Object-Oriented Programming

Lecture 2

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

Object construction

- the constructor - the special method for proper initialisation of the fields
- called automatically on every object creation
- the name of the constructor is the same as the name of the type (class)
- many constructors with arbitrary number of arguments, no return value!

```
struct Obj
{
  int a, b; //fields

  Obj(int _a = 0, int _b = 0) //constructor
  {
    a = _a;
    b = _b;
  }
};

Obj x, y(1), z(1, 2), v = 3; //the implicit constructor calls
Obj t[5], *s = new Obj, *p = new Obj[3]; //also here
```

Default constructor

Default constructor:

- which may be called without arguments (any arguments with default values)
- automatically created if (and only if) there are no explicit constructors, but without any initialisation of the fields (risky when there are any dynamically allocated fields in the class)

```
struct Obj
{
  int a, b; //fields

  Obj() //default constructor
  {
    a = 0;
    b = 0;
  }
};

Obj x, t[5]; //default constructor calls
Obj *s = new Obj, *p = new Obj[3]; //also here
```

Default constructor

Default constructor:

- if any constructors are defined, the default constructor is not created automatically
- important – always define the default constructor (to simplify future inhertitance from the class)

```
struct Obj
{
  int a, b; //fields

  Obj(int _a, int _b = 0) //1- or 2-parameter constructor
  {
    a = 0;
    b = 0;
  }
};

Obj y(1), z(1, 2); //proper creation
Obj x; //compile-time error without the default constructor
```

Copy constructor

Copy constructor:

- takes as the only argument a reference to an existing object of the same type
- automatically created if (and only if) there is no explicit copy constructor, but copying the fields by simple bit-copy (risky when there are any pointer fields)
- important – always define the copy constructor for a class with pointer fields

```
struct Obj
{
  int a, b; //fields
  … //default constructor here
  Obj(const Obj &o) //copy constructor
  {
    a = o.a;
    b = o.b;
  }
};
Obj x;
Obj y(x), z = x; //the implicit copy constructor calls
```

Converting constructor

Converting constructor:

- takes as the only argument a value of another type (converts this type to the type of a class)

```
struct Obj
{
  int a, b; //fields
  … //default constructor here
  Obj(int v) //converting constructor (int → Obj)
  {
    a = b = v;
  }
  bool equals(const Obj &o) //method with object argument
  {
    return (a == o.a) && (b == o.b);
  }
};

Obj y(1), z = 2; //the implicit converting constructor calls
if (y.equals(5)) ... //also here
```

Object destruction

- the destructor - the special method for proper cleanup
- called automatically when the object goes out of scope and on explicit deletion
- the name of the destructor is the same as the name of the type (class) but with leading tilde (~)
- only one destructor without arguments, no return value!

```
struct Obj
{
  int a, b; //fields
  … //default constructor here
  ~Obj() //destructor with any cleaning operations
  { ... }
};

{
  Obj x, *p = new Obj; //constructor calls
  delete p; //the explicit destructor call (object *p)
} //the implicit destructor call (object x)
```

Object destruction

The destructor:

- automatically created if (and only if) there is no explicit destructor, but without any cleanup operations (risky when there are any dynamically allocated fields in the class)
- important – always define the destructor for a class with the constructor dynamically allocating its fields
- destructors for objects are called by the compiler in the reverse order of the object creation order

```
{
  Obj x, *p = new Obj, z;
  {
    Obj y;
  } //y destructed
  delete p; //*p destructed
} //z and then x destructed
```

Encapsulation : data protection

- code safety and independence
- better team support with the code separation
- without «giving a monkey a razor» (black-box solution)
- different levels of data access in a class (through access specifiers):
  - **public** – available for everyone
  - **private** – available only for the member functions of that class (methods)
  - **protected** – like private, but additionally available for the inherited classes
  - **friend** – available for implicitly pointed non-member functions or other classes
- constant methods guaranteeing no changes of an object they are called for

Data protection : struct

Within the struct statement all members are public by default:

```
struct List
{
 private: //hidden members
  int value;
  List *next;
  void printList(); //internal method

 public: //public interface
  List(int _value = 0);
  List(const List &node);
  ~List();
  int getValue() const { return value; } //getter method
  void setValue(int _value) { value = _value; } //setter method
  … //other methods
};

List x(1);
x.setValue(2); //public method call
x.printList(); //compile-time error
```

Data protection : class

Within the class statement all members are private by default:

```
class List
{
 //hidden members, private keyword optional
  int value;
  List *next;
  void printList(); //internal method

 public: //public interface
  List(int _value = 0);
  List(const List &node);
  ~List();
  int getValue() const { return value; } //getter method
  void setValue(int _value) { value = _value; } //setter method
  … //other methods
};

List x(1);
x.setValue(2); //public method call
x.printList(); //compile-time error
```