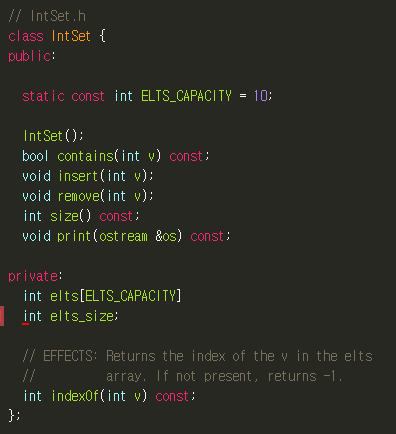
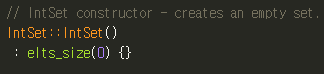
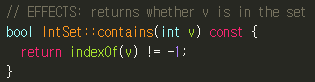
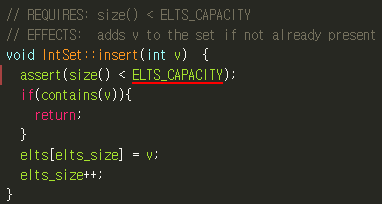
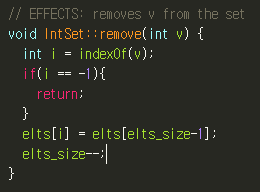
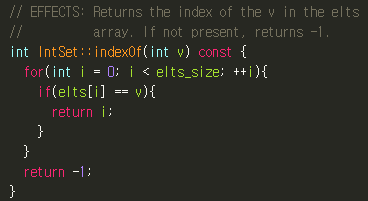
Container ADTs

Container: ADT whose purpose is to hold other objects

1. IntSet: unordered collection of unique elements

* Why Fixed capacity: Implementation needs to know how much space to allocate.
* Static: shared among all instances of the class (like global variable but lives inside a class’s scope)

Inset.h  
  
  
  
  
  
  
  
  
  
  
IntSet.cpp

1. Constructor
2. IntSet::contains
3. IntSet::insert
4. IntSet::remove (erasing the element with the last element)
5. IntSet::indexOf

Measuring Performance

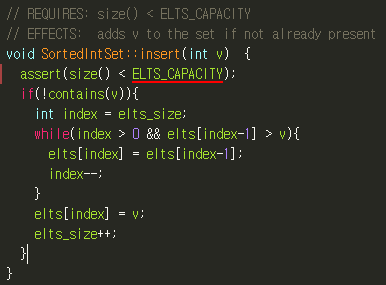
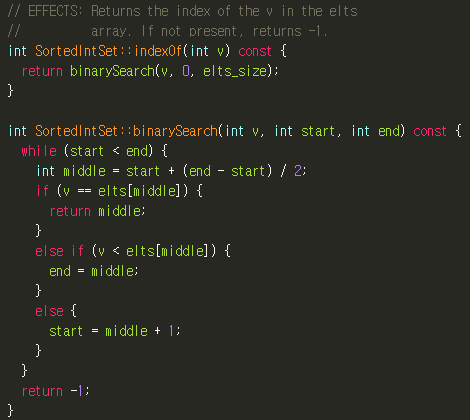
Big-O or Asymptotic : how the # of steps scales with the size of the input

1. O(1) – Constant time (flat line)  
   ex) What is the first element of an N element array?
2. O(logN) – Logarithmic time  
   ex) Finding a name in a phone book
3. O(n) – Linear (straight line)  
   ex) What is the largest element in an N element array?
4. O() – Quadratic (slightly curved line)  
   ex) What is the energy of a matrix of width N? (double nested for loop)

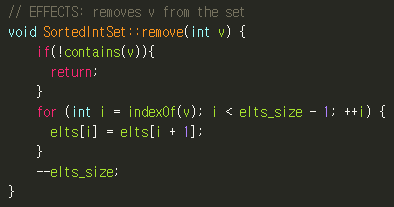
Set Efficiency

ctor – O(1) , size – O(1) , insert – O(n) , remove – O(n) , contains – O(n)

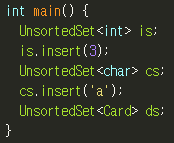
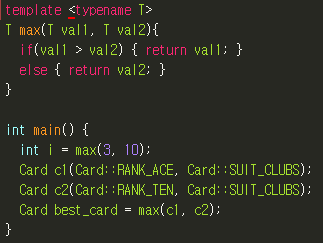
1. SortedIntSet

