Week 10
Lecture No. 24
Compile Time Polymorphism
Operator Overloading

Operator overloading

- Consider the following class:

```
class Complex{
private:
   double real, img;
public:
   Complex Add(const Complex &);
   Complex Subtract(const Complex &);
   Complex Multiply(const Complex &);
   …
}
```

Operator overloading

- Function implementation:

```
Complex Complex::Add(
        const Complex & c1){
   Complex t;
   t.real = real + c1.real;
   t.img  = img  + c1.img;
   return t;
}
```

Operator overloading

- The following statement:

```
Complex c3 = c1.Add(c2);
```

Adds the contents of **`c2`** to **`c1`** and assigns it to **`c3`** (copy constructor)

Operator overloading

- To perform operations in a single mathematical statement e.g:

```
c1+c2+c3+c4
```

- We have to explicitly write:

```
c1.Add(c2.Add(c3.Add(c4)))
```

Operator overloading

- Alternative way is:

```
t1 = c3.Add(c4);
t2 = c2.Add(t1);
t3 = c1.Add(t2);
```

Operator overloading

- If the mathematical expression is big:
  - Converting it to C++ code will involve complicated mixture of function calls
  - Less readable
  - Chances of human mistakes are very high
  - Code produced is very hard to maintain

Operator overloading

- C++ provides a very elegant solution:

  "Operator overloading"

- C++ allows you to overload common operators like **+**, **−** or ***** etc…

- Mathematical statements don't have to be explicitly converted into function calls

Operator overloading

- Assume that operator **`+`** **`has`** been overloaded
- Actual C++ code becomes:

```
c1+c2+c3+c4
```

- The resultant code is very easy to read, write and maintain

Operator overloading

- C++ automatically overloads operators for pre-defined types
- Example of predefined types:

```
int
float
double
char
long
```

Operator overloading

- Example:

```
float x;
int y;
x = 102.02 + 0.09;
Y = 50 + 47;
```

Operator overloading

The compiler probably calls the correct overloaded low level function for addition i.e:

```
// for integer addition:
Add(int a, int b)


// for float addition:
Add(float a, float b)
```

Operator overloading

- Operator functions are not usually called directly
- They are automatically invoked to evaluate the operations they implement

Operator overloading

- List of operators that can be overloaded in C++:

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

Operator overloading

►List of operators that can't be overloaded:

```
.     .*     ::     ?:     #     ##
```

►Reason: They take name, rather than value in their argument except for **?:**

►**?:** is the only ternary operator in C++ and can't be overloaded

Operator overloading

- The precedence of an operator is **NOT** affected due to overloading
- Example:

```
c1*c2+c3
```

```
c3+c2*c1
```

both yield the same answer

Operator overloading

- Associativity is **NOT** changed due to overloading
- Following arithmetic expression always is evaluated from left to right:

```
c1 + c2 + c3 + c4
```

Operator overloading

- Unary operators and assignment operator are right associative, e.g:

  `a=b=c` is same as `a=(b=c)`

- All other operators are left associative:

  `c1+c2+c3` is same as

  `(c1+c2)+c3`

Operator overloading

- Always write code representing the operator
- Example:

  Adding subtraction code inside the + operator will create chaos

Operator overloading

- Creating a new operator is a syntax error (whether unary, binary or ternary)
- You cannot create $

Operator overloading

- Arity of an operator is NOT affected by overloading
- Example:

  Division operator will take exactly two operands in any case:

```
b = c / d
```

Binary operators

- Binary operators act on two quantities
- Binary operators:

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	,	->*	->		

Overloading Binary Operators

- A binary operator can be overloaded as a non-static member function with one parameter or as a non-member function with two parameters (one of those parameters must be either a class object or a reference to a class object).
- A non-member operator function is often declared as friend of a class for performance reasons.

