Advanced C++

CS240 Spring ‘19

Class Enhancements

Overloading Operators

- Most operators can be overloaded so that their behavior can be redefined for any class
    - You define behavior for +, -, *, etc.
- Operators are overloaded using operator functions
    - <type> **operator** sign (parameters) { /*... body ...*/ }
        - *int operator +(int x){ return this.num + x };*

Overloading Operators

- Where have we seen this already?
    - the iostream object overloads the << operator
        - *iostream operator <<(std::string){...}*
- You can overload the following operators:

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td><td>~</td></tr>
<tr><td>!</td><td>=</td><td><</td><td>></td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td></tr>
<tr><td>^=</td><td>&=</td><td>|=</td><td><<</td><td>>></td><td><<=</td><td>>>=</td><td>==</td><td>!=</td></tr>
<tr><td><=</td><td>>=</td><td>&&</td><td>||</td><td>++</td><td>--</td><td>,</td><td>->*</td><td>-></td></tr>
<tr><td>( )</td><td>[ ]</td><td>new</td><td>delete</td><td>new[]</td><td>delete[]</td><td></td><td></td><td></td></tr>
</table>

Friend Classes

- You can declare another unrelated class as a Friend
  - class F is friend of class C
- All class F member functions are friends of C and can access private class/instance variables
  - NOT reciprocated
- Friendship granted, not taken
  - Syntax: friend class F
  - Goes inside class definition of "authorizing" class

Problems with Friend Classes

- Violates encapsulation
  - Easily overused out of laziness rather than good design
  - DO NOT OVERUSE FRIEND CLASSES
    - *Good Examples*
      - Make Node variables private, then make Linked List a friend class
    - *Bad Examples*
      - Make class A a friend class of class B to avoid writing getters

Templates

Function Templates

- C++ functions work on specific types.
  - We need to write different routines to perform the same operation on different data types.
    - *We have overloading to 'hide' some of this*
- However, the operation we are performing is the same

```
int maximum(int a, int b, int c) {
        int max = a;
        if (b > max) max = b;
        if (c > max) max = c;
        return max;
}
float maximum(float a, float b, float c) {
        float max = a;
        if (b > max) max = b;
        if (c > max) max = c;
        return max;
}
```

Templates

- Type-independent patterns that can work with multiple data types.
    - Template programming
    - Makes our code more reusable
- Two kinds of Templates
    - Function Templates
    - Class Templates

Templatize the function

```
template <class T> T maximum(T a, T b, T c) {
	T max = a;
	if (b > max) max = b;
	if (c > max) max = c;
	return max;
}
```

Using a Template Function

- The compiler creates the function for you at compile time using the pattern you gave it
    - Call template functions just like you would a normal function, and the type gets automatically filled in
- Functions must be ‘viable’ for that type
    - This check is performed when you call the function
        - *A form of **duck typing***

Put Templates in Header Files

- Templates are patterns, not executable code
  - They are declarations for an entire function
  - The template function itself is incomplete because the compiler will need to know the actual type to generate code.
- C++ compiler will generate the real function based on the use of the function template.

Template Classes

- To make a class into a template, prefix the class definition with the syntax:
  - template <class T>
- Here **T** is just a type parameter. Like a function parameter, it is a placeholder.
  - When the class is instantiated, T is replaced by a real type.

Creating a Template Class

- Define template classes similar to functions
    - ```
      template <class T> class MyClass{
          T val;
      };
      ```
- To define an external method, use the following syntax:
    - ```
      template <class T> //You need this for every method
      T className< T >::memberName(T parameter)
      ```

Using the Template CLass

- To use your Template class:
    - MyClass<int> obj;
- Class and methods have to go into the same header file, no .cpp file
    - Remember, templates are declarations, so the compiler needs to generate the implementation
        - *You can think of it as the compiler writing the .cpp file for you*

Classwork

Templatizing

Why doesn't C++ use Templates?

- If templates allow us to make classes generic, why don't we have a standard set of generic ADT's?
- Why are we remaking the wheel?
  - Other than because I am making you

Standard Template Library

- Part of C++ standard
  - Each C++ compiler ships with STL
- Provides template container classes such as:
  - vector (like Java's ArrayList)
  - list (double-linked list)
  - stack
  - queue

Using the STL

- STL headers:
  - Headers for containers have the same name as container
    - *E.g. use #include <vector> for vector container*
- All STL functions and classes are defined in namespace std
  - Either prepend names with std::
    - *E.g. std::vector<int> v*
  - Or add *using namespace std;*

