Object Oriented Programming (CS1143)

Week 15

Department of Computer Science

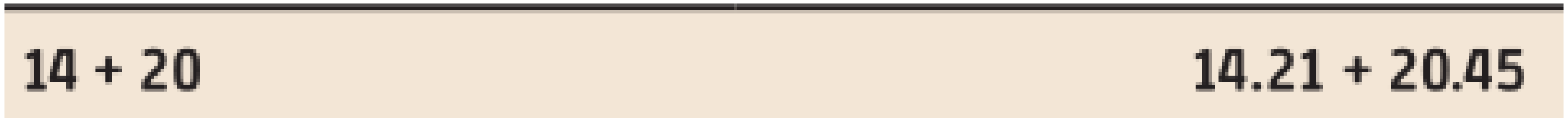
Capital University of Science and Technology (CUST)

Outline

- Operator Overloading
- Unary Operators
- Binary Operators

Operator Overloading

- C++ uses symbols (called operators) to manipulate fundamental data types such as integers and floating points. For example ‘+’ for addition, ‘-’ for subtraction etc.
- Most of these symbols are overloaded to handle several data types. For example, the symbol for the addition operator of two fundamental data types (x + y) can be used to add two values of type int, long, double etc.
- This means that the following two expressions in C++ use the same symbol with two distinct interpretations. The first symbol means add two integers; the second symbol means add two reals.
- Operator overloading means using the same operator for different interpretations.

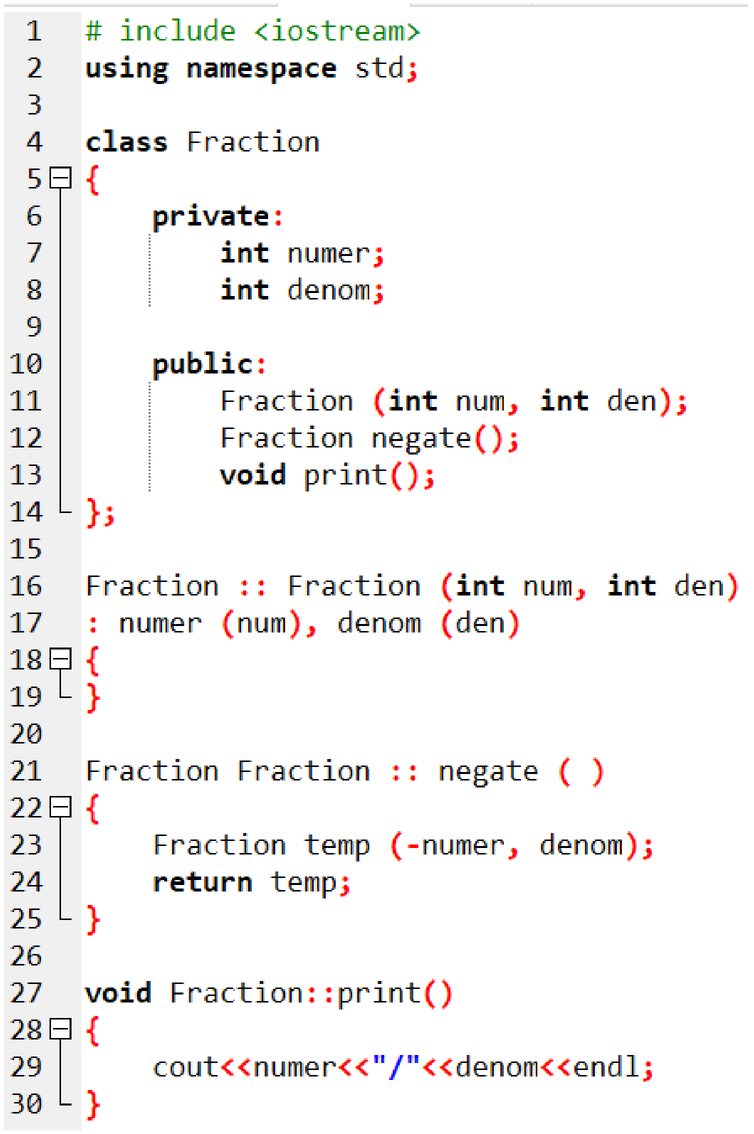


14 + 20	14.21 + 20.45

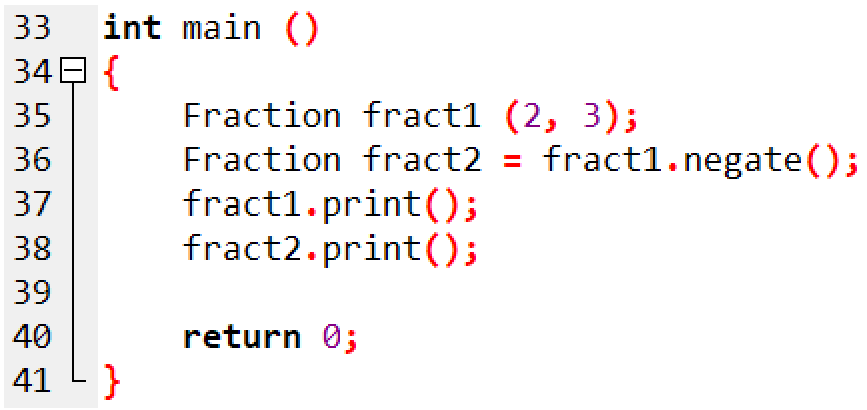
Overloading Operators for User Defined Types

- We know that we can add two objects of fundamental type such as (x + y) when both x and y are variables of type integer or floating point.
- If we make a class Fraction representing fractions (numerator/denominator), we may want to add two fractions in a similar way (fract1 + fract2) .
- In other words we want to treat the user-defined types the same as fundamental types.
- There are two ways to do it.
  - Using functions.
  - Using Operator Overloading.

