Programming, numerics and optimization

Lecture A-4: Object-oriented programming

Łukasz Jankowski
ljank@ippt.pan.pl

Institute of Fundamental Technological Research
Room 4.32, Phone +22.8261281 ext. 428

April 6, 2021[1]

[1] Current version is available at http://info.ippt.pan.pl/~ljank.

Disclaimer

I do realize that

- For those of you that know OOP, this lecture will contain nothing new.
- For those of you that do not know OOP, this lecture will be much too condensed.
- OOP would require a dedicated course.

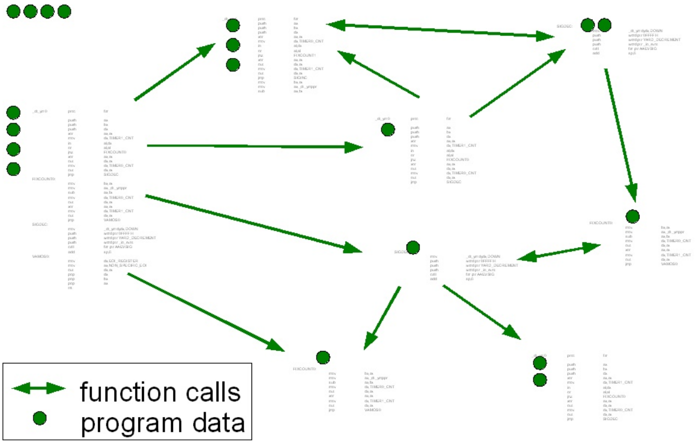
OOP is required for HW6 and HW7. If OOP/C++ is new for you, these HWs will probably demand a lot of additional work.

Outline

Outline

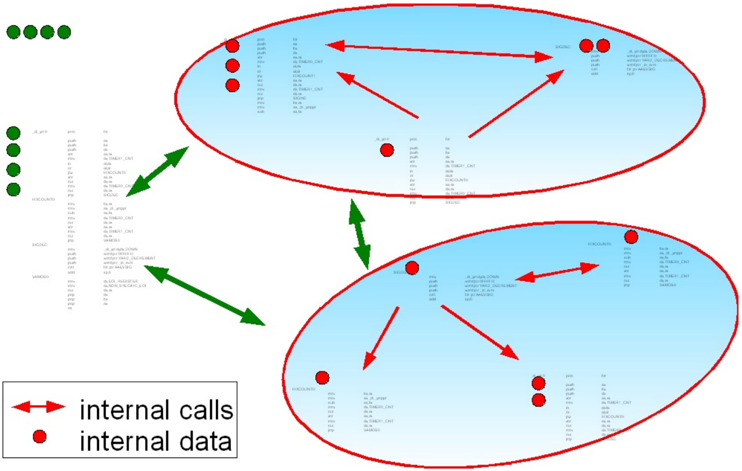
Object-oriented programming (OOP)

Structure of a typical non-object code



Object-oriented programming (OOP)

Objects (may) bring order



Object-oriented programming (OOP)

Object paradigm is an answer to the following deficiencies of procedural programming:

- Data separated from processing information.
- The code (and the real problem it solves) scattered into separate, unrelated data chunks and functions.

Objects and classes

Types in C++

1. Fundamental: **bool**, **char**, **int**, **float**, **double**, **void**, ...
2. Compound (built-in or user-defined): pointers, arrays, structures, classes, ...

Classes are user-defined types, which group together:

- Data (internal content of an object), e.g. matrix elements
- Functions (often called methods)
  - Defining operations on the data, e.g. matrix :: det()
  - Responsible for the behavior of an object, e.g. point :: show()

Objects and classes

- Classes corresponds to types.
- Objects (of a given class) correspond to variables (of a given type). An object is called to be an *instance* of his class.

The idea is to have e.g. a type (class) vector to use it like **int**:

```
int a = 5, b = 2, c;
c = 2*a*b;
cout <<c;
```

```
vector a(5,4), b(12,1), c;
c = a-b;
cout <<c.norm();
```

Data:	vector elements
Functions:	multiplication, addition, subtraction, showing etc.

Basic concepts

OOP is described in a variety of ways:

- from ideologically negative (e.g. hard-core proponents of functional programming),
- to pragmatic,
- to ideologically positive ("*an OO program is an ensemble of communicating agents...*").