  
1) SortedIntSet::indexOf – O(logn) , Advantage of Sorting  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
2) SortedIntSet::insert – O(n) + O(logn)   
1. Check if element already exists (O(logn)) 2. not, move every element larger than v one to the right (O(n))

3) SortedIntSet::remove – O(n) + O(logn)



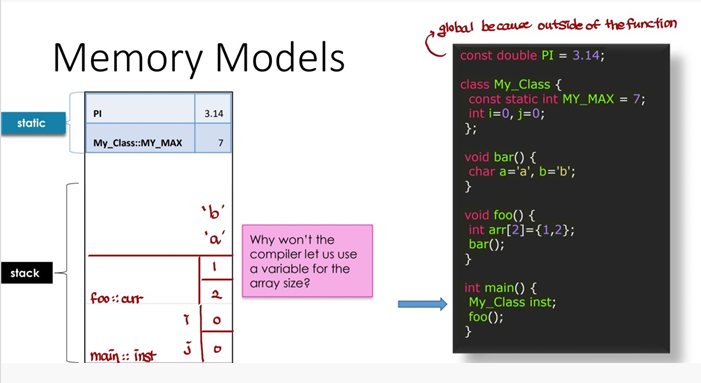
Templates  
idea: What about Charset? Do we just copy and paste? No. Write all the code in generic way.

Template is a model producing code ->   
  
  
  
Function template  
: can be instantiated to make versions to work with different types of inputs.

Compiler is able to deduce which version of max we want in each case from the argument types.

Due to compiler details, we can’t split our files into .h and .cpp files as usual. Instead, just include implementation of templated code in header file.

Dynamic memory

* Automatic variables – “normal” variables that are created and destroyed on the stack when we enter / leave their scompe
* Static variables – are created at the start of the program and aren’t destroyed until the program terminates  
  (ex) global variables or static class members)

Limitation of Automatic Variables

1. Compiler needs to know how large arrays are
2. Lifetime of objects are limited to scope they are declared

Dynamic Memory  
: it gives us the power to get over both these limitations.

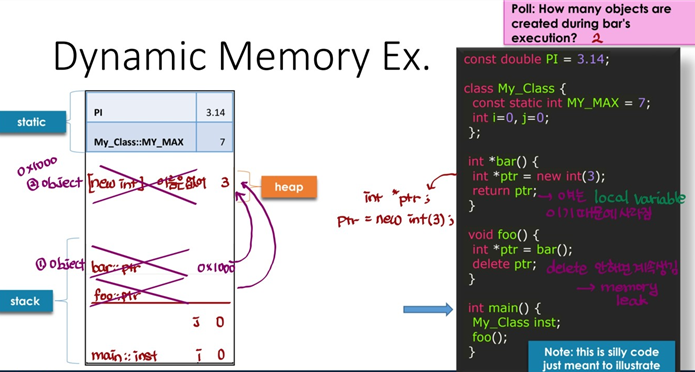
* Objects are created by a new expression
* Objects are destroyed by a delete expression

The new Operator

* Space for an int is created on heap
* The new expression evluates to the address of the object (So, need to use ptr)

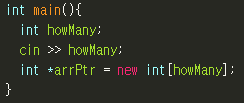
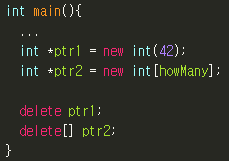
The delete Operator

