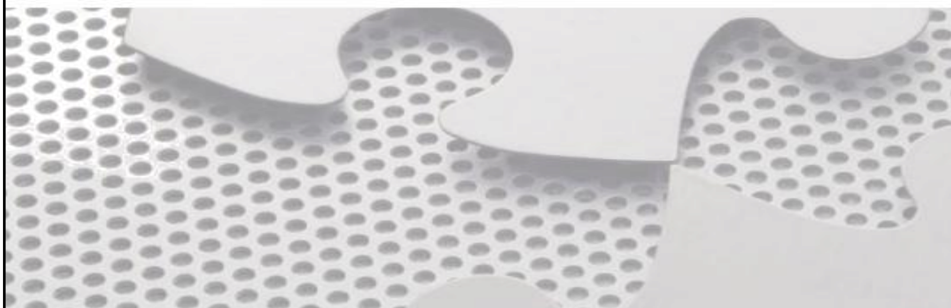
Programming Languages
Third Edition



Chapter 11

Abstract Data Types and Modules

Objectives

- Understand the algebraic specification of abstract data types
- Be familiar with abstract data type mechanisms and modules
- Understand separate compilation in C, C++ namespaces, and Java packages
- Be familiar with Ada packages
- Be familiar with modules in ML

Objectives (cont'd.)

- Learn about modules in earlier languages
- Understand problems with abstract data type mechanisms
- Be familiar with the mathematics of abstract data types

Introduction

- Data type: a set of values, along with certain operations on those values
- Two kinds of data types: predefined and user-defined
- Predefined data types:
  - Insulate the user from the implementation, which is machine dependent
  - Manipulated by a set of predefined operations
  - Use is completely specified by predetermined semantics

Introduction (cont'd.)

- User-defined data types:
  - Built from data structures using language's built-in data types and type constructors
  - Internal organization is visible to the user
  - No predefined operations
- Would be desirable to have a mechanism for constructing data types with as many characteristics of a built-in type as possible
- **Abstract data type** (or **ADT**): a data type for constructing user-defined data types

Introduction (cont'd.)

- Important design goals for data types include modifiability, reusability, and security
- **Encapsulation**:
  - Collection of all definitions related to a data type in one location
  - Restriction on the use of the type to the operations defined at that location
- **Information hiding**: separation and suppression of implementation details from the data type's definition

Introduction (cont'd.)

- There is sometimes confusion between a **mechanism** for constructing types and the **mathematical concept** of a type
- Mathematical models are often given in terms of an **algebraic specification**
- **Object-oriented programming** emphasizes the concept of entities to control their own use during execution
- Abstract data types do not provide the level of active control that represents true object-oriented programming

Introduction (cont'd.)

- The notion of an abstract data type is independent of the language paradigm used to implement it
- **Module**: a collection of services that may or may not include data type(s)

The Algebraic Specification of Abstract Data Types

- **Complex** data type: an abstract data type which is not a built-in type in most languages
  - Used to represent a complex number of the form $x = iy$ where i represents the complex number $\sqrt{-1}$
  - Must be able to create a complex number from a real and imaginary part, plus functions to extract the real and imaginary parts
- **Syntactic specification**: name of the type and names of the operations, including a specification of their parameters and returned values
  - Also called the **signature** of the type

The Algebraic Specification of Abstract Data Types (cont'd.)

- Function notation is used to specify the operations of the data type *f:X→Y*
- Signature for complex data type:

type complex **imports** real

operations:

+: complex × complex → complex
-: complex × complex → complex
*: complex × complex → complex
/: complex × complex → complex
-: complex → complex
makecomplex: real × real → complex
realpart: complex → real
imaginarypart: complex → real

The Algebraic Specification of Abstract Data Types (cont'd.)

- This specification lacks any notion of semantics, or the properties that the operations must actually possess
- In mathematics, semantic properties of functions are often described by **equations** or **axioms**
  - Examples of axioms: associativity, commutative, and distributive laws
- Axioms can be used to define semantic properties of complex numbers, or the properties can be **derived** from those of the real data type

The Algebraic Specification of Abstract Data Types (cont'd.)

