

# Goals for This Lecture:

- Understand Multi-dimensional Arrays in C++
- Understand pointers
- Understand dynamic arrays
- Understand the mechanics of separate file compilation
- Recode our ODE solvers in C++

# C++ Multidimensional Arrays

- C++ allows the declaration of multi-dimensional arrays

- Form:

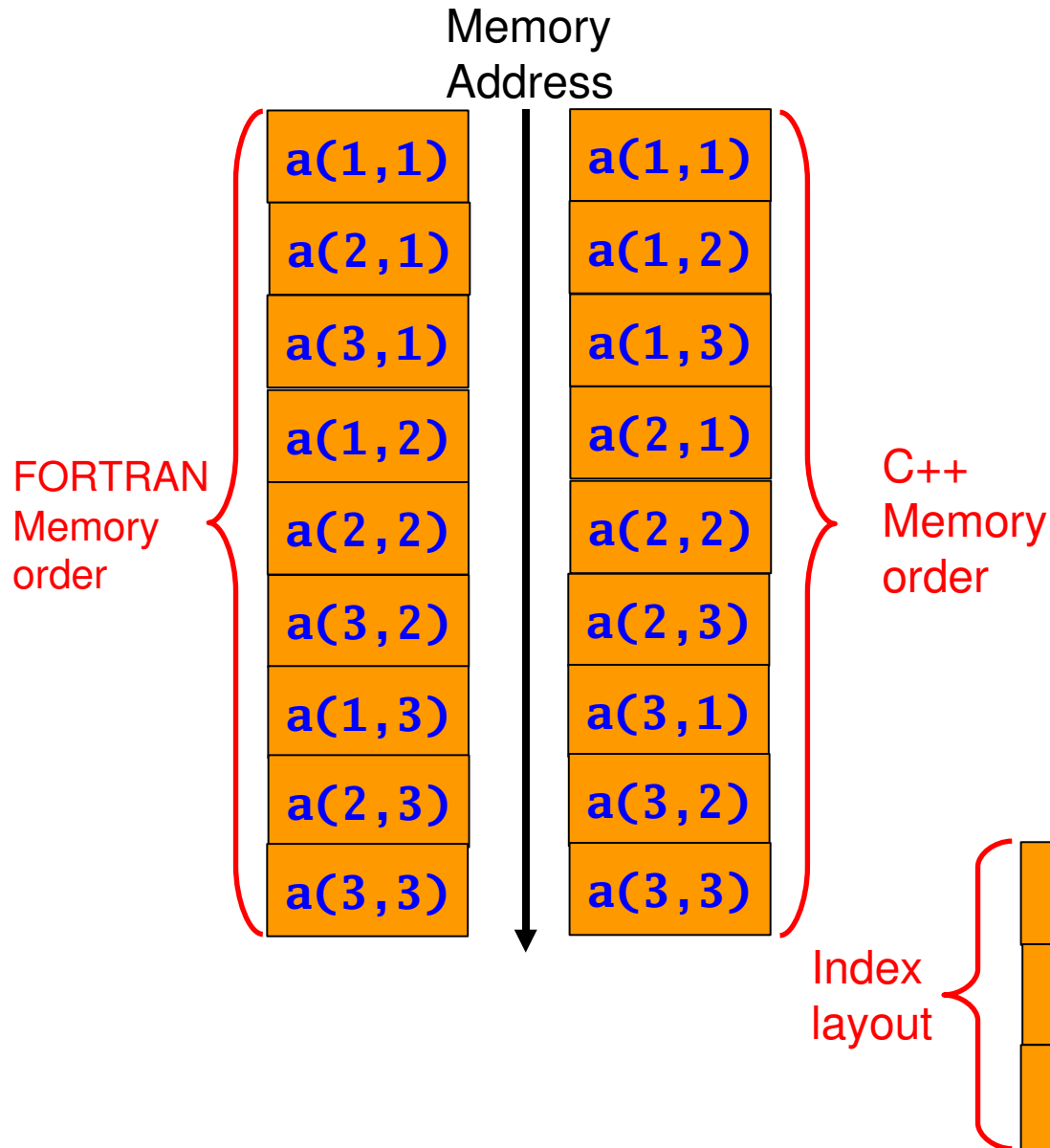
```
type array_name[size_dim_1][size_dim_2]...[size_dim_last];
```

- Example:

```
double matrix[100][100];
```

- The type can be any of the C++ types, e.g. int, float, char, bool, etc.
- A two-dimensional array can be thought of as a 1-d array in which each element is another 1-d array
- An N-dimensional array can be thought of as an 1-d array of N-1 dimensional arrays

