Advanced Programming Concepts with C++

Operator Overloading

and some review!

Fall 2021
CSI2372[A]

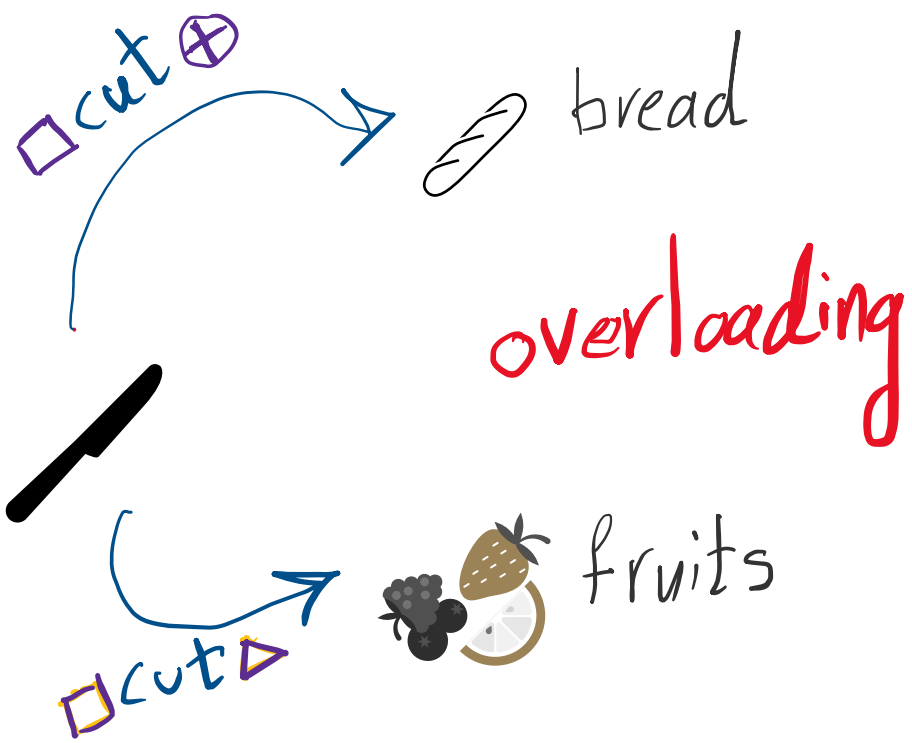
Amir EAMAN
Postdoc Researcher
Faculty of Engineering
uOttawa

