CS217 OBJECT ORIENTED PROGRAMMING

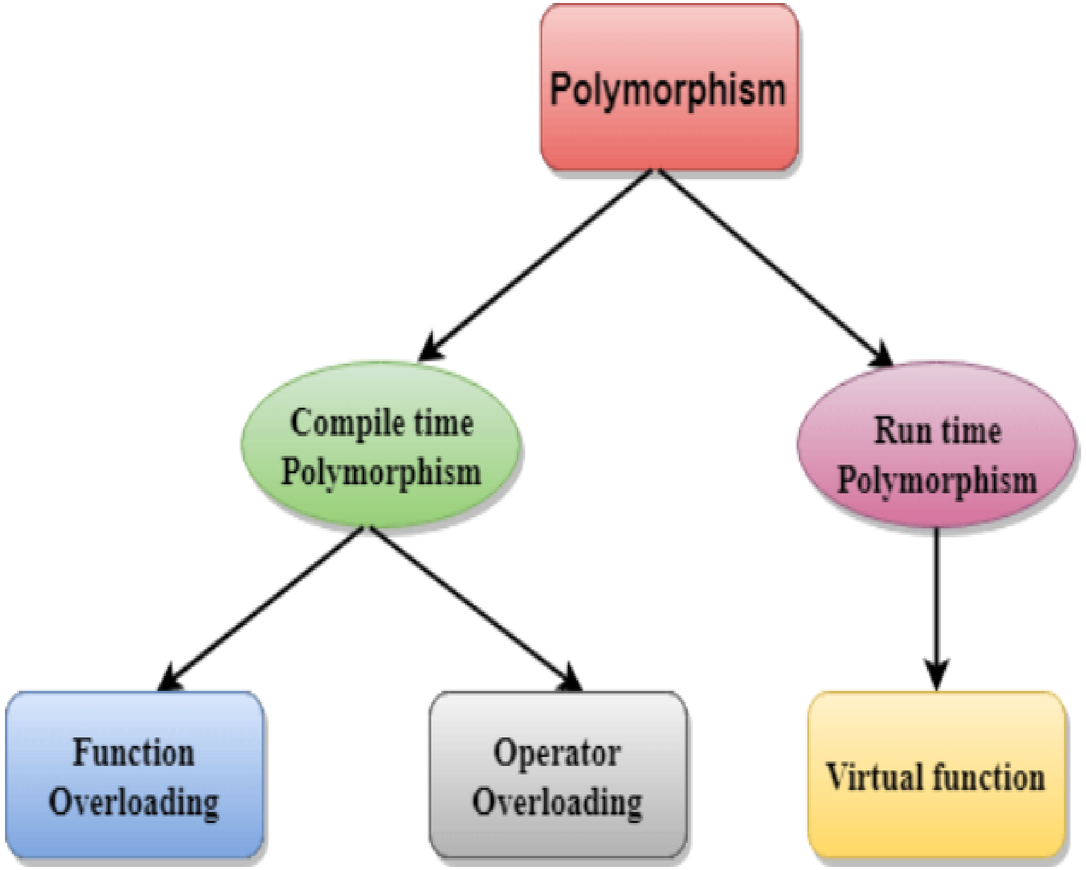
Spring 2020

NATIONAL UNIVERSITY OF COMPUTER & EMERGING SCIENCES

POLYMORPHISM

- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.

- **A real-life example of polymorphism**: A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.



Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

OVERLOADING

- Function Overloading
- Operator Overloading

FUNCTION OVERLOADING

- Function Overloading is defined as the process of having two or more function with the same name, but different in parameters(different signatures).

- The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

```cpp
class Cal {
    public:
        static int add(int a,int b){
                return a + b;
        }

        static int add(int a, int b, int c){
                return a + b + c;
        }
};

int main() {
    Cal C;
    cout << C.add(10, 20) <<endl;
    cout << C.add(12, 20, 23);
   return 0;
}
```

OPERATOR OVERLOADING (CHAPTER 11, BOOK#1)

- C++ fundamental data types can be used with C++'s rich collection of operators.
  - Operators provide you with a concise notation for expressing manipulations of data of fundamental types.
- You can use operators with user-defined types as well.
  - Although C++ does not allow new operators to be created, it does allow most existing operators to be **overloaded** so that, when they're used with objects, they have meaning appropriate to those objects .
  - One example of an overloaded operator built into C++ is **<<**, which is used both as the stream insertion operator and as the bitwise left-shift operator.
  - **>>** is also overloaded; it's used both as the stream extraction operator and as the bitwise right-shift operator. Both of these operators are overloaded in the C++ Standard Library.