STL categories

- STL can be broken into several categories
  - Containers
    - *Data Structures that organize and provide access to data*
  - Iterators
    - *Allow for abstracted iteration through containers*
  - Algorithms
    - *Standard search, sort, shuffle, etc. algorithms*
  - Utilities
    - *i.e. miscellaneous*

STL Containers

- A container is a holder object that stores a collection of other objects (its elements).
  - implemented as class templates
- Manages storage for its elements
- Provides methods to access the elements
  - either directly or through iterators
    - *More on iterators later…*

3 Kinds of Containers

- Sequence
  - List - doubly linked list
  - Vectors - dynamic array
- Associative
  - Map - A Hashmap with key/value pairs
  - Set - unique sorted values
- Adaptors
  - Stack - LIFO data structure
  - Queue - FIFO data structure

Vectors

- The STL Vector is a dynamic array
  - Grows and shrinks as needed
  - Memory automatically managed
- Incremental vs Geometric expansion
  - Incremental expansion grows by 1 element as elements are inserted
  - Geometric Expansion grows by a factor of the current size

Vectors up Close

- Reminder: add/remove from vector is O(1)
    - No matter where it is
- CRUD Operations
    - Create: push_back()
    - Read : operator[], at()
    - Update: operator[]
    - Delete: clear(), pop_back()

CREATE: STL vector container

```
int main(){
    std::vector<int> v;
    for (size_t i = 0; i < 10; i++) {
        /* Adds element i to the end of a vector */
        v.push_back(i+1);
    }
}
```

READ: STL vector

```
std::vector<int> v;

for (size_t i = 0; i < v.size(); i++) {
    /* bounds checked */
    cout << v.at(i) << endl;
}
for (size_t i = 0; i < v.size()+1; i++) {
    /* no bounds checking */
    cout << v[i] << endl;
}
```

Update: STL vector

```
std::vector<int> v;

i = 2;
int & var = v.at(i);
var = 6; //update by reference

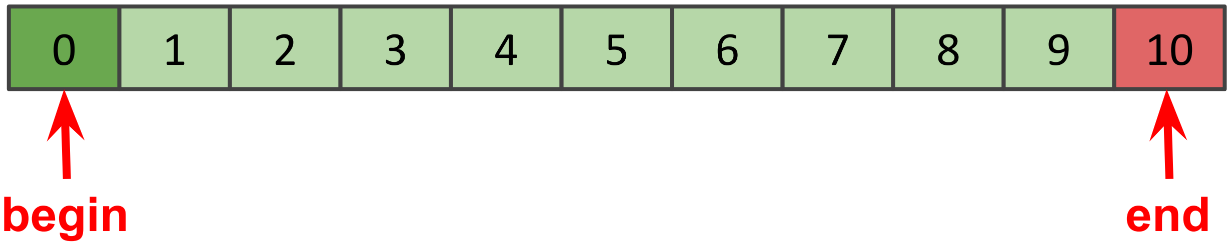
v[i] = 6; //update directly
```

DELETE: STL vector

```
std::vector<int> v;

//deletes the last element
v.pop_back();
//deletes all elements
v.clear();
```

Traversing an Array with Pointers




```
int array[10];
/* Pointer to the beginning of the array */
int* begin = &array[0];
/* Pointer to the element AFTER the last element*/
int* end = &array[10];
for (int* current = begin; current != end; ++current) {
   *current = 0
}
```

Iterators

- Every STL: class has its own internal iterator
- Iterators allow you to traverse the Data Structure without needing to know the implementation
    - How does the vector store data internally? As linked list, as array, by magic?
        - *Probably magic, but who cares, I'll just use the iterator*

Iterator Example

- The simplest form of an iterator is the pointer
  - You can use a pointer to iterate through the container
    - *How do you know when you are at the end?*
    - *What if you accidentally go past the end?*
- Internal Iterators are like safety nets
  - Won't go out of bounds
  - Don't need to know the length

3 Types of Iterators

- Forward Iterators
  - Can only go forward sequentially
- Bidirectional Iterators
  - Can move forward or backwards through the container
- Random Access
  - Can access any element from any other element

