ECE2800J Final RC Part 1

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

Members in a class can be devided into:

- ***private***: visible **only** to other members of this class; default member type
- ***public***: visible to anyone who sees the class definition
- ***protected***: can be seen by all members of this class and any derived classes; makes the members fragile

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

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.

1.3 `const` Qualifier

Each member function of a class has an extra, implicit parameter named ***this***.

```
class A {
public:
    void foo();
};
// when we write:
void foo() {
    // ...
}
// we are actually writing:
void foo(A* this) {
    // ...
}
```

`const` keyword modifies the implicit *this* pointer, and makes *this* now a pointer to a const instance.

```
class A {
public:
    void foo() const;
};
// now we are actually writing:
void foo(const A* this) {
    // ...
}
```

`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**

```
// if we want:
void A::g() const { f(); }
// then we need:
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**.

```
// initialization in the class declaration:
class Foo {
    const int n = 8;
public:
    Foo();
};
// initialization n in the constructor:
class Foo {
    const int n;
public:
    Foo();
};
Foo::Foo() {n = 8;}     // error
Foo::Foo() : n(8) {}    // correct
```

1.4 Function Efficiency

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

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:

1. *add operations* (subtype can do more!)

```
  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. *strengthen postconditions* (subtype can do better!)

```
class SortedIntSet : public IntSet {
public:
    void insert(int v) override;
    // EFFECTS: after insertion, the elements are sorted in ascending order
};
```

3. *weaken preconditions* (subtype requires less!)

```
class SafeRemoveIntSet : public IntSet {
public:
    void remove(int v) override;
    // REQUIRES: nothing (even if v not in the set)
    // EFFECTS: if v in this, removes it
    // if v NOT in this, does nothing safely (no error)
};
```

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

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

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.

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;
    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

```
class IntSet {  // abstract class & abstract base class
public:
    // four pure virtual functions:
    virtual void insert(int v) = 0;
    virtual void remove(int v) = 0;
    virtual bool query(int v) = 0;
    virtual int size() = 0;
};
```

- ***Pure virtual functions***: declared not to exist.
- ***Abstract base class/ Virtual base class***: an "interface-only" class, doesn't have any real implementation.
- ***Abstract class***: a class with one or more pure virtual functions.
- 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

```
// suppose in header file:
IntSet *getIntSet();

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

// then we can get a pointer to an IntSet without knowing its actual type:
IntSet *s = getIntSet();
```

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

```
// if we define a sortedIntSet, to hold its rep invariant, we create a
private checker:
bool repOK() {
    // EFFECTS: returns true if the rep. invariants hold
    return strictSorted(elts, numElts);
}
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;
}
// 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

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