Slides by Andrew DeOrio

# EECS 280

## Programming and Introductory Data Structures

Interfaces and Invariants

# Review: data abstraction

- **Data abstraction** lets us separate *what* a type is (and what it can do) from *how* the type is implemented

- In C++, we use a `class` to implement data abstraction
  - We can create an Abstract Data Type (ADT) using a `class`

- ADTs let us model complex phenomena
  - More complex than built-in data types like `int`, `double`, etc.

- ADTs make programs easier to maintain and modify
  - You can change the implementation and no users of the type can tell

# Review: data abstraction

- Two benefits of data abstraction

1. Information Hiding

- Protect and hide our code from other code that that uses it

2. Encapsulation

- Keeping data and relevant functions together

# Review: containers

- The purpose of a *container* is to hold other objects
- For example, an array can hold integers:

```
int array[10];
a[4] = 517;
```

- We can use a class to create containers with more features, like a mathematical set of integers

```
class IntSet {
  //...
};
```

# IntSet ADT

- Our `IntSet` and application that uses it are organized into three files
- Abstract description of values and operations
  - *What* the data type does, but not *how*

`IntSet.h`

- Implementation of the ADT
  - *How* the data type works

`IntSet.cpp`

- Using the ADT

`main.cpp`

IntSet.h

# Review: IntSet

```
class IntSet {
  //OVERVIEW: mutable set of ints with bounded size
 public:
  IntSet();
  void insert(int v);
  void remove(int v);
  bool query(int v) const;
  int size() const;
  void print() const;
  static const int ELTS_CAPACITY = 100;
 private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  int indexOf(int v) const;
};
```

IntSet.cpp

# Review: IntSet

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

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

bool IntSet::query(int v) const  {
  return indexOf(v) != ELTS_CAPACITY;
}
// ...
```

main.cpp

# Using IntSet

```
#include "IntSet.h"
int main () {
  IntSet is;
  is.insert(7);
  is.insert(4);
  is.insert(7);
  is.print();
  is.remove(7);
  is.print();
  return 0;
}
```

```
g++ IntSet.cpp main.cpp
./a.out
{ 7 4 }
{ 4 }
```

IntSet.h

# Problem with IntSet.h

```
class IntSet {
  //OVERVIEW: mutable set of ints with bounded size
 public:
  IntSet();
  void insert(int v);
  void remove(int v);
  bool query(int v) const;
  int size() const;
  void print() const;
  static const int ELTS_CAPACITY = 100;
 private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  int indexOf(int v) const;
};
```

**Problem:** `IntSet` mixes abstraction (public members functions) and implementation (private member variables

# Problem with IntSet.h

- Member data in the declaration have two undesirable effects:

1. It complicates the class declaration, making it harder to read and understand
2. It communicates information to the programmer that s/he isn't supposed to know

# Interfaces

- An *interface*-only abstract class provides two main benefits

1. Provide a class declaration without any member variables
2. Force all derived classes to behave the same way

- Our interface class will never be instantiated because there is no implementation
- In C++, we will use pure virtual functions

# Review: pure virtual functions

- Because there will be no implementation, we need to declare member functions in a special way:
  - Declare each method as a `virtual` function
  - “Assign" a `0` to each of these virtual functions
- These are called *pure virtual functions*

```
class Shape {
public:
  virtual double area() const = 0;
  virtual void print() const = 0;
};
```

IntSet.h

# Exercise

- Convert this class to an abstract class

```
class IntSet {
  //OVERVIEW: mutable set of ints with bounded size
 public:
  IntSet();
  void insert(int v);
  void remove(int v);
  bool query(int v) const;
  int size() const;
  void print() const;
  static const int ELTS_CAPACITY = 100;
 private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  int indexOf(int v) const;
};
```

IntSet.h

# IntSet abstract class

```
class IntSet {
  //OVERVIEW: mutable set of ints with bounded size
 public:
  virtual void insert(int v) = 0;
  virtual void remove(int v) = 0;
  virtual bool query(int v) const = 0;
  virtual int size() const = 0;
  virtual void print() const = 0;
  static const int ELTS_CAPACITY = 100;
};
```

- No constructor
- No private member variables
- All pure virtual functions

# Abstract class review

- A class with any *pure virtual function* is an *abstract class*
- You **cannot** create an instance of an abstract class, because there can be no implementation

