Review from Tuesday’s Lecture

- Dynamic memory makes programs more flexible.
- Pointers are variables that store the memory locations of other variables.
- Pointers may be dereferenced, assigned, copied, and compared.
- When trying to understand how pointer assignments and pointer dereferencing work, be sure to draw pictures like the ones we drew in class.
- Use `new` and `delete` to dynamically allocate and deallocate single variables or class objects.
- Use `new []` and `delete []` to dynamically allocate and deallocate arrays.
- Pointers to dynamically allocated arrays point to the first location in the array.
- Pointer variables that point to arrays are treated the same as array names.
Dynamic Allocation of Objects

Recall our Date class:

```cpp
class Date {
    public:
        Date();
        Date(int aMonth, int aDay, int aYear);
        void inputDate();
        void setDate(int aDay, Month aMonth, int aYear);
        int getDay() const;
        Month getMonth() const;
        int getYear() const;
        bool isPayDay() const;
        bool isEqual(const Date& date2) const;
        bool sameDayAs(const Date& date2) const;
        void print() const;
        void increment();

    private:
        int day;
        Month month;
        int year;
};
```
• We can dynamically allocate Date objects:

```cpp
Date * d_ptr;
d_ptr = new Date( 30, 5, 1990 );
```

• Since *d_ptr dereferences a pointer, we can call member functions using the * and . operators. For example,

```cpp
(*d_ptr).setDate( 21, 9, 2001 );
(*d_ptr).day = 21; // only legal if "day" is made public
```

Note that the parentheses around *d_ptr are required because of operator precedence.

• This is a pain. Fortunately, we can use operator->

```cpp
d_ptr->setDate( 21, 9, 2001 );
d_ptr->day = 21; // only legal if "day" is made public
```

which are exactly equivalent to the statements above.

• Pointers to objects will be necessary for building data structures using non-contiguous memory, for example, linked lists and trees.

**Address Arithmetic**

Consider the definitions

```cpp
float * a = new float[ n ];
int i;
```

• The expression

```cpp
a[i]
```

is exactly equivalent to

```cpp
*(a+i)
```

• a+i means: take the value of the pointer a and add i locations for floats to it

• *(a+i) dereferences this location

• Thus, adding (or subtracting) an integer from a pointer has the effect of shifting the pointer value by that number of array entries.
Using Pointers to Step Through Arrays

We can use pointers to step through arrays.

- Consider the example of summing the values in a floating point array, named \texttt{f_array}

  ```
  float sum = 0;
  for ( int i = 0; i < n; ++ i )
    sum += f_array[ i ];
  ```

- This can be rewritten using pointers and pointer arithmetic as

  ```
  float sum=0;
  for ( float *p = f_array; p < f_array + n; ++ p )
    sum += *p;
  ```

- The comparison, pointer addition, increment and dereferencing operators all work correctly.
Pointers As Function Parameters

• When pointers are passed by value, the value of the pointer in
  the calling function (as opposed to the called function) can not
  be changed, but the value it points to can. For example:

```c
void
Swap( float * a, float * b)
{
    float temp = *a;
    *a = *b;
    *b = temp;
}
```

// ...

```c
int main()
{
    float a[ 10 ];
    for ( int i=0; i<10; ++i ) a[ i ] = i;
    Swap( a, a+5 );
    Swap( a+2, a+4 );
    ...
}
```

• If you want to change the value of a pointer, it must be passed by
  reference. One example, which we will revisit soon, is dynamically
  allocating an array:

```c
void Allocator( int * & array_ptr, int n )
{
    array_ptr = new int[ n ];
}
```

• We will look carefully at how this works.
**const Pointers and Pointers to const Objects**

There are a number of ways of using `const` with pointers.

- For a pointer to a const object:
  ```c++
  const int * p;
  ```
  Here, you can re-assign the pointer, but you can not change the `int` value pointed to.

- For a const pointer to an object:
  ```c++
  int * const p;
  ```
  Here, you can change the `int` value pointed to, but not the pointer.

- Finally, you can have a const pointer to a const value:
  ```c++
  const int * const p;
  ```

  These are somewhat subtle, and we will not be making substantial use of them.

**Arrays of Pointers and Dynamic 2D Arrays**

- We can even have arrays of pointers, or pointers to pointers.

- The most common use and the only one we will consider for now is two-dimensional arrays where both dimensions are specified at run-time:
  ```c++
  int nrows, ncols;
  cout << "Enter the number of rows and columns => ";
  cin >> nrows >> ncols;
  float ** matrix;
  matrix = new float* [ nrows ];
  for ( int r=0; r<nrows; ++r )
    matrix[ r ] = new float[ ncols ];
  ```

  After this, `matrix` can be treated as an ordinary 2-d array, and accessed as

  ```c++
  matrix[i][j]
  ```
Grading Statistics Example

We will revisit the earlier example of calculating grade statistics. Here are some improvements:

- The scores array size need not be specified in advance, and is determined automatically just by reading in the scores.
- The scores array is dynamically allocated inside a function, which necessitates passing a pointer by reference.
- Several examples of using pointer arithmetic to step through arrays are illustrated.
- The mode, which is the most frequently occurring grade, is calculated by dynamically allocating a counting array based on the scores actually read.

Summary of Major Ideas

- Class objects can be dynamically allocated. Public member functions and member variables are accessed through `operator->`
- `a[i]` is equivalent to `*(a+i)`
- Arrays can be accessed by using pointers, and pointer arithmetic can be used to step through the array. On the other hand, using array subscripts generally makes the code easier to read.
- Pass a pointer by reference if you need to have the pointer itself changed.
- These ideas are illustrated in the grading statistics example program.