# Multi-dimensional Array Storage



- In FORTRAN, multi-dimensional arrays are ordered in column-major order
- Elements are stored in memory in a sequence where the left-most index varies most rapidly
- This is the reverse of other languages like C & C++ where elements are stored in row-major order
- In C++ the arrays are stored in memory so that the rightmost index varies most rapidly as one moves through memory

# C++ Multidimensional Array Parameters

- When a multidimensional array is used as a parameter in a C++ function the size of the first dimension is not given but the size of the remaining dimensions must be given
- Example:

```
void print_matrix(double matrix[][100], int msize);
```
- Usually you will have to supply other parameters to describe the extent of each dimension, e.g. **msize**
- This bizarre requirement is a result of the fact that C++ does not truly have multidimensional arrays
  - Instead it has arrays of arrays
  - **matrix[]** is an array of 1-d double arrays of extent 100
  - This means that for multi-d arrays even the C++ analog of a FORTRAN deferred-size type declaration is difficult

# C++ Pointers

- A **pointer** is the memory address of a variable or other object
- It's called a pointer because the address is thought of as pointing to where the variable lives
- You have already been using pointers without knowing it
- When using call-by-reference argument passing the function is passed a pointer to the argument variable
- When an array is given to a function a pointer to the beginning of the array is actually passed to the function
- Pointers are needed to create dynamically allocated arrays in C++
- Pointers are the most dangerous aspect of C or C++ programming that you will encounter

# C++ Pointers

- C++ allows the declaration of pointers for any base type of variable
- Form:

```
type *variable1, *variable2, ...;
```

- Example:

```
double *Ptr_x, *Ptr_y;
```

- The **\*** is called the dereferencing operator
- If **Ptr\_x** points to the variable **x**, then **\*Ptr\_x** references the value located at the address pointed to by **Ptr\_x**, i.e. **\*Ptr\_x** has the same value as **x**
- The addressing operator **&** can be used to access the address of a variable
- Example:  

```
Ptr_x = &x ; // assigns the address of x to Ptr_x
```

# Use Extreme Caution with Pointers!

- Pointers are confusing & dangerous in many ways
- This is especially true when assigning values to pointers
- Example:
  - The statement  
`Ptr_x = Ptr_y;`  
is not the same as  
`*Ptr_x = *Ptr_y;`
  - The first statement alters the pointer `Ptr_x` to point to the same address that `Ptr_y` points to.
  - The second statement changes the value in the location pointed to by `Ptr_x` to be the same as that of `*Ptr_y`
  - Always use extreme caution when programming with pointers!!

# Pointers and the **new** operator

- Pointers can be used to access variables that are created dynamically with the **new** operator
- The **new** operator allocates space for a new variable of a given type and returns the address of that space

- Example of use:

```
Ptr_x = new double ; // Allocates new memory for a double
```

The variables value can now be referenced as **\*Ptr\_x**

- You can also initialize this variable during it's creation by specifying a value in parenthesis after the type:

```
Ptr_x = new double(3.14);
```

- The memory for the variable can (and should!) be released after you are done with it by means of the **delete** operator
- Example:
- **delete Ptr\_x; // deletes dynamic variable pointed to by Ptr\_x**

# Arrays and Pointers

- Array variables are actually pointers
- When you declare an array such as:  
**double data[3]={1.1, 2.2, 3.3} ;**
- The identifier **data** (without brackets) is actually a pointer  
**data** points to the first element of the array  
**data+1** points to the second element of the array  
**data+2** points to the third element of the array  
**\*data** has the value 1.1  
**\*(data+1)** has the value 2.2  
**\*(data+2)** has the value 3.3
- Thus **data[2]** is just shorthand notation for **\*(data+2)**

# Pointers and Dynamic Arrays

- Pointers can be used along with the new operator to create dynamic arrays

- Example:

```
double *Ptr_data;
```

```
Ptr_data = new double[10]; // create 10 element array
```

```
...
```

```
delete [] Ptr_data ; // Delete the memory
```

- The identifier **Ptr\_data** (without brackets) is actually a pointer to the beginning of the dynamically allocated array
- This can now be used like any other array, with each element being referenced using the square bracket notation

- Example:

```
Ptr_data[3] = 5.5; // Assign value 5.5 to fourth element of array
```

# An example of dynamic array usage

- Let's recode our ODE solvers as C++ code
- Use dynamic arrays in C++ instead of allocatable arrays in FORTRAN 95
- Still need `prob_size`, `prob_init_cond`, `prob_rhs` functions
  - No problem in C++
- We want to keep the problem specific routines in a separate file from the generic numerical method routines.

