Function Overloading

C++ enables several functions of the same name to be defined

- as long as those functions' signatures (i.e., arguments' type, order, and number) are different.

```
#include <iostream.h>
int square(int x) {return x * x};

double square(double y) {return y * y};

main() {
    cout << "The square of 7 is " << square(7)
         << "The square of 7.5 is " << square(7.5);
}
```

Compare this to macros in C

C Macros

```
#define INVALIDSQUARE(x) x * x

#define SQUARE(x) (x) * (x)

cout << SQUARE(7);
    is expanded to . . .

cout << SQUARE(2 + 3);
    is expanded to . . .
```

Why invalid?

+ C++ has inline functions that also does type checking of args:
  inline int square(int x) { return x * x; }

+ C++ has function templates that does a similar job but also do type checking of args.

Function Templates

Overloaded functions are normally used to perform similar operations on different types of data; this may be performed more compactly and conveniently using template functions.

```
template <class T>
void printArray(T *array, const int count)
{
    for (int i=0; i<count; i++)
        count << array[i] << " ";
    cout << '\n';
}
```

When the compiler detects a printArray invocation in client's source code, the type of first arg. in caller function is subsituted for T.


```
main()
{
    const int aCount = 5, bCount = 7, cCount = 6;
    int a[aCount] = {1, 2, 3, 4, 5};
    int b[bCount] = {1.1, 2.2., 3.3., 4.4., 5.5, 6.6, 7.7};
    int c[cCount] = "HELLO";

    cout<<"Array a contains: \n";
    printArray(a, aCount);

    cout<<"Array b contains: \n";
    printArray(b, bCount);

    cout<<"Array c contains: \n";
    printArray(c, cCount);
}
```



```
class Rational {
public:
    // constructors
    Rational    ();
    Rational    (int);
    Rational    (int, int);
    Rational    (const Rational &);

    // accessor functions
    int         numerator     () const;
    int         denominator   () const;

    // assignments
    void        operator =    (const Rational &);
    void        operator +=   (const Rational &);

private:
    // data areas
    int         top;
    int         bottom;

    // operation used internally
    void        normalize     ();
};
```

Now we can treat instances of our rational data type as mathematical objects

How to declare new rational numbers?

```
Rational x;         // implicitly activates(calls) default constructor
Rational y(3);      // implicitly activates Rational(int)
Rational y = 3;     // equivalent to the above declaration
Rational z(2, 3);   // implicitly activates Rational(int, int)

Rational t (y);     // implicitly activates Rational(const Rational&)
```

Accessor functions

```
int Rational::numerator() const
{   // return the numerator field of a rational number
    return top;
}

int  Rational:: denominator() const
{   // return the denominator field of a rational number
    return bottom;
}
```

Overloaded Operator Functions

```
Rational operator + (const Rational &left,
                     const Rational &right)
{  // return sum of two rational numbers
   Rational result (
        left.numerator() * right.denominator() +
        right.numerator() * left.denominator() );
   return result;
}
```

Note that, this is not a method of the class

⇩

```
Rational Rational::operator + (const Rational &right)
{  // return sum of two rational numbers
   Rational result (
        this.numerator() * right.denominator() +
        right.numerator() * this.denominator() );
   return result;
}
```

top — *bottom*

Note that, this is a method of the Rational class, and the first arg. is implicitly "this"

Think about implementation of operator unary – and operator <

Member Function Operators

Assignments operators are not defined as simple functions but as member functions in order to be able to access the private data members. Operator = is an operator, not a statement.

```
void Rational::operator = (const Rational &right)
{  // simply copy values from rhs of assignment
   top = right.numerator();
   bottom = right. denominator();
}
void Rational::operator += (const Rational &right)
{  // modify by adding rhs
   top = top * right.denominator() + bottom * right.numerator();
   bottom *= right. denominator();

   // normalize the result, ensuring lowest denom.
   normalize();
}
```

Note that, these are methods of the Rational class, and the first arg. is implicitly "this"

Think about implementation of operator /= function

Constructor Implementation

Note that a constructor does not declare any return type

The heading for a constructor can be followed by a sequence of initializers.

```
Rational::Rational(int numerator) : top(numerator)
{  // by default initialize the denominator to one
   bottom = 1;
}
Rational::Rational(int numerator) : top(numerator), bottom(1)
{
   // no further initialization is necessary
}

Rational::Rational() : top(0), bottom(1)
{
   // no further initialization is necessary
}
```

Default constructor i.e., no args

Copy Constructor

In addition to being invoked in response to a declaration statement, a constructor will also be invoked when it is necessary to copy a data value from one location to another

```
Rational::Rational(const Rational &value)
   : top(value.numerator()), bottom(value.denominator())
{
   // no further initialization is necessary
}
```

Constructors are also used implicitly by the C++ language to define conventions:

```
   Rational x, y;
   . . .
   x = y  * 3;
```

3 converted into a rational number

Streams

There is a standard output stream (ostream) and a standard input stream (istream).

We can overload << and >> operators to define print methods for new classes

```
ostream & operator << (ostream &out, const Rational r)
{  // print representation of rational number on an output stream
   out << r.numerator() << '/' << r.denominator();
   return out;
}
istream & operator >> (istream &in, Rational &r)
{  // read a rational number from an input stream
   int t, b;
   char ch;
   // read top, / character, and bottom (assuming they are consecutive: 3/7)
   in >> t >> ch >> b;
   r = Rational(t, b);
   return in;
}
```