There is no a generally-agreed set of features, an OO language should support. However, the typical features include:

- Encapsulation
- Inheritance
- Polymorphism
- Dynamic dispatch

Basic concepts
Encapsulation

Encapsulation usually refers to

- grouping together data with functions that operate on them

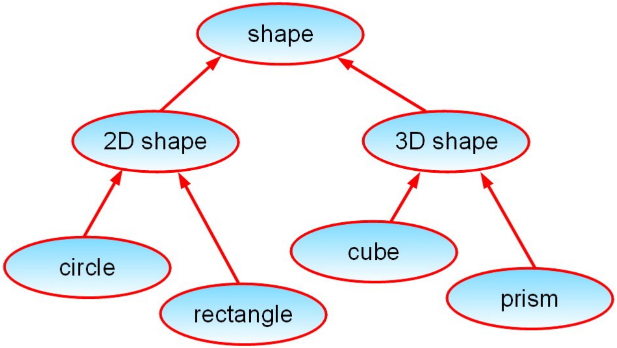
  E.g. an object matrix might bundle matrix entries together with methods like det().

- protecting a part of the internal data of an object from external access and modifications

  E.g. matrix entries might be private data (accessible only indirectly).

Basic concepts
Inheritance

Classes in an OO language can form a class hierarchy.



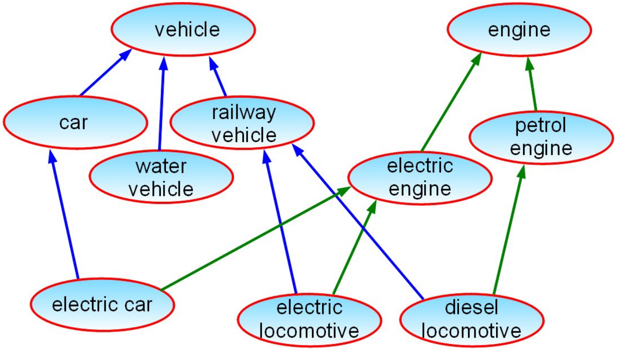
Objects of the derived class

- inherit their properties (data & methods) from the base class,
- specialize the base class.

Basic concepts

Multiple inheritance

Some of OO languages (like C++) allow multiple inheritance.

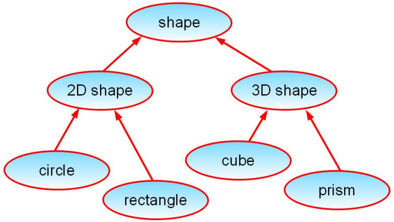


With inheritance, parent classes are often abstract classes[2].

[2] An abstract class is non-instantiable (each actual vehicle must be either an electric car or an electric locomotive or a diesel locomotive etc.: there is no such a thing as a “pure vehicle”).

Basic concepts

Polymorphism, dynamic dispatch



The (possibly abstract) class shape might have a (virtual) member function shape.draw(). This function is inherited by the derived classes, which have to implement it according to their characteristics.

Assume we have an array of pointers to different shapes.

- Thanks to polymorphism, they can be all treated as being of the (possibly abstract) type shape. The array can be thus defined as shape ∗ptr[shape_no];.
- Thanks to dynamic dispatch, given an object of the class shape, it can be ordered to be drawn by calling the method shape.draw(). Depending on its actual type (circle? cube?), the method of the proper derived class will be executed (decided upon not during compilation, but in runtime when the type of the actual object is known).

Advantages and disadvantages

Advantages of OOP

- better representation of real-world entities by grouping together data with operations (e.g. a matrix is no longer just a 2D array but a *matrix*), inheritance, polymorphism, etc.
- hidden implementation details and protection of internal data
- operator overloading
- brings order into the system model and the code (easier production and maintenance)
- potential for reusability

Disadvantages of OOP

- slower code
- code duplications
- additional abstraction layer

Outline

Objects and classes
Class definition

An object consists of

- Interface (e.g. for a matrix class: $+$, $-$, $*$, transpose(), det(), invert (), eigenvalues () etc.)
- Hidden implementation (internal data structures, internal operations etc.).

```
class className {
   private:
      // private data members
      // private functions
   public:    // class interface
      // public data members (not recommended)
      // public functions
};
```

Objects and classes
Class definition — example

Class definition

```
class vector {
    private:
        int x,y;
    public:
        void setXY(int a, int b) {x=a; y=b;}
        double getLength(void) const;
}; // notice the semicolon!

double vector::getLength(void) const {
    return sqrt(x*x+y*y);
}
```

The **const** modifier ensures here that the member function does not modify the object (its data members).