```
int main() {
  IntSet is;   //compile error!
  return 0;
}
```

- You **can** create a pointer or reference to an abstract class

```
int main() {
  IntSet *is_ptr;  //OK
  IntSet &is_ref;  //OK
  return 0;
}
```

# Implementing the interface

- Now, we will provide an implementation in a derived class

```
class IntSetUnsorted : public IntSet {
  //IntSetUnsorted inherits all IntSet's member
  //functions and overrides them
};
```

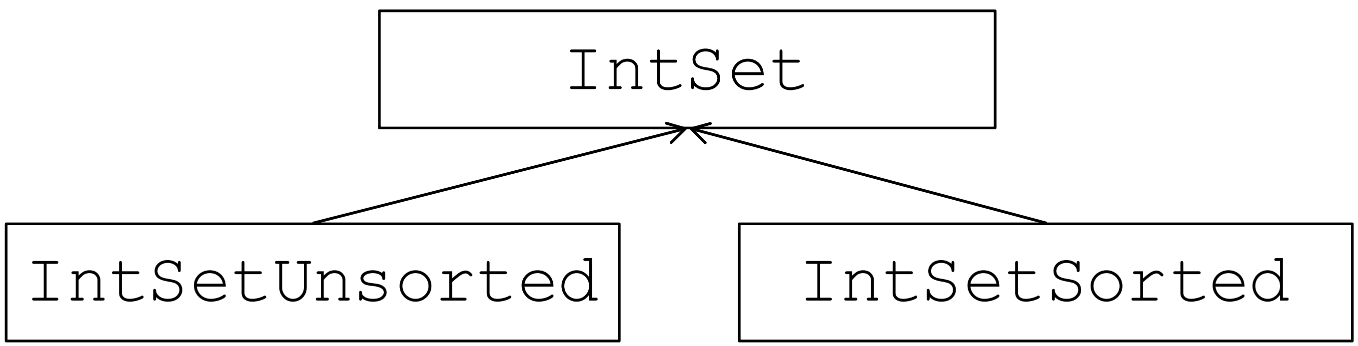
- We can even write multiple implementations

```
class IntSetSorted : public IntSet {
  //IntSetSorted inherits all IntSet's member
  //functions and overrides them
};
```

- Both derive from `IntSet`, so both will behave the same way
- Exercise: draw the class hierarchy

# Class hierarchy

```
class IntSet { /*...*/ };
class IntSetUnsorted : public IntSet { /*...*/ };
class IntSetSorted : public IntSet { /*...*/ };
```



# Implementing the interface

- Override all the member functions

```
class IntSetUnsorted : public IntSet {
 public:
  IntSetUnsorted();
  virtual void insert(int v);
  virtual void remove(int v);
  virtual bool query(int v) const;
  virtual int size() const;
  virtual void print() const;
 private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  int indexOf(int v) const;
};
```

# Implementing the interface

- The derived class includes a constructor

```
class IntSetUnsorted : public IntSet {
 public:
  IntSetUnsorted();
  virtual void insert(int v);
  virtual void remove(int v);
  virtual bool query(int v) const;
  virtual int size() const;
  virtual void print() const;
 private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  int indexOf(int v) const;
};
```

# Implementing the interface

- Sorted version (similar code as last lecture)

```
class IntSetSorted : public IntSet {
 public:
  IntSetSorted();
  virtual void insert(int v);
  virtual void remove(int v);
  virtual bool query(int v) const;
  virtual int size() const;
  virtual void print() const;
 private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  int indexOf(int v) const;
};
```

IntSet.h

# Using the interface

- To use our `IntSet` interface, all we need to know are the public member functions in `IntSet.h`

```
class IntSet {
  //OVERVIEW: a mutable set of ints with bounded size
public:
  virtual void insert(int v) = 0;
  virtual void remove(int v) = 0;
  virtual bool query(int v) const = 0;
  virtual int size() const = 0;
  virtual void print() const = 0;
  static const int ELTS_CAPACITY = 100;
};
```

main.cpp

# Using the IntSet interface

