Review from Monday’s Lecture

- Types and defining new types
- Class declaration: member variables and member functions
- Using the class member functions
- Class scope
- Member function implementation
- Classes vs. structs
- Rewriting the week 4 homework solution using classes

Today’s Class — C++ Classes; Pointers

Koenig and Moo: Section 10.1

- Rewriting the week 4 homework solution, continued
- Pointers
- Arrays, array initialization and string literals
- Arrays and pointers

Reminder

- Use examples from class and the exercises from labs and homeworks when writing your programs. You can cut and paste and modify the code that has been provided.
- If you have a question, first look back through the examples to find something similar.
- We will try to generate more examples to help.
Overview

• Pointers store memory addresses

• Pointers are the iterators for arrays

• Pointers are also the primitive mechanism underlying vector iterators, list iterators, and map iterators.

• Dynamic memory is accessed through pointers.
**Pointer Example**

Consider the following code snippet:

```c
float x = 15.5;
float *p;
p = &x;
*p = 72;
if ( x > 20 )
    cout << "Bigger\n";
else
    cout << "Smaller\n";
```

The output is

```
Bigger
```

because `x == 72.0`. What’s going on?

**Pointer Variables and Memory Access**

- `x` is an ordinary integer, but `p` is a pointer that can hold the memory address of an integer.

- The difference is explained in the following picture:
• Every variable is attached to a location in memory. This is where the value of that variable is stored. Hence, we draw a picture with the variable name next to a box that represents the memory location.

• Each memory location also has an address, which is itself just an index into the giant array that is the computer memory.

• The value stored in a pointer variable is an address in memory. In this case, the statement

\[ p = \&x; \]

Takes the address of x’s memory location data and stores it (the address) in the memory location associated with p.

• Since the value of this address is much less important than the fact that the address is x’s memory location, we depict the address with an arrow.

• The statement

\[ *p = 72; \]

causes the computer to get the memory location stored at p, then go to that memory location, and store 72 there. This writes the 72 in x’s location.

• The distinction between p and *p for pointers is just like the distinction between p and *p for iterators.

**Defining Pointer Variables**

```c
int * p, q;
float *s, *t;
```

• Here, p, s and t are all pointers, but q is NOT. You need the * before each variable name.

• There is no initialization of pointer variables in this two-line snippet, so a statement of the form

\[ *p = 15; \]

will cause some form of “memory exception”.

```c
```
Operations on Pointers

- The unary operator \* in the expression \*p is the “dereferencing operator”. It means “follow the pointer”

- The unary operator & in the expression &x means “take the memory address of.”

- Pointers can be assigned. We will look at the following example in detail.

  ```
  float x=5, y=9;
  float *p = &x, *q = &y;
  *p = 17.0;
  *q = *p;
  q = p;
  *q = 13.0;
  ```

  What are the values of x and y at the end?

- Assignments of integers or floats to pointers and assignments mixing pointers of different types are illegal. Continuing with the above example:

  ```
  int *r;
  r = q;     // Illegal: different pointer types;
  p = 35.1;  // Illegal: float assigned to a pointer
  ```

- Comparisons between pointers of the form

  ```
  if ( p == q )
  ```

  or

  ```
  if ( p != q )
  ```

  are legal and very useful! Less than and greater than comparisons are also allowed. These are used when the pointers are to array elements.
Null Pointers

- Pointers that don’t point anywhere useful should be given the value 0. This is a legal pointer value.
  - Many compilers define NULL to be a special identifier equal to 0.
- Comparing a pointer to 0 is very useful. It indicates whether or not a pointer has a legal address. For example,

```c++
if ( p != 0 )
    cout << *p << endl.
```

- Dereferencing a null pointer leads to memory exceptions.

Arrays

- Here’s a quick example to remind you about how to use an array:

```c++
const int n = 10;
double a[n];
int i;
for ( i=0; i<n; ++i )
    a[i] = sqrt( double(i) );
```

- The size of array `a` is fixed at compile time. That’s why vectors are preferrable, which can grow and shrink dynamically, in response to the demands of the application.

Array Initialization

- We can initialize arrays when we define them. We did this in the file `Date.cpp`

```c++
```

This gave us a kind of a table to quickly determine the number of days in each month.

- We do not need the 12 here, the compiler figures it out automatically:
const int numDays[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

• We can do this for non-const arrays as well.

Character Arrays and String Literals

• In the line
  
  ```cpp
  cout << "Hello!" << endl;
  ``"

  "Hello!" is a string literal. It is also an array of characters.

• The array can be initialized as:
  
  ```cpp
  char h[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
  ```

  or as

  ```cpp
  char h[] = "Hello!";
  ```

  In either case, h has 7 characters, the last one being the null character.

• The C and C++ languages have many functions for manipulating these “C-style strings”. We don’t study them much here because the standard string library is much more logical and easier to use.

• One place we will use them is in file names and command-line arguments, which we will study next week.

Conversion Between Standard Strings and C-Style String Literals

• We have been creating standard strings from C-style strings all semester, for example:

  ```cpp
  string s( "Hello!" );
  ```

  or

  ```cpp
  string s( h );
  ```

  where h is as defined above.

• You can obtain the C-style string from a standard string using the member function `c_str`, as in `s.c_str()`.
Pointers As Array Iterators

- Pointers are the iterators for arrays.
- The above code may be rewritten:
  
  ```
  const int n = 10;
  double a[n];
  double *p;
  for ( p=a; p<a+n; ++p )
    *p = sqrt( p-a );
  ```

  This may seem like a complicated piece of code, but most of it is similar to what you have already seen with vectors and vector iterators.

- The assignment:
  
  ```
  p = a;
  ```

  is different, however. It takes the address of the start of the array and assigns it to `p`.

  - This illustrates the important fact that the name of an array is in fact a **pointer to the start of a block of memory**.
  - We will come back to this several times!

  The line `p=a` could also have been written:

  ```
  p = &a[0];
  ```

- The test `p<a+n` checks to see if the value of the pointer (the address) is less than `n` array locations beyond the start of the array.
  
  - We could also have used the test `p != a+n`

- By incrementing, `++p`, we make `p` point to the next location in the array.

- In the assignment
  
  ```
  *p = sqrt( p-a )
  ```

  `p-a` is the number of array locations between `p` and the start. This is an integer. The square root of this value is assigned to `*p`.

- We will draw a picture of this in class.

- Do you see how pointers are just like vector iterators?
Exercises

1. Write code to sort an array into increasing order.

2. Write code to print the array \( a \) backwards, using pointers.

3. Write code to print every other value of the array, again using pointers.