Objects and classes
Class definition — example

Class definition

```
int main(void) {
    vector a;           // object definition

    a.setXY(1,1);
    cout <<a.getLength() <<endl;
    // cout <<"a.x = " <<a.x <<endl; ERROR!

    system("pause");
// for Dev C++ users only
    return 0;
}
```

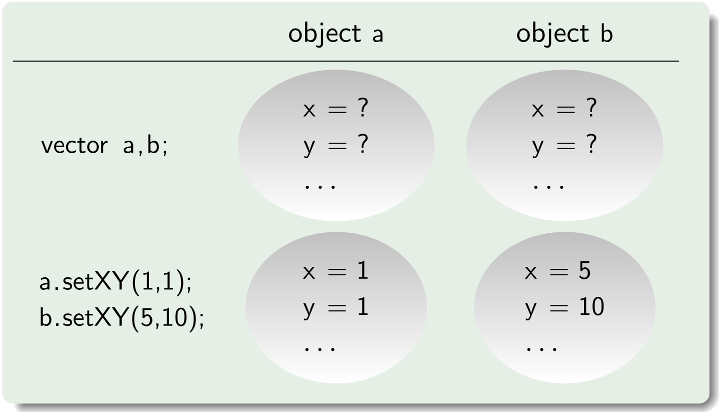
Objects and classes
Pointers to objects

As variables of any other type, objects can be created and destroyed dynamically.

```
int main() {
   vector *pa;          // pointer to a variable
                        // of the type vector

   pa = new vector;     // create a vector
   pa->setXY(1,1);
   cout <<pa->getLength() <<endl;
   delete pa;           // delete the object *pa
   pa = NULL;

   return 0;
}
```

Objects and classes
Internal data

Internal data are specific to each object, that is each object has its own set of the internal data.



Outline

Constructor

Constructor is a special function, which is

- automatically called when a new object is *created*
- used to initialize internal variables, assign dynamic memory for internal storage etc.
- cannot be called at a later time

Constructor

- must have the same name as the class
- has no return type (not even **void**)
- can be overloaded

If no constructor is provided, a default constructor is automatically created (no arguments, no initialization).

Constructor — example

```
class vector {
    private:
        int x,y;
    public:
        vector(int a, int b);            // constructor
        void setXY(int a, int b) {x=a; y=b;}
        double getLength(void) const;
};

vector::vector(int a, int b) {
    x = a;
    y = b;
}
```

Constructor — example

`vector a;`	ERROR: if there is a user-defined constructor, then no default constructor is created. But here the user-defined constructor for the class vector requires arguments.
`vector a(1,2);`	right! Call the user-defined constructor.
`vector *pa;`	a pointer to an object of the type vector
`pa = new vector;`	ERROR: there is no no-argument constructor.
`pa = new vector(1,2);`	right!

```
...
delete pa;
```

Overloaded constructors

A class can have many constructors, provided they are distinguishable by the number or types of their arguments.

```
class vector {
  private:
    int x,y;
  public:
    vector(void);           // default constructor
    vector(int a, int b); // constructor
    void setXY(int a, int b) {x=a; y=b;}
    double getLength(void) const;
};

vector::vector(int a, int b) {x = a; y = b;}
vector::vector(void) {x = 0; y = 0;}
```

Constructor with default arguments

As any other function, constructor can take default arguments.

```
class vector {
    private:
        int x,y;
    public:
        vector(int a=0, int b=0);        // constructor
        void setXY(int a, int b) {x=a; y=b;}
        double getLength(void) const;
};

vector::vector(int a, int b) {
    x = a;
    y = b;
}
```

Destructor

Destructor is a special function, which is

- automatically called when an object is *destroyed* (at end of its scope or when deleted with **delete**)
- used to clean-up (dynamic memory used by the object, closing open files etc.)

Destructor

- must have the same name as the class, preceded with a tilde $\sim$
- has **void** arguments and no return type (not even **void**)
- cannot be overloaded

Destructor — example

```
class vector {
   private:
      int *x, *y;
   public:
      vector(int a=0, int b=0);  // constructor
      ~vector(void);             // destructor
      // ...
};
vector::vector(int a, int b) {
   x = new int(a);
   y = new int(b);
}
vector::~vector(void) {
   delete x;
   delete y;
}
```

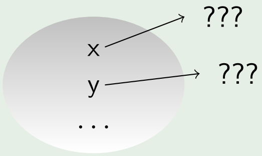
Destructor — example

If an object uses dynamic memory

- a user-defined constructor should be provided for allocation
- destructor should deallocate it to avoid memory leakage

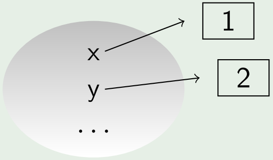
if there was only the default constructor

vector a;



with the user-defined constructor

vector a(1,2);

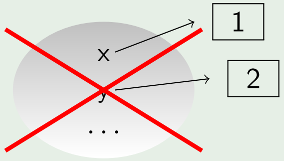


Destructor — example

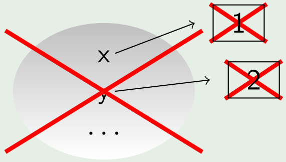
If an object uses dynamic memory

- a user-defined constructor should be provided for allocation
- destructor should deallocate it to avoid memory leakage

without a user-defined destructor



with a user-defined destructor



Copy constructor

Copy constructor is a special constructor, which is called whenever a new object is created (copied) from an existing object.

