

CS 213

Data Structures

- Data Abstraction
- C++ Review
(Struct vs. Class)

1

Data type, Data structure and ADT

Data Type

- set of values that the variable may assume
- set of data and operations that can be performed on the data
- vary from language to language
- e.g integer, Boolean, string

2

1. **Atomic Data Type** - having identical properties

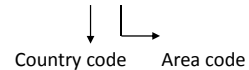
Example :

integer values : ... , -2, -1, 0, 1, 2, ...
operations : *, +, /, ++, --

character values : .., 'A', 'B', .., 'a', 'b', ...
operations : +, -, ...

2. **Composite Data Type** - can be broken out into subfield that have meaning

Example : telephone number 66 2 7654321



3

Definitions

Data Structure

- collections of variables, possibly of several different *data types* connected in various way

Example :

array - values can be of any one type

record - values can be possibly dissimilar

4

Definitions

Abstract Data Type (ADT)

- mathematical model with a collection of operations defined on that model

Example : Set

operations -> MAKENULL(A)
 -> UNOIN(A,B,C)
 -> SIZE (A)

5

Why ADT ?

- It 's impossible to design a programming language that already contains the data types of all imaginable applications
- ADT is designed and implemented by programmer
- Defining an ADT is to define not just a data object but also the operations performed on them

6

Data Abstraction

- The process of defining Abstract Data Type which concentrates on the essential properties of the data and leave all implementation details until later

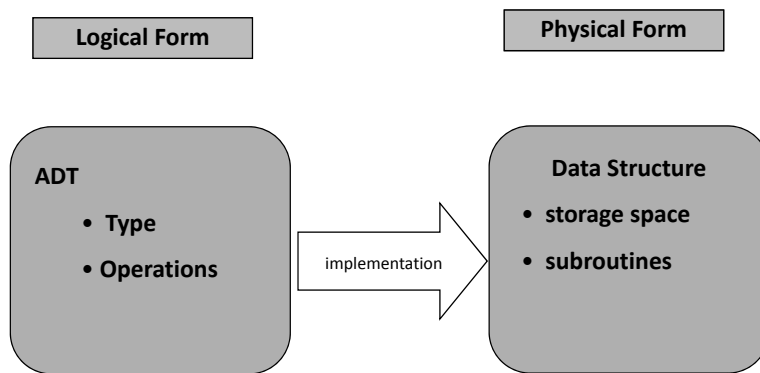
7

Programming Development takes place in two stages :

- Data Abstraction and definition of an algorithm
- The translation of the former into programming language
 - the mapping of the ADT into data type of the implementation language
 - the translation of the abstract operations into functions or procedures

8

ADT and Data Structure



9

Example : Rational Number

- A rational number is a number that can be expressed as the quotient of two integers.
- Operations - creation of a rational number
 - addition
 - multiplication
 - testing for equality

10

ADT Rational

```

/* value definition*/
abstract typedef <integer, integer> RATIONAL;
condition RATIONAL[1] != 0;

/* operator definition*/
abstract RATIONAL makerational(a,b)
int a,b;
precondition b != 0;
postcondition makerational[0] == a;
                 makerational[1] == b;
  
```

11

```

abstract RATIONAL add(a,b)           /* written a + b */
RATIONAL a,b;
postcondition add[1] == a[1] * b[1];
                 add[0] == a[0] * b[1] + b[0] * a[1];

abstract RATIONAL mult(a,b)          /* written a * b */
RATIONAL a,b;
postcondition mult[0] == a[0] * b[0];
                 mult[1] == a[1] * b[1];

abstract RATIONAL equal(a,b)         /* written a == b */
RATIONAL a,b;
postcondition equal == (a[0] * b[1] == b[0] * a[1]);
  
```

12

Rational Implementation

Two approaches

1. represents ADT with **struct** in C and transforms its operations using C **functions**
2. represents ADT and its operations with C++ **class**

13

Structs

- A **struct** holds data, like an array
- Each unit of data in a struct is called a **data member** (or **member**)
- In a struct, each data member can have a different data type

14

1. Representing the *Rational* structure

We can represent a rational number using structures as follows;

```
struct rational {  
    int numerator;  
    int denominator;  
};
```

A rational *r* is declared by

```
struct rational r;
```

15

1. Representing the *Rational* structure (cont.)

An alternative way of declaring is:

```
typedef struct {  
    int numerator;  
    int denominator;  
} RATIONAL;
```

A rational *r* is declared by

```
RATIONAL r;
```

16

Reduced Rational Number

Define a **reduced rational number** as a rational number for which there is no integer that evenly divides both the denominator and the numerator.