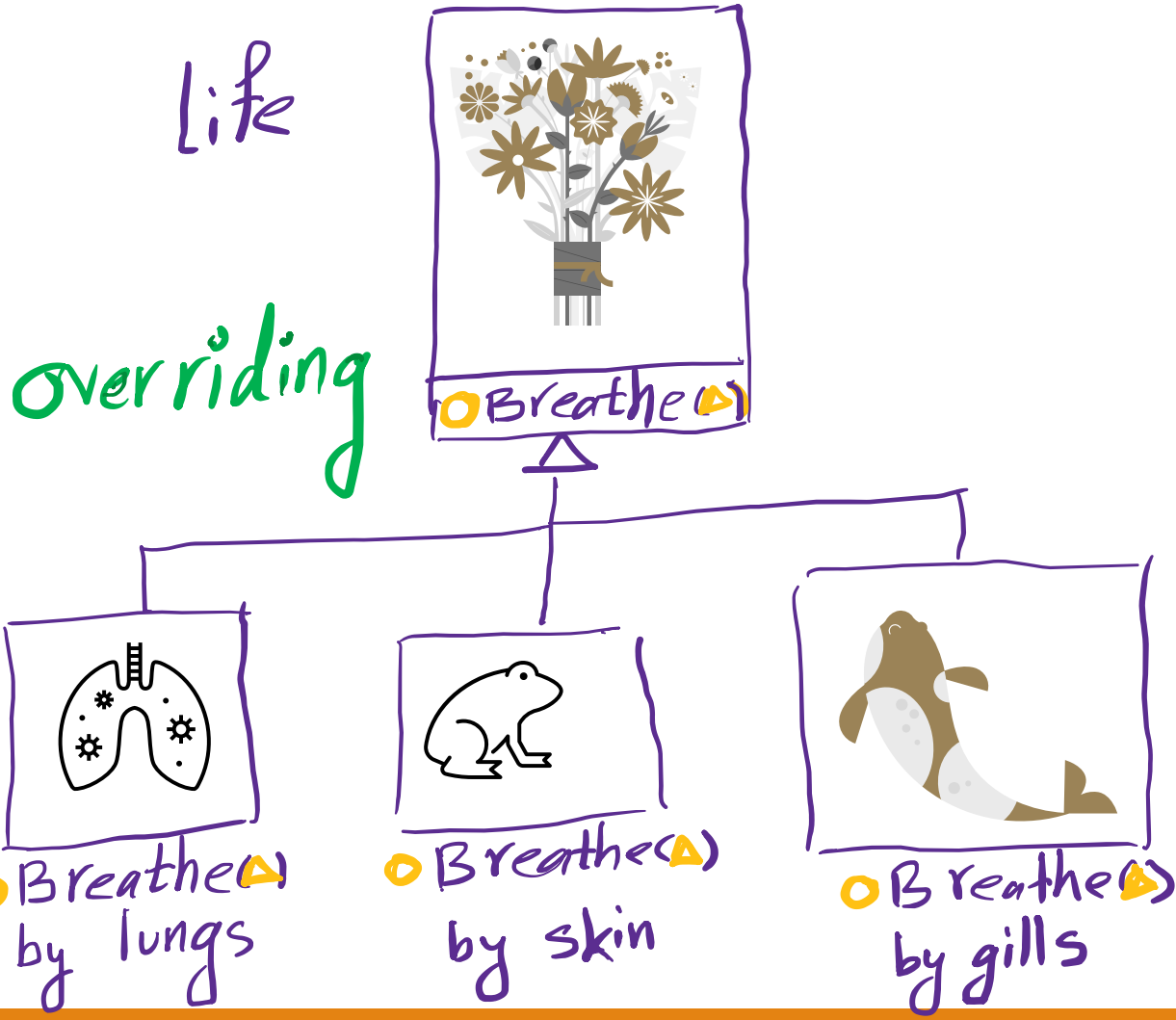
Review0: Function overriding VS. Function overloading

Polymorphism associates many meaning to one function name (this is called **overriding**), which happens at run-time through late-binding mechanism. So, late binding and polymorphism and virtual functions are all the same concept.

1. **Overriding**: We say a function definition is overridden when it is a virtual function in the base class and its definition is changed in a derived class (**the overridden function has *exactly* the same function name, return type, input parameter types, and number of input parameters as the virtual function in the base class**). If in a derived class we change the definition of a non virtual base class function, it is called **redefining**.
   - Remember that if the definition of a virtual function in a derived class is different from the virtual function of the base class (in function name or return type or input parameter types, or number of input parameters) it is not overriding, and we can check this with *optional* keyword `override` which appears after the declaration of the function in the derived class.
2. **Overloading**: We have different function definitions with the same function name, that is function definitions either **have different parameters types or different number of parameters** but they all have the same function name.

Overriding VS. **Overloading** – graphical explanations!





Review1: Copy Assignment Operator (CAO) and Copy Constructor (CC)

```
1  #include <iostream>
2  class MyClass
3  {
4      int a;
5  public:
6      MyClass() { a = 0; } // 1
7      MyClass(int i) { a = i; } //2
8      MyClass(int i, int j) { a = i-j; } // 3
9      MyClass(const MyClass & ini) { a = ini.a; } // 4
10     MyClass& operator=(const MyClass& ini) { a = 0; return *this; } // 5
11     ~MyClass() {} //6
12 };
13 int main()
14 {
15     MyClass c1; //calls 1
16     MyClass c2 { 2 }; //calls 2
17     MyClass c3 ( 3,4 );//calls 3
18     MyClass c4 = c3; //calls 4 -- equivalent to MyClass c4 { c3};which calls 4
19     c3 = c2;//calls 5
20
21 }//calls the destructor for c4,c3,c2,c1
```

overload

initialization → construction.

copy assignment.

copy constructor (CC)

assignment.