Binary operators

- General syntax:

  Member function:

```
TYPE1 CLASS::operator B_OP(
                          TYPE2 rhs){
   ...
}
```

Binary operators

- General syntax:

  Non-member function:

```
TYPE1 operator B_OP(TYPE2 lhs,
                    TYPE3 rhs){
   ...
}
```

Binary operators

- The "`operator OP`" must have at least one formal parameter of type class (user defined type)
- Following is an error:

```
int operator + (int, int);
```

Binary operators

- Overloading + operator:

```
class Complex{
private:
   double real, img;
public:
   …
   Complex operator +(const
            Complex & rhs);
};
```

Binary operators

```
Complex Complex::operator +(
        const Complex & rhs){
   Complex t;
   t.real = real + rhs.real;
   t.img = img + rhs.img;
   return t;
}
```

Binary operators

- The return type is Complex so as to facilitate complex statements like:

```
Complex t = c1 + c2 + c3;
```

- The above statement is automatically converted by the compiler into appropriate function calls:

```
(c1.operator +(c2)).operator
+(c3);
```

Binary operators

- If the return type was **void**,

```
class Complex{
   ...
public:
   void operator+(
        const Complex & rhs);
};
```

Binary operators

```
void Complex::operator+(const
          Complex & rhs){
     real = real + rhs.real;
     img = img + rhs.img;
};
```

Binary operators

- we have to do the same operation `c1+c2+c3` as:

```
c1+c2

c1+c3

// final result is stored in c1
```

Binary operators

- Drawback of void return type:
  - Assignments and cascaded expressions are not possible
  - Code is less readable
  - Debugging is tough
  - Code is very hard to maintain

Binary operators

- The above examples don't handle the following situation:

```
Complex c1;

c1 + 2.325
```

- To do this, we have to modify the **`Complex`** class

Binary operators

- Modifying the complex class:

```
class Complex{

   ...

   Complex operator+(const
      Complex & rhs);

   Complex operator+(const
      double& rhs);

};
```

Binary operators

```
Complex operator + (const double&
                                rhs){
   Complex t;
   t.real = real + rhs;
   t.img = img;
   return t;
}
```

Binary operators

- Now suppose:

```
Complex c2, c3;
```

- We can do the following:

```
Complex c1 = c2 + c3;
```

and

```
Complex c4 = c2 + 235.01;
```

Binary operators

- But problem arises if we do the following:

```
Complex c5 = 450.120 + c1;
```

- The + operator is called with reference to `450.120`

- No predefined overloaded + operator is there that takes **`Complex`** as an argument

Binary operators

► Now if we write the following two functions to the class, we can add a **`Complex`** to a **`real`** or vice versa:

```
Class Complex{

    …

    friend Complex operator + (const
    Complex & lhs, const double & rhs);

    friend Complex operator + (const
    double & lhs, const Complex & rhs);

}
```

Binary operators

```
Complex operator +(const Complex &
        lhs, const double& rhs){

    Complex t;
    t.real = lhs.real + rhs;
    t.img = lhs.img;
    return t;
}
```

Binary operators

```
Complex operator + (const double &
        lhs, const Complex & rhs){

    Complex t;
    t.real = lhs + rhs.real;
    t.img = rhs.img;
    return t;
}
```

Binary operators

```
Class Complex{

   …

   Complex operator + (const
                              Complex &);

   friend Complex operator + (const
            Complex &, const double &);

   friend Complex operator + (const
            double &, const Complex &);

};
```

Binary operators

- Other binary operators are overloaded very similar to the + operator as demonstrated in the above examples
- Example:

```
Complex operator * (const Complex &
        c1, const Complex & c2);

Complex operator / (const Complex &
        c1, const Complex & c2);

Complex operator - (const Complex &
        c1, const Complex & c2);
```

Assignment operator

- Consider a string class:

```
class String{
    int size;
    char * bufferPtr;
public:
    String();
    String(char *);
    String(const String &);
    …
};
```

Assignment operator

```
String::String(char * ptr){
    if(ptr != NULL){
            size = strlen(ptr);
            bufferPtr = new char[size+1];
            strcpy(bufferPtr, ptr);
    }
    else{
            bufferPtr = NULL; size = 0; }
}
```

Assignment operator

```
String::String(const String & rhs){
    size = rhs.size;
    if(rhs.size != 0){
            bufferPtr = new char[size+1];
            strcpy(bufferPtr, ptr);
    }
    else
            bufferPtr = NULL;
}
```

