The Class Construct

Defining objects with attributes and behavior

Class Types

- Class construct
  - Allows programmers to define new data types for representing information
  - Class type objects can have both attribute components and behavior components
  - Provides object-oriented programming in C++
- Example we shall consider is
  - RectangleShape

Terminology

- Client
  - Program using a class
- Object behaviors
  - Realized in C++ via member functions (methods)
    - RectangleShapes can be drawn or resized
- Object attributes
  - Are known as data members in C++
    - RectangleShapes have width, height, position, color

Member Functions

- Provide a controlled interface to data members and object access and manipulation
  - Create objects of the class
  - Inspect, mutate, and manipulate object of the class
  - Can be used to keep data members in a correct *state*
    - SetSize()
    - SetColor()
    - Draw()

Member Functions

- *Constructors*
  - Member functions that initialize an object during its definition

    ```
    RectangleShape R(W, x, y, c, w, h);
    ```

    - Constructors do not have a type
      - Considered superfluous

Member Functions

- *Inspectors*
  - Member functions that act as a messenger that returns the value of an attribute
  - Example
    - RectangleShapes have an inspector GetColor()

```
color CurrColor = R.GetColor();
```

Member Functions

- *Mutators*
  - Changes the value of an attribute
  - Example
    - RectangleShapes have a mutator SetColor()

```
R.SetColor(Black);
```

Member Functions

- *Facilitators*
  - Causes an object to perform some action or service
  - Example
    - RectangleShapes have a facilitator Draw()

```
R.Draw();
```

Our Goal

- Well-defined representations that allow objects to be created and used in an intuitive manner
  - User should not have to bother with unnecessary details
- Example
  - programming a microwave to make popcorn should not require a physics course

Golden Rule

- Use information hiding and encapsulation to support integrity of data
  - Put implementation details in a separate module
    - Implementation details complicate the class declarations
  - Data members are private so that use of the interface is required
    - Makes clients generally immune to implementation changes

Another Golden Rule

- Keep it simple – class minimality rule
  - Implement a behavior as a nonmember function when possible
  - Only add a behavior if it is necessary

Abstract Data Type

- Well-defined and complete data abstraction using the information-hiding principle
- ADT: “a class of objects whose logical behavior is defined by a set of values and a set of operations"

Rational Number Review

- Rational number
  - Ratio of two integers: a/b
    - Numerator over the denominator

- Standard operations
  - Addition

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+bc}{bd}$$

  - Subtraction

$$\frac{a}{b}-\frac{c}{d}=\frac{ad-bc}{bd}$$

Multiplication

$$\frac{a}{b}*\frac{c}{d}=\frac{ac}{bd}$$

Division

$$\frac{a}{b}/\frac{c}{d}=\frac{ad}{bc}$$

Abstract Data Type

- Consider

```
Rational a(1,2);    // a = 1/2
Rational b(2,3);    // b = 2/3
cout << a << " + " << b << " = " << a + b;
Rational s;         // s = 0/1
Rational t;         // t = 0/1
cin >> s >> t;
cout << s << " * " << t << " = " << s * t;
```

- Observation
  - Natural look that is analogous to fundamental-type arithmetic objects

Rational Attributes

- A numerator and denominator
  - Implies in part a class representation with two private **`int`** data members
    - NumeratorValue and DenominatorValue

Rational Public Behaviors

- Rational arithmetic
  - Addition, subtraction, multiplication, and division
- Rational relational
  - Equality and less than comparisons
    - Practice rule of class minimality

