Abstract Data Type

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Contents

- Data types
- Data value and operations
- Built-in data types
- Complex data types
- Abstract data types
- Primitive operations
- Complicated Operations
- Data encapsulation
- Violation of data encapsulation
- Advantages of ADT
- Complex number example of ADT

Data Types Revisited

What is a data type ?

A data type defines the characteristics of a set of data.

This includes

- The range of values it stores
- The memory resources required to store allowable values

Example: 4-bit integers

Range:	-8 to +7
Memory Resources:	4 bits

Data Types Revisited

- A data type is an *abstraction*. It is a template for the type of data to be stored.
- We design data types to model a specific set of data. In a programming language we create *instances* of a data type by declaring variables of that type:

e.g. in C

```
int sum;
```

creates an instance of a 32-bit integer variable.

Data Types

- ✧ Instances of an data type have to obey the *operational specification* of the ADT

Example

```
char string1[50];
char string2[50];
gets(string1);
gets(string2);
puts(string1*string2);
```

invalid operation on strings

Data Types

- ✧ <u>Operations</u> cannot be applied to an instance of a data type unless they are defined.
- ✧ <u>Operations</u> on an data type are specified as functions with strictly defined inputs and output i.e. inputs and output are required to be of the correct data type.

Example: An integer addition function

```
int add (int x, int y)
{
   return x+y;
}
```

Built-In Data Types

- ✧ One of the key responsibilities of a compiler is to identify incompatibilities in the function parameters at compile-time.

Built-in Data Types

C provides a number of built-in data types e.g.

int	(integer)
float	(real number)
char	(alphanumeric character)

- ✧ For each data type there is a set of compatible operations that can be applied to instances of that data type.

Complex Data Types

- ✧ In the course of Data Structure we saw that built-in data types are not sufficient when modelling complex sets of data.
- ✧ When implementing a programming solution we look to represent entities in their most natural form.

Example: A Student Record System

Fundamental Entity or Object:

A student record

Entity composed of:

Name, ID, Age, Phone, Address, ...

Abstract Data Type (ADT)

- In the course of “Program Design”, we focused primarily on the algorithmic solution to problem specifications i.e. writing instructions to perform a particular task.
- Of greater importance in Computer Science is the design and representation of the entities that a program manipulates.
- The identification and representation of the entities involved in any particular application is the single most important step in the design of any computer program.
- The data specification (values that are stored) and the operational specification (operations on the data) of the entity is referred to as an Abstract Data Type (ADT).

Abstract Data Type (ADT)

Definition:

An Abstract Data Type is an entity with a collection of attributes and a set of operations defined on it.

- Types of Operations: In general there are three types of operations that are performed on an ADT:

1. Constructor Operations
2. Accessor Operations
3. Testing Operations

Primitive Operations

Constructor Operations

These operations are used to construct instances of an ADT or to set the values of its attributes

Ex. Assume the following attributes for an ADT are defined:

```
fraction
{
    integer numerator
    integer denominator
}
```

Primitive Operations

Sample Constructor Operations

Module Name:	Initialise_Fraction
Inputs:	integer a, integer b
Output:	fraction f with numerator a and denominator b

Module Name:	Change_Numerator
Inputs:	integer n, fraction f
Output:	Fraction f′ with numerator n and the same denominator

Primitive Operations

Module Name:	Change_Denominator
Inputs:	integer d, fraction f
Output:	Fraction f ′ with denominator d and the same numerator

Primitive Operations

Accessor Operations

These operations are used to determine the value of any of the ADT′s attributes.

Module Name:	Get_Numerator
Input:	fraction f
Output:	the numerator of f

Module Name:	Get_Denominator
Input:	fraction f
Output:	the denominator of f

Primitive Operations

Testing Operations

These operations test the properties of the ADT′s attributes.

Module Name:	Is_Simplified
Inputs:	fraction f
Output:	TRUE if the numerator and denominator of f has no factors otherwise FALSE

E.g. 7/13 is simplified
4/12 is not simplified

Complicated Operations

Using these simple operations, more complicated tasks can be performed:

Ex. Fraction Arithmetic

Module Name:	Fraction_Multiply
Inputs:	fraction a, fraction b
Output:	fraction c

Process:

```
numerator ← Get_Numerator(a) * Get_Numerator(b)
denominator ← Get_Denominator(a) * Get_Denominator(b)
c ← Initialise_Fraction(numerator, denominator)
RETURN c
```

Complicated Operations

Module Name:	Fraction_Subtraction
Inputs:	fraction a, fraction b
Output:	fraction c
Process:	

```
numerator ← ( Get_Numerator(a) * Get_Denominator(b) ) +
            ( Get_Numerator(b) * Get_Denominator(a) )
denominator ← Get_Denominator(a) * Get_Denominator(b)
c ← Initialise_Fraction(numerator,denominator)
RETURN c
```

