Chapter 11

Abstract Data Types and Encapsulation Concepts

"속내용 감추기 "

What makes the programming difficult is that computer representations of data are unnatural !

Abstraction is a weapon against the complexity of programming; its purpose is to simplify the programming process. It is an effective weapon because it allows programmers to concentrate on essential attributes and ignore subordinate attributes

The two primary features of data abstraction are
- encapsulation of data objects with their associated operations, and
- information hiding

11.1 The Concept of Abstraction

- **General concept of abstraction**
  - In general, the concept of abstraction holds that some category of processes or objects can be represented by only *a subset of its attributes*. These are the essential attributes of category, with all the other attributes *abstracted away* or *hidden*
  - Abstraction is a weapon against the complexity of programming;
    ⇔ It is a effective weapon because it allows programmers to concentrate on essential attributes and ignore subordinate attributes
  - Although the concept of abstraction is relatively simple, *its use did not become convenient and safe until language were designed to support it*
    ⇔ language supports for data abstraction (Abstract Data Type)

- **Two Kinds of abstractions in PL**
  - ① ***Process Abstraction***
    - ⇔ all subprograms are process abstraction
    - ⇔ they are way of allowing a program to specify that some process is to be done, without spelling out how it is to be done (at least in the calling program)

```
SORT_INT (LIST, LENGTH)
```

→ an abstraction of the actual sorting process, whose algorithm is not specified

→ the call is independent of the algorithm implemented in the called subprogram

- Bubble sort ? Quick sort ?

→ the essential attributes are the name of array to be sorted, the type of its elements, and the array's length

  - ② ***Data Abstraction***
    - ⇔ representations and *implementation details* are hidden from programmer

11.2 Introduction to Data Abstraction

- Data abstraction as a concept of programming methodologies was discovered much later than process abstraction
  - ⇔ data-oriented programming
- It is a weapon against complexity, a mean of making large and/or complicated programs more manageable

(1) Floating-Point as an Abstract Data Type

- **all built-in types are abstract data type**
  - Example : Floating-point data type
    - ⇔ provides a means of creating variables for floating-point data
    - ⇔ provides a set of arithmetic operations (+, *, −, /) for manipulating object of the type

```
float a ;
int b ;
```

- ***Information Hiding*** in Floating-point types
  - the actual format of the data value in a floating-point memory cell is usually hidden from the user
    - ⇔ the user is not allowed directly manipulate the actual representation of floating point objects
  - the only operations available are those provided by the system
    - ⇔ the user is not allowed to create new operations on data of the type
  - These make it possible to have a flexible data representation, rather than one fixed in some particular format
    - ⇔ allows program portability between implementations, even though the implementations may use different representations of floating-point values

(2) User-Defined Abstract Data Types

- (User-Defined) *Abstract data type* is a data type that satisfies the following two conditions :
  - the representation, or definition, of the type and the operations on objects of the type are described in a single syntactic unit (*encapsulation*)
    - ⇔ *grouping*
    - ⇔ *compilation unit*
  - the representation of objects of the type are hidden from the program units that use the type, so that the only direct operations possible on those objects are those provided in the type's definition (*information hiding*)
- The advantages of packaging the representation and operations in a single syntactic unit are :
  - *Localized modifications (by encapsulation)*
    - ⇔ program units that use the type are not able to "*see*" the representation detail, and thus their code cannot depend on that representation
      - ⇒ representation can be changed at any time without affecting the program units that use the type
  - *Increased reliability (by information hiding)*
    - ⇔ program units cannot change part of the underlying representation directly, either intentionally or by accident, thus increasing the integrity of such objects

- Example : abstract data type *stack*
  - Operations (abstract properties of *stack*)

```
⇔ create(stack)
⇔ destroy(stack)
⇔ empty(stack)
⇔ push(stack, element)
⇔ pop(stack)
⇔ top(stack)
```

array or linked list implementation ?

⇔ the goal of data abstraction is to provide the facilities so that programs can be written that depend only on the abstract properties, not on the representation of data objects

Usage of *stack*

```
int i, k;
stack STK1, STK2;
....
push(STK1, COLOR1) ;
push(STK1, COLOR2) ;
...
if (not empty(STK1)) then TEMP := top(STK1) ;
...
push(STK2, TEMP) ;
......
```

11.3 Design Issues for Abstract Data Types

- A complete facility for defining abstract data type in a language must provide *a syntactic unit* that can be encapsulate *the type definition* and *subprogram definitions* of the abstraction operations
  - it must be possible to make the type names and subprogram headers visible to other program units that use the abstraction (*interface*)
  - assignment and comparison for equality are the only operations that should be builtin
- The encapsulation requirement of abstract data type can be met in two distinct ways
  - an encapsulation construct can be designed to provide a single data type and its operations
    - ⇔ Concurrent Pascal, Smalltalk, C++
  - to provide a more generalized encapsulation construct that can define any number of entities, any of which can be selectively specified to be visible outside the encapsulating unit
    - ⇔ Modula-2, Ada
- Design Issues
  - restricting the kinds of types that can be abstract
  - whether abstract data types can be generic (or parameterized)
  - how imported types can be qualified to prevent collision between local and nonlocal names