Rational Public Behaviors

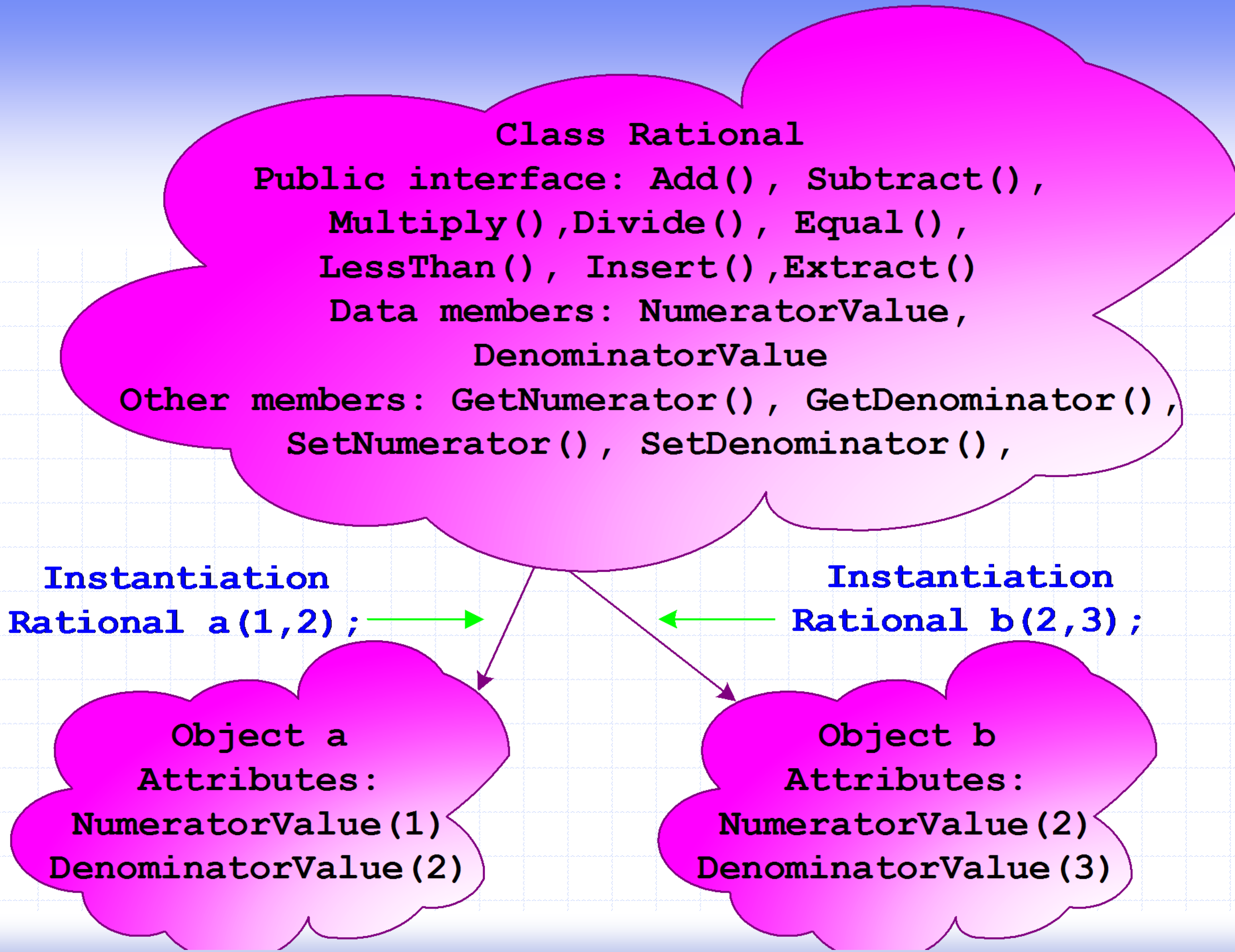
- Construction
  - Default construction
    - Design decision 0/1
  - Specific construction
    - Allow client to specify numerator and denominator
  - Copy construction
    - Provided automatically
- Assignment
  - Provided automatically
- Insertion and extraction (output and input)

Non-Public Behaviors

- Inspection and mutation of data members
  - Clients deal with a `Rational` object!

Auxiliary Behaviors

- Operations (necessarily public)
  - Arithmetic, relational, insertion, and extraction operations
    - Provides the natural form we expect
      - Class definition provides a functional form that auxiliary operators use


Class Rational
Public interface: Add(), Subtract(),
Multiply(),Divide(), Equal(),
LessThan(), Insert(),Extract()
Data members: NumeratorValue,
DenominatorValue
Other members: GetNumerator(), GetDenominator(),
SetNumerator(), SetDenominator(),
Instantiation
Rational a(1,2);
Instantiation
Rational b(2,3);
Object a
Attributes:
NumeratorValue(1)
DenominatorValue(2)
Object b
Attributes:
NumeratorValue(2)
DenominatorValue(3)