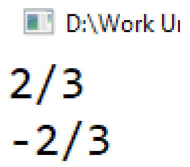
Example: Using Function to change sign



```cpp
# include <iostream>
using namespace std;

class Fraction
{
    private:
        int numer;
        int denom;

    public:
        Fraction (int num, int den);
        Fraction negate();
        void print();
};

Fraction :: Fraction (int num, int den)
: numer (num), denom (den)
{
}

Fraction Fraction :: negate ( )
{
    Fraction temp (-numer, denom);
    return temp;
}

void Fraction::print()
{
    cout<<numer<<"/"<<denom<<endl;
}
```



```cpp
int main ()
{
    Fraction fract1 (2, 3);
    Fraction fract2 = fract1.negate();
    fract1.print();
    fract2.print();

    return 0;
}
```



D:\Work U

```
2/3
-2/3
```

Description

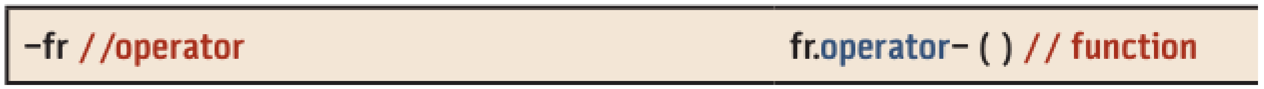
- Inside negate(), we are not changing the host object. The returned object is a copy of the host object after modifications.

Using Operator Overloading

- Instead of using a function call to negate a fraction, we can overload the symbol “-” to do the same thing.
- Overloading is a powerful capability of the C++ language that allows the user to redefine operators for user-defined data types.
- That means writing **- fract1** instead of **fract1.negate().**

Operator Function

- To overload an operator for a user-defined data type, we must write a function named **operator function**, a function that acts as an operator.
- The name of this function starts with the reserved word **operator** and is followed by the symbol of the operator that we want to overload.
  - For example **operator-()**
- After overloading, we can either use the operator itself or the function operator like a normal function.

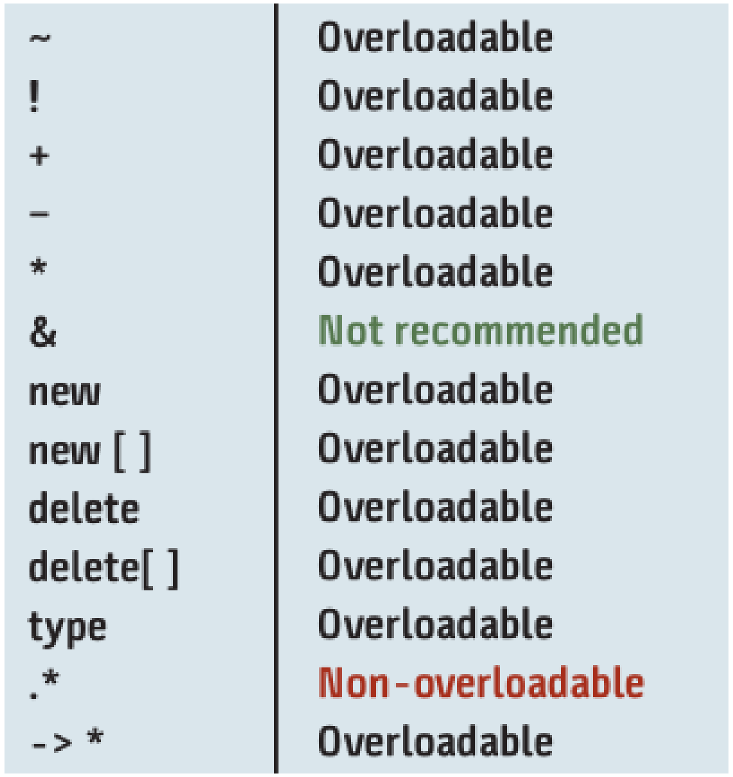


```
-fr //operator                    fr.operator- ( ) // function
```

- The version on the left is more concise and intuitive. The whole purpose of operator overloading is to use the operator itself to mimic the behavior of built-in types.

Overloading Unary Operators

Unary Operators



~	Overloadable
!	Overloadable
+	Overloadable
–	Overloadable
*	Overloadable
&	Not recommended
new	Overloadable
new [ ]	Overloadable
delete	Overloadable
delete[ ]	Overloadable
type	Overloadable
.*	Non-overloadable
-> *	Overloadable

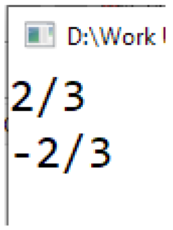
Overloading Unary Operators

- In a unary operator, the only operand is the host object of the operator function.
  - Host object is the one that called the function.
- We have no parameter object. This means that we should only think about two objects: the host object and the returned object.
  - Returned object is the one returned from the function.
- In the following example, we are not changing the host object. The returned object is a new object which has the same data as the host object but after required modifications.