Review2: Overriding and Polymorphism -example

```
1   #include <iostream>
2   class BaseClass
3   {
4   public:
5       BaseClass() { }
6       virtual void vFunction() { std::cout << "the vFucntion of Base #1\n"; } // #1
7       void nvFunction() { std::cout << "the nvFucntion of Base #2\n"; }//#2
8       ~BaseClass() {}
9   };
10  class DerivedClass:public BaseClass
11  {
12  public:
13      DerivedClass() {};
14      virtual void vFunction() { std::cout << "the vFucntion of Drived #3\n"; } //overloading 1    //#3
15      void nvFunction() { std::cout << "the nvFucntion of Derived #4\n"; } //redefining 2  //#4
16      ~DerivedClass() {};
17  };
```

Review2: Overriding and Polymorphism -example

```
18  int main()
19  {
20      BaseClass bc;
21      DerivedClass dc;
22      BaseClass *b_ptr = &bc;
23      b_ptr->vFunction(); //calls 1
24      b_ptr->nvFunction();//calls 2
25      b_ptr = &dc;
26      b_ptr->vFunction();//calls 3
27      b_ptr->nvFunction();//calls 2
28
29  }   ≤ 2ms elapsed
```

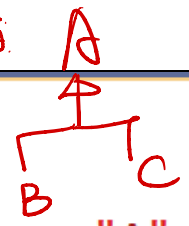
pointer points to the derived class

↳ because nv is not virtual function, so it still calls to the base class

Review3: Polymorphism and Casting of Pointers (in a short program!)

① A *ptr ← base class pointer can points to derived class B, C. or also points to class A.

② function (A* parameter-ptr) ↑ can accept the input of &B, &C, &A



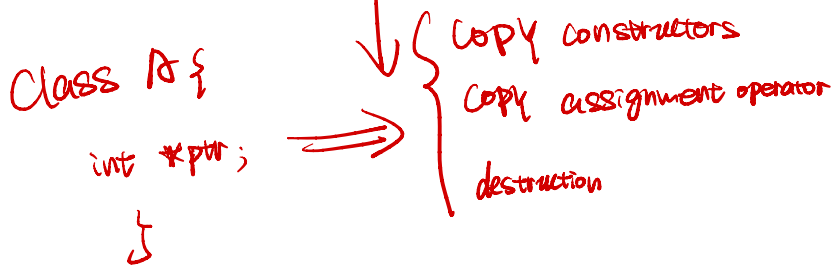

```
1   #include<iostream>
2   using namespace std;
3   class A { public: virtual void foo() { cout << "A"; } };
4   class B :public A { public: virtual void foo() { cout << "B"; } };
5   class C :public A { public: virtual void foo() { cout << "C"; } };
6   int main() {
7   A * ptr = nullptr;
8   ptr = new B; ptr->foo(); //prints B
9   ptr = new C; ptr->foo(); //prints C
10  C* ptr_c = dynamic_cast<C*>(ptr); // works! because current ptr is C
11  B* ptr_b = dynamic_cast<B*>(ptr); // not works! return nullptr
12  ptr = new A; ptr->foo(); //prints A
13
14  }
```

polymorphism

address

Review4: The Big rule (the 3 musketeers)

- Any **class** with **pointer attributes** must define the 3 musketeers (CC, CAO, D).
- Having **deep copy** is better than shallow copy for these classes!



What's the difference between shallow copy and deep copy

Signature of a Function

- Overloading a function name means the program has two or more different definitions of that function name.
- The definitions of an **overloaded function** must have either **different** number of parameters or different type of parameters (but they all have the same function name).

- A **function's signature** includes:
  - The name of the function
  - The type and the number of the parameters

Return type is not include of function's signature

- Remember that the return type is not included in the **function's signature** (especially, talking in the chapter overloading!).

- When overloading a function name, the overloaded functions must have different signatures.

```
int overloadFunction1(int a) { std::cout << "with int parameter"; };

double overloadFunction1(int a) { std::cout << "with return double"; };
```

double overloadFunction1(int a)
Search Online
cannot overload functions distinguished by return type alone

Operator Overloading

- Operators such as +, -, %, ==, and so forth actually are functions that are used with a different syntax. For example, we write i + 8 rather than + (i , 8) , but the + operators is a function which has two input arguments. The arguments are often called **operands**.

- Operator overloading allows us to use traditional operators, such as +,-,==, and so forth, with objects of class types that users define (user-defined types).