Library Components

- Rational.h
  - Class definitions and library function prototypes
- Rational.cpp
  - Implementation source code – member and auxiliary function definitions
    - Auxiliary functions are assisting global functions that provide expected but non-member capabilities
- Rational.obj
  - Translated version of Rational.cpp (linkable)
- Rational.lib
  - Library version of Rational.obj that is more readily linkable

MyProgram.cpp

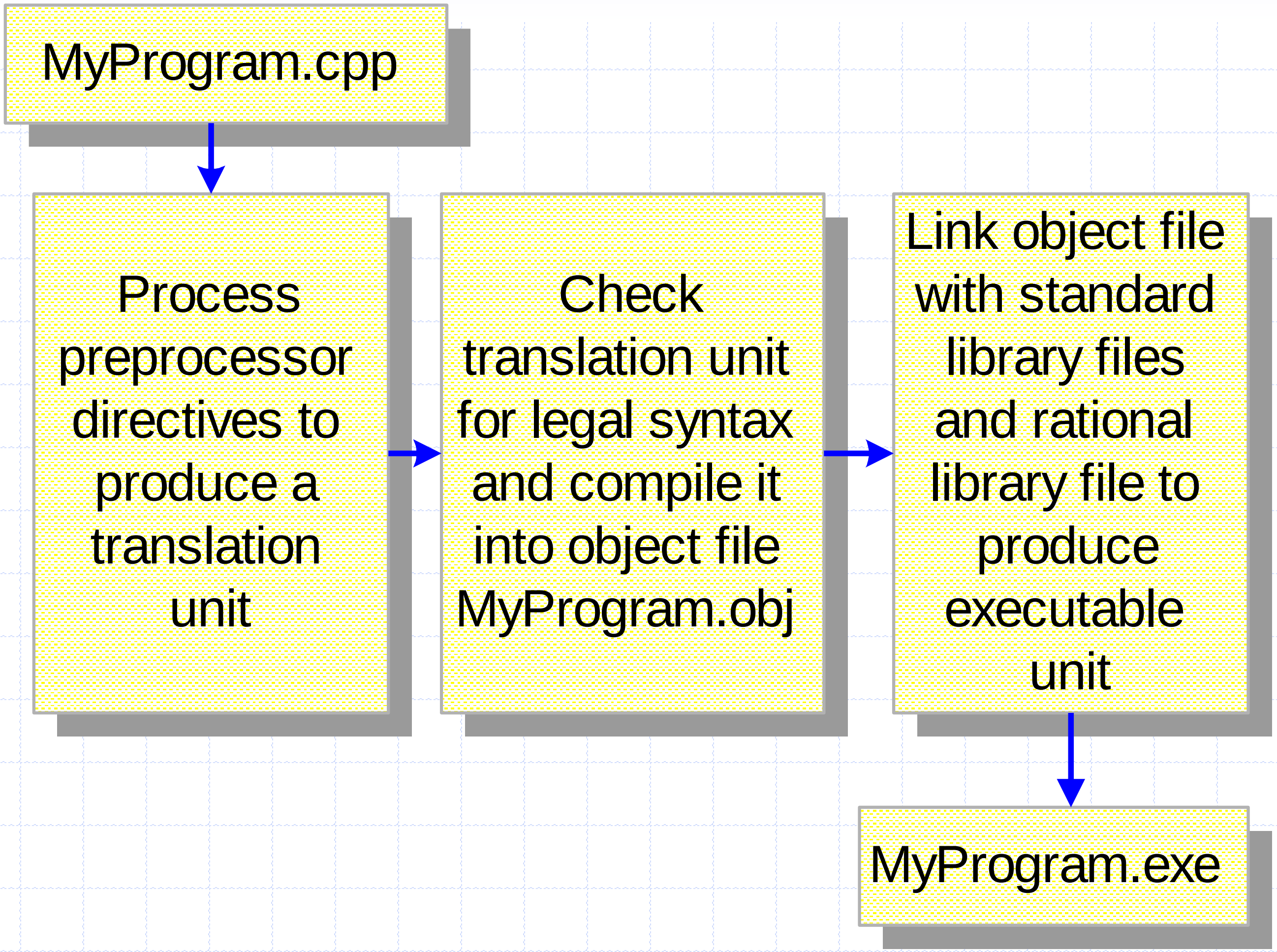
Making use of the Rational class. The header file provides access to the class definition and to auxiliary function prototypes. The header file does not provide member and auxiliary definitions

```
#include <iostream>
using namespace std;
#include "rational.h"
int main() {
   Rational r;
   Rational s;
   cout << "Enter two rationals(a/b): ";
   cin >> r >> s;
   Rational Sum = r + s;
   cout << r << " + " << s << " = " << Sum;
   return 0;
}
```