Example: Overloading - operator

```cpp
# include <iostream>
using namespace std;

class Fraction
{
    private:
        int numer;
        int denom;

    public:
        Fraction (int num, int den);
        Fraction operator- ( );
        void print();
};

Fraction :: Fraction (int num, int den)
: numer (num), denom (den)
{
}

Fraction Fraction :: operator- ( )
{
    Fraction temp (-numer, denom);
    return temp;
}

void Fraction::print()
{
    cout<<numer<<"/"<<denom<<endl;
}
```

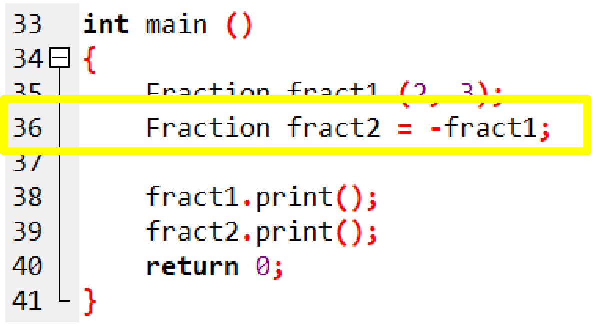
```cpp
int main ()
{
    Fraction fract1 (2, 3);
    Fraction fract2 = -fract1;

    fract1.print();
    fract2.print();
    return 0;
}
```



```
2/3
-2/3
```

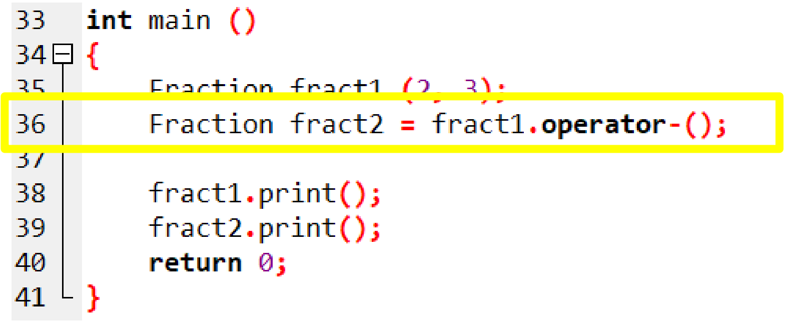
Description

- Because we have an operator function for “-” (line 23), we can call it like we did on line 36.



```
33   int main ()
34   {
35       Fraction fract1 (2, 3);
36       Fraction fract2 = -fract1;
37
38       fract1.print();
39       fract2.print();
40       return 0;
41   }
```

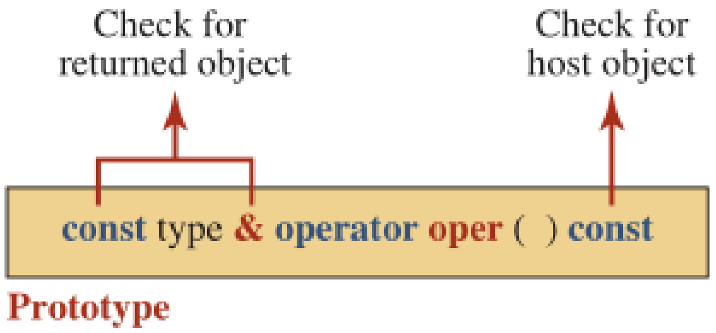
- We can also call this operator function as a normal function as shown below with the same result.

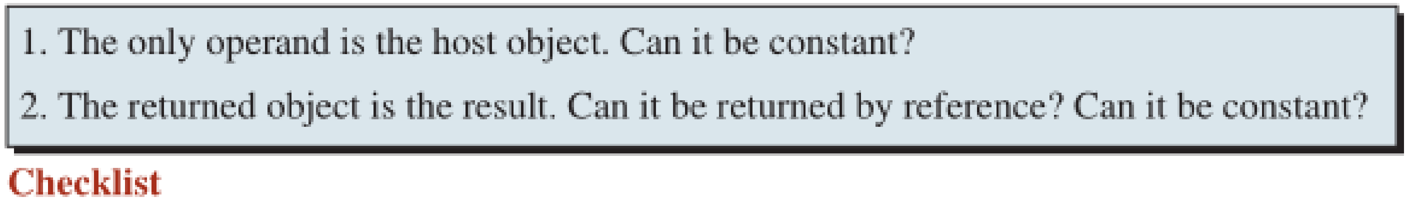


```
33   int main ()
34   {
35       Fraction fract1 (2, 3);
36       Fraction fract2 = fract1.operator-();
37
38       fract1.print();
39       fract2.print();
40       return 0;
41   }
```

- Clearly the first method is much better and that is the whole point of operator overloading.

