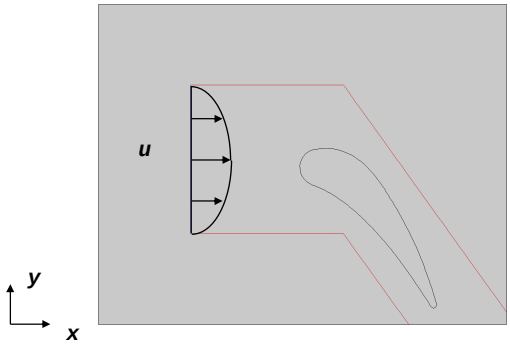
- Define→User_Defined Functions→Compile…
- Click on the "Add" button to browse and add source and header files
- Click on "Build" button to compile and then "Load" to load the library to a case file
- The compilation log appears on the Fluent console window and in a file named log
- To unload a compiled UDF, use
  Define→User_Defined Functions→Manage, select the library, then click Unload button



Using UDFs - Example

- A non-uniform inlet velocity is to be imposed on the 2D turbine vane shown below. The x-velocity variation is to be specified as

$$u(y) = 20\,[\,1 - (y/0.067)^2\,]$$

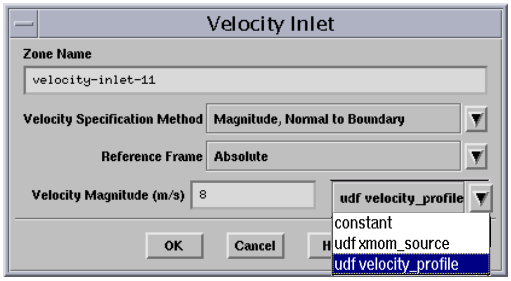


A Source Code Example

```
#include "udf.h"

DEFINE_PROFILE(velocity_profile, thread, position)
{
   real x[3]; /* this will hold the position vector*/
   real y;
   face_t f;

   begin_f_loop(f, thread)
    {
     F_CENTROID(x,f,thread);
     y = x[1];
     F_PROFILE(f, thread, position) = 20.*(1.- y*y /
                                          (.067*.067));
    }
   end_f_loop(f, thread)
 }
```

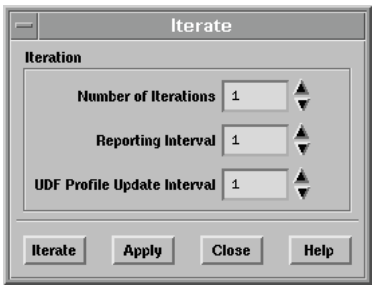
Activating the UDF

- Access the **boundary condition** panel
- Switch from **constant** to the **UDF function** in the **Velocity Magnitude** dropdown list



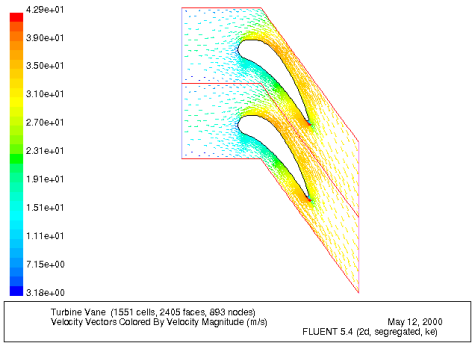
Run the Calculation

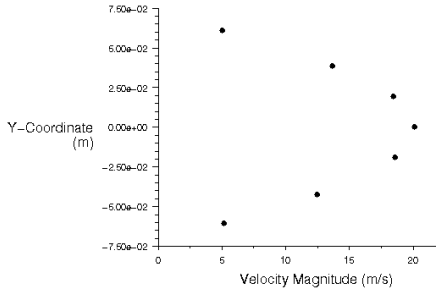
- Run the calculation as usual
- You can change the **UDF Profile Update Interval** in the **Iterate panel** (here it is set to 1)

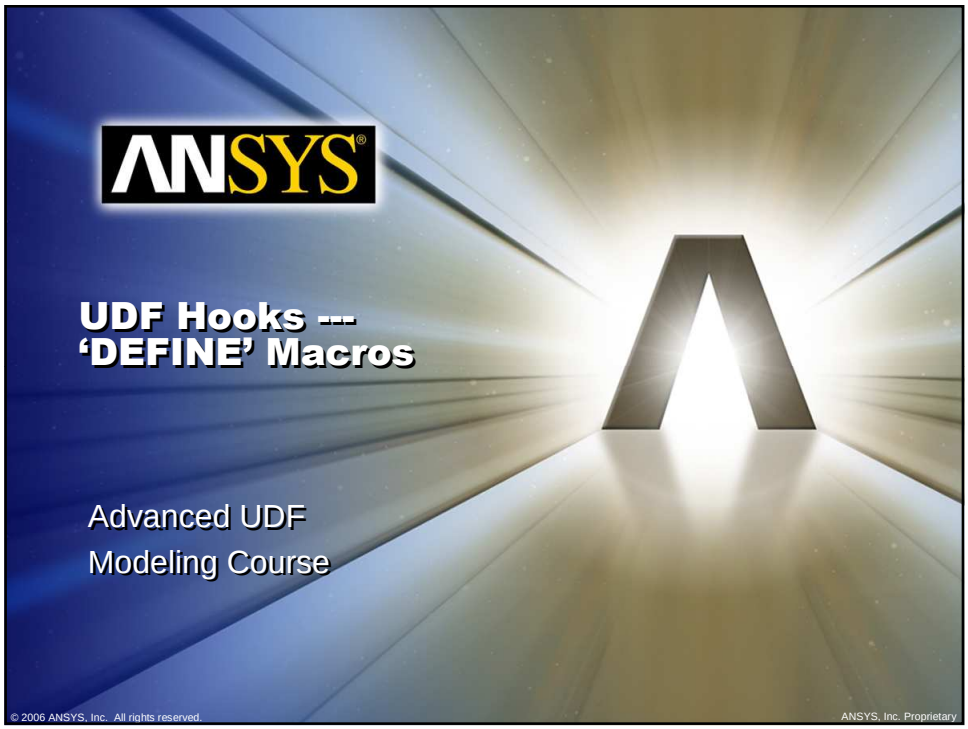


Solution of Example problem

- The figure at right shows velocity field throughout turbine blade passage
- The bottom figure shows the velocity plot at the inlet
- Notice the imposed parabolic profile







Boundary Profiles: DEFINE_PROFILE

- You can use this UDF to specify
  - Wall
    - temperature
    - heat flux, shear stress
  - Inlets
    - velocity
    - temperature
    - turbulence
    - species
    - scalars
- The macro **begin_f_loop** loops over all faces on the selected boundary thread
- The **F_PROFILE** macro applies the value to face, **f** on the **thread**

It's a must!

User specified name

Arguments from the solver to this UDF

```
#include "udf.h"

DEFINE_PROFILE(w_profile, thread, position)
{
   face_t f;
   real b_val;

   begin_f_loop(f, thread)
   {
     b_val = …/* your boundary value*/
     F_PROFILE(f, thread, position) = b_val;
   }
   end_f_loop(f, thread)
}
```

thread : The thread of the boundary to which the profile is attached

position : A solver internal variable (identifies the stack location of the profile in the data stack)

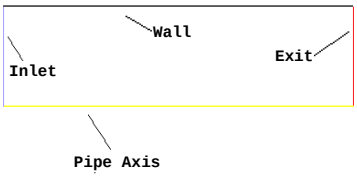
User can rename the variables at will:
DEFINE_PROFILE(my_prof, t, pos)

Example 1: Transient Inlet Velocity

- Pulsatile flow in a tube

  $V_x = V_o + A \sin(\omega t)$

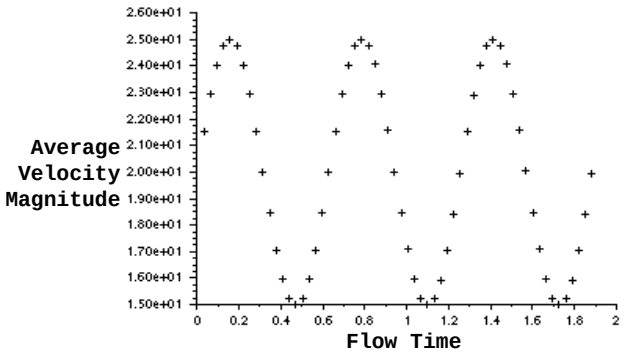
  where V_o = 20 m/s, A = 5 m/s, ω = 10 rad/s
- Boundary condition is applied at inlet




```
#include "udf.h"
DEFINE_PROFILE(unsteady_v, t, pos)
{
  real time, velocity;
  face_t f;
  begin_f_loop(f, t)
    {
     time = RP_Get_Real("flow-time");
     velocity = 20.0 +
        5.0*sin(10.*time);
     F_PROFILE(f, t, pos) = velocity;
    }
  end_f_loop(f, t)
}
```

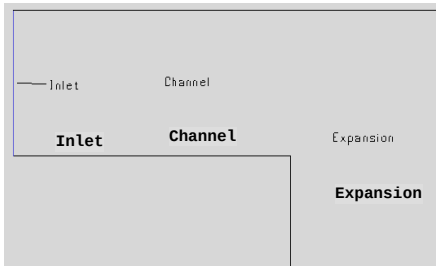

Example 1: Results of Transient Inlet Velocity

- Time history of the average velocity at the pipe exit shows sinusoidal oscillation with a mean of 20 and amplitude of 5.



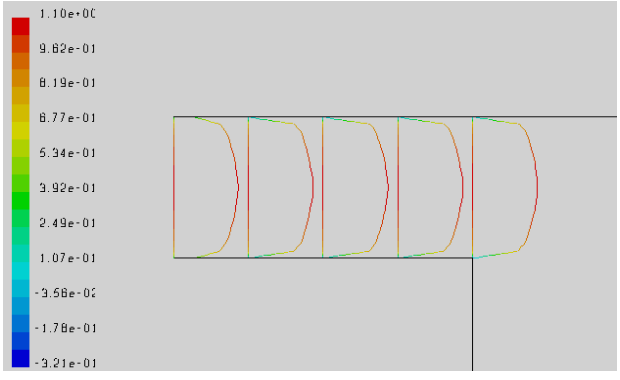
Example 2: Fully Developed Turbulent Inlet

- Profiles for inlet velocity, **k** and **ε** are used to approximate fully developed flow conditions
- Velocity profile follows 1/7 power law
- Turbulent kinetic energy varies linearly from a near-wall peak to a prescribed core-flow value
- Dissipation is prescribed by a mixing-length model
- Used to minimize the domain size and sensitivity to inlet boundary conditions



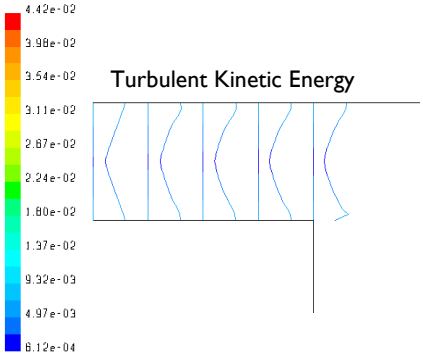
Example 2: Results of Fully Developed Turbulent Inlet

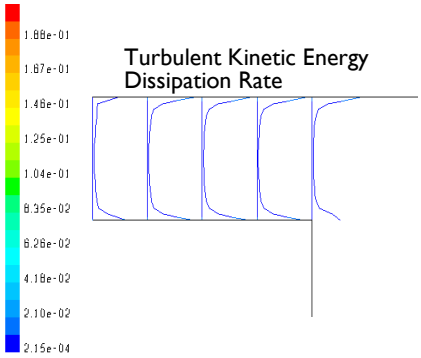
- Axial velocity profile changes little downstream of inlet boundary



Example 2: Results of Fully Developed Inlet

- Turbulence quantities change little downstream of the inlet



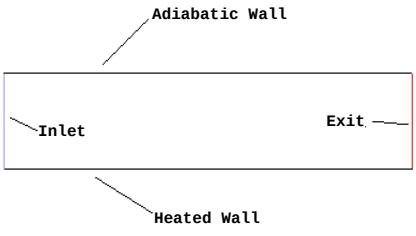


Example 3: Sinusoidal Wall Temperature

- Lower wall temperature varies sinusoidally with x-position according to

$$T_x = 300 + 100 \sin(\pi x/L)$$

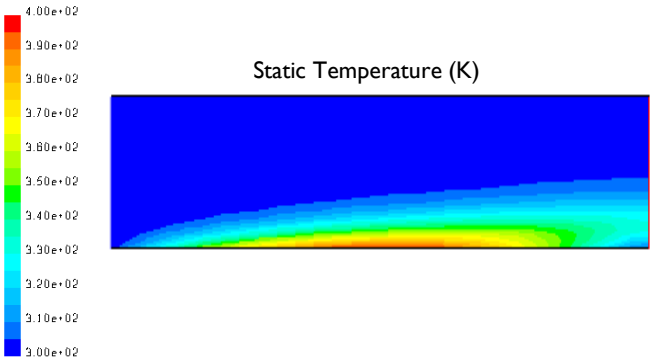
- Inlet fluid enters at 300 K
- Upper wall is insulated