- Example: complex addition can be based on real addition

$$\text{realpart}(x + y) = \text{realpart}(x) + \text{realpart}(y)$$
$$\text{imaginarypart}(x + y) = \text{imaginarypart}(x) + \text{imaginarypart}(y)$$

  - This allows us to prove arithmetic properties of complex numbers using the corresponding properties of reals
- A complete algebraic specification of type complex combines signature, variables, and equational axioms
  - Called the **algebraic specification**

type complex **imports** real

operations:

$+$: complex $\times$ complex $\to$ complex
$=$: complex $\times$ complex $\to$ complex
$*$: complex $\times$ complex $\to$ complex
$/$: complex $\times$ complex $\to$ complex
$-$: complex $\to$ complex
makecomplex : real $\times$ real $\to$ complex
realpart : complex $\to$ real
imaginarypart : complex $\to$ real

variables: x,y,z: complex; r,s: real

axioms:

realpart(makecomplex(r,s)) = r
imaginarypart(makecomplex(r,s)) = s
realpart($x + y$) = realpart(x) + realpart(y)
imaginarypart($x + y$) = imaginarypart(x) + imaginarypart(y)
realpart($x - y$) = realpart(x) - realpart(y)
imaginarypart($x - y$) = imaginarypart(x) - imaginarypart(y)
. . .
(more axioms)
. . .

The Algebraic Specification of Abstract Data Types (cont'd.)

- The equational semantics give a clear indication of implementation behavior
- Finding an appropriate set of equations, however, can be difficult
- Note that the **arrow** in the syntactic specification separates a function's domain and range, while **equality** is of values returned by functions
- A specification can be **parameterized** with an unspecified data type

The Algebraic Specification of Abstract Data Types (cont'd.)

type queue(element) **imports** boolean

operations:

createq: queue
enqueue: queue × element → queue
dequeue: queue → queue
frontq: queue → element
emptyq: queue → boolean

variables: q: queue; x: element

axioms:

emptyq(createq) = true
emptyq(enqueue(q,x)) = false
frontq(createq) = error
frontq(enqueue(q,x)) = if emptyq(q) then x else frontq(q)
dequeue(createq) = error
dequeue(enqueue(q,x)) = if emptyq(q) then q else enqueue(dequeue(q),x)

The Algebraic Specification of Abstract Data Types (cont'd.)

- `createq`: a constant
  - Could be viewed as a function of no parameters that always returns the same value – that of a new queue that has been initialized to empty
- **Error axioms**: axioms that specify error values
  - Provide limitations on the operations
  - Example: `frontq(createq) = error`
- Note that the dequeue operation does not return the front element; it simply throws it away

The Algebraic Specification of Abstract Data Types (cont'd.)

- Equations specifying the semantics of the operations can be used as a specification of the properties of an implementation
- There is no mention of memory or of assignment
  - These specifications are in purely functional form
- In practice, abstract data type implementations often replace the functional behavior with an equivalent imperative one
- Finding an appropriate axiom set for an algebraic specification can be difficult

The Algebraic Specification of Abstract Data Types (cont'd.)

- Can make some judgments about the kind and number of axioms needed by looking at the syntax of the operations
- **Constructor**: an operation that creates a new object of the data type
- **Inspector**: an operation that retrieves previously constructed values
  - **Predicates**: return Boolean values
  - **Selectors**: return non-Boolean values
- In general, we need one axiom for each combination of an inspector with a constructor

The Algebraic Specification of Abstract Data Types (cont'd.)

- Example:
  - The queue's axiom combinations are:

    emptyq(createq)
    emptyq(enqueue(q,x))
    frontq(createq)
    frontq(enqueue(q,x))
    dequeue(createq)
    dequeue(enqueue(q,x))
  - Indicates that six rules are needed

Abstract Data Type Mechanisms

- A mechanism for expressing abstract data types must have a way of separating the signature of the ADT from its implementation
  - Must guarantee that any code outside the ADT definition cannot use details of the implementation and must operate on a value of the defined type only through the provided operations
- ML has a special ADT mechanism called `abstype`

Abstract Data Type Mechanisms (cont'd.)

```
(1) abstype 'element Queue = Q of 'element list
(2) with
(3)  val createq = Q [];
(4)  fun enqueue (Q lis, elem) = Q (lis @ [elem]);
(5)  fun dequeue (Q lis) = Q (tl lis);
(6)  fun frontq (Q lis) = hd lis;
(7)  fun emptyq (Q []) = true | emptyq (Q (h::t)) = false;
(8) end;
```

Figure 11.1 A queue ADT as an ML `abstype`, implemented as an ordinary ML list

Abstract Data Type Mechanisms (cont'd.)

