Lecture 3

Object Oriented Programming

Instructor: Dr. Danish Shehzad

Recap

- Pointers
- Referencing and Dereferencing
- Arithmetic on Pointers
- Relationship between Pointers and Arrays
- Accessing array elements through pointers

Today's Lecture

- Accessing 2 D Arrays using pointers
- Passing Arguments to Functions by Reference using Pointers
- Constant Pointers
- Sizeof
- Arrays of Pointers

Accessing 1-Demensional Array

We can access all element of List [50] using Pointers and for loop combinations.

```
…
…
     int List [ 50 ];
     int *Pointer;
     Pointer = List;
     for ( int i = 0; i < 50; i++ )
     {
           cout << *Pointer;
           Pointer++; // Address of next element
     }
```

Address	Data
980	Element 0
982	Element 1
984	Element 2
986	Element 3
988	Element 4
990	Element 5
992	Element 6
994	Element 7
996	Element 8
…	
…	
998	Element 50

This is Equivalent to

```
     for ( int loop = 0; loop < 50; loop++ )
           cout <<  Array [ loop ] ;
```

Accessing 2-Demensional Array

- Note that the statements
  - int *Pointer;
  - Pointer = &List [3];
- represents that we are accessing the address of 4th slot.
- In 2-Demensional array the statements
  - int List [5] [6];
  - int *Pointer;
  - Pointer = &List [3];
- Represents that we are accessing the address of 4th row
- **or** the address the 4th row and 1st column.

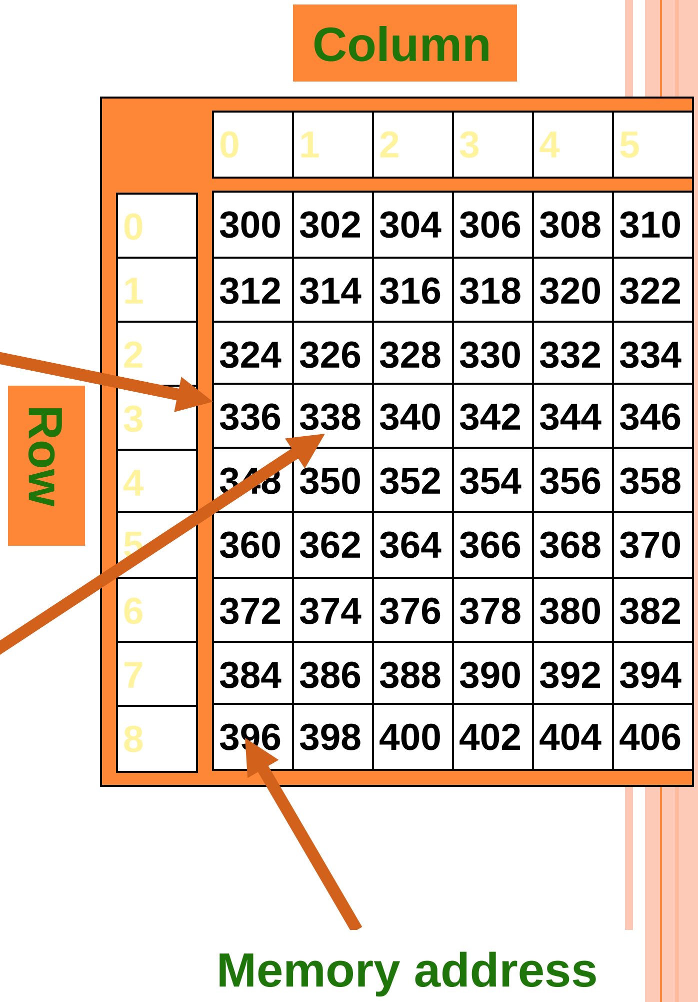
Address	Data
980	Element 0
982	Element 1
984	Element 2
986	Element 3
988	Element 4
990	Element 5
992	Element 6
994	Element 7
996	Element 8
…	
…	
998	Element 50

Accessing 2-Demensional Array

- int List [9] [6];
- int *ptr;
- ptr = &List [3];

To access the address of 4th row 2nd column we can increment the value of (ptr).