Example:

$\frac{1}{2}$, $\frac{2}{3}$, and $\frac{10}{1}$ are all reduced.

$\frac{2}{4}$ reduced to lowest terms is $\frac{1}{2}$
so $\frac{2}{4}$ is equal to $\frac{1}{2}$

17

Euclid's Algorithm

1. Let a be the larger of the *numerator* and *denominator* and let b be the smaller.
2. Divide b into a , finding a quotient q and a remainder r (that is, $a = q * b + r$).
3. Set $a = b$ and $b = r$.
4. Repeat steps 2 and 3 until b is 0.
5. Divide both the *numerator* and the *denominator* by the value of a .

18

Example :

Let us reduce $\frac{8}{36}$ to its lowest terms.

Step 0	numerator = 8	denominator = 36
Step 1	a = 36 b = 8	
Step 2	a = 36 b = 8	q = 4 r = 4
Step 3	a = 8 b = 4	
Step 4 and 2	a = 8 b = 4	q = 2 r = 0
Step 3	a = 4 b = 0	
Step 5	$\frac{8}{4} = 2$	$\frac{36}{4} = 9$

Thus $\frac{8}{36}$ in lowest term is $\frac{2}{9}$.

19

Function to reduce a rational number

```
void reduce (struct rational *inrat, struct rational *outat)
{
    int a, b, rem;
    if (inrat->numerator > inrat->denominator) {
        a = inrat->numerator;
        b = inrat->denominator;
    } /* end if */
    else {
        a = inrat->denominator;
        b = inrat->numerator;
    } /* end else */
}
```

20

```

while (b != 0) {
    rem = a % b;
    a = b;
    b = rem;
} /* end while */
outrat->numerator = inrat->numerator / a;
outrat->denominator = inrat->denominator / a;
} /* end reduce */

```

21

Implement *equal* Function

```

#define TRUE 1
#define FALSE 0

int equal (struct rational * rat1, struct rational *rat2)
{ struct rational r1, r2;
  reduce(rat1, &r1);
  reduce(rat2, &r2);
  if (r1.numerator == r2.numerator &&
      r1.denominator == r2.denominator)
    return(TRUE);
  return(FALSE);
} /* end equal */

```

22

Classes in C++

- A **class** embodies the concept of abstract data type by defining
 - the set of values of a given type and
 - the set of operations that can be performed on those values
- A variable of a class type (or an instance) is known as an **object**
- The operations on that type are called **methods**

23

- If object A invokes a method *m* on another object B, we say
 - “A is sending message *m* to B”
- After B get the message, it may carry out a transformation in response to that message

24

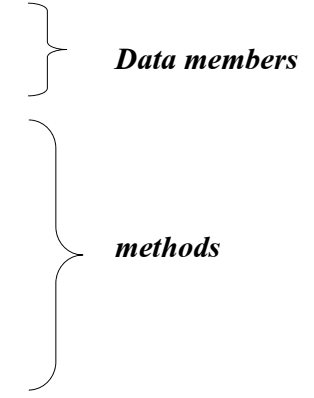
The C++ syntax for *class*

```
class name {  
    private :  
        // hidden features go here  
  
    public :  
        // visible features go here  
};
```

25

2. Representing the *Rational* class

```
class Rational {  
    long numerator;  
    long denominator;  
    void reduce(void);  
public :  
    Rational add(Rational);  
    Rational mult(Rational);  
    Rational divide(Rational);  
    int equal(Rational);  
    void print(void);  
    void setrational(long, long);  
}
```



Data members

methods

26

public and *private* labels

Determine the visibility of the class members

public

- A member that is ***public*** may be accessed by any methods in any classes

private

- A member that is ***private*** may only be accessed by methods in its class
- By default, the members defined at the beginning of a class definition are ***private***

27

Information Hiding

- By using ***private*** data members, we can change the internal representation of the object without having an effect on other parts of the program that use the object
- The user of the class do not need to know internal details of how the class is implemented.

