CSCI-1200 Computer Science II — Spring 2006
Lecture 17 — Pointers, Dynamic Memory, and Command-Line Arguments

Review from Lecture 16

- Pointer variables, arrays, character arrays, and pointers as array iterators.

Today's Lecture — Pointers and Dynamic Memory

- K & M rest of Chapter 10; Malik, 753-781
- Arrays and pointers, different types of memory, and dynamic allocation of arrays.

17.1 Three Types of Memory

- Automatic memory: memory allocation inside a function when you create a variable. This allocates space for local variables in functions and deallocates it when variables go out of scope. For example:

```
int x;
double y;
```

- Static memory: variables allocated statically (with the keyword `static`). They are are not eliminated when they go out of scope. They retain their values, but are only accessible within the scope where they are defined. We will not be discussing these much.

```
static int counter;
```

- Dynamic memory: explicitly allocated as needed. This is the focus of today's lecture.

17.2 Dynamic Memory

- Dynamic memory is:
  - created using the `new` operator,
  - accessed through pointers, and
  - removed through the `delete` operator.
- Here's a simple example involving dynamic allocation of integers:

```
int * p = new int;
*p = 17;
cout << *p << endl;
int * q;
q = new int;
*q = *p;
*p = 27;
cout << *p << " " << *q << endl;
int * temp = q;
q = p;
p = temp;
cout << *p << " " << *q << endl;
delete p;
delete q;
```

- The expression `new int` asks the system for a new chunk of memory that is large enough to hold an integer. Therefore, the statement `int * p = new int;` allocates a new chunk of memory large enough to hold an integer, and stores the memory address of the start of this memory in the pointer variable `p`.

- The statement `delete p;` takes the integer memory pointed by `p` and returns it to the system for re-use.
- In between the `new` and `delete` statements, the memory is treated just like memory for an ordinary variable, except the only way to access it is through pointers. Hence, the manipulation of pointer variables and values is similar to the examples covered in Lecture 16 except that there is no explicitly named variable other than the pointer variable.
- Dynamic allocation of primitives like ints and doubles is not very interesting or significant. What's more important is dynamic allocation of arrays and objects.

17.3 Exercise

- What's the output of the following code? Be sure to draw a picture to help you figure it out.

```
double * p = new double;
*p = 35.1;
double * q = p;
cout << *p << " " << *q << endl;
p = new double;
*p = 27.1;
cout << *p << " " << *q << endl;
*q = 12.5;
cout << *p << " " << *q << endl;
delete p;
delete q;
```

17.4 Dynamic Allocation of Arrays

- Declaring the size of an array at compile time doesn't offer much flexibility. Instead we can *dynamically* allocate an array based on data. This gets us part-way toward the behavior of a vector. Here's an example:

```
int main() {
  cout << "Enter the size of the array: ";
  int n;
  cin >> n;
  double *a = new double[ n ];

  int i;
  for ( i=0; i<n; ++i )
    a[i] = sqrt( i );

  for ( i=0; i<n; ++i )
    if ( double(int(a[i])) == a[i] )
      cout << i << " is a perfect square " << endl;

  delete [] a;
  return 0;
}
```

- Consider the line: `double *a = new double[ n ];`
  - The expression `new double[ n ]` asks the system to *dynamically* allocate enough consecutive memory to hold n double's (usually $8n$ bytes).
  - What's crucially important is that `n` is a variable. Therefore, its value and, as a result, the size of the array are not known until the program is executed. When this happens, the memory must be allocated dynamically.
  - The address of the start of the allocated memory is assigned to the pointer variable `a`.

- After this, `a` is treated as though it is an array. For example: `a[i] = sqrt( i );`
  In fact, the expression `a[i]` is exactly equivalent to the pointer arithmetic and dereferencing expression `*(a+i)` which we have seen several times before.
- After we are done using the array, the line: `delete [] a;` releases the memory allocated for the entire array. Without the `[]`, only the first double would be released.
- Since the program is ending, releasing the memory is not a major concern. In more substantial programs it is ABSOLUTELY CRUCIAL. If we forget to release memory repeatedly the program can be said to have a *memory leak*. Long-running programs with memory leaks will eventually run out of memory and crash.

17.5 Exercises

1. Write code to dynamically allocate an array of `n` integers, point to this array using the integer pointer variable `a`, and then read n values into the array from the stream `cin`.

2. Now, suppose we wanted to write code to double the size of array `a` without losing the values. This requires some work: First allocate an array of size `2*n`, pointed to by integer pointer variable *temp* (which will become `a`). Then copy the `n` values of `a` into the first `n` locations of array `temp`. Finally delete array `a` and assign `temp` to `a`.

Why don't you need to delete `temp`? Note: The code for part 2 of the exercise is very similar to what happens inside the `resize` member function of vectors!

17.6 Dynamic Allocation: Arrays of Class Objects

- We can dynamically allocate arrays of structs and class objects:

```
#include <iostream>

class Foo {
public:
  Foo() {
    static int counter = 1;
    a = counter;
    b = 100.0;
    counter++;
  }
  double value() const { return a*b; }
private:
  long long int a;
  double b;
};

int main() {
  int n;
  cin >> n;
  Foo *things = new Foo[n];
  for (Foo* i = things; i < things+n; i++)
    cout << "Foo stored at: " << (int)i << " has value " << i->value() << endl;
  delete [] things;
}
```

- For class objects, the default constructor (the constructor that takes no arguments) must be defined. Vectors do not require default constructors — another advantage of vectors over arrays.

17.7 Dynamic Allocation Example: The Sieve of Eratosthenes, Revisited

- We return to the problem of finding all primes. The first time we did this we used a `list`. Now we use a dynamically allocated array:

```
// Using Sieve of Eratostenes determine all prime numbersless than or
// equal to an integer provided on the command line.
#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
  // Check usage.  Take n from the 1st argument.  Make sure it is positive.
  if (argc != 2) {
    cerr << "Usage:\n  " << argv[0] << " n\n" << "where n is a positive integer\n";
    return 0;
  }
  int n = atoi(argv[1]);
  if (n <= 0) {
    cerr << "Usage:\n  " << argv[0] << " n\n" << "where n is a positive integer\n";
    return 0;
  }

  // Create and initialize a dynamically allocated array
  bool * is_prime_sieve;
  is_prime_sieve = new bool[n+1];
  int i;
  is_prime_sieve[0] = is_prime_sieve[1] = false;
  for (i=2; i<=n; ++i)  is_prime_sieve[i] = true;

  // Check each integer up to n to see if it is prime.  Output those that are.
  int prime_count = 0;
  for (i = 2; i <= n ; ++i) {
    // If a number is not prime, then we can skip it.
    if (is_prime_sieve[i]) {
      cout << i << " is prime\n";
      ++prime_count;
      // No multiples of i are prime.  Mark these multiples in the array.
      for (int j = 2*i; j <= n; j += i)
        is_prime_sieve[j] = false;
    }
  }
  cout << "\nThere are " << prime_count << " primes <= " << n << endl;

  //  Release the dynamic memory through the delete operator
  delete [] is_prime_sieve;
  return 0;
}
```

- Once n is taken from the command-line, we define a `bool` pointer and then make it point to the start of a dynamically-allocated array of `bool`s. Each entry from 0 to n is used to indicate whether or not the index value is prime.
- After this, `is_prime_sieve` is treated like an ordinary array.
- The dynamic memory is deleted at the end.

17.8 Exercise: Dynamic Allocation Arrays vs. Vectors

- Rewrite the previous example using vectors instead of arrays.