* Follow the pointer to a dynamic object and destroy whatever object was there

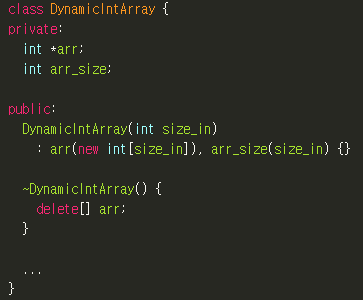
Lifetime of Objects

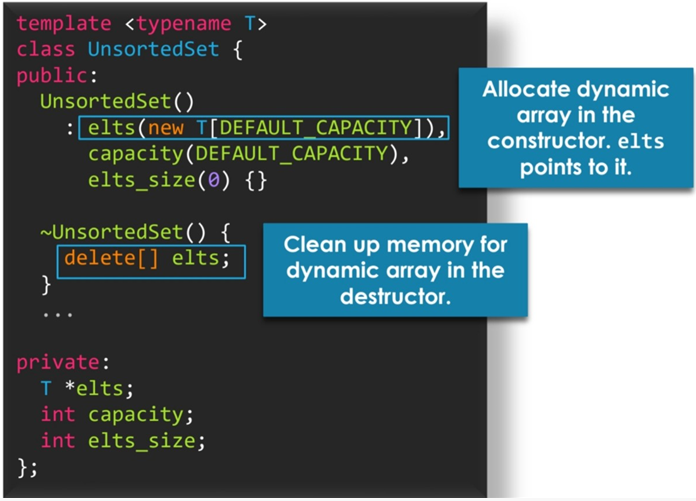
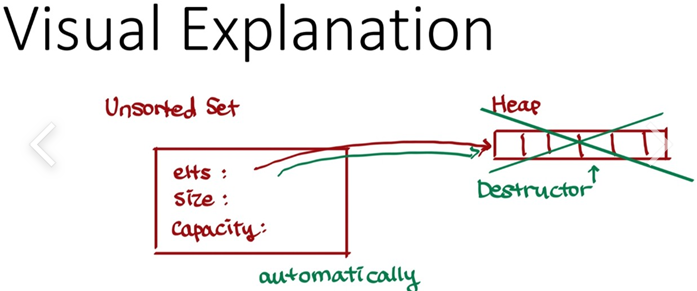
* Static (global, static class variables): Lifetime is entire program
* Automatic (local variables): Lifetime is the scope of the variable
* Dynamic: Lifetime spans from creation via new to dstruction via delete, managed by programer

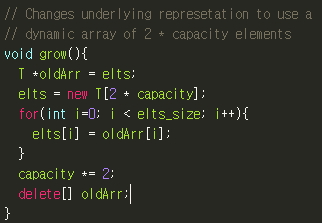
Dynamic Arrays

* The size of a statically allocated array must be known at compile time
* The size of a dynamically allocated array may be determined at runtime
* The result of a new expression that creates a dynamic array is a pointer to the first element
* Use a special delete[] syntax for arrays.

Managing Dynamic Memory

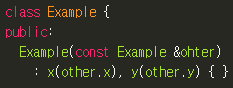
Destructors  
: are called whenever a class object’s lifetime ends (depends on storage duration)  
defined in class as “~[Class name]”

Upgrading UnsortedSet  
: Let’s remove the fixed capacity restriction. We’ll use dynamic memory in the implementation

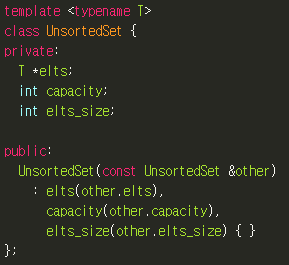
To insert, we need grow() function

Two Kinds of Copies

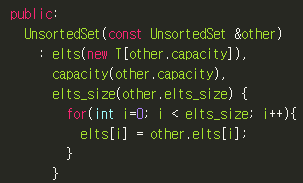
* Copy constructor: Used when initializing a new object
* Assignment operator: Used for assigning a new value to a preexisting object

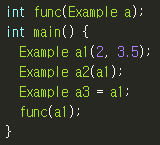
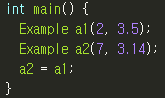
The Copy Constructor

Always pass by reference.  
If passing by value, this leads to infinite recursion

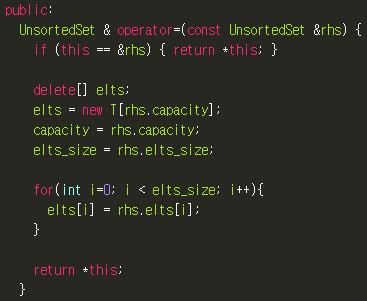
UnsortedSet Copy Constructor

This is shallow copy,  
create custom copy constructors that do deep copies instead

This is deep copy,  
allocating our own array with the same size and  
copying over each element individually.

