OPERATOR OVERLOADING

Fundamentals

- There are many operators available that work on built-in types, like int and double.
- **Operator overloading** -- is the creation of new versions of these operators for use with user-defined types.
- **Operators** usually refer to C++ predefined operators:
  - arithmetic operators: **+, -, *, /, %**
  - relational operators: **<, <=, ==, !=, >, >=**
  - assignment operator: **=**
  - logical operators: **&&, ||**, !
  - input/output operators: <<, >>
- It is not as difficult as it sounds. Some things to note:
  - An **operator** in C++ is just a **function** that is called with special notation (usually more intuitive or familiar notation). Overloading an operator simply involves writing a function.
  - C++ already does some operator overloading implicitly on built-in types. Consider the fact that the + operator already works for ints, floats, doubles, and chars. There is really a different version of the + operator for each type.

Operator overloading is done for the purpose of using familiar operator notation on **programmer-defined** types (classes).

Some rules regarding operator overloading

- Overloading an operator cannot change its precedence.
- Overloading an operator cannot change its associativity.
- Overloading an operator cannot change its "arity" (i.e. number of operands) You cannot change the number of arguments that an operator takes.
- It is not possible to create new operators -- only new versions of existing ones.
- Operator meaning on the built-in types cannot be changed.
- The following operator can only be overloaded as member functions: =, [], -> and ().
- The following operator cannot be overloaded: the dot operator (.), the scope resolution operator (::), sizeof, ?: and .*.
- An overloaded operator cannot have default arguments.

Format

- An operator is just a function. This means that it must be created with a return type, a name, and a parameter list
- The rules above give some restrictions on the parameter list
- The *name* of an operator is always a conjunction of the keyword `operator` and the operator symbol itself. Examples:
    - `operator+`
    - `operator++`
    - `operator<<`
    - `operator==`
- So the format of an operator overload declaration is just like that of a function, with the keyword `operator` as part of the name:
- *returnType* `operator`*OperatorSymbol* (*parameterList*);

Overloading operator implementation

- Operations for **C++ primitive data types** (such as *int*, *char* and *double*) are predefined in C++ language.

```
int main()
{
   int x, y = 3;

   x = y + 10;     // OK (int + int), other arithmetic operators -, *, /,
%
                   //   also OK (int = int) -- assignment
   if (x < y)      // OK (int < int), other logical operators <=, >, >=,
==, !=
         ...
```

- But, for user-defined data types (Classes), C++ doesn't have definitions for those operators.

```
class Money
{
public:
  Money();
  Money(int d, int c);
  Money(int allc);

  double getAmount(); // Returns the amount as a double
     void printMoney(); // prints a money to cout, in the form $xx.yy
   private:
     int dollar;
     int cent;
   };
   int main()
   {
      Money m1(3, 25), m2(19, 5);
```

```
   Money m3 = m1 + m2; // SYNTAX ERROR!! operator+ is not defined for
Money

   cout << m1;         // SYNTAX ERROR!! operator<< is not defined for
Money
```

- An **Operator is essentially a function**. So, we can look at expressions with operators as function call. Those functions have **operator keyword** in front of them (in prefix notation).
- ```
  x = x + 5;   // infix notation, same as: x = operator+(x, 5);
  cout << x;   // infix notation, same as: operator<<(cout, x);
  ```

- So if you want to use familiar syntax with classes, **you must write the definition for the (overloading) operators** for the class.

The Thee ways

- There are 3 ways to define overloaded operators:
    1. Member function
    2. Nonmember function
    3. Friend function

Member function

- The first way is by **class method** (member function). This is the most popular way.

  Many experts advocate always overloading operators as member operators rather than as nonmembers. It is more in the spirit of object-oriented programming and is a bit more efficient since the *definition can directly reference member variables*."

