1 Introduction

An Abstract Data Type (ADT) is a data type that is defined by the user. In contrast with the data types that are built-in in the C language (e.g., int, char, float, etc.), the ADTs are completely invented by the programmer. Since an ADT is an invention of the programmer, the programmer is in charge of supplying everything that is related to the data type to the compiler. E.g., if I decide to create a data type to represent fractions, then each component of such data type has to be accurately described by the programmer (since the C compiler does not know anything about fractions).

Remember that, when we talked about data types for the first time, we defined a data type to be composed by two parts:

- a Domain, i.e., the set of possible values that can be assigned to objects of such data type
- a set of Operations that are provided to manipulate objects belonging to the data type.

Thus, when we want to describe any data type (built-in or ADT), we need to provide these two pieces of information. A precise description of a data type is called Specification of the Data Type. Each time we want to define a new ADT, the first thing that we need to produce is its specification. Observe that the specification of the ADT is just a description of the ADT, and is not related to any specific implementation of the ADT.

2 Specification of the Fraction ADT

Here below we are providing a precise specification of the Fraction ADT. In the specification we are identifying the domain of the type, as well as all the operations that we will provide with such data type. These operations are going to be the only actions that we will be allowed to perform on objects of this type. The specification is presented in Fig. 1.

As can be seen from the specification, the operations have been classified in four groups:
1. **Constructors:** these are the operations that are used to create objects belonging to the data type. It is very important to understand the meaning of types and constructors. Let us assume that we are defining a data type called `Fraction`. This means that in our program we can define variables of this type, e.g.,

```c
main()
{
    Fraction x, y;
    ....
}
```

Now, it is important to remember that `x, y` in this code fragment are *not* objects of type `Fraction`, but they are simply “containers” that can store fractions. If we want an actual fraction, we need to explicitly create it using one of the constructor operations. E.g.,

```c
main()
{
    Fraction x, y;

    x = CreateFraction(1, 6); // create fraction 1/6
    ....
}
```

2. **Destructors:** these are operations used to destroy object of the ADT that are not needed any longer. This allows the programmer to perform garbage collection during the execution of the program.

3. **Inspectors:** these are operations that are used to access attributes of the objects and/or test properties of the objects. E.g., a typical inspector is an operation that allows to test whether two objects of an ADT have the same value.

4. **Modifiers:** these are operations that allow one to either modify one object or create a new object from other objects.
3 Implementation

A separate component of an ADT is its implementation. The implementation is going to be a library that provides:

- a representation for the objects of the new type in terms of C data types (structures, arrays, etc.); a *typedef* will be provided to associate to such memory representation the name of the new data type.

- separate C functions to implement all the ADT operations described in the specification.

We expect the library to export:

- the name of the new type (but not its internal representation, which is going to be hidden in the library)

- the functions representing the operations of the ADT

Here below we provide the code for the implementation of the Fraction ADT. Let us start by providing the interface of the library (file `fraction.h`):
// fraction.h: this is the interface of the library to
// implement the Fraction ADT

#include "genlib.h"

// definition of the Fraction type
typedef struct my_fraction *Fraction;

// Operations

// CreateFraction
//   Inputs: two integers (n,m)
//   Output: the fraction n/m, if m is not zero; otherwise NULL
//   Effect: creates a new object of type Fraction and initializes
//           it to the value n/m

Fraction CreateFraction(int n, int m);

// DestroyFraction
//   Inputs: one object of type Fraction
//   Outputs: none
//   Effect: the object given as input is destroyed

void DestroyFraction(Fraction input);

// EqualFractions
//   Inputs: two objects of type fraction (p,q)
//   Outputs: a boolean value
//   Effect: returns TRUE if the fractions p and q have the same
//           value, FALSE otherwise

bool EqualFractions(Fraction p, Fraction q);

// GreaterThanFraction
//   Inputs: two objects of type fraction (p,q)
//   Output: a boolean value
//   Effect: returns TRUE if p has a value greater than q

bool GreaterThanFraction(Fraction p, Fraction q);