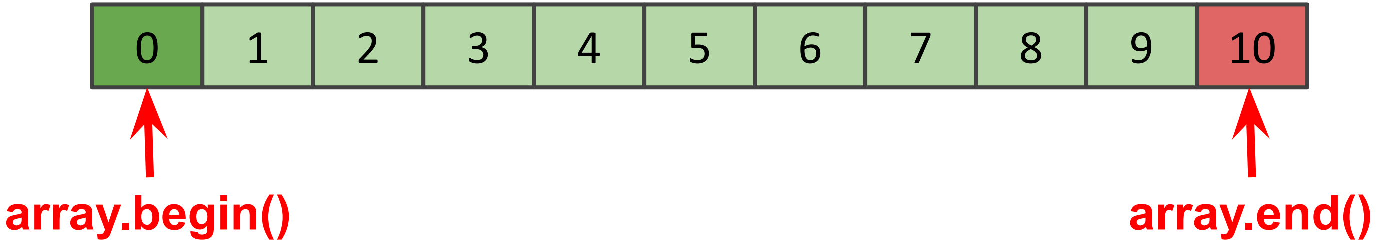
Iterator Operations

- Traversal
  - begin() / end()
    - *Iterator to beginning or end*
  - prev() / next()
    - *Get iterator to previous or next element*
- Access
  - *
    - *Access the value at that element*
  - ++/--/+val/-val
    - *Increment or decrement the iterator*

Certain types of Iterators only support some operations

i.e. a forward iterator cannot decrement or get previous

Traversing STL Vector with Iterators




```
std::vector<int> array;
for(std::vector<int>::iterator current = array.begin();
                                                current != array.end();
++current){
    *current = 0
}
```

What if we want to traverse backwards?

Classwork

Shopping List

C++ 11/14

Can we use templates to make working with pointers safer?

- STL class templates for pointers
  - commonly called a ‘smart pointer’
- Adds Java-like garbage collection
- Once created, you can use it exactly like a pointer
  - ...and there’s almost no overhead

Constructing a unique_ptr

- A unique_ptr object is always the unique owner of the associated raw pointer.
- Create a unique_ptr object through raw pointer
  - `std::unique_ptr<Task> taskPtr(new Task(23));`
- We cannot use assignment (why?)
  - `std::unique_ptr<Task> taskPtr2 = new Task(); //Error`

copy vs move

- unique_ptr<> is not copyable
  - Hence we can not create copy of a unique_ptr object either through copy constructor or assignment operator.
- We cannot copy a unique_ptr object, but we can move them.
  - unique_ptr can transfer ownership to another unique_ptr.

std::move

- std::move() will transfer the associated raw pointer to a new smart pointer
    - std::unique_ptr<Task> int_ptr2 = std::move(int_ptr);
        - *The parameter to move will be empty after transferring the ownership of its raw pointer*
- Additional smart pointer operations
    - sptr.reset() //deletes the object associated with the pointer
    - sptr.release() //returns the pointer and relinquishes ownership

std::unique_ptr

- The unique_ptr template class captures sole ownership of a pointer
- Guarantees single ownership of a pointer.
    - You cannot have two objects pointing to the same reference
    - What if you need a shared pointer?

std::shared_ptr

- STL class template for a shared pointer
- Allows itself to be copied
  - Uses reference counting (like Java) to know when it can be deleted.
- Use only when you need a shared reference

Auto keyword

- In C++11 and greater the compiler can infer the type of a variable at the point of declaration,
  - instead of putting in the variable type, you can just write auto:
    - *int x = 4; can now be replaced with: auto x = 4;*
    - *Use -std=c++14 flag with g++*
- ‘auto’ is really for working with templates and iterators:
  - vector<int> vec;
    auto itr = vec.begin(); // instead of vector<int>::iterator itr

Range Based For loop

- Most languages have a foreach loop that automatically advances through each element of a container
- C++11 added a foreach to C++
  - ```
    vector<int> vec;
    for (int i : vec ){
        cout << i;
    }
    ```
    ```

Automatic Iteration

- Combine with auto to make iteration a breeze
  - ```
    vector<int> vec;
    for ( auto data : vec ){
            cout  << data << endl;
    }
    ```
    ```

Modifying the Contents of the Container

- To modify the values in the container or to avoid copying large objects, you can make the loop variable a reference:
    - ```
      for (int& i : vec ){
              i++; // increments the value in the vector
      }
      ```
      ```

Default Behavior and Iteration

- Without the reference, the values in the container would not be changed because you are modifying a copy
    - This is the default, safe behavior. Why?
        - *Because, in general, you should not modify the contents of a container while iterating through a container*

Problem

- What happens when I overload a method to take a pointer or integer, and call it with NULL
  - I get an error (if I'm lucky) because it is an ambiguous call
- Pointer are treated as integers, and NULL is another name for 0
  - We need an actual type that equates to a pointer that can be NULL

nullptr

- C++ 11 adds nullptr_t, a distinct type that degrades to a pointer and has one value:
  - null pointer literal, called ***nullptr***.
- nullptr is not itself a pointer type
  - Instead, it degrades to a pointer type with the value NULL.
- You should always use nullptr instead of NULL to indicate an invalid pointer