- Instead of writing, for example,
  - `MyClass mc1,mc2,mc3;`
  - `mc1.add(mc2.add(mc3))`

  we just simply write: `mc1+mc2+mc3;`
- Other benefit: the code is more readable: examples) `mc1 < mc2`

  `mc2 == mc1`

Operator Overloading

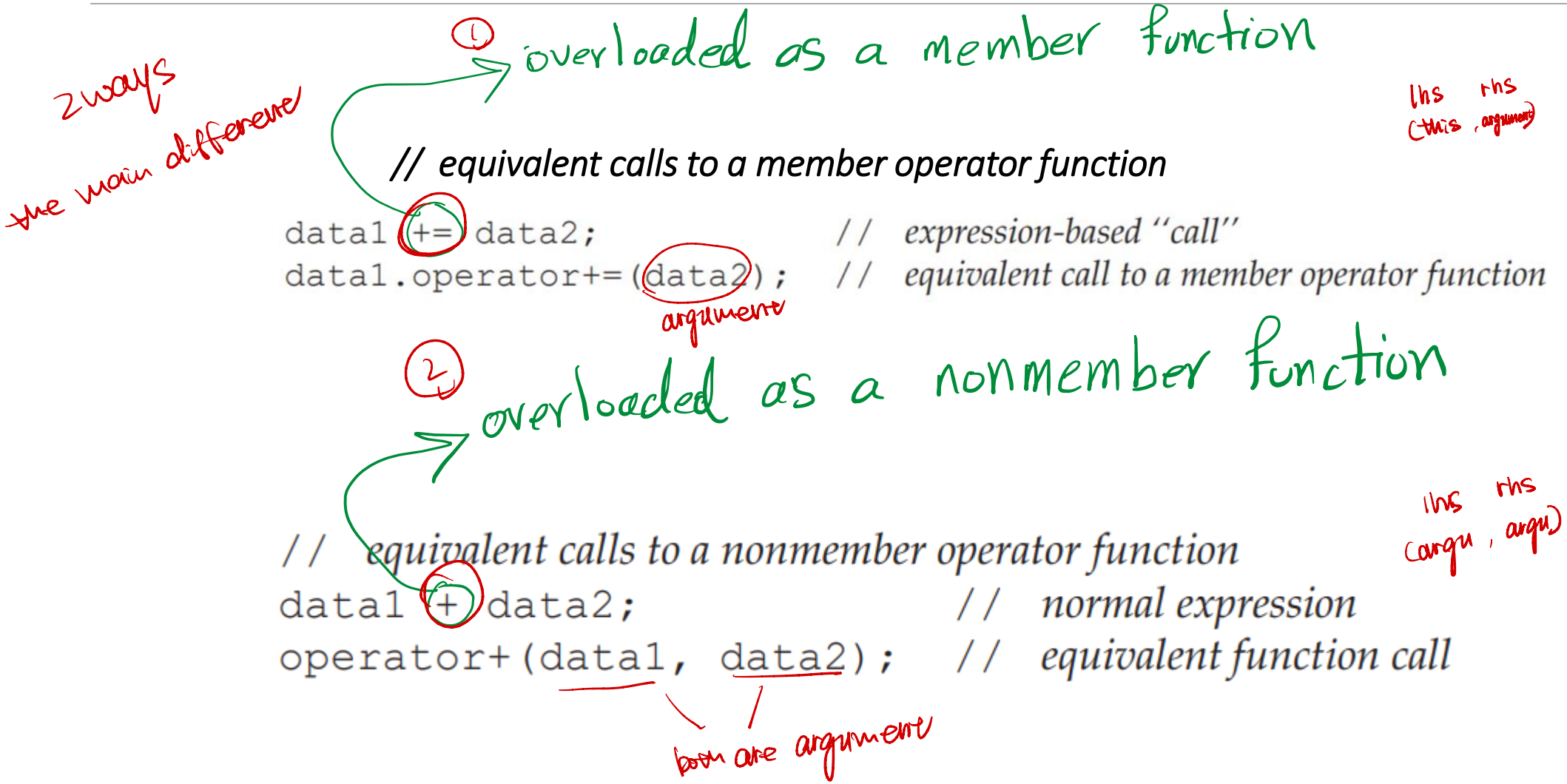
- The meaning of an operator with operands of primitive types cannot be changed.

```
// error: cannot redefine the built-in operator for ints
int operator+(int, int);
```

- We can overload only existing operators and **cannot** invent new operator symbols. For example, we **cannot** define the **operator**** to provide exponentiation.
- Some symbols including **(+,-,*,** and **&**)are both unary and binary operators, so the number of parameters specifies which n-ary is used.

