

CSCI-1200 Data Structures — Fall 2024

Lecture 6 — Templated Classes & Vector Implementation

Review from Lectures 4 & 5 and Lab 4

- Arrays and pointers; Pointer arithmetic and dereferencing; Types of memory (“automatic”, static, dynamic)
- Dynamic allocation of arrays; Drawing pictures to explain stack vs. heap memory allocation; Memory debugging

Today’s Lecture

- Designing our own container classes:
 - Mimic the interface of standard library (STL) containers
 - Study the design of memory management.
 - Move toward eventually designing our own, more sophisticated classes.
- Vector implementation
- Templated classes (*including* compilation and instantiation of templated classes)
- Copy constructors, assignment operators, and destructors

6.1 Templated Class Declarations and Member Function Definitions

- In terms of the layout of the code in `vec.h` (pages 5 & 6 of the handout), the biggest difference is that this is a *templated class*. The keyword `template` and the template type name must appear before the class declaration:

```
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 preceded by the phrase: `template <class T>` and then when `Vec` is referred to it must be as `Vec<T>`. For example, for member function `create` (two versions), we write:

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

6.2 Typedef

- The keyword `typedef` can be used to create an *alias* / alternate name / shortcut for an existing type. Once created the names are used as ordinary type names.
- For example `Vec<int>::size_type` is the return type of the `size()` function, defined here as an `unsigned int`. We define the `Vec<T>::size_type` type in the `public` area of `Vec`.

6.3 Vector Public Interface

- In creating our own version of the STL vector class, we focus on the familiar functions in 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 resize(size_type n, const T& fill_in_value = T());  
    void clear();  
    bool empty() const;  
    size_type size() const;
```

- To implement our own generic (a.k.a. templated) vector class, we will implement all of these operations, manipulate the underlying representation, and discuss memory management.

6.4 Member Variables

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

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

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

6.5 `operator[]`

- Access to the individual locations of a `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` objects.
 - A const version is the one called for `const Vec` objects. This also returns `m_data[i]`, but as a const reference, so it can not be modified.

6.6 Increasing the Size of the Vec – `push_back`

- `push_back(T const& x)` adds to the end of the array, increasing `m_size` by one `T` location. But what if the allocated array is full (`m_size == m_alloc`)?
 1. Allocate a new, larger array. The best strategy is generally to double the size of the current array. Why?
 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.
- Only when we are sure there is enough room in the array should we actually add the new object to the back of the array, in the first unused space of the current array allocation.

6.7 Exercises

- Finish the definition of `Vec::push_back`.
- Write the `Vec::resize` function.

