Lecture Contents:

- Pointers
  - Pointer variables
  - Memory management

- Dynamic Arrays
  - Creating and using
  - Pointer arithmetic
  - Arrays & functions
  - 2D dynamic arrays
**Introduction**

- **Pointer definition:**
  - Memory address of a variable

- **Recall: memory divided**
  - Numbered memory locations
  - Addresses used as name for variable

- **You've used pointers already!**
  - Call-by-reference parameters
    - Address of actual argument was passed

**Pointer Variables**

- **Pointers are "typed"**
  - Can store pointer in variable
  - **Not int, double, etc.**
    - Instead: A POINTER to int, double, etc.!
  - Allocated in the heap
    - also called "free store"
Declaring Pointer Variables

- Example:
  
  ```c
  double *p;
  ```

  - "*" is added before variable name
  - `p` is declared as a "pointer to double" variable
  - Can hold only pointers to variables of type double
    - Not other types!

- int *p1, *p2, v1, v2;
  
  - "*" must be used before each variable
  - `p1, p2` hold pointers to `int` variables
  - `v1, v2` are ordinary `int` variables

---

& Operator

- The "address of" operator

- Example
  
  ```c
  int val = 5;
  int *ptr = &val;
  ```

  - "ptr equals address of val"
  - Or "ptr points to val"

- Recall: call-by-reference
  
  - No coincidence!
  - call-by-reference parameters pass "address of" the actual argument
Recall:

```c
int *p1, *p2, v1, v2;
p1 = &v1;
```

Two ways to refer to v1 now:

- Variable v1 itself:
  ```c
  cout << v1;
  ```
- Via pointer p1:
  ```c
  cout << *p1;
  ```

Dereference operator, *

- Pointer variable "de-referenced"
- Means: "Get data that p1 points to"
"Pointing to" Example

- **Consider:**
  - \( v_1 = 0; \)
  - \( p_1 = &v_1; \)
  - \( *p_1 = 42; \)
  - \( cout << v_1 << endl; \)
  - \( cout << *p_1 << endl; \)

- **Produces output:**
  - 42
  - 42

- \( p_1 \) and \( v_1 \) refer to same variable

Pointer Assignments

- **Pointer variables can be "assigned":**
  - ```c++
  int *p1, *p2;
  p2 = p1;
  ```
  - Assigns one pointer to another
  - "Make \( p_2 \) point to where \( p_1 \) points"

- **Do not confuse with:**
  - ```c++
  *p1 = *p2;
  ```
  - Assigns "value pointed to" by \( p_1 \), to "value pointed to" by \( p_2 \)


**Pointer Assignments**

Display 10.1  Uses of the Assignment Operator with Pointer Variables

\[ p1 = p2; \]

Before:

\[
\begin{array}{c}
\text{p1} \\
\text{p2}
\end{array}
\]

\[
\begin{array}{c}
8 \\
9
\end{array}
\]

After:

\[
\begin{array}{c}
\text{p1} \\
\text{p2}
\end{array}
\]

\[
\begin{array}{c}
8 \\
9
\end{array}
\]

\[ *p1 = *p2; \]

Before:

\[
\begin{array}{c}
\text{p1} \\
\text{p2}
\end{array}
\]

\[
\begin{array}{c}
8 \\
9
\end{array}
\]

After:

\[
\begin{array}{c}
\text{p1} \\
\text{p2}
\end{array}
\]

\[
\begin{array}{c}
9 \\
9
\end{array}
\]

**Pointing**

- **Terminology, view**
  - Talk of "pointing", not "addresses"
  - Pointer variable "points to" ordinary variable
  - Makes visualization clearer
    - "see" memory references
    - Arrows

Dr. Amal Khalifa, 2012
The **new** Operator

- **Operator** `new` creates variables
- Can dynamically allocate variables

```
p1 = new int;
```
- Creates new "nameless" variable, and assigns `p1` to "point to" it
- No identifiers to refer to them
  - *Just a pointer!*
- Value can be accessed with `*p1`
  - Use just like ordinary variable