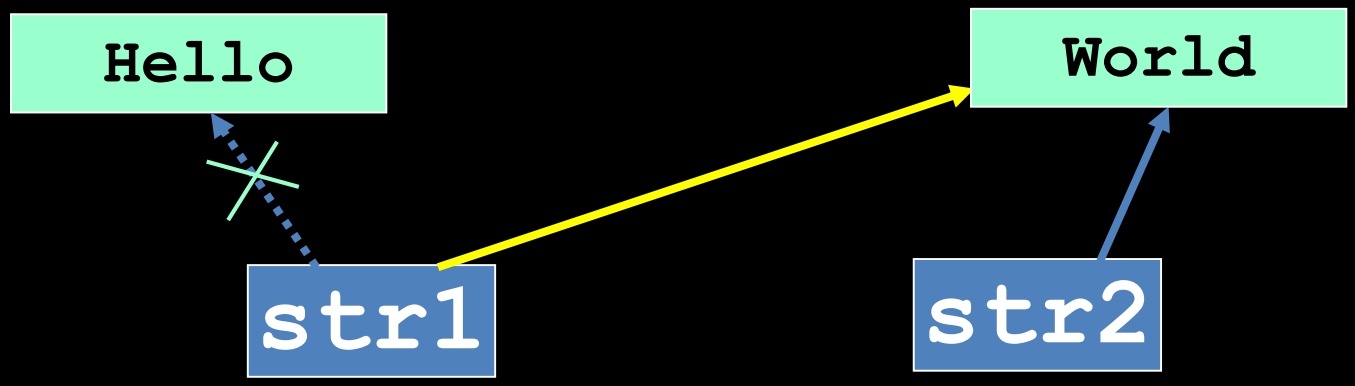
Assignment operator

```
int main(){
   String str1(“Hello");
   String str2(“World”);
   str1 = str2;
   return 0;
}
```

Member wise copy assignment

Assignment operator

- Result of `str1 = str2` (memory leak)



Assignment operator

- Modifying:

```
class String{
   …
public:
   …
   void operator =(const String &);
};
```

Assignment operator

```
void String::operator = (const String & rhs){
    size = rhs.size;
    if(rhs.size != 0){
        delete [] bufferPtr;
        bufferPtr = new char[rhs.size+1];
        strcpy(bufferPtr,rhs.bufferPtr);
    }
    else
            bufferPtr = NULL;
}
```

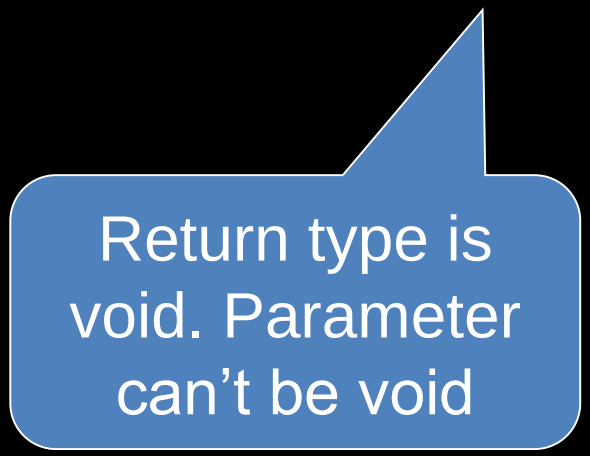
Assignment operator

```
int main(){
   String str1(“ABC");
   String str2(“DE”), str3(“FG”);
   str1 = str2;              // Valid…
   str1 = str2 = str3;       // Error…
   return 0;
}
```

Assignment operator

- `str1=str2=str3` is resolved as:

```
str1.operator=(str2.operator=
                        (str3))
```



Assignment operator

►Solution: modify the `operator =` function as follows:

```
class String{
   …
public:
   …
   String & operator = (const
                        String &);
};
```

Assignment operator

```
String & String :: operator = (const String &
                                        rhs){
   size = rhs.size;
   delete [] bufferPtr;
   if(rhs.size != 0){
      bufferPtr = new char[rhs.size+1];
      strcpy(bufferPtr,rhs.bufferPtr);
   }
   else bufferPtr = NULL;
   return *this;
}
```

