11. Abstract Data Type and Encapsulation Constructs

- **Topics**
  - The Concept of Abstraction
  - Introduction to Data Abstraction
  - Design Issues for Abstract Data Types
  - Language Examples
  - Parameterized Abstract Data Types
  - Encapsulation Constructs
  - Naming Encapsulations

11.1. The Concept of Abstraction

- An *abstraction* is a view or representation of an entity that includes only the most significant attributes
- The concept of *abstraction* is fundamental in programming (and computer science)
- Nearly all programming languages support *process abstraction* with subprograms
- Nearly all programming languages designed since 1980 support *data abstraction*

11.2. Introduction to Data Abstraction

- An abstract data type is an *enclosure* that includes only the
  - *data representation* of one specific data type and
  - the *subprograms* that provide the operations for that type
- An instance of an abstract data type is called an *object*.

11.2.1. Floating-Point as an Abstract Data Type

- The type name, *double, real, float, ...*
  - specifies the data representation
  - specifies valid operations
  - hides the implementation of the data representation
  - hides the implementation of the operations

11.2.2. User-Defined Abstract Data Types

- An *abstract data type* is a user-defined data type that satisfies the following two conditions:
  - The representation of, and operations on, objects of the type are defined in a single syntactic unit
  - The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition

- Advantages of Data Abstraction
  - Advantage of the first condition
– Program organization, modifiability (everything associated with a data structure is together), and separate compilation

• Advantage the second condition
  – Reliability—by hiding the data representations, user code cannot directly access objects of the type or depend on the representation, allowing the representation to be changed without affecting user code

• Language Requirements for ADTs
  • A syntactic unit in which to encapsulate the type definition
  • A method of making type names and subprogram headers visible to clients, while hiding actual definitions
  • Some primitive operations must be built into the language processor

11.2.3. An Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create(stack)</td>
<td>Create an possibly initializes a stack object</td>
</tr>
<tr>
<td>destroy(stack)</td>
<td>Reclaims storage from the stack object</td>
</tr>
<tr>
<td>empty(stack)</td>
<td>A predicate (or Boolean) function that returns true if the specified stack is empty and false otherwise.</td>
</tr>
<tr>
<td>push(stack,element)</td>
<td>Pushes the specified element on the specified stack</td>
</tr>
<tr>
<td>pop(stack)</td>
<td>Removes the top element from the specified stack.</td>
</tr>
<tr>
<td>top(stack)</td>
<td>Returns a copy of the top element from the specified stack.</td>
</tr>
</tbody>
</table>

11.3. Design Issues for Abstract Data Types

• Design Issues
  • What is the form of the container for the interface to the type?
  • Can abstract types be parameterized?
  • What access controls are provided?

11.4. Language Examples

11.4.1. Abstract Data Types in Ada

• The encapsulation construct is called a package
  – Specification package (the interface)
  – Body package (implementation of the entities named in the specification)
• Information Hiding
The specification package has two parts, public and private
- The name of the abstract type appears in the public part of the specification package. This part may also include representations of unhidden types
- The representation of the abstract type appears in a part of the specification called the private part
  - More restricted form with limited private types
    - Private types have built-in operations for assignment and comparison
    - Limited private types have NO built-in operations
- Reasons for the public/private spec package:
  1. The compiler must be able to see the representation after seeing only the spec package (it cannot see the private part)
  2. Clients must see the type name, but not the representation (they also cannot see the private part)
- Having part of the implementation details (the representation) in the spec package and part (the method bodies) in the body package is not good
- One solution: make all ADTs pointers
- Problems with this:
  1. Difficulties with pointers
  2. Object comparisons
  3. Control of object allocation is lost

11.4.1.1. An Example

package Stack_Pack is
  type stack_type is limited private;
  max_size: constant := 100;
  function empty(stk: in stack_type) return Boolean;
  procedure push(stk: in out stack_type; elem:in Integer);
  procedure pop(stk: in out stack_type);
  function top(stk: in stack_type) return Integer;
private  -- hidden from clients
  type list_type is array (1..max_size) of Integer;
  type stack_type is record
    list: list_type;
    toposub: Integer range 0..max_size) := 0;
  end record;
end Stack_Pack;

11.4.2. Abstract Data Types in C++

- Based on C struct type and Simula 67 classes
- The class is the encapsulation device
- All of the class instances of a class share a single copy of the member functions
- Each instance of a class has its own copy of the class data members
• Instances can be static, stack dynamic, or heap dynamic

• Information Hiding
  – Private clause for hidden entities
  – Public clause for interface entities
  – Protected clause for inheritance (Chapter 12)

• Constructors:
  – Functions to initialize the data members of instances (they do not create the objects)
  – May also allocate storage if part of the object is heap-dynamic
  – Can include parameters to provide parameterization of the objects
  – Implicitly called when an instance is created
  – Can be explicitly called
  – Name is the same as the class name