Producing MyProgram.exe

- Preprocessor combines the definitions and prototypes in iostream and rational headers along with MyProgram.cpp to produce a compilation unit
  - Compiler must be told where to look for Rational.h
- Compiler translates the unit and produces MyProgram.obj
- Compiler recognizes that MyProgram.obj does not contain actual definitions of Rational constructor, +, >>, and <<
- Linker is used to combine definitions from the Rational library file with MyProgram.obj to produce MyProgram.exe
  - Compiler must be told where to find the Rational library file

Producing MyProgram.exe



Rational Header File Overview

- File layout
  - Class definition and library prototypes nested within preprocessor statements
    - Ensures one inclusion per translation unit
  - Class definition precedes library prototypes

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
  // …
} ;
// library prototypes …
#endif
```

Class Rational Overview

```
class Rational {        // from rational.h
   public:
      // for everybody including clients
   protected:
      // for Rational member functions and for
      // member functions from classes derived
      // from rational
   private:
      // for Rational member functions
} ;
```

Rational Public Section

```
public:
   // default constructor
   Rational();
   // specific constructor
   Rational(int numer, int denom = 1);
   // arithmetic facilitators
   Rational Add(const Rational &r) const;
   Rational Multiply(const Rational &r) const;
   // stream facilitators
   void Insert(ostream &sout) const;
   void Extract(istream &sin);
```

Rational Protected Section

```
protected:
    // inspectors
    int GetNumerator() const;
    int GetDenominator() const;
    // mutators
    void SetNumerator(int numer);
    void SetDenominator(int denom);
```

Rational Private Section

```
private:
   // data members
   int NumeratorValue;
   int DenominatorValue;
```

Auxiliary Operator Prototypes

```
// after the class definition in rational.h

Rational operator+(
   const Rational &r, const Rational &s);

Rational operator*(
   const Rational &r, const Rational &s);

ostream& operator<<(
   ostream &sout, const Rational &s);

istream& operator>>(istream &sin, Rational &r);
```

Auxiliary Operator Importance

```
Rational r;
Rational s;
r.Extract(cin);
s.Extract(cin);
Rational t = r.Add(s);
t.Insert(cout);
```

```
Rational r;
Rational s;
cin >> r;
cin >> s;
Rational t = r + s;
cout << t;
```

- Natural look

Const Power

```
const Rational OneHalf(1,2);
cout << OneHalf;                    // legal
cin >> OneHalf;                     // illegal
```

Rational Implementation

```
#include <iostream>        // Start of rational.cpp
#include <string>
using namespace std;
#include "rational.h"

// default constructor
Rational::Rational() {
   SetNumerator(0);
   SetDenominator(1);
}
```

Which objects are being referenced?

- Example

```
      Rational r;          // r = 0/1
```

Remember

- Every class object
  - Has its own data members

  - Has its own member functions
    - When a member function accesses a data member
      - By default the function accesses the data member of the object to which it belongs!

        - No special notation needed

Remember

- Auxiliary functions
  - Are not class members

  - To access a public member of an object, an auxiliary function must use the dot operator on the desired object

    object.member

Specific Constructor

```
// (numer, denom) constructor
Rational::Rational(int numer, int denom) {
   SetNumerator(numer);
   SetDenominator(denom);
}
```

- Example

```
Rational t(2,3);    // t = 2/3

Rational u(2);      // u = 2/1 (why?)
```

Inspectors

```
int Rational::GetNumerator() const {
   return NumeratorValue;
}

int Rational::GetDenominator() const {
   return DenominatorValue;
}
```

Which object is being referenced?

Why the const?

- `Where' are the following statements legal?