Temperature: `F_PROFILE(f, t, pos) = 300.+100.*sin(PI*x/0.005);`

Example 3: Results of Sinusoidal Wall Temperature

- Wall (and fluid) temperature reaches peak at midlength of channel



Source Terms (1)

- The solvers compute source terms using the "linearized form"

$$S = A + B\,\phi$$

where ϕ is the dependent variable, A is the explicit part of the source term and $B\phi$ is the implicit part

- A recommended linearization is $S = S^* + \left(\frac{\partial S}{\partial \phi}\right)^* (\phi - \phi^*)$ where ϕ is the dependent variable

- FLUENT Solver will automatically determine whether the user-supplied source is enhancing the numerical stability (namely, the diagonal dominance of the system matrix)

Source Terms (2)

- Source term UDFs can be created for the governing equations:
  - continuity
  - momentum
  - k, ε
  - energy
  - species
  - User-defined scalars
- Energy source term UDFs may also be defined for solid zones
- NOTE: The units of all source terms are expressed in terms of the volumetric generation rate. For example, a source term for the continuity equation would have units of ($kg/s/m^3$)

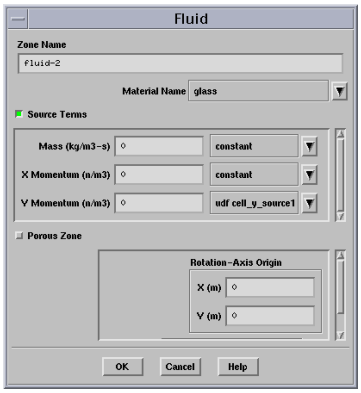
Source Terms (3)

- Solver call this UDF for each cell in the zone
- The solver passes the UDF the cell pointer associated with the cell
- The variable **dS[eqn]** sets up the implicit part of the source term for the equation the source term is used for
- Note that the UDF returns a real value for the explicit part of the source, the implicit part dS[eqn] is returned in a referenced array

```
include "udf.h"

DEFINE_SOURCE(cell_y_source1,
        cell, thread, dS, eqn)
{

real source;

/* S = source + dS[eqn]*phi */

dS[eqn] =  /* expression */

source =   /* expression */

 return source;
}
```

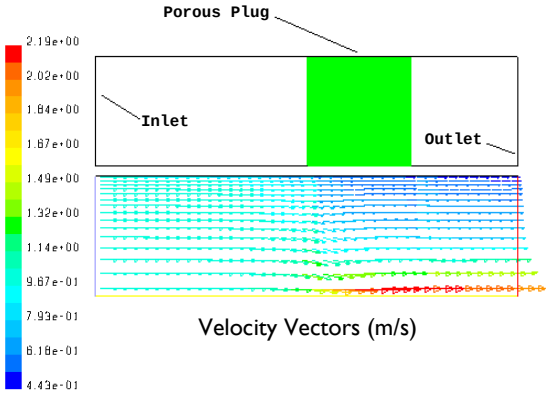
Source Terms (4)

- To activate source terms **`Define⌫Boundary Conditions⌫fluid-1`** and click on **`Source Terms`**



Example 4: Position Dependent Porous Media

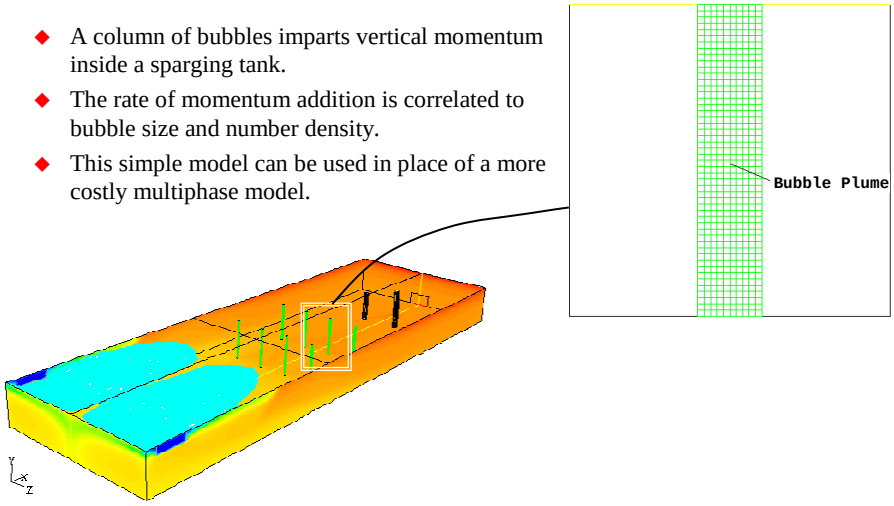
- Channel flow with porous plug
- x-momentum loss is linear in y-position, starting from zero at lower wall
- Fluid flows preferentially near the bottom of the channel



Velocity Vectors (m/s)

Example 5: Bubble Generated Momentum

- A column of bubbles imparts vertical momentum inside a sparging tank.
- The rate of momentum addition is correlated to bubble size and number density.
- This simple model can be used in place of a more costly multiphase model.

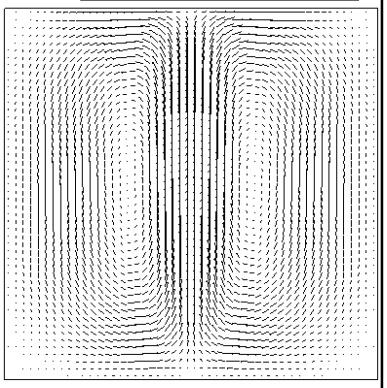


Example 5: Bubble Generated Momentum

- The rising plume of bubbles creates circulation throughout the tank

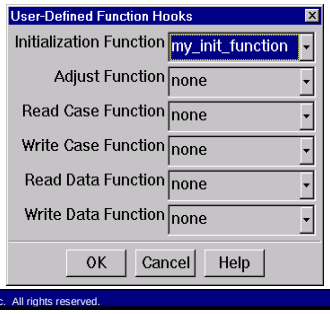
$v=g*r^2*\rho/(3*\mu)$
$Drag=4*\pi*\mu*r*v$
$N = f*h/v$
$Source=N*Drag*100/Volume$

```
#include "udf.h"
real bubbler_vol=0.;/*static variable*/
DEFINE_SOURCE(mom_y_src, c, t, rj, eqn)
{
#define PI 3.14159
#define GRAV 9.81
#define bub_rad 1.e-3
real bub_vel,f_d,bub_freq=5.,bubbler_ht=1.;
float bub_num, source;
cell_t cc;
rj[eqn] = 0.0;
if(bubbler_vol == 0.) /*Bubbler volume*/
{begin_c_loop(cc, t)
 bubbler_vol=bubbler_vol+C_VOLUME(cc,t);
 end_c_loop(cc, t)}
/* Calculate force for single bubble */
bub_vel=GRAV*pow(bub_rad,2.)*C_R(c,t)/
                        (3.*C_MU_L(c,t));
f_d =4.*PI*C_MU_L(c,t)*bub_rad*bub_vel;
bub_num = (bub_freq*bubbler_ht/bub_vel);
source = bub_num*f_d*100./bubbler_vol;
return source;
}
```

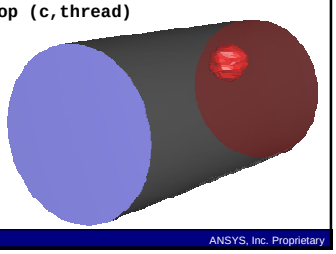


Initialization and Example 6

- Initializes solutions for entire domain, similar to “patching” of values
- Executed once at the beginning of solution process
- Initializatio Function appears under **Define➔ User_Defined➔ Function_hooks…**

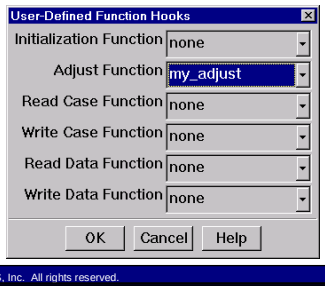



```
#include "udf.h"
DEFINE_INIT(my_init_function,domain)
{
  cell_t c;
  Thread *thread;
  real xc[ND_ND];
  thread_loop_c (thread,domain)
 {
      begin_c_loop (c,thread)
    {
      C_CENTROID(xc,c,thread);
if (sqrt(ND_SUM(pow(xc[0]-0.5,2.),
     pow(xc[1] - 0.5,2.),
    pow(xc[2] - 0.5,2.))) < 0.25)
    C_T(c,thread) = 400.;
else
      C_T(c,thread) = 300.;
    }
      end_c_loop (c,thread)
 }
}
```



Adjust Function and Example 7

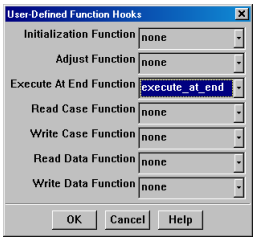
- Function called for every iteration
- Integrate the turbulent dissipation over the whole domain and print it to the text user interface
- Adjust Function appears under **Define➔User_Defined➔ Function_hooks…**




```
DEFINE_ADJUST(my_adjust, domain)
{
/* Integrate dissipation. */
 real sum_diss=0.;
 Thread *t;
 cell_t c;
 thread_loop_c (t,domain)
 {
  begin_c_loop (c,t)
  sum_diss += C_D(c,t)* C_VOLUME(c,t);
  end_c_loop (c,t)
 }
 Message("Volume integral of turbulent
   dissipation : %g\n",sum_diss);
}
```

Execute_at_End Function and Example 8

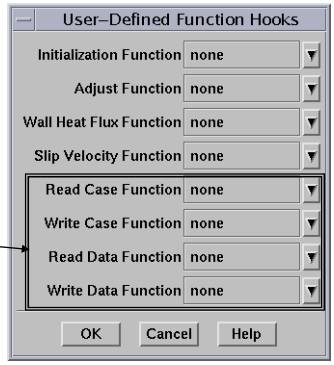
- This is a general purpose macro executed at the end of
  - an iteration in a steady state run, or
  - at the end of a time step in a transient run.
- UDF for integrating turbulent dissipation and printing it to console window at the end of the current iteration or time step
- This Function appears under **Define➔User_Defined➔ Function_hooks…**




```
#include "udf.h"
DEFINE_EXECUTE_AT_END(execute_at_end)
{ Domain *d; Thread *t;
  real sum_diss=0.;
  cell_t c;
  d = Get_Domain(1);
  thread_loop_c (t,d)
  { if (FLUID_THREAD_P(t))
    { begin_c_loop (c,t)
      sum_diss+=C_D(c,t)*C_VOLUME(c,t);
  end_c_loop (c,t)
  }
}
printf("Volume integral of turbulent
   dissipation: %g\n", sum_diss);
   fflush(stdout); }
```


User Defined I/O

- Ability to read/write custom data in case/data files
  - Can save and restore custom variables of any data types (e.g., integer, real, Boolean, structure)
  - Useful to save "***dynamic***" information (e.g., number of occurrences in conditional sampling)
  - Defined using **DEFINE_RW_FILE** macro
  - Selected in the User-Defined Function Hooks panel



User Defined I/O (2)

```
#include "udf.h"
int count = 0;  /* define and initialize static variable
   count */
DEFINE_ADJUST(it_count, domain)
{
  count++;
  printf("count = %d\n",count);
}
DEFINE_RW_FILE(writer, fp)
{
  printf("Writing UDF data to data file...\n");
  fprintf(fp, "%d",count); /* write out count to data
   file */
}
DEFINE_RW_FILE(reader, fp)
{
  printf("Reading UDF data from data file...\n");
  fscanf(fp, "%d",&count); /* read count from data file
   */
}
```


