Abstract Data types

Ivor Page\textsuperscript{1}

2.1 Introduction

An abstract data type (ADT) is a class specification that enables objects to be created at run time that (1) hold data that is hidden from the users, and (2) provide interface functions that simplify and control access to the stored data. We use the term encapsulation to describe the first concept, and abstraction to describe the second. In Object Oriented Languages, inheritance and polymorphism are also provided.

\textsuperscript{1}University of Texas at Dallas
Some object oriented languages also facilitate generic types. These enable versions of the same ADT to be created at run time that hold differing data types. For example, a generic stack class can be used to create a stack ADT that holds integers, another that holds doubles, another that holds user defined objects, and so on.

In this section we shall explore the implementation of these ideas in C++ and Java.

2.2 Basic Class Definitions in C++

A class definition for an ADT usually contains declarations of the variables to be held, one or more constructor functions, and the interface functions that provide the abstraction. In addition, a destructor function is usually required if the class variables include one or more pointers. A copy constructor and a function that overloads the assignment operator are also needed in some cases.
The copy constructor is called when a new object is created at run time and the values of an existing object are to be copied into it. This happens during a declaration of the form:

```
RationalNumber y(2,3); // calls constructor
```

```
RationalNumber x(y); // calls copy constructor
```

and when an object is passed by value into a function.

If the class does not contain a definition of a copy constructor, the default copy constructor is used. It just copies the values of the variables in the object. If the class contains pointers, it is likely that a user-defined copy constructor will be needed.
The overload of the assignment operator is called when one user defined object is assigned to another of the same type, as in:

```c++
RationalNumber x(1,2), y; // calls constructor twice
...
y = x; // calls overload of =
```

If a user-defined overload of the assignment operator is not provided, a default overload function is used. It simply copies the values from the object on the right to the object on the left. As with the copy constructor, classes containing pointers will require an overload of the assignment operator if their objects are to be assigned to one another.
Here is a class interface for a simple linked list in C++:

class LinkedList {
public:
    LinkedList(); // constructor, creates an empty list
    void addToHead(myObject &x);
    myObject& removeHead();
    bool is Empty();
private:
    myObject* head;
};
Without adding a copy constructor and an overload of the assignment operator, the following statements would cause a *shallow copy* to be made in which just the pointer value is copied:

```c
myObject x, y(3); // calls the constructor
myObject z(y);    // calls default copy constructor
                  // now z and y point to the same elements
x = y;            // calls default overload of =
                  // now x and y point to the same elements
z = reverse(x);   // just passes the head pointer
```

Usually an assignment is expected to make a complete copy of all the elements in an object. This is called a *deep copy*. We also expect a function call to behave as if a deep copy of the arguments is made. In the case of the `reverse` function above, the object `x` will almost certainly be mutated by the function. For these statements to behave as normally expected, both a copy constructor and an overload of the assignment operator must be provided.
Here is a class interface for a rational number in C++:

```cpp
class Rational {
public:
    explicit Rational(int n=0, int d=1); // constructor w/ default args
    const Rational(const Rational&); // copy constructor - not needed
    int getNumerator() const;
    int getDenominator() const;
    double toDouble() const;
    const Rational& operator=(const Rational&); // overload of assignment
protected:
    int numerator, denominator; // the class variables
};
```
The member function bodies are typically defined in a separate file. Here are some examples:

```cpp
Rational::Rational(int n, int d)
    : numerator(n), denominator(d) {}

Rational::Rational(const Rational& x)
    numerator = x.numerator;
    denominator = x.denominator;

const Rational& Rational::operator=(const Rational &x) {
    if(this!=&x) { // standard alias test
        numerator = x.numerator;
        denominator = x.denominator;
    }
    return *this;
}

int Rational::getNumerator() const {
    return numerator;
}
```
The default copy constructor and default assignment operator would work just fine with this class.

You should understand why some arguments are passed by reference and why the result of the assignment overload is passed by reference. You should also understand the use of \texttt{explicit} and \texttt{const} in the above.

The class isn’t very useful at present. It defines a numeric type intended to represent a rational number, the ratio of two integers. There isn’t much that can be done with the class unless we add arithmetic functions to the interface.

Inheritance enables us to declare a superclass of Rational that has additional functionality. In fact we can declare several versions of Rational, each with slightly different functionality.
Here is the interface of one derived class with functions to provide arithmetic operations:

class RationalNumber: public Rational { // inherit the Rational class
public:
    RationalNumber(int n=0, int d=0); // constructor
    RationalNumber add(const RationalNumber&) const;
    RationalNumber subt(const RationalNumber&) const;
    RationalNumber mult(const RationalNumber&) const;
    RationalNumber div(const RationalNumber&) const;
    void print() {
        cout << endl << numerator << "": " << denominator;
    }
};
Now we can declare and do arithmetic on RationalNumbers:

void main()
{
    RationalNumber a(2,3), b(4,5), c,d,e,f;
    c = a.add(b);
    d = a.subt(b);
    e = a.mult(b);
    f = a.div(b);
    ...
}