- Since the operator is applied on a (existing) class object, the number of parameters to the operator is one less:
    - **Binary operators have 1 parameter**, the second operand to the operator. The first operand is the object in which the overloaded operator is called/invoked.
    - **Unary operators** (e.g. unary - for negative) **have 0 parameter**.
  ```

Example

// filename: money.h -- Header file for class Money

```
#ifndef MONEY_H
#define MONEY_H

#include <iostream>
using namespace std;

class Money
{
public:
  Money() : dollar(0), cent(0) {}
  Money(int d, int c) : dollar(d), cent(c) {}
  Money(int allc);

  Money operator+(const Money & mo2) const;
  Money operator-(const Money & mo2) const; // binary -
  Money operator-() const;                  // unary -
  bool operator==(const Money & mo2) const;

  bool operator<=(const Money & mo2) const;

  int getDollars() const { return dollar; }
  int getCents() const { return cent; }

  // friend functions
  friend ostream& operator<<(ostream& out, const Money & m);
  friend istream& operator>>(istream& in, Money & m);
  friend bool operator>(const Money &, const Money &);

private:
  int dollar;
  int cent;
};

// prototypes of overloaded operators implemented as
// regular functions
bool operator<(const Money &, const Money &);
bool operator!=(const Money &, const Money &);

#endif
```

// filename: money.cpp -- Implementation file for class Money

```
#include "money.h"

Money::Money(int allc)
{
  dollar = allc / 100;
  cent = allc % 100;
}

Money Money::operator+(const Money & m2) const
```

```
{
   int total = (dollar * 100 + cent) +
               (m2.dollar * 100 + m2.cent);
   Money local(total);
   return local;
}




Money Money::operator-(const Money & m2) const // *this - mo2
{
   int diff = (dollar * 100 + cent) -
              (m2.dollar * 100 + m2.cent);
   Money local(diff);
   return local;
}

Money Money::operator-() const // unary -
{
   int neg = - (dollar * 100 + cent);
   Money local(neg);
   return local;
}

bool Money::operator==(const Money & m2) const
{
   int thistotal = (dollar * 100 + cent);
   int m2total = (m2.dollar * 100 + m2.cent);
   return (thistotal == m2total);
}

bool Money::operator<=(const Money & m2) const
{
   int thistotal = (dollar * 100 + cent);
   int m2total = (m2.dollar * 100 + m2.cent);
   return (thistotal <= m2total);
  /*
  if (*this < m2 || *this == m2)
    return true;
  else
    return false;
    */
}

// no keyword "friend" in the function definition
ostream& operator<<(ostream& out, const Money & m)
{
   out << "$" << m.dollar  // dollar private in m -- OK
       << "." << m.cent;   // cent private in m -- OK

   return out;
}

istream& operator>>(istream& in, Money & m)
{
  char dollarSign;
  double moneyAsDouble;
```

```
  in >> dollarSign;    // first eat up '$'
  in >> moneyAsDouble; // xx.yy

  m.dollar = static_cast<int>(moneyAsDouble);
  m.cent = static_cast<int>(moneyAsDouble * 100) % 100;

  return in;
}

// friend function
bool operator>(const Money & m1, const Money & m2)
{
   int thistotal = m1.dollar * 100 + m1.cent;
   int m2total = m2.dollar * 100 + m2.cent;
   return (thistotal > m2total);
}

bool operator<(const Money & m1, const Money & m2) // note: 2 arguments and
NO Money::
{
   int thistotal = m1.getDollars() * 100 + m1.getCents();
   int m2total = m2.getDollars() * 100 + m2.getCents();
   return (thistotal < m2total);
}

bool operator!=(const Money & m1, const Money & m2)
{
   int thistotal = m1.getDollars() * 100 + m1.getCents();
   int m2total = m2.getDollars() * 100 + m2.getCents();
   return (thistotal != m2total);
}
```

____
_

```
// filename: myMoneyApp.cpp
//
// An application program which uses Money objects (defined in
// "money.h").