Properties and Example 9

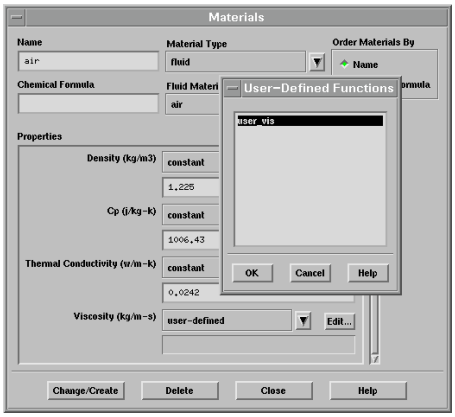
- UDF's can be used to define
  - Viscosity
  - Thermal Conductivity
  - Mass Diffusivity
  - Density
- UDF's cannot be used to define specific heat
- The function is called for every cell in the zone

$$\mu = \begin{cases} 5.5\,10^{-3} & T > 288K \\ 143.2 - 0.49725\,T & 286K \le T \le 288\,K \\ 1 & T < 286K \end{cases}$$

```
#include "udf.h"
DEFINE_PROPERTY(user_vis, cell, thread)
{
  real temp, mu_lam;
  temp = C_T(cell, thread);
  {
   if (temp > 288.)
     mu_lam = 5.5e-3;
   else if (temp >= 286.&& temp<=288.)
     mu_lam = 143.2135 - 0.49725 * temp;
    else
       mu_lam = 1.0;
  }
  return mu_lam;
}
```

Properties (2)

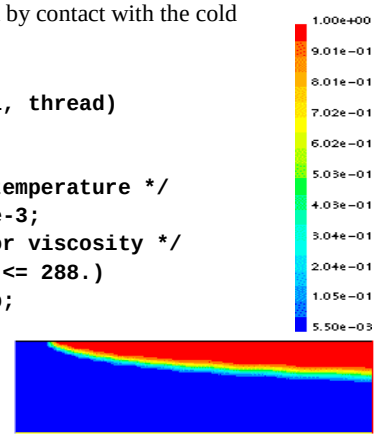
- To activate the UDF, select user-defined from the property drop down list
- When you select the user-defined option, a panel will appear with the names of your UDF's
- Select the name of the appropriate UDF



Example 10: Temperature Dependent Viscosity

- Warm fluid enters the channel flowing from left to right.
- Viscosity increases as the fluid is cooled by contact with the cold upper wall.

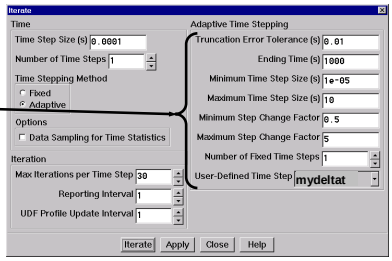
```
#include "udf.h"
DEFINE_PROPERTY(user_vis, cell, thread)
{real temp, mu_lam;
temp = C_T(cell, thread);
{/* Limit viscosity for high temperature */
if (temp > 288.) mu_lam = 5.5e-3;
/* Otherwise, use a profile for viscosity */
else if (temp >= 286. && temp <= 288.)
mu_lam = 143.2135-0.49725*temp;
else
mu_lam = 1.0;
}
return mu_lam;
}
```



Contours of molecular viscosity (kg/ms)

Time Step: DEFINE_DELTAT

- In Fluent, you may use adaptive timesteping based on minimum and maximum values of timesteps as well as other parameters
- Adaptive timestepping is activated by selecting the corresponding radio-button in the **Solve-Iterate** panel for unsteady problems
- **DEFINE_DELTAT** lets the user control the timestep based on any custom logic/algorithm



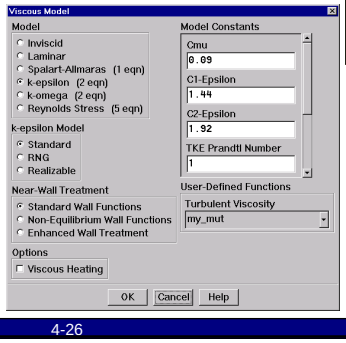

```
#include "udf.h"
DEFINE_DELTAT(mydeltat, domain)
{
real time_step;
real flow_time =
   RP_Get_Real("flow-time");
if (flow_time < 0.5)
   time_step = 0.1;
 else
   time_step = 0.2;
return time_step;
}
```


Turbulent Viscosity: DEFINE_TURBULENT_VISCOSITY

- Any custom relation for the turbulent viscosity formulation can be adopted using this UDF hook
- The variable names for the constants in the standard k-ε model are:
  - C_1 : **M_keC1**
  - C_2 : **M_keC2**
  - C_μ : **M_keCmu**
  - σ_k : **M_keigk**
  - σ_ε : **M_keige**
  - σ_ε : **M_keprt**

```
DEFINE_TURBULENT_VISCOSITY(my_mut,cell,thread)
{
 real mu_t;
 real rho = C_R(cell,thread);
 real k=C_K(cell,thread);
 real epsilon=C_D(cell,thread);
 mut= M_keCmu*rho*SQR(k)/epsilon;
 return mut;
}
```

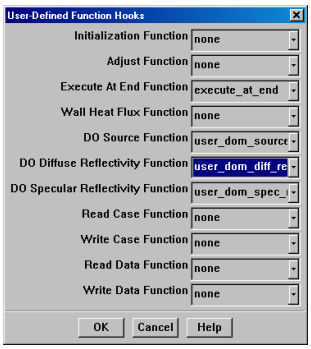
$$\mu_t = C_\mu \rho \frac{k^2}{\varepsilon}$$



Radiation Reflectivity: Discrete Ordinate Model Only

- **Diffused Reflectivity**
- Modify the interfacial reflectivity at diffusely reflecting semi-transparent walls, based on the refractive index
- This function is called for each semi-transparent wall and each band (non-gray DO Model)
- The function can be used to modify interface values of diffuse reflectivity and diffuse transmissivity
- In this example, reflectivity values are not customized: they are just printed

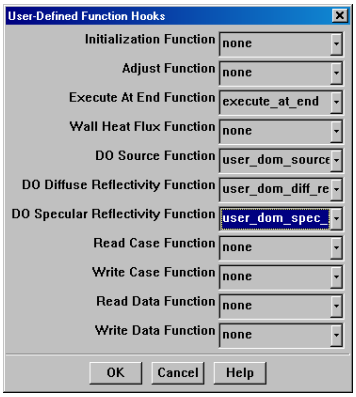
```
#include "udf.h"
DEFINE_DOM_DIFFUSE_REFLECTIVITY
(user_dom_diff_refl, t, nband,
 n_a, n_b, diff_ref_a, diff_tran_a,
 diff_ref_b, diff_tran_b)
{
 printf("diff_ref_a=%f diff_tran_a=%f\n",
        *diff_ref_a, *diff_tran_a);
 printf("diff_ref_b=%f diff_tran_b=%f \n",
        *diff_ref_b, *diff_tran_b);
}
```



Radiation Reflectivity: Discrete Ordinate Model (2)

- **Specular Reflectivity**
- Modify the specular reflectivity and transmitivity at semi-transparent walls, along direction s at a face (f)
- The same UDF is called for all the faces of the semi-transparent wall, for each of the directions

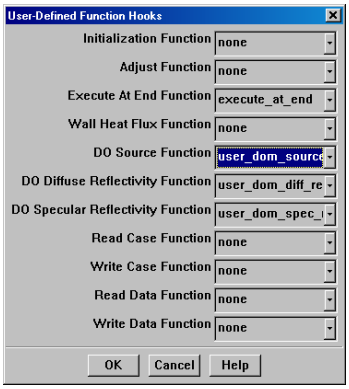
```
#include "udf.h"
DEFINE_DOM_SPECULAR_REFLECTIVITY
       (user_dom_spec_refl, f, t, nband, n_a,
n_b,
              ray_direction, en,
internal_reflection,
              specular_reflectivity,
specular_transmissivity)
{ real angle, cos_theta;
  real PI = 3.141592;
  cos_theta = NV_DOT(ray_direction, en);
  angle = acos(cos_theta);
  if (angle >45 && angle < 60)
     { *specular_reflectivity = 0.3;
       *specular_transmissivity = 0.7;
 } }
```



Emission & Scattering: Discrete Ordinate Source Macro

- Can be used to modify the emission and scattering terms in the radiative transport equation

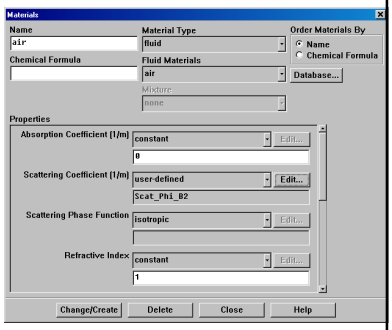
```
#include "udf.h"
DEFINE_DOM_SOURCE(user_dom_source,
c, t, ni, nb, emission,
in_scattering, abs_coeff,
scat_coeff)
{
 *emission *= 1.05;
}
```



Scattering Phase Function: Discrete Ordinate Model

- Define the radiation scattering phase function for the Discrete Ordinates (DO) model
- The function computes two values: the fraction of radiation energy scattered from direction ***i*** to direction ***j***, and the forward scattering factor
- Look at the UDF manual for a complete listing of the UDF for backward and forward scttering phase functions after Jendoubi et al *J. Thermophys. Heat Transfer*, 7(2):213-219, 1993
- This function is loaded as user-defined scattering coefficient in the materials panel

```
#include "udf.h"
DEFINE_SCAT_PHASE_FUNC(Scat_Phi_B2,c,fsf)
{
 real phi=0;
 *fsf = 0;
 phi = 1.0 - 1.2*c + 0.25*(3*c*c-1);
 return (phi);
}
```

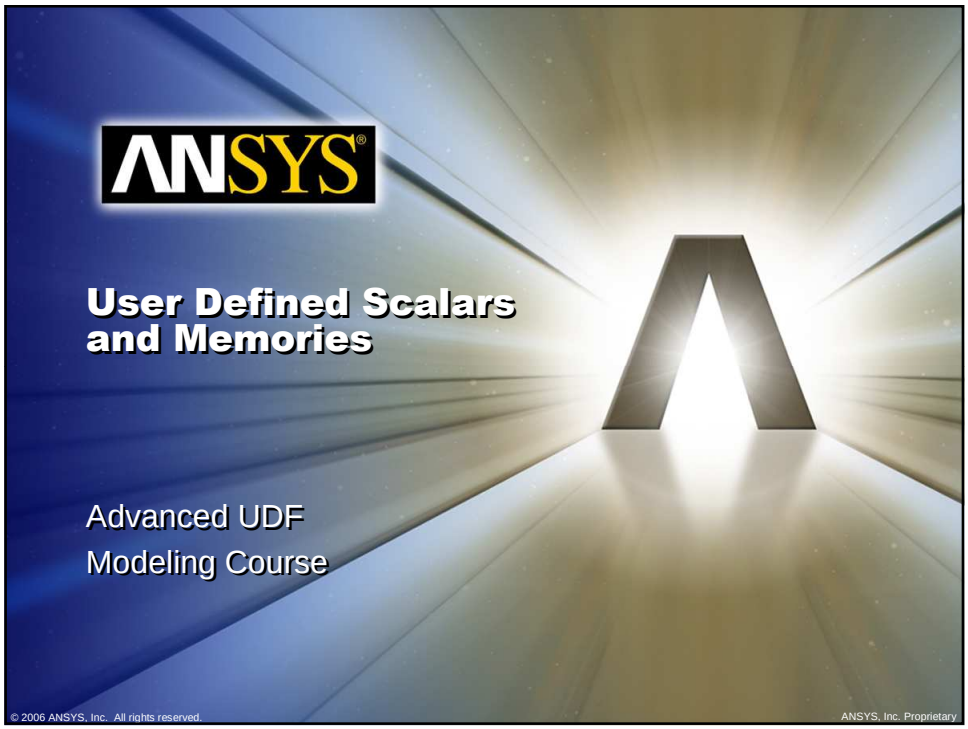


Additional Macros

- There are a number of additional model specific macros
  - You can learn more about these from the UDF manual section 4.3

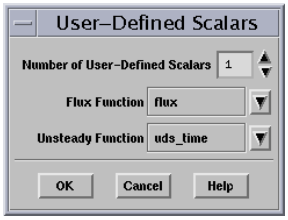
```
DEFINE_CHEM_STEP( name, ifail, n, dt, p, temp, yk)
DEFINE_NET_REACTION_RATE( name, p, temp, yi, rr, jac)
DEFINE_NOX_RATE ( name, c, t, NOx)
DEFINE_PRANDTL_D ( name, c, t)
DEFINE_PR_RATE ( name, c, t, r, mw, ci, p, sf,
                 dif_index,   cat_index, rr)
DEFINE_SR_RATE ( name, f, t, r, my, yi, rr)
DEFINE_VR_RATE ( name, c, t, r, mw, yi, rr, rr_t)
DEFINE_TURB_PREMIX_SOURCE ( name, c, t, turb_flame_speed,
                            source)
```

