ECE2800J RC 4

1. Abstract Data Types (ADTs)

An **Abstract Data Type (ADT)** provides an abstract description of values and operations. (We can leave off the details of **how**)

1.1 Class overview

In C++, a class gives:

- ***values*** → data members
- ***operations*** → methods/member functions

Below are the declarations that usually appear in a `.h` file:

```
const int MAX_ELTS = 100;

// OVERVIEW: a mutable set of integers
class IntSet {
private:
    int elts[MAX_ELTS];
    int numElts;
    int indexOf(int v);

public:
    IntSet();
        // default constructor
    void insert(int v);
        // MODIFIES: this
        // EFFECTS: this = this + {v}
    void remove(int v);
        // MODIFIES: this
        // EFFECTS: this = this - {v}
    bool query(int v);
        // EFFECTS: returns true if v in this
    int size();
        // EFFECTS: returns |this|
};
```

By default, every member of a class is ***private***. A private member is visible **only** to other members of this class. The ***public*** keyword is used to signify that some members are visible to anyone who sees the class definition. The ***protected*** keyword means the member can be seen by all members of this class and any derived classes (not used often, details will be introduced later).

Below are the definitions that usually appear in a `.cpp` file: (remember to include the `.h` file)

```
int IntSet::indexOf(int v) {
    for (int i = 0; i < numElts; i++) {
        if (elts[i] == v) return i;
    }
    return MAXELTS;
}

IntSet::IntSet() : elts{0}, numElts(0) {}

void IntSet::insert(int v) {
    if (indexOf(v) == MAXELTS) {
        if (numElts == MAXELTS) throw MAXELTS;
        elts[numElts++] = v;
    }
}

void IntSet::remove(int v) {
    int victim = indexOf(v);
    if (victim != MAXELTS) {
        elts[victim] = elts[numElts-1];
        numElts--;
    }
}

bool IntSet::query(int v) const {
    for (int i = 0; i < numElts; ++i) {
        if (elts[i] == v) return true;
    }
    return false;
}

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

1.2 Constructor

```
IntSet();
```

A constructor:

- has the **same name** as the class
- has **no return type** or **argument**
- is called automatically when an object is created (and is guaranteed the first)

Initialization Syntax:

```
IntSet::IntSet() : elts{0}, numElts(0) {
    // optional body
}
```

Initialization happens in the **order of declaration** in the class,
not in the order written in the initialization list. It is a good practice to keep them in the same order to avoid confusion.

1.3 `const` Qualifier

Each member function of a class has an extra, implicit parameter named ***this***. `const` keyword modifies the implicit *this* pointer: *this* is now a pointer to a const instance.

`const` + member function

```
bool query(int v) const;
int size() const;
```

A `const` member function:

- treats `this` as a pointer to a constant object
- cannot modify any data members
- **can only call other `const` member functions**

```
void A::g() const { f(); }
```

```
void A::f() {...} //error
void A::f() const {...} // correct
```

`const` + data member

A `const` data member must be initialized in the **class declaration** or the **initialization syntax**.

```
class Foo {
    const int n;
public:
    Foo();
};

Foo::Foo() {n = 8;}     // error
Foo::Foo() : n(8) {}    // correct
```

1.4 Function Efficiency

Time complexity using big-O notation:

- `O(1)` means the time is constant .
- `O(n)` means the time linearly increases with the size of the input.
- etc.

2. Subtypes

Notation: **S <: T** means S is a ***subtype*** of T, while T is a ***supertype*** of S. (e.g. dog <: animal>)

Subtype can be used wherever its supertype is expected.

Ways to create a subtype:

- *add operations*
- *strengthen postconditions*
- *weaken preconditions*

```
class MaxIntSet : public IntSet {
    // OVERVIEW: a set of integers,
    // where |set| <= 100
public:
    int max();
```

```
        // REQUIRES: set is non-empty.
        // EFFECTS: returns largest element in set.
};
```

2.1 Inheritance

```
class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};
```

Public inheritance

All public members of the base are also public in the derived class; all private members of the base are also private in the derived class.

```
class B : public A {
    // x public
    // y protected
    // z private
};
```

Protected inheritance

All public and protected members of the base class are protected in the derived class.

```
class C : protected A {
    // x protected
    // y protected
    // z private
};
```

If a member is ***protected***, it means "can be only seen by all members of this class and any derived classes". Protected data members make derived classes extremely fragile, and it is a matter of taste as to whether it's worth doing.

Private inheritance

All members of the base class are private in the derived class.

```
class D : private A {
    // x private
    // y private
    // z private
};
```

A derived class is **not necessarily** a subtype of the base class.

2.2 Substitution Principle

1. The new method must do everything the old method did; it is allowed to do more as well.
2. It must require no more of the caller than the old method did, but it can require less.

