Operator Overloading

OBJECTIVES

1. What operator overloading is and how it makes programs more readable and programming more convenient.
2. To redefine (overload) operators to work with objects of user-defined classes.
3. The differences between overloading unary and binary operators.
4. To convert objects from one class to another class.
5. When to, and when not to, overload operators.
6. To use overloaded operators and other member functions of standard library class string.
7. To use keyword explicit to prevent the compiler from using single-argument constructors to perform implicit conversions.

Introduction

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
- **Examples**
  1. <<

     Stream insertion, bitwise left-shift

  2. +

     Performs arithmetic on multiple items (integers, floats, etc.)

Rules for Overloading Operator

1. Only existing operators can be overloaded . New operators cannot be created.
2. The overloaded operators must have at least one operand that is of user defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded.
6. We cannot use friend function to overload certain operators. However, member functions can be used to overload them.
7. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
8. Binary arithmetic operators such as +,-,* and / must explicitly return a value . They must not attempt to change their own arguments.

Operators that cannot be overloaded	
Sizeof	**Size of operator**
.	**Membership operator**
*****	**Pointer -to -member operator**
::	**Scope resolution operator**
?:	**Conditional Operator**

Where a friend function cannot be used

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
□	Class member access operator

Fundamentals of Operator Overloading

Types for operator overloading

1. Can use existing operators with user-defined types
2. Cannot create new operators

Overloading operators

3. Create a function for the class
4. Name of operator function
5. Keyword `operator` followed by symbol

Example: `operator+` for the addition operator +

Good Programming Practice

Overloaded operators should mimic the functionality of their built-in counterparts—

for example,

the + operator should be overloaded to perform addition, not subtraction.

Fundamentals of Operator Overloading

- Using operators on a class object
  - It must be overloaded for that class
    - Assignment operator (=)
      - Memberwise assignment between objects

- Overloading provides concise notation
  - `object2 = object1.add( object2 );`

  vs.

  `object2 = object2 + object1;`

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operations.

2. Declare the operator function operator operator() in the public part of the class .

3. It may be either a member function or a friend function.

4. Define the operator function to implement the required operations.

Operator Functions
Class Members vs. Global Members

- **Operator functions**
  - **As member functions**
    - **Leftmost object must be of same class as operator function**
    - **Use `this` keyword to implicitly get left operand argument**
    - **Called when**
      - **Left operand of binary operator is of this class**
      - **Single operand of unary operator is of this class**
  - **As global functions**
    - **Need parameters for both operands**
    - **Can have object of different class than operator**
    - **Can be a `friend` to access `private` or `protected` data**

1. Operator functions must be either member functions (non-static) or friend functions.

A basic difference between them is that friend function will have only one argument for unary operators and two for binary operators, while a member function has no argument for unary operators and only one for binary operators.

2. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions.

3. Arguments may be passed either by value or by reference.

Overloading
Stream Insertion and Stream Extraction Operators

- **<< and >> operators**
  - **Already overloaded to process each built-in type**
  - **Can also process a user-defined class**
    - **Overload using global, `friend` functions**

- **Example program**
  - **Class `PhoneNumber`**
    - **Holds a telephone number**
  - **Print out formatted number automatically**

```
(123) 456-7890
```

```
#include<iostream>
#include<string>
using namespace std;
      class Person
{
private:
string name;
int age;

public:
Person()
{name = "noname";
age = 0;
}

friend void operator<< (ostream &output, Person &p);
friend void operator>>(istream &input, Person &p);
};
```

```
void  operator<< (ostream &output, Person &p)
{
output << “Its fun" << endl;
output << "my name is " << p.name << "and my age is" << p.age;
}

void  operator>>(istream &input, Person &p)
{
input >> p.name >> p.age;

}

int main()
{
Person ki;
cout << "Enter the name and age" <<endl;
cin >> ki;
cout << ki;
system("pause");
return 0;
}
```

Software Engineering Observation 11.3

New input/output capabilities for user-defined types are added to C++ without modifying C++'s standard input/output library classes.

This is another example of the extensibility of the C++ programming language.

