Project #03: myset — a replacement for std::set

Complete By: Part 1 by Sunday October 8th @ 11:59pm,
Part 2 by Wednesday October 11th @ 11:59pm

Assignment: implement myset class

Policy: Individual work only, late work *is* accepted (part 2)

Submission: Part 1: via Blackboard (see "assignments", "P03-01")
Part 2: via Blackboard (see "assignments", "P03-02")

Assignment

The assignment is two-fold: (1) develop a **myset<T>** class that operates much like the built-in std::set<T> container, and (2) develop a set of **unit tests** to test your myset<T> implementation. We will be using Visual Studio 2017 for this assignment.

Your myset<T> class is a generic container that supports the following operations:

- default constructor
- copy constructor
- destructor

- size()
- empty()
- clear()

- insert(e)
- operator+=
- find(e)
- [e]

- operator=
- begin()
- end()

Another section in this handout will describe these functions in more detail. The implementation of containers is a mixture of old-school C and modern C++. You'll follow this practice and implement your myset<T> class using either an array, linked-list, or binary search tree; hashing is also an option, but I don't recommend it because it's hard to create a viable hash function when you don't know what type T is. Your implementation may not use any of the built-in algorithms or containers.

A set (or myset in this case) is a collection of unique elements --- as in a mathematical set, there are no duplicates. Here's an example of using a set in a main program:

```
#include <iostream>
#include <string>
#include "myset.h"

using namespace std;

int main()
{
   myset<string>  s;
   string         input;

   s.insert("pear");  // elements are inserted in order:
   s.insert("banana");
   s += "apple";

   cin >> input;  // get a string from the user:

   if (s[input])  // does s contain input string?
      cout << "set contains '" << input << "'" << endl;
   else
      cout << "set does NOT contain '" << input << "'" << endl;

   for (auto e : s)  // foreach works and iterates in order:
   {
      cout << e << endl;  // apple, banana, pear:
   }

   return 0;
}
```

Of course, we don't want to develop our code this way --- we want to use unit testing so we'll have a testing approach that is automated, repeatable, and easy to determine correctness. Here's the same idea in the form of a Visual Studio unit test:

```
TEST_METHOD(jhummel_test_1234)
{
  myset<string>  s;
  string         input;

  Assert::IsTrue(s.insert("pear"));    // insert in order:
  Assert::IsTrue(s.insert("banana"));
  s += "apple";

  input = "pear";
  Assert::IsTrue(s[input]);   // found:
  input = "apple";
  Assert::IsTrue(s[input]);   // found:
  input = "banana";
  Assert::IsTrue(s[input]);   // found:
```

```
    input = "peach";
    Assert::IsTrue(!s[input]);  // not found:

    // expected results, in this order:
    vector<string> V = { "apple", "banana", "pear" };

    int i = 0;

    for (auto e : s)
    {
      Assert::IsTrue(e == V[i]);
      i++;
    }
  }
```

More on unit testing in another section of this document.