28

Maintenance

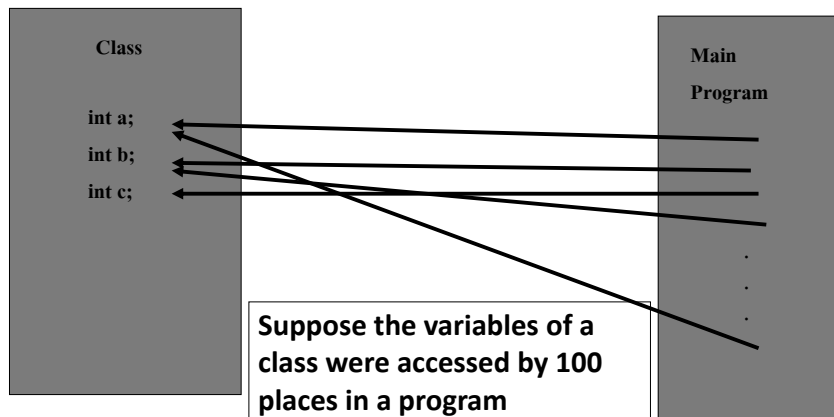
- Programs have to be maintained (modified) to keep up with the changes
- When programs are maintained, the data members sometimes have to change
- **Private** data members make it easier to maintain a program

30

- We do not want outsiders to manipulating either the **rational** or **denominator** numbers.
- They can be accessed only by the **public** methods which are the interface for the class
- For example, the method **reduce()** which is the function to reduce the internal representations of the **Rational** to lowest term.
- The outside world has no cause to call **reduce()** because the other methods (**setrational**, **add**, **mult** and **devide**) will automatically ensure that the resulting number is in reduced form by calling **reduce()** internally.

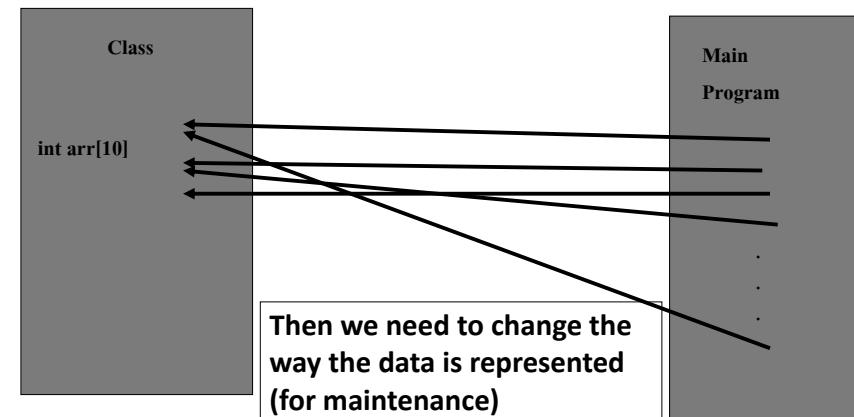
29

If Data Members were Accessed Directly...



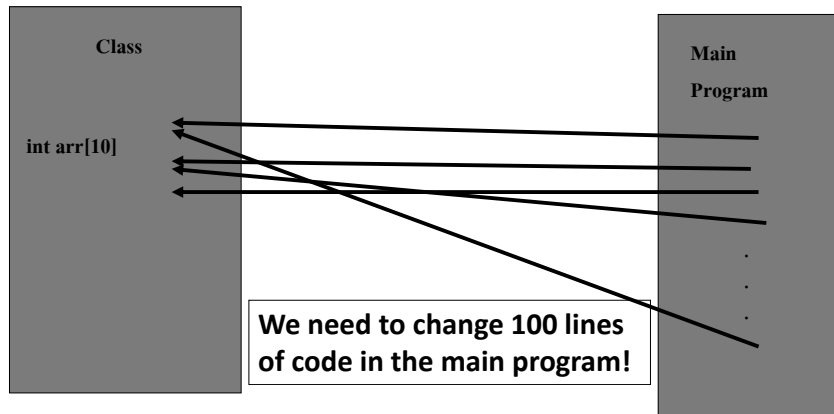
31

If Data Members were Accessed Directly... (cont.)