- Multiphase specific macros will be discussed later



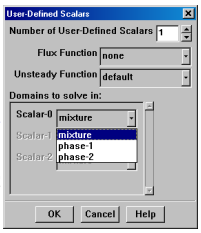
User Defined Scalars (1)

- FLUENT can solve generic transport equations for User Defined Scalars
- The menu is accessed through **`Define→Models→User-Defined Scalars…`**
- User specifies number of User- Defined Scalars and UDF can be used for parts of scalar transport equation :
  - Advective: **`DEFINE_UDS_FLUX`**
  - Unsteady: **`DEFINE_UDS_UNSTEADY`**
  - Diffusivity: **`DEFINE_DIFFUSIVITY`**



$$\frac{\partial}{\partial t}(\rho\phi) + \frac{\partial}{\partial x_j}(\rho u_j \phi) = \frac{\partial}{\partial x_j}\left(D\frac{\partial \phi}{\partial x_j}\right) + S$$

$$\frac{\partial}{\partial t}(\alpha\rho\phi) + \frac{\partial}{\partial x_j}(\alpha\rho u_j \phi) = \frac{\partial}{\partial x_j}\left(D\frac{\partial \phi}{\partial x_j}\right) + S$$

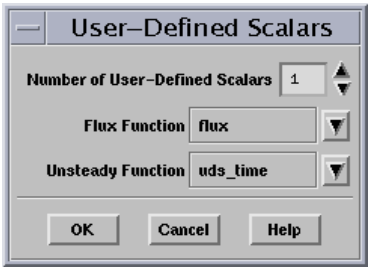


Scalars are phase-specific in multiphase models
Will be discussed later

User Defined Scalars (2)

- User Defined Scalar convective and time derivatives can be modified

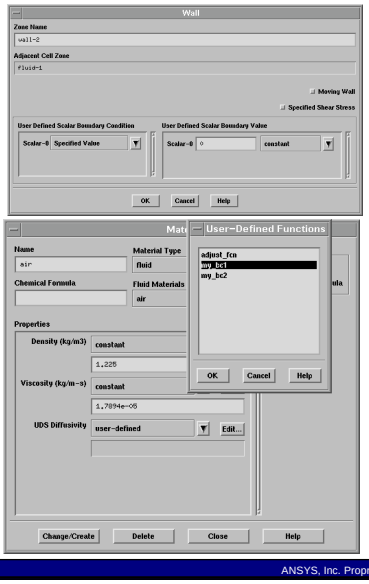
```
DEFINE_UDS_FLUX(flux, f, t, i)
{
 if (i == 0) return 0.;
 if NNULLP(THREAD_STORAGE(t,SV_FLUX))
    return F_FLUX(f,t);
  return 0.;
}
```




```
DEFINE_UDS_UNSTEADY(uns_time, cell,
    thread, i, apu, su)
{
 real physical_dt, vol, rho, phi_old;
  physical_dt = RP_Get_Real("physical-
    time-step");
  vol = C_VOLUME(cell,thread);

  rho = Rhod;
  *apu = -rho*vol /
    physical_dt;/*implicit part*/
  phi_old =
    C_STORAGE_R(cell,thread,SV_UDSI_M1(
    i));
  *su  =
    rho*vol*phi_old/physical_dt;/*expli
    cit part*/
}
```


User Defined Scalars (3)

- The Boundary Conditions for the User Defined Scalar can be specified as **`Specified Flux`** or **`Specified Value`**
- The diffusivity for the User Defined Scalar can be specified through **`Material→user-defined→diffusivity`** panel as a constant or as User-Defined Function



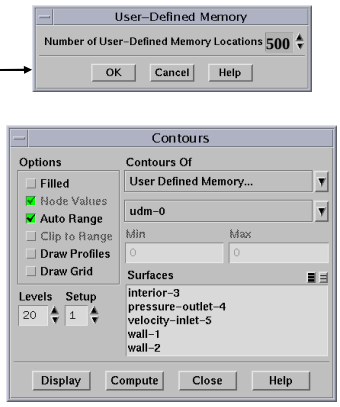
User Defined Scalars (4)

- The User Defined Scalars and their gradients can be used in UDF's

```
DEFINE_ADJUST(adjust_fcn,domain)
{
  Thread *t;
  int nt;
  cell_t c;
  face_t f;
  int ns;
  real p_dis = 0.;
/* Do nothing if gradient isn't allocated yet. */
if (! Data_Valid_P())return;
/* Compute power dissipated. */
thread_loop_c (t, domain)
  if (FLUID_THREAD_P(t))
   {
   begin_c_loop_all (c,t)
     {
C_UDSI(c,t,1) +=
    K_EL*NV_MAG2(C_UDSI_G(c,t,0))*C_VOLUME(c,t);
     }
   end_c_loop_all (c, t)
   }
}
```


User Defined Memory (UDM)

- User-allocated memory
  - Allow users to allocate memory (up to 500 locations) to store and retrieve the values of *field variables* computed by UDF's (for postprocessing and use by other UDFs)
  - Same array dimension and size as any flow variable
  - UDMs are not solved by the solver
  - Number of User-Defined Memory Locations is specified in the User-Defined Memory panel
  - Accessible via macros
    - Cell values: **`C_UDMI(c,t,i)`**
    - Face values: **`F_UDMI(f,t,i)`**
  - Saved to FLUENT data file



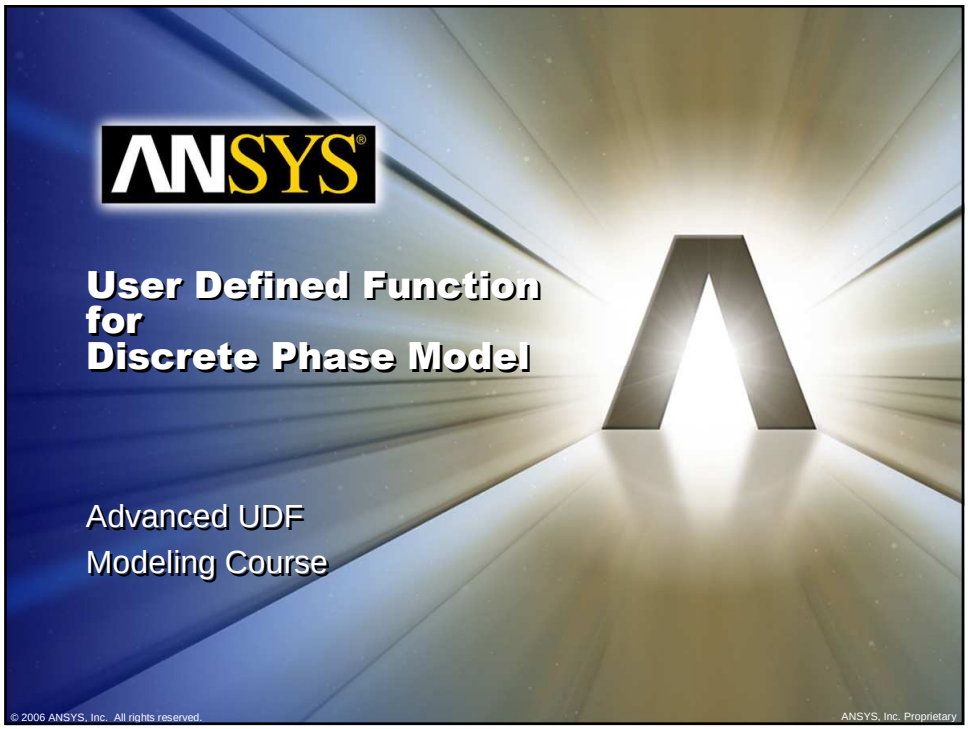
User Defined Memory (2)

```
DEFINE_ON_DEMAND(scaled_temp)
{
  Domain *domain = Get_domain(1);
/* Compute scaled temperature store in user-defined
   memory  */
thread_loop_c(t,domain)
 {
 begin_c_loop(c,t)
  {
    temp = C_T(c,t);
    C_UDMI(c,t,0)=(temp - tmin)/(tmax-tmin);
  }
end_c_loop(c,t)
 }
}
```


Execute on Demand

- This provides a hook to execute any set of calculation or I/O operations at will of the user while the solver is not iterating
- Executed instantaneously when activated by user
- Define→User-Defined →Execute on Demand...


```
extern Domain *domain;
#define SETMIN(a,b)((b)<(a)?(a=b):(a))
#define SETMAX(a,b)((b)>(a)?(a=b):(a))
DEFINE_ON_DEMAND(scaled_temp)
{
thread_loop_c(t,domain)
{
  real tmin=-1.e10, tmax=1.e10;
  /* Compute min & max temperature */
  begin_c_loop(c,t)
    {
     SETMIN(tmin,C_T(c,t));
     SETMAX(tmax,C_T(c,t));
    }
     end_c_loop(c,t)
 }
}
```



DPM Macros (1)

- **`Tracked_particle *p`** DPM Datatype
- DPM tracks particles in Lagrangian frame
- Particle data at current position
  - **`P_DIAM(p)`** Particle diameter
  - **`P_VEL(p)[I]`** Particle Velocity
  - **`P_T(p)`** Particle Temperature
  - **`P_RHO(p)`** Particle density
  - **`P_MASS(p)`** Particle mass
  - **`P_TIME(p)`** Current time for particle
  - **`P_DT(p)`** Particle time step
  - **`P_LF(p)`** Particle liquid fraction
  - **`P_VFF(p)`** Particle volatile fraction

DPM Macros (2)

- Values of particle properties at entry to current cell
  - `P_DIAM0(p)` Diameter
  - `P_VEL0(p)[i]` Velocity
  - `P_T0(p)` Temperature
  - `P_RHO0(p)` Density
  - `P_MASS0(p)` Mass
  - `P_TIME0(p)` Time
  - `P_LF0(p)` Liquid fraction
- Values of particle properties at injection into domain
  - `P_INIT_DIAM(p)` Diameter
  - `P_INIT_MASS(p)` Mass
  - `P_INIT_RHO(p)` Density
  - `P_INIT_TEMP(p)` Temperature
  - `P_INIT_LF(p)` Liquid fraction

DPM Macros (3)

- `P_EVAP_SPECIES_INDEX(p)` Evaporating species index in mixture
- `P_DEVOL_SPECIES_INDEX(p)` Devolatilizing species index in mixture
- `P_OXID_SPECIES_INDEX(p)` Oxidizing species index in mixture
- `P_PROD_SPECIES_INDEX(p)` Combustion product species index in mixture
- `P_CURRENT_LAW(p)` Current law index
- `P_NEXT_LAW(p)` Next particle law index

DPM Macros (4)

- Material Properties for particles
  - `P_MATERIAL(p)` Material pointer for particles
  - `DPM_SWELLING_COEFFI(p)` Swell coefficient for devolatilization
  - `DPM_EMISSIVITY(p)` Particle radiation emissivity
  - `DPM_SCATT_FACTOR(p)` Particle radiation scattering factor
  - `DPM_EVAPORATION_TEMPERATURE(p)` Evaporation temperature
  - `DPM_BOILING_TEMPERATURE(p)` Boiling temperature
  - `DPM_LATENT_HEAT(p)` Latent Heat
  - `DPM_HEAT_OF_PYROLYSIS(p)` Heat of pyrolysis
  - `DPM_HEAT_OF_REACTION(p)` Heat of reaction
  - `DPM_VOLATILE_FRACTION(p)` Volatile fraction
  - `DPM_CHAR_REACTION(p)` Char fraction
  - `DPM_SPECIFIC_HEAT(p, t)` Specific Heat at temperature t

DPM Functions (1)

- The following functions can be modeled:
  - Body force - custom body forces on the particles
  - Drag - user defined drag coefficient between particles and fluid
  - Source Terms - access particle source terms
  - Output - user can modify what is written out to the sampling plane output
  - Erosion - called when particle encounters “reflecting” surface
  - DPM Law - custom laws for particles
  - Scalar Update - allows users to update a scalar every time a particle position is updated
  - Switch - change the criteria for switching between laws

DPM Functions (2)

DEFINE_DPM_BODY_FORCE	Body force
DEFINE_DPM_DRAG	Drag
DEFINE_DPM_SOURCE	Source terms
DEFINE_OUTPUT	Output
DEFINE_DPM_LAW	Custom law
DEFINE_DPM_EROSION	Erosion
DEFINE_DPM_INJECTION_INIT	Initialize injections
DEFINE_DPM_SCALAR_UPDATE	Update scalars
DEFINE_DPM_SWITCH	Switch laws

* Note: the arguments to these functions are described in the UDF manual posted in http://www.fluentusers.com/fluent6/doc/ori/html/udf/main_pre.htm