If no user-defined copy constructor is provided, a default copy constructor is provided automatically (*shallow*: blindly copies internal variables). When a class uses dynamic memory, it should always have a user-defined copy constructor (*deep*: intelligently copying data pointed to, not the pointers only).

Copy constructor is defined as a usual constructor, but with a single argument: a **const** object of the same type passed by reference. The current object (the one being created) is created (copied) from the passed object.

Copy constructor — example

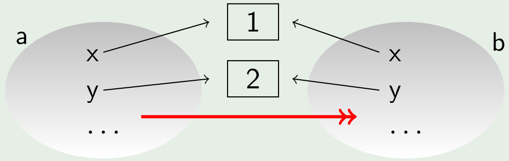
```
class vector {
  private:
    int *x,*y;
  public:
    vector(const vector &v); // copy constructor
    vector(int a, int b);    // constructor
  //...
};

vector::vector(const vector &v) {
  x = new int(*v.x);   // access to private members
  y = new int(*(v.y));// inside the class only
}                      // (even from another object)
```

An argument to a copy constructor has to be passed by reference (passing by value would require an already defined copy constructor). However, it is passed with a modifier **const** (**const** vector &v) to protect it against modifications.

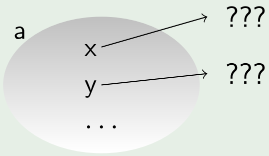
Copy constructor — example

A default (shallow) copy constructor with a class using dynamic memory copies blindly (shallowly) pointer by pointer instead of allocating additional memory.

```
vector a(1,2);
vector b = a;
```



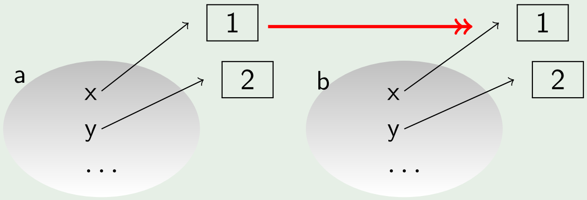
At the end of the scope the object b is destroyed first and the dynamic memory is deallocated. When the object a is then destroyed, it tries to deallocate the memory already deallocated!



Copy constructor — example

A class using dynamic memory should always have a deep, user-defined copy constructor to intelligently copy the data and allocate additional memory if necessary.

vector a (1,2);
vector b = a;



At the end of the scope each objects frees its own allocated memory.

Copy constructor

Copy constructor (either user-defined or default) is called in three situations: when an object is

1. created from another object of the same type

```
vector a(1,1), b = a, c(a);
```

2. passed by value as an argument to a function

```
void doSomething(vector v) {...};
```

3. returned by value from a function

```
vector f(...) {
    vector a(1,1);
    //...
    return a;
}
```

Outline

Overloaded operators

In mathematics, two vectors or matrices can be simply added, just as two numbers. The same functionality, for example

```
vector a(1,1), b(2,2), c = a+b;
int scalarProduct = a*b;
```

is possible in C++ by means of overloaded operators, which are defined either as

- class member functions or as
- global functions.

Most of operators can be overloaded, e.g.

+	-	*	/	=	==	<	>	-=
+=	&&	<<	++	--	&	!	[]	()

Overloaded operator — as a class member function

```
class vector {
    private:
        int x,y;
    public:
        vector(int a=0, int b=0);
        int operator*(const vector &v) const;
    // ...
};

int vector::operator*(const vector &v) const {
    return x*v.x+y*v.y;
}
```

Note the direct access to the private data members.

Overloaded operator — as a global function

However, it is not always possible to add an operator as a class member function, e.g. to ostream (object cout) to instruct it about printing objects of the type vector. In such cases, an operator has to be defined as a global function and looses the direct access to private data members.

```
class vector {
   private:
      int x,y;
   public:
      vector(int a=0, int b=0);
      int getX(void) const {return x;}
      int getY(void) const {return y;}
};

int operator*(const vector &v1, const vector &v2) {
   return v1.getX()*v2.getX() + v1.getY()*v2.getY();
}
```

Result by value and by reference

As other functions, overloaded operators can return objects by value or by reference.