- ML translator responds with a description of the signature of the type:

```
type 'a Queue
val createq = - : 'a Queue
val enqueue = fn : 'a Queue * 'a -> 'a Queue
val dequeue = fn : 'a Queue -> 'a Queue
val frontq = fn : 'a Queue -> 'a
val emptyq = fn : 'a Queue -> bool
```

- Since ML has parametric polymorphism, the `Queue` type can be parameterized by the type of the element to be stored in the queue

Abstract Data Type Mechanisms (cont'd.)

```
abstype Complex = C of real * real
with
 fun makecomplex (x,y) = C (x,y);
 fun realpart (C (r,i)) = r;
 fun imaginarypart (C (r,i)) = i;
 fun +: ( C (r1,i1), C (r2,i2) ) = C (r1+r2, i1+i2);
 infix 6 +: ;
  (* other operations *)
end;
```

Figure 11.2 A complex number ADT as an ML `abstype`

Abstract Data Type Mechanisms (cont'd.)

- ML allows user-defined operators, called **infix functions**
  - Can use special symbols
  - Cannot reuse the standard operator symbols
- Example: we have defined the addition operator on complex number to have the name `+:` as an infix operator with a precedence level of 6 (same as built-in additive operators)

Abstract Data Type Mechanisms (cont'd.)

- The `Complex` type can be used as follows:

```
- val z = makecomplex (1.0,2.0);
val z = - : Complex
- val w = makecomplex (2.0,~1.0); (* ~ is negation *)
val w = - : Complex
- val x = z +: w;
val x = - : Complex
- realpart x;
val it = 3.0 : real
- imaginarypart x;
val it = 1.0 : real
```


Modules

- A pure ADT mechanism does not address the entire range of situations where an ADT-like abstraction mechanism is useful in a language
- It makes sense to encapsulate the definitions and implementations of a set of standard functions that are closely related and hide the implementation details
  - Such a package is not associated directly with a data type and does not fit the format of an ADT mechanism

Modules (cont'd.)

- Example: a complier is a set of separate pieces

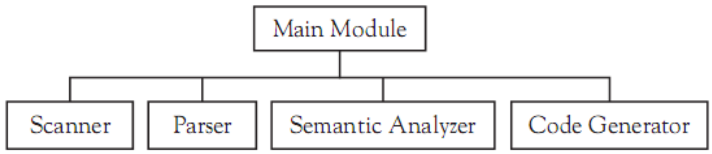


Figure 11.3 Parts of a programming language compiler

- **Module**: a program unit with a public interface and a private implementation
- As a provider of services, modules can export any mix of data types, procedures, variables, and constants

Modules (cont'd.)

- Modules assist in the control of **name proliferation**
  - They usually provide additional scope features
- A module exports only names that its interface requires, keeping hidden all others
- Names are **qualified** by the module name to avoid accidental name clashes
  - Typically done by using the dot notation
- A module can document dependencies on other modules by requiring explicit import lists whenever code from other modules is used

Separate Compilation in C and C++

- C does not have any module mechanisms
  - Has separate compilation and name control features that can be used to simulate modules
- Typical organization of a queue data structure in C:
  - Type and function specifications in a **header file** `queue.h` would include type definitions and function declarations without bodies (called **prototypes**)
  - This file is used as a specification of the queue ADT by textually including it in client code and implementation code using the C preprocessor `#include` directive

Separate Compilation in C and C++ (cont'd.)

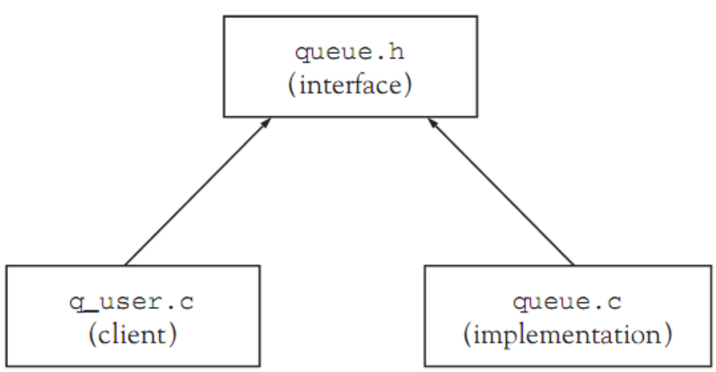


Figure 11.4 Separation of specification, implementation, and client code

Separate Compilation in C and C++ (cont'd.)

```
(1) #ifndef QUEUE_H
(2) #define QUEUE_H

(3) struct Queuerep;
(4) typedef struct Queuerep * Queue;
(5) Queue createq(void);
(6) Queue enqueue(Queue q, void* elem);
(7) void* frontq(Queue q);
(8) Queue dequeue(Queue q);
(9) int emptyq(Queue q);

(10) #endif
```

Figure 11.5 A `queue.h` header file in C

Separate Compilation in C and C++ (cont'd.)

