EECS 280 – Lecture 11

Containers ADTs

2/14/2022

Review: Information Hiding in C++

`Triangle.h`
Interface

- What a `Triangle` does.

```
class Triangle {
private:
  double a, b, c;

public:
  Triangle();
  Triangle(double a_in,
           double b_in,
           double c_in);
  double area() const;
  double perimeter() const;
  void scale(double s);
};
```

`Triangle.cpp`
Implementation

- Details of how it does it.

```
#include "Triangle.h"

Triangle::Triangle(double a_in,
  double b_in, double c_in)
  : a(a_in), b(b_in), c(c_in) { }



void Triangle::scale(double s) {
  a *= s;
  b *= s;
  c *= s;
}
```

The <u>scope resolution operator</u> (::) allows us to refer to the member function from outside.

Building a Container ADT

- A **container** is an ADT whose purpose is to hold other objects.
- Examples:
  - arrays
  - `vector`
- Let's add another: `IntSet`
  - A set is an unordered collection of unique elements. In this case, integers.

Using a Set

What can you do with a set?

- Insert a value
- Remove a value
- Check if a value is in the set
- Get the size of the set
- Print out the set

```
int main() {
  IntSet set;
  set.insert(7);
  set.insert(32);

  cout << "Size: " << set.size() << endl;
  set.print(cout);

  set.insert(42);
  set.remove(32);

  cout << "Contains 32? " << set.contains(32) << endl;
  cout << "Contains 42? " << set.contains(42) << endl;
}
```

Motivation: Why sets?

- Task: Find a list of the unique words in a Piazza Post.*
- The right data structure makes the algorithm easy.
  - Insert each word into a set. Print the set. Done.

```
set<string> unique_words(const string &str) {
 istringstream source(str);
 set<string> words;
 string word;

 // Read word by word from the
 // stringstream and insert into the set
 while (source >> word) {
   words.insert(word);
 }
 return words;
}
```

*In P5, we give you a function that does precisely this!

The IntSet Interface (IntSet.h)

```
class IntSet {
public:
  // Maximum capacity of a set.
  static const int ELTS_CAPACITY = 10;
  // REQUIRES: size() < ELTS_CAPACITY
  // EFFECTS:  adds v to the set
  void insert(int v);

  // EFFECTS: removes v from the set
  void remove(int v);

  // EFFECTS: returns whether v is in the set
  bool contains(int v) const;

  // EFFECTS: returns the number of elements
  int size() const;

  // EFFECTS: prints out the set
  void print(std::ostream &os) const;
};
```

We'll look at the static keyword on the next slide.

These are all just member function <u>declarations</u>. We'll write all of the <u>implementations</u> separately in the .cpp file.

These consts mean the functions don't modify the IntSet.

We'll add private members soon!

Diversion: Static Data Members

- A data member declared using the `static` keyword is "shared" among all instances of the class.

```
class IntSet {
public:
  // Maximum capacity of a set.
  static const int ELTS_CAPACITY = 10;

  ...
};
```

This is commonly used for constants.

Note this is just one of many different uses of the static keyword.

Diversion: Static Data Members

- A data member declared using the `static` keyword is "shared" among all instances of the class.
- It's like a global variable, but better.
  - It still has static storage duration, meaning it lives throughout the whole program, just like a global.
  - But it lives inside a class's scope – more organized than just being in the global scope.
- To access outside class scope, use `IntSet::ELTS_CAPACITY`.

```
class IntSet {
public:
  // Maximum capacity of a set.
  static const int ELTS_CAPACITY = 10;

  ...
};
```

This is commonly used for constants.

Note this is just one of many different uses of the static keyword.

Why Fixed Capacity?

