# Computer Science II — CSci 1200
## Lecture 6
## Pointers, Dynamic Memory

**Review from Lecture 5**

- Pointer variables

- Arrays

- Pointer arithmetic and dereferencing

**Added Material, Not In the Printed Notes: Sorting an Array**

- Arrays may 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

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

  sorts the values in the array into increasing order.

**Today's Lecture — Pointers and Dynamic Memory**

- Arrays and pointers

- Different types of memory

- Dynamic allocation of arrays

- Examples and applications

**Three Types of Memory**

- Automatic memory: memory allocation inside a function when you create a variable. For example:

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

  This allocates space for local variables in functions and eliminates it when variables go out of scope. This memory is *allocated on the stack*.

- Static memory: variables allocated statically, as in

  ```
  static int x;
  ```

  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.

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

**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.

- This memory is allocated from and returned to a special area of memory called the *heap*. By contrast local variables and function parameters are allocated from a different memory area called the *stack*.

- In between the `new` and `delete` statements, 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 manipulations in the examples from Lecture 5 except that there is no explicitly named variable other than the pointer variable.

- Dynamic allocation of primitives like ints and doubles is not very significant, however. What is more important is dynamic allocation of arrays.

**Exercise**

What is the output of the following code? Be sure to draw a picture of the stack and the heap 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;
```

**Dynamic Allocation of Arrays**

- Declaring the size of an array at compile time does not offer much flexibility.

- We can dynamically allocate an array based on data. This gets us part-way toward the behavior of a vector.

- Here is 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.

**Exercises**

1. Write code to

   (a) dynamically allocate an array of `n` integers,
   (b) point to this array using the integer pointer variable `a`, and
   (c) then read $n$ values into the array from the stream `cin`.

2. Now, suppose we wanted to double the size of array `a` without losing the values. This requires some work.

   (a) Write code to allocate an array of size `2*n`, pointed to by integer pointer variable *temp* (which will become `a`).
   (b) Write code to copy the `n` values of `a` into the first `n` locations of array `temp`.

(c) Write code to delete array `a`.

(d) Write code to assign `temp` to `a`.

3. Why don't you need to delete `temp`?

The code for part 2 of the exercise is very similar to what happens inside the `resize` member function of vectors!

## Dynamic Allocation: Arrays of Class Objects

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

```
int n;
cin >> n;
class foo {
public:
   string a, b;
};
foo * farray = new foo[n];
```

- For class objects, the default constructor (the constructor that takes no arguments) must be defined.

  - Fortunately, vectors do not require default constructors — another advantage of vectors over arrays.

## Prime Numbers: Sieve of Eratosthenes

- We will explore the problem of finding all primes less than a given integer, $n$, and introduce the Sieve of Eratosthenes algorithm.

- The algorithm is a "casting out" algorithm: each new prime is used to cast out all of its multiples from a list of potential primes.

- We will discuss the idea and then implement the algorithm **during lecture** using the skeleton program provided on the next page of this handout.

- The program as we implement it will involve dynamic memory allocation of an array. We can also use vectors and (eventually) lists.

```cpp
#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;

int
main( int argc, char* argv[] )
{
  //  Check the number of command-line arguments
  if ( argc != 2 )
    {
      cerr << "Usage:\n  " << argv[0] << " n\n\n"
           << "where n is a positive integer\n";
      return 0;
    }

  //  Take n from the 1st argument by converting the string to an
  //  integer.  Make sure it is positive.
  int n = atoi( argv[1] );
  if ( n <= 0 )
    {
      cerr << "Usage:\n  " << argv[0] << " n\n\n"
           << "where n is a positive integer\n";
      return 0;
    }

  //  Start here...

  return 0;
}
```