- Definition of the `Queue` data type is hidden in the implementation by defining Queue to be a pointer type
  - Leaves the actual queue representation structure as an **incomplete type**
  - Eliminates the need to have the entire `Queue` structure declared in the header file
- The effectiveness of this mechanism depends solely on convention
  - Neither compilers nor linkers enforce any protections or checks for out-of-date source code

C++ Namespaces and Java Packages

- `namespace` mechanism in C++ provides support for the simulation of modules in C
  - Allows the introduction of a named scope explicitly
  - Helps avoid name clashes among separately compiled libraries
- Three ways to use the namespace:
  - Use the scope resolution operator (`::`)
  - Write a `using` declaration for each name from the namespace
  - "Unqualify" all names in the namespace with a single `using namespace` declaration


```
#ifndef QUEUE_H
#define QUEUE_H

namespace MyQueue
{ struct Queuerep;
  typedef Queuerep * Queue;
            // struct no longer needed in C++
  Queue createq();
  Queue enqueue(Queue q, void* elem);
  void* frontq(Queue q);
  Queue dequeue(Queue q);
  bool emptyq(Queue q);
}

#endif
```

Figure 11.8 The `queue.h` header file in C++ using a namespace

C++ Namespaces and Java Packages (cont'd.)

- Java has a namespace-like mechanism called the **package**:
  - A group of related classes
- Can reference a class in a package by:
  - Qualifying the class name with the dot notation
  - Using an import declaration for the class or the entire `package`
- Java compiler can access any other public Java code that is locatable using the search path
- Compiler will check for out-of date source files and recompile all dependent files automatically

Ada Packages

- Ada's module mechanism is the **package**
  - Used to implement modules and parametric polymorphism
- Package is divided into two parts:
  - **Package specification**: the public interface to the package, and corresponds to the signature of an ADT
  - **Package body**
- Package specifications and package bodies represent compilation units in Ada and can be compiled separately

```
package ComplexNumbers is

type Complex is private;

function "+"(x,y: in Complex) return Complex;
function "-"(x,y: in Complex) return Complex;
function "*"(x,y: in Complex) return Complex;
function "/"(x,y: in Complex) return Complex;
function "-"(z: in Complex) return Complex;
function makeComplex (x,y: in Float) return Complex;
function realPart (z: in Complex) return Float;
function imaginaryPart (z: in Complex) return Float;

private
   type Complex is
   record
      re, im: Float;
   end record;
end ComplexNumbers;
```

Figure 11.10 A package specification for complex numbers in Ada

Ada Packages (cont'd.)

- Any declarations in a **private** section are inaccessible to a client
- Type names can be given in the public part of a specification, but the actual type declaration must be given in the private part of the specification
- This violates the two criteria for abstract data type mechanisms:
  - The specification is dependent on the implementation
  - Implementation details are divided between the specification and the implementation

Ada Packages (cont'd.)

- Packages in Ada are automatically namespaces in the C++ sense
- Ada has a `use` declaration analogous to the using declaration of C++ that dereferences the package name automatically
- **Generic packages**: implement parameterized types

Ada Packages (cont'd.)

```
generic
  type T is private;
package Queues is
  type Queue is private;
  function createq return Queue;
  function enqueue(q:Queue;elem:T) return Queue;
  function frontq(q:Queue) return T;
  function dequeue(q:Queue) return Queue;
  function emptyq(q:Queue) return Boolean;
private
  type Queuerep;
  type Queue is access Queuerep;
end Queues;
```

Figure 11.12 A parameterized queue ADT defined as an Ada generic package specification

Modules in ML

- In addition to the abstract definition, ML has a more general module facility consisting of three mechanisms:
  - **Signature**: an interface definition
  - **Structure**: an implementation of the signature
  - **Functions**: functions from structures to structures, with structure parameters having “types” given by signatures
- Signatures are defined using the `sig` and `end` keywords

