Operator Overloading

make user-defined operators
the same as builtin operators

What Is Operator Overloading?

- Operator overloading is a kind of polymorphism, called ad-hoc polymorphism.
- A builtin operator like + can be used to denote operations of several different data types.
- For example, + can denote the integer addition or double addition.

How Is Operator Overloading Implemented?

```
int  i1 = 1, i2 = 2, i3;
double  d1 = 1.0, d2 = 2.0, d3;

i3 = i1 + i2;
d3 = d1 + d2;
```

Compiler uses the types of operands to distinguish different operations.

How Is Operator Overloading Implemented?

```
int  i = 1;
double  d1 = 2.0, d2;

d2 = i + d1;
```

Is there an addition for an integer operand and a double operand?

No! Then, how to do it?

The integer value of i is converted automatically into a double value!

Can an operation of a user-defined data type be denoted as an operator like + too?

```
// complex.h  -- The interface
class Complex
{
public :
    Complex(double, double) ;
    const Complex operator +(const Complex &) const;
private :
    double  r;
    double  i;
} ;
```

```
Complex  c1(2, 2), c2(4, 4), c3;
c3 = c1 + c2;
           //  c3 = c1.operator+(c2);
```

```
// complex.cpp  --  The implementation
#include “complex.h”
const Complex  Complex::operator +(const Complex & c) const
{
    return Complex(r + c.r, i + c.i);
}
```

The Keyword const

```
const Complex  Complex::operator +(const Complex & c) const
{
   return Complex(r + c.r, i + c.i);
}
```

- The second const means that the parameter c cannot be changed in the function.
- The third const means that the member variables r and i cannot be changed in the function.
- The first const means that the returned object cannot be changed.

```
Complex  c1(1, 1), c2(3, 3), c3(5, 5), c4;
c4 = c1 + c2 + c3;

c4 = (c1 + c2) + c3;
```

(1, 1) + (3, 3)

The calling object is the first operand

↓

(4, 4) + (5, 5)

const 1

↓

(9, 9) = (9, 9)

nonconst 3

2 const

(c1 + c2).input(); // error

```
// complex.h  -- The interface
class Complex
{
public :
   Complex( ) ;
   Complex(double) ;
   Complex(double, double) ;
      …
private :
   double  r;
   double  i;
} ;
```

Binary operator member functions are not symmetric!

```
double d = 1;
Complex  c1(2, 2), c2;
c2 = c1 + d;     //  c2 = c1.operator+(d);
```

The + operator of Complex requires a Complex operand. The value of d is automatically converted into a Complex!

```
c2 = d + c1;    //  c2 = d.operator+(c1);
```

double has no + operator for Complex operand and no converter from Complex to double!

Friend Functions

- Friend functions are nonmember functions that have all the privileges of member functions.
- The most common kinds of friend functions are overload operators.

```
// complex.h  -- The interface
class Complex
{
public :
   Complex(double, double) ;
   friend const Complex operator +(const Complex &, const Complex &);
private :
   double  r;
   double  i;
} ;
```

```
double d;
Complex  c1(2, 2), c2(4, 4), c3;
c3 = c1 + c2;
c3 = c1 + d;      // operato+(c1, d)
c3 = d + c2;      // operator+(d, c2)
```

```
// complex.cpp  --  The implementation

const Complex  operator +(const Complex & c1, const Complex & c2)

   return Complex(c1.r + c2.r, c1.i + c2.i);
```

Binary operator nonmember functions are symmetric!

Overloading the Assignment Operator =

```
int  x, y, z;
x = y = z = 1;
```

```
int  x, y, z;
x = (y = (z = 1));
```

```
String  x, y, z;
x = y = z = “Hello World!”;
```

```
String  x, y, z;
x = (y = (z = “Hello World!”));
```

```
x.operator=( y.operator=( z.operator=( “Hello World!” )));
```

The Assignment Operator =

- The default assignment operator only performs memberwise copies.
- A class should define its own assignment operator if it has pointer member variables.
- The assignment operator must be a member function.

```
// String.h  -- The interface
class String
{
public :
   String & operator = (const String &);
private :
   char *  s;
} ;
```

```
// String.cpp  --  The implementation

String & operator = (const String & str);
{
   if (this != &str) {            // “this” is a pointer to the current object.
      delete [ ] s;               // Avoid memory leak!
      s = new char[strlen(str.s) + 1];
      strcpy(s, str.s);
   }
   return *this;
}
```

```
String  s1(“Hello”), s2(“World”);
s1 = s2;
```

What happens when the default assignment operator is used?

```
String  s1(“Hello”);
s1 = s1;
```

What happens?