#include <iostream>
using namespace std;

#include "money.h"

int main()
{
   Money m1(2, 98), m2(15, 2), m3;

   m3 = m1 + m2; // member function operator+

   cout << m1 << " + " << m2 << " = " << m3 << endl;

   if (m1 != m2)
     cout << "Not equals.\n";
   else
    cout << "Equals.\n";
```

```
   if (m1 > m2)
     cout << "Greaterthan.\n";
   else
    cout << "NOT Greaterthan.\n";

   bool ans = m1 > m2;
   cout << ans << endl; // prints 0 (false) or 1 (true)


   system("pause");
   return 0;
}
```

Run time ouput

```
$2.98 + $15.2 = $18.0
Not equals.
NOT Greaterthan.
0
```

Top-level (Nonmember) function

- Another way to overload operators is by regular, **non-member** functions.
- Since the operator is NOT a class method, all operands involved in the operator become the parameters:
  - **Binary operators have 2 parameters**, the second operand to the operator. The first operand is the object in which the overloaded operator is called/invoked.
  - **Unary operators have 1 parameter**.

- Also the operator **cannot** access private members in the parameter objects.

```
class Money
{
public:
  Money();
  Money(int d, int c);
  Money(int allc);

  int getDollars() const;
  int getCents() const;
  ...
     // note: NO method for operator<
   private:
     int dollar;
     int cent;
   };
   // Definition of regular, non-member functions.

   // mo1 < mo2
   bool operator<(const Money & m1, const Money & m2) // note: 2
   arguments and NO Money::
```

```
    {
       int thistotal = m1.getDollars() * 100 + m1.getCents();
       int m2total = m2.getDollars() * 100 + m2.getCents();
       return (thistotal < m2total);
    }
```

Friend function

- Yet another way is to use **friend function**. Friend functions are declared within a class, but **they are NOT class methods.**
- A friend function is actually a regular function which has a privilege to **access private members** in the parameter objects.

```
class Money
{
public:
  Money();
  Money(int d, int c);
  Money(int allc);
  Money operator+(const Money & mo2) const;
  ...
  // friend functions
  friend ostream& operator<<(ostream& out, const Money & m); // to be able
to do cout << obj
  friend istream& operator>>(istream& in, Money & m);  // to be able to do
cint >> obj

private:
  int dollar;
  int cent;
};
// no keyword "friend" in the function definition
ostream& operator<<(ostream& out, const Money & m)
{
   out << "$" << m.dollar  // dollar private in m -- OK
       << "." << m.cent;   // cent private in m -- OK

   return out;
}

istream& operator>>(istream& out, Money & m)
{
  char dollarsign;
  double moneyAsDouble;

  in >> dollarSign;    // first eat up '$'
  in >> moneyAsDouble; // xx.yy

  m.dollar = static_cast<int>(moneyAsDouble);
  m.cent = static_cast<int>(moneyAsDouble * 100) % 100;

  return in;
}
```

Important Remarks

- For all 3 ways for operator overloading, they are all **called the same way** (i.e., infix notation)**.**

```
int main()
{
   Money m1(2, 98), m2(15, 2), m3;

   m3 = m1 + m2;              // member function operator+

   if (m1 == m2)              // member function operator==
     cout << "same amount";

   bool ans = m1 < m2;        // non-member function operator<

   cout << p1;                // friend function operator<<
```

- Remember parameters to the operators are **POSITIONAL**: the order of 1st/2nd parameter DOES matter.

```
int main()
{
   Money m1(2, 98), m2(15, 2), m3;

   m3 = m1 * 0.8;  // (a) 1st arg Money, 2nd arg double
   m3 = 1.7 * m1;  // (b) 1st arg double, 2nd arg Money
```

- So for operators WHOSE 1st ARGUMENT IS NOT A CLASS OBJECT, you must write them as friend or regular functions.

```
class Money
{
public:
  Money();
  Money(int d, int c);
  Money(int allc);
  Money operator+(const Money & mo2) const;
  ...
  Money operator*(double r) const;    // for usage case (a), class method
works

private:
  int dollar;
  int cent;
};


// ANOTHER operator*, for usage case (b).
// It has to be a friend or regular function.
// Here is implemented as a regular function.
Money operator*(double r, const Money & m);  // prototype only here

.....
```

```
Money operator*(double r, const Money & m)
{
   int total = m.toAllCents() * r;
   Money local(total);
   return local;
   }
```

- Also for the reason above, the operator<< and operator>> are often implemented as friend functions.