OPERATOR OVERLOADING (CHAPTER 11, BOOK#1)

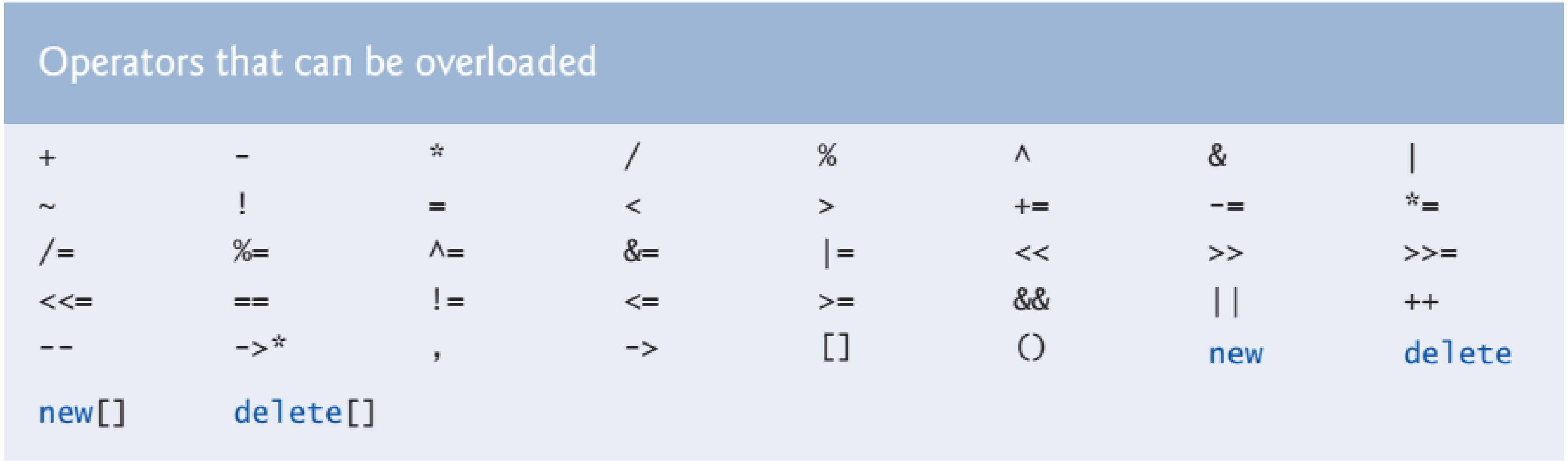
- An operator is overloaded by writing a non-static member function definition or global function definition as you normally would, except that the function name now becomes the keyword `operator` followed by the symbol for the operator being overloaded.
  - For example, the function name `operator+` would be used to overload the addition operator (**+**).

- When operators are overloaded as member functions, they must be *non-static*, because they must be called on an object of the class and operate on that object.

OPERATOR OVERLOADING (CHAPTER 11, BOOK#1)

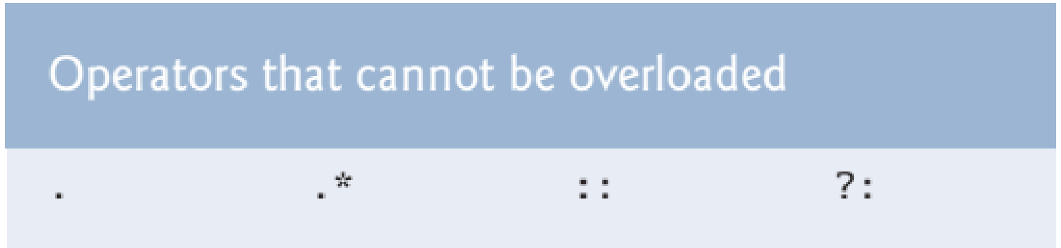
- To use an operator on class objects, that operator ***must* be overloaded**—with the following exceptions.
  1. The **assignment operator** (**=**) may be used with objects (without overloading) to perform member wise assignment of the object's data members.
  2. The **address** (**&**) and **comma** (**,**) operators may also be used with objects of any class without overloading.

- Overloading is especially appropriate for mathematical classes.



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

Fig. 11.1 | Operators that can be overloaded.



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

Fig. 11.2 | Operators that cannot be overloaded.

