Computer Science II — CSci 1200
Lecture 7
Classes and Dynamically-Allocated Member Variables: Vectors

Test 1

- Tuesday, February 13, 2:00-3:30.
- Location is TBD
- Coverage is Lectures 1-7, HW 1-3, Labs 1-4
- Closed-book and closed-notes
- Practice problems and solutions will be posted on-line soon.

Review from Lecture 6

- Arrays and pointers
- Different types of memory
- Dynamic allocation of arrays

Today's Lecture

- Designing our own container classes
- Vectors and dynamic member variables
- Templated classes
- Copy constructors, assignment operators and destructors
- Implementation

Reading: Ford&Topp, Sections 5.3-5.5

Designing Our Own Containers

- Mimic the interface of standard library containers
- Study the design of memory management code and iterators.
- Move toward eventually designing our own, more sophisticated classes.

Vector Public Interface

In creating our own version of the vector class, we will start by considering the public interface:

```
public:   //  Member functions and other operators
  T& operator[] ( size_type i );
  const T& operator[] ( size_type i ) const;
  void push_back(const T& t);
  void clear();
  bool empty() const;
  iterator erase( iterator p );
  size_type size() const;
  void resize( size_type n );

public:   //  Iterator operations
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end();
```

- This is an excerpt from the `Vec.h` header file, handed out with these notes.
- This appears to be quite simple and in fact it is.
- We will focus on each piece, but especially on the use of templates, on the underlying representation, and on memory management.

Templated Class Declarations and Member Function Definitions

In terms of just the layout of the code in `Vec.h`, the biggest difference is that this is a *templated class*.

- The keyword `template` and the template type name must appear before the class declaration, as in

  ```
  template <class T> class Vec
  ```

- Within the class declaration, `T` is used as a type and all member functions are said to be "templated over type `T`".
- In the actual text of the code files, templated member functions are often defined (written) inside the class declaration.
- The templated functions defined outside the template class declaration must be preceeded by the phrase

  ```
  template <class T>
  ```

  and then when `Vec` is referred to it must be

  ```
  Vec<T>
  ```

- Therefore for member function `create` (two versions), we have:

```
template <class T> void Vec<T>::create
```

Syntax and Compilation

- Templated classes and templated member functions are not created (compiled) until they are needed.
- Compilation of the class declaration is triggered by a line of the form

  ```
  Vec<int> v1;
  ```

  with `int` replacing `T`. This also compiles the default constructor for `Vec<int>` — because it is used here.
- Other member functions are not compiled unless they are used.
- When a different type is used with `Vec`, for example in the declaration

  ```
  Vec<double> z;
  ```

  the template class declaration is compiled again, this time with `double` replacing `T` instead of `int`. Again, however, only the member functions used are compiled.
- This is very different from ordinary classes, which are usually compiled separately and all functions are compiled regardless of whether or not they are needed.
- The code of templated classes AND the member functions must be available where they are used.
- As a result, member functions definitions are often included in the class declaration or attached below but still in the .h file. If member function definitions are placed in a separate .cpp file, this file must be `#include`d, just like the .h file, because the compiler needs to see it to generate code.

Member Variables

Now, looking inside the `Vec<T>` class at the member variables:

- `m_data` is a pointer to the start of the array (after it has been allocated)
  - Recall the (near) equivalence between pointers and arrays.
- `m_size` indicates the number of locations currently in use in the vector — exactly what the `size()` member function should return,
- `m_alloc` is the number of entries in the dynamically allocated block of memory — the actually-allocated array!

Drawing a picture, which we will do in class, will help clarify this, especially the distinction between `m_size` and `m_alloc`.

Typedefs, Iterators and Pointers

- Several types are created through typedef statements in the first public area of Vec.
- Once created the names are used as ordinary class type names. Therefore,

  ```
  Vec<int>::iterator
  ```

  is an iterator type defined by the Vec<int> class. For our purposes, we just think of an iterator as a pointer, in other words a T *.
- Also,

  ```
  Vec<int>::size_type
  ```

  is the size type, defined here as an unsigned int.
- Thus, internal to the declarations and member functions, T* and iterator **may be used interchangeably**.
- Importantly, the ++ and -- operators on pointers automatically apply to iterators.

operator[]

- Access to the individual locations of the Vec is provided through operator[].
  - Syntactically, use of this operator is translated by the compiler into a call to a function called operator[]. For example, if v is a Vec<int>, then

    ```
    v[i] = 5;
    ```

    translates into

    ```
    v.operator[](i) = 5;
    ```
- In most classes there are two versions of operator[]:
  - A non-const version returns a reference to m_data[i]. This is applied to non-const Vec<T> objects, allowing the contentns of the internal array to be modified.
    - This is a “backdoor” way to modify the internal content of the Vec<T> object, and it makes sense here because it matches the intuitive meaning of operator.
  - A const version is the one called for const str objects. This also returns m_data, but as a const reference, so it can not be modified. As a result, it is the one used for const Vec<T> objects.