```
int main()
{
  Money m1(2, 98);

  cout << m1;
}
```

- **Automatic Type Promotion**
  - If an operator is expecting a class object but received a different type, if there is a constructor in the class which can convert it to the class, the conversion/promotion is *automatically* applied by the compiler.

  Example: Suppose the operator+ is implemented as a member function in Money. Then in the application:

```
int main()
{
  Money m1(3, 25), m2;

  m2 = m1 + 6; // 2nd operand is int, not Money.
               // This int is promoted to a Money object by
               // the Money constructor which has one int argument
               // if it is defined (and in our example, it is).
```

Overloading Other Operators

- **Operator[]** -- index operator

```
// A class with an array of five double's.
class DoubleArray5
```

```
{
public:
  DoubleArray5(double initvalue);
  ...
  double& operator[](int index) const; // index is the parameter

private:
  int ar[5];
};

// NOTE: return by reference(&)
double& DoubleArray5operator[](int index) const
{
  if (index <= 0 || index > 5)
  {
    cout << "ILLEGAL INDEX.\n";
    exit(1);
  }
  else
    return ar[index];
}

int main()
{
  DoubleArray5 myarray(0.0);

  double d = myarray[2]; // myarray.operator[](2)

  myarray[1] = 5.7;  // can be used on the LHS of =
  ...
   }
```

- **Operator++ and operator-- --** increment/decrement operators

```
// A counter class
class Counter
{
public:
  Counter(int c = 0);  // initializes 'count' to c
  ...
  Counter operator++();       // pre-increment
  Counter operator++(int i);  // post-increment

private:
  int count;
};

Counter Counter::operator++()
{
  // First, increment 'count'.
  count++;
  // Second, create a local object with the new count.
  Counter local(count);
  // Third, return the local object.
  return local;
}
```

```
Counter Counter::operator++(int i) // param value is IGNORED!!
{
  // First, create a local object with the current count.
  Counter local(count);
  // Second, increment count.
  count++;
  // Third, return the local object.
  return local;
   }
```

- **Operator= --** assignment operator

  This operator will be discussed later in conjunction with pointers.

- ## Type conversion

Conversion	Routine in Destination	Routine in source
Basic to basic (float to int)	Built in	Built in
Basic to class (int to obj)	Constructor	
Class to Basic (obj to int)		Operator function
Class to class (obj to otherObj)	Constructor	Operator function

- Example: Class to basic and basic to class
  - Metric system vs English system

```
const float MTF=3.280833;
Class Es
{
        int feet;
        int inches;
        public:
                Es(int f, float i)
                {
                        feet=f;
                        inches=i;
                }
                //basic to class
                Es(float m) //m is a metric value
                {
                        float fi=MTF *m;
                        feet=fi;
                        inches=12*(fi-feet)
                }
                //class to basic
                operator float()
                {
                        float ff=inches/12;
                        ff+=feet;
                        return ff/MTF;
}
//In Main

Es e(2,3.0);
float y;
```

```
y=e; //class to basic
e=y; //basic to class
```

- Example: Class to class - Polar to Cartesian

```
Polar p;
Cartesian c;
p=c;
//or
c=p;

Class Cartesian
{
	double x;
	double y;
	public:
		Cartesian()
		{x=0;y=0;}
		Cartesian(doubly x, double y)
		{
			this.x=x;
			this.y=y;
		}
		//added constructor
		Cartesian(Polar p)
		{
			double r=P.getRadius();
			double  a=p.getAngle();
			x=r*cos(a;)
			y=r*cos(a);
		}
};
Class Polar
{

	double radius;
	double  angle;
	public:
		Polar()
		{
			radius=0;
			angle=0;
		}
		Polar (double r, double a)
		{
			radius=r;
			angle=a;
		}
```

```
        operator Cartesian()
        {
            double x=Radius*cos(angle);
            double y=radius*sin(angle);
            return Cartesian(x,y);
        }
};

//In the main
Polar p(10,.5);
Cartesian c;
c=p;
```