6.8 Default Versions of Assignment Operator and Copy Constructor Are Dangerous!

- What would happen if we don't implement our own versions of the copy constructor and the assignment operator?
- C++ compilers provide default versions of these if they are used but not explicitly implemented. These defaults just copy the values of the member variables, one-by-one. For example, the default copy constructor would look like this:

```
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 is dangerous and incorrect behavior for the `Vec` class. We don't want to just copy the `m_data` pointer. We really want to create a copy of the entire array! Let's look at this more closely...

6.9 Exercise: Compiler-Written Default Copy Constructor

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<double>`.

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

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

6.10 Implementation Requirements for Classes With Dynamically Allocated Memory

- For `Vec` (and other classes with dynamically-allocated memory) to work correctly, each object must do its own dynamic memory allocation and deallocation. We must be careful to keep the memory of each object instance separate from all others.
- All dynamically-allocated memory for an object should be released when the object is finished with it or when the object itself goes out of scope (through what’s called a *destructor*).
- To prevent the creation and use of default versions of these operations, we must write our own:
 - *Copy constructor*
 - *Assignment operator*
 - *Destructor*

6.11 Copy Constructor

- This constructor must dynamically allocate any memory needed for the object being constructed, copy the contents of the memory of the passed object to this new memory, and set the values of the various member variables appropriately.
- **Exercise:** In our `Vec` class, the actual copying is done in a private member function called `copy`. Write the private member function `copy`.

6.12 Destructor (the “constructor with a tilde/twiddle”)

- The destructor is called implicitly when an automatically-allocated object goes *out of scope* or a dynamically-allocated object is *deleted*. It can never be called explicitly!
- The destructor is responsible for deleting 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.

6.13 The “this” pointer

- All class objects have a special pointer defined called `this` which simply points to the current class object, and 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.

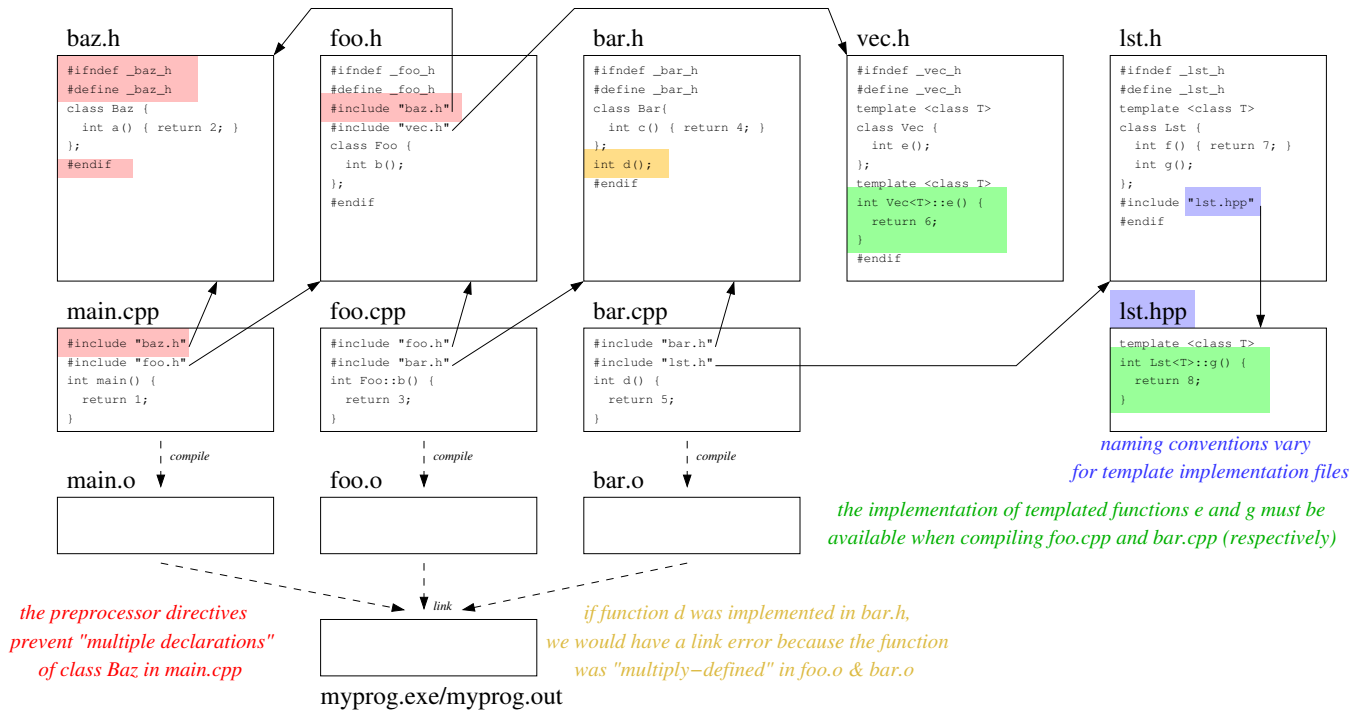
6.14 Assignment Operator

- 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.
- The implementation of an assignment operator usually takes on the same form for every class:
 - 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. In fact, it often makes sense to write a private helper function used by both the copy constructor and the assignment operator.
 - Return a reference to the (copied) current object, using the `this` pointer.

6.15 Syntax and Compilation

- Templated classes and templated member functions are not created/compiled/instantiated 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 – *make sure to use/test of all of the functions you write!*
 - 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 that are 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 templated class declaration and the code for all used member functions must be provided where they are used. As a result, member functions definitions for templated classes are usually included within the class declaration or implemented outside of the class declaration but still in the `.h` file. If member function implementations are placed in a separate file, this file must be `#include-d`, just like the `.h` file, because the compiler needs to see it in order to generate code. (Normally we don't `#include .cpp` implementation files!)
- Reminder: Including function implementations outside of the class declaration in the `.h` for non-templated classes can lead to compilation errors about functions being “multiply defined”.
- The diagram below shows the typical and suggested file organization for non-templated vs. templated classes. Common mistakes and the resulting compilation errors are noted.



6.16 Vec Declaration & Implementation (vec.h)