11.4 Language Examples

(1) SIMULA 67 Classes

- the first language facilities for the direct support of data abstraction

- *Encapsulation*
  - A SIMULA 67 class definition is *a template for type*
  - Instances of class are created dynamically at the request of the user program and can be referenced only with pointer variables
  - Syntactic form of a class definition

```
class class_name ;
    begin
      -- class variable definition --
      -- class subprogram definitions --
      -- class code section --
   end class_name ;
```

    - ⇔ the code section of a class instance is executed only once, at instance creation time (for initialization of class variables)
  - SIMULA 67's contribution to data abstraction is to have the class construct allow data declarations and the procedures that manipulate them to be syntactically encapsulated
- *Information Hiding*
  - the variables that are declared in a SIMULA 67 class are not hidden from other program units that allocate class objects using the class
    - ⇔ can be accessed by the class subprograms, or
    - ⇔ directly through their names
      - ⇒ does not provide information hiding facilities completely

(2) Abstract Data Types in Ada

- *Encapsulation*
  - the encapsulating constructs, or module, in Ada is called *packages*
  - packages can have two parts, each of which is also called package
    - ⇔ specification package
    - ⇔ body package
- *Information Hiding*
  - the specification has two sections;
    - ⇔ entirely visible to importers
    - ⇔ partially visible outside the package (*private*)

```
package LINKED_LIST_TYPE is
    type NODE_TYPE is private ;          -- visible part
    ...
    private                              -- hidden part
        type NODE_TYPE ;
        type PTR is access NODE_TYPE ;
        type NODE_TYPE is
            record
                INFO : INTEGER ;
                LINK : PTR ;
            end record ;
end LINKED_LIST_TYPE ;
```

visible part

hidden part

- **Example : stack ADT**

Specification

```
package STACKPACK is
   type STACKTYPE is limited private ;
   MAX_SIZE : constant := 10 ;
   function EMPTY (STK : in STACKTYPE) return BOOLEAN ;
   procedure PUSH (STK : in out STACKTYPE; ELEMENT : in INTEGER);
   procedure POP(STK : in out STACKTYPE) ;
   function TOP (STK : in STACKTYPE)  return INTEGER ;
 private:
      type LIST_TYPE is array (1..MAX_SIZE) of INTEGER ;
      type STACKTYPE is
         record
              LIST : LIST_TYPE ;
              TOPSUB : INTEGER range 0..MAX_SIZE := 0 ;
         end record
end STACKPACK ;
```

visible

hidden

Body

```
with TEXT_IO ; use TEXT_IO ;
package body STACKPACK is
    function EMPTY (STK: in STACKTYPE) return BOOLEAN is
        begin
            return STK.TOPSUB = 0
        end EMPTY ;
    procedure PUSH (STK : in out STACKTYPE; ELEMENT : in INTEGER) is
        begin
           if STK.TOPSUB >= MAX_SIZE
               then PUT_LINE("ERROR-Stack Overflow" ;
               else STK.TOPSUB:=STK.TOPSUB+1; STK.LIST(YOPSUB):=ELEMENT;
           end if
    end PUSH
   .....
end STACKPACK
```

```
with STACKPACK, TEXT_IO ;
use STACKPACK, TEXT_IO ;
procedure USE_STACKS is
     TOPONE : INTEGER ;
     STACK : STACKTYPE ;
     begin
        ....
        PUSH (STACK, 42) ;
        PUSH(STACK, 17) ;
        POP(STACK) ;
        TOPONE := TOP(STACK) ;
        ....
        TOPSUB := … /* ? */
    end USE_STACKS ;
```

Usage

(3) Abstract Data Types in C++

- C++ is a language that was created by adding facilities to support OOP to C, it also supports data abstraction with class

- **Encapsulation**
  - a C++ *class* is a template for a data type and therefore can be instantiated any number of times
    - ⇔ data members (instance variable)
    - ⇔ member functions (method)
  - all instance of a class share a single set of *member functions*, but each instance gets its own set of the class's *data members*
  - class instance (object)
    - ⇔ static
    - ⇔ semidynamic
      - ⇒ created by elaboration of an object declaration
    - ⇔ explicit dynamic
      - ⇒ created by *new*, *delete* operators