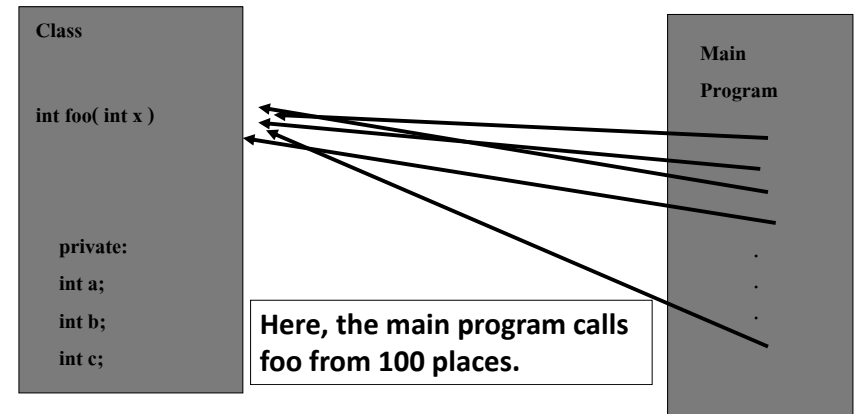
32

If Data Members were Accessed Directly... (cont.)



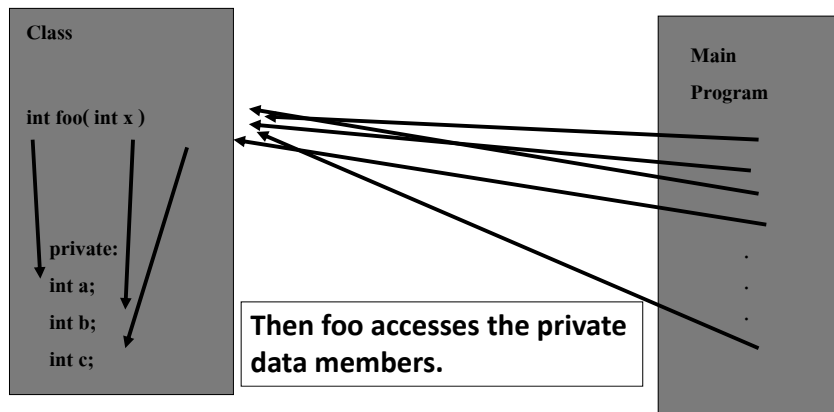
33

Data Members Should Be Private



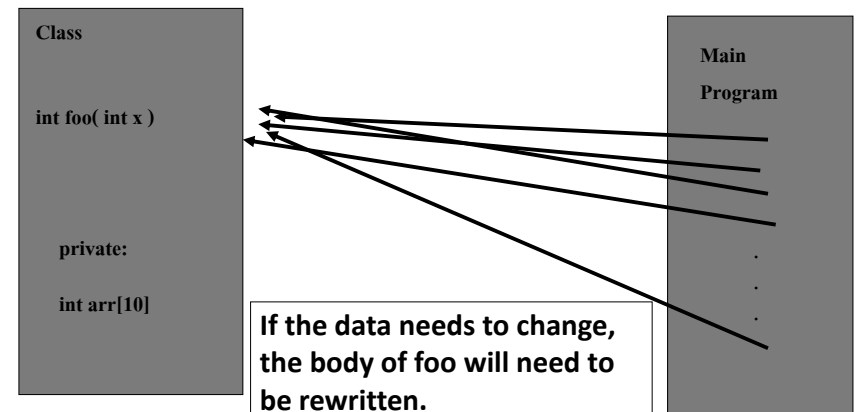
34

Data Members Should Be Private (cont.)