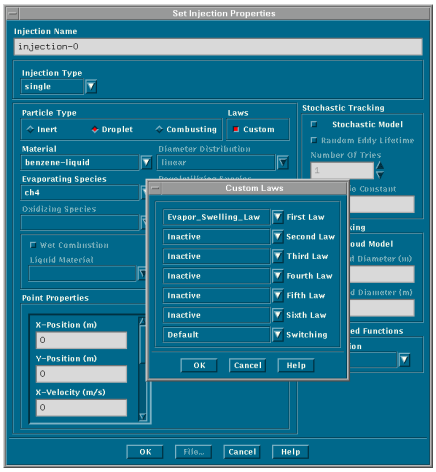
DPM Functions (3)

- The function shown models a custom law
- The parameter p is a pointer to data structure of type Tracked Particle

```
#include "udf.h"
#include "dpm.h"
DEFINE_DPM_LAW(Evapor_Swelling_Law, p, ci)
{
  real swelling_coeff = 1.1;
  /* first, call standard evaporation routine to calculate mass and
   heat transfer */
  Vaporization_Law(p);
  /* compute new particle diameter and density */
  P_DIAM(p) = P_INIT_DIAM(p)*(1. + (swelling_coeff - 1.)*
              (P_INIT_MASS(p) P_MASS(p))/
              (DPM_VOLATILE_FRACTION(p)*P_INIT_MASS(p)));
  P_RHO(p)  = P_MASS(p) / (3.14159*P_DIAM(p)
                           *P_DIAM(p)*P_DIAM(p)/6);
  P_RHO(p)  = MAX(0.1, MIN(1e5, P_RHO(p)));
}
```

DPM Functions (4)

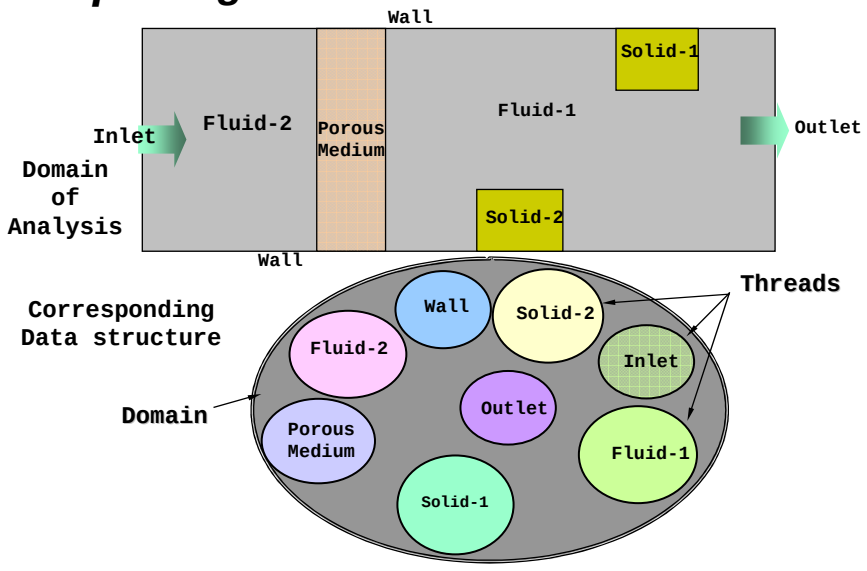
- The law is activated through **Define➔Models➔Dispersed Phase➔Injections…Create**
- The Set Injections Properties panel comes up where Custom is activated under Laws
- This brings up the Custom Laws panel where the user can specify the appropriate law



UDFs for Multiphase Flows

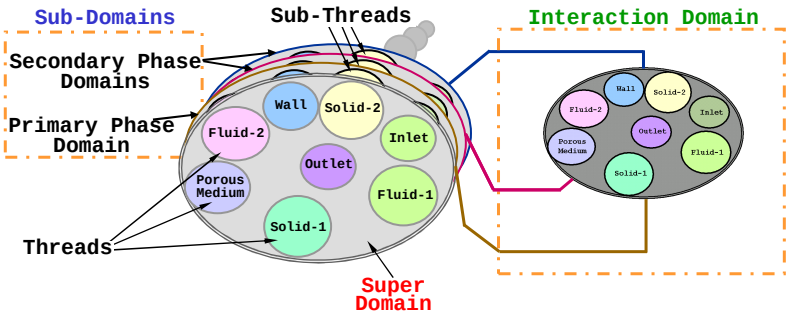
Advanced UDF Modeling Course

Recap: Single Phase Data Structure



Data Structure for Multiphase Models

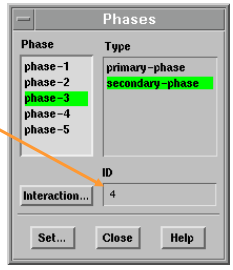
- Data Structure in multiphase models involve multiple domains:
  - Super Domain: This is the top-level domain contains all phase-independent and **mixture** data: geometry, connectivity, property
  - Sub-domains (phase domains): Each phase has a sub-domain that inherits the mixture-specific data and maintains the phase-specific data
  - Interaction Domain: To activate the phase interaction mechanisms



The Threads in Multi-Domains

- The mixture:
  - In single-phase, a mixture represents the sum over all the species
  - In multiphase it represents the sum over all the phases
- This distinction is important
  - Also, the code will later be extended to multiphase multi-component fluids (where, for example, *a phase could be a mixture of species*)
- Thread data structures:
  - Threads must be associated with the super domain and all sub-domains
    - For each cell (or face) thread of the super domain, there is a corresponding cell (or face) thread for each sub-domain
    - Some of the information defined in one thread of the super domain is shared with the corresponding threads of each of the sub-domains
    - Threads associated with the super domain are referred to as super-threads
    - Threads associated with the subdomain are referred to as phase-level threads or sub-threads

The Domain-Ids and The Thread-Ids



- For multiphase models, the domains need to be identified by unique Ids (including Interaction domain)
- **`Domain_ID`** of the super (mixture) domain is always '**`1`**'
- **`Domain_ID`**s are not necessarily ordered sequentially
- Therefore, to access the phase domains, each phase also has a **`phase_domain_index`**:
  - '**`0`**' for the primary phase
  - '**`N-1`**' for the last secondary phase
  - **`phase_domain_index`** is used in UDFs to retrieve phase thread pointers
    - Useful when you want to access data for another phase from an UDF for a particular phase

Domain Looping Macro

- **`sub_domain_loop(subdomain, mixture_domain, phase_index)`**
  - Loops over all phases (sub-domains) in a mixture
  - **`mixture_domain`** is already available
  - **`subdomain`**, **`phase_index`** are defined locally, initialized within the macro
- An Example for the loop, **`Domain_ID`** and the **`phase_domain_index`**:

```
DEFINE_ADJUST(print_id, mix_domain)
   {
   Domain *s_d;/*subdomain pointer, locally defined*/
   int p_d; /* loop counter for phase_domain_index, locally defined*/
   int p_d_id; /* mix_domain is available*/
   sub_domain_loop(s_d, mix_domain, p_d)
    {
      p_d_id = DOMAIN_ID(s_d);
      Message("the phase domain id = %d and the phase
               domain index = %d\n", p_d_id, p_d);
    }
   }
```

Thread Looping Macro

- **sub_thread_loop(subthread,mixture_thread,phase_index)**
  - Loops over all threads in a mixture
  - **mixture_thread** is already available
  - **subthread** & **phase_index** are defined locally, initialized within the macro

An Example:

```
/*compute bulk density of mixture and store it in a UDM*/
DEFINE_ADJUST(calc_den, mix_domain)
{ Thread *mix_thread;
  thread_loop_c(mix_thread,mix_domain)
  { cell_t c;
begin_c_loop(c,mix_thread)
    { Thread *s_t;
      int p_d_i;
      C_UDMI(c,mix_thread,0) = 0.;
      sub_thread_loop(s_t, mix_thread, p_d_i)
      C_UDMI(c,mix_thread,0) += C_VOF(c,s_t)*C_R(c,s_t);
    end_c_loop(c,mix_thread)
    }}}
```


Other Looping Macros

- **mp_thread_loop_c(cell_thread, mixture_domain, pt)**
  - **cell_thread** is a pointer to mixture thread in the **mixture_domain**
  - **mixture_domain** is already assumed to be available
  - **pt** is an array of thread pointers
- **mp_thread_loop_f(face_threads, mixture_domain, pt)**
  - **face_threads** is a pointer to face thread in the **mixture_domain**
  - **mixture_domain** is already assumed to be available
  - **pt** is an array of thread pointers pointing to the phase-level threads

An Example:

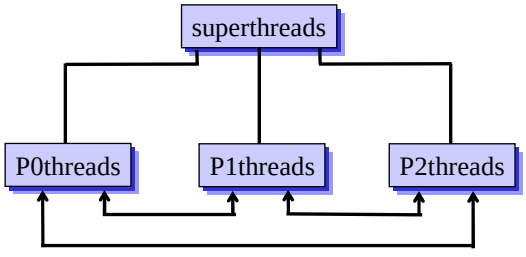
```
DEFINE_ADJUST(print_vof, mix_domain)
{ Thread *mix_thread;
  Thread **pt;
  mp_thread_loop_c(mix_thread, mix_domain, pt)
  {cell_t c;
   begin_c_loop(c, mix_thread)
       Message("cell volume fraction = %f\n",C_VOF(c,pt[0]));
   end_c_loop(c, mix_thread)}}
```

Access the Right Thread / Domain

- While writing UDF's, it is important that the right thread / domain is accessed
  - **`C_R(cell,thread)`** will return
    - The mixture density if **`thread`** is the mixture thread or
    - The phase densities if it is the **`phase thread`**
- In general the type of **`'DEFINE'`** macro determines which thread or domain (mixture or phase) gets passed to your UDF
  - **`DEFINE_INIT`** and **`DEFINE_ADJUST`** functions always get passed the domain structure associated with the super domain
  - **`DEFINE_ON_DEMAND`** functions are not passed any domain structures
  - If your UDF is not explicitly passed the pointer to the thread or domain required, then you can use a multiphase-specific utility macro to retrieve it

Superthreads and Phasethreads

- Each Thread is also in a hierarchy that matches that of the domains
- The "**superthreads**" are where the "mixture" of the phases is stored and so are often called the **"mixture threads"**
- Shared values such as the cell's geometry data are stored in the superthread
- Each Phase has its own set of threads known as a **"subthreads"** or "**phase threads**"



Phase Subthreads

Phase Thread Interactions

Access Variables External to a UDF

- `Get_Domain(Domain-ID)`
  - Usage: `Domain *domain=Get_Domain(n);`

    '**n**' is the **`Domain-ID`**, as appear in Define-Phase GUI. It is always '1' for the mixture domain
- `DOMAIN_ID(domain)`
  - Usage: `int domain_id = Domain_ID(subdomain);`

    '**`subdomain`**' is the pointer to a phase-level domain; **`domain_id`** upon return is the same integer ID displayed in the GUI under Define-Phases panel
- `DOMAIN_SUB_DOMAIN(mixture_domain,ph_domain_index)`
  - Usage:
    ```
    Domain *mixture_domain;
    Domain *subdomain = DOMAIN_SUB_DOMAIN
           (mixture_domain, phase_domain_index);
    ```
    returns the phase pointer subdomain for the **`phase_domain_index`**

Access Variables External to a UDF (2)

- `THREAD_SUB_THREAD(mixture_thread,ph_domain_index)`
  - Usage:
    ```
    int ph_d_index = 0; /* primary phase index is 0 */
    Thread *mix_th;/* mixture-level thread pointer */
    Thread *subth=THREAD_SUB_THREAD(mix_th, ph_d_index);
    ```
    returns the phase-level thread pointer for the given **`ph_d_index`**
- `THREAD_SUB_THREADS(mixture_thread)`
  - Usage:
    ```
    Thread *mixture_thread;
    Thread **pt; /* initialize pt: pointer array */
    pt = THREAD_SUB_THREADS(mixture_thread)
    ```
    returns the pointer array, **`pt`**, whose elements contain pointers to phase-level threads (subthreads)

Access Variables External to a UDF (3)

- **THREAD_SUPER_THREAD(subthread)**
  - Usage:

```
Thread *subthread; /*pointer to a phase thread within the mixture*/
Thread *mix_thread=THREAD_SUPER_THREAD(subthread)
```

    Given a phase thread pointer, it returns the super-thread (mixture-thread) pointer
- **DOMAIN_SUPER_DOMAIN(subdomain)**
  - Usage:

```
Domain *subdomain; /*pointer to a phase domain within the mixture*/
Domain *mixture_domain = DOMAIN_SUPER_DOMAIN(subdomain)
```