myset<T>

What follows is the complete class definition of **myset<T>**. Most of the code has been omitted; where you see "TODO" is where you must supply the missing implementation. As noted earlier, your set should be implemented using an array, linked-list, or binary search tree. However, you cannot use any of the built-in algorithms or containers, and you must meet the requirements specified in the comments. For example, insert(e) must have a time <= O(N) --- this means you cannot simply insert e at the end of the array and then resort. Similarly, and find(e) must have a time <= O(lgN).

```
/*myset.h*/

//
// myset data structure, i.e. a data structure modelled
// after std::set in modern C++.
//
// YOUR NAME
// U. of Illinois, Chicago
// CS 341, Fall 2017
// Project #03
//

#pragma once

using namespace std;

template<typename T>
class myset
{
private:
  //
  // Allows iteration through the set in element order.
  //
  class iterator
  {
```

```
private:

public:
  //
  // Constructor
  //
  iterator()
  {
    //
    // TODO:
    //
  }

  //
  // Destructor
  //
  ~iterator()
  {
    //
    // TODO:
    //
  }

  //
  // ++iter
  //
  // Advances to the next element in the set; elements are
  // visited in element order.  Operation is undefined if
  // iterator == end().
  //
  iterator& operator++()
  {
    //
    // TODO: advance "this" iterator
    //

    //
    // return "this" updated iterator:
    //
    return *this;
  }

  //
  // *iter
  //
  // Returns the element denoted by this iterator; the
  // operation is undefined if iterator == end().
  //
  T& operator*()
  {
    //
    // TODO:
    //
    T  defaultValue;
```

```
      //
      // Note: this code is wrong, you cannot return a reference
      // to a local variable -- the local is destroyed when
      // function returns. Must return something with a longer
      // lifetime, i.e. data in the heap.
      //

      return defaultValue;
    }

    //
    // lhs != rhs
    //
    // Returns true if "this" iterator != rhs iterator.
    //
    bool operator!=(const iterator& rhs)
    {
      //
      // TODO:
      //
      return false;
    }

    //
    // lhs == rhs
    //
    // Returns true if "this" iterator == rhs iterator.
    //
    bool operator==(const iterator& rhs)
    {
      //
      // TODO:
      //
      return true;
    }
  };

  //
  // Data members for myset:
  //

  //
  // TODO:
  //


public:
  //
  // Default constructor: constructs a new, empty set
  //
  myset()
  {
    //
```

```
  // TODO:
  //
}

//
// Copy constructor:
//
myset(const myset& other)
{
  //
  // TODO:
  //
}

//
// Destructor: destroy all elements in set.
//
~myset()
{
  //
  // TODO:
  //
}

//
// size()
//
// Returns # of elements in the set.
//
int size() const
{
  //
  // TODO:
  //
  return -1;
}

//
// empty()
//
// Returns true if set is empty, false if not.
//
bool empty() const
{
  //
  // TODO:
  //
  return false;
}

//
// clear()
//
// Empties the set, destroying all the elements.  Afterwards,
```

```
// the size of the set is 0.
//
void clear()
{
  //
  // TODO:
  //
}

//
// insert(e)
//
// Inserts the element e into the set.  Returns true if e
// was inserted, false if e was already in the set (and thus
// not inserted again).  Elements are inserted in element
// order as defined by the < operator; this enables in order
// iteration.
//
// Requirements:
//   1. Insert time must be <= O(N).
//   2. Set grows as needed to accommodate new elements.
//   3. Assumes only < operator; two elements x and y are
//      equal if (!(x<y)) && (!(y<x)).
//
bool insert(const T& e)
{
  //
  // TODO:
  //
  return false;
}

//
// += e
//
// Inserts e into the set; see insert(e).
//
myset& operator+=(const T& e)
{
  //
  // insert e into "this" set:
  //
  this->insert(e);

  //
  // return "this" updated set:
  //
  return *this;
}

//
// find(e)
//
// Searches the set for the element e, returning an iterator
```

```
// to e if found.  If e is not found, end() is returned, i.e.
// an iterator denoting one past the last element.
//
// Requirements:
//   1. Find time must be <= O(lgN).
//   2. Assumes only < operator; two elements x and y are
//      equal if (!(x<y)) && (!(y<x)).
//
iterator find(const T& e)
{
  //
  // TODO:
  //
  return iterator();
}

//
// [e]
//
// Returns true if set contains e, false if not.
//
// Requirements:
//   1. operation time must be <= O(lgN).
//   2. Assumes only < operator; two elements x and y are
//      equal if (!(x<y)) && (!(y<x)).
//
bool operator[](const T& e)
{
  //
  // TODO:
  //
  return false;
}

//
// lhs = rhs;
//
// Makes a deep copy of rhs (right-hand-side) and assigns into
// lhs (left-hand-side).  Any existing elements in the lhs
// are destroyed.
//
// Notes:
//   1. this is essentially a clear() followed by a copy().
//   2. the lhs operand is hidden --- it's "this" object.
//
myset& operator=(const myset& rhs)
{
  //
  // TODO:
  //

  //
  // return "this" updated set:
  //
```

```
    return *this;
  }

  //
  // begin()
  //
  // Returns an iterator denoting the first element in the
  // set; iteration is performed in element order (as defined
  // by the < operator).
  //
  iterator begin()
  {
    //
    // TODO:
    //
    return iterator();
  }

  //
  // end()
  //
  // Returns an iterator denoting one past the last element
  // in the iteration.
  //
  iterator end()
  {
    //
    // TODO:
    //
    return iterator();
  }
};
```

Suggestion: while a binary search tree (with smart pointers?) is perhaps the most efficient implementation, I suggest you keep things simple and use an array --- much like vector<T>. If you use an array, you must start with an initial capacity of 4, and then dynamically grow the underlying array as the set becomes full. Arrays make the iterator implementation much easier.