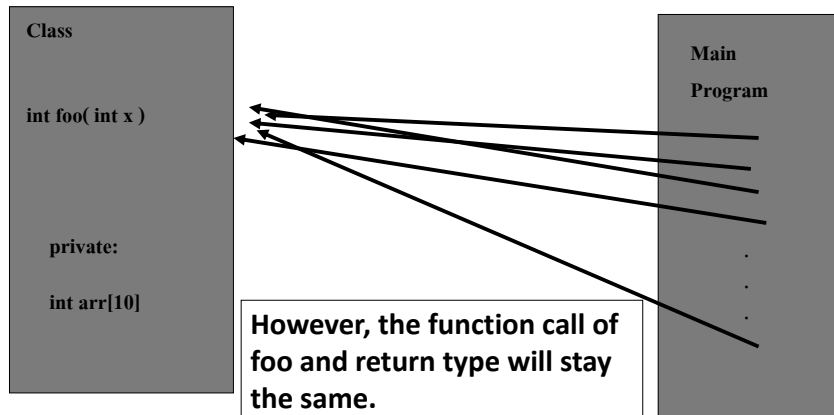
35

Data Members Should Be Private (cont.)



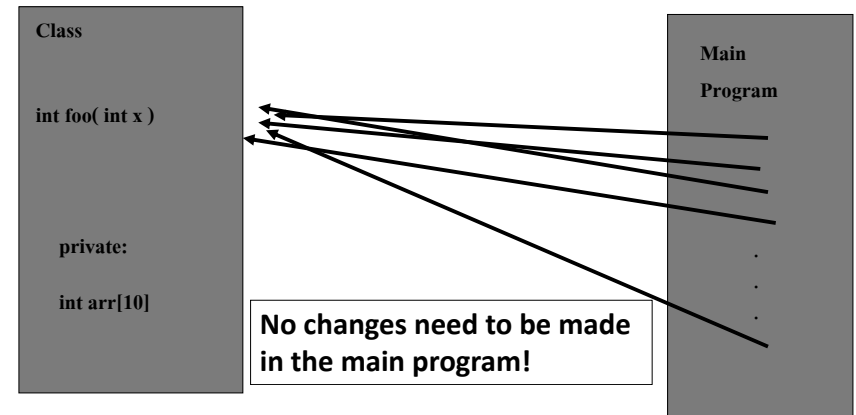
36

Data Members Should Be Private (cont.)



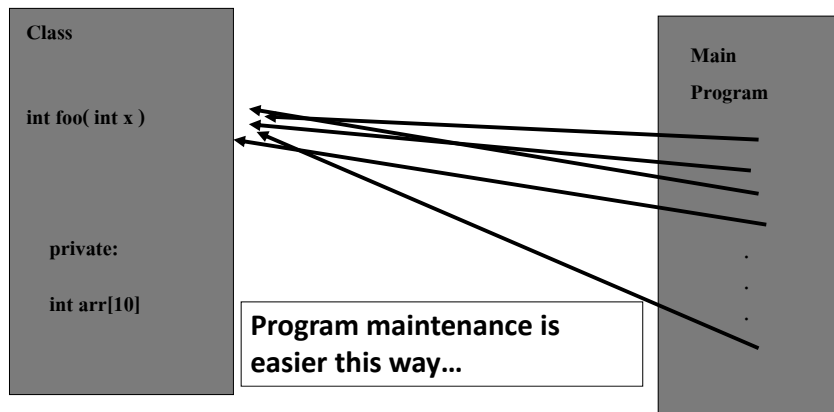
37

Data Members Should Be Private (cont.)



38

Data Members Should Be Private (cont.)



39

Implement the *Rational* methods

1. Within the declaration of class

```
class Rational {
    long numerator;
    ....
public :
    Rational add(Rational);
    ....
    void print(void);
    void setrational(long n, long d)
    {
        if (d == 0)
            cerr << "ERROR: denominator may not be zero ";
        numerator = n;
        denominator = d;
        reduce();
    } // end setrational
} // end class Rational
```

Private members can be referenced directly from the method of the class

40

- When **setrational** is called,
`opnd1.setrational(int1, int2);`
- References to **numerator** and **denominator** within **setrational** will refer to **opnd1.numerator** and **opnd1.denominator**
- And the call **reduce** is to **opnd1.reduce()**

41

Implement the *Rational* methods

2. Outside the declaration of class

- declare the class definition in header file
- implement the class methods in “.cpp”
file which includes its header file in
#include statement