Operators That Cannot Be Overloaded				
::	.*	.	?:	sizeof

Operator Overloading – calling an overloaded operator

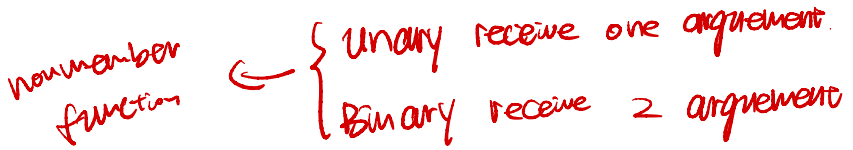


Operator overloading – part 1- through member functions

- Operators can be overloaded through class **member functions** or **non-member functions**.
- **Exceptions)** Following operators must be overloaded as **member functions** (not non-member operator overloading):

  `()` (**function-call** operator)
  `[]` (**subscript** operator)
  `->` (**arrow** operator)
  `=` (**assignment** operator)

  - In **member function** overloaded operators the **left-hand operand** is accessible in the body of definition through `this` pointer.
  - Member operator functions have one less parameters (n-1) than the number of operands for a n-ary operator.

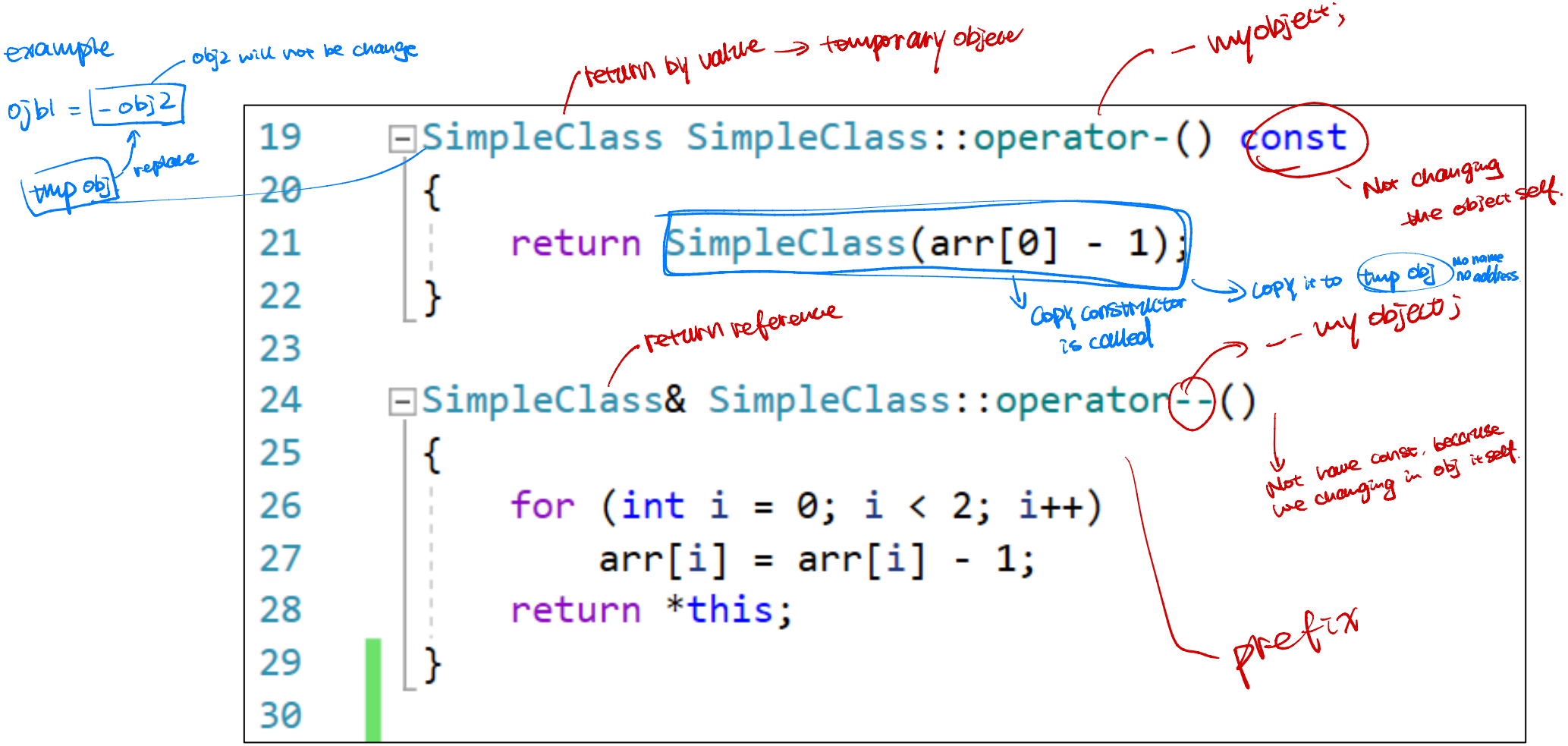


Operator overloading - member functions - example

```
1   #include <iostream>
2
3   class SimpleClass
4   {
5       int arr[2];
6   public:
7       SimpleClass(int i=0) { arr[0] = i; arr[1] = i+1 ;}
8       SimpleClass(const SimpleClass& rhs) { arr[0] = rhs.arr[0]; arr[1] = rhs.arr[1]; }
9
10      SimpleClass operator-() const;      //unary minus operator
11      SimpleClass &operator--();          //unary double minus operator
12      SimpleClass &operator+(const SimpleClass &rhs);     //binary plus operator
13      bool operator==(const SimpleClass &rhs) const; //double equal operator
14      SimpleClass& operator=(const SimpleClass& rhs); // copy assignment operator
15
16      ~SimpleClass() { }
17  };
18
```

Copy constructor
- assignment "="
- passing by value
- return by value

Operator overloading - member functions - example



Operator overloading - member functions - example

```
31  SimpleClass& SimpleClass::operator+(const SimpleClass& rhs)
32  {
33      for (int i = 0; i < 2; i++)
34          arr[i] = arr[i] + rhs.arr[i];
35      return *this;
36  }
37
38  bool SimpleClass::operator==(const SimpleClass& rhs)const
39  {
40      return (arr[0] == rhs.arr[0] && arr[1] == rhs.arr[1]);
41  }
42
43  SimpleClass& SimpleClass::operator=(const SimpleClass& rhs)
44  {
45      for (int i = 0; i < 2; i++)
46          arr[i] = rhs.arr[i];
47      return *this;
48  }
```

Operator overloading -member functions - example

```
49
50  int main()
51  {
52      SimpleClass sc1{ 1 };    //calls line 7
53      SimpleClass sc2( 2 );    //calls line 7
54      SimpleClass sc3={ 3 };   //calls line 7