• Destructors
  – Functions to cleanup after an instance is destroyed; usually just to reclaim heap storage
  – Implicitly called when the object’s lifetime ends
  – Can be explicitly called
  – Name is the class name, preceded by a tilde (~)
11.4.2.1. An Example

```cpp
struct StackException {
    StackException(const char* m)
    {
        cout << endl;
        cout << "I am the Stack and I am " << m << ".";
        cout << endl;
    }
};

class Stack {
    int size;    //Number of available elements to implement the stack
    int tos;    //Index of the element on top of the stack
    int* S;    //An array of integers used to implement the stack

public:
    Stack(int sz=100):size(sz),tos(-1){S=new int[size];}
    ~Stack();
    bool empty(void){return tos<0;}
    bool full(void){return tos>=size-1;}
    void push (int v)
    {  if (full()) throw StackException("full");
       S[++tos]=v;
    }
    void pop ()
    {  if (empty()) throw StackException("empty");
       tos--;
    }
    int top (void)
    {  if (empty()) throw StackException("empty");
       return S[tos];
    }
};
```

• Interface file Stack.h

```cpp
class Stack {
    int size;    //Number of available elements to implement the stack
    int tos;    //Index of the element on top of the stack
    int* S;    //An array of integers used to implement the stack

public:
    Stack(int sz=100);
    ~Stack();
    bool empty(void);
    bool full(void);
    void push (int v);
    void pop ()
    int top (void);
};
```
• Implementation file Stack.cpp

```cpp
struct StackException {
    StackException(const char* m) {
        cout << endl;
        cout << "I am the Stack and I am " << m << " ";
        cout << endl;
    }
};
Stack::Stack(int sz):size(sz),tos(-1){S=new int[size];}
Stack::~Stack() {if (S) delete [] S;}
bool Stack::empty(void){return tos<0;}
bool Stack::full(void){return tos>=size-1;}
void Stack::push (int v) {
    if (full()) throw StackException("full");
    S[++tos]=v;
}
void Stack::pop () {
    if (empty()) throw StackException("empty");
    tos--;
}
int Stack::top (void) {
    if (empty()) throw StackException("empty");
    return S[tos];
}
```

• Evaluation of ADTs in C++ and Ada
  – C++ support for ADTs is similar to expressive power of Ada
  – Both provide effective mechanisms for encapsulation and information hiding
  – Ada packages are more general encapsulations; classes are types
• Friend functions or classes - to provide access to private members to some unrelated units or functions
  – Necessary in C++

### 11.4.3. Abstract Data Types in Java

• Similar to C++, except:
  – All user-defined types are classes
  – All objects are allocated from the heap and accessed through reference variables
  – Individual entities in classes have access control modifiers (private or public), rather than clauses
  – Java has a second scoping mechanism, package scope, which can be used in place of friends
    • All entities in all classes in a package that do not have access control modifiers are visible throughout the package
class StackClass {
  private:
    private int[] stackRef;
    private int[] maxLength, topIndex;
  public StackClass() // a constructor
  { 
      stackRef = new int[100];
      maxLength = 99;
      topIndex = -1;
  }
  public void push(int num) {...};
  public void pop() {...};
  public int top() {...};
  public boolean empty() {...};
}

11.4.4. Abstract Data Types in C#

- Based on C++ and Java
- Adds two access modifiers, internal and protected internal
- All class instances are heap dynamic
- Default constructors are available for all classes
- Garbage collection is used for most heap objects, so destructors are rarely used
- structs are lightweight classes that do not support inheritance
- Common solution to need for access to data members: accessor methods (getter and setter)
- C# provides properties as a way of implementing getters and setters without requiring explicit method calls
11.4.5. Abstract Data Types in Ruby

- Encapsulation construct is the class
- Local variables have “normal” names
- Instance variable names begin with “at” signs (@)
- Class variable names begin with two “at” signs (@@)
- Instance methods have the syntax of Ruby functions (def ... end)
- Constructors are named initialize (only one per class)—implicitly called when new is called
  - If more constructors are needed, they must have different names and they must explicitly call new
- Class members can be marked private or public, with public being the default
- Classes are dynamic

```ruby
class StackClass 
  def initialize 
    @stackRef = Array.new 
    @maxLen = 100 
    @topIndex = -1 
  end 
  def push(number) end 
  def pop end 
  def top end 
  def empty end 
end 
```
11.5. Parameterized Abstract Data Types

- Parameterized ADTs allow designing an ADT that can store any type elements (among other things) – only an issue for static typed languages
- Also known as generic classes
- C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs

