Object-Oriented Programming

Lecture 3

Agata Półrola
<agata.polrola@wmii.uni.lodz.pl>

Static members

Static fields:

- common memory variables shared by all objects of a class
- available for all methods of a class
- declared inside a class, but initialised outside with scope operator (::) regardless of the access specifier used

Static methods:

- working for the class, not for a particular object
- called by the class with scope operator or by its objects with dot operator
- cannot access the ordinary object members (only static)

Static members

```
class List
{
  int value;
  List *next;
  static List *head; //static field, one for the whole class
  void printList(); //internal method

 public:
  List(int _value = 0);
  List(const List &node);
  ~List();
  static void print() //static method
  {
    head->printList(); //ok, head is the static field
    next->printList(); //compile-time error, next is non-static
  }
};

List *List::head = NULL; //initialisation of the static field

List::print(); //static method call
List::head = new List; //compile-time error, private field
```

this

- the pointer to the object which a method is called for
- passed to each method as the first implicit parameter

```
List::List(int _value = 0)
{
   this->value = _value;
   this->next = head;
   head = this;
   cout << "Constructor: ";
   this->printList();
}

List::List(int _value = 0)
{
   value = _value;
   next = head;
   head = this;
   cout << "Constructor: ";
   printList();
}
```

Friend functions

- external functions declared by a class as friends
- full or read-only access to the private members of the class objects given as parameters or created locally

```
class List
{
  int value;
  List *next;
  static List *head;
  void printList();

 public:
  … //methods
  friend int look(const List &node); //friendship declaration
};

int look(const List &node) //friend function definition
{
  node.next = NULL; //compile-time error, read-only access
  return node.value; //access to the private member of object
}
```

Overloading of operators

- redefinition of the built-in operator functions for a class objects as operands
- name of the function consists of the keyword operator followed by the operator name or symbol, i.e., operator+, operator=, etc.
- overloaded operators can be methods or friend functions (some of them must be methods)
- there is no possibility to create any new operator, nor to change the priority of the evaluation of the operators
- It is a «syntactic sugar» – do not overload any operator if this is not necessary for the improvement of the code readability

Unary operators

- without arguments if overloaded as a class method (member function)
- with one argument if overloaded as a friend function
- suggestion – it should be a constant method

```
class Vector
{
  double x, y;
 public:
  Vector(double _x = 0, double _y = 0) { x = _x; y = _y; }
  Vector operator-() const
  {
    return Vector(-x, -y);
  }
};

Vector v1(2, -3);
Vector v2(-v1); //unary operator call
v2 = -v1; //also here
v2 = v1.operator-(); //alternative call
```

Binary operators

- with only one argument if overloaded as a class method (member function)
- with two arguments if overloaded as a friend function
- suggestion – it should be a friend function with constant arguments to enable the automatic conversions of the both operands from simpler types

```
class Vector
{
  double x, y;
 public:
  Vector(double _x = 0, double _y = 0) { x = _x; y = _y; }
  friend Vector operator+(const Vector &v1, const Vector &v2);
};

Vector operator+(const Vector &v1, const Vector &v2)
{
  return Vector(v1.x + v2.x, v1.y + v2.y);
}
Vector x(2, -3), y(-1, 4), z;
z = x + y; z = x + 2; z = 2 + x; //binary operator calls
```