PRECEDENCE, ASSOCIATIVITY AND NUMBER OF OPERANDS

- The precedence of an operator cannot be changed by overloading.
  - However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.

- The associativity of an operator (i.e., whether the operator is applied right-to-left or left-to-right) cannot be changed by overloading.

- It isn't possible to change the "arity" of an operator (i.e., the number of operands an operator takes):
  - Overloaded unary operators remain unary operators; overloaded binary operators remain binary operators.
  - C++'s only ternary operator (?:) cannot be overloaded.

- Operators &, *, + and - all have both unary and binary versions; these unary and binary versions can each be overloaded.
- Unary operator overloading (++,--,+,-,*,&)
- Binary operator overloading

- It isn’t possible to create new operators; only existing operators can be overloaded.

- The meaning of how an operator works on objects of fundamental types cannot be changed by operator overloading.
  - You cannot, for example, change the meaning of how + adds two integers.

- Overloading an assignment operator(=) and an addition operator(+) to allow statements like `object2 = object1 + object2` does not imply that the `+=` operator is also overloaded to allow statements such as `object2 += object1;`
  - Such behavior can be achieved only by explicitly overloading operator += for that class.

- When overloading `()`, `[]`, `->` or any of the assignment operators, the operator overloading function must be declared as a class member. For the other operators, the operator overloading functions can be class members or standalone functions.

- When an operator function is implemented as a member function, the leftmost operand must be an object (or a reference to an object) of the operator's class.
  - Operator member functions of a specific class are called (implicitly by the compiler) only when the left operand of a binary operator is specifically an object of that class, or when the single operand of a unary operator is an object of that class.

- If the left operand must be an object of a different class or a fundamental type, this operator function must be implemented as a global function.

- A global operator function can be made a **`friend`** of a class if that function must access **`private`** or **`protected`** members of that class directly.

- **Global operator function vs member operator function**

```
class Counter
{
private:
    int count;
public:
    Counter() : count(0)
    { }
    int get_count()
    { return count; }

    void operator ++ ()               //  increment (prefix)
    {
        ++count;
    }
};
int main(){
    Counter c1, c2; int i=10;
    ++c1;
    ++i;
    ++c2;
    ++c2;
    c1 = ++c2; //error
    cout << “\nc1=” << c1.get_count();
    cout << “\nc2=” << c2.get_count() << endl;
    return 0;
}
```

OPERATOR RETURN VALUES

- What if we have following statement in main():

```
c1 = ++c2
```

  - Because we have defined the ++ operator to have a return type of `void` in the `operator++()` function, while in the assignment statement it is being asked to return a variable of type `Counter`
  - We can't use ++ to increment counter object in assignments, it must always stand alone with its operand.
  - To make it possible to use our homemade operator++() in assignment expressions, we must provide a way for it to return a value

OVERLOADING UNARY OPERATOR

- A unary operator for a class can be overloaded as a non-static member function with no arguments or as a global function with one argument that must be an object (or a reference to an object) of the class.
- It works only with one class objects.
- It is a overloading of an operator operating on a single operand.

```
class Counter
{
  int count;

public:

  Counter() : count(0){ }

  int get_count()
     { return count; }

  Counter operator ++ (){
     Counter temp;
     temp.count = ++count;
     return temp;
}
};
```

```
int main()
{
  Counter c1, c2;
  cout << “\nc1=” << c1.get_count(); //0
  cout << “\nc2=” << c2.get_count(); //0

  ++c1; //1

  c2 = ++c1; //c1=2, c2=2
  cout << “\nc1=” << c1.get_count(); //2
  cout << “\nc2=” << c2.get_count(); //2

  return 0;
}
```

```
class Counter
{
  int count;


public:


  Counter() : count(0){}

  int get_count()
     { return count; }


 Counter operator ++ (int)

 {
     Counter temp;
     temp.count = count++;
     return temp;
  }



 Counter operator ++ ()

  {
     Counter temp;
     temp.count = ++count;
     return temp;
  }


};
```

```
int main()
{
  Counter c1, c2;
  cout << “\nc1=” << c1.get_count();// 0
  cout << “\nc2=” << c2.get_count();// 0


  ++c1; // 1

  c2 = ++c1; // c1=2, c2= 2
  cout << “\nc1=” << c1.get_count(); // 2
  cout << “\nc2=” << c2.get_count(); // 2


  c2 = c1++; // c2= 2, c1= 3
  cout << “\nc1=” << c1.get_count(); // 3
  cout << “\nc2=” << c2.get_count(); // 2



return 0;
}
```