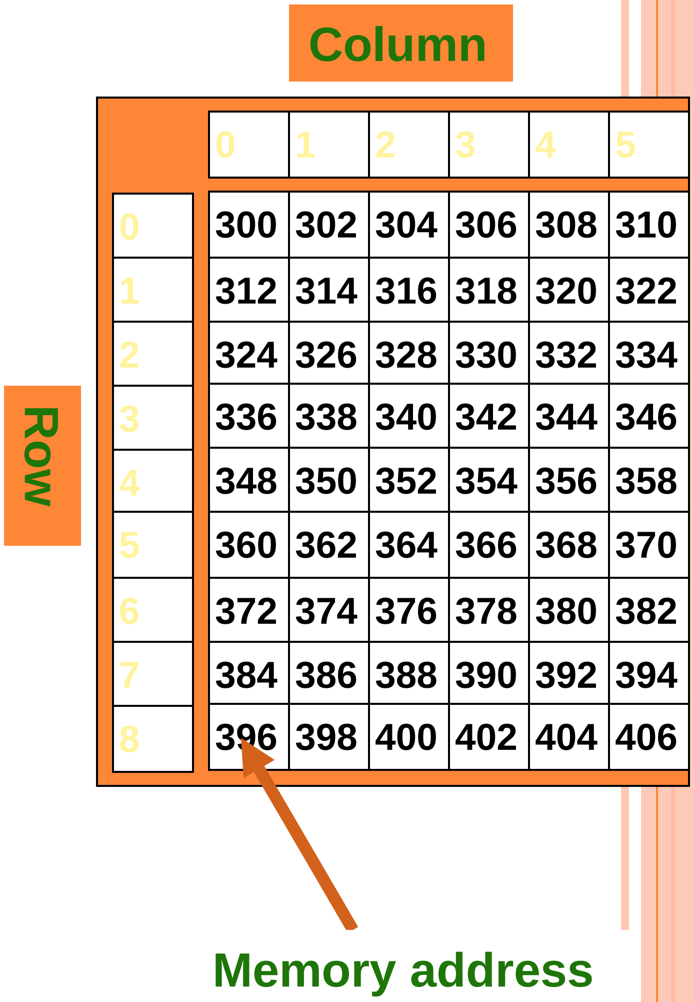
- ptr++; // address of 4th row 2nd column (faster than normal array accessing **Why?**)
- Equivalent to List [3][1] ;



	0	1	2	3	4	5
0	300	302	304	306	308	310
1	312	314	316	318	320	322
2	324	326	328	330	332	334
3	336	338	340	342	344	346
4	348	350	352	354	356	358
5	360	362	364	366	368	370
6	372	374	376	378	380	382
7	384	386	388	390	392	394
8	396	398	400	402	404	406

Accessing 2-Demensional Array

- We know computer can perform only one operation at any time (remember fetch-decode-execute cycle).
- Thus to access List [3][1] element (without pointer) two operations are involved:-
  - **First to determine row List [3]**
  - **Second to determine column List[3][1]**
- But using pointer we can reach the element of 4th row 2nd column (directly) by increment our pointer value.
  - **ptr++; // 4th row 2nd column**
  - **ptr+1; // 4th row 3rd column**
  - **ptr+2; // 4th row 5th column**



Column

Row

	0	1	2	3	4	5
0	300	302	304	306	308	310
1	312	314	316	318	320	322
2	324	326	328	330	332	334
3	336	338	340	342	344	346
4	348	350	352	354	356	358
5	360	362	364	366	368	370
6	372	374	376	378	380	382
7	384	386	388	390	392	394
8	396	398	400	402	404	406

Memory address

Passing Arguments to Functions by Reference with Pointers

- Three ways to pass arguments to a function
  - Pass-by-value
  - Pass-by-reference with reference arguments
  - Pass-by-reference with pointer arguments
- A function can `return` only one value
- Arguments passed to a function using reference arguments
  - Function can modify original values of arguments

Difference in Reference variable and pointer variable

- For reference variable
  - Pass direct by variable name
  - Get by &var in function
  - User variable name as it is in function to execute

- For pointer variable
  - Pass address of variable
  - Get by pointer variable
  - Use pointers in the function to execute

Quiz # 1

- Write C++ programs to implement Pass-by-value and Pass-by-reference to swap two values

Difference in Reference variable and pointer variable

```
void swap(int& x, int& y)
{
    int z = x;
    x = y;
    y = z;
}
```

```
void swap(int* x, int* y)
{
    int z = *x;
    *x = *y;
    *y = z;
}
```