# The ode\_euler solver

- Let's recode our ODE solvers as C++ code
- Use dynamic arrays in C++ instead of allocatable arrays in FORTRAN 95
- Still need prob\_size, prob\_init\_cond, prob\_rhs functions
  - No problem in C++
- We want to keep the problem specific routines in a separate file from the generic numerical method routines.

# Implementing a problem independent Euler's Method Code

! Purpose: Integrate a set of coupled ODEs with Euler's method  
! Note: User must supply subroutines prob\_size, prob\_init\_conds, and prob\_rhs  
!

```
program ode_euler
implicit none
integer :: neqns          ! Number of equations
integer :: i              ! Implicit DO-loop index
real(kind=kind(1.0d0)), allocatable :: y(:), rhs(:) ! y & r.h.s.
real(kind=kind(1.0d0)) :: t, tstop, dt      ! Time, Stopping time, & timestep
call prob_size(neqns)      ! Get # of equations
allocate(y(neqns))        ! Allocate y & ynew arrays to correct size
allocate(rhs(neqns))
call prob_init_conds(t,tstop,dt,y) ! Get initial conditions
open(unit=17,file='results.dat') ! Open a file for output
do while(t < tstop)      ! Integrate forward in time
    call prob_rhs(t,y,rhs) ! Evaluate right-hand-side of ODEs
    y = y+dt*rhs          ! Apply Euler's method
    t = t+dt              ! Increment time
    write(17,'(t2,20(1x,es12.5))') t,(y(i),i=1,neqns) ! Write out results
enddo
close(unit=17)
stop
end program ode_euler
```

# The C++ Equivalent

```
#include <iostream>
#include "ode_funcs.h"          // Include the ODE function header file
using namespace std;
int main(){
    double t, tstop, dt ;
    double *y, *rhs;          // Pointers to arrays
    int neqns;                // Size of arrays
    neqns = prob_size();      // Get problem size
    y = new double[neqns];    // Allocate y array
    rhs = new double[neqns];  // Allocate rhs array
    prob_init_conds(t,tstop,dt,y); // Get initial conditions
    while(t < tstop){        // While loop
        prob_rhs(t,y,rhs) ;   // Call r.h.s. function
        for(int i=0; i < neqns; i++) y[i]=y[i]+dt*rhs[i] ; // Update y
        t = t+dt;             // Increment the time
        cout << " " << t ;    // Output solution @ time t
        for(int i=0; i < neqns; i++) cout << " " << y[i] ;
        cout << endl ;
    } // End of while loop
    delete [] y ;             // Deallocate space for y
    delete [] rhs;           // Deallocate space for rhs
    return(0);
} // End of main
```

# The Header File

```
//  
// This file contains the function prototypes for use  
// in the ODE solvers package. The user must supply  
// functions that match the following declarations.  
//
```

```
// Declaration of problem size function
```

```
int prob_size();
```

```
// Declaration of problem initial conditions function
```

```
void prob_init_conds(double& t, double& tstop,  
                    double& dt, double y[]);
```

```
// Declaration of problem right-hand-side function
```

```
void prob_rhs(double& t, double y0[], double rhs[]);
```

## The prob\_size subroutine

**! Purpose: Return the number of equations for  
! the orbital problem (through the argument  
! neqns)**

**subroutine prob\_size(neqns)**

**! Number of equations**

**implicit none integer, intent(out) :: neqns**

**neqns = 4 ! Initialize this to 4**

**return**

**end subroutine prob\_size**

## The prob\_size subroutine in C++

```
int  prob_size()  
{  
    return(4); // Return number of equations as 4  
}
```

## The prob\_init\_conds subroutine

**! Purpose: Return the initial conditions for  
! the orbital problem**

```
subroutine prob_init_conds(t,tstop,dt,y)
implicit none
real(kind=kind(1.0d0)), intent(out) :: t, tstop, dt, y(4)
t = 0.0d0 ! Initialize t to zero
tstop = 2.0d0 ! Initialize tstop to two years
write(*,'(t2,a20,$)') "Enter timestep size:"
read(*,*) dt
y(1) = 1.0d0 ! Initialize x coordinate
y(2) = 0.0d0 ! Initialize y coordinate
y(3) = 0.0d0 ! Initialize x velocity
y(4) = 6.29d0 ! Initialize y velocity
return
end subroutine prob_init_conds
```

## The prob\_init\_conds function in C++

```
void prob_init_conds(double& t, double& tstop,  
                    double& dt, double y0[])  
{  
    t = 0.0;           // Set initial time  
    tstop = 2.0;      // Set stopping time  
    dt = 0.01;        // Set step size  
    y0[0] = 1.0;      // Initial value of x  
    y0[1] = 0.0;      // Initial value of y  
    y0[2] = 0.0;      // Initial value of vx  
    y0[3] = 6.29;     // Initial value of vy  
}
```