```
#ifndef Vec_h_
#define Vec_h_

// We ensure that that m_size is always <= m_alloc and when a push_back or resize
// call would violate this condition, the data is copied to a larger array.

template <class T> class Vec {
public:
    // TYPEDEFS
    typedef unsigned int size_type;

    // CONSTRUCTORS, ASSIGNMENT OPERATOR, & DESTRUCTOR
    Vec() { create(); }
    Vec(size_type n, const T& t = T()) { create(n, t); }
    Vec(const Vec& v) { copy(v); }
    Vec& operator=(const Vec& v);
    ~Vec() { destroy(); }

    // MEMBER FUNCTIONS AND OTHER OPERATORS
    T& operator[] (size_type i) { return m_data[i]; }
    const T& operator[] (size_type i) const { return m_data[i]; }
    void push_back(const T& t);
    void resize(size_type n, const T& fill_in_value = T());
    void clear() { destroy(); create(); }
    bool empty() const { return m_size == 0; }
    size_type size() const { return m_size; }

private:
    // PRIVATE MEMBER FUNCTIONS
    void create();
    void create(size_type n, const T& val);
    void copy(const Vec<T>& v);
    void destroy() { delete [] m_data; }

    // REPRESENTATION
    T* m_data; // Pointer to first location in the allocated array
    size_type m_size; // Number of elements stored in the vector
    size_type m_alloc; // Number of array locations allocated, m_size <= m_alloc
};

// Create an empty vector (null pointers everywhere).
template <class T> void Vec<T>::create() {
    m_data = NULL;
    m_size = m_alloc = 0; // No memory allocated yet
}

// Create a vector with size n, each location having the given value
template <class T> void Vec<T>::create(size_type n, const T& val) {
    m_data = new T[n];
    m_size = m_alloc = n;
    for (size_type i = 0; i < m_size; i++) {
        m_data[i] = val;
    }
}

// Assign one vector to another, avoiding duplicate copying.
template <class T> Vec<T>& Vec<T>::operator=(const Vec<T>& v) {
    if (this != &v) {
        destroy();
        copy(v);
    }
    return *this;
}
```

```

// Create the vector as a copy of the given vector.
template <class T> void Vec<T>::copy(const Vec<T>& v) {
    // EXERCISE: COMPLETE THIS FUNCTION

}

// Add an element to the end, resize if necessary.
template <class T> void Vec<T>::push_back(const T& val) {
    if (m_size == m_alloc) {
        // Allocate a larger array, and copy the old values
        // EXERCISE: COMPLETE THIS FUNCTION

    }
    // Add the value at the last location and increment the bound
    m_data[m_size] = val;
    ++ m_size;
}

// If n is less than or equal to the current size, just change the size. If n is
// greater than the current size, the new slots must be filled in with the given value.
// Re-allocation should occur only if necessary. push_back should not be used.
template <class T> void Vec<T>::resize(size_type n, const T& fill_in_value) {
    // EXERCISE: COMPLETE THIS FUNCTION

}

#endif

```