FOR UNARY - OPERATOR (NEGATION)

```
 Vector operator – ( ) const
{
      Vector temp;
      temp.x = -x;
      temp.y = -y;
      return temp;
}
```

FOR PREFIX ++ OPERATOR

```
void operator ++ ( )
{
     ++x;
     ++y;
}
```

(Works the same way for prefix decrement operator)

FOR POSTFIX ++ OPERATOR

```
Vector operator ++ (int)
{
     Vector temp;
     temp.x = x++;
     temp.y = y++;
     return temp;
}
```

FRIEND FUNCTION AND OPERATOR OVERLOADING

- Friend function using operator overloading offers better flexibility to the class.
  - These functions are not a members of the class and they do not have '**this**' pointer.
  - When you overload a unary operator you have to pass one argument.
  - When you overload a binary operator you have to pass two arguments.

- Friend function can access private members of a class directly.

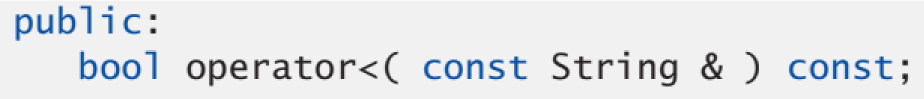
```
friend return-type operator operator-symbol (Variable 1, Varibale2)
{
     //Statements;
}
```

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


class UnaryFriend
{
 int a=10, b=20, c=30;


public:
 void getvalues()
 {
   cout<<"Values of A, B & C\n";
   cout<<a<<"\n"<<b<<"\n"<<c<<"\n";
  }


 void show()
 {
   cout<<a<<"\n"<<b<<"\n"<<c<<"\n";
 }


void friend operator-(UnaryFriend &x);



};
```

```
void operator-(UnaryFriend &x)
{
     x.a = -x.a; //Object name must be used
     x.b = -x.b;
     x.c = -x.c;
}


int main()
{
     UnaryFriend x1;
     x1.getvalues();
     cout<<"Before Overloading\n";
     x1.show();
     cout<<"After Overloading \n";
     -x1; //operator-(x)
      x1.show();
      return 0;
}
```

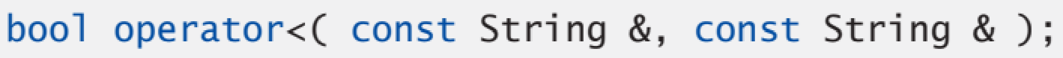
OVERLOADING BINARY OPERATORS

- A binary operator can be overloaded as a non-static member function with one parameter or as a global function with two parameters (one of those parameters must be either a class object or a reference to a class object).
- E.g. string y,z;
  - When overloading binary operator **<** as a non-static member function of a **String** class with one argument, if **y** and **z** are String-class objects, then **y < z** is treated as if **y.operator<(z)** had been written, invoking the **operator<** member function.



```
public:
   bool operator<( const String & ) const;
```

  - As a global function, binary operator **<** must take two arguments—one of which must be an object (or a reference to an object) of the class. If **y** and **z** are String-class objects or references to String-class objects, then **y < z** is treated as if the call **operator<(y, z)** had been written in the program, invoking global-function operator< declared



```
bool operator<( const String &, const String & );
```

```
class Vector
{
  int x, y;
  public:
  Vector( int x, int y)
  {
        this->x = x; this->y = y;
  }
  void printXY()
  {
        cout << "x: " << x << endl;
        cout << "y: " << y << endl;
  }
  Vector operator+(const Vector& ob)
  {
        Vector temp;
        temp.x = x + ob.x;//10+8 = 18
        temp.y = y + ob.y;//15+6 = 21
        return temp;
  }
};
```

```
int main()
{
    Vector v1(10, 15);//v1.x=10,v1.y=15
    Vector v2(8, 6);//v2.x=8,v2.y=6
    Vector v3 = v1 + v2;//v1.operator+(v2)

    v3.printXY();
}

// prints x: 18 & y: 21
```

ASSIGNMENT (OVERLOADING):

- Overload following unary operators in Counter class:
- -- (pre/post fix), *, - (negation), + (positive)

- Findout some global operator function working for more than one class (Application Wise).

- Binary Operator Overloading: Global Function (application wise program),

- String Class: Readout (library): Global Operator Function, Member Operator Function

- Overload these binary operators (globally and as a member function):
- *, +, -, /, %