

# Goals for This Lecture:

- Understand the pass-by-value and pass-by-reference argument passing mechanisms of C++
- Understand the use of C++ arrays
- Understand how arrays are passed to C++ functions

# Call-by-value argument passing

- C++ differs from FORTRAN in the way it passes arguments by default
- The default argument passing scheme is referred to as call-by-value
- The values of the arguments are plugged into the parameters of the function
- The parameters are not associated with the arguments
- Therefore when using the call-by-value argument passing scheme changes to the parameters in the function are not manifested outside of the function
- C++ has a call-by-reference scheme akin to that of FORTRAN but it must be invoked explicitly

# Call-by-reference argument passing

- Call-by-reference argument passing in C++ is done through the use of the referencing operator & in conjunction with the type definition
- Also known as the addressing operator
- The parameters in both the function declaration and function definition must be declared as call by reference parameters
- Warning: lots of potential for errors here

- Example:

```
double add_two(double& x) ; // function declaration
double add_two(double& y) // function definition
{ y = y+2.0; return(y); }
```

- Note: y will be changed in the calling program
- Argument in function invocation must be a variable (not an expression)

# C++ Arrays

- Like FORTRAN C++ provides arrays to handle lists of data
- Arrays of any intrinsic type (int, float, double, char, etc.) can be defined
- Arrays can be manipulated on an element-by-element basis
- Unlike FORTRAN, C++ does not provide:
  - The ability to carry out whole-array operations
  - The capability of referencing a subset of an array by specifying index ranges
- All C++ array indices start at zero
  - Unlike FORTRAN where by default they start at one
  - C++ does not have the ability to specify the range of an array

## A C++ Array Example

- Like FORTRAN C++ provides arrays to handle lists of data

```
#include <iostream>
using namespace std ;
const int N_POINTS=4 ;
main() {
    double xData[4]={1.1, 1.89, 3.2, 4.05};
    double xTheory[N_POINTS]={1.0,2.0,3.0,4.0};

    for(i=0; i < N_POINTS; i++){
        cout << " diff of i=" << i << " is " <<
            xData[i]-xTheory[i] << endl ; }
    return(0);
}
```

# Declaring C++ Arrays

- Arrays are declared by specifying the size in square brackets after the array name (much like FORTRAN)
- Note the square brackets unlike FORTRAN's parenthesis
- Valid array indices start at zero and run to one less than the number in the declaration, e.g. **0, 1, 2, 3**

```
#include <iostream>
using namespace std ;
const int N_POINTS=4 ;
main() {
    double xData[4]={1.1, 1.89, 3.2, 4.05};
    double xTheory[N_POINTS]={1.0,2.0,3.0,4.0};

    for(i=0; i < N_POINTS; i++){
        cout << " diff of i=" << i << " is " <<
            xData[i]-xTheory[i] << endl ; }
    return(0);
}
```

# Declaring C++ Arrays

- Good programming tip: Always declare arrays using a global constant for size
- Allows easy changing of array sizes and loop ranges throughout the code

```
#include <iostream>
using namespace std ;
const int N_POINTS=4 ;
main() {
    double xData[4]={1.1, 1.89, 3.2, 4.05};
    double xTheory[N_POINTS]={1.0,2.0,3.0,4.0};

    for(i=0; i < N_POINTS; i++){
        cout << " diff of i=" << i << " is " <<
            xData[i]-xTheory[i] << endl ; }
    return(0);
}
```

# Initializing C++ Arrays

- C++ arrays can be initialized on the declaration line by placing a list of elements in curly brackets after an assignment operator
- No implied do-loop constructor capabilities for initialization unlike FORTRAN

```
#include <iostream>
using namespace std ;
const int N_POINTS=4 ;
main() {
    double xData[4]={1.1, 1.89, 3.2, 4.05} ;
    double xTheory[N_POINTS]={1.0,2.0,3.0,4.0} ;

    for(i=0; i < N_POINTS; i++){
        cout << " diff of i=" << i << " is " <<
            xData[i]-xTheory[i] << endl ; }
    return(0);
}
```

# Referencing C++ Array Elements

- C++ array elements can be referenced by placing the element number in square brackets after the array name
- You are responsible for avoiding out-of-bounds element references
- **No range subscripting capabilities unlike FORTRAN**

```
#include <iostream>
using namespace std ;
const int N_POINTS=4 ;
main() {
    double xData[4]={1.1, 1.89, 3.2, 4.05} ;
    double xTheory[N_POINTS]={1.0,2.0,3.0,4.0} ;

    for(i=0; i < N_POINTS; i++){
        cout << " diff of i=" << i << " is " <<
            xData[i]-xTheory[i] << endl ; }
    return(0);
}
```

# Referencing C++ Array Elements

- C++ array elements can be referenced by placing the element number in square brackets after the array name
- You are responsible for avoiding out-of-bounds element references
- Indexed array elements can be used anywhere an element of the base type could be used
- No range subscripting capabilities unlike FORTRAN

```
#include <iostream>
using namespace std ;
const int N_POINTS=4 ;
main() {
    double xData[4]={1.1, 1.89, 3.2, 4.05} ;
    double xTheory[N_POINTS]={1.0,2.0,3.0,4.0} ;

    for(i=0; i < N_POINTS; i++){
        cout << " diff of i=" << i << " is " <<
            xData[i]-xTheory[i] << endl ; }
    return(0);
}
```

# C++ Arrays as Function arguments

- Entire C++ arrays can be passed into C++ functions just as in FORTRAN
- Both the function declaration and the function definition should specify that the parameter is an array
- When a whole array is passed into a C++ function the pass-by-reference mechanism is used
  - Done for efficiency reasons (to avoid copying of many elements)
  - Do not need to specify the array as a pass-by-reference parameter
- Changes to elements of a parameter arrays in a C++ function will result in changes in the elements of the array argument

# C++ Arrays in Functions

- Arrays are declared in C++ functions in a manner that is akin to a deferred-size array declaration in FORTRAN
- The size of the array has to be provided to the function somehow
  - Passed in as an argument
  - Provided as a global parameter (bad idea)
- No notion of deferred-shape arrays unlike FORTRAN 95!
  - No way to get info about extent or rank of array
- No notion of automatic arrays unlike FORTRAN 95!

```
some_function(size) {  
    int size;  
    double array[size]; // Illegal declaration  
}
```

# Bubble Sort Function Example

```
void bubble_sort(double array[], int asize)
{
    int i, j;           // Loop indices
    double tempv;      // Temporary variable for swapping
    for(j=1; j < asize; j++) { // Repeat bubble pass N-1 times

        for(i=0; i < asize-1 ; i++) { // Loop over elements

            if( array[i] > array[i+1] ) // If ith element > i+1th
            { tmpv = array[i+1]; // Then swap the elements
              array[i+1] = a[i];
              array[i] = tmpv;
            }

        } // End of inner for loop over i
    } // End of outer for loop over j
}
```

# Calling the Bubble Sort Function

```
#include <iostream>
using namespace std;
                // Declare the bubble sort function
void bubble_sort(double array[], int asize);

main(){
    const int SIZE=4;                // Size of array
    double a[SIZE]={4.2,3.14,-5.2,0.1}; // Array to be sorted

    bubble_sort(a,SIZE);            // Sort the array

    for(int i=0; i < SIZE ; i++)    // Write out sorted array
        cout << i << " " << a[i] << endl ;

return(0);    // Return a no-error value
}
```

# Assignment

- Read sections 5.2-5.3 of Savitch