Operator Overloading

Understanding the Benefits of Overloading

- Having more than one function with the same name is beneficial because you can use one easy-to-understand function name without paying attention to the data types involved
- Polymorphism allows the same operation to be carried out differently, depending on the object
- Some reserve the term polymorphism (or pure polymorphism) for situations in which one function body is used with a variety of arguments

Using the + Operator Polymorphically

- Separate actions can result from what seems to be the same operation or command
- The + operator has a variety of meanings, which include:
  - Alone before a value (called unary form), + indicates a positive values, as in the expression +7
  - Between two integers (called binary form), + indicates integer addition, as in the expression 5+ 9
  - Between two floating-point numbers (also called binary form), + indicates floating-point addition, as in the expression 6.4 + 2.1

Operator Overloading Syntax

- Syntax is:

operator*@*(argument-list)

--- operator is a function

--- @ is one of C++ operator symbols (+, -, =, etc..)

Examples:

operator+

operator-

operator*

operator/

Fundamentals of Operator Overloading

- Use operator overloading to improve readability
  - Avoid excessive or inconsistent usage
- Format
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded.
  - **`operator+`** would be used to overload the addition operator (**+**)

8.3 Restrictions on Operator Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Implementing Operator Overloading

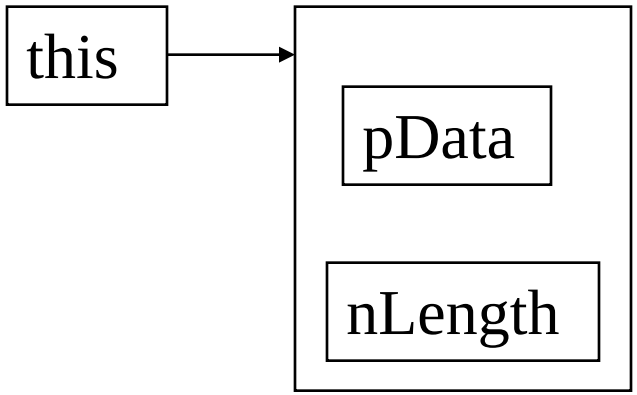
- Two ways:
  - Implemented as member functions
  - Implemented as non-member or Friend functions
    - the operator function may need to be declared as a friend if it requires access to protected or private data
- Expression *obj1@obj2* translates into a function call
  - *obj1.operator@(obj2)*, if this function is defined within class obj1
  - *operator@(obj1,obj2)*, if this function is defined outside the class obj1

Operator Functions as Class Members vs. as `friend` Functions

- Operator overloading functions can be member functions or non-member functions (friend).
- Operator functions as member functions
  - Leftmost operand must be an object (or reference to an object) of the class
  - If left operand of a different type, operator function must be a non-member function
- A non-member operator function must be a friend if private or protected members of that class are accessed directly

The "*this*" pointer

- Within a member function, the *this* keyword is a pointer to the current object, i.e. the object through which the function was called
- C++ passes a hidden *this* pointer whenever a member function is called
- Within a member function definition, there is an implicit use of *this* pointer for references to data members



CStr object
(*this)

Data member reference	Equivalent to
pData	this->pData
nLength	this->nLength

```
#include<iostream>
using namespace std;
class Number
{
        int n;
public:
        Number(int a=0):n(a){}
        void print();
        Number operator ++();
};
void Number::print()
{
        cout<<"Number="<<n;
}
```

```
Number Number::operator++()
{
        n++;
        return *this;
}

int main()
{
        Number ob1(10);
        Number ob2;
        ob2 = ++ob1; //obj1.operator++();
        ob1.print();
}
```

- How do we write post increment operator ++?
- Give a dummy int parameter to ++ operator function.

```
Number Number::operator++()//pre increment
{
        n++;
        return *this;
}
ob2 = ++ob1; //equals to   ob1.operator++();
Number Number::operator++(int a )//post increment
{
        Number temp=*this;
        n++;
        return temp;
}
ob2 = ob1++; //equals to   ob1.operator++(0);
```

*** Here we used int a as a dummy parameter for performing post increment.

Binary operators

- Take one argument for member function. The first operand is current object.

```
Number Number::operator+ (Number N1)
{
    Number temp;
    temp.n = n+N1.n;

      return temp;
}
```

```
Number ob1(10),ob2(5),ob3;
ob3 = ob1+ob2;//ob1 is current object and ob2 is N1
```

***ob3=ob1+ob2 is equals to ob3=ob1.operator+(ob2)

+= operator

```
Number Number::operator+= (Number N1)
{
   Number temp;
      n+=  N1.n;
       return *this;
}
ob1+=ob2;  //in main
```

Equals to

```
ob1.operator+=(ob2)
```

Fundamentals of Operator Overloading

- Assignment operator (**=**)
  - may be used with every class without explicit overloading
  - *memberwise assignment*

```
Number Number::operator= (Number N1)
{
        n = N1.n;
        return *this;
}

ob1=ob2;        //in main
ob1.operator=(ob2);
```

