EECS 280 – Lecture 12

Containers Part 2 and Templates

2/21/2022

Time Complexity

- Question: How long does an algorithm take to execute relative to its input size?
- This is the **time complexity** of an algorithm.

- Some common complexities:
  - $O(1)$ – Constant Time.
    e.g. Accessing an element at index `i`
  - $O(n)$ – Linear Time.
    e.g. Printing out all elements in an array.
  - $O(n^2)$ – Quadratic Time.
    e.g. Computing the energy of a `Matrix` with side length `n`.

Set Efficiency

```
IntSet::IntSet() : elts_size(0) { }
```

```
IntSet::size() { return elts_size; }
```

```
void IntSet::insert(int v) {
  assert(size() < ELTS_CAPACITY);
  if (contains(v)) {
    return;
  }
  elts[elts_size] = v;
  ++elts_size;
}
```

```
void IntSet::remove(int v) {
  if (!contains(v)) {
    return;
  }
  elts[indexOf(v)] = elts[elts_size - 1];
  --elts_size;
}
```

```
bool IntSet::contains(int v) const {
  return indexOf(v) != -1;
}
```

Question

Which column in the table shows the correct time complexities for the IntSet functions?

A B C D

	Time A	Time B	Time C	Time D
ctor	O(1)	O(1)	O(n)	O(1)
size	O(1)	O(1)	O(1)	O(1)
insert	O(n)	O(n)	O(n)	O(1)
remove	O(1)	O(n)	O(n)	O(1)
contains	O(1)	O(n)	O(n)	O(n)

```
int IntSet::indexOf(int v) const {
  for (int i = 0; i < elts_size; ++i) {
    if (elts[i] == v) {
      return i;
    }
  }
  return -1;
}
```

SortedIntSet

- Let's consider another data **representation** for a different implementation of the set interface.
- Idea:
  - Store a **sorted** array of integers in the set.
  - Store how many elements are used.

`elts`

3	5	6	7						
0	1	2	3	4	5	6	7	8	9

`elts_size`

4

```
class SortedIntSet {
private:
  int elts[ELTS_CAPACITY];
  int elts_size;
};
```

Representation Invariants

- What representation invariants do we need for the `SortedIntSet` class?

```
class SortedIntSet {
private:
  int elts[ELTS_CAPACITY];
  int elts_size;
};
```

Valid Size

`0 <= elts_size`

`elts_size <= ELTS_CAPACITY`

Valid Elements

The first `elts_size` elements of `elts` comprise the set <u>and are in sorted order.</u> No duplicates.

`elts`

3	**5**	**6**	**7**						
0	1	2	3	4	5	6	7	8	9

`elts_size`

4

Solution: SortedIntSet::insert

```
void SortedIntSet::insert(int v) {
  assert(size() < ELTS_CAPACITY);
  if (!contains(v)) {
    int index = elts_size;
    while (index > 0 && elts[index - 1] > v) {
      elts[index] = elts[index - 1];
      --index;
    }
    elts[index] = v;
    ++elts_size;
  }
}
```

elts

3	**5**	**6**	**7**						
0	1	2	3	4	5	6	7	8	9

elts_size

4

Solution: SortedIntSet::remove

```
void SortedIntSet::remove(int v) {
  if (!contains(v)) {
    return;
  }
  for (int i = indexOf(v); i < elts_size - 1; ++i) {
    elts[i] = elts[i + 1];
  }
  --elts_size;
}
```

elts

3	5	6	7						
0	1	2	3	4	5	6	7	8	9

elts_size

4

The Advantage of Sorting

- We can now use a binary search for `indexOf`, which will run much faster than a linear search.

-7	-5	-1	0	2	4	5	8	11	17
0	1	2	3	4	5	6	7	8	9

```
int SortedIntSet::indexOf(int v) const {
  int start = 0;
  int end = elts_size;
  while (start < end) {
    int middle = start + (end - start) / 2;
    if (v == elts[middle]) {
      return middle;
    } else if (v < elts[middle]) {
      end = middle;
    } else {
      start = middle + 1;
    }
  }
  return -1;
};
```

Set Efficiency

- How efficient is each operation?