Overloading Unary Operators

1. Can overload as non-static member function with no arguments

2. Can overload as global function with one argument
   Argument must be class object or reference to class object

Unary minus operator is overloaded

```
#include<iostream>
using namespace std;
class space
{
private:
    int x;
    int y;
    int z;
public:
    void getdata(int a, int b, int c);
    void display(void);
    void operator-();
};
void space::getdata(int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
```

```
void space::operator-()
{
    x = -x;
    y = -y;
    z = -z;
}

int main()
{
    space s;
    s.getdata(10, -20, 30);
    -s;
    cout << "s:" <<  endl;
    s.display();
    system("pause");
    return 0;

}
```

Program to overload shorthand operator

```
#include<iostream>
using namespace std;
class Marks
{
private:
    int marks;
public:
    Marks()
    {
        marks = 0;
    }
    Marks(int m)
    {
        marks = m;
            }
    void YourMarkPlease()
    {
        cout << "your mark is" << marks << endl;
    }
    void operator+=(int bonusmark)
    {
        marks = marks + bonusmark;
    }
    friend void operator-=(Marks &curobj, int reducedmark);
};
```

```
void operator-=(Marks &curobj, int reducedmark)
{
    curobj.marks -= reducedmark;

}
int main()
{
    Marks kimark(45);
    kimark.YourMarkPlease();
    int x = 20;
    kimark += 20;
    kimark.YourMarkPlease();
    kimark -= x;
    kimark.YourMarkPlease();
    system("pause");
    return 0;
}
```

Program to overload Increment and Decrement operator (Postfix)

```
#include<iostream>
using namespace std;
class Marks
{
private:
    int marks;
public:
    Marks()
    {
        marks = 0;
    }
    Marks(int m)
    {
        marks = m;
    }
    void YourMarkPlease()
    {
        cout << "your mark is" << marks << endl;
    }
    Marks operator++(int)
    {
        Marks duplicate(*this);
        marks = marks + 1;//mark+=1
        return duplicate;
    }
    friend Marks operator--(Marks &, int);
};
Marks operator--(Marks &m, int )
{
    Marks dup(m);
    m.marks -= 1;
    return dup;
}
int main()
{
    Marks ki(75),ki1(55);
    ki.YourMarkPlease();
    (ki++).YourMarkPlease();
    ki.YourMarkPlease();
    ki1.YourMarkPlease();
    (ki1--).YourMarkPlease();
    ki1.YourMarkPlease();
    system("pause");
}
```

Program to overload array subscript operator

```
#include<iostream>
using namespace std;
class Marks
{
private:
    int subjects[3];

public:
    Marks(int sub1, int sub2, int sub3)
    {
        subjects[0] = sub1;
        subjects[1] = sub2;
        subjects[2] = sub3;
    }

    int operator[](int position)
    {
        return subjects[position];
    }
};
```

```
int main()
{
    Marks ki(22, 44, 33);
    cout << ki[0] << endl;
    cout << ki[1] << endl;
    cout << ki[2] << endl;
    system("pause");
    return 0;
}

/*
```

Program to overload Function Call operator

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

class Marks
{
private:
    int marks;
public:
    Marks(int m)
    {
        marks = m;
        cout << "constructor is called" << endl;
    }
    void whatsYourMarks()
    {
        cout << "hey I got" << marks << "marks" << endl;
    }
    Marks operator()(int mk)
    {
        marks = mk;
        cout << "operator function is called" << endl;
        return *this;
    }
};
```

```
int main()
{
    Marks ki(85);
    ki.whatsYourMarks();
    ki(44);
    ki.whatsYourMarks();
    system("pause");
    return 0;
}
```

Program to overload the class member access operator

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

class Marks
{
private:
    int marks;
public:
    Marks(int m)
    {
        marks = m;
        cout << "constructor is called" << endl;
    }
    void whatsYourMarks()
    {
        cout << "hey I got" << marks << "marks" << endl;
    }
    Marks *  operator -> ()
    {
        return this;
    }
};
```

```
int main()
{
    Marks kimarks(65);
    kimarks.whatsYourMarks();
    kimarks->whatsYourMarks();
    system("pause");
    return 0;
}
```