CSCI-1200 Computer Science II — Fall 2007
Lecture 6 — Pointers & Dynamic Memory

Review from Lecture 5

- Pointer variables, arrays, pointer arithmetic and dereferencing, character arrays, and calling conventions.

Today's Lecture — Pointers and Dynamic Memory

Optional Reading: Koening & Moo rest of Chapter 10

- Arrays and pointers, different types of memory, dynamic allocation of arrays, and examples.

6.1 Three Types of Memory

- Automatic memory: memory allocation inside a function when you create a variable. This allocates space for local variables in functions (on the *stack*) 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.

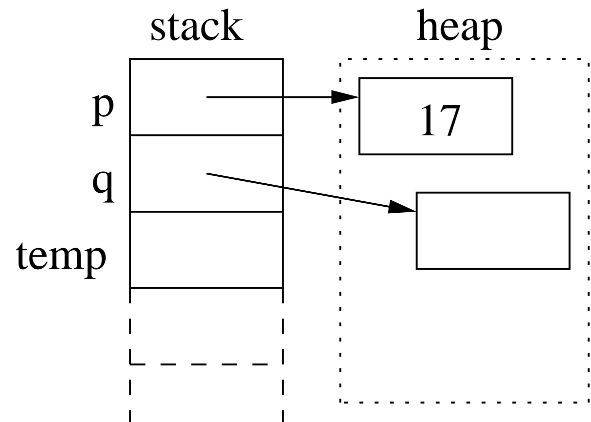
```
static int counter;
```

- Dynamic memory: explicitly allocated (on the *heap*) as needed. This is our focus for today.

6.2 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 and returns the address of that memory. Therefore, the statement `int * p = new int;` allocates memory from the heap and stores its address 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.

- This memory is allocated from and return to a special area of memory called the *heap*. By contrast, local variables and function parameters are placed on the *stack* as discussed last lecture.
- 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 15 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.

6.3 Exercise

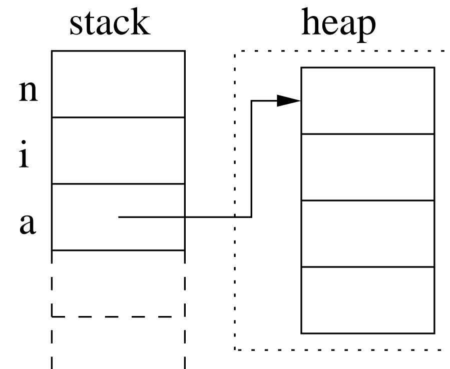
- What's the output of the following code? Be sure to draw a picture to help you figure it out.

```
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;
```

6.4 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 the standard library vector class. Here's an example:

```
int main() {
  cout << "Enter the size of the array: ";
  int n,i;
  cin >> n;
  double *a = new double[ n ];
  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. However, to demonstrate that you understand memory allocation & deallocation, you should *always* delete dynamically allocated memory in CS2, even if the program is terminating.
  - 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.

6.5 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!

6.6 Dynamic Allocation: Arrays of Class Objects

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

```
#include <iostream>
using namespace std;

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];
  cout << "size of foo object: " << sizeof(Foo) << endl;
  for (Foo* i = things; i < things+n; i++)
    cout << "Foo stored at: " << i << " has value " << i->value() << endl;
  delete [] things;
}
```

- The default constructor (the constructor that takes no arguments) must be defined in order to allocate an array of objects. In contrast, the default constructor is not necessary for vectors.

6.7 Dynamic Allocation Example: The Sieve of Eratosthenes

- Let's explore the problem of finding all primes less than a given integer, n, using the Sieve of Eratosthenes algorithm.
- The algorithm is a "casting out" algorithm: each discovered prime is used to cast out all of its multiples from a list of potential primes.
- We'll use dynamic memory allocation of an array to implement the algorithm. Later we'll see how do do this with STL vectors and lists.

```
// Using Sieve of Eratostenes determine all prime numbers less 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;
  }
  int prime_count = 0;

  // Create and initialize a dynamically allocated array to keep track
  // of whether the number 1->n are prime









  // Proces and output the numbers that are prime









  //  Release the dynamically allocated memory



  cout << "\nThere are " << prime_count << " primes <= " << n << endl;
  return 0;
}
```