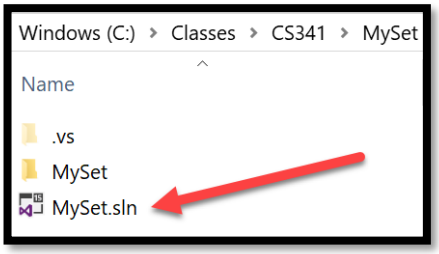
Downloading the Code

The initial code for myset<T>, and some unit tests, are provided in the form of a Visual Studio 2017 solution. Browse to the course web page (http://www.joehummel.net/cs341.html), open "Projects", open "project03-files", and download the .zip file "MySet.zip". Here's the link:

https://www.dropbox.com/s/mdnopeqjglncuz6/MySet.zip?dl=0

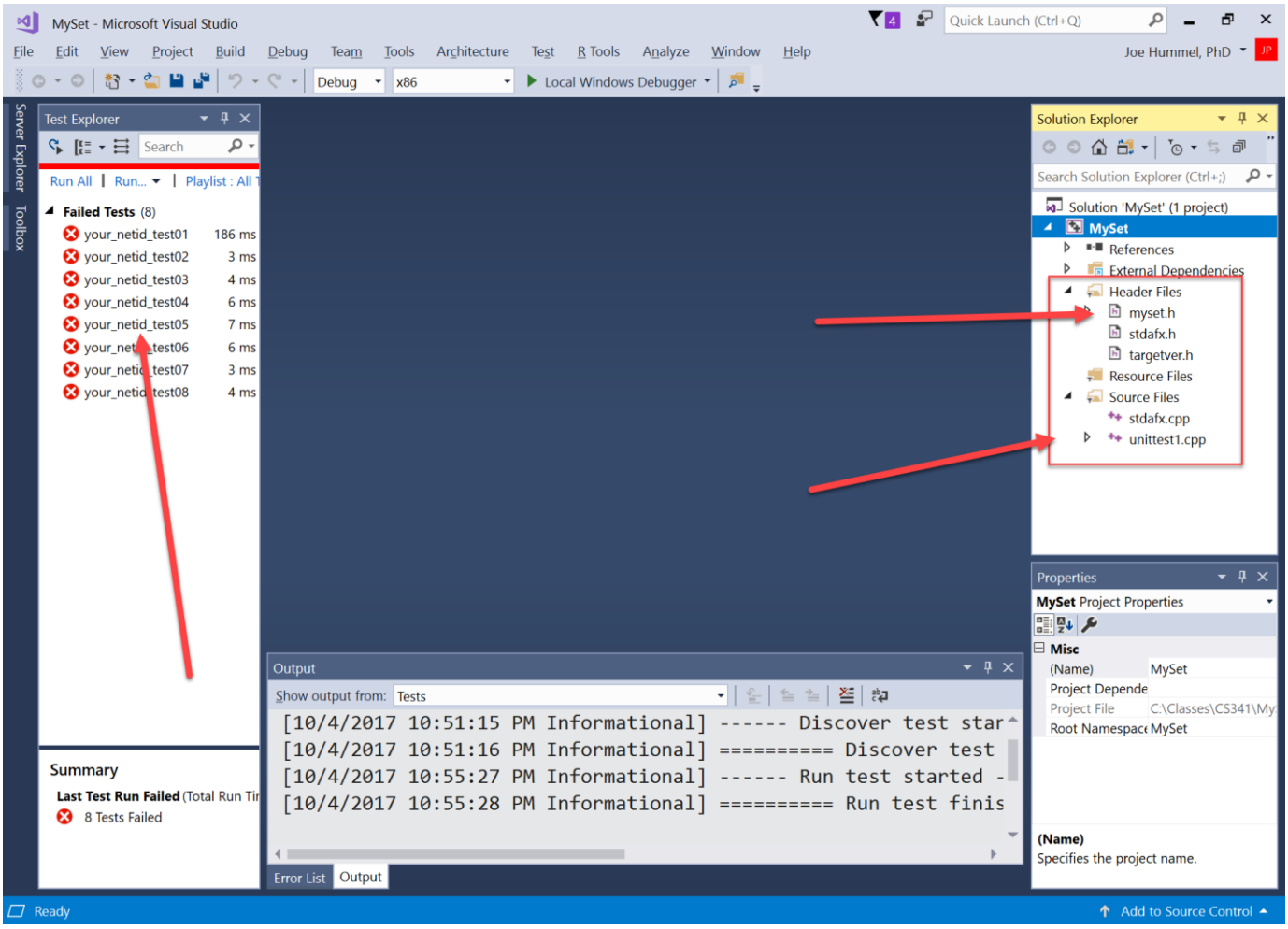
Double-click to open the .zip, extract the MySet folder, close the .zip --- and delete the .zip file. Focus on the extracted folder. First, move the MySet folder to your desktop, or a folder on your local hard drive; Visual

Studio doesn't work well when the project folder is on a cloud drive like dropbox or onedrive.