Initialization vs Assignment  
  
  
initialization is giving a  
Assignment is giving a new value  
first-time value as part  
to an object that already exists  
of creating the object

The Assignment Operator

Assigning the value of one compound object to another uses an overloaded assignment operator  
  
check for self-assignment  
  
kill old array  
Make new array

Copy over all the elements

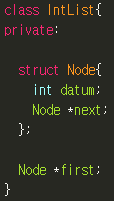
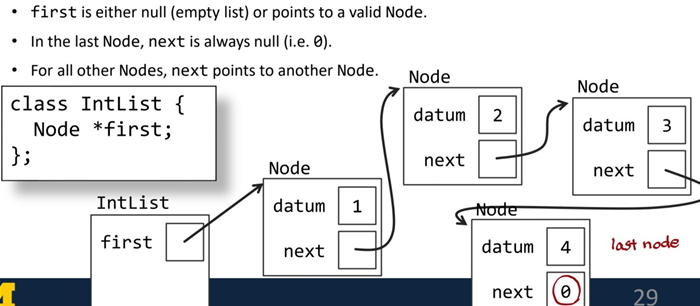
* If it’s the same line as declaration, we use copy ctr  
  Ex) UnsortedSet<int> s2 = s1

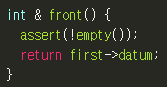
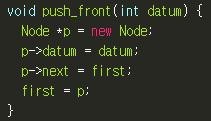
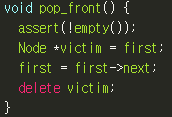
The Big Three : Destructor, Copy Constructor, Assignment Operator

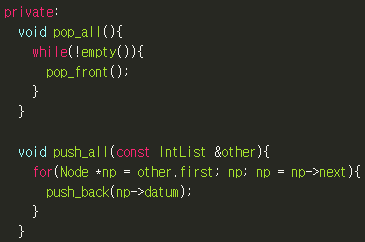
* If you need to provide a custom implementation for any of them, you almost certainly need to provide a custom implementation for all of them.

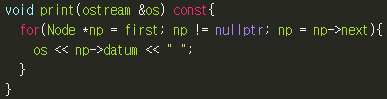
When? If you need a deep copy. Check the constructor, if it creates dynamic memory, prob need big three. If some members are pointers, prob need the big three.

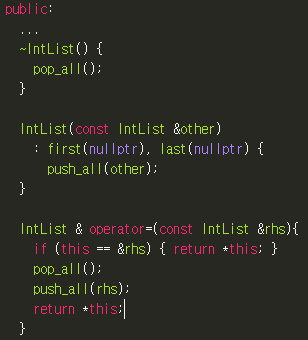
Linked List  
motivation: What if we want to add/remove an element in the beginning

* Allow data to be stored non-contiguously(인접하여)
* If we want to insert 2 in the container, we just put ‘2’ into any memory and set as next pointer
* Linked list = a collection of nodes
* Each node consists of a piece of data the address of another node
* Put the Node struct itself insde the IntList class (Node can only be used inside the class and its member functions)

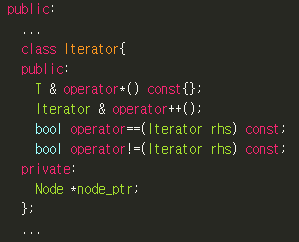


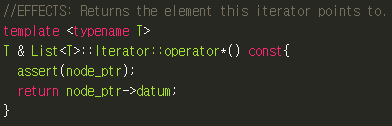
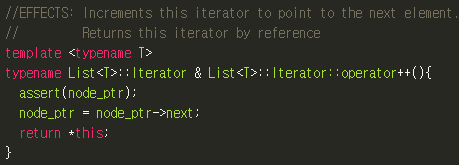


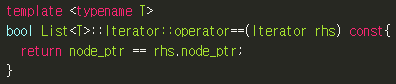
Big Three of IntList

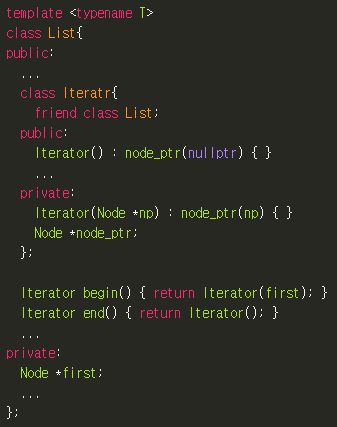
* Destructor - Free resources : pop\_all()
* Copy Constructor – Deep copy resources from other : push\_all()
* Assignment Operater – Free old resources, Deep copy resources from rhs : pop\_all(), push\_all()