---

**Basic Pointer Manipulations Example:**

```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
}
```
Basic Pointer Manipulations Example:

```
16    p1 = new int;
17    *p1 = 88;
18    cout << "*p1 == " << *p1 << endl;
19    cout << "*p2 == " << *p2 << endl;
20    cout << "Hope you got the point of this example!\n"; 
21    return 0;
22 }
```

**SAMPLE DIALOGUE**

*\(\text{p1} == 42\)*  
*\(\text{p2} == 42\)*  
*\(\text{p1} == 53\)*  
*\(\text{p2} == 53\)*  
*\(\text{p1} == 88\)*  
*\(\text{p2} == 53\)*  
Hope you got the point of this example!

---

**Basic Pointer Manipulations**

(a) Before space created  
(b) The `new` operator dynamically allocates space  
(c) Value assigned to a nameless variable  
(d) Pointing to the same location  
(e) Changing value!!  
(f) Allocating new space  
(g) Assigning values
Memory Management

- Too many new allocations →
  - Too much space needed
  - Heap (free store) becomes "full" →
  - Future "new" operations will fail → NULL

Memory management
- Heap size varies with implementations → typically large
- Memory IS finite
  - Regardless of how much there is!
- Still good practice

Checking new Success

- Older compilers:
  - Test if null returned by call to new:
    ```
    int *p;
    p = new int;
    if (p == NULL)
    {
        cout << "Error: Insufficient memory.\n";
        exit(1);
    }
    ```
  - If new succeeded, program continues

- Newer compilers:
  - If new operation fails:
    - Program terminates automatically
    - Produces error message
**delete Operator**

- De-allocate dynamic memory
  - When no longer needed
  - Returns memory to “free store”
  - Example:
    ```
    int *p;
    p = new int(5);
    ... //Some processing...
    delete p;
    ```
  - De-allocates dynamic memory "pointed to by pointer p"
    - Literally "destroys" memory

**Dangling Pointers**

- delete p;
  - Destroys dynamic memory
  - But p still points there!
    - Called "dangling pointer"
  - If p is then de-referenced ( *p )
    - Unpredicatable results!
    - Often disastrous!

- Avoid dangling pointers
  - Assign pointer to NULL after delete:
    ```
    delete p;
    p = NULL;
    ```
Pointers and Functions

- **Pointers**
  - Are full-fledged types
  - Can be used just like other types
  - Can be function parameters
  - Can be returned from functions

- **Example:**
  ```
  int* findOtherPointer(int* p);
  ```
  - This function declaration:
    - Has "pointer to an int" parameter
    - Returns "pointer to an int" variable
Example: Call-by-value Pointers

Write a “swap” function that takes two integer pointers.

Dynamic vs. Automatic Variables

- **Dynamic variables**
  - Created with `new` operator
  - Created and destroyed while program runs

- **Local variables**
  - Declared within function definition
  - Not dynamic
    - Created when function is called
    - Destroyed when function call completes
  - Often called "automatic" variables
    - Properties controlled for you
  - What about pointers as parameters??
    - Passing by value or by reference??
Dynamic Arrays

Array variables
- Array limitations
  - Must specify size first
  - May not know until program runs!
  - Really pointer variables!
- Must "estimate" maximum size needed
  - Sometimes OK, sometimes not
  - "Wastes" memory

Dynamic arrays
- Dynamic arrays
  - Allocated in the heap
  - Can grow and shrink as needed
  - Allocated at run time
Array Variables
• Recall: arrays stored in memory sequentially
  • Array variable "refers to" first indexed variable
  • So array variable is a kind of pointer variable!

Example:
```c
int a[6];
int * p;
```
• a and p are both pointer variables!