    It returns the mixture domain pointer

Access Variables External to a UDF (4)

- **PHASE_DOMAIN_INDEX(subdomain)**
  - Usage:

```
Domain *subdomain; /*points to a phase domain within the mixture*/
int phase_domain_index = PHASE_DOMAIN_INDEX(subdomain)
```

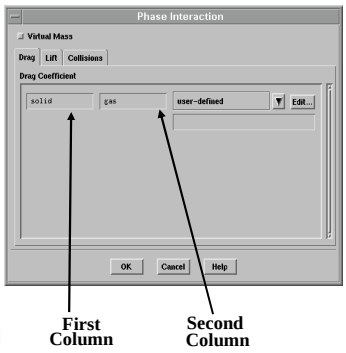
  returns the phase domain index for the phase domain (subdomain) pointer;
  It is an integer that starts with '0' for the primary phase and is incremented by one for each secondary phase

- **THREAD_DOMAIN(thread)**
  - Usage:

```
Thread *subthread; /*points to a phase thread within the mixture*/
Thread *mix_thread=THREAD_SUPER_THREAD(subthread)
Domain *subd=THREAD_DOMAIN(subthread); /*points to a phase domain*/
Domain *mixd=THREAD_DOMAIN(subthread); /*points to mixture domain*/
```

    returns domain pointer for the **thread**

Exchange Macros(1)



- **`DEFINE_EXCHANGE_PROPERTY ( name, c, mixture_thread, second_column_phase_index, first_column_phase_index)`**
  - This macro is used to specify custom drag & lift coefficients for the Eulerian multiphase model
  - **`mixture_thread`** points to the mixture thread
  - **`c`** is the index of a cell on the **`mixture_thread`**
  - **`first_column_phase_index`** and **`second_column_phase_index`** are integer identifiers corresponding to the pair of phases in your multiphase flow
  - The identifiers correspond to the phases that are selected in the Phase-Interaction panel in the GUI
  - The UDF returns the real value of the lift or drag coefficient

Exchange Macros(2)

- **`DEFINE_VECTOR_EXCHANGE_PROPERTY(name, c,mixture_thread, second_column_phase_index,first_column_phase_index, vector_result)`**
  - This macro is used to specify custom slip velocities for multiphase Mixture model
  - **`mixture_thread`** points to the mixture thread
  - **`c`** is the index of a cell on the **`mixture_thread`**
  - **`first_column_phase_index`** and **`second_column_phase_index`** are integer identifiers corresponding to the pair of phases in your multiphase flow
  - The identifiers correspond to the phases that are selected in the **`Phase-Interaction`** panel in the GUI
  - The UDF is passed the real pointer to the slip velocity vector **`vector_result`**, and it will need to set the components of the slip velocity vector

Exchange Macros(3)

An Example:

```
#include "udf.h"
#include "sg_mphase.h"
DEFINE_VECTOR_EXCHANGE_PROPERTY(custom_slip, c, mixture_thread,
second_column_phase_index, first_column_phase_index, vector_result)
{  real grav[2] = {0., -9.81};
   real K = 5.e4;
   real pgrad_x, pgrad_y;
   Thread *pt, *st;/* thread pointers for primary & secondary phases*/

   pt = THREAD_SUB_THREAD(mixture_thread, second_column_phase_index);
   st = THREAD_SUB_THREAD(mixture_thread, first_column_phase_index);
/* Now the threads are known for primary (0) & secondary(1) phases */
   pgrad_x = C_DP(c, mixture_thread)[0];
   pgrad_y = C_DP(c, mixture_thread)[1];
  vector_result[0] = -(pgrad_x/K)+(((C_R(c,st)-C_R(c,pt))/K)*grav[0]);
  vector_result[1] = -(pgrad_y/K)+(((C_R(c,st)-C_R(c,pt))/K)*grav[1]);
}
```


Cavitation Macros

- **`DEFINE_CAVITATION_RATE (name,c,t,p,rhoV,rhoL,vofV, p_v,n_b,mdot)`**
- You can use this macro to model the creation of vapor due to pressure tension in a multiphase flow
  - It is applied in the **`User-Defined-Function-Hooks`**→**`Cavitation-Mass-Rate-Function`** panel
  - **`t`** is a pointer to the mixture-level thread
  - **`c`** is the index of a cell on the thread pointed to by **`t`**
  - The remaining arguments are real pointers to the following data:
    - Shared pressure (**`p`**), vapor density (**`rhoV`**), liquid density (**`rhoL`**), vapor volume fraction (**`vofV`**), vaporization pressure (**`p_v`**), number of bubbles per unit volume (**`n_b`**), and rate of vapor formation (**`mdot`**)
  - The UDF sets the value referenced by the real pointer **`mdot`**, to the cavitation rate

Miscellaneous: Multiphase Macros

- Phase diameter
  - `C_PHASE_DIAMETER(c,phase_thread)`
- Phase Volume-fraction
  - `C_VOF(c,phase_thread)`
- Phase velocity gradients
  - `C_U_G(c, phase_thread)`
  - `C_V_G(c, phase_thread)`
  - `C_W_G(c, phase_thread)`

Miscellaneous : Multiphase Macros

- Phase volume fraction gradients: `C_VOF_G(c,phase_thread)`
  - Memory needs to be allocated and gradients need to be explicitly calculated

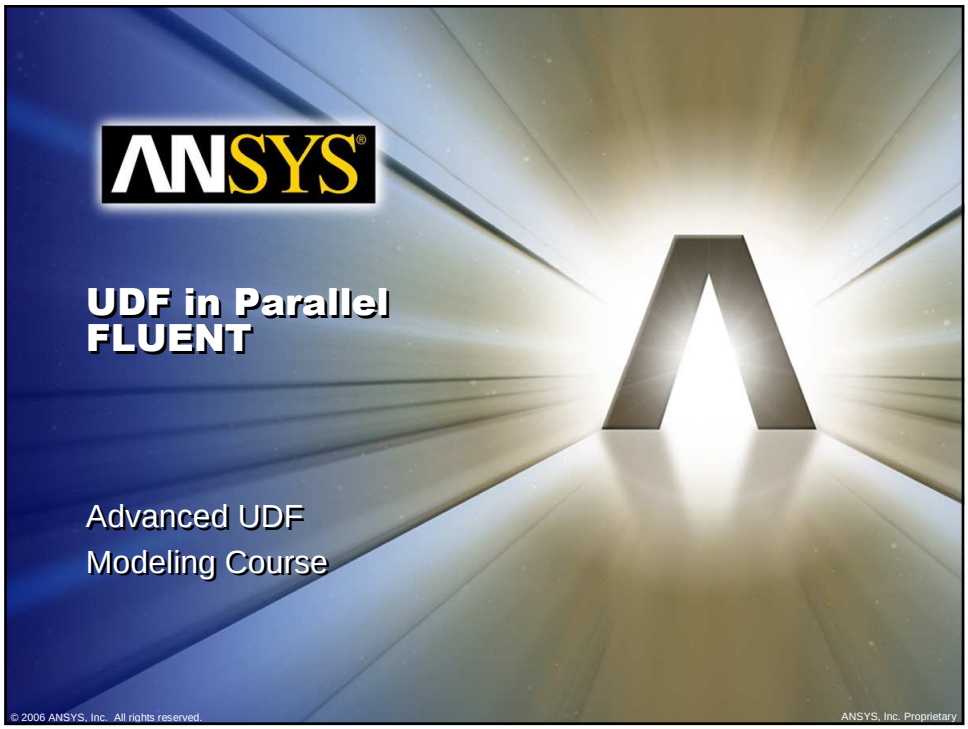
```
Domain   *pDomain = DOMAIN_SUB_DOMAIN(domain,P_PHASE);
Alloc_Storage_Vars   (pDomain,SV_VOF_RG,SV_VOF_G,SV_NULL);
Scalar_Reconstruction(pDomain,SV_VOF,-1,SV_VOF_RG,  NULL);
Scalar_Derivatives   (pDomain,SV_VOF,-1,SV_VOF_G,SV_VOF_RG,
                      Vof_Deriv_Accumulate);
```

Miscellaneous: Multiphase Macros

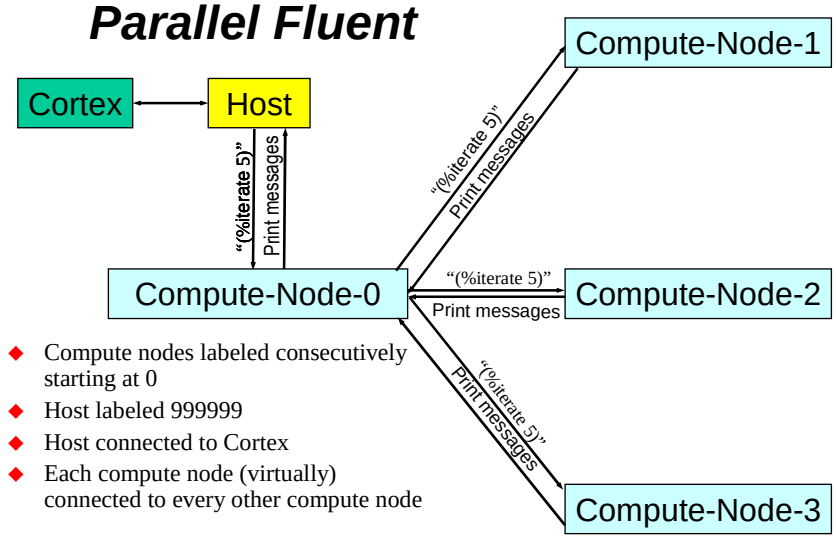
- Check if a given thread is a “super” or “sub” thread

`THREAD_SUPER_THREAD(thread)` is **`NULL`** for mixture thread, and not **`NULL`** for phase threads

  - mixture : **`if ( NULLP (THREAD_SUPER_THREAD(thread)))`**
  - phase : **`if (!NULLP (THREAD_SUPER_THREAD(thread)))`**

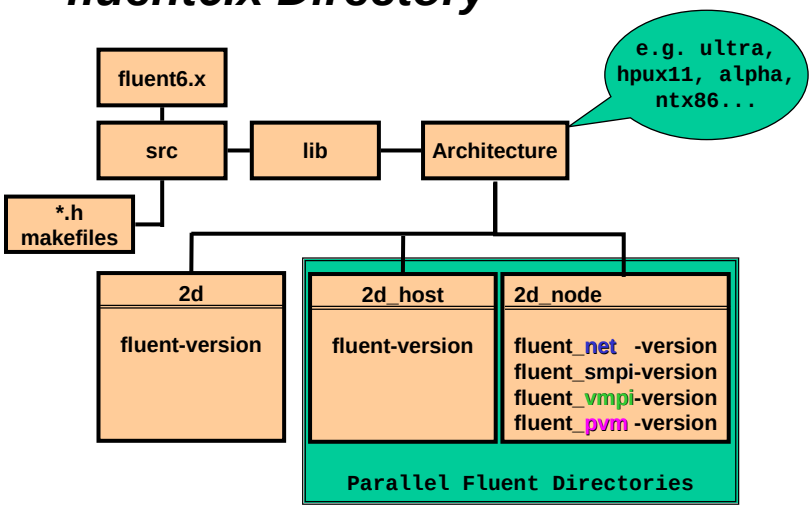


Parallel Fluent



- Compute nodes labeled consecutively starting at 0
- Host labeled 999999
- Host connected to Cortex
- Each compute node (virtually) connected to every other compute node

fluent6.x Directory



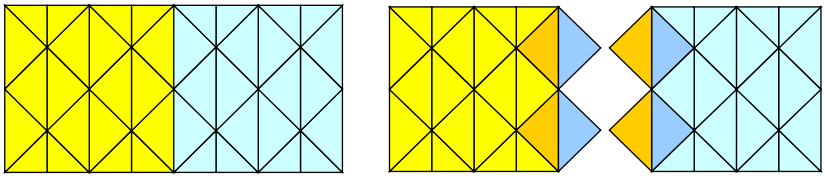
Intro to Compiler Directives

- "**#if**" is a compiler directive (similar to "**#define**")
- A "**#endif**" is used to close a "**#if**

```
#if RP_NODE   /* Compute-Node */
#if RP_HOST   /* Host */
#if PARALLEL  /* Equivalent to #if RP_HOST||RP_NODE*/
#if !PARALLEL /* Serial */
#if RP_HOST
     Message("I'm the Host process \n");
#endif
#if RP_NODE
     Message("I'm the Node process number:%d \n", myid);
#endif
```

Partitioning (1)