In C++ the arithmetic operators can be overloaded, but not in Java.
Here is the class RationalNumber with member function versions of the operator overloads:

class RationalNumber: public Rational { // inherit the Rational class
public:
    RationalNumber(int n=0, int d=0); // constructor
    RationalNumber operator+(const RationalNumber&) const;
    RationalNumber operator-(const RationalNumber&) const;
    RationalNumber operator*(const RationalNumber&) const;
    RationalNumber operator/(const RationalNumber&) const;
    void print() {
        cout << endl << numerator << " : " << denominator;
    }
};
Now main can be rewritten as follows:

```cpp
void main()
{
    RationalNumber a(2,3), b(4,5), c,d,e,f;
    c = a+b;
    d = a-b;
    e = a*b;
    f = a/b;
    ...
}
```

Comparison operators such as `==`, `<`, `<=`, `>`, `>=`, can also be over-loaded in C++.
2.3 Pointers and Virtual Functions

When user defined objects are referenced by pointers, additional rules apply. This is particularly true when inheritance is used. For example, if we create a new RationalNumber on the heap at runtime as follows:

```cpp
RationalNumber *px;
px = new RationalNumber(2,3);
```

then its functions and those of the base class, Rational, are available via the pointer:

```cpp
int z = px->getNumerator();
px->print();
```
In abstract data types that contain non-homogeneous types, we declare the pointers to refer to the base class. For example, we might have a queue of graphical objects including spheres, cylinders, polyhedra, cubes, etc. Each of these might have an identical set of function signatures. Each would inherit a base class, say GraphicalObject. Polymorphism should enable us to call those functions via the pointers. We need an additional trick to make this work. Here is an example:

```cpp
Rational *py;  // pointer to the base class

py = new RationalNumber(6,8);  // make a derived class object

int iq = py->getNumerator();
py->print();
```

We can still access the member functions of the base class via the pointer, `py->getNumerator()`, is still OK, but we cannot access member functions of the derived class, `py->print()` is not allowed.
To make the functions of the derived class available via pointers to a base class, those function prototypes must also be declared within the base class with the qualifier `virtual`.

The base class would include the prototype:

```
virtual void print() = 0;
```

We set the body to zero when the function has no value in the base class. Doing so makes the function and the base class _abstract_, which means that you cannot declare objects of type `Rational`.

Since you may want to store your derived classes in ADTs that use pointers, it’s essential to include virtual functions in the base class for all derived class functions.
2.4 A Little OOD

When designing a class hierarchy, it is usually best to flesh out all the lowest-level derived classes that you will need. For example, in the case of the hierarchy of graphical objects, each lowest-level object may need member functions to draw itself, to intersect itself with a given ray, to detect if it is on the visible side of a given plane, etc. Once all these interfaces have been designed, similarities between the lowest-level classes are considered. The Sphere, Cylinder and Cone classes will have identical functions for ray-intersection. The Polyhedra class probably doesn’t inherit the Plane class, but contains multiple Plane objects. Plane probably inherits Polygon, and so on. The hierarchy is naturally built from the bottom up. Than all of the functions in the lowest-level classes are declared virtual in the base class.
2.5 Templates

Templated classes facilitate generic ADTs. Consider the templated stack class:

```cpp
template <class T>
class Stack {
public:
    Stack(); // constructor
    ~Stack(); // destructor
    bool isEmpty() const;
    const T & top() const; // peek top element
    void push(const T &x);
    T topAndPop(); // remove and return top element
    const Stack & operator=(const Stack & x);
private:
    T *topOfStack; // pointer to first element
};
```
This is a bare-bones version. See page 94 of the text for the full version. We can now declare stacks to hold any types:

```c
Stack<double> doubleStack();
Stack<char*> stringStack();
Stack<RationalNumber> rnStack();
```

Although templates provide for generic ADTs, you should note that the added syntax can be cumbersome and its main advantage is in saving you from typing a separate class definition for each object type to be represented.
2.6 Arguments of the ADT

There are two ways in which an object may be passed into and out of an ADT: by passing the object itself, or by passing a pointer to the object. Say, for example, that we wanted a stack of the user defined objects, PlayingCard. The declaration could be:

\[
\text{Stack<PlayingCard> deck(); OR Stack<PlayingCard*> deck();}
\]

In the first version, copies are made of the playing cards as they are passed into and out of the ADT and the actual elements stored within the ADT are playing cards. This kind of interface would be inefficient if large objects had to be copied. It is, however, simpler to program, and less prone to errors. If the pointer interface is used, the playing cards “held within” the ADT are not protected from mutation by functions outside the ADT. The pointer interface is, however, very convenient and fast, so it is used extensively.
2.7 Java ADTs

In Java there are no destructors, no copy constructors, no operator overloads, and no pointers. These differences, and others, tend to make Java a simpler language to use. Problems of memory leakage that are common to C++ programs, and often extremely difficult to find, cannot exist in Java. The downsides are that Java programs run slower than their C++ equivalents, and the run time of any function is less predictable than in C++ because of behind-the-scenes garbage collection. However, Java is gaining more acceptance in industry, probably because of its simplicity and the huge array of built-in packages, including windows-style GUIs, 2D and 3D graphics, networking, security (encryption), and database access. Most of this functionality is not built-into C++.
Unlike C++, all functionality in Java programs must be contained within classes. Strict type checking is enforced, forcing the programmer to apply casts to all but the simplest objects. Implicit conversion, or promotion, takes place amongst integer types, byte, short, char, int, long, float, double, but only left to right within this list. Constructors are by-definition explicit. These rules create a low-inference, no-surprise programmer interface.

Local class variables and member functions are public unless declared otherwise.

Functions declared within the main public class (the same name as the filename) must be declared static since they will be called from the body of that class, which is, by definition a static context.