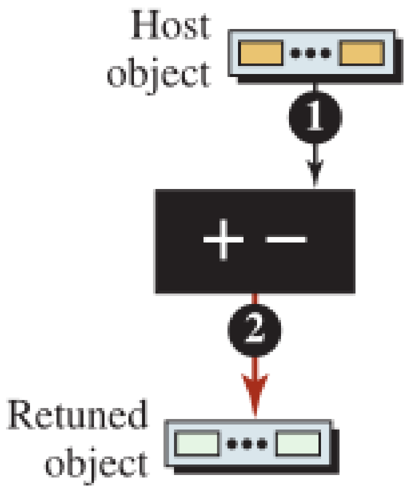
Guidelines for overloading Unary Operators


Check for returned object
Check for host object
const type & operator oper ( ) const
Prototype


1. The only operand is the host object. Can it be constant?
2. The returned object is the result. Can it be returned by reference? Can it be constant?

Checklist

For + or – operators ideally we want the following



Prototye

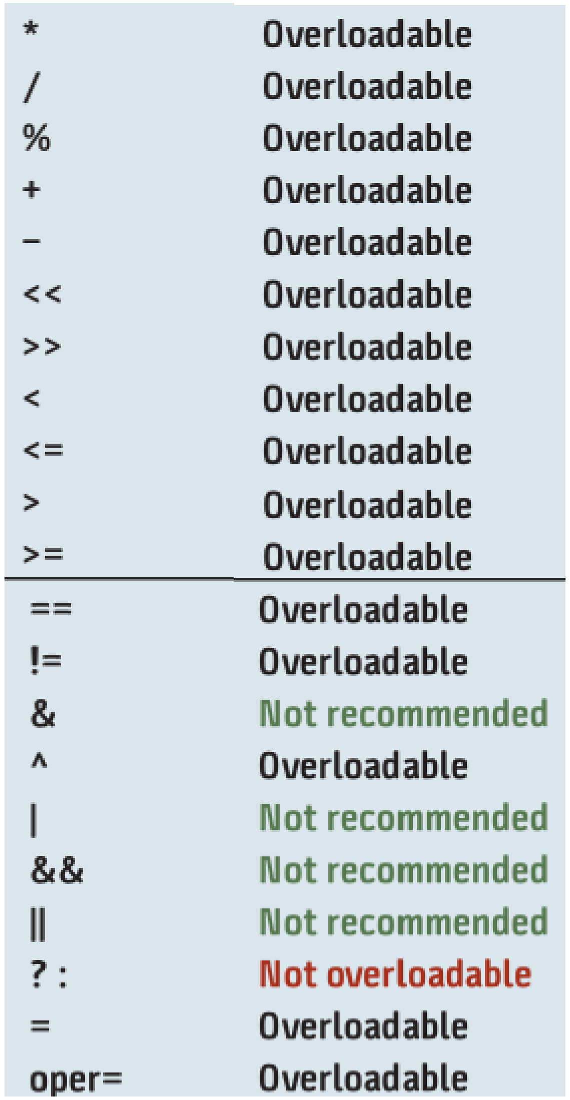
const type **operator** ± ( ) **const**

Checklist

1. Host object does not change (constant).
2. Returned object
    a. is created as a new object (no reference).
    b. is used as *rvalue* (constant).

Overloading Binary Operators

Binary Operators



*	Overloadable
/	Overloadable
%	Overloadable
+	Overloadable
–	Overloadable
<<	Overloadable
>>	Overloadable
<	Overloadable
<=	Overloadable
>	Overloadable
>=	Overloadable
==	Overloadable
!=	Overloadable
&	Not recommended
^	Overloadable
|	Not recommended
&&	Not recommended
||	Not recommended
? :	Not overloadable
=	Overloadable
oper=	Overloadable

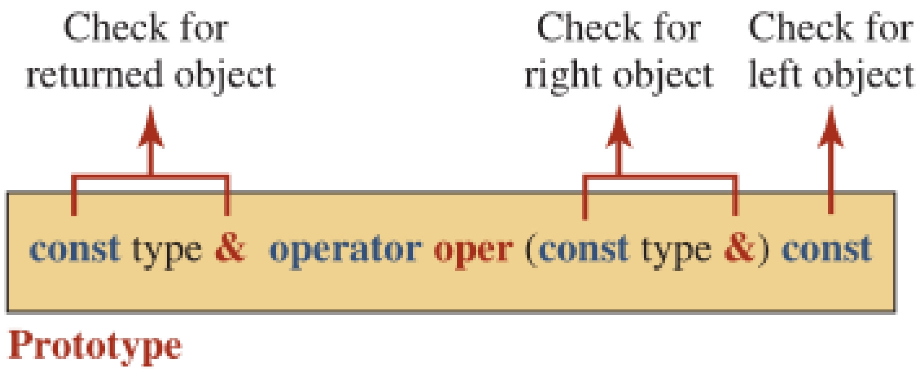
Overloading Binary Operators

- A binary operator has two operands (left operand and right operand).
- If we want to overload a binary operator as a member function, we must consider one of the operands as the host object and the other as the parameter object.
- It is more natural to use the left operand as the host object and the right operand as the parameter object.
- For this reason, it is common to overload only those binary operators as member functions in which the left operand (which becomes the host object) has a different role than the right operand (which becomes the parameter object).

Overloading Binary Operators Contd..

- Among the binary operators, the assignment and compound assignment operators (= , += , −= , *= , /= , and %=) are the best candidate for this purpose.
- In each of these operators, the left operand plays a different role from the right operand.
- The left operand represent an lvalue, but the right operand is an rvalue.
- This means that we have three objects to consider.
- The left operand becomes the host object, the right operand becomes the parameter object, and the returned value of the operation is the value of the host object after the changes.