Data Encapsulation

- Data Encapsulation or "Information Hiding" is the process whereby only the constructor, accessor and testing operations have direct access to the attributes of the ADT.
- Consider a C implementation of the fraction ADT. The following structure is defined:

```
typedef struct fraction_ADT
{
        int numerator;
        int denominator;
} fraction;
```

C Implementations

- C implementations of the constructor and accessor operations could be declared as:

```
fraction Initialise_Fraction(int a, int b)
{
   fraction f;

   f.numerator = a;
   f.denominator = b;
   return f;
}
```

C Implementations

```
int Get_Numerator (fraction f )
{
   return f.numerator;
}

int Get_Denominator (fraction f )
{
   return f.denominator;
}
```

Violation of Data Encapsulation

- Disastrous results can occur when the principle of data encapsulation is violated.
- Suppose the programmer implements the Fraction_Multiply() operation as follows:

```
fraction Fraction_Multiply(fraction a, fraction b)
{
   fraction c;
   c.numerator = a.numerator * b.numerator
   c.denominator = a.denominator * b.denominator
   return c;
}
```

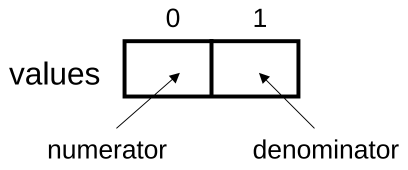
Constructor and accessor functions violated

Progressive Modifications

- If the programmer decides to change the internal representation of the Fraction ADT′s attributes then the Fraction_Multiply() function will be invalid.

E.g. the programmer decides to use an array to represent the fraction ADT attributes

```
typedef struct fraction_ADT
{
   int values[2];
} fraction;
```



Advantages of Data Encapsulation

- Assuming valid constructors, accessors and testing operations are provided, we do not need to know or access the internal representation of the attributes of an ADT.
- This principle of Data Encapsulation is an important tool in modern software engineering particularly in a team environment.
- Each programmer need NOT be aware of the internal representation of the other′s ADT. The only interface required by the programmers is the constructor, accessor and testing operations.

Advantages of ADT

1. The specification of ADT′s leads to more readable code e.g. Get_Numerator(f) is more readable than f[0].

2. ADT′s impose a modular design on code construction.

   Once an ADT′s attributes and fundamental operations have been specified, a number of programmers can begin to implement the program. One group can implement the ADT′s while others write the code that uses them.

Advantages of ADT (cont′d)

3. Code Re-Use

   ADT′s like libraries of functions can be stored and used in other programs by simply including the ADT′s internal representation and fundamental operations.

4. ADT′s have been studied extensively in Computer Science and a wealth of knowledge is available which describes the most efficient ADT′s for certain applications. Their internal representation, constructor, accessor and testing functions have been highly developed and tested over the years.

ADT Example: Complex Number

- Goal: Specify an ADT that can be used to represent a ‟complex number″ in a mathematical application.

1. Specify an internal representation of its attributes
2. Define three constructor operations:
   - initializeComplexNumber()
   - setRealPart()
   - setImaginaryPart()
3. Define two accessor operations:
   - getRealPart()
   - getImaginaryPart()
4. Using your ADT write the following modules:
   - addComplexNumbers()
   - findConjugate()

Complex Number (cont′d)

Representation of Attributes

Example: $\underbrace{3.2}_{\text{real part}}+\underbrace{4.5i}_{\text{imaginary part}}$

```
complex_number
{
        double real
        double imaginary
}
```

Complex Number (cont′d)

<u>Constructor Operations</u>

```
Module Name:    initialiseComplexNumber
Inputs:         double r, double i
Output:         complex_number c
Process:
        c.real ← r
        c.imaginary ← i
        return c
```

Complex Number (cont’d)

Module Name: setRealPart
Inputs: double r, complex_number c
Output: complex_number c
Process:

```
    c.real ← r
    return c
```

Complex Number (cont’d)

Module Name: setImaginaryPart
Inputs: double i, complex_number c
Output: complex_number c
Process:

```
    c.imaginary ← i
    return c
```

Complex Number (cont’d)

Accessor Operations

Module Name: getRealPart
Inputs: complex_number c
Output: double r
Process:

```
    r ← c.real
    return r
```

Complex Number (cont’d)

Module Name: getImaginaryPart
Inputs: complex_number c
Output: double i
Process:

```
    i ← c.imaginary
    return i
```

Complex Number (cont'd)

Functions

Module Name:	addComplexNumbers
Inputs:	complex_number c_1, complex_number c_2
Output:	complex_number c_3

Process:

```
    real ← getRealPart(c_1)+getRealPart(c_2)
    imaginary ← getImaginaryPart(c_1)+getImaginaryPart(c_2)
    c_3 ← initialiseComplexNumber(real, imaginary)
    return c_3
```

Complex Number (cont'd)

Module Name:	findConjugate
Inputs:	complex_number c_1
Output:	complex_number c_2

Process:

```
    real ← getRealPart(c_1)
    imaginary ← -getImaginaryPart(c_1)
    c_2 ← InitialiseComplexNumber(real, imaginary)
    return c_2
```