Modules in ML (cont’d.)

```
signature QUEUE =
   sig
   type 'a Queue
   val createq: 'a Queue
   val enqueue: 'a Queue * 'a -> 'a Queue
   val frontq: 'a Queue -> 'a
   val dequeue: 'a Queue -> 'a Queue
   val emptyq: 'a Queue -> bool
   end;
```

Figure 11.15 A `QUEUE` signature for a queue ADT in ML

Modules in ML (cont'd.)

```
structure Queue1: QUEUE =
   struct
   datatype 'a Queue = Q of 'a list
   val createq = Q [];
   fun enqueue(Q lis, elem) = Q (lis @ [elem]);
   fun frontq (Q lis) = hd lis;
   fun dequeue (Q lis) = Q (tl lis);
   fun emptyq (Q []) = true
     | emptyq (Q (h::t)) = false;
   end;
```

Figure 11.16 An ML structure `Queue1` implementing the `QUEUE` signature as an ordinary built-in list with wrapper

Modules in ML (cont'd.)

```
structure Queue2: QUEUE =
   struct
   datatype 'a Queue = Createq
                     | Enqueue of 'a Queue * 'a ;
   val createq = Createq;
   fun enqueue(q,elem) = Enqueue (q,elem);
   fun frontq (Enqueue(Createq,elem)) = elem
   | frontq (Enqueue(q,elem)) = frontq q;
   fun dequeue (Enqueue(Createq,elem)) = Createq
                     | dequeue (Enqueue(q,elem))
                               = Enqueue(dequeue q, elem);
   fun emptyq Createq = true | emptyq _ = false;
   end;
```

Figure 11.17 An ML structure `Queue2` implementing the `QUEUE` signature as a user-defined linked list

Modules in ML (cont'd.)

- ML signatures and structures satisfy most of the requirements for abstract data types
- Main difficulty is that client code must explicitly state the implementation to be used in terms of the module name
  - Code cannot be written to depend only on the signature, with the actual implementation structure to be supplied externally to the code
  - This is because ML has no explicit or implicit separate compilation or code aggregation mechanism

Modules in Earlier Languages

- Historically, modules and abstract data type mechanisms began with Simula67
- Languages that contributed significantly to module mechanisms in Ada and ML include CLU, Euclid, Modula-2, Mesa, and Cedar

Euclid

- In the Euclid programming language, modules are types
- Must declare an actual object of the type to use it
- When module types are used in a declaration, a variable of the module type is created, or **instantiated**
- Can have two different instantiations of a module simultaneously
- This differs from Ada or ML, where modules are objects instead of types, with a single instantiation of each


```
type ComplexNumbers = module
   exports(Complex, add, subtract, multiply,
           divide, negate, makeComplex,
          realPart, imaginaryPart)
   type Complex = record
      var re, im: real
   end Complex

   procedure add (x,y: Complex, var z: Complex) =
   begin
       z.re := x.re + y.re
       z.im := x.im + y.im
   end add

   procedure makeComplex
          (x,y: real, var z:Complex) =
   begin
       z.re := x
       z.im := y
   end makeComplex
               ...
end ComplexNumbers
```

Euclid (cont'd.)

```
var C1,C2: ComplexNumbers
var x: C1.Complex
var y: C2.Complex


C1.makeComplex(1.0,0.0,x)

C2.makeComplex(0.0,1.0,y)
(* x and y cannot be added together *)
```


CLU

- In CLU, modules are defined using the **cluster** mechanism
- The data type is defined directly as a cluster
- When we define a variable, its type is not a cluster but what is given by the **rep** declaration
- A cluster in CLU refers to two different things:
  - The cluster itself
  - Its internal representation type

CLU (cont'd.)

```
Complex = cluster is add, multiply,...,
          makeComplex, realPart, imaginaryPart
    rep = struct [re,im: real]
    add = proc (x,y: cvt ) returns (cvt)
      return
      (rep${re: x.re+y.re, im: x.im+y.im})
    end add
      ...
    makeComplex = proc (x,y: real) returns (cvt)
      return (rep${re:x, im:y})

    realPart = proc(x: cvt) returns (real)
      return(x.re)
    end realPart

end Complex
```


CLU (cont'd.)