42

```
// -----Rational.cpp -----
#include "Rational.h"
void Rational :: setrational (long n, long d)
{
    if (d == 0)
        cerr << "ERROR: denominator may not be zero ";
    numerator = n;
    denominator = d;
    reduce();           // reduce to the lowest term
} // end setrational

Rational Rational : : add(Rational r)
{
    .....
}
```

- The notation “**Rational::**” introduces a scope and specifies that the function **setrational** being defined is the method of class **Rational**

43

Using class *Rational*

- Suppose the **class Rational** were defined in “rational.h”

```
Rational opnd1, opnd2, result ;           //declare Rational objects
opnd1.setrational(2,5);                   // set the data  in Rational object
opnd2.setrational(1,5);

result = opnd1.add(opnd2);                 // send the message add to opnd1
result.print();

result = opnd1.mult(opnd2);               // send the message mult to opnd1
result.print();
```

44

Overloading

- C++ allows function names to be **overload**.
- The same function name can apply to different functions if their parameters are of different types

45

Example : we can define another method, **add** (to add a rational with an integer in class **Rational** by including the below in public section of class definition

Rational add(long);

```
Rational Rational::add (Rational r)
{
    .....
}
Rational Rational::add(long i)
{
    Rational r;
    r.setrational(i, 1);
    return add(r);
}
```

46

Inheritance

- Consider the case of division, we can write a method **Rational divide(int)** to compute r/i , but how do we compute i/r ?
- Integers are a form of rationals. We can therefore represent every integer by a rational.
- We do this by defining a new class **Integer** which **inherits** the members of the class **Rational**.

47

- In the new class, we must make sure that the denominator is always 1
- We define two versions of **setrational**.

```
class Integer::public Rational {
    public :
        void setrational(long, long);
        void setrational(long);
}
```

48

- The class *Rational* is called the **base class** of the class *Integer*.
- In order for the method of **inherited class** (*setrational*) to access the *Rational* members *numerator* and *denominator*, those members must be defined as **protected** rather than **private**

```
class Rational {
    protected
        long numerator;
        .....
    public
        .....
};
```

49

```
void Integer::setrational(long num, long denom)
{
    if (denom != 1)
        cerr << "ERROR: non-integer assigned to Integer variable";
    numerator = num;
    denominator = 1;
}

void Integer::setrational(long num)
{
    numerator = num;
    denominator = 1;
}
```

50

Constructor

- Suppose *r* is a *Rational* and *i* is an integer, the followings are valid

```
r.add(i);
i.add(r);
r.divide(i);
i.divide(r)
```

51

- A special method of class that invoked whenever an object of that class is created
- A **constructor** always is named with the same name as the class itself.
- More than one constructor can be written for a class; constructors differ by the number and types of parameters
- A constructor with no parameters is called a **default constructor**
- If a constructor is not written by the programmer, the compiler supplies a default constructor which does nothing

52

- We can replace *setrational* with **constructor**

Example : suppose that we include the following 3 members in class definition

```
Rational(void);
Rational(long);
Rational(long, long);
```

53

```
Rational::Rational(void)
{
    // assume the rational number is 0;
    numerator = 0;
    denominator = 1;
}
Rational::Rational(long i)
{
    numerator = i;
    denominator = 1;
}
Rational::Rational(long num, long denom)
{
    numerator = num;
    denominator = denom;
}
```

54

Struct vs. Class

- Then when we declare an object to be a *Rational*, the appropriate constructor is invoked

```
Rational r;
```

//the first constructor is invoked initialize r to the rational (0/1)

```
Rational r(3);
```

//the second constructor is invoked initialize r to the rational (3/1)

```
Rational r(2,5);
```

//the second constructor is invoked initialize r to the rational (2/5)

55

- Functions can be placed in a struct, but only when necessary
- The public and private keywords can be left out of a class (rare).
- The public and private keywords can be placed into a struct (rare).
- So what is the difference between a struct and a class?

56

Struct vs. Class (cont.)

- If public and private are not used in a struct, all data members are public by default.
- If public and private are not used in a class, all data members are private by default.

57

Conventions

- By convention, we use structs when we want all data members to be public
 - structs are typically defined and used within the client's program, not in a separate file
 - typically used for records of information
- By convention, we use classes when we want all data members to be private (for maintenance purposes)

58