Overloading subscript operator

- Can we used to implemented arrays which will check the boundaries

```
#include<iostream>
using namespace std;
int i=0;
class subscript
{
  int array[10];
  public:
  subscript(int k)
  {
     for(i=0;i<10;array[i]=k,i++);
   }
   int &operator[](int k)
   {
       if(k >= 10)
      {
           cout<<"\n wrong subscript...\n";
           exit(1);
       }
       else
       return array[k];
    }
};
int main()
{
 class subscript ss(0);
 cout<<"\n now we are going to place 13 element in 2nd
      index";
 ss[2]=13;
 cout<<"\n now we are going to place 22 element in 10
      index";
 ss[10]=22;              // error because size is exceeding
}
```

Overloading Output

- The << operator also is overloaded by C++
- It is both a bitwise left-shift operator and an output operator; it is called the insertion operator when used for output
- The << operator acts as an output operator only when `cout` (or another output stream object) appears on the left side
- When you use `cout` in a program, you must include #include<iostream>
- The preceding function, called operator<<(), returns a reference to ostream

Overloading Output

- It accepts two arguments: a reference to ostream (locally named `out` in this example) and an integer (locally named `in` in this example)
- C++ overloads the << operator to work with the built-in data types; you also may overload the << operator to work with your own classes
- To overload << operator so it can work with a Sale object, you must add the overloaded operator <<() function to the Sale class

Overloading Output

- **The operator <<() function is a friend to the class of the object it wants to print out, e.g. Sale here.**

```
ostream& operator<<(ostream &out, const Sale &aSale)
 {
     out<<"Sale #"<<aSale.receiptNum
          <<" for $"<<aSale.saleAmount<<endl;
     return (out);
 }
```

Figure 8-11 Overloaded operator<<() function for the Sale class

Overloading Input

- If the << operator can be overloaded for output, it makes sense that the >> operator also can be overloaded for input
- The advantage of overloading operators such as >> is that the resulting programs look cleaner and are easier to read
- You can create an extraction operator, or operator>>() function, that uses istream (which is defined in iostream.h, along with ostream) by using a prototype as follows:

```
friend istream& operator>>(istream &in, Sale &Sale);
```

Overloaded Operator>>() Function for the Sale Class

```
istream& operator>>(istream &in, Sale &aSale)
 {
   cout<<endl;   // to clear the buffer
   cout<<"Enter receipt number ";
   in>>aSale.receiptNum;
   cout<<"Enter the amount of the sale ";
   in>>aSale.saleAmount;
   cout<<endl<<"     Thank you!"<<endl;
   return(in);
 }
```

Figure 8-13 Overloaded operator>>() function for the Sale class

Converting between Types

- Cast operator
  - Convert objects into built-in types or other objects
  - Conversion operator must be a non-**`static`** member function.
  - Cannot be a **`friend`** function
  - Do not specify return type

For user-defined class **`A`**

```
A::operator char *() const;      // A to char
A::operator int() const;         //A to int
A::operator otherClass() const; //A to otherClass
```

- When compiler sees **`(char *) s`** it calls

```
s.operator char*()
```

```
#include<iostream>
using namespace std;
class A {
    int a_int;
    char* a_carp;
    public:
    void display()
    {
        cout<<"\n we are in A class::";
        cout<<a_int<<"\t"<<a_carp;
    }
    void getdata(int x, char *y)
    {
        a_int=x;
        a_carp=y;
    }
    operator int() { return a_int; }
    operator char*() { return a_carp; }
};
```

```
class B : public A {
        float b_float;
        char* b_carp;
        public:
        void print()
        {
            cout<<"\n we are in B class::";
            cout<<b_float<<"\t"<<b_carp;
        }
        void setdata(float x, char *y)
        {
            b_float=x;
            b_carp=y;
        }
        operator float() { return b_float; }
        operator char*() { return b_carp; }
};
```

```
int main () {
        A a_obj;
        cout<<"\n a object members values are as follows::";
        a_obj.getdata(23,"ism");
        a_obj.display();
        B b_obj;
        cout<<"\n b object member values are as follows::";
        b_obj.setdata(10.23,"tech");
        b_obj.print();
        //  long a = b_obj;  //error becase long is not a member of B class
        cout<<"\n\n we are going to type cast to class object to int::";
        int i=a_obj;
        cout<<"\n after typecast by using \"A\" object value is::"<<i;
        cout<<"\n\n we are going to type cast class object to float";
        float f=b_obj;
        cout<<"\n after trypecast by using \"B\" object value is::"<<f;
        cout<<"\n\n now we are going to typecast class object to char pointer";
        char *cptr=a_obj;
        cout<<"\n after typecast by using \"A\" object value is:"<<cptr;
        cout<<"\n";
}
```