- `cvt` (for `convert`) converts from the external type (with no explicit structure) to the internal rep type and back again

```
max(2.1,3); // which max?
```

Modula-2

- In Modula-2, the specification and implementation of an abstract data type are separated into a `DEFINITION MODULE` and an `IMPLEMENTATION MODULE`
- `DEFINITION MODULE`: contains only definitions or declarations
  - These are the only declarations that are exported (usable by other modules)
- `IMPLEMENTATION MODULE`: contains the implementation code

Modula-2 (cont'd.)

```
DEFINITION MODULE ComplexNumbers;

TYPE Complex;

PROCEDURE Add (x,y: Complex): Complex;
PROCEDURE Subtract (x,y: Complex): Complex;
PROCEDURE Multiply (x,y: Complex): Complex;
PROCEDURE Divide (x,y: Complex): Complex;
PROCEDURE Negate (z: Complex): Complex;
PROCEDURE MakeComplex (x,y: REAL): Complex;
PROCEDURE RealPart (z: Complex) : REAL;

PROCEDURE ImaginaryPart (z: Complex) : REAL;

END ComplexNumbers.
```

Modula-2 (cont'd.)

- A client module uses a data type by importing it and its functions from the data type's module
- Modula-2 uses the **dereferencing** `FROM` clause
  - Imported items must be listed by name in the `IMPORT` statement
  - No other items (imported or locally declared) may have the same names as those imported

Problems with Abstract Data Type Mechanisms

- Abstract data type mechanisms use separate compilation facilities to meet protection and implementation independence requirements
- ADT mechanism is used as an interface to guarantee consistency of use and implementation
- But ADT mechanisms are used to create types and associate operations to types, while separate compilation facilities are providers of services
  - Services may include variables, constants, or other programming language entities

Problems with Abstract Data Type Mechanisms (cont'd.)

- Thus, compilation units are in one sense more general than ADT mechanisms
- They are less general in that the use of a compilation unit to define a type does not identify the type with the unit
  - Thus, not a true type declaration
- Also, units are static entities that retain their identity only before linking
  - Can result in allocation and initialization problems

Problems with Abstract Data Type Mechanisms (cont'd.)

- Using separate compilation units to implement abstract data types is therefore a compromise in language design
- It is a useful compromise
  - Reduces the implementation question for ADTs to one of consistency checking and linkage

Modules Are Not Types

- In C, Ada, and ML, problems arise because a module must export a type as well as operations
- Would be helpful to define a module to be a type
  - Would prevent the need to arrange to protect the implementation details with an ad hoc mechanism such as incomplete or private declarations
- ML makes this distinction by containing both an `abstype` and a module mechanism
- Module mechanism is more general, but a type must be exported

Modules Are Not Types (cont'd.)

- `abstype` is a data type, but its implementation cannot be separated from its specification
  - Access to the details of the implementation is prevented
- Clients of the `abstype` implicitly depend on the implementation

Modules Are Static Entities

- An attractive possibility for implementing an abstract data type is to simply not reveal a type at all
  - Avoids possibility of clients depending in any way on implementation details
  - Prevents clients from misuse of a type
- Can create a package specification in Ada in which the actual data type is buried in the implementation
  - This is pure imperative programming

Modules Are Static Entities (cont'd.)

- Normally this would imply that only one entity of that data type could be in the client
  - Otherwise, the entire code must be replicated
- This is due to the static nature of most module mechanisms
- In Ada, the generic package mechanism offers a way to obtain several entities of the same type by using multiple instantiations of the same generic package

Modules Are Static Entities (cont'd.)

```
generic
  type T is private;
package Queues is
  procedure enqueue(elem:T);
  function frontq return T;
  procedure dequeue;
  function emptyq return Boolean;
end Queues;
```


Modules That Export Types Do Not Adequately Control Operations on Variables of Such Types

- In the C and Ada examples given, variables of an abstract type had to be allocated and initialized by calling a procedure in the implementation
  - The exporting module cannot guarantee that the initializing procedure is called before the variable is used
- Also allows copies to be made and deallocations performed outside the control of the module
  - Without the user being aware of the consequences
  - Without the ability to return deallocated memory to available storage

Modules That Export Types Do Not Control Operations (cont'd.)