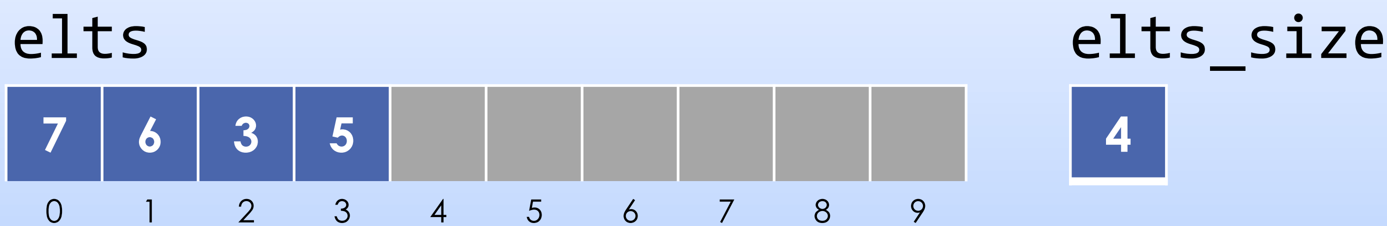
- Basically, our implementation needs to know how much space to allocate.
  - Right now, this has to be known at compile time (e.g. the size of an array to store elements in the `IntSet`).
  - When we learn about dynamic memory, we'll see how to fix this…

```
class IntSet {
public:
  // Maximum capacity of a set.
  static const int ELTS_CAPACITY = 10;

  ...
};
```

IntSet Data Representation

- First, let's pick a **representation** for the data. What do we need to store?
  - Store an array of the integers in the set.
  - Store how many array elements are being used.



The public interface from earlier.

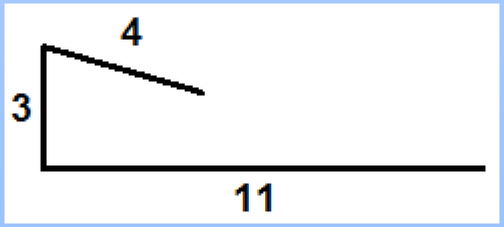
```
class IntSet {
  ...
private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

Recall: Representation Validity

- Data **representation** doesn't always match the desired **abstraction** perfectly.
- Example:
  Represent a `Triangle` as three doubles.

```
class Triangle {
  double a;
  double b;
  double c;
};
```

- Problem:
  This is too flexible! Some combinations of three doubles are not **valid** `Triangles`!



Recall: Representation Invariants

- A problem for compound types…
  - Some combinations of member values don't make sense together.
- We use **representation invariants** to express the conditions for a **valid** compound object.
- For Triangle:

Nonnegative Edge Lengths	Triangle Inequality
`0 < a`	`a + b > c`
`0 < b`	`a + c > b`
`0 < c`	`b + c > a`

Representation Invariants

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

```
class IntSet {
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. No duplicates.

`elts`

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

`elts_size`

4

Containers and `size_t`

- The STL defines a special unsigned integer type, `size_t`.
  - `size_t` is guaranteed to hold numbers large enough to represent the largest possible size of an object or a container.
- Many container ADTs use `size_t`.

```
class IntSet {
public:
  static const size_t ELTS_CAPACITY = 10;
  void insert(int v);
  size_t size() const;
  ...
private:
  size_t elts_size;
  ...
};
```

IntSet using size_t.

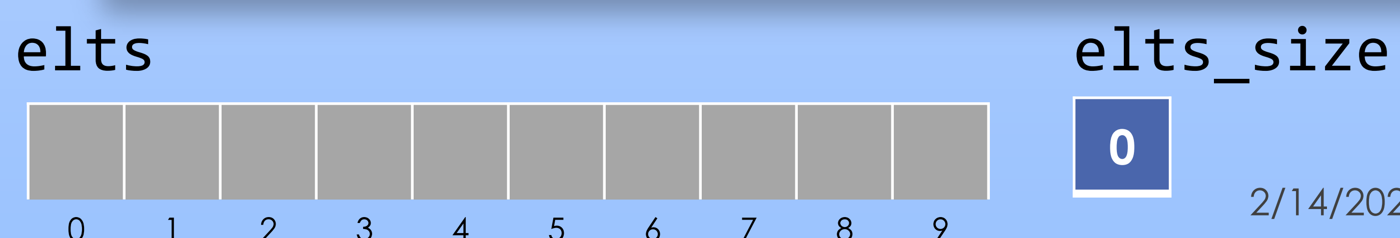
- The STL generally uses `size_t` for containers (e.g. `std::vector`). However, in these lecture slides, we use regular `int`.