Return-By-Value v.s. Return-By-Reference

```
x.operator=( y.operator=( z.operator=( “Hello World!” )));
```

How many “Hello World!” is created using return-by-value?

```
x.operator=( y.operator=( z.operator=( “Hello World!” )));
```

7 6 5 4 3 2 1

copy constructor

copy constructor

copy constructor

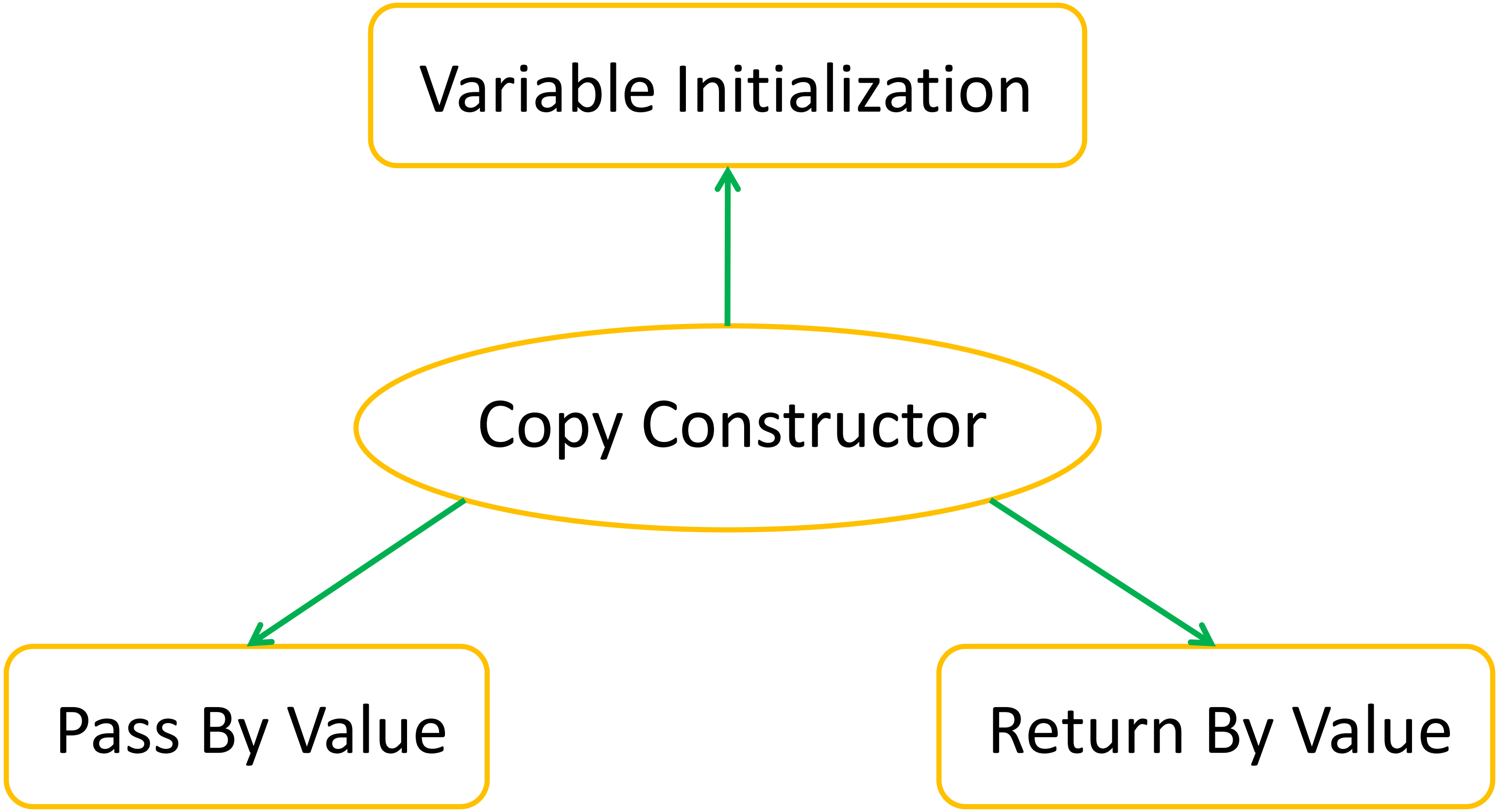
constructor

How many “Hello World!” is created using return-by-reference?

```
x.operator=( y.operator=( z.operator=( “Hello World!” )));
```

4 3 2 1

constructor



Variable Initialization

Copy Constructor

Pass By Value

Return By Value

The Big Three

- The copy constructor, the destructor, and the assignment operator are called the big three because if you need any of them, you need all three.
- If a class does not define these member functions, the compiler will define a default version for them.
- If a class has pointer member variables, the class should define its own version to handle dynamic memory allocation and deallocation.