Default Versions of Assignment Operator and Copy Constructor Are Dangerous!

- Before we write the copy constructor and the assignment operator, we consider what would happen if we did not write them.
- C++ compilers provide default versions of these if they are not provided.
- These defaults just copy the values of the member variables, one-by-one. For example, the default copy constructor behaves just like the following:

  ```
  template <class T>
  Vec<T> :: Vec( const Vec<T>& v )
   : m_data(v.m_data), m_size(v.m_size), m_alloc(v.m_alloc)
  {}
  ```

  In other words, it would construct each member variable from the corresponding member variable of `v`.
- This can be dangerous, as the following exercise illustrates.

Exercise

Suppose we used the default version of the assignment operator and copy constructor in our `Vec<T>` class. What would be the output of the following program? Assume all of the operations **except** the copy constructor behave as they would with a `std::vector<int>`.

```
Vec<int> v(4, 0.0);
v[0] = 13.1;  v[2] = 3.14;
Vec<int> u(v);
u[2] = 6.5;
u[3] = -4.8;
for ( unsigned int i=0; i<4; ++i )
  std::cout << u[i] << " " << v[i] << std::endl;
```

Explain why this happens by drawing a picture of the memory of both `u` and `v`.

Classes With Dynamically Allocated Memory

- Each object must do its own dynamic memory allocation
- We must be careful to keep the memory of each object instance separate from all others
- This requires that we write (very carefully) our own:
  - Copy constructor
  - Assignment operator
- Dynamic memory should be released when an object is finished with it. This is done through what is called a destructor.

Copy Constructor

- This constructor must allocate a new array for the objects being constructed, copy the contents of the array of the passed object, and set the pointer, size and allocation member variables appropriately.
- The actual copying is done in a private member function called `copy`.

Exercise

Write the private member function `copy`.

Aside (1): the "this" pointer

- All class objects have a special pointer defined called `this`.
- The `this` pointer simply points to the current class object; it may not be changed.
- The expression `*this` is a reference to the class object.
- The `this` pointer is used in several ways:
  - Make it clear when member variables of the current object are being used.
  - Check to see when an assignment is self-referencing.
  - Return a reference to the current object.

Aside (2): Assignment operators, generally speaking

- Assignment operators of the form

  ```
  v1 = v2;
  ```

  are translated by the compiler as

  ```
  v1.operator=(v2);
  ```

- Cascaded assignment operators of the form

  ```
     v1 = v2 = v3;
  ```

  are translated by the compiler as

  ```
  v1.operator=(v2.operator=(v3));
  ```

- Therefore, the value of the assignment operator (`v2 = v3`) must be suitable for input to a second assignment operator. This in turn means the result of an assignment operator ought to be a reference to an object.

Assignment operator for Vec

The implementation of an assignment operator usually takes on the same form for every class. This is illustrated by `Vec<T>`:

- Do no real work if there is a self-assignment.
- Otherwise, destroy the contents of the current object then copy the passed object, just as done by the copy constructor.
- Return a reference to the (copied) current object, using the `this` pointer.

Destructor

- Called implicitly when an automatic object goes out of scope or a dynamic object is deleted.
  - It can never be called explicitly!
- Must delete the dynamic memory owned by the class.
- The syntax of the function definition is a bit weird.
  - The `~` has been used as a logic negation in other contexts.

Increasing the Size of the Vec

`push_back( T const& x )`

- Adds to the end of the array, increasing `m_size` by one `T` location.
- But wait, what if `m_size == m_alloc` already, which means the allocated array is full?
- We need to increase the size of the array. Therefore we need to
  1. Allocate a new, larger array. The best strategy is generally to double the size of the current array.
  2. If the array size was originally 0, doubling does nothing. We must be sure that the resulting size is at least 1.
  3. Then we need to copy the contents of the current array.
  4. Finally, we must delete current array, make the `m_data` pointer point to the start of the new array, and adjust the `m_size` and `m_alloc` variables appropriately.

  We already did something very similar to this in Lecture 6.
- Only when we are sure there is enough room in the array (or we've made enough room) should we actually add the new object to the back of the array.

Final exercise:

Write the member function `push_back` and `erase`. These form the first checkpoint in lab.