- In C, `x:=y` performs assignment by sharing the object pointed to by `y`
  - `x=y` tests pointer equality, which is not correct when `x` and `y` are complex numbers
- In Ada, we can use a **limited private type** as a mechanism to control the use of assignment and equality
  - Clients are prevented from using the usual assignment and equality operations
  - Package ensures that equality is performed correctly and that assignment deallocates garbage

Modules That Export Types Do Not Control Operations (cont'd.)

```
package ComplexNumbers is

type Complex is limited private;

-- operations, including assignment and equality
  ...
function equal(x,y: in Complex) return Boolean;
procedure assign(x: out Complex; y: in Complex);

private
   type ComplexRec;
   type Complex is access ComplexRec;

end ComplexNumbers;
```

Modules That Export Types Do Not Control Operations (cont'd.)

- C++ allows overloading of assignment and equality
- Object-oriented languages use **constructors** to solve the initialization problem
- ML limits the data type in an `abstype` or `struct` specification to types that do not permit the equality operation
  - Type parameters that allow equality testing must be written with a double apostrophe `''a` instead of a single apostrophe `'a`

Modules That Export Types Do Not Control Operations (cont'd.)

- In ML, types that allow equality must be specified as `eqtype`
  - Example:

```
signature QUEUE =
   sig
   eqtype ''a Queue
   val createq: ''a Queue
   ...etc.
  end;
```

Modules Do Not Always Adequately Represent Their Dependency on Imported Types

- Modules often depend on the existence of certain operations on type parameters
  - May also call functions whose existence is not made explicit in the module specification
- Example: data structures such as binary search tree, priority queue, or ordered list all required an order operation such as the less-than arithmetic operation “<“
- C++ templates mask such dependencies in specifications

Modules Do Not Always Represent Their Dependency (cont’d.)

- Example: in C++ code
  - Template min function specification

```
template <typename T>
T min( T x, T y);
```

  - Implementation shows the dependency

```
// C++ code
template <typename T>
T min( T x, T y)
// requires an available < operation on T
{ return x < y ? x : y;
}
```

Modules Do Not Always Represent Their Dependency (cont'd.)

- In Ada, can specify this requirement using additional declarations in the generic part of a package declaration:

```
generic
type Element is private;
with function lessThan (x,y: Element) return Boolean;
package OrderedList is
...
end OrderedList;
```

- Instantiation must provide the `lessThan` function:

```
package IntOrderedList is new
   OrderedList (Integer,"<");
```


Modules Do Not Always Represent Their Dependency (cont'd.)

- Such a requirement is called **constrained parameterization**
- ML allows structures to be explicitly parameterized by other structures
  - This feature is called a **functor** (a function on structures)

```
functor OListFUN (structure Order: ORDER):
ORDERED_LIST =
   struct
   ...
   end;
```

Modules Do Not Always Represent Their Dependency (cont'd.)

- The functor can be applied to create a new structure:

```
structure IntOList =
   OlistFUN(structure Order = IntOrder);
```

- This makes explicit the appropriate dependencies, but at the cost of requiring an extra structure to be defined that encapsulates the required features


```
signature ORDER =
   sig
   type Elem
   val lt: Elem * Elem -> bool
   end;

signature ORDERED_LIST =
   sig
   type Elem
   type OList
    val create: OList
   val insert: OList * Elem -> OList
   val lookup: OList * Elem -> bool
   end;
functor OListFUN (structure Order: ORDER):
ORDERED_LIST =
   struct
   type Elem = Order.Elem;
   type OList = Order.Elem list;
   val create = [];
   fun insert ([], x) = [x]
     | insert (h::t, x) = if Order.lt(x,h) then x::h::t
                             else h:: insert (t, x);
   fun lookup ([], x) = false
```

Figure 11.18 The use of a functor in ML to define an ordered list *(continues)*

Modules Do Not Always Represent Their Dependency (cont'd.)

```
       | lookup (h::t, x) =
           if Order.lt(x,h) then false
           else if Order.lt(h,x) then lookup (t,x)
           else true;
     end;

structure IntOrder: ORDER =
     struct
     type Elem = int;
     val lt = (op <);
     end;

structure IntOList =
     OListFUN(structure Order = IntOrder);
```

Figure 11.18 The use of a functor in ML to define an ordered list

Module Definitions Include No Specification of the Semantics of the Provided Operations

- In almost all languages, no specification of the behavior of the available operations of an abstract data type is required
- The Eiffel object-oriented language does allow the specification of semantics
  - Semantic specifications are given by preconditions, postconditions, and invariants
- Preconditions and postconditions establish what must be true before and after the execution of a procedure

Module Definitions Include No Specification of Semantics (cont'd.)