55      sc1 = sc2;               //calls line 43
56      sc3 = -sc3;              //calls line 19 then 43 then 16
57      bool same = (sc2 == sc3);//calls line 38
58      sc3 = sc2 + sc1;         //calls line 31 then 43
59  //  sc3 = 1 + sc1;           // ERROR
60      sc1 = sc1 + 1;           //calls line 7 then 31 then 43 then 16
61  }
```

initialization.

operator equal

destroy the temp that created by line 19.

return by value → equal operator

implicit (temp obj)

not an object of simpleclass

When we are defining overloading operator using member function LHS should always be object of class.

Name	Value
sc1	{arr=0x00aff924 {3, 5} }
sc2	{arr=0x00aff914 {4, 6} }
sc3	{arr=0x00aff904 {4, 6} }

solution. using non-member function.

Operator overloading -member functions , prefix postfix versions

As we know:

`arr[i++]` means: first `arr[i]` and then `i++` //post-increment

`arr[++i]` means: first `i++` and then a`rr[i]` //pre-increment

For the **++ unary operator**:

```
++num;     // i.oprator++()
num++;     // i.operator++(int)
```

int is just to provide a hint to the compiler that ++ is post-increment

Best Practices

To be consistent with the built-in operators, the prefix operators should return a reference to the incremented or decremented object.

Best Practices

To be consistent with the built-in operators, the postfix operators should return the old (unincremented or undecremented) value. That value is returned as a value, not a reference.

Operator overloading -member functions – prefix postfix versions

```
//  postfix: increment/decrement the object but return the unchanged value
StrBlobPtr StrBlobPtr::operator++(int)
{
    //  no check needed here; the call to prefix increment will do the check
    StrBlobPtr ret = *this;    //  save the current value
    ++*this;       //  advance one element; prefix ++ checks the increment
    return ret;    //  return the saved state
}
```

hint to determin pre or post.

(i++)
return the original value

```
//  prefix: return a reference to the incremented/decremented object
StrBlobPtr& StrBlobPtr::operator++()
{
    //  if curr already points past the end of the container, can't increment it
    check(curr, "increment past end of StrBlobPtr");
    ++curr;          //  advance the current state
    return *this;
}
```

(++i)
return the changed value

Operator overloading – part 2-through non-member functions

always → is wrong!