- Domain Decomposition Technique: Splits the domain across Compute Nodes
- Because Fluent's algorithms expect a cell to be on both sides of an interior face, copies of the neighboring partition's cells are kept on each Node
- Compute Node 0 has copies of the cells on the other side of all partition faces and Compute Node 1 has corresponding cell copies from Node 0

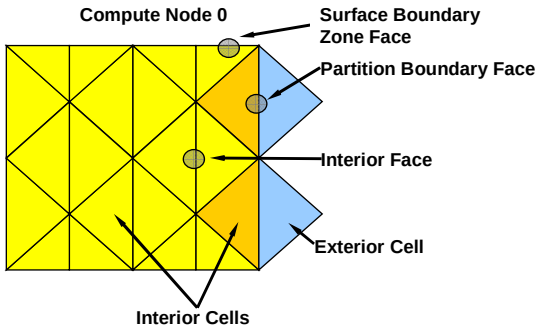


Domain Decomposition

Distribution across Compute Nodes

Partitioning (2)

The main cells of the partition are designated "Interior" cells and the additional copied cells from other Compute Nodes are designated "Exterior" cells
The Partition Boundary Faces are a special type of Interior face



Partitioned Thread Loop (1)

```
begin_c_loop(c,t)
  {
  }
end_c_loop(c,t)
```

- In parallel use, above loop construct loops through the Exterior cells too
- Use `begin_c_loop_int(c,t)` in all UDFs that are to be parallelized:

```
begin_c_loop_int(c,t)
  {
  }
end_c_loop_int(c,t)
```

- This loop excludes the exterior cells to replicate serial `begin_c_loop(c,t)`
- Another loop construct loops through the exterior cells only :

```
begin_c_loop_ext(c,t)
  {
  }
end_c_loop_ext(c,t)
```

- It is rarely used in UDFs and does nothing if compiled in serial version

Partitioned Thread Loop (2)

- Similar loops exist for faces:

```
begin_f_loop_all(f,t)
{…}
end_f_loop_all(f,t)

begin_f_loop_int(f,t)
{…}
end_f_loop_int(f,t)
```

- But you can simply use the standard loop and check to see if the face is "allocated" to this Thread using:

```
begin_f_loop (f,t)
{
  if(PRINCIPAL_FACE_P(f,t))
  {…}
}
end_f_loop(f,t)
```

Inter-Process Communication (1)

- Each compute node maintain local cache of individual variables
- Synchronization or make global reduction of such data involves communication in a particular order
- Consider the simple operation of passing a user defined cortex parameter that is set using scheme but is used in a UDF

Serial Code

```
DEFINE_INIT(set_temp,domain)
{
 real i_temp;
 i_temp = RP_Get_Real("user-temp");
 begin_c_loop(c,t)
  C_T(c,t)=i_temp;
 end_c_loop(c,t)
}
```

Combined (Serial & Parallel) Code

```
DEFINE_INIT(set_temp,domain)
{
 real i_temp;
#if !RP_NODE /* i.e. serial or host */
 i_temp = RP_Get_Real("user-temp");
#endif
host_to_node_real_1(i_temp);
#if !RP_HOST /* i.e. serial or node */
 begin_c_loop_int(c,t)
   C_T(c,t)= i_temp;
 end_c_loop_int(c,t)
#endif
}
```


Inter-Process Communication (2)

- To ensure same code to work for serial and parallel versions, negated compiler directives are mostly used:

```
#if !RP_NODE  /* i.e. serial or host */
#if !RP_HOST  /* i.e. serial or node */
#if !PARALLEL /* i.e. serial only    */
```

- The macro "**`host_to_node_real_1(i_temp);`**" is defined as a **Send** command in the Host version, a **Receive** command in the Compute Node versions and does **Nothing** in the Serial version
- The reciprocal command to **`host_to_node_real_1()`** is **`node_to_host_real_1();`**
- But this only sends the value of **`temp`** from **`Node0`** to the Host
- The formal broadcasting and host communication can be done as below:

```
temp = PRF_GRSUM1(temp);  /*This sums up temp over all nodes*/
                          /*All nodes now have temp=sum     */
node_to_host_real_1(temp);/*only Node0 sends data to Host   */
```

Global Reduction

This combination process is called “Reduction” and there are a number of ways to reduce your data depending on what you want:

1) If you want the total value over all the Nodes, you use a Summation Reduction
2) If you want the Max or Min over all the Nodes use a High or Low Reduction
3) If you want a logical test over all nodes use an And or Or Reduction

There are different macros depending on what data type you’re sending:

```
count    = PRF_GISUM1(count);   /* Total Integer count */
min_temp = PRF_GRLOW1(min_temp);/* Global minimum */
PRF_GLOR(sonic_tests, 3, work); /* Arrays can be reduced too,
                                   needs a work array */
PRF_GRSUM4(v_x,v_y,v_z,v_mag);  /* 4 vars are reduced at a time */
```


Example UDF (1)

- Find totals and averages of a property over all the cells
- Purpose is to write an UDF that works for both Parallel and Serial solvers

```
#include "udf.h"
DEFINE_ON_DEMAND(av_pres_in_thread)
{int thread_id;
 real vol_sum=0.0, pres_sum=0.0;
#if !RP_HOST                              /* serial or node */
  cell_t c; Thread *t;
#endif /* !RP_HOST */
#if !RP_NODE                              /* serial or host */
  thread_id=RP_Get_Integer("udf/av_thread_id");
#endif /* !RP_NODE */
  host_to_node_int_1(thread_id);          /* Passes on  serial */
#if !RP_HOST                              /* serial or node */
  t= Lookup_Thread(Get_Domain(1), thread_id);
  begin_c_loop_int(c,t)                   /* Internal cells only*/
    {vol_sum  += C_VOLUME(c,t);
     pres_sum += C_P(c,t) * C_VOLUME(c,t);}
  end_c_loop_int(c,t)
#endif /* !RP_HOST */                     /* Continued */
```

Example UDF (2)

```
#if RP_NODE
  Message("Sub totals on Node %d: %f,%f\n",myid ,
                                  pres_sum ,vol_sum);
#endif /* RP_NODE */
  vol_sum  = PRF_GRSUM1(vol_sum);
  pres_sum = PRF_GRSUM1(pres_sum);
#if RP_NODE
  Message("Reduced vals Node %d: %f,%f\n",myid ,
                                  pres_sum ,vol_sum);
#endif /* RP_NODE */
node_to_host_real_2(vol_sum,pres_sum);
#if !RP_NODE                          /* i.e., host or serial*/
  Message("Avg. pressure over Thread %d is %f Pa\n", thread_id,
                                          pres_sum/vol_sum);
#endif /* !RP_NODE */
}
```


Message0 ( )

- A function that can be run on node0 that prints directly to the cortex window
- Also works for serial processes

```
Message0("Average pressure over Thread %d ",thread_id);
Message0("is %f Pa\n",pres_sum/vol_sum);
```

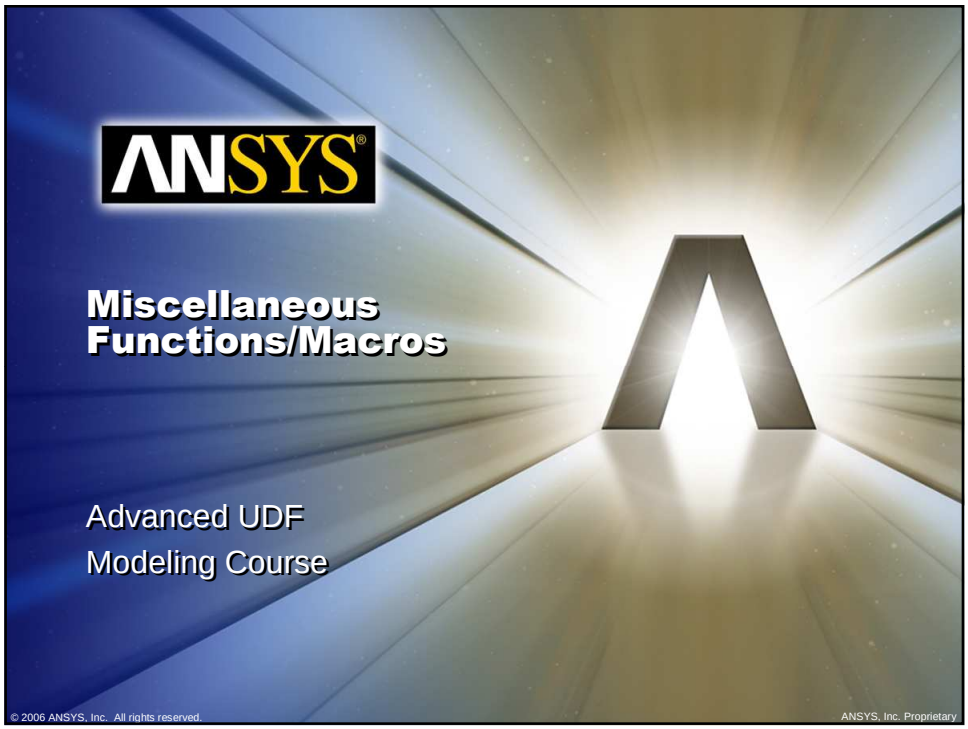
- Note the exact similarity of the function “Message0” with Message and printf commands

Parallel File Output

- In a parallel session, file I/O can be done only through the **Node_Zero**

Example:

```
#if PARALLEL
if (I_AM_NODE_ZERO_P)
{ sprintf (ntim,"outfile-%d", ntime);
  if (fd == NULL)   /*Open a new file */
   {fd = fopen(ntim,"w");}
/* if new file "open" failed, try to append */
  if (fd == NULL) /* reopen the file in append-mode*/
   {fd = fopen(ntim,"a");
    Message( "Appending to existing file: %s", ntim);
    fprintf(fd,"\nAppend begins at: %f \n", f_time);}
}
#endif
```



Trigonometric Functions

- `double acos (double x);` returns the arc-cosine of x
- `double asin (double x);` returns the arc-sine of x
- `double atan (double x);` returns the arc-tangent of x
- `double atan2 (double x, double y);` returns the arc-tangent of x/y
- `double cos (double x);` returns the cosine of x
- `double sin (double x);` returns the sine of x
- `double tan (double x);` returns the tangent of x
- `double cosh (double x);` returns the hyperbolic cosine of x
- `double sinh (double x);` returns the hyperbolic sine of x
- `double tanh (double x);` returns the hyperbolic tangent of x

Miscellaneous Math-Functions

- `double sqrt (double x);` returns the square root of x
- `double pow (double x, double y);` returns x^y
- `double exp (double x);` returns e^x
- `double log (double x);` returns ln(x)
- `double log10 (double x);` returns $\ln_{10}(x)$
- `double fabs (double x);` returns |x|
- `double ceil (double x);` smallest integer not less than x
- `double floor (double x);` largest integer not greater than x

- The macro **`UNIVERSAL_GAS_CONSTANT`** returns the value of the universal gas constant (8314.34), which is expressed in SI units of J/Kmol-K
- The macro **`M_PI`** returns the value of π

Standard I/O Functions

- use **`Message`** instead of **`printf`** in compiled UDFs (UNIX only)

```
Message ("Volume integral: %g\n", sum_vol);
```

- `FILE *fopen(char *filename, char *type);` opens a file
- `int fclose(FILE *fd);` closes a file
- `int fprintf(FILE *fd, char *format, ...);` formatted print to a file
- `int printf(char, *format, ...);` print to screen
- `int fscanf(FILE *fd, char *format, ...);` formatted read from a file

See your system manual pages for more details

Note that for parallel runs, the I/O macros need to be different

- Example:

```
FILE *fd;
real f1, f2;
fd = fopen("data.txt","r");
fscanf(fd, "%f %f",&f1,&f2);
fclose(fd);
```

Special Macro's

- `cxboolean Data_Valid_P()` Equals `1` if data is available, `0` if not
  Usage: `if(!Data_Valid_P())return;`
- `cxboolean FLUID_THREAD_P(t0)` true if thread t0 fluid thread
- `cxboolean SOLID_THREAD_P(t0)` true if thread t0 is solid thread
- `cxboolean BOUNDARY_FACE_THREAD_P(t0)` true if thread t0 is boundary thread
- `NULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))`
  - Checks for storage allocation of user defined scalars
- `CURRENT_TIME` Real current flow time (in seconds)
- `CURRENT_TIMESTEP` Real current physical time step size (in sec)
- `PREVIOUS_TIME` Real previous flow time (in seconds)
- `PREVIOUS_2_TIME` Real flow time two steps back in time (in sec)
- `N_TIME` Integer number of time steps
- `N_ITER` Integer number of iterations