```
class SafeMaxIntSet : public MaxIntSet {
    // OVERVIEW: a mutable set of integers,
    // where |set| <= 100
public:
    int max();
        // EFFECTS: if set is non-empty, returns largest element in set.
        // otherwise, returns INT_MIN.
};
```

In this method, we've both weakened the preconditions AND strengthened the postconditions of "old" max.

So, if we declared one object of each type, calling the max method would give us the "right" one:

```
MaxIntSet ms;
SafeMaxIntSet ss;
ss.max();   // calls SafeMaxIntSet::max()
            // returns INT_MIN on empty set
ms.max();   // calls MaxIntSet::max()
            // is undefined on empty set
```

3. Virtual Functions

C++ dispatches member functions based on:

- ***apparent type*** → the declared type of the reference
- ***actual type*** → the real type of the referent

```
class PosIntSet : public IntSet {
    // OVERVIEW: a mutable set of positive integers
public:
    void insert(int v);
        // EFFECTS: if v is positive
        // and s has room to include it,
        // s = s + {v}.
        // if v <= 0, throw int -1
        // if s is full, thrown int MAXELTS
};
```

`PosIntSet` is not a subtype! Because code that is correctly written to use an `IntSet` could fail when using a `PosIntSet`.

```
void PosIntSet::insert(int v) {
    if (v <= 0) throw -1; // doesn't pass the substitution principle
    IntSet::insert(v);
}
```

When we call the following:

```
PosIntSet s;    // actual type: PosIntSet
IntSet* p = &s; // apparent type: IntSet
IntSet& r = s;  // apparent type: IntSet
```

In default situation, C++ chooses the method to run based on its **apparent type**, which is not we wish to run.

So we introduce ***virtual***, a way to tell C++ to choose the **actual type**.

```
class IntSet {
    ...
public:
    ...
```

```
    virtual void insert(int v);
    ...
};
```

This tells the compiler "*someone might override my implementation: always check at run-time to see which version to call.*"

4. Interfaces

To the caller, an ADT is only an ***interface*** (the contract for using things of this type).

How to provide a class definition that carries no implementation details (i.e., data members) to the client programmer, yet still has interface information?

4.1 Abstract Base Class

Abstract base class/ Virtual base class: an "interface-only" class, from which an implementation can be derived. It doesn't have any real implementation!

```
class IntSet {
public:
    virtual void insert(int v) = 0;
    virtual void remove(int v) = 0;
    virtual bool query(int v) = 0;
    virtual int size() = 0;
};
```

These functions are called ***pure virtual functions*** and are declared not to exist.

A class with one or more Pure Virtual Functions is an ***abstract class***. You cannot create any instances of an abstract class, because there are no implementation.

So how to give it an implementation?

```
class IntSetImpl : public IntSet {
private:
    int elts[MAX_ELTS];
    int numElts;

public:
    IntSetImpl();           // used to be in the base class
    void insert(int v);
```

```
    void remove(int v);
    bool query(int v);
    int size();
};

static IntSetImpl intSetImpl;

IntSet* getIntSet() {
    return &intSetImpl;
}
```

- The interface (the abstract base class) is typically defined in a public header (.h) file.
- The implementation (the derived class) is defined in a source (.cpp) file

```
// header file
IntSet *getIntSet();
    // EFFECTS: returns a pointer to the IntSet

// source file
static IntSetImpl impl;
IntSet *getIntSet() {
    return & impl;
}

// it's valid now!
IntSet *s = getIntSet();
```

The .cpp file defines a single, static instance (only visible to the .cpp file) of the implementation and body of the access function.

Exercise: what are the outputs?

```
class A {
public:
    void f() { std::cout << "A::f()\n"; }
    virtual void g() = 0;
    virtual void h() { std::cout << "A::h()\n"; }
};

class B : public A {
public:
```

```
    virtual void f() { std::cout << "B::f()\n"; }
    void g() { std::cout << "B::g()\n"; }
    void h() { std::cout << "B::h()\n"; }
};

class C : public B {
public:
    void f() { std::cout << "C::f()" << std::endl; }
    void h() { A::h(); }
};

class D : public C {
public:
    virtual void g() { std::cout << "D::g()" << std::endl; }
    void h() { std::cout << "D::h()" << std::endl; }
};
```

Note: Virtualness is inherited; only the **base class** must declare `virtual`.

`D d;`	`A* apd = &d;`	`B* bpd = &d;`	`C c;`	`A & arc = c;`
`d.f();`	`apd->f();`	`bpd->f();`	`c.f();`	`arc.f();`
`d.g();`	`apd->g();`	`bpd->g();`	`c.g();`	`arc.g();`
`d.h();`	`apd->h();`	`bpd->h();`	`c.h();`	`arc.h();`

5. Invariants

A ***representation invariant (rep invariant)*** describes the conditions that must hold on those members for the representation to correctly implement the abstraction.

Rep invariant:

- must hold after constructor and after each method
- methods may assume the invariant holds before being called
- use a private checker for defensive programming

```
bool strictSorted(int a[], int size) {
    // REQUIRES: a has size elements
    // EFFECTS: returns true if a is sorted with no duplicates
    if (size <= 1) return true;
```

```
    for (i=0; i<size-1; i++){
        if (a[i] >= a[i+1]) {
            return false;
        }
    }
    return true;
}

// following is a private checker
bool repOK() {
    // EFFECTS: returns true if the rep. invariants hold
    return strictSorted(elts, numElts);
}

// add this line right before returning from any function that modifies any
of the representation
assert(repOK());
```

Answers

Variable	d	apd	bpd	c	arc
`f()`	`C::f()`	`A::f()`	`C::f()`	`C::f()`	`A::f()`
`g()`	`D::g()`	`D::g()`	`D::g()`	`B::g()`	`B::g()`
`h()`	`D::h()`	`D::h()`	`D::h()`	`A::h()`	`A::h()`

References

- cppreference
- Lecture slides 13–15
- Mid RC Part 4 ECE2800J25FA