↳ Do not have access to the private of the class

Overloading operators through **non-member functions** usually requires to declare them as **friend** functions of the classes. The `this` pointer **cannot** be used in the definitions of the **non-member** overloading functions because they are not member functions. Remember that `this` pointer is just available in member functions of classes.

If the class have getter and setter then it is not necessary to declare friend

```
1   #include <iostream>
2
3   class SimpleClass
4   {
5       friend bool operator!=(const SimpleClass& lhs, const SimpleClass& rhs);
6       friend SimpleClass operator+(const SimpleClass& lhs, const SimpleClass& rhs);
7       int arr[2];
8   public:
9       SimpleClass(int i = 0) { arr[0] = i; arr[1] = i + 1; }
10      ~SimpleClass() { }
11  };
12
```

can access all private variable

Not member function

!=
+
} binary operators

Operator overloading – part 2-through non-member functions

obj1 != obj2.

```
13  bool operator!=(const SimpleClass& lhs, const SimpleClass& rhs) {
14      return((lhs.arr[0] == rhs.arr[0]) && (lhs.arr[1] == lhs.arr[1]));
15  }
16
17  SimpleClass operator+(const SimpleClass& lhs, const SimpleClass& rhs) {
18      SimpleClass temp;
19      for (int i = 0; i < 2; i++)
20          temp.arr[i] = lhs.arr[i] + rhs.arr[i];
21      return temp;
22  }
23
24   int main()
25   {
26       SimpleClass sc1{ 1 };
27       SimpleClass sc2(2);
28       SimpleClass sc3 = { 3 };
29       bool bl = (sc3 != sc2);
30       sc3 = 1 + sc1;          // correct, not an ERROR anymore
31   }
```

return by value

temp obj

Overloading input/output operators << and >>

Operator overloading – operators << and >>

The overloading of operators << and >> are implemented as **non-member functions**. This is because in these overloadings the lhs (left-hand side) operand is not an object of user defined classes (instead, usually they are IO library such as streams classes or files, like istream, ostream, file,..., so forth)

<< the stream **insertion** operator (**output** operator)
>> the stream **extraction** operator (**input** operator)

cout << myobj;
Not obj of the class.

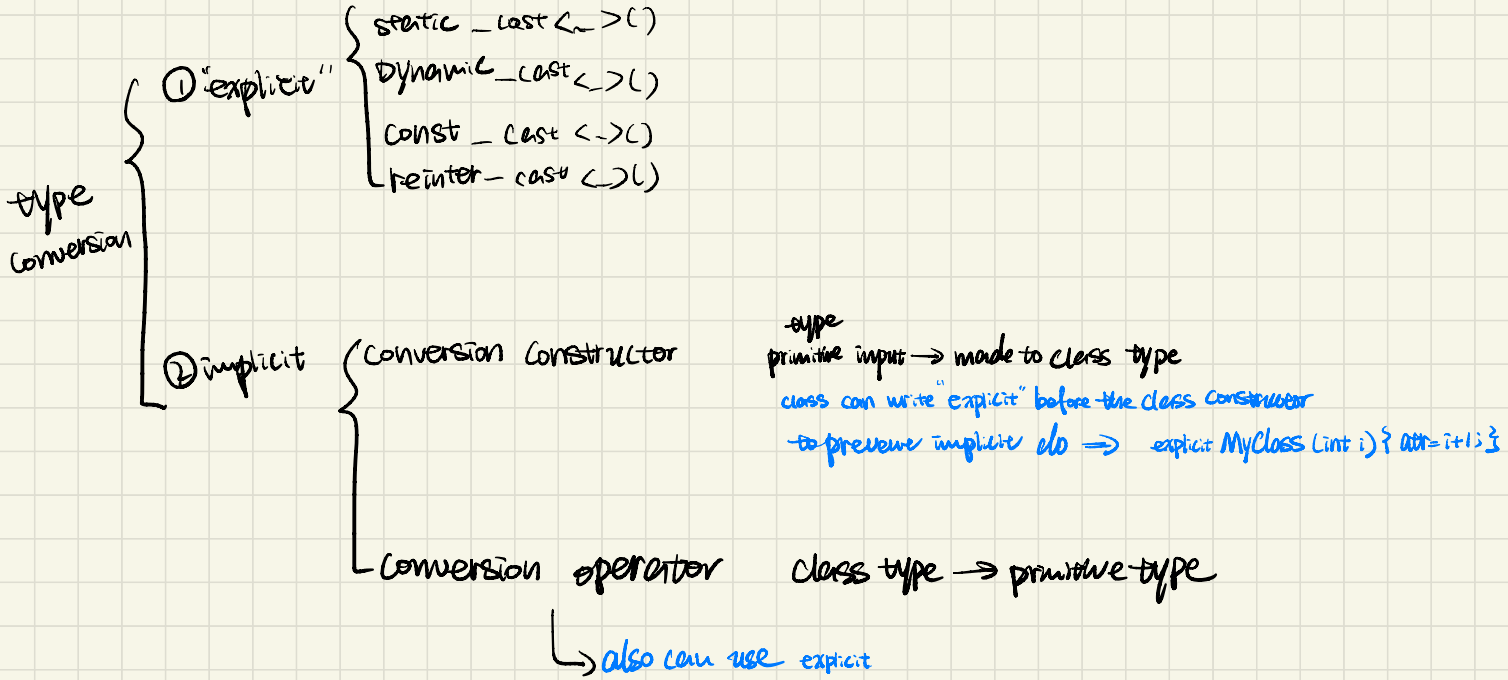
Remember that IO operations must be **nonmember functions**.