L-Values & R-Values

- Consider x = x + 5.
- The x on the right-hand-side of = denotes the value (or storage content) of x. The value of x is called the r-value of x.
- The x on the left-hand-side of = denotes the address (or storage location) of x. The address of x is called the l-value of x.

```
String  s1, s2(“Hello World!”), s3(“Hello Universe!”);

s1 = s2 = s3;


(s1 = s2) = s3;     // Return-by-reference returns an address!

                    // What happens?


double d1, d2 = 1.0, d3 = 2.0;

(d1 = d2) = d3;     // This is legal for builtin data types!

                      // So, it is better legal for user-defined data types
```

Return-By-Reference

```
double & rbrFunction(double & x)
{
   return x;
}

double  d = 2.0;
cout  <<  rbrFunction(d)  <<  endl;       //  2.0
 rbrFunction(d) = 4.0;       //  Function call appears in lhs of =
cout  <<  d  <<  endl;                    //  4.0
```

Return Types

- Return-by-value: calls copy constructor, cannot be used as an l-value, can be changed directly.
- Return-by-constant-value: calls copy constructor, cannot be used as an l-value, cannot be changed directly.
- Return-by-reference: does not call copy constructor, can be used as an l-value, can be changed directly.
- Return-by-constant-reference: does not call copy constructor, cannot be used as an l-value, cannot be changed directly.

Overloading the Array Operator [ ]

```
CharPair  a(‘A’, ‘B’);
cout  <<  a[1]  <<  a[2]  <<  endl;

a[1] = ‘C’;
a[2] = ‘D’;
cout  <<  a[1]  <<  a[2]  <<  endl;

cout  <<  “Enter two letters (no spaces):\n”;
cin  >>  a[1]  >>  a[2];
cout  <<  “You entered:\n”;
cout  <<  a[1]  <<  a[2]  <<  endl;
```

Overloading the Array Operator [ ]

```
class CharPair
{
public:
   CharPair( );
   CharPair(char fVal, char sVal) : first(fVal), second(sVal) {  }
   char & operator [ ](int);
private:
   char first;
   char second;
}
```

Overloading the Array Operator [ ]

```
char & operator [ ](int index)
{
   if  (index == 1)
      return first;
   else if (index == 2)
      return second;
   else {
      cout << “Illegal index value.\n”);
      exit(1);
}
```

Overloading the Array Operator [ ]

- The operator [ ] can be overloaded to access elements in an aggregate data type.
- The operator [ ] must be a member function.
- The parameter of the operator [ ] must be an integer type, that is, enum, char, int, long or an unsigned version of one of these types.
- If the operator [ ] can appear in an expression on the lhs of an assignment operator, then it must return a reference.

Overloading <<

```
Complex c1(1, 1), c2(2, 2);
cout  <<  c1  <<  c2;

(cout  <<  c1)  <<  c2;
```

What are the parameter types of <<?

What is the return type of <<?

Can << be implemented as a member function?

```
friend ostream & operator<<(ostream & out, const Complex & c)
{
   out  <<  c.r  <<  “ +i ”  <<  c.i;
   return out;
}
```

Overloading >>

```
Complex  c1, c2;
cin  >>  c1  >>  c2;

(cin  >>  c1)  >>  c2;
```

What are the parameter types of >>?

What is the return type of >>?

Can >> be implemented as a member function?

```
friend istream & operator>>(istream & in, Complex & c)
{
   char  ch;
   in >> c.r  >>  ch;
   if (ch != ‘+’) { cout << “No + in Complex number.\n”; exit(1); }
   in >> ch;
   if (ch != ‘i’) { cout << “No i in Complex number.\n”; exit(1); }
   in >> c.i;
   return in;
}
```

Rules on Overloading Operators

- When overloading an operator, at least one parameter (one operand) of the resulting overloaded operator must be of a class type.
- Most operators can be overloaded as a member of the class or a friend of the class.
- The following operators can only be overloaded as members of the class: =, [ ], ->, ( ).

Rules on Overloading Operators

- You cannot create a new operator. All you can do is overloading existing operators.
- You cannot change the number of arguments that an operator takes. For example, you cannot change % from a binary to a unary operator when you overload %.

Rules on Overloading Operators

- You cannot change the precedence and associativity of an operator.
- The following operators cannot be overloaded: ., ::, sizeof, ?:, and .*.
- An overloaded operator cannot have default arguments.

How a Member Function Correctly Access Its Member Variables?

```
String  s1, s2, s3(“Hello World!”);
s1 = s3;                s1.operator=(s3);
s2 = s3;                s2.operator=(s3);
```

```
String & operator = (const String & str)
{
   if (this != &str) {
      delete [ ] s;
      s = new char[strlen(str.s) + 1];
      strcpy(s, str.s);
   }
   return *this;
}
```

How a Member Function Correctly Access Its Member Variables?

For each member function, the compiler automatically adds “this” as its first parameter!

```
String  s1, s2, s3(“Hello World!”);
s1 = s3;           s1.operator=(&s1, s3);
s2 = s3;           s2.operator=(&s1, s3);
```

```
String & operator = (String *this, const String & str)
{
   if (this != &str) {
      delete [ ] this->s;
      this->s = new char[strlen(str.s) + 1];
      strcpy(this->s, str.s);
   }
   return *this;
}
```