```
#include "IntSet.h"
int main () {
  IntSet is;
  is.insert(7);
  is.insert(4);
  is.insert(7);
  is.print();
  is.remove(7);
  is.print();
  return 0;
}
```

Q: What’s wrong with this code?

main.cpp

# Using the IntSet interface

```
#include "IntSet.h"
int main () {
  IntSet is;
  is.insert(7);
  is.insert(4);
  is.insert(7);
  is.print();
  is.remove(7);
  is.print();
  return 0;
}
```

Q: What's wrong with this code?

A: It won't compile! `IntSet` is an abstract class. You can't create an instance of an abstract class.

# Review: factory functions

```
//EFFECTS: asks user to select a shape
//         returns a pointer to correct object
Shape * ask_user();
```

- `ask_user()` is an example of a *factory function*
- A factory function creates objects for another programmer who doesn't need to know their actual types
- That's exactly what we want for `IntSetSorted` and `IntSetUnsorted`

# IntSet factory function

```
// Dirty global variable trick.  We'll fix this
// when we learn about dynamic memory.
static IntSetUnsorted g_is;

//EFFECTS: returns a pointer to an IntSet
IntSet * IntSet_factory() {
  return &g_is;
}
```

main.cpp

# Using the IntSet interface

```
#include "IntSet.h"
int main () {
  IntSet is;
  is.insert(7);
  is.insert(4);
  is.insert(7);
  is.print();
  is.remove(7);
  is.print();
  return 0;
}
```

Exercise: adapt this code to use the new factory function.
Hint: you'll need to use pointers

main.cpp

# Using the IntSet interface

```
#include "IntSet.h"
int main () {
  IntSet *is = IntSet_factory();
  is->insert(7);
  is->insert(4);
  is->insert(7);
  is->print();
  is->remove(7);
  is->print();
  return 0;
}
```

Solution

main.cpp

# Using the IntSet interface

```
#include "IntSet.h"
int main () {
  IntSet *is = IntSet_factory();
  is->insert(7);
  is->insert(4);
  is->insert(7);
  is->print();
  is->remove(7);
  is->print();
  return 0;
}
```

```
static IntSetSorted g_is;
IntSet * IntSet_factory() {
  return &g_is;
}
```

Here's the cool part: we can switch implementations, and everything still works!

Why? *Liskov Substitution Principle*

# Invariant review

- An *invariant* is something that is always true
- In other words, it is a set of conditions that must always evaluate to true at certain well-defined points in the program, otherwise, the program is incorrect

- We've seen three kinds of invariants:
  - Recursive invariant
  - Iterative invariant
  - Representation invariant

# Recursive invariant review

- A *recursive invariant* applies to the arguments of a recursive function
- A *recursive invariant* describes the conditions that must be satisfied by any call to the function
- For example, in our tail-recursive factorial function, the recursive invariant was `n! * result == num!`

```
int fact_helper(int n, int result) {/*...*/}

int factorial(int num) {
  return fact_helper(num, 1);
}
```

# Iterative invariant review

- An *iterative invariant* applies to each loop variable
- An *iterative invariant* must be true before and after each iteration
- For example, in our iterative factorial function, the iterative invariant was `n! * result == num!`

```
int  factorial(int num) {
  int n = num;
  int result = 1;
  while (n!=0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

# Representation invariant review

- A *representation invariant* applies to the data members of an ADT
  - Recall that member variables are a class's representation
- A *representation invariant* must be true immediately before and immediately after any member function execution

- Think of it as a "sanity check" for the ADT

# Representation invariant details

- A *representation invariant* must be true immediately before and immediately after any member function execution

- Each method in the class is free to assume that the invariant is true on entry if:
  - The representation invariant holds at the required times, **and**
  - Each data element is truly private

- This is true because the only code that can change them belongs to the methods of that class, and those methods always establish the invariant

# Representation invariant details

- A *representation invariant* must be true immediately before and immediately after any member function execution

- **Exception**: constructors
- The constructor establishes the representation invariant, so the representation invariant only has to hold at the end

- **Exception**: destructors
- There is one sort of method, called a *destructor*, where the representation invariant only has to hold at the beginning
  - We won't see this until later

# Representation invariant example

- We've seen two examples of representation invariants, both applying to the private members of an `IntSet` representation:

```
int elts[ELTS_CAPACITY];
int elts_size;
```

- For the unsorted version, the invariant is:
  - The first `elts_size` members of `elts` contain the integers comprising the set, with no duplicates.