Difference in Reference variable and pointer variable

```
int main()
{
void centimize(double &);

double var=10.0;
cout<<"var= " << var <<"inches" << endl;
centimize(var);
cout<<"var= " << var <<"centimeters" << endl;
   return 0;
}

void centimize(double& v)
{
            v *=2.54;
}
```

```
int main()
{
void centimize(double *);

double var=10.0;
cout<<"var= " << var <<"inches" << endl;
centimize(&var);
cout<<"var= " << var <<"centimeters" << endl;
   return 0;
}

void centimize(double* v)
{
                 *v *=2.54;
}
```

Difference in Reference variable and pointer variable

- A pointer can be re-assigned while reference cannot, and must be assigned at initialization only.
- Pointer can be assigned NULL directly, whereas reference cannot.
- Pointers can iterate over an array, we can use ++ to go to the next item that a pointer is pointing to.
- A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.
- A pointer to a class/struct uses ‘->'(arrow operator) to access it’s members whereas a reference uses a ‘.'(dot operator)
- A pointer needs to be dereferenced with * to access the memory location it points to, whereas a reference can be used directly.

USING CONST WITH POINTERS

- `const` qualifier
  - Indicates that value of variable should not be modified
  - `const` used when function does not need to change the variable's value
- Principle of least privilege
  - Award function enough access to accomplish task, but no more
  - Example
    - A function that prints the elements of an array, takes array and `int` indicating length
      - Array contents are not changed – should be `const`
      - Array length is not changed – should be `const`

Using const with Pointers (Cont.)

- Four ways to pass pointer to function
  1. Nonconstant pointer to nonconstant data
     - Highest amount of access
     - Data can be modified through the dereferenced pointer
     - Pointer can be modified to point to other data
       - Pointer arithmetic
         - Operator ++ moves array pointer to the next element
     - Its declaration does not include `const` qualifier

```
void app(int *x) {
    for(int i=0;i<5;i++){
          cout<<(*x)++<<" ";
          x++;
    }
    return;
}
```

Using const with Pointers (Cont.)

- Four ways to pass pointer to function (Cont.)

  2. Nonconstant pointer to constant data

     - Pointer can be modified to point to any appropriate data item
     - Data cannot be modified through this pointer
     - Provides the performance of pass-by-reference and the protection of pass-by-value

```
void app(const int *x) {
   for(int i=0;i<5;i++){
        cout<<*x<<" ";
        x++;
   }
   return;
}
```

8.5 Using const with Pointers (Cont.)

- Four ways to pass pointer to function (Cont.)
  - 3. Constant pointer to nonconstant data
    - Always points to the same memory location
      - Can only access other elements using subscript notation
    - Data can be modified through the pointer
    - Default for an array name
      - Can be used by a function to receive an array argument
    - Must be initialized when declared

```
void app( int *const x) {
    for(int i=0;i<5;i++){
        cout<<(*x)++<<" ";
    }
    return;
}
```

8.5 Using const with Pointers (Cont.)

- Four ways to pass pointer to function (Cont.)
  4. Constant pointer to constant data
     - Least amount of access
     - Always points to the same memory location
     - Data cannot be modified using this pointer

```
void app(const int *const x) {
    for(int i=0;i<5;i++){
        cout<<(*x)<<" ";
    }
    return;
}
```

Home Work

- Bubble Sort
- [21 6 32 78 32 6 98 0 34 89 23 56 87]
- Pass by reference using pointers

Software Engineering Observation

- When passing an array to a function, also pass the size of the array (rather than building into the function knowledge of the array size). This makes the function more reusable.

SIZEOF OPERATORS

- sizeof operator
  - Returns size of operand in bytes
  - For arrays, sizeof returns

    (size of 1 element) * (number of elements)
  - If sizeof(int) returns 4 then

```
        int myArray[ 10 ];
        cout << sizeof( myArray );
```