Open the MySet folder, and you should see another MySet folder (which contains the source files) and a Visual Studio Solution file --- if you have file extensions visible, this file will end in .sln. Now double-click on the solution file (.sln), and the program will be opened in Visual Studio. Go ahead and build the program using the Build menu; it should build without error. Now try to run the program using the traditional Ctrl+F5 --- it will fail. Why? Because there is no main() program.

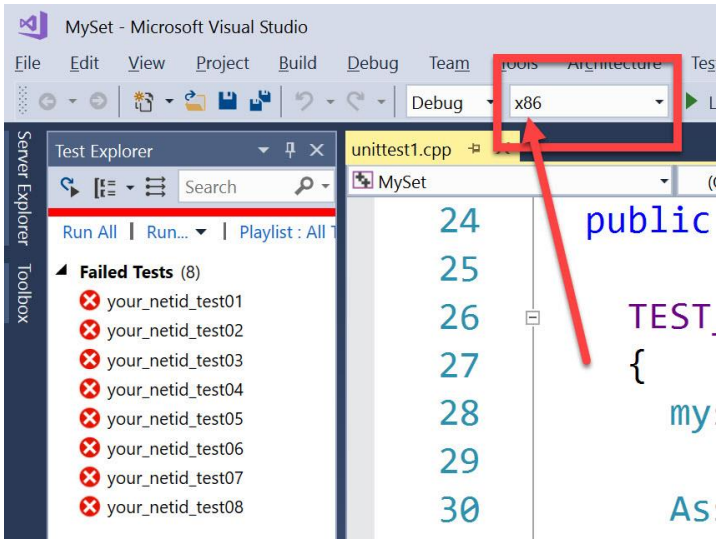
Instead, since we are using Visual Studio's unit testing framework, whenever we want to run we have to "run the tests". Drop the **Test** menu, select **Windows**, and then select **Test Explorer**. You should see something like this:



The program is provided with 8 unit tests. To run the tests, click the "Run All" link at the very top of the **Test Explorer** window (top-left). Unfortunately, you will fail all 8 tests because myset<T> has not yet been implemented.

NOTE NOTE NOTE: if you have a 64-bit computer, Visual Studio may default to "x64" mode --- i.e. 64-bit

mode. For some reason I have yet to discover, the tests only run in x86 mode --- i.e. 32-bit mode. So if you are expecting to pass some test cases but they all continue to fail, make sure Visual Studio is in x86 mode as shown below:



Unit testing

Testing data structures is a great example of the power of unit testing. Data structures have a well-defined API, and are critical components of any software system. This makes them ideal candidates for unit testing. Like most testing, unit testing is an art. The general idea is to start small, and your initial unit tests so focus on testing one feature at a time. For example, here's the first unit test from the provided source file "unittest1.cpp":

```
TEST_METHOD(your_netid_test01)
{
  myset<int>  s;

  Assert::IsTrue(s.size() == 0);
  Assert::IsTrue(s.empty());
}
```

The test creates a new set, and then checks to make sure the initial **size is 0** and the initial **state is empty**. That's it. The second test inserts a single element, and then checks the state of the set:

```
TEST_METHOD(your_netid_test02)
{
  myset<int>  s;

  Assert::IsTrue(s.size() == 0);
  Assert::IsTrue(s.empty());

  Assert::IsTrue(s.insert(12));

  Assert::IsTrue(s.size() == 1);
  Assert::IsTrue(!s.empty());
}
```

As you can imagine, the 3rd test might insert multiple elements and see what happens. Or what I did was switch over to a set of strings, and then make sure that both insert and lookup [ ] work:

```
TEST_METHOD(your_netid_test03)
{
  myset<string>  s;

  Assert::IsTrue(s.size() == 0);
  Assert::IsTrue(s.empty());

  Assert::IsTrue(s.insert("apple"));

  Assert::IsTrue(s.size() == 1);
  Assert::IsTrue(!s.empty());

  Assert::IsTrue(s["apple"]);
}
```

And so on. The idea is to write hundreds and hundreds of tests, trying to break your implementation.