Guidelines for overloading Binary Operators


Check for returned object
Check for right object
Check for left object
const type & operator oper (const type &) const
Prototype

1. The right object is the parameter object. Can it be passed by reference? Can it be constant?
2. The left object is the host object. Can it be constant?
3. The returned object is value of the operation. Can it be returned by reference? Can it be constant?

Overloading the assignment operator =

- This is an asymmetric operation in which the nature of the left and right operands is different.
- The left operand is an lvalue object that will be changed.
- The right operand is an rvalue object that should not be changed in the process.
- To use this operator, the left and the right operands must already exist. We only change the left object so it is an exact copy of the right object.
- If we do not define an assignment operator for our class, the system provides one which may not work like we want. So it is always a good idea to overload = operator.

```cpp
# include <iostream>
using namespace std;

class Fraction
{
    private:
        int numer;
        int denom;

    public:
        Fraction (int num, int den);
        Fraction& operator= (Fraction& right);
        void print();
};

Fraction :: Fraction (int num, int den)
: numer (num), denom (den)
{}

Fraction& Fraction :: operator= (Fraction& right)
{
    numer = right.numer;
    denom = right.denom;
    return *this;
}

void Fraction::print()
{
    cout<<numer<<"/"<<denom<<endl;
}
```

```cpp
int main ()
{
    Fraction fract1 (2, 3);
    Fraction fract2 (1,2);
    fract2 = fract1;

    fract1.print();
    fract2.print();
    return 0;
}
```

Description

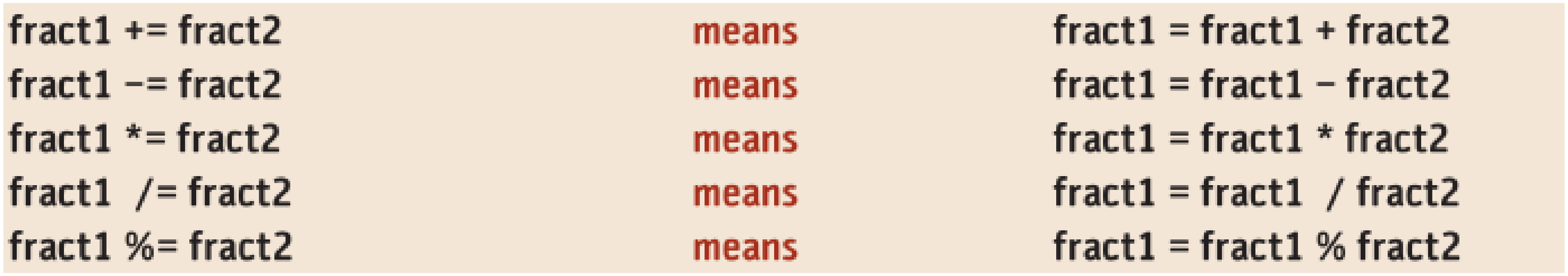
- Line 37 has the same effect as

  fract2.operator=(fract1)

- The returned object is returned by reference so that we can chain this operation. For example
  - fract1=fract2=fract3

Overloading Compound Assignment Operators

- +=
- -=
- *=
- /=
- %=

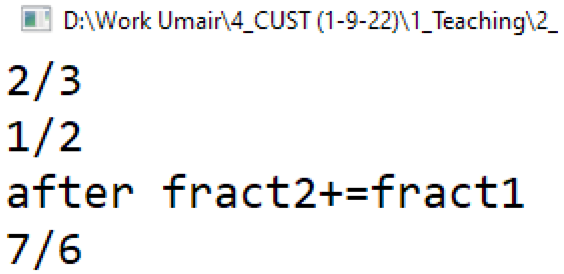


fract1 += fract2	means	fract1 = fract1 + fract2
fract1 -= fract2	means	fract1 = fract1 - fract2
fract1 *= fract2	means	fract1 = fract1 * fract2
fract1 /= fract2	means	fract1 = fract1 / fract2
fract1 %= fract2	means	fract1 = fract1 % fract2

Overloading +=

```cpp
# include <iostream>
using namespace std;

class Fraction
{
    private:
        int numer;
        int denom;

    public:
        Fraction (int num, int den);
        Fraction& operator+= (Fraction& right);
        void print();
};

Fraction :: Fraction (int num, int den)
: numer (num), denom (den)
{}

Fraction& Fraction :: operator+= (Fraction& right)
{
    numer = numer * right.denom + denom * right.numer;
    denom = denom * right.denom;
    return *this;
}

void Fraction::print()
{
    cout<<numer<<"/"<<denom<<endl;
}
```

```cpp
int main ()
{
    Fraction fract1 (2, 3);
    Fraction fract2 (1,2);
    fract1.print();
    fract2.print();

    cout<<"after fract2+=fract1"<<endl;
    fract2 += fract1;
    fract2.print();
    return 0;
}
```