	`IntSet`	`SortedIntSet`	`???`
`insert`	O(n)	O(n)	
`remove`	O(n)	O(n)	
`contains`	O(n)	O(log n)	
`size`	O(1)	O(1)	
constructor	O(1)	O(1)	

Binary Search

Single-Type Containers

- Idea[1]: Let's just copy and paste `IntSet`, then change `int` to `char` everywhere. Easy.

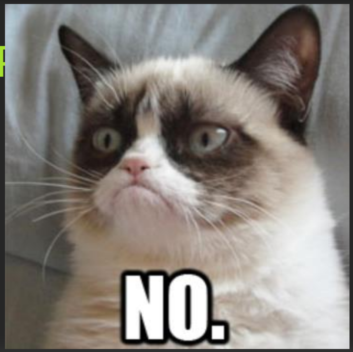
```
class IntSet {
public:
  ...
  void insert(int v);
  void remove(int v);
  bool contains(int v) const;
  int size() const;
  ...

private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

```
class CharSet {
public:
  ...
  void insert(char v);
  void remove(char v);
  bool contains(char v) const;
  int size() const;
  ...

private:
  char elts[ELTS_CAI
  int elts_size;
  ...
};
```

Is this a good approach?



Single-Type Containers

- Better Idea: Write all the code in a generic way, then let the compiler copy/paste for us.

```
class IntSet {
public:
  ...
  void insert(int v);
  void remove(int v);
  bool contains(int v) const;
  int size() const;
  ...

private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

```
template <typename T>
class UnsortedSet {
public:
  ...
  void insert(T v);
  void remove(T v);
  bool contains(T v) const;
  int size() const;
  ...

private:
  T elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

Templates

- A **template** is a model for producing code.
  - You write a generic, flexible version.
  - The compiler **instantiates** hard copies of code from the template as needed.

- Flexibility comes from **template parameters**.

```
template <typename T>
class UnsortedSet {
  ... // use T in code
};
```

T can become any type when the template is instantiated!

1 You can use any name for the template parameter.
`Value_type` is another common name when working with containers.

Using Templates

- The compiler can see which kinds of `UnsortedSet<T>` you use and instantiates (produces) a version of the code **for each different type `T`**.

T=int

T=char

T=Card

```
int main() {
  UnsortedSet<int> is;
  is.insert(3);
  is.insert(7);
  is.insert(8);
  cout << is; // { 3, 7, 8 }

  UnsortedSet<char> cs;
  cs.insert('a');
  cs.insert('e');
  cs.insert('i');
  cout << cs; // { a, e, i }

  UnsortedSet<Card> ds;
}
```

Note: These containers are still homogenous. T can only be one type per template instantiation.

Using Templates

UnsortedSet.h

```
template <typename T>
class UnsortedSet {
public:
  ...
  void insert(T v);
  void remove(T v);
private:
  T elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

```
#include "UnsortedSet.h"
int main() {
  UnsortedSet<int> is;
  UnsortedSet<Card> ds;
}
```

The compiler instantiates the template as needed according to how it is used in the code.

```
class UnsortedSet<int> {
public:
  ...
  void insert(int v);
  void remove(int v);
private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

```
class UnsortedSet<Card> {
public:
  ...
  void insert(Card v);
  void remove(Card v);
private:
  Card elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

Function Templates

- A **function template** can be instantiated to make versions to work with different types of inputs.

```
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}

int main() {
  int i = max(3, 10);
  double d = max(3.14, 3.33);

  Card c1(Card::RANK_ACE, Card::SUIT_CLUBS);
  Card c2(Card::RANK_TEN, Card::SUIT_HEARTS);
  Card best_card = max(c1, c2);
}
```

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

We'll start again in five minutes.

Compiling Templates

- A template parameter can potentially take on any type, but it might not compile!