- Invariants establish what must be true about the internal state of the data in an abstract data type
- Example: the enqueue operation in Eiffel:

```
enqueue (x:element) is
require
                    not full
ensure
                    if old empty then front = x
                    else front = old front;
                    not empty
end; -- enqueue
```


Module Definitions Include No Specification of Semantics (cont'd.)

- `require` section establishes preconditions
- `ensure` section establishes postconditions
- These requirements correspond to the algebraic axioms:

  frontq(enqueue(q,x)) = if emptyq(q) then x else frontq(q)

  emptyq(enqueue(q,x)) = false

The Mathematics of Abstract Data Types

- An abstract data type is said to have **existential type**
  - It asserts the existence of an actual type that meets its requirements
- An actual type is a set with operations of the appropriate form
  - A set and operations that meet the specification are a **model** for the specification
- It is possible for no model to exist, or many models

The Mathematics of Abstract Data Types (cont'd.)

- Potential types are called sorts, and potential sets of operations are called signatures
  - Thus a sort is the name of a type not yet associated with any actual set of values
  - A signature is the name and type of an operation or set of operations that exists only in theory
- A model is then an actualization of a sort and its signature and is called an algebra
- Algebraic specifications are often written using the sort-signature terminology

The Mathematics of Abstract Data Types (cont'd.)

sort queue(element) **imports** boolean

signature:

createq: queue
enqueue: queue $\times$ element $\rightarrow$ queue
dequeue: queue $\rightarrow$ queue
frontq: queue $\rightarrow$ element
emptyq: queue $\rightarrow$ boolean

axioms:

emptyq(createq) = true
emptyq (enqueue (q, x)) = false
frontq(createq) = error
frontq(enqueue(q,x)) = if emptyq(q) then x else frontq(q)
dequeue(createq) = error
dequeue(enqueue(q,x)) = if emptyq(q) then q else enqueue(dequeue(q), x)

The Mathematics of Abstract Data Types (cont'd.)

- We would like to be able to construct a unique algebra for the specification to represent the type
- Standard method to do this:
  - Construct the **free algebra of terms** for a sort
  - Form the **quotient algebra** of the equivalence relation generated by the equational axioms
- Free algebra of terms consists of all legal combinations of operations

The Mathematics of Abstract Data Types (cont'd.)

- Example: free algebra for sort queue(integer) and signature shown earlier includes:

createq
enqueue (createq, 2)
enqueue (enqueue(createq, 2), 1)
dequeue (enqueue (createq, 2))
dequeue (enqueue(enqueue (createq, 2), -1))
dequeue (dequeue (enqueue (createq, 3)))
etc.

- Note that the axioms for a queue imply that some terms are actually equal:

dequeue (enqueue (createq, 2)) = createq

The Mathematics of Abstract Data Types (cont'd.)

- In the free algebra, no axioms are true
  - To make them true (to construct a type that models the specification), must use axioms to reduce the number of distinct elements in the free algebra
- This can be done by constructing an **equivalence relation** == from the axioms
  - "==" is an equivalence relation if it is symmetric, transitive, and reflexive:

if $x == y$ then $y == x$ (symmetry)
if $x == y$ and $y == z$ then $x == z$ (transitivity)
$x == x$ (reflexivity)

The Mathematics of Abstract Data Types (cont'd.)

- Given an equivalence relation `==` and a free algebra `F`, there is a unique well-defined algebra `F/==` such that `x=y` in `F/==` if and only if `x==y` in `F`
  - The algebra `F/==` is called the quotient algebra of `F` by `==`
  - There is a unique "smallest" equivalence relation making the two sides of every equation equivalent and hence equal in the quotient algebra
- The quotient algebra is usually taken to be the data type defined by an algebraic specification

The Mathematics of Abstract Data Types (cont'd.)

- This algebra has the property that the only terms that are equal are those that are provably equal from the axioms
- This algebra is called the **initial algebra** represented by the specification
  - Using it results in what are called **initial semantics**
- In general, axiom systems should be **consistent** and **complete**
  - Another desirable property is **independence**: no axiom is implied by other axioms

The Mathematics of Abstract Data Types (cont'd.)

- Deciding on an appropriate set of axioms is generally a difficult process
- **Final algebra**: an approach that assumes that any two data values that cannot be distinguished by inspector operations must be equal
  - The associated semantics are called **final semantics**
- A final algebra is also essentially unique
- **Principle of extensionality** in mathematics:
  - Two things are equal precisely when all their components are equal