D:\Work Umair\4_CUST (1-9-22)\1_Teaching\2_

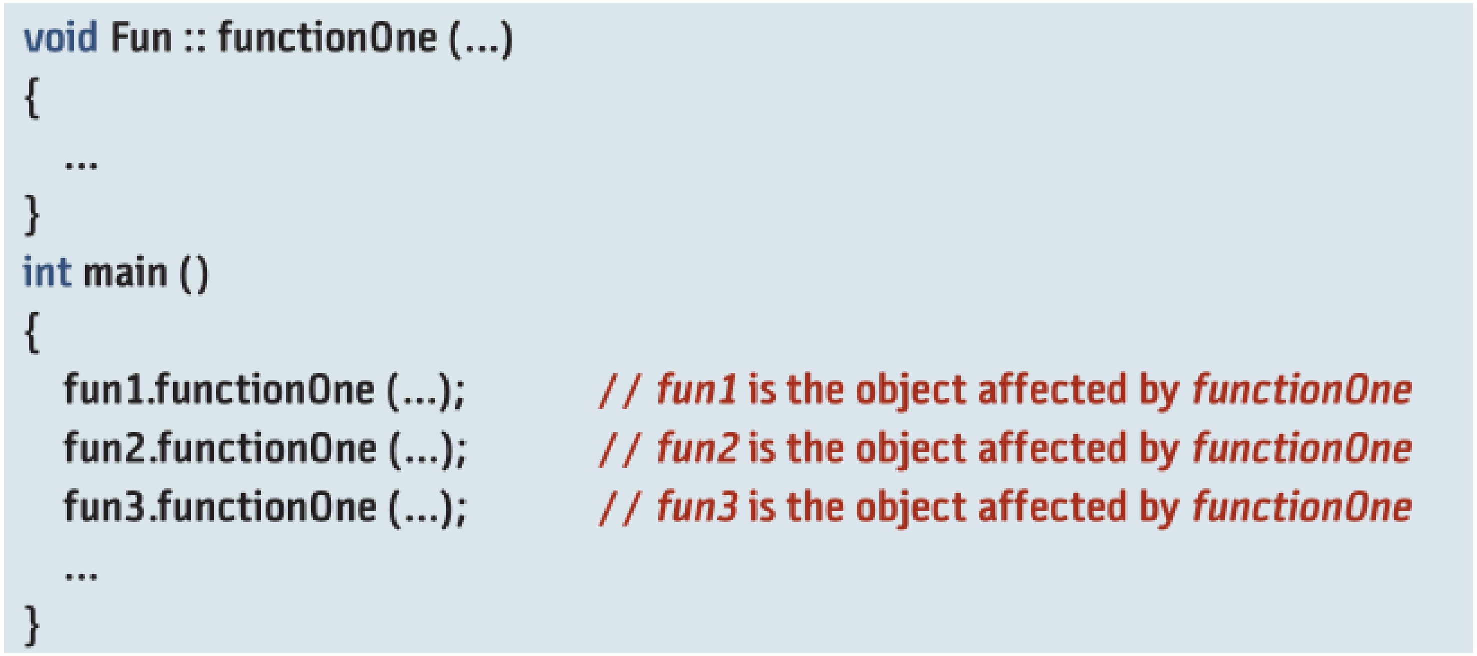
```
2/3
1/2
after fract2+=fract1
7/6
```

Three Roles of an Object

Roles of an Object

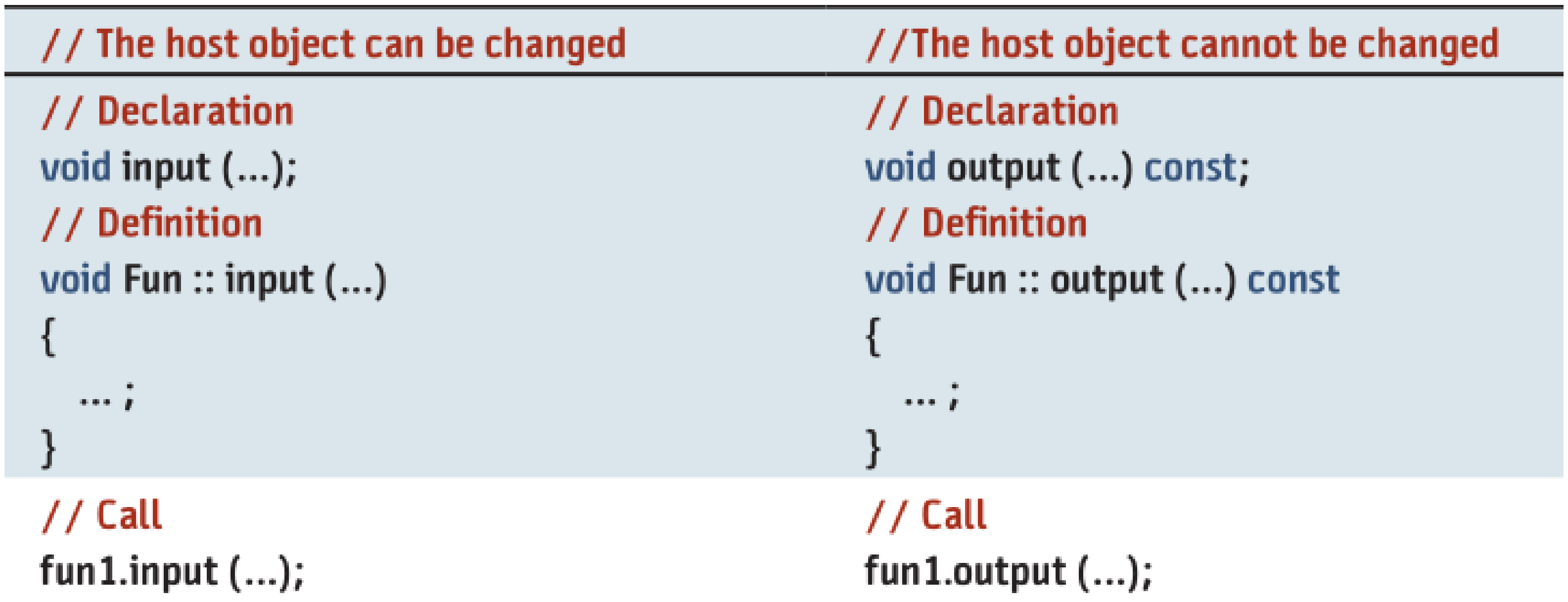
- Objects of user-defined types can play three different roles in a function:
  - host object
  - parameter object
  - returned object.