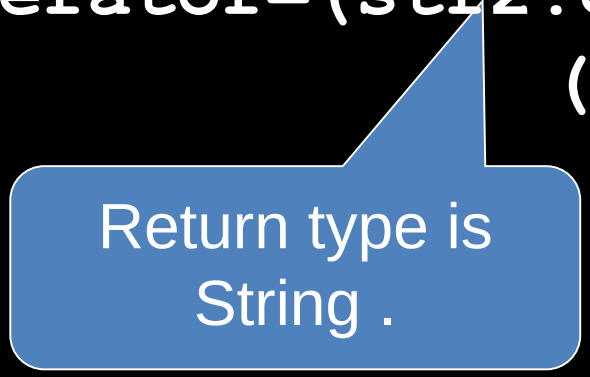
Assignment operator

```
void main(){
    String str1("AB");
    String str2("CD"), str3("EF");
    str1 = str2;
    str1 = str2 = str3;  // Now valid…
}
```

Assignment operator

- `str1=str2=str3` is resolved as:

```
str1.operator=(str2.operator=
                (str3))
```

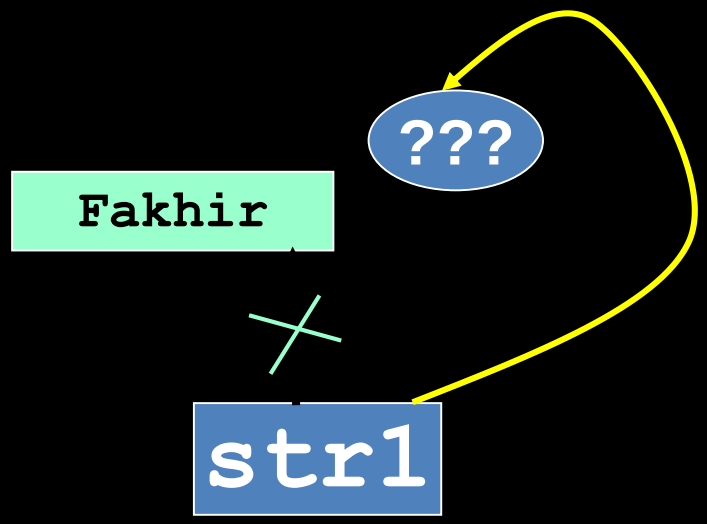


Assignment operator

```
int main(){
   String str1("Fakhir");
   // Self Assignment problem…
   str1 = str1;
   return 0;
}
```

Assignment operator

- Result of `str1 = str1`




```
    …
  //    size = rhs.size;
  //    delete [] bufferPtr;
…
```

Assignment operator

```
String & String :: operator = (const
                                    String & rhs){
 if(this != &rhs){
   size = rhs.size;
   delete [] bufferPtr;
   if(rhs.bufferPtr != NULL){
      bufferPtr = new char[rhs.size+1];
      strcpy(bufferPtr,rhs.bufferPtr);
   }
   else bufferPtr = NULL;
 }
 return *this; }
```

Assignment operator

- Now self-assignment is properly handled:

```
int main(){
   String str1("Fakhir");
   str1 = str1;
   return 0;
}
```

Assignment operator

►Solution: modify the **operator=** function as follows:

```
class String{
   …
public:
   …
   const String & operator=
                  (const String &);
};
```

Stream Insertion and Extraction operator

Stream Insertion operator

- Often we need to display the data on the screen
- Example:

```
int i=1, j=2;
cout << "i= "<< i << "\n";
Cout << "j= "<< j << "\n";
```

We must know following things before we start overloading these operators.

1) cout is an object of ostream class and cin is an object istream class

2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them friend.