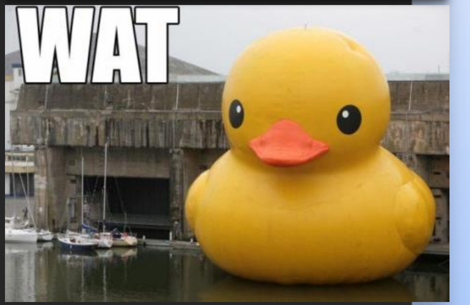
```
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}

int main() {

  Duck d1("Donald");
  Duck d2("Scrooge");
  Duck best_duck = max(d1, d2);
  cout << best_duck.getName() << " wins!" << endl;

}
```

This doesn't compile.



Compiling Templates

1. Instantiate the template for the given type.
2. Check to make sure it compiles.

Instantiated with T = Duck

```
Duck max(Duck val1, Duck val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}

int main() {

  Duck d1("Donald");
  Duck d2("Scrooge");
  Duck best_duck = max(d1, d2);
  cout << best_duck.getName() << " wins!" << endl;
}
```

Error: Operator > is not defined for types Duck, Duck.

When you get an error, look for where it was used. duck.cpp:15:30 required from here.

Compiling Templates

- Template instantiation is done during compilation proper, which is before the linking phase.
- This means that definitions for all functions must be included in each compilation unit.
- We can't exactly split into `.h` and `.cpp` as usual.

library.h

```
template <typename T>
T max(T val1, T val2);
```

main.cpp

```
#include "library.h"
int main() {
  int i = max(3, 10);
}
```

Error: Definition is needed here to instantiate the template.

library.cpp

```
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}
```

```
g++ main.cpp library.cpp
(The linking step)
```

Compiling Templates

- Basically, we need to get everything into the `.h` file.
- Idea: You could just literally put everything there…

library.h

```
template <typename T>
T max(T val1, T val2);
...
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}
```

We declare the interface at the top of the file.

The implementation comes later to keep it separate.

main.cpp

```
#include "library.h"
int main() {
  int i = max(3, 10);
}
```

OK: Definition is available now.

```
g++ main.cpp
(The linking step)
```

For logistical reasons, we will use this pattern on P4 and P5.

Compiling Templates

- Basically, we need to get everything into the `.h` file.
- Better idea: `#include` the implementation at the bottom of the `.h` file. Call it a `.tpp` file in this pattern.
  - We get still get everything in the `.h`, but we have a clean separation of the interface and implementation.

library.h

```
template <typename T>
T max(T val1, T val2);
#include "library.tpp"
```

library.tpp

```
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}
```

main.cpp

```
#include "library.h"
int main() {
  int i = max(3, 10);
}
```

OK: Definition is available now.

```
g++ main.cpp
(The linking step)
```

library.tpp not needed here.

For logistical reasons, we will NOT use this pattern on P4 or P5.

Include Guards

- It's possible to accidentally **`#include`** a library twice. This causes compile errors.

Don't change the given include guards on the projects!

library.h

```
#ifndef LIBRARY_H
#define LIBRARY_H

// Some library code

#endif /* LIBRARY_H */
```

Add include guards to prevent this.

library2.h

```
#include "library.h"
// Some other library
// code that builds on
// the stuff in the
// first library
```

main.cpp

```
#include "library.h"
#include "library2.h"
int main() {
   // Use the libraries
}
```

Member Function Templates

```
template <typename T>
class UnsortedSet {
public:
  void insert(T v);
...
};

template <typename T>
void UnsortedSet<T>::insert(T v) {
  assert(size() < ELTS_CAPACITY);
  if (contains(v)) {
    return;
  }
  elts[elts_size] = v;
  ++elts_size;
}
...
```

The definition also needs to be a template.

We need to specify this is a member of `UnsortedSet<T>`.

Exercise: `fillFromArray`

- Write a function template that fills an `UnsortedSet<T>` with elements from an array of `T`.

```
template <typename T>
void fillFromArray(          set,           arr, int n) {

  // YOUR CODE HERE

}

int main() {
  UnsortedSet<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fillFromArray(set1, arr1, 4);

  UnsortedSet<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fillFromArray(set2, arr2, 3);
}
```

Also fill in the missing parameter types!