## The prob\_rhs subroutine

**! Purpose: Return the right-hand-side for**

**! the orbital problem**

```
subroutine prob_rhs(t,y0,rhs)
```

```
implicit none real(kind=kind(1.0d0)), intent(in) :: t, y0(4)
```

```
real(kind=kind(1.0d0)), intent(out) :: rhs(4)
```

```
real(kind=kind(1.0d0)) :: x, y, vx, vy
```

**! Constants in units of solar masses and AU**

```
real(kind=kind(1.0d0)), parameter :: grav=39.47, msun=1.0
```

```
x = y0(1) ! Unpack y0 vector
```

```
y = y0(2)
```

```
vx = y0(3)
```

```
vy = y0(4)
```

```
rhs(1) = vx ! r.h.s. of x position equation
```

```
rhs(2) = vy ! r.h.s. of y position equation
```

```
rhs(3) = -grav*msun*x/(x**2 + y**2)**1.5d0 ! r.h.s. of X vel. equation
```

```
rhs(4) = -grav*msun*y/(x**2 + y**2)**1.5d0 ! r.h.s. of X vel. equation
```

```
return
```

```
end subroutine prob_rhs
```

## The prob\_rhs function

```
void prob_rhs(double& t, double y0[], double rhs[])
{
    const double GRAV=39.47; // Gravitational constant
    const double MSUN = 1.0; // Mass of sun
    double x, y, vx, vy;

    x = y0[0]; // Unpack the y0 vector
    y = y0[1];
    vx = y0[2];
    vy = y0[3];

    // Now calculate the right-hand-side
    // of each equation

    rhs[0] = vx ;
    rhs[1] = vy ;
    rhs[2] = -GRAV*MSUN*x/pow((x*x+y*y),1.5) ;
    rhs[3] = -GRAV*MSUN*y/pow((x*x+y*y),1.5) ;
}
```

# The Second Order Runge-Kutta Code

```
program ode_rk2
implicit none
integer :: neqns ! Number of equations
integer :: i ! Implicit DO-loop index
real(kind=kind(1.0d0)), allocatable :: y(:), k1(:), rhs(:) ! y, y1, & r.h.s
real(kind=kind(1.0d0)) :: t, tstop, dt ! Time, Stopping time, & timestep
call prob_size(neqns) ! Get # of equations
allocate(y(neqns)) ! Allocate y & ynew arrays to correct size
allocate(k1(neqns))
allocate(rhs(neqns))
call prob_init_conds(t,tstop,dt,y) ! Get initial conditions
open(unit=17,file='results.dat') ! Open a file for output

do while(t < tstop) ! Integrate forward in time
  call prob_rhs(t,y,rhs) ! Evaluate right-hand-side of ODEs
  k1 = dt*rhs ! Apply first step
  call prob_rhs(t+0.5d0*dt,y+0.5d0*k1,rhs) ! Evaluate right-hand-side at midpoint
  y = y+dt*rhs ! Apply first step
  t = t+dt ! Increment time
  write(17,'(t2,20(1x,es12.5))') t,(y(i),i=1,neqns) ! Write out results
enddo
close(unit=17)
stop
end program ode_rk2
```

# The Second Order Runge-Kutta Code in C++

```
#include <iostream>
#include "ode_funcs.h"          // Include the ODE function header file
using namespace std;
int main() {
    double t, tstop, dt , thalf;
    double *y, *y1, *rhs;      // Pointers to arrays
    int neqns;                 // Size of arrays
    neqns = prob_size();       // Get problem size
    y = new double[neqns];     // Allocate y array
    rhs = new double[neqns];   // Allocate rhs array
    y1 = new double[neqns];    // Allocate y1 array
    prob_init_conds(t,tstop,dt,y); // Get initial conditions
    while(t < tstop)          { // While loop
        prob_rhs(t,y,rhs) ;    // Call r.h.s. function
        for(int i=0; i < neqns; i++) y1[i]=y[i]+0.5*dt*rhs[i] ; // Calculate y1
        thalf = t+0.5*dt;
        prob_rhs(thalf,y1,rhs) ; // Call r.h.s. function
        for(int i=0; i < neqns; i++) y[i]=y[i]+dt*rhs[i] ; // Update y
        t = t+dt;              // Increment the time
        cout << " " << t ;     // Output solution @ time t
        for(int i=0; i < neqns; i++) cout << " " << y[i] ;
        cout << endl ;        } // End of while loop
    delete [] y ; delete [] rhs; delete [] y1 ; // Deallocate space
    return(0); } // End of main
```

# Assignment

- Read sections 10.1-10.2 of Savitch