Object Oriented Methodology

Week – *5*, Lecture – *3*

Overloading

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

The three types of Constructors

We have discussed three types of constructors till now

- The default constructor – one with no arguments
- The parameterised constructor – one with one or more arguments
- The copy constructor – one which takes a single argument by reference, of the same type as the class

A class can have one or more Constructors

- This includes both, the defined constructors and those added by the compiler implicitly

Which constructor is invoked, depends on how the object is being created …

- … e.g., if no arguments are supplied, the default constructor must be invoked (provided that it is present)

One way to look at different constructors is that they *perform the same task differently* …

- … *for different situations*

This is an example of *method overloading*

Overloaded methods

A method is considered to be overloaded when it has two or more definitions in a class

Two or more function definitions are considered as overloaded versions of each other when …

- … the name of the method is the same, and …
- … the parameter list is different
- The return type does not matter – it could be the same or different for different definitions

Example – Overloaded Constructors

```
Toy2::Toy2(string s, int i)
{
        id = counter++;
        name = s;
        price = i;
        cout<<"Creating a new "<<name<<" with id "<<id<<endl;
}

Toy2::Toy2(Toy2& copy)
{
        id = copy.id;
        name = copy.name;
        price = copy.price;
        cout<<"Creating a copy of "<<name<<" with id "<<id<<endl;
}
```

Two constructors in the same class – what differs is the argument list

Overloaded methods

A method is considered to be overloaded when it has two or more definitions in a class

Two or more function definitions are considered as overloaded versions of each other when …

- … the name of the method is the same, and …
- … the parameter list is different
- The return type does not matter – it could be the same or different for different definitions

Overloading is useful when a behaviour of a class differs with respect to different scenarios

- This is different from overriding, where the changes in behaviour are part of different classes

In general, whenever you hear the term “overloading”, it means “method overloading”

- Another type of overloading – which is allowed in fewer languages – is called “operator overloading”
- While method overloading is extremely common, operator overloading is often not that appreciated

Example – Overloaded Methods

```
class ToyDemonstrator
{
        public:
                void show_demo(Toy2&);
                void show_demo(BatteryOperatedToy2&, string, int);
};
```

Assume that we need a demonstrator for showing a demo of our toys

We represent this demonstrator by a class, which has an overloaded method – `show_demo()`

Although the first version is applicable to all the toys, the second version provides the demonstrator with some batteries to show a battery-operated toy better !!

Example – Overloaded Methods

```
void ToyDemonstrator::show_demo(Toy2& toy)
{
        cout<<"Let's play with "<<toy.get_name()<<endl;
        toy.start_playing();
        cout<<"I guess it's enough :P"<<endl;
        toy.stop_playing();
}

void ToyDemonstrator::show_demo(BatteryOperatedToy2& toy,
                string battery_type, int number_of_batteries)
{
        cout<<"I need to put some batteries in it first !!"<<endl;
        toy.put_batteries(battery_type, number_of_batteries);
        show_demo(toy);
        cout<<"Let me take the batteries out..."<<endl;
        toy.take_out_all_batteries();
}
```

Something like this …

Example – Overloaded Methods

```
Toy2 t("Blocks", 399);
BatteryOperatedToy2 bt("Kids' Piano", 399, "AA", 3);
ToyDemonstrator td;
cout<<"Showing demo for "<<t.get_name()<<endl;
td.show_demo(t);
cout<<endl;
cout<<"Showing demo for "<<bt.get_name()<<endl;
td.show_demo(bt, "AA", 3);
```

Example – Overloaded Methods

```
Toy2 t("Blocks", 399);
BatteryOperatedToy2 bt("Kids' Piano", 399, "AA", 3);
ToyDemonstrator td;
cout<<"Showing demo for "<<t.get_name()<<endl;
td.show_demo(t);
cout<<endl;
cout<<"Showing demo for "<<bt.get_name()<<endl;
td.show_demo(bt, "AA", 3);
```

The version to invoked is decided by the compiler by inspecting the arguments of the method call

The compiler tries its best to find the right version, which includes performing any implicit data type conversions

However, if it fails to find any matching definition based on the parameters, you will get a compilation error !!

Overloaded Operators

Assume that you have a class to represent a complex number

- Typically, you'll need two fields in this class – one for the real part and the other for the imaginary part

Now, if you have two objects of this class – representing two complex numbers …

- … how can you add them together? On the same lines, how can you subtract one from another?

One way to do so will be by defining methods – member functions or global functions

- In this case, typical expressions that you will see will include
- `Complex c3 = c1.add(c2);`
- `Complex c3 = add(c1, c2);`

Wouldn't it be better if you could simply write something like `Complex c3 = c1 + c2;` ?