11.5.1. Ada

- Ada Generic Packages
  - Make the stack type more flexible by making the element type and the size of the stack generic

    generic
    Max_Size: Positive;
    type Elem_Type is private;
    package Generic_Stack is
      Type Stack_Type is limited private;
      function Top(Stk: in out StackType) return Elem_type;
      ...
    end Generic_Stack;
    Package Integer_Stack is new Generic_Stack(100,Integer);
    Package Float_Stack is new Generic_Stack(100,Float);

11.5.2. C++

- Classes can be somewhat generic by writing parameterized constructor functions

    class Stack {
      ...
      Stack (int size)
      { stk_ptr = new int [size];
        max_len = size - 1;
        top = -1;
      }
    ...
    Stack stk(100);
The stack element type can be parameterized by making the class a templated class

```cpp
struct StackException {
    StackException(const char* m)
    {
        cout << endl;
        cout << "I am the Stack and I am " << m << ";";
        cout << endl;
    }
};

template class<T>
class Stack {
    int size;    //Number of available elements to implement the stack
    int tos;    //Index of the element on top of the stack
    T* S;    //An array of integers used to implement the stack
public:
    Stack(int sz=100):size(sz),tos(-1){S=new T[size];}
    ~Stack () {if (S) delete [] S;}
    bool empty(void){return tos<0;}
    bool full(void){return tos>=size-1;}
    void push (T v)
    {
        if (full()) throw StackException("full");
        S[++tos]=v;
    }
    void pop ()
    {
        if (empty()) throw StackException("empty");
        tos--;
    }
    T top (void)
    {
        if (empty()) throw StackException("empty");
        return S[tos];
    }
};
```

```
Stack<int> S;
S.push(15);
```
11.5.3. Java 5.0

- Generic parameters must be classes
- Most common generic types are the collection types, such as LinkedList and ArrayList
- Eliminate the need to cast objects that are removed
- Eliminate the problem of having multiple types in a structure
- Parameterized Classes in Java 5.0
  - Prior to Java 5.0
    
    \begin{verbatim}
    ArrayList myArray=new ArrayList();
    myArray.add(0,new Integer(47));
    Integer myInt=(Integer)myArray.get(0);
    \end{verbatim}

  - Java 5.0
    
    \begin{verbatim}
    ArrayList <Integer> myArray=new ArrayList <Integer>();
    \end{verbatim}

11.5.4. C# 2005

- Similar to those of Java 5.0
- Elements of parameterized structures can be accessed through indexing

11.6. Encapsulation Constructs

11.6.1. Introduction

- Large programs have two special needs:
  - Some means of organization, other than simply division into subprograms
  - Some means of partial compilation (compilation units that are smaller than the whole program)
- Obvious solution: a grouping of subprograms that are logically related into a unit that can be separately compiled (compilation units)
- Such collections are called encapsulation

11.6.2. Nested Subprograms

- Organizing programs by nesting subprogram definitions inside the logically larger subprograms that use them
- Nested subprograms are supported in Ada, Fortran 95, Python, and Ruby
11.6.3. Encapsulation in C

- Files containing one or more subprograms can be independently compiled
- The interface is placed in a header file
- Problem: the linker does not check types between a header and associated implementation
- #include preprocessor specification – used to include header files in applications

11.6.4. Encapsulation in C++

- Can define header and code files, similar to those of C
- Or, classes can be used for encapsulation
  - The class is used as the interface (prototypes)
  - The member definitions are defined in a separate file
- Friends provide a way to grant access to private members of a class

11.6.5. Ada Packages

- Ada specification packages can include any number of data and subprogram declarations
- Ada packages can be compiled separately
- A package’s specification and body parts can be compiled separately

11.7. Naming Encapsulations

- Large programs define many global names need a way to divide into logical groupings
- A naming encapsulation is used to create a new scope for names

11.7.1. C++ Namespaces

- Can place each library in its own namespace and qualify names used outside with the namespace
- C# also includes namespaces

11.7.2. Java Packages

- Packages can contain more than one class definition; classes in a package are partial friends
- Clients of a package can use fully qualified name or use the import declaration
11.7.3. Ada Packages

- Packages are defined in hierarchies which correspond to file hierarchies
- Visibility from a program unit is gained with the with clause

11.7.4. Ruby Modules

- Ruby classes are name encapsulations, but Ruby also has modules
- Typically encapsulate collections of constants and methods
- Modules cannot be instantiated or subclassed, and they cannot define variables
- Methods defined in a module must include the module’s name
- Access to the contents of a module is requested with the require method

11.8. Summary

- The concept of ADTs and their use in program design was a milestone in the development of languages
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding
- Ada provides packages that simulate ADTs
- C++ data abstraction is provided by classes
- Java’s data abstraction is similar to C++
- Ada, C++, Java 5.0, and C# 2005 support parameterized ADTs
- C++, C#, Java, Ada, and Ruby provide naming encapsulations
-