CH-230-A

Programming in C and C++

C/C++

Tutorial 12

Dr. Kinga Lipskoch

Fall 2020

Operator Overloading (1)

- Overloaded operators allow a different syntax for function/method calls
- Operator overloading allows the programmer to use language operators for user defined data type. For example

```
1  Car first, second, third;
2  ...
3  third = first + second;
what this could mean is left to the programmer
```

New operators cannot be introduced

Operator Overloading (2)

- It is possible to overload operators only for user defined data types
 - ▶ Thus it is not possible to alter the meaning of the + operator between ints, floats, and so on
 - C code should compile with C++ compilers without changing its functionality
- Overloading an operator should help the reader of the program and not the programmer
 - You can define an operator + between two instances of a salary class which subtracts them
 - Whether this makes sense or not is up to your design choices and skills

Sorting Students by Grade

With overloaded operators

```
1 Student *list;
2 ...
3 for (i = ...)
4   if (list[i] > list[j]) {
5     tmp = list[i];
6     list[i] = list[j];
7     list[j] = tmp;
8   }
9 ...
```

Without overloaded operators

```
1 Student *list;
2 ...
3 for (i = ...) {
4    if (list[i].getGrade() >
5        list[j].getGrade()) {
6         ...
7    }
8 }
```

Which one is "cleaner"?

Using/Calling Operators

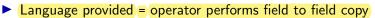
- ► From the previous example it should be clear that no magic things are happening, but just a compact (and clear) syntax for method calls can be used
- Whenever the compiler finds an operator involving user defined types, it verifies if it is possible to find a proper definition
- Operators are overloaded as there can be more than just one version: + between two instances of car, + between an instance of car and an instance of bike, and so on
- ► Types help in determining the version which should be called

Operators, Methods or friend Functions

- ▶ To carry out their task operators need to access class data
- ▶ Then, they have to be either methods or friend functions
 - ► There are some guidelines
 Student3.h Student3.cpp studentoperator.cpp
- ▶ It is possible to overload both unary and binary operators

Overloading the = Operator

► Overloading the = operator is fundamental if your class deals with pointers as properties





- ► If the class has pointers, different instances end up with sharing the same memory
 - ► Also, there could be memory leaks charbuffer.h charbuffer.cpp
- ► The operator = must be a method and must return a reference to the class



► This allows iterated assignments (a=b=c;)

Overloadable Operators

Unary Operators	Binary Operators
+ - & ! ~	+ - * / % ^ & <<
++	>> += -= *= /= %= ^=
(both prefix and postfix)	&= = >>= <<= == !=
[]->()	< > <= >= && ->*

new and delete can be overloaded as well

What Should Operators Return?

- ► A reference, it they modify the involved argument(s) (like =, +=, etc.)
- Return a reference to the modified object, usually by using this
- ▶ A new instance if they do not modify arguments, but rather use them to produce new information (like most binary operators: +, -, etc.)
- A bool if it is a boolean operator

Member or friend?

The following table proposed in the textbook can be taken as a general guideline

Operator	Use
Unary operators	Member
= () [] -> ->*	Must be member
+= -= /= *= ^= &= = %= >>= <<=	Member
Other binary operators	Friend or member

complex.h

complex.cpp

testcomplex2.cpp

Overloading << and >>

2

► For example for the Complex class:

```
\bigcirc
```

```
friend ostream& operator << (ostream& out,
const Complex &z)</pre>
```



```
friend istream& operator>>(istream& in,
Complex &z)
```

Polymorphism

- ► The last cornerstone of OOP has no correspondence in non-OOP languages
- ► It deals with writing code which can correctly operate even with data types unknown at compile time
- Just one additional keyword, virtual, which offers a wide range of applications

Starting from Upcasting

- Upcasting means that a derived class can be used wherever an ancestor class is expected
 - ► This because the interface of the parent is inherited
- ➤ A derived class can redefine a method, i.e., it can provide a new implementation
- ► randomnonvirtual.cpp
- randomvirtual.cpp

What Happens?

- ▶ In the first example the redefinition of the method was not perceived, while in the second "the expected" behavior was observed
- It should be evident that at compile time there is not enough information to bind the method call to the right method to execute
 - ► This is called late binding



The virtual Keyword

- When a method is declared as virtual, late binding is requested
- virtual should be specified just in the declaration of the base class
- ▶ Late binding is inherited from there on
 - The redefinition of virtual methods is called overriding
- When dealing with virtual methods, each object carries some sort of info to make its identification possible at runtime

virtual and Pointers

- Be aware: the late binding feature works only if you deal with pointers to instances, and not if you directly work with instances
 - ► And of course, also with C++ references
- ► When passing objects to methods/functions, using C++ references speeds up parameters passing and enables polymorphism

The Added Value of Polymorphism

- ▶ By mean of polymorphism it is possible to write general purpose code
 - ► When possible, general code always deals with base classes, and calls virtual methods
- ► Then, the code will work even with later introduced data types (new classes)

What Should be virtual?

- ► The decision is up to the designer
 - ► Some languages (like Java) make all methods virtual
- Virtual method calls introduce overhead
 - ► Run time binding
 - Making always every method virtual is poor design
- ▶ The choice is done at the base class level
 - If a method is not virtual in the base class, it cannot be made virtual in a derived class

How Does Polymorphism Work?

- In order to correctly manage it, it is useful to know how polymorphism is implemented
 - For every class with virtual methods, a table is created; the table holds the address of the virtual methods (VTABLE)
 - ► In addition, a pointer to this table is stored inside the class; this pointer is invisible (VPTR)
 - When a virtual method is called, the pointer is used to access the table, and then from the table the address of the function is read
 - As each class with virtual elements is shaped this way, the compiler can insert the code to resolve the calls without knowing the type
 - ► The intermediate indirection through VTABLE is the reason for the slower performance of virtual method calls

15 a->method();

Calls Through the Virtual Table

```
class Base {
                                             VTABLE
    public:
                                                           CODE
                                                          SEGMENT
       virtual void
3
         method()=0;
       // VPTR INSERTED
                                              method
  };
  class Derived
       : public Base {
8
                                                           Method
    public:
                                                         implementation
       void method() { };
       // VPTR INSERTED
                                 a. VPTR
14 Base *a= new Derived;
```