// PrintFraction
//   Input: one object of type fraction
//   Output: none
//   Effect: print the value of the input fraction on the screen

void PrintFraction (Fraction x);
// AddFractions
// Inputs: two objects of type fraction
// Output: one object of type fraction
// Effect: a new fraction is created, whose value is the sum
//         of the values of the two fractions given in input

Fraction AddFractions (Fraction p, Fraction q);

// MultiplyFractions
// Inputs: two objects of type fraction
// Output: one object of type fraction
// Effect: a new fraction is created, whose value is the product
//         of the values of the two fractions given in input

Fraction MultiplyFractions (Fraction p, Fraction q);

//- End of Interface
The implementation part of the library is stored in the file called fraction.c, and its content is the following:

```c
//-- library Fraction: Implementation Part
#include <stdio.h>
#include "fraction.h" // include the interface

//-- define the basic representation of fractions
struct my_fraction
{
    int num;
    int den;
};

//-- Operations

//-- Constructor

Fraction CreateFraction(int n, int m)
{
    Fraction new;

    // create the memory for new object
    new = malloc(sizeof(struct my_fraction));
    if (new == NULL)
        Error("Not enough memory for the new object");
    new->num = n;
    new->den = m;
    return new;
}

//-- Destructor

void DestroyFraction(Fraction input)
{
    free(input);
}

//-- Inspector: EqualFractions

bool EqualFractions (Fraction p, Fraction q)
{
    int g1, g2;

    g1 = gcd(p->num, p->den);
    g2 = gcd(q->num, q->den);
    if ((p->num / g1) == (q->num / g2) &
```
(p->den / g1) == (q->den / g2) )
    return TRUE;
else
    return FALSE;
}

// Inspector: GreaterThanFraction
bool GreaterThanFraction(Fraction p, Fraction q)
{
    if ( (p->num / p->den) > (q->num / q->den))
        return TRUE;
    else
        return FALSE;
}

// Inspector: PrintFraction
void PrintFraction (Fraction p)
{
    printf(''%d
'', p->num);
    printf(''----
'');
    printf(''%d
'', p->den);
}

// Modifier: AddFractions
Fraction AddFractions (Fraction p, Fraction q)
{
    Fraction result;
    // create new object for result
    result = CreateFraction(0,1);

    result->num = p->num * q->den + p->den * q->num;
    result->den = p->den * q->den;
    return result;
}

// Modifier: MultiplyFractions
Fraction MultiplyFractions (Fraction p, Fraction q)
{
    Fraction result;
    // create new object for result
    result = CreateFraction(0,1);

    result->num = p->num * q->num;
    result->den = p->den * q->den;
return result;
}
Specification Fraction
{
  DOMAIN: the set of all possible fractions (i.e., the set of all the numbers that can be expressed as $\frac{n}{m}$ where $n$ and $m$ are two integers and $m$ is not zero)

  OPERATIONS:
  Constructors
    - CreateFraction
      Inputs: two integers (n,m)
      Output: the fraction n/m, if m is not zero; otherwise NULL
      Effect: creates a new object of type Fraction and initializes it to the value n/m

  Destructors:
    - DestroyFraction
      Inputs: one object of type Fraction
      Outputs: none
      Effect: the object given as input is destroyed

  Inspectors:
    - EqualFractions
      Inputs: two objects of type fraction (p,q)
      Outputs: a boolean value
      Effect: returns TRUE if the fractions p and q have the same value, FALSE otherwise
    - GreaterThanFraction
      Inputs: two objects of type fraction (p,q)
      Output: a boolean value
      Effect: returns TRUE if p has a value greater than q
    - PrintFraction
      Input: one object of type fraction
      Output: none
      Effect: print the value of the input fraction on the screen

  Modifiers
    - AddFractions
      Inputs: two objects of type fraction
      Output: one object of type fraction
      Effect: a new fraction is created, whose value is the sum of the values of the two fractions given in input
    - MultiplyFractions
      Inputs: two objects of type fraction
      Output: one object of type fraction
      Effect: a new fraction is created, whose value is the product of the values of the two fractions given in input
}

Figure 1: Specification of the Fraction ADT