Miscellaneous: Vector Utilities

- **`ND_ND`** in the declaration of a vector or matrix stands for the actual fluent dimension (2D / 3D)
- **`X[ND_ND]`** is equivalent to:
  - 2D: **`X[2]`**
  - 3D: **`X[3]`**
- **`NV_MAG`** computes the magnitude of a vector: **`X[ND_ND]`**
- **`NV_MAG(x)`** is equivalent to:
  - 2D: **`sqrt(x[0]*x[0] + x[1]*x[1]);`**
  - 3D: **`sqrt(x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);`**
- **`NV_MAG2`** computes the sum of squares of vector components
- **`NV_MAG2(x)`** is equivalent to:
  - 2D: **`(x[0]*x[0] + x[1]*x[1]);`**
  - 3D: **`(x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);`**

Miscellaneous: Vector Utilities

- **ND_SUM** computes the sum of **ND_ND** arguments
- **ND_SUM(x,y,z)** is equivalent to:
  - 2D: `x + y;`
  - 3D: `x + y + z;`
- **ND_SET** generates **ND_ND** assignment statements
  - 2D: **ND_SET(u,v,C_U(c,t),C_V(c,t))** is equivalent to:
    - `u = C_U(c, t);`
    - `v = C_V(c, t);`
  - 3D: **ND_SET(u,v,w,C_U(c,t),C_V(c,t),C_W(c,t))** is equivalent to:
    - `u = C_U(c, t);`
    - `v = C_V(c, t);`
    - `w = C_W(c, t);`

Miscellaneous: Vector Utilities

- **NV_V** performs an operation on two vectors
  - `NV_V(a, =, x);`
  - `a[0] = x[0]; a[1] = x[1];` etc.
  - Note that if you use `+ =` instead of `=` in the above equation, then you get `a[0]+= x[0];` etc.
- **NV_VV** is a vector operator . The operation that is performed on the elements depends upon what is used as an argument in place of the + signs
  - `NV_VV(a, =, x, +, y)`/* The '+' symbol can be replaced by (-, /,*) */
  - 2D: `a[0]= x[0]+y[0], a[1]= x[1]+y[1];`
  - 3D: `a[0]= x[0]+y[0], a[1]= x[1]+y[1], a[2]= x[2]+y[2];`

Miscellaneous: Vector Utilities

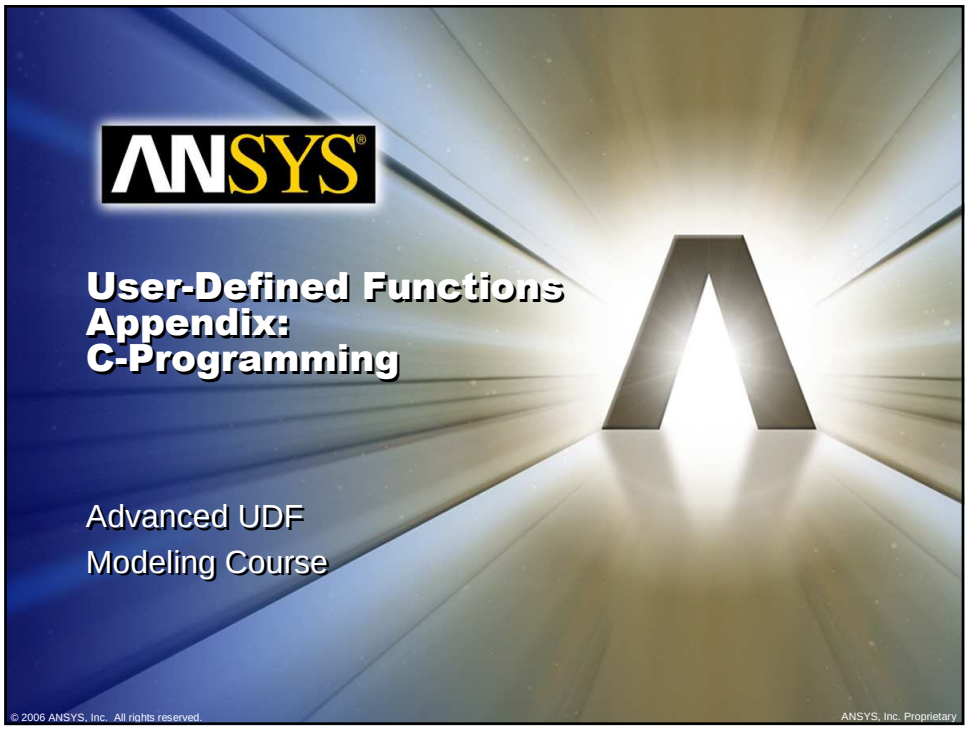
- **NV_V_VS** adds a vector to another which is multiplied by a scalar
  - `NV_V_VS(a,=, x,+,y,*,0.5);`
  - 2D: `a[0]=x[0]+(y[0]*0.5), a[1]=x[1]+(y[1]*0.5);`
  - Note that + sign can be replaced by -, /, or *, and '*' sign can be replaced by '/'
- **NV_VS_VS** adds a vector to another which are each multiplied by a scalar
  - `NV_VS_VS(a,=,x,*,2.0,+,y,*,0.5);`
  - 2D:
    ```
    a[0]=(x[0]*2.0)+(y[0]*0.5),
    a[1]=(x[1]*2.0)+(y[1]*0.5);
    ```
  - Note that + sign can be used in place of -, *, or /, and '*' sign can be replaced by '/'

Miscellaneous: Vector Utilities

- The dot products of two sets of vector or components
- **ND_DOT(x,y,z,u,v,w)** is equivalent to:
  - `2D: (x*u+y*v);`
  - `3D: (x*u+y*v+z*w);`
- **NV_DOT(x,u)** is equivalent to:
  - `2D: (x[0]*u[0]+x[1]*u[1]);`
  - `3D: (x[0]*u[0]+x[1]*u[1]+x[2]*u[2]);`
- **NVD_DOT(x,u,v,w)** is equivalent to:
  - `2D: (x[0]*u+x[1]*v);`
  - `3D: (x[0]*u+x[1]*v+x[2]*w);`
- **NV_CROSS(a,x,y)** is available for **3D** only:
  - It returns the cross product of vectors **x** and **y** in the new vector **a**

Closure

- All UDF-s must be written in SI units
- UDF-s open up a virtually endless opportunity to extend the modeling capabilities of the basic FLUENT code
- Details of the examples and all working macros & parameters are available in the UDF manual at Fluent User Services Center



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`**

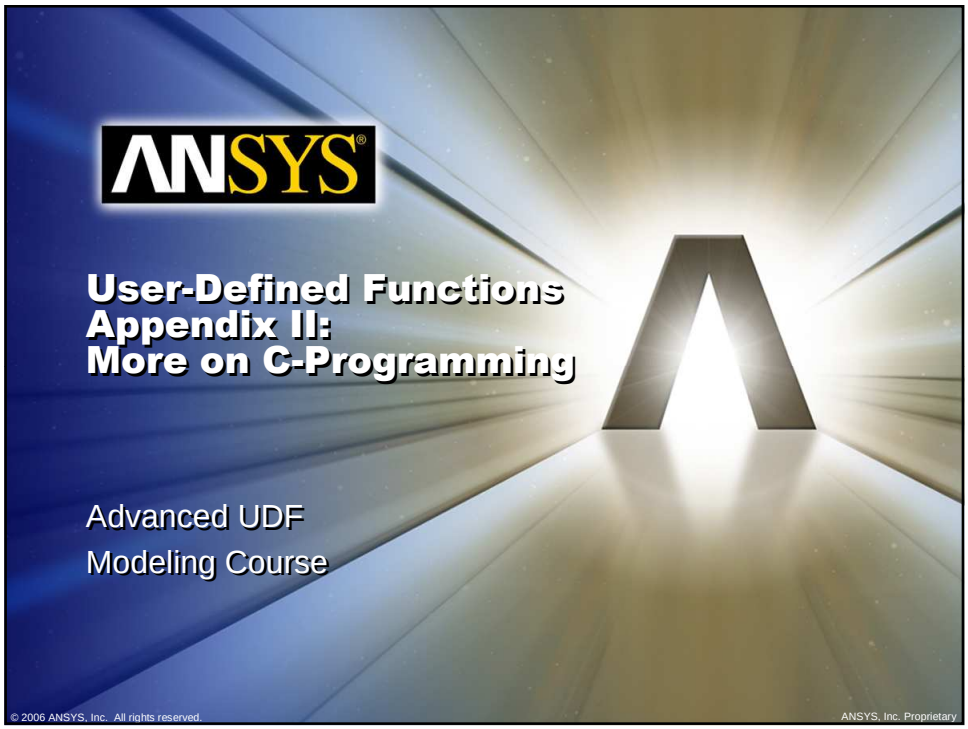
```
#include "udf.h"
#include "mystuff.h"
```

  - 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`



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`**

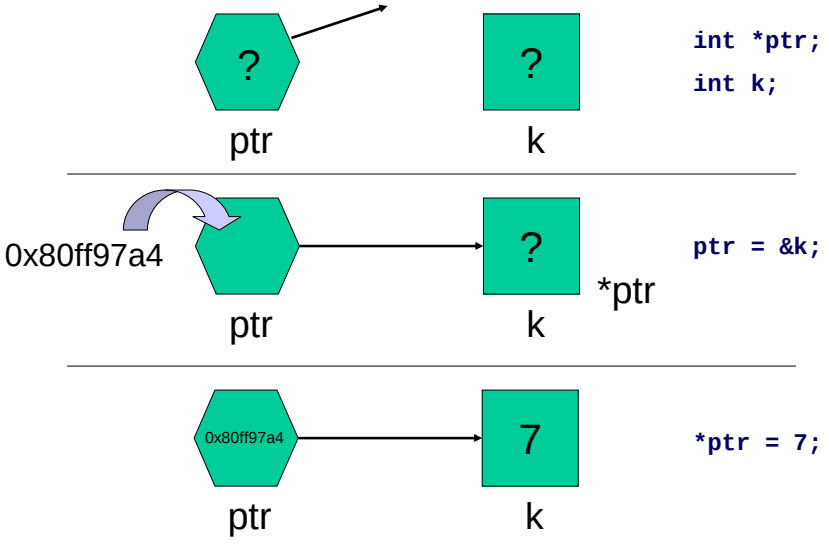
Pointers (2)

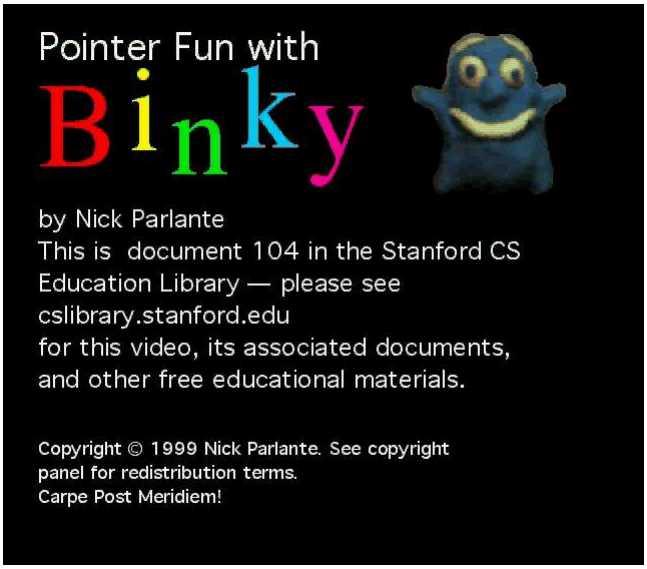
- Suppose we want a variable that holds a memory location (or *address*)
- Such a variable is called a *pointer*
- Consider the declaration
  - `int *ptr;`
- The **`*`** informs the compiler we wish to set aside enough memory for an address
- The **`int`** informs the compiler we wish to store the address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





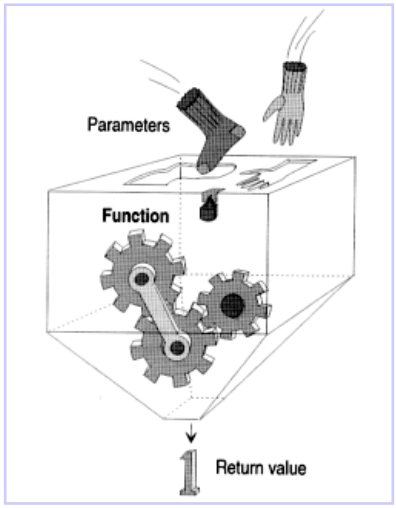
Exercise: Pointer1

- 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