IntSet Constructor

- We need to ensure that the representation invariants are initially set up correctly.
- Let's do this with a constructor.

```
class IntSet {
public:
  IntSet();
  ...
};
```

Again, we only declare the constructor here (in the .h file) because all implementation details should go in the .cpp file.



An `IntSet` Implementation (`IntSet.cpp`)

- We define all our member functions separately in the `.cpp` file (using the scope resolution operator `::`).

Constructor implementation.

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

void IntSet::insert(int v) {
  // CODE
}
void IntSet::remove(int v) {
  // CODE
}
bool IntSet::contains(int v) const {
  // CODE
}

...
```

Code Reuse

- Observe that both `remove` and `contains` need to find "where" an element is.
- Let's write a private member function that serves as a helper.

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

If an element is found, returns its index.

If not found, return -1.

IntSet::contains

- Now, let's write implementations for the member functions specified in the `IntSet` Interface.

```
bool IntSet::contains(int v) const {

  return indexOf(v) != -1;

}
```

elts

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

elts_size

4

Exercise: IntSet::insert

Question

Which of these are correct?

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

```
void IntSet::insert(int v) {
  assert(size() < ELTS_CAPACITY);

  ++elts_size;
  elts[elts_size - 1] = v;
}
```

```
void IntSet::insert(int v) {
  assert(size() < ELTS_CAPACITY);
  if (contains(v)) { return; }
  elts[0] = v;
  ++elts_size;
}
```

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

elts

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

elts_size

4

Exercise: IntSet::remove

<u>Question</u>

Which of these are correct?

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

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

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

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

elts

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

elts_size

6

Insertion (Output) Operator

- Some operator overloads use **non-member** functions.
- If we implement the << operator, we can `cout` sets.
  - Uses a **non-member** function named `operator<<`.

IntSet.h

```
class IntSet {
  ...
};

std::ostream &operator<<(std::ostream &os, const IntSet &s);
```

IntSet.cpp

```
std::ostream &operator<<(std::ostream &os, const IntSet &s) {
  s.print(os);
  return os;
}
```

```
int main() {
  IntSet set;
  cout << set;
}
```

```
int main() {
  IntSet set;
  operator<<(cout, set);
}
```

Subscript Operator

- Other operator overloads use **member** functions.
- Let's use the [] operator to check for an element.
  - Uses a **member** function named `operator[]`.

IntSet.h

```
class IntSet {
public:
  ...
  bool operator[](int v) const;
};
```

IntSet.cpp

```
bool IntSet::operator[](int v) const {
  return contains(v);
}
```

```
int main() {
  IntSet set;
  ...
  if( set[32] ) {
    ...
```

```
int main() {
  IntSet set;
  ...
  if( set.operator[](32) ) {
    ...
```

Exercise: Overloading +=


```
class IntSet { ... };

int main() {
  IntSet set;
  set += 3;
  set += 5;
  cout << set; // {3, 5}
}
```

<u>Question</u>

Which of these are correct overloads for the += operator, implemented as a <u>member</u> function?

```
void operator+=(IntSet &s, int v) {
  s.insert(v);
}
```

```
void operator+=(IntSet &s, int v) {
  this->insert(v);
}
```

```
void IntSet::operator+=(int v) {
  this->insert(v);
}
```

```
void IntSet::operator+=(int v) {
  insert(v);
}
```

```
void IntSet::operator+=(IntSet &s, int v) {
  s.insert(v);
}
```

**The function could also return the given `IntSet` to support operator chaining.*