Stream Insertion operator

```
Complex c1;
cout << c1;
cout << c1 << 2;
```

// Compiler error: binary '<<' : no operator // defined which takes a right-hand // operand of type ‘class Complex’

Stream Insertion operator

```
class Complex{
      …
public:
      …
      void operator << (const
                        Complex & rhs);
};
```

Stream Insertion operator

```
int main(){
     Complex c1;
     cout << c1;          // Error
     c1 << cout;
     c1 << cout << 2; // Error
     return 0;
};
```

Stream Insertion operator

```
class Complex{
    …
public:
    …
    void operator << (ostream &);
};
```

Stream Insertion operator

```
void Complex::operator <<
        (ostream & os){
    os  << '('  << real
        << ',' << img << ')';
}
```

Stream Insertion operator

```
class Complex{

   ...

   friend ostream & operator <<
   (ostream & os, const Complex
                        & c);

};
```

Note: return type is NOT `const`

Note: this object is NOT `const`

Stream Insertion operator

// we want the output as: *(real, img)*

```
ostream & operator << (ostream &
                os, const Complex & c){
    os << '(' << c.real
       << ','
       << c.img << ')';
    return os;
}
```

Stream Insertion operator

```
Complex c1(1.01, 20.1),
        c2(0.01, 12.0);


cout << c1 << endl << c2;
```

Stream Insertion operator

Output:

```
( 1.01 , 20.1 )
( 0.01 , 12.0 )
```

Stream Insertion operator

```
cout << c1 << c2;
```

is equivalent to

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

Stream Extraction Operator

- Overloading “>>” operator:

```
class Complex{

   ...

   friend istream & operator
   >> (istream & i, Complex &
         c);

};
```

Note: this object is NOT `const`

Stream Extraction Operator

```
istream & operator << (istream
        & in, Complex & c){
    in >> c.real;
    in >> c.img;
    return in;
}
```

Stream Extraction Operator

- Main Program:

```
Complex c1(1.01, 20.1);

cin >> c1;

// suppose we entered
// 1.0025 for c1.real and
// 0.0241 for c1.img

cout << c1;
```

Stream Extraction Operator

Output:

```
( 1.0025 , 0.0241 )
```

Other Binary operators

- Overloading comparison operators:

```
class Complex{
public:
   bool operator == (const Complex & c);
//friend bool operator == (const
//Complex & c1, const Complex & c2);
   bool operator != (const Complex & c);
//friend bool operator != (const
//Complex & c1, const Complex & c2);
   …
};
```

Other Binary operators

```
bool Complex::operator ==(const
Complex & c){
   if((real == c.real) &&
      (img == c.img)){
        return true;
   }
   else
        return false;
}
```

Other Binary operators

```
bool operator ==(const
Complex& lhs, const Complex& rhs){
   if((lhs.real == rhs.real) &&
      (lhs.img == rhs.img)){
        return true;
   }
   else
        return false;
}
```

Other Binary operators

```
bool Complex::operator !=(const
Complex & c){
   if((real != c.real) ||
      (img != c.img)){
        return true;
   }
   else
        return false;
}
```

Unary Operators

Unary Operators

- Unary operators:
  - **& * + - ++ -- ! ~**
- Examples:

--x

-(x++)

!(*ptr ++)

Unary Operators

- Unary operators are usually prefix, except for ++ and --
- ++ and -- both act as prefix and postfix
- Example:

```
h++;

g-- + ++h - --i;
```

Unary Operators

- General syntax for unary operators:

  Member Functions:

  TYPE & operator OP ();

  Non-member Functions:

  Friend TYPE & operator OP (TYPE & t);

Unary Operators

- Overloading unary '-':

```
class Complex{
...
Complex  operator - ();
// friend Complex operator
//                  -(Complex &);
}
```

Unary Operators

- Member function definition:

```
Complex Complex::operator -(){
    Complex temp;
    temp.real = -real;
    temp.img = -img;
    return temp;
}
```

Unary Operators

Complex c1(1.0 , 2.0), c2;

c2 = -c1;

// c2.real = -1.0

// c2.img = -2.0

► Unary '+' is overloaded in the same way

Unary Operators

- Unary operators are usually prefix, except for `++` and `--`
- `++` and `--` both act as prefix and postfix
- Example:

```
-h++;

-g-- + ++h - --i;
```