- **Information Hiding**
  - C++ class can contain both hidden and visible entities
    - ⇔ private : hidden entities
    - ⇔ public : visible entities (class interface)
    - ⇔ protected : related to subclass
  - Class constructor function
    - ⇔ used to initialize and provide parameters to the object creation process
    - ⇔ it is implicitly called when an instance of the class type is created
  - Class destructor function
    - ⇔ it is implicitly called when the life time of an instance of the class type ends

- **Example**

```
#include <iostream.h>
class stack {
   private :
      int *stack_ptr ;
      int max_len ;
int top_ptr ;

public :

      stack() { /* Constructor */
         stack_ptr = new int [100];
         max_len = 99 ;
         top_ptr = -1
      } ;
      ~stack() { /* Destructor */
         delete stack_ptr ;
      } ;
      void push(int number) {
         if (top_ptr == max_len)
            cout << "Error -- Stack is full \n";
         else stack_ptr[++top_ptr] = number ;
      }
      void pop() {
         if (top_ptr == -1)
            cout << "Error-- Stack is empty \n";
         else top_ptr-- ;
      }
      int top() { return(stack_ptr[top_ptr]) ;
      }
      int empty() { return(top_ptr == -1) ;
      }
}
```

```
/* A stack class generic in the size */
stack (int size) {
    stk_ptr = new int [size] ;
    max_len = size - 1 ;
    top = -1 ;
}

stack stk(100), stk() ;
```

Overloaded Constructor functions

Class (type) 이름

```
main() {
   int top_one ;
   stack stk ;
   ...
   stk.push(42) ;
   stk.push(17) ;
   stk.pop() ;
   top_one = stk.top();
   ...
   stk.top_ptr++:
}
```

(5) Abstract Data Types in Java

– Java support abstract data types is similar to C++

- **Differences**
  - all user-defined data types in Java are classes
    - ⇔ Java does not include `struct`
  - all objects are allocated from the heap and accessed through reference variables
  - *a method body must appear with its corresponding method header*
    - ⇔ a Java abstract data type is both declared and defined in a single syntactic unit
  - the lack of a destructor in the Java version, obviated by Java's implicit garbage collection

```
class StackClass {
   private:
        private int [] *stackRef;
        private int [] maxLen, topIndex;
        public StackClass() { // a constructor
                stackRef = new int [100];
                maxLen = 99;
                topPtr = -1;
        };
        public void push (int num) {…};
        public void pop () {…};
        public int top () {…};
        public boolean empty () {…};
}
```


11.5 Parameterized Abstract Data Types

- Why ?
  - to design a stack abstract data type that can store any scalar type elements rather than be required to write a separate stack abstraction for every different scalar types
  - C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs
- Generic Packages (Ada) ➔ a generic stack abstract type

Specification

```
generic
     MAX_SIZE : POSITIVE ;
     type ELEMENT_TYPE is private ;
package GENERIC_STACK is
     type STACKTYPE is limited private ;
     function EMPTY(STK : in STACKTYPE) return BOOLEAN ;
     procedure PUSH(STK : in out STACKTYPE; ELEMENT : in ELEMENT_TYPE) ;
     procedure POP(STK : in out STACKTYPE);
     function TOP(STK : in STACKTYPE) return ELEMENT_TYPE ;
  private
    type LIST_TYPE is array (1..MAX_SIZE) of ELEMENT_TYPE ;
    type STACKTYPE is
         record
            LIST : LIST_TYPE ;
            TOPSUB : INTEGER range 0..MAX_SIZE := 0 ;
         end record
end GENERIC_STACK ;
```

visible

hidden

```
package INTEGER_STACK is new GENERIC_STACK(100, INTEGER) ;
package FLOAT_STACK is new GENERIC_STACK(500, FLOAT) ;
```

New ADT creation

- **Parameterized ADTs in C++**
  - The stack element type can be parameterized by making the class a templated class

```
template <class Type>
class Stack {
  private:
    Type *stackPtr;
    const int maxLen;
    int topPtr;
  public:
    Stack() {  // Constructor for 100 elements
      stackPtr = new Type[100];
      maxLen = 99;
      topPtr = -1;
    }
    Stack(int size) {  // Constructor for a given number
      stackPtr = new Type[size];
      maxLen = size - 1;
      topSub = -1;
    }
    ……
}
```

  - Instantiation: `Stack<int> myIntStack;`

11.6 Encapsulation Constructs

- **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**

① **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, JavaScript, and Ruby

② **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

11.7 Naming Encapsulation

- **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

① **C++ Name spaces**
  - Can place each library in its own namespace and qualify names used outside with the namespace
  - C# also includes namespaces

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

③ **Ada Packages**
  - *Packages* are defined in hierarchies which correspond to file hierarchies
  - Visibility from a program unit is gained with the *`with`* clause