Getting Started

There are two ways to get started: write more tests, and then implement the necessary functionality in "myset.h" to pass the tests. Or implement some functionality in "myset.h", and then write one or more tests to test it. Start with the basic functionality of myset<T>:

- constructor
- size()
- empty()
- insert(e)
- [e]

Initially, don't insert too many elements, just get myset to work for 3-4 elements. Then get it to work for 100's of elements, dynamically growing the set as appropriate.

Note that when you need to run a test, you don't have to run all the tests. You can right-click on a given test in the Test Explorer, and "run selected test". After you have the basics working, I would focus on iterators next:

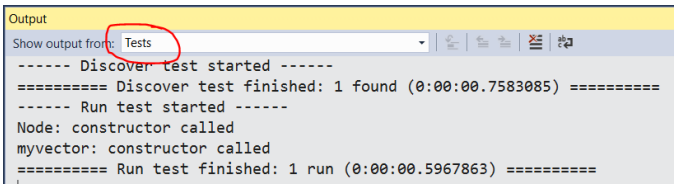
- Iterator class
- begin()
- end()
- find(e)

Once those are working, then focus on the remaining functions in myset<T>.

Debugging

The Test menu allows you to select and run an individual test — this is helpful when debugging a particular function or scenario. For help debugging, you can set a breakpoint in your code (e.g. within "myset.h"), then debug by right-clicking on a test in the "Test Explorer" window pane and selecting "Debug Selected Tests". The debugger will stop when your breakpoint is reached; you can then hover over variables to look at their contents, single step via Debug menu, etc. You set a breakpoint by clicking in the left margin of the editor window; a red circle appears. You can set as many breakpoints as you want. I often start by setting a breakpoint in the unit test code, at the first Assert::IsTrue statement. I press F10 to step from stmt to stmt until the code breaks. Then I start over, and press F11 to step-into the code that is failing.

If you prefer "print" debugging, note that with unit testing there's no console window, so you cannot **cout** debugging messages. If you want to output debug messages from your code, use




```
Logger::WriteMessage("myset: constructor called");  // debug output:
Logger::WriteMessage( to_string(NumElems).c_str() );
```

The output shows up in the Output menu when you run tests — select "Tests" from the drop-down. You may need to #include <string> to gain access to the **to_string()** function in the std library.

Part 1: The power of unit testing

Given a standard testing framework, programmers can easily share tests. We will do that here for the benefit of everyone. As part of this assignment, you are required to upload 5 new unit tests to blackboard — do not include the 8 tests that are already provided. Come up with 5 new tests on your own. **Name the unit tests with your netid** so we may combine all the unit tests together for sharing with the class:

```
TEST_METHOD(your_netid_test01)
{
   .
   .
   .
}
```

Your tests must compile, and perform a non-trivial test (which we'll define as containing at least 10 different Assert::IsTrue statements).

On Blackboard, look under "Assignments", and submit your "unittest1.cpp" file using the link "P03-01 Unit Tests". You must submit the tests on BB before the due date for part 1 (see page 1). These tests will be collected and shared with the class the following day. However, note that we are expecting you to test your own code --- we may provide some of our instructor unit tests, but not all. This implies you'll need tests for your copy constructor, operator=, and iterators. And can you think of a way to test your destructor? [*Hint: do not call the ~myset destructor directly, but instead store into the set some objects of your own design where you modify their destructor to make sure it's being called when the set is destroyed?*]

Part 2: myset<T>

Part 2 is to complete the implementation of myset<T>. Additional testing will be made available 24-48 hours before the due date on http://repl.it. But the bulk of the testing is your responsibility.

Electronic Submission — part #2

For part 2 you need only submit your final "myset.h" file. Before you submit, be sure the top of the file contains a brief description, along with your name, etc. For example:

```
//
// myset data structure, i.e. a data structure modelled
// after std::set in modern C++.
//
// YOUR NAME
// U. of Illinois, Chicago
// CS 341, Fall 2017
// Project #03
//
```

Your code should be readable with proper indentation and naming conventions; commenting is expected where appropriate.

When you're ready, submit your "myset.h" file on Blackboard: look under "Assignments", and submit the "myset.h" file using the link "P03-02 MySet".

Policy

Late work *is* accepted for part 2 of the assignment. You may submit as late as 24 hours after the deadline for a penalty of 25%. After 24 hours, no submissions will be accepted.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. via Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is described here:

http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at http://www.uic.edu/depts/dos/studentconductprocess.shtml .