Computer Science II — CSci 1200
Lecture 5
Pointers, Arrays, Pointer Arithmetic

Review from Lecture 4

- Extended example of student grading program
- Passing comparison functions to `sort`
- Non-member operators

Reading: Lectures 1-4

Several students have asked about reading material with to accompany the lecture notes. Here are references to some of the topics covered in Lectures 1-4. In addition, material on C++ constructs is covered almost any introductory C++ textbook.

Topic	**Ford & Topp**	**Koenig & Moo**
Order notation	127-139	
Strings	pp. 28-35	Ch. 1 & 2
Vectors	Ch. 4	Ch. 3
Recursion	pp. 146-168	
Classes	pp. 8-27, 53-81	Sec. 4.2-4.4, Ch. 9

Today's Lecture — Pointers and Arrays

- Pointers
- Arrays, array initialization and string literals
- Arrays and pointers

Reading: Ford & Topp, pp. 219-228; Koenig & Moo, Sec. 10.1

Overview

- Pointers store memory addresses.
- They can be used to access the values stored at their stored memory address.
- They can be incremented, decremented, added and subtracted.
- Dynamic memory is accessed through pointers.
- Pointers are also the primitive mechanism underlying vector iterators, which we have used with `std::sort` and will continue to use many times throughout the semester.

Pointer Example

Consider the following code segment:

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

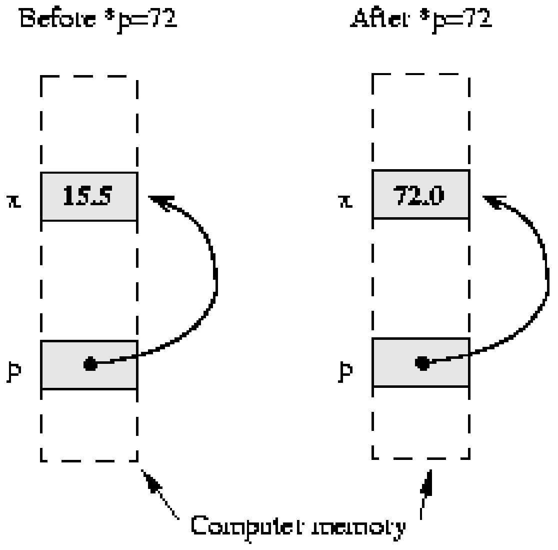
The output is

```
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 variable.
- 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 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 rather than with its actual value.
- 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.

  - `*p` is an *l-value* here.

Defining Pointer Variables

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

- Here, `p`, `s` and `t` are all pointer variables (pointers, for short), but `q` is NOT. You need the * before each variable name.
- There is no initialization of pointer variables in this two-line sequence, so a statement of the form

  ```
  *p = 15;
  ```

  will cause some form of "memory exception". This means your program will crash!

Operations on Pointers

- The unary operator * in the expression `*p` is the "dereferencing operator". It means "follow the pointer". `*p` is either an l-value or an r-value, depending on which side of the `=` it appears on.
- The unary operator & in the expression `&x` means "take the memory address of."
- Pointers can be assigned. This just copies memory addresses as though they were values (which they are). 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 useful only when the pointers are to locations within an array.

Exercise

What is the output of the following code sequence?

```
int x = 10, y = 15;
int *a = &x;
cout << x << " " << y << endl;
int *b = &y;
*a = x * *b;
cout << x << " " << y << endl;
int *c = b;
*c = 25;
cout << x << " " << y << endl;
```

Null Pointers

- Pointers that don't (yet) point anywhere useful should be given the value 0, a legal pointer value.
  - Most compilers define NULL to be a special pointer equal to 0.
- Comparing a pointer to 0 is very useful. It can be used to indicate whether or not a pointer has a legal address. (But don't make the mistake of assuming pointers are automatically initialized to 0.) For example,

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

  tests to see if p is pointing somewhere that appears to be useful before accessing the value stored at that location.
- Dereferencing a null pointer leads to memory exceptions (program crashes).

Arrays

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

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

- Remember: the size of array `a` is fixed at compile time. vectors act like arrays, but they can grow and shrink dynamically in response to the demands of the application.

Stepping through Arryas with Pointers

- Pointers are the iterators for arrays.
- The array initialization code above,

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

  can be rewritten as

  ```
  const int n = 10;
  double a[n];
  double *p;
  for ( p=a; p<a+n; ++p )
     *p = sqrt( p-a );
  ```

- The assignment

  ```
  p = a;
  ```

  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];
  ```

  which means "find the location of `a[0]` and take its address".

- 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.

Exercises

For each of the following problems, you may only use pointers and not subscripting:

1. Write code to print the array `a` backwards, using pointers.
2. Write code to print every other value of the array `a`, again using pointers.
3. Write a function that checks whether the contents of an array are sorted into increasing order. The function must accept two arguments: a pointer (to the start of the array), and an integer indicating the size of the array.

Character Arrays and String Literals

The remaining notes will only be covered in lecture if there is time. Mostly, they are for your own reference.

- In the line

  ```
  cout << "Hello!" << endl;
  ```

  `"Hello!"` is a *string literal.* It is also an array of characters (with no associated variable name).
- A char array can be initialized as:

  ```
  char h[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
  ```

  or as

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

  In either case, array `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 do not study them much anymore because the standard string library is much more logical and easier to use.
- One place we do use them is in file names and command-line arguments, as you have already seen.
- We have been creating standard strings from C-style strings all semester, for example:

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

  or

  ```
  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()`.