- For the sorted version, the invariant is:
  - The first `elts_size` members of `elts` contain the integers comprising the set, from lowest to highest, with no duplicates.

# Representation invariant example

- We used these invariants to write the methods of each implementation.
- For example:

```
insert(int v)              // unsorted version
  if v not in elts         // don't allow duplicates
    elts[elts_size] = v// this breaks invariant
    ++elts_size            // this restores it

insert(int v)             // sorted version
  if v not in elts        // don't allow duplicates
    make gap in array // this breaks invariant
    elts[gap] = v         // restore elts invariant
    ++elts_size           // restore elts_size invar.
```

# Check the representation invariant

- We've seen representation invariants written in English:
  - `IntSetSorted`: The first `elts_size` members of `elts` contain the integers comprising the set, from lowest to highest, with no duplicates.
- We can also write representation invariants in code
- Let's add a private member function to `IntSetSorted` called `check_invariant()` that checks the representation invariant

# Exercise

```
class IntSetSorted : public IntSet {
  //...
private:
  //Represent a set of size N as an sorted set of
  //integers, with no duplicates, stored in the
  //first N slots of the array
  int elts[ELTS_CAPACITY];

  //Number of elements currently in the set
  int elts_size;

  //EFFECTS: returns true if rep. invariant holds
  bool check_invariant() const;
};
```

Write this function using a loop

# Using check_invariant()

- You can use check functions for defensive programming
- Add a check to the end of each member function
- Add a check to the beginning too, if you're really paranoid

- `assert()` is the perfect way to make these checks

# assert()

- A programmer's best friend
- `assert(STATEMENT);`
  - Make sure that `STATEMENT` is true
  - If not, crash the program with a debugging message

```
#include <cassert>
void IntSetSorted::insert(int v) {
  assert(check_invariant());
  //...
}
```

```
a.out: Interfaces_and_Invariants.cpp:146:
IntSetUnsorted::insert(): Assertion
`check_invariant()' failed.
Aborted (core dumped)
```

# assert() + check_invariant()

- Add a check to each member function
- Check at the end of the constructor

```
IntSetSorted::IntSetSorted()
  : elts_size(0) {
  assert(check_invariant());
}
```

- Check at the beginning and at the end of other functions

```
void IntSetSorted::insert(int v) {
  assert(check_invariant());
  //...
  assert(check_invariant());
}
```

# assert() + check_invariant()

- `const` member functions can't change any member variable, so we only need a check at the beginning

```
int IntSetSorted::size() const {
  assert(check_invariant());
  return elts_size;
}
```

# Check the Representation Invariant

- Let's a add a similar private member function called `check_invariant()` that checks the representation invariant for `IntSetUnsorted`

# Exercise

```
class IntSetUnsorted : public IntSet {
  //...
private:
  //Represent a set of size N as an unsorted set of
  //integers, with no duplicates, stored in the
  //first N slots of the array
  int elts[ELTS_CAPACITY];

  //Number of elements currently in the set
  int elts_size;

  //EFFECTS: returns true if rep. invariant holds
  bool check_invariant() const;
};
```

Write this function using nested loops

# Invariants and assert()

- The nested loop in `check_invariant()` might be slow
  - It's worth it during testing, but not in the final product
- Disable the check by compiling with the `NDEBUG` preprocessor variable defined
- There are two ways to do this:
  1. Add a line of code
     ```
     #define NDEBUG   //disable assert()
     ```
  2. Specify it on the command line of the compiler:
     ```
     g++ -DNDEBUG ...
     ```
- This way, you can turn it off for production code, but leave it in during development and testing

# Invariants and assert()

```
#define NDEBUG // disables assert statements!
void IntSet::insert(int v) {
  // ...
 assert(check_invariant());
}

void IntSet::remove(int v) {
  // ...
 assert(check_invariant());
}
```