Creating Dynamic Arrays

• Very simple!
• Use new operator
  • Dynamically allocate with pointer variable
  • Treat like standard arrays

Example:
```c
double * a;
a = new double[5]; //Size in brackets
```
• Creates dynamically allocated array variable d, with ten elements, base type double
Deleting Dynamic Arrays

- Allocated dynamically at run-time
  - So should be destroyed at run-time

- Example:
  ```
  d = new double[10];
  ... //Processing
  delete [] d;
  d = NULL;
  ```
  - De-allocates all memory for dynamic array
  - Brackets indicate "array" is there
  - Recall: d still points there!
    - Should set d = NULL;

Array Variables → Pointers

- a and p are pointer variables
  - Can perform assignments:
    ```
    p = a;  // Legal.
    ```
    - p now points where a points
      - To first indexed variable of array a
    - a = p;  // ILLEGAL!
      - Array pointer is CONSTANT pointer!
      - Array was allocated in memory already
      - Variable a MUST point there…always!
        - Cannot be changed!
**Pointer Arithmetic**

- Pointer $\rightarrow$ address $\rightarrow$ integer
  - Cannot be used as numbers, even though it "is a" number
  - "Address" arithmetic
    - Can use ++ and -- on pointers
    - Only addition/subtraction on pointers
    - No multiplication, division

---

**Example:**

```c
int array[3] = {0, 1, 2};
int *ptr = array;
```

- `*ptr;` evaluates to address of array[0]
- `*ptr++;` evaluates to address of array[1]
- `*ptr+=2;` evaluates to address of array[2]
Accessing elements

Step thru

• Use pointer arithmetic!
  • for (int i = 0; i < arraySize; i++)
    cout << *(d + i) << " " ;

indexing

• Ordinary array index !
  • for (int i = 0; i < arraySize; i++)
    cout << d[i] << " " ;

Example:

Use a dynamic array to get the scores of 5 students and show how much each score differs from the highest score.

SAMPLE DIALOGUE
Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4
Arrays as function parameters

- Arrays == address
- Pointers == address
- Example:

```c
void FillArray(int* a, int Size);
int Search(int* Arr, int Size, int Target);
```

Example:

Rewrite your program to use two functions:
- a function finds the highest score,
- Another function to display the result

Hint:
the parameters can be either array or pointer type.

Sample Dialogue:
Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4
Function that Returns an Array

- Array type NOT allowed as return-type of function

- Example:
  ```
  int [] someFunction(); // ILLEGAL!
  ```

- Instead return pointer to array base type:
  ```
  int* someFunction(); // LEGAL!
  ```

Example: the doublers

Write a function that takes an integer array and returns a pointer to another array of the same size in which each element is double the corresponding element in the input array.
Two dimensional “dynamic” arrays

- Array of Arrays
- Pointer to pointers

A 5x4 integer array

Code skeleton

```c++
//pointer declaration
int **Matrix;

//memory allocated for elements of rows.
Matrix = new int* [ROWS] ;

//memory allocated for elements of each column.
for( int i = 0 ; i < ROWS ; i++ )
    Matrix[i] = new int[COLUMNS];

//do some processing...
for( int i = 0 ; i < ROWS ; i++ )
    for( int j = 0 ; j < COLUMNS; j++ )
        Matrix[i][j] = i * j;

//free the allocated memory
for( int i = 0 ; i < ROWS ; i++ )
    delete [] Matrix[i] ;
delete [] Matrix ;
```
Example: Transposing a matrix

In **linear algebra**, the **transpose** of a **matrix** $A$ is another matrix $A^T$ (also written $A'$, $A^\text{tr}$ or $A^t$). It is created by writing the rows of $A$ as the columns of $A^T$.

Write a function that computes the transpose of a matrix.

Example: TWO-DIMENSIONAL GRADING PROGRAM

Write a program that uses a two-dimensional **dynamic** array named `grade` to store the grade records for a small class.

*The first array index is used to designate a student, and the second array index is used to designate a Quiz.*

The program should be able to:

- Store the grade data for a number of students in various quizzes
- Compute the average quiz score for each of the students
- Compute the average score on each quiz.
- Display the grade records for the entire class

<table>
<thead>
<tr>
<th>Student</th>
<th>Ave</th>
<th>Quiz1</th>
<th>Quiz2</th>
<th>Quiz3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7.7</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7.3</td>
<td>8</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

Quiz Average = 7.0, 5.0, 7.5
That’s all for today!!

Thanks.....