Solution: `fillFromArray`

- Write a function template that fills an `UnsortedSet<T>` with elements from an array of `T`.

```
template <typename T>
void fillFromArray(UnsortedSet<T> &set, const T *arr, int n) {
  for (int i = 0; i < n; ++i) {
    set.insert(arr[i]);
  }
}

int main() {
  UnsortedSet<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fillFromArray(set1, arr1, 4);

  UnsortedSet<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fillFromArray(set2, arr2, 3);
}
```

Static vs. Dynamic Polymorphism

- Recall: Polymorphism is a property where one thing can take on many forms.
- The template mechanism gives you **parametric polymorphism**.
  - The template parameter `T` can take the form of `int`, `char`, etc.

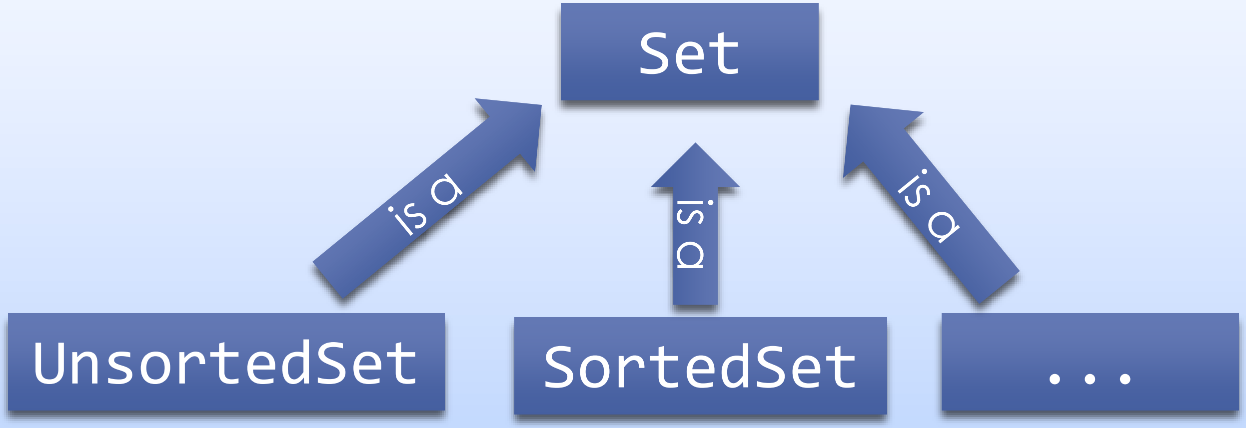
Happens at compile time. We call this static polymorphism.

- Compare this with **subtype polymorphism**.
  - A pointer to a base class can take the form of any derived class (i.e. use dynamic type).

Happens at runtime. We call this dynamic polymorphism.

Static vs. Dynamic Polymorphism

- We could have implemented the two set ADTs using **dynamic polymorphism**.



- We chose **static polymorphism** instead.
  - Dynamic polymorphism requires member lookup at runtime, so there is an extra cost.
  - The type of set we're using doesn't change at runtime, so we don't need dynamic polymorphism.

Using Sets

- With static polymorphism, if we decide to use `SortedSet` instead, there may be a lot of places where we need to change the type.

```
template <typename T>
void fillFromArray(UnsortedSet<T> &set, const T *arr,
                   int n);

int main() {
  UnsortedSet<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fillFromArray(set1, arr1, 4);

  UnsortedSet<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fillFromArray(set2, arr2, 3);
}
```

Type Aliases

- Instead, we can introduce a type alias with the `using` keyword.

```
template <typename T>
using Set = UnsortedSet<T>;

template <typename T>
void fillFromArray(Set<T> &set, const T *arr,
                   int n);

int main() {
  Set<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fillFromArray(set1, arr1, 4);

  Set<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fillFromArray(set2, arr2, 3);
}
```

Now this is the only place we need to change.