CIS 343 - Structure of Programming Languages

Nathan Bowman

Based on slides by: Ira Woodring

Abstract Data Types (Follows the Sebesta Text Chapter 11)

Most modern programming languages support Object-Oriented (OO) programming.

One of the fundamental building blocks of OOP is abstract data types.

Abstraction is a way of looking at some entity that only focuses on the most significant parts.

We discuss two types of abstraction in this class,

Process abstraction occurs when we give a name to a sequence of statements. This allows us to call upon a sequence of statements instead of manually executing each statement ourselves. We see this with subprograms (functions). This was the first type of abstraction in Computer Science, and even Plankalkül supported it in the 1940s.

Data abstraction is much newer, with the ideas for it beginning in the 1960s and COBOL. C also provided data abstraction with the concept of structs. This allows us to create composite types.

Abstract Data Types (ADTs) are enclosures that define a specific type of data *and* the subprograms that operate on that data type. For instance, we may define an ADT called Spaceship that includes spaceship information, and also the functions that we would call to operate on the spaceship data.

Abstract Data Types (ADTs)

ADTs have access controls that can hide information from outside the ADT instances.

This is very important! It means that the user doesn't have to know anything about the underlying storage or function of the ADT. All the users need to know is how to use it.

Abstract Data Types (ADTs)

Contrast this with `struct` in C

`structs` can hold a variety of data, but the programmer must know exactly what is inside the struct to use it

There is no abstraction of storage details and no guarantees that the user will not put the struct into a bad state (enter a negative number for `length`, for example)

Abstract Data Types (ADTs)

In its purest form, an ADT would be accessed *only* through methods

Users would have no idea what was stored in the ADT

May be helpful to think of an ADT as a "`struct` with functions," but in this sense it is exactly the opposite of a `struct`

Abstract Data Types (ADTs)

Through the rest of this semester we will use the term ADT to refer to the construct or definition of an ADT. We will use the term **object** to refer to a particular instance of an ADT.

Abstract Data Types (ADTs)

Why ADTs?

Like all abstractions, we hope they will make programs less complex.

Also allow us to separate code into areas of concern. Subprograms (called "methods" in an ADT) operate only on data relevant data and are scoped accordingly

Instead of concentrating on the big picture of what the program is trying to accomplish, we focus on writing robust definitions of the smaller units of which the program is composed.

Abstract Data Types (ADTs)

An important point: **all** data types can be ADTs. We tend to forget this when we think of built-in types, but even built-in types are ADTs.

Consider a floating point number. We don't have to understand how it is stored; in fact there have been multiple different ways to store a floating point number that systems have used in the past. All we really care about is how to use one.

Even the concept of the byte is an ADT; in reality it is just the turning on or off of some switch. We don't care how that happens - just that we can use one.

Abstract Data Types (ADTs)

This brings up an important point: **information hiding** is the concept that we can use an ADT without knowing how it works internally.

This implies:

- that ADTs can be very robust. By hiding the information of implementation we prohibit users from "damaging" our internal data.
- that our code can still work the same regardless of the system on which we are running. For instance, if we run on a system that doesn't follow the same standard for the storage of floating point numbers, we can still use floating point numbers. How they are stored on the systems is not important to us.

We will use the term **clients** to refer to a program unit that uses a specific ADT.

Abstract Data Types (ADTs)

Other benefits of information hiding (besides increased integrity of the data) are

- the amount of variables and code the programmer has to be aware of is reduced.
- name conflicts are less likely (scope of the variables are smaller).
- changes in how the ADT was programmed do not cause us to have to rewrite all code that uses the ADT

Abstract Data Types (ADTs)

We need to be careful about designing our ADTs. If we aren't careful about adhering to proper practice we run the risk of losing the benefits of information hiding.

Therefore we should (in general) not set access to our data as public.

We should always use accessors.

Abstract Data Types (ADTs)

The reasons accessors are better are

- we can have read-only data by having a getter but no setter.
- we can include constraints in our setters. For example we might want to limit possible changes to our state to certain value ranges.
- (as already mentioned) changing the way the ADT is implemented does not affect clients.

Abstract Data Types (ADTs)

For a language to be able to provide ADTs it must

- provide a syntactic unit to enclose the declaration of the type and prototypes of the subprograms that implement the operations.
- a means to make subprograms visible to clients so they can ask the ADT to manipulate data on its behalf.
- must provide external visibility for the name, but hide the representation

Abstract Data Types (ADTs)

In addition to the syntactic unit issues, language designers must decide if an ADT can be parameterized. Parameterizing an ADT means specializing it so that can hold specific data types.

We have seen parameterized types in Java; for instance when defining an ArrayList&lt>. ArrayLists&lt> require us to denote the type of ArrayList we wish to use by filling in the "&lt>" section of the declaration. This is important for generic programming.

Abstract Data Types (ADTs)

In C++ for instance (note C++ does have a `std::stack`. This is NOT that structure.):

```
template <typename T>
class Stack {

  public:
    void push(T const & e);
    void pop();
    T top() const;

  private:
    std::vector<T> elements;
};
```

Abstract Data Types (ADTs)

We would then be able to create an object of this type by parameterizing our declaration:

```
Stack<int> intStack;
Stack<float> floatStack;
Stack<Concert> concertStack;
```

Abstract Data Types (ADTs)

Other issues are the types of access control provided and how those access controls are specified, and whether the ADT must be specified with its interface and implementation together or separately.

Abstract Data Types (ADTs)

Once a program is large enough, we will often need a further abstraction in addition to ADTs -- encapsulation

Encapsulation is grouping related subprograms and ADTs for purposes of compilation or scoping

Abstract Data Types (ADTs)

You are familiar with one example of encapsulation in the form of Java packages

For example, Java has its own `Scanner` class, but perhaps you are writing your own code and it makes sense to have a completely unrelated `Scanner`

This is no problem, because Java provides `Scanner` inside the `java.util` package

Until you `import java.util.Scanner`, you can have access to Java's scanner without it polluting your

namespace