- In C++, it is possible to do so by *overloading the + operator* "for the `Complex` class"

Overloading the operator means providing or changing the definition for an expression …

- … containing that *operator* "for the said type of operands"

Example – Overloaded Operators

```
Toy3 t1("Blocks", 399);
Toy3 t2("Rings", 499);
Toy3 t3("Guitar", 349);
cout<<"We have these toys with us:"<<endl;
cout<<"1. "<<t1.get_name()<<" (Rs "<<t1.get_price()<<")"<<endl;
cout<<"2. "<<t2.get_name()<<" (Rs "<<t2.get_price()<<")"<<endl;
cout<<"2. "<<t3.get_name()<<" (Rs "<<t3.get_price()<<")"<<endl;
Toy3 t4 = t1>t2>t3;
cout<<"The costliest toy is "<<t4.get_name()<<endl;
t4 = t1<t2<t3;
cout<<"The cheapest toy is "<<t4.get_name()<<endl;
```

Example – Overloaded Operators

```
Toy3 t1("Blocks", 399);
Toy3 t2("Rings", 499);
Toy3 t3("Guitar", 349);
cout<<"We have these toys with us:"<<endl;
cout<<"1. "<<t1.get_name()<<" (Rs "<<t1.get_price()<<")"<<endl;
cout<<"2. "<<t2.get_name()<<" (Rs "<<t2.get_price()<<")"<<endl;
cout<<"2. "<<t3.get_name()<<" (Rs "<<t3.get_price()<<")"<<endl;
Toy3 t4 = t1>t2>t3;
cout<<"The costliest toy is "<<t4.get_name()<<endl;
t4 = t1<t2<t3;
cout<<"The cheapest toy is "<<t4.get_name()<<endl;
```

Here, you can see two rather weird expressions – one that applies the > operator on toy objects, and the other where you see the < operator doing something similar

This is because we have overloaded these operators for the `Toy3` class

We define > operator as the “costly operator” (it returns the costlier of the two toys) and < as the “cheap operator” (it returns the cheaper of the two toys)

Example – Overloaded Operators

```
// v3 of the class Toy
class Toy3
{
        private:
                string name;
                int price;
                int id;

                static int counter;

        public:
                Toy3(string, int);
                Toy3(Toy3&);
                virtual void start_playing();
                virtual void stop_playing();
                string get_name();
                void set_price(int);
                int get_price();
                int get_id();

                static void reset_counter();

                Toy3& operator > (Toy3&);
                friend Toy3& operator < (Toy3&, Toy3&);
};
```

There are two ways in which an operator's overloaded definition can be provided

Example – Overloaded Operators

```
// v3 of the class Toy
class Toy3
{
        private:
                string name;
                int price;
                int id;

                static int counter;

        public:
                Toy3(string, int);
                Toy3(Toy3&);
                virtual void start_playing();
                virtual void stop_playing();
                string get_name();
                void set_price(int);
                int get_price();
                int get_id();

                static void reset_counter();

                Toy3& operator > (Toy3&);
                friend Toy3& operator < (Toy3&, Toy3&);
};
```

There are two ways in which an operator's overloaded definition can be provided

You can provide it as a member function of the class

Example – Overloaded Operators

```
// v3 of the class Toy
class Toy3
{
        private:
                string name;
                int price;
                int id;

                static int counter;

        public:
                Toy3(string, int);
                Toy3(Toy3&);
                virtual void start_playing();
                virtual void stop_playing();
                string get_name();
                void set_price(int);
                int get_price();
                int get_id();

                static void reset_counter();

                Toy3& operator > (Toy3&);
                friend Toy3& operator < (Toy3&, Toy3&);
};
```

There are two ways in which an operator's overloaded definition can be provided

Or, as a `friend` function to this class

Member vs Friend Function forms

Operator overloading is a complex topic – we will not discuss each aspect of it …

- … mostly because it is not as often used because of the associated problems
- Still, we will discuss the two different forms of operator definitions

Definitions for the operators are provided in the form of functions (for a binary operator)

- The name of the definition function is the keyword `operator`, followed by the operator
- If the method is a member function of the class, the calling object is provided implicitly to the definition
- e.g. when you write the expression `(c1 + c2)`, it is equivalent to `c1.add(c2)`
- If the method is a friend function of the class, the function is passed two arguments
- e.g. when you write the expression `(c1 + c2)`, it is equivalent to `add(c1, c2)`
- In most cases, the return type is the same as the class for which the operator is being overloaded

Usually, to avoid creation of a new object, we pass and return values by reference

Example – Overloading by member function

```
Toy3& Toy3::operator >(Toy3& other_toy)
{
        return other_toy.get_price() > get_price() ?
                other_toy : *this;
}
```

Observe here that we are only taking one argument in the function – the second argument to be precise in the expression `t1>t2`, whereas the first object invokes the method (and hence, is implicitly available)

For example, in the above case, `get_price()` implies `t1.get_price()` and `other_toy.get_price()` implies `t2.get_price()`

Example – Overloading by friend function

```
Toy3& operator <(Toy3& first_toy, Toy3& second_toy)
{
        return first_toy.get_price() <= second_toy.get_price() ?
                first_toy : second_toy;
}
```

When the definition is provided as a friend function, both objects must be caught as two distinct parameters

For the expression `t1<t2`, the `first_toy` represents `t1` and `second_toy` represents `t2`

A word on using Operator Overloading

Keep it in mind that everything you can do with overloaded operators can also be done otherwise

- For example, with an appropriately named member function or friend function

In fact, *in most cases* it is better to define methods than to overload operators

- For example compare the two expressions:
  `t3 = t1<t2;` and
  `t3 = cheaper_of_the_two(t1, t2);`
- The latter is probably better for readability of the code

Unless the semantics of the operator is really obvious - *I strongly suggest against overloading operators*

- It may lead to code that is not readable and hard to maintain !

Additional Reading

If you wish to know more about Operator Overloading, check out the following link:

- https://www.geeksforgeeks.org/operator-overloading-c/