Host Object



```
void Fun :: functionOne (...)
{
   ...
}
int main ()
{
   fun1.functionOne (...);          // fun1 is the object affected by functionOne
   fun2.functionOne (...);          // fun2 is the object affected by functionOne
   fun3.functionOne (...);          // fun3 is the object affected by functionOne
   ...
}
```

Two cases for Host Object and how to write the function

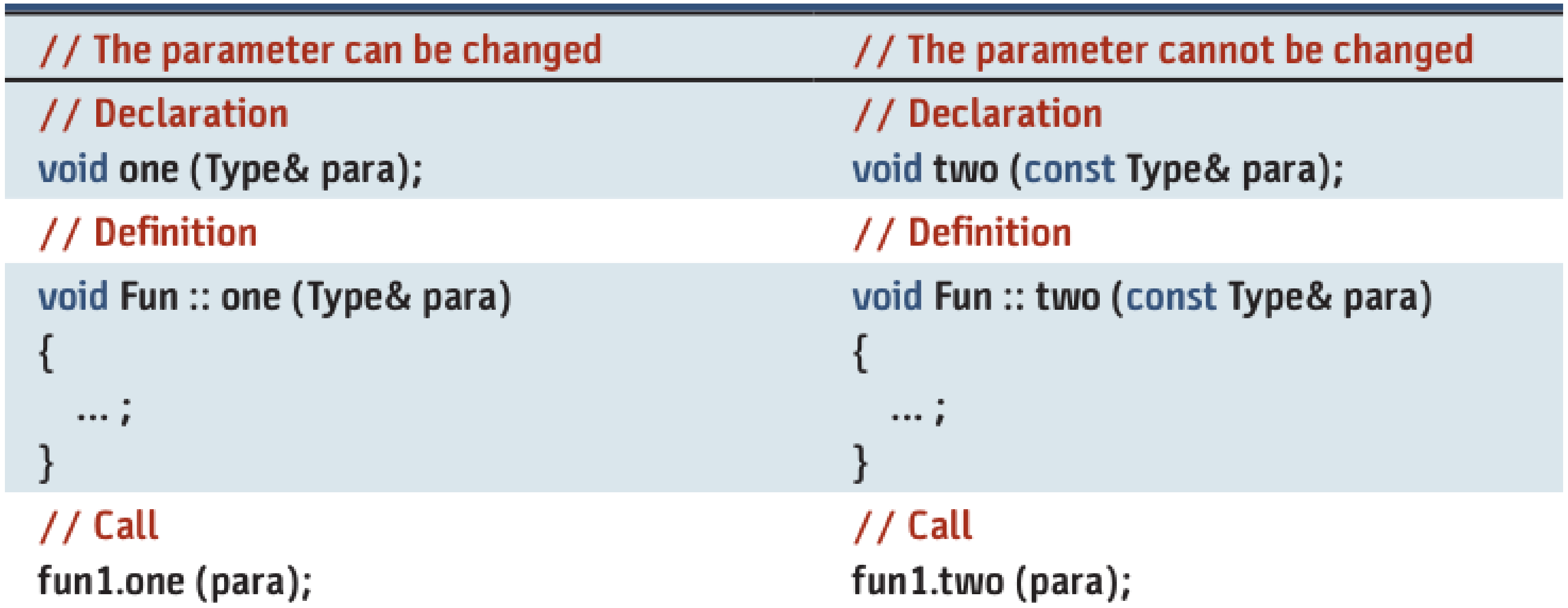


// The host object can be changed	//The host object cannot be changed
// Declaration	// Declaration
void input (...);	void output (...) const;
// Definition	// Definition
void Fun :: input (...)	void Fun :: output (...) const
{	{
... ;	... ;
}	}
// Call	// Call
fun1.input (...);	fun1.output (...);

Parameter Objects

- A parameter object is different from a host object.
- The host object is the hidden part of the member function; a parameter object must be passed to the member function.
- The first method, **pass-by-value**, is normally not used when the parameter is an object of a user-defined type because it involves calling the copy constructor, making a copy of the object, and then passing it to the function. It is very inefficient.
- The second method, **pass-by-reference**, is the most common method we encounter in practice. We do not copy the object; we just define an alias name in the function header so we can access the object. The function can use this name to access the original object and operate on it. There is no cost of copying.