Unary Operators

- Behavior of ++ and -- for pre-defined types:
  - Post-increment **++** :
    - Post-increment operator ++ increments the current value and then returns the previous value
  - Post-decrement -- :
    - Works exactly like post ++

Unary Operators

- Example:

```
int x = 1, y = 2;
cout << y++ << endl;
cout << y;
```

- Output:

```
2
3
```

Unary Operators

- Example:

```
int y = 2;
y++++;          // Error
y++ = x;        // Error
```

Post-increment ++ returns by value, hence an error while assignment

Unary Operators

- Behavior of ++ and -- for pre-defined types:
  - Pre-increment ++ :
    - Pre-increment operator ++ increments the current value and then returns it's reference
  - Pre-decrement -- :
    - Works exactly like Pre-increment ++

Unary Operators

- Example:

```
int  y = 2;
cout << ++y << endl;
cout << y << endl;
```

- Output:

```
3
3
```

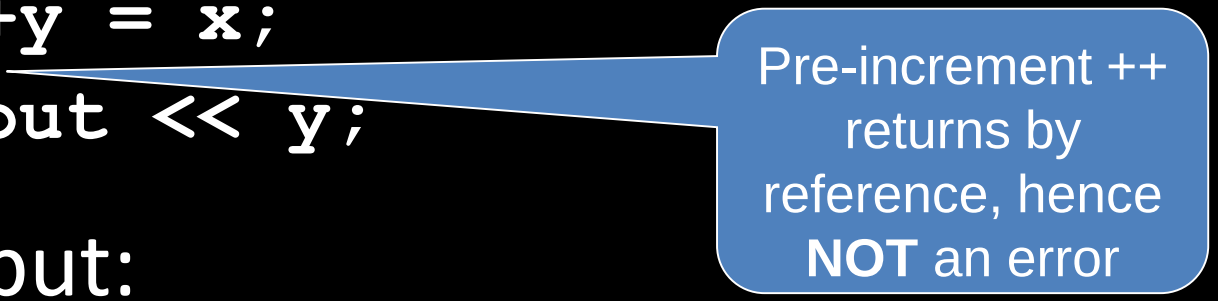
Unary Operators

- Example:

```
int x = 2, y = 2;
++++y;
cout << y;
++y = x;
cout << y;
```

- Output:

```
4
2
```



Unary Operators

- Example (Pre-increment):

```
class Complex{
   double real, img;
public:
...
   Complex & operator ++ ();
// friend Complex & operator //
                ++(Complex &);
}
```

Unary Operators

- Member function definition:

```
Complex & Complex::operator++(){
   real = real + 1;
   return * this;
}
```

Unary Operators

- Friend function definition:

```
Complex & operator ++ (Complex
                         & h){
  h.real += 1;
  return h;
}
```

Unary Operators

```
Complex h1, h2, h3;

++h1;
```

► Function `operator++()` returns a reference so that the object can be used as an *lvalue*

```
++h1 = h2 + ++h3;
```

Unary Operators

- How does a compiler know whether it is a pre-increment or a post-increment ?

Unary Operators

- A post-fix unary operator is implemented using:

  Member function with 1 dummy int argument

  OR

  Non-member function with two arguments

Unary Operators

- In post increment, current value of the object is stored in a temporary variable
- Current object is incremented
- Value of the temporary variable is returned

Unary Operators

- Post-increment operator:

```
class Complex{
...
   Complex operator ++ (int);
// friend Complex operator
//   ++(const Complex &, int);
}
```

Unary Operators

- Member function definition:

```
Complex Complex::operator ++
                          (int){
   complex t = *this;
   real += 1;
   return t;
}
```

Unary Operators

- Friend function definition:

```
Complex operator ++ (const
                Complex & h, int){
   complex t = h;
   h.real += 1;
   return t;
}
```

Unary Operators

- The dummy parameter in the operator function tells compiler that it is post-increment
- Example:

```
Complex h1, h2, h3;
h1++;
h3++ = h2 + h3++; // Error…
```

Unary Operators

- The *pre* and *post* decrement operator `--` is implemented in exactly the same way

Self Reading

Operator Overloading from
C++ How to Program (Dietal & Dietal)