Operator overloading – operators << and >> - example

```
1   #include <iostream>
2   class SimpleClass
3   {
4       friend std::ostream& operator<<(std::ostream& os, const SimpleClass& rhs);
5       friend std::istream& operator>>(std::istream& in, SimpleClass& rhs);
6       int arr[2];
7   public:
8       SimpleClass(int i = 0) { arr[0] = i; arr[1] = i + 1; }
9       ~SimpleClass() { }
10
11  };
```

Operator overloading – operators << and >> - example

```
13  // overloaded <<
14  std::ostream& operator<< (std::ostream& os, const SimpleClass& rhs) {
15      os << rhs.arr[0] << " " << rhs.arr[1] << std::endl;
16      return os;
17  }
18
19  // overloaded  >>
20  std::istream& operator>> (std::istream& in, SimpleClass& rhs) {
21      int i;
22      in >> i;
23      rhs = SimpleClass{ i };
24      return in;
25  }
26
27  int main()
28  {
29      SimpleClass sc4{6};
30      std::cout << sc4;
31
32      SimpleClass sc5;
33      std::cin >> sc5;
34      std::cout << sc5;
35  }
```



- be explicit in you programming
- compiler choses the one that need less type casting

Conversion Operators (Type-cast operator)

- A conversion operator is a member function which implicitly converts **a value of the class type** to a value of **some other type**.
- Note that we can use the `this` pointer in the body of this function because conversion operator is a member function.

Syntax)

```
operator destination-type() const;
```

Example) If the following **conversion operator** is a member of the MyClass class, the compiler uses this conversion operator to convert a **MyClass object** to **int**. The definition of this conversion operator must **return** an **int** value.

```
operator int() const; //implicitly converts MyClass type to int
```

Conversion Operators can cause surprising results! So, similar to **Conversion Constructor**, we can use the `explicit` keyword to prevent the compiler to perform implicit conversions. Instead, do **the conversion explicitly!**

Example

```cpp
#include <iostream>
#include<string>
class SimpleClass
{
    int arr[2];

public:
    operator std::string() const; // conversion operator: converts SimpleClass to
      the string type

public:
    SimpleClass(int i = 0) { arr[0] = i; arr[1] = i + 1; }
};

SimpleClass::operator std::string() const
{
    std::string st = std::to_string(arr[0]) + std::to_string(arr[1]);
    return st; //return type must be string
}
```

```cpp
int main()
{
    SimpleClass sc4 { 6 };
    std::string s = sc4;
    std::cout << s;

}
```

programming implicit to call

```cpp
#include <iostream>
#include<string>
class SimpleClass
{
private:
    int arr[2];

public:
    explicit operator std::string() const; // conversion operator: converts
      SimpleClass to the string type

public:
    SimpleClass(int i = 0) { arr[0] = i; arr[1] = i + 1; }
};

SimpleClass::operator std::string() const
{
    std::string st = std::to_string(arr[0]) + std::to_string(arr[1]);
    return st; //return type must be string
}
```

prevent programm to call implicit.

```cpp
int main()
{
    SimpleClass sc4 { 6 };
    std::string s = sc4;
    std::cout << s;

}
```

Need change to

static_cast<std::string>sc4;

explicit type casting it automatic jump to casting