Two types of parameter objects



// The parameter can be changed	// The parameter cannot be changed
// Declaration	// Declaration
void one (Type& para);	void two (const Type& para);
// Definition	// Definition
void Fun :: one (Type& para)	void Fun :: two (const Type& para)
{	{
... ;	... ;
}	}
// Call	// Call
fun1.one (para);	fun1.two (para);

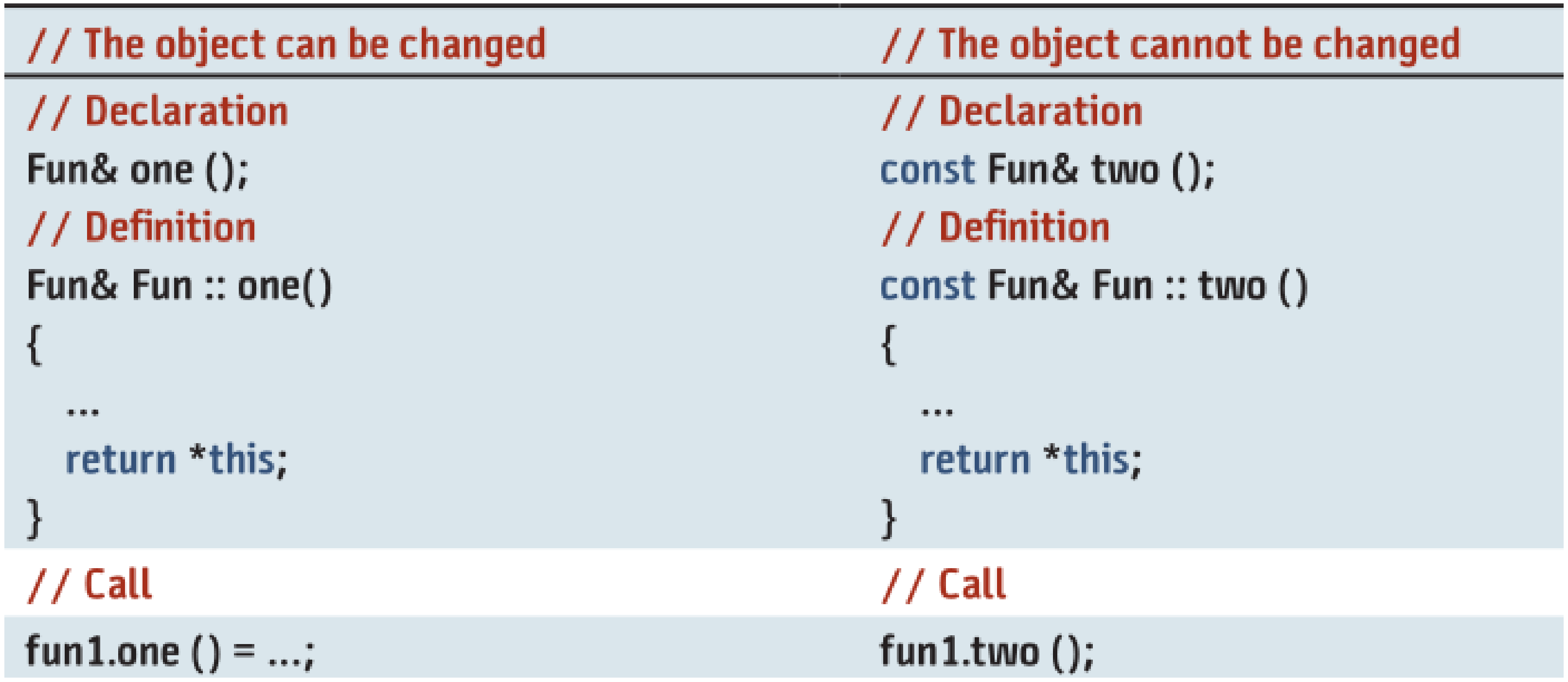
Returned Objects

- If the object to be returned is created inside the function we return it by value.
- Return-by-reference eliminates the cost of copying, but it is not possible to use it when the object is created inside the body of the function.
- When the function terminates, the object created inside the body is destroyed, and we cannot make a reference to a destroyed object.

Two types of return-by-value

//The object can be changed	//The object cannot be changed
// Declaration	// Declaration
`Fun one (int value);`	`const Fun two (int value);`
// Definition	// Definition
`Fun Fun :: one (int value)`	`const Fun Fun :: two (int value)`
`{`	`{`
`   ...`	`   ...`
`   Fun fun (value);`	`   Fun fun (value);`
`   return fun;`	`   return fun;`
`}`	`}`
// Call	// Call
`fun1.one (value) = ...;`	`fun1.two (value);`

Two examples of return-by-reference



// The object can be changed	// The object cannot be changed
// Declaration	// Declaration
`Fun& one ();`	`const Fun& two ();`
// Definition	// Definition
`Fun& Fun :: one()`	`const Fun& Fun :: two ()`
`{`	`{`
`   ...`	`   ...`
`   return *this;`	`   return *this;`
`}`	`}`
// Call	// Call
`fun1.one () = ...;`	`fun1.two ();`

This is all for Week 15