Iterators  
motivation: Learn how to access the Nth element of a linked list using iterators  
(Through for loop won’t work since Node type is private and it would break the interface)  
-> Traversal by pointer (We want a public interface to traverse through a linked list)

* An iterator is an object that works like a pointer
* This can be implemented by a class that overloads the appropriate operators (\*, ++, ==, !=)
* The Iterator class is defined inside the List class
* Store a pointer to the node holding the current element

The \* operator The ++ operator (prefix)

The == operator

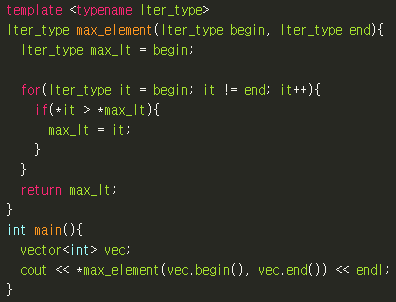
Friend Declarations

The begin() uses private (first). We are calling this inside of the list class so it has access to list private member variables but it’s outside of the nested iteration class.

Therefore, we use a friend declaration to give List special privileges to access the private members of Iterator.

Begin() can now access the private constructor.

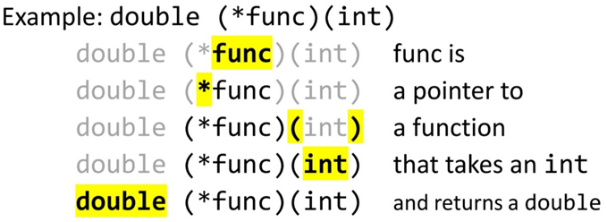
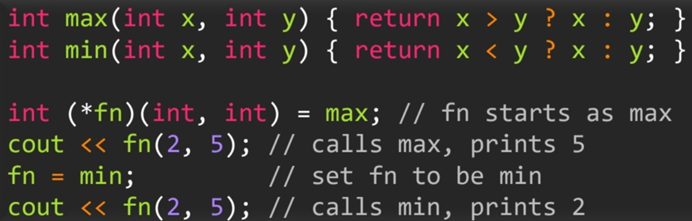
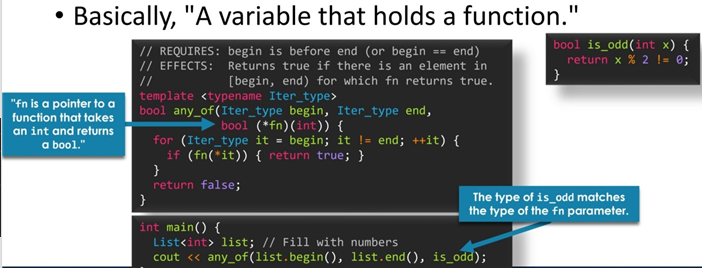
Generic Function  
motivation: Most containers include an associated Iterator data type

Max\_element

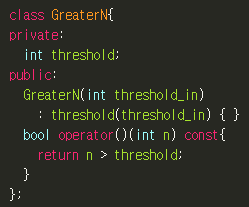
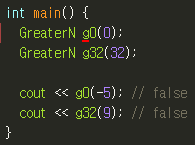
Here, Iter\_type evaluates to  
vector<int>::iterator

As long as we are working with a container that supports iterators, we don’t ever have to write that maximum-finding again.

Function Pointers

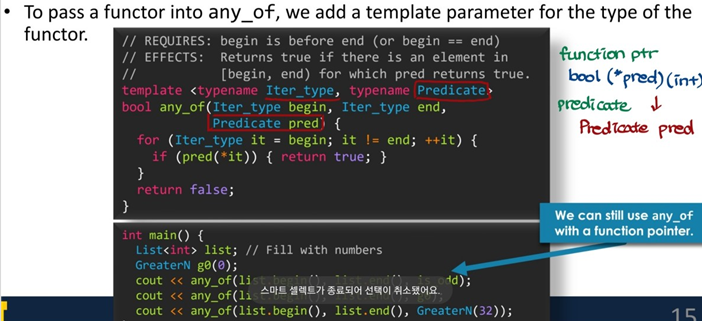
* C++ declarations are read inside out
* Double (\*func)(int)

Functors

* Class type objects that pretend to be functions

here, ( ) is ctor

here, ( ) is calling function



Functor as Parameters

Comparators

* Function that compares two arguments and returns true/false depending on their ordering