CSCI-1200 Data Structures — Spring 2013
Lecture 5 — Pointers, Arrays, Pointer Arithmetic

Announcements: Test 1 Information

- Test 1 will be held **Tuesday, February 12th, 2012 in 2 groups from 8-9:50am or 10-11:50am in Sage 3303**. *Schedule conflicts will be collected tomorrow in lab.* No make-ups will be given except for pre-approved absense or illness, and a written excuse from the Dean of Students or the Student Experience office or the RPI Health Center will be required.
- Coverage: Lectures 1-6, Labs 1-3, and Homeworks 1-2.
- Closed-book and closed-notes *except for 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed.* Computers, cell-phones, calculators, PDAs, music players, etc. are not permitted and must be turned off and placed under your desk.
- All students must bring their Rensselaer photo ID card.
- Practice problems from previous exams are available on the course website. Solutions to the problems will be posted later. The best way to prepare is to completely work through and write out your solution to each problem, *before* looking at the answers.
- The exam *will* involve handwriting code on paper (and other short answer problem solving). Neat legible handwriting is appreciated. We will somewhat forgiving to minor syntax errors – it will be graded by humans not computers :)

Review from Last Week

- C++ class syntax, designing classes, classes vs. structs; Passing comparison functions to `sort`; Non-member operators.

Today's Lecture — Pointers and Arrays

Optional Reading: Ford & Topp pp. 219-228; Koenig and Moo, Section 10.1

- 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 use more extensively throughout the semester.

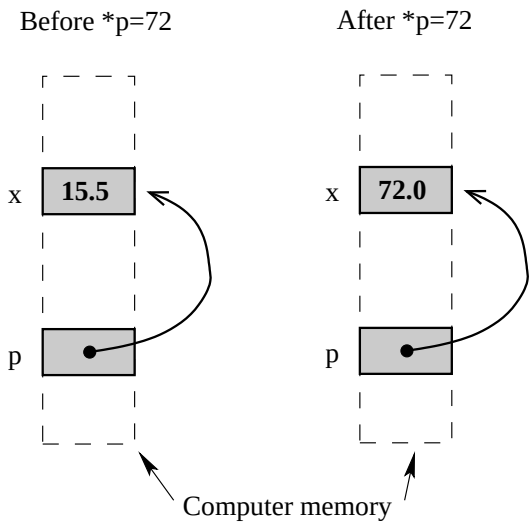
1.1 Pointer Example

- Consider the following code segment:

```
float x = 15.5;
float *p; /* equiv:  float* p;  or  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?



1.2 Pointer Variables and Memory Access

- `x` is an ordinary float, but `p` is a pointer that can hold the memory address of a float variable. The difference is explained in the picture above.
- 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. 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.
- 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.

  Note: `*p` is an *l-value* in the above expression.

1.3 Defining Pointer Variables

- In the example below, `p`, `s` and `t` are all pointer variables (pointers, for short), but `q` is NOT. You need the * before each variable name.

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

- There is no initialization of pointer variables in this two-line sequence, so the statement below is dangerous, and may cause your program to crash! (It won't crash if the uninitialized value happens to be a legal address.)

```
*p = 15;
```

1.4 Operations on Pointers

- The unary (single argument/operand) operator * in the expression `*p` is the "dereferencing operator". It means "follow the pointer" `*p` can be 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). Let's work through the example below (and draw a picture!). What are the values of `x` and `y` at the end?

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

- 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.

1.5 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;
```

1.6 Null Pointers

- Pointers that don't (yet) point anywhere useful should be explicitly assigned the value 0, a legal pointer value.
  - Most compilers define NULL to be a special pointer equal to 0.
- Comparing a pointer to NULL 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 NULL.) For example,

```
if ( p != NULL )
  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 a memory exception (which causes the program to crash).

1.7 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. STL vectors act like arrays, but they can grow and shrink dynamically in response to the demands of the application.

1.8 Stepping through Arrays with Pointers

- Pointers are the *iterators* for arrays.
- The array initialization code above, 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! We could also write this line as:

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

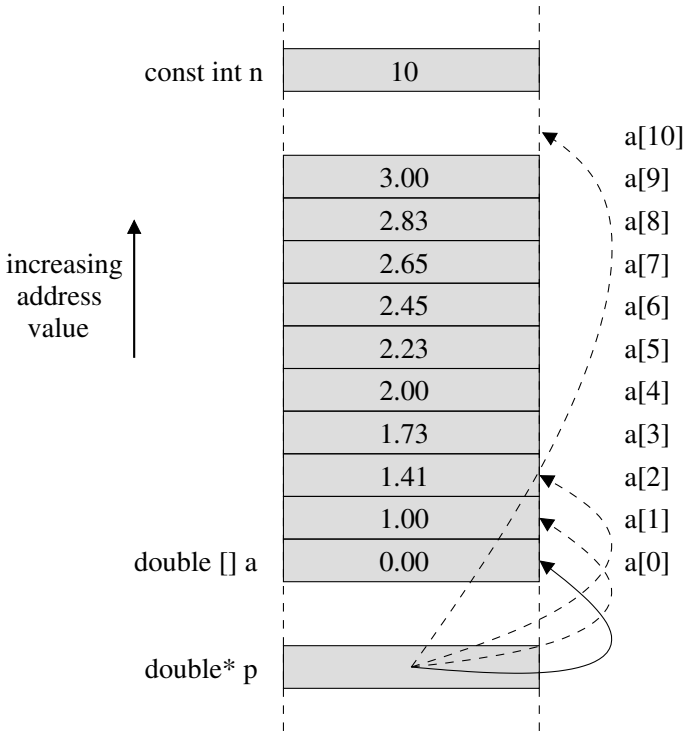
  which means "find the location of `a[0]` and take its address".

- 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`.

Here's a picture to explain this example:



1.9 Sorting an Array

- Arrays may be sorted using `std::sort`, just as vectors may. Pointers are used in place of iterators. For example, if `a` is an array of doubles and there are `n` values in the array, then here's how to sort the values in the array into increasing order:

  ```
  std::sort( a, a+n )
  ```

1.10 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 of doubles 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.

1.11 Character Arrays and String Literals

- In the line below `"Hello!"` is a *string literal* and it is also an array of characters (with no associated variable name).

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

- 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 don't 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.

1.12 Conversion Between Standard Strings and C-Style String Literals

- We have been creating standard strings from C-style strings all semester. Here are 2 different examples:

```
string s1( "Hello!" );
string s2( 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 `s1.c_str()`.

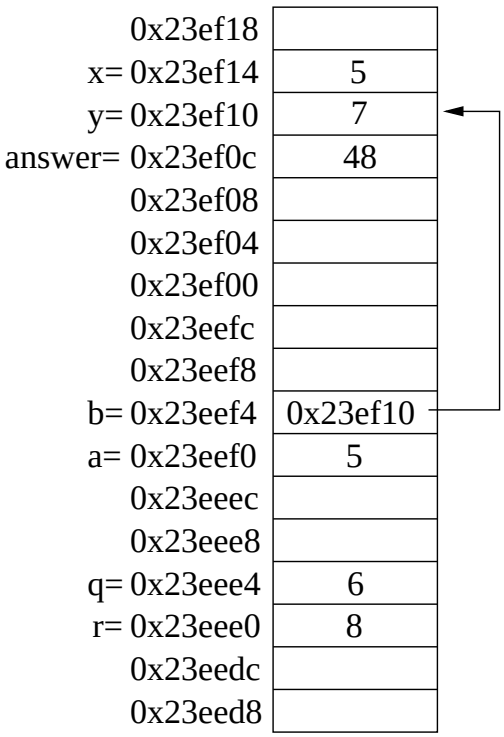
1.13 C Calling Convention

- A *calling convention* is a standardized method for passing arguments between the caller and the function. Calling conventions vary between programming languages, compilers, and computer hardware.
- In C on x86 architectures here is a generalization of what happens:
  1. The caller puts all the arguments on the *stack*, in reverse order.
  2. The caller puts the address of its code on the stack (the *return address*).
  3. Control is transferred to the callee.
  4. The callee puts any local variables on the stack.
  5. The callee does its work and puts the return value in a special *register* (storage location).
  6. The callee removes its local variables from the stack.
  7. Control is transferred by removing the address of the caller from the stack and going there.
  8. The caller removes the arguments from the stack.
- On x86 architectures the addresses on the stack are in descending order. This is not true of all hardware.

1.14 Poking around in the stack

```
int foo(int a, int *b) {
  int q = a+1;
  int r = *b+1;
  cout << "address of a = " << &a << endl;
  cout << "address of b = " << &b << endl;
  cout << "address of q = " << &q << endl;
  cout << "address of r = " << &r << endl;
  cout << "value at " << &a << " = " << a << endl;
  cout << "value at " << &b << " = " << b << endl;
  cout << "value at " << b << " = " << *b << endl;
  cout << "value at " << &q << " = " << q << endl;
  cout << "value at " << &r << " = " << r << endl;
  return q*r;
}

int main() {
  int x = 5;
  int y = 7;
  int answer = foo (x, &y);
  cout << "address of x = " << &x << endl;
  cout << "address of y = " << &y << endl;
  cout << "address of answer = " << &answer << endl;
  cout << "value at " << &x << " = " << x << endl;
  cout << "value at " << &y << " = " << y << endl;
  cout << "value at " << &answer << " = " << answer << endl;
}
```



Sample Output

```
address of a = 0x23eef0
address of b = 0x23eef4
address of q = 0x23eee4
address of r = 0x23eee0
value at 0x23eef0 = 5
value at 0x23eef4 = 0x23ef10
value at 0x23ef10 = 7
value at 0x23eee4 = 6
value at 0x23eee0 = 8
address of x = 0x23ef14
address of y = 0x23ef10
address of answer = 0x23ef0c
value at 0x23ef14 = 5
value at 0x23ef10 = 7
value at 0x23ef0c = 48
```