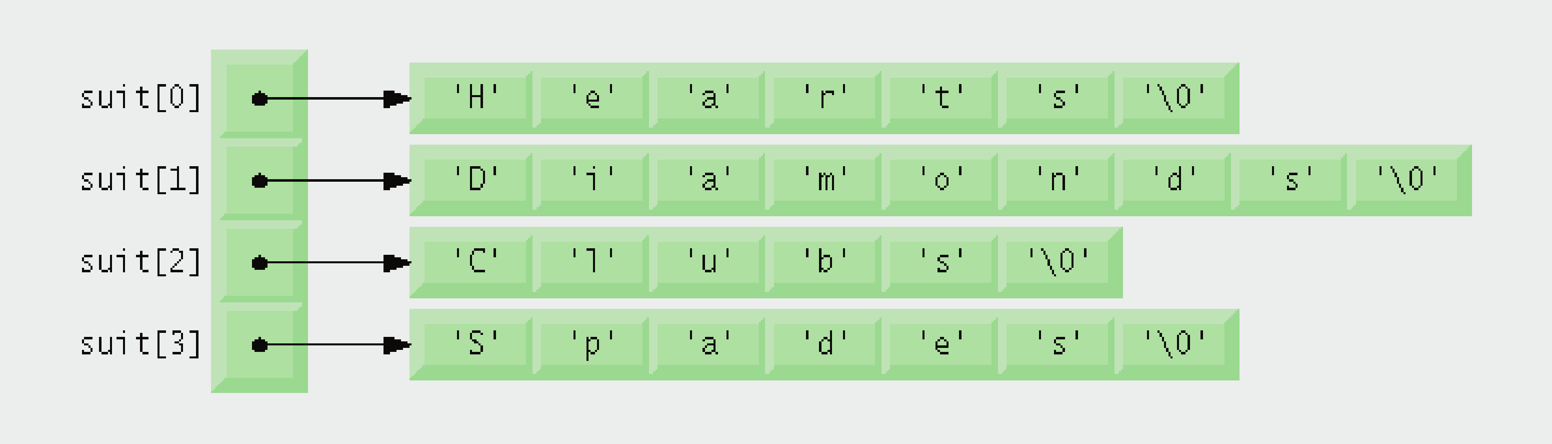
  will print 40
  - Can be used with
    - Variable names
    - Constant values
- How to find the number of elements present in the array. Hint:use sizeof function only

```cpp
// Fig. 8.17: fig08_17.cpp
// Demonstrating the sizeof operator.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   char c; // variable of type char
   short s; // variable of type short
   int i; // variable of type int
   long l; // variable of type long
   float f; // variable of type float
   double d; // variable of type double
   long double ld; // variable of type long double
   int array[ 20 ]; // array of int
   int *ptr = array; // variable of type int *

   cout << "sizeof c = " << sizeof c
      << "\tsizeof(char) = " << sizeof( char )
      << "\nsizeof s = " << sizeof s
      << "\tsizeof(short) = " << sizeof( short )
      << "\nsizeof i = " << sizeof i
      << "\tsizeof(int) = " << sizeof( int )
      << "\nsizeof l = " << sizeof l
      << "\tsizeof(long) = " << sizeof( long )
      << "\nsizeof f = " << sizeof f
      << "\tsizeof(float) = " << sizeof( float )
      << "\nsizeof d = " << sizeof d
      << "\tsizeof(double) = " << sizeof( double )
      << "\nsizeof ld = " << sizeof ld
      << "\tsizeof(long double) = " << sizeof( long double )
      << "\nsizeof array = " << sizeof array
      << "\nsizeof ptr = " << sizeof ptr << endl;
   return 0; // indicates successful termination
} // end main
```

Arrays of Pointers

- Arrays can contain pointers
  - Commonly used to store array of strings (string array)
    - Array does not store strings, only pointers to strings
    - Example

      -
        ```
        const char *suit[ 4 ] =
        { "Hearts", "Diamonds", "Clubs", "Spades" };
        ```
        - Each element of `suit` points to a `char *` (string)
    - `suit` array has fixed size (4), but strings can be of any size
    - Commonly used with command-line arguments to function `main`


suit[0]
'H' 'e' 'a' 'r' 't' 's' '\0'
suit[1]
'D' 'i' 'a' 'm' 'o' 'n' 'd' 's' '\0'
suit[2]
'C' 'l' 'u' 'b' 's' '\0'
suit[3]
'S' 'p' 'a' 'd' 'e' 's' '\0'

Graphical representation of the suit array.

Fundamentals of Characters and Pointer-Based Strings (Cont.)

- String assignment
  - Character array
    - `char color[] = "blue";`
      - Creates 5 element `char` array `color`
        - Last element is `'\0'`
  - Variable of type `char *`
    - `char *colorPtr = "blue";`
      - Creates pointer `colorPtr` to letter `b` in string `"blue"`
        - `"blue"` somewhere in memory
  - Alternative for character array
    - `char color[] = { 'b', 'l', 'u', 'e', '\0' };`

- Some Examples (CodeProject)
- *http://www.codeproject.com/Articles/11560/Pointers-Usage-in-C-Beginners-to-Advanced*

- Questions?