```
int a = GetNumerator();
int b = t.GetNumerator();
```

Numerator Mutator

```
void Rational::SetNumerator(int numer) {
   NumeratorValue = numer;
}
```

Why no const?

- Where are the following statements legal?

```
SetNumerator(1);

t.SetNumerator(2);
```

Denominator Mutator

```
void Rational::SetDenominator(int denom) {
   if (denom != 0) {
        DenominatorValue = denom;
   }
   else {
        cerr << "Illegal denominator: " << denom
         << "using 1" << endl;
        DenominatorValue = 1;
   }
}
```

- Example

```
t.SetDenominator(5);
```

Addition Facilitator

```
Rational Rational::Add(const Rational &r) const {
   int a = GetNumerator();
   int b = GetDenominator();
   int c = r.GetNumerator();
   int d = r.GetDenominator();
   return Rational(a*d + b*c, b*d);
}
```

- Example

```
cout << t.Add(u);
```

Multiplication Facilitator

```
Rational Rational::Multiply(const Rational &r)
 const {
   int a = GetNumerator();
   int b = GetDenominator();
   int c = r.GetNumerator();
   int d = r.GetDenominator();
   return Rational(a*c, b*d);
}
```

- Example

```
t.Multiply(u);
```

Insertion Facilitator

```
void Rational::Insert(ostream &sout) const {
   sout << GetNumerator() << '/' << GetDenominator();
   return;
}
```

- Example

  ```
  t.Insert(cout);
  ```

- Why is `sout` a reference parameter?

Basic Extraction Facilitator

```
void Rational::Extract(istream &sin) {
   int numer;
   int denom;
   char slash;
   sin >> numer >> slash >> denom;
   assert(slash == '/');
   SetNumerator(numer);
   SetDenominator(denom);
   return;
}
```

- Example

```
t.Extract(cin);
```

Auxiliary Arithmetic Operators

```
Rational operator+(
 const Rational &r, const Rational &s) {
  return r.Add(s);
}

Rational operator*(
 const Rational &r, const Rational &s) {
  return r.Multiply(s);
}
```

- Example

```
cout << (t + t) * t;
```

Auxiliary Insertion Operator

```
ostream& operator<<(
 ostream &sout, const Rational &r) {
   r.Insert(sout);
   return sout;
}
```

- Why a reference return?
- Note we can do either

```
      t.Insert(cout); cout << endl;  // unnatural
      cout << t << endl;             // natural
```

Auxiliary Extraction Operator

```
// extracting a Rational
istream& operator>>(istream &sin, Rational &r) {
   r.Extract(sin);
   return sin;
}
```

- Why a reference return?
- We can do either

```
      t.Extract(cin);                    // unnatural
      cin >> t;                          // natural
```

What's Happening Here?

- Suppose the following definitions are in effect

```
Rational a(2,3);
Rational b(3,4);
Rational c(1,2);
```

- Why do the following statements work

```
Rational s(a);
Rational t = b;
c = a;
```

- C++ has automatically provided us a copy constructor and an assignment operator

Copy Construction

- Default copy construction
  - Copy of one object to another in a bit-wise manner
    - The representation of the source is copied to the target in a bit-by-bit manner
  - This type of copy is called *shallow copying*
- Class developers are free to implement their own copy constructor
- Rational does need a special one, but we will define one for the experience

A Rational Copy Constructor

```
Rational::Rational(const Rational &r) {
   int a = r.GetNumerator();
   int b = r.GetDenomiator();

   SetNumerator(a);
   SetDenominator(b);
}
```

```
      Rational s(a);
      Rational t = b;
```

Gang Of Three

- If it is appropriate to define a copy constructor then
  - Consider also defining
    - Assignment operator
      - Copy source to target and return target
        - A = B = C
    - Destructor
      - Clean up the object when it goes out of scope

- We give the name *Gang of three* to the
  - Copy constructor, assignment operator, and the destructor

A Rational Assignment Operator

```
Rational& Rational::operator =(const Rational &r) {
   int a = r.GetNumerator();
   int b = r.GetDenomiator();

   SetNumerator(a);
   SetDenominator(b);

   return *this;
}
```

*this is C++ syntax for the object whose member function was invoked

```
      a = b;
      a = b = c;
```

Rational Destructor

```
Rational::~Rational() {
   // nothing to do
}
```