Pointers

Accessing memory and dynamic memory allocation
Goals

By the end of this lesson you will be able to:

- Understand the concept and usefulness of pointers
- Understand syntax related to using pointers
- Understand syntax related to dynamic memory allocation
- Understand the duality of pointers and arrays
What are pointers?

- A pointer is a specific piece of data, specifically a memory address, that “points” to some other data that we may use.
- Pointers as a larger topic includes things such as understanding dynamic memory allocation and access.
- Pointers and dynamic memory concepts allow you to do some very powerful things, however it is easier to make mistakes with them than other programming concepts.
What’s the use?

• Pointers are necessary for a variety of things such as:
  – Dynamic memory allocation (e.g. if you do not know when writing/compiling the program how much memory you need)
  – Creation and use of more complex data structures
  – Handling large amounts of data faster and using less space

• Behind the scenes for a lot of things in C++ there is heavy pointer/dynamic memory allocation usage
Creating a pointer

- Pointers can be created as such:
  ```
  int *p;  // p is a pointer to an int
  ```
- Note that if you do the following:
  ```
  int *p, q;
  then p is a pointer and q is a regular integer
  ```
Suppose \( x \) is a variable of any datatype (int, double, string, or any user created type such as a typedef, enum, or struct).

To obtain the memory address where the value of \( x \) is stored at, use the reference operator, \&, as such: \&x

What can you do with this?

- Display it:
  \[
  \text{cout} \ll \&x \ll \text{endl};
  \]

- Set a pointer equal to it:
  \[
  \text{int } *p = \&x; \quad \text{OR} \quad \text{int } *p;
  \text{p} = \&x;
  \]
Dereference operator

- If you have a pointer that has a value (a memory address) stored in it, how can you access that memory? Use the dereference operator, *, as such: *p

- What can you do with this?
  - Access (to display or to change the value):
    ```cpp
    int *p, x = 5;
p = &x;
cout << x << ' ' << *p << endl; // 5 5
    x++;
cout << x << ' ' << *p << endl; // 6 6
    (*p)++;
cout << x << ' ' << *p << endl; // 7 7
    ```
Don’t get confused!

- You know that & is used in functions to “pass by reference”
- When using & for references in the context of pointers it has a different (but similar) meaning
- & has other uses, such as aliases (similar to the above uses) and bitwise AND (totally different use)
C-style passing by reference

- In C there is no “pass by reference” so passing by reference is accomplished by passing a pointer to a function
- It is more conventional to pass by reference in C++ when possible
Dynamic arrays

- In C++ you must declare array sizes at compile time, but by using dynamic memory allocation you can declare them at run time instead.

- To allocate:
  ```cpp
  int *x = new int[expr];
  ```

- To deallocate:
  ```cpp
  delete [] x;
  ```

- You should deallocate memory when done with it, otherwise this causes a **memory leak**.
Dynamic arrays

- Use dynamic arrays in the same way that you use normal arrays.
- Alternately, use * and + to access array elements.
  int *x = new int[10];
  x[3] = 5;
  is the same as:
  *(x + 3) = 5;
- This notation also works for normal arrays.
  int x[10];
  *(x + 3) = 5; // same as x[3] = 5
Segmentation faults

- Dealing with arrays, pointers, and dynamically allocated memory can easily lead to errors, most notably segmentation faults.

- Be careful not to dereference a pointer (memory address) that you do not have access to:
  - Attempting to access an element outside the bounds of the array
  - Attempting to use a pointer that you have already deleted
• Get input from the user to specify a number of values, create an array of that specified size, fill it with random numbers between 0 and 1000, then calculate the standard deviation of the values in the array