Assume a and b are vectors, while k is an integer:

```
vector a(1,1), b(5,15);
int k=10;
```

- Operator + (as in a+b) should return *a new object* of the type vector (by value) and leave the objects a and b unaltered.
- Operator *= (as in a*=k) should return *the modified object* a (by reference).

Result by value and by reference

```
class vector {
   private: int x,y;
   public:  vector(int a=0, int b=0) {x=a; y=b};
            vector operator+(const vector &v) const;
            vector & operator+=(const vector &v);
};

vector vector::operator+(const vector &v) const {
   return vector(x+v.x,y+v.y);
} // a new object is created, copied & returned

vector & vector::operator+=(const vector &v) {
   x += v.x;
   y += v.y;
   return *this;
} // the current object (*this) is modified & returned
```

ostream (cout) insertion operator<<

```
class vector {
  private: int x,y;
  public:  vector(int a=0, int b=0) {x=a; y=b};
           int getX(void) const {return x;}
           int getY(void) const {return y;}
};

ostream & operator<< (ostream &ostr, const vector &v) {
  ostr <<"x = " <<v.getX() <<"y = " <<v.getY() <<endl;
  return ostr;
}

int main() {
  vector a(1,1), b(2,2);
  cout <<a <<endl <<b;
  return 0;
}
```

ostream (cout) insertion **operator**<<

```
ostream & operator<< (ostream &ostr, const vector &v) {
  ostr <<"x = " <<v.getX() <<"y = " <<v.getY() <<endl;
  return ostr;
}
```

- Defined as a global function (since as a member function it should be added to the class ostream, which cannot be modified).
- Returns the same ostream by reference in order to enable chaining: cout <<a <<endl <<b;
- Both arguments are passed by reference.
- Only the second argument (vector) is **const**.

Assignment **operator**=

An assignment operator = is very similar to the copy constructor.

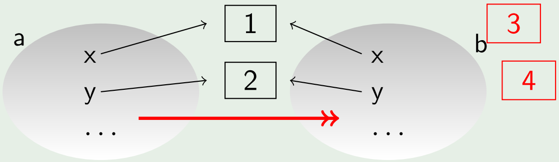
- A user-defined version is necessary when the class uses dynamic memory,
- otherwise a default (shallow) **operator**= is created automatically.

```
class vector {
   private:  int *x,*y;
   public:   vector(const vector &v);
             vector(int a, int b);
             vector &operator=(const vector &v);
};
vector & vector::operator=(const vector &v) {
   *x = *v.x;
   *y = *v.y;
   return *this;
}
```

Assignment **operator**=

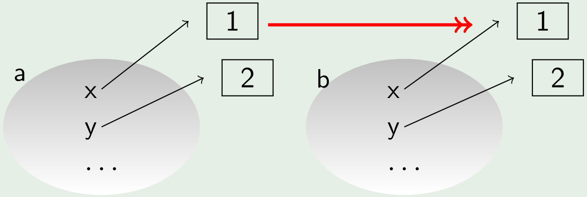
with the default shallow operator=

```
vector a(1,2);
vector b(3,4);
b = a;
```



with a deep user-defined operator=

```
vector a(1,2);
vector b(3,4);
b = a;
```



Outline

STL vector class

Standard (pointer-based) arrays are possible in C++ and very quick, but they have disadvantages:

- troublesome allocation and destruction, especially when multidimensional,
- troublesome resizing.

Standard Templates Library (STL), now a part of the C++ Standard Library, is an object library providing generic containers (arrays, lists etc.) and algorithms for objects of any type, provided they have proper (default or user-defined)

- copy constructor,
- destructor,
- assignment (=) and comparison (==) operators.

See `http://www.sgi.com/tech/stl` or any tutorial on the web.

STL vector class — 1D example

```
#include <vector>
```

1D arrays

```
vector<int> V;          // definition of the variable

V.resize(100, 0);       // resize and initialize
V[10] = 10;
V.push_back(3);         // add 3 at the end
cout <<V.size();        // now 101 elements
```

STL vectors are automatically destroyed in a proper way at the end of their scopes.

STL vector class — 2D example

```
#include <vector>
```

2D arrays

```
vector<vector<int> > V2d;    // note the space!

V2d.resize(100);             // 100 rows
for(int i=0; i<V2d.size(); ++i)
   V2d[i].resize(100);
// each row has 100 elements
V2d[10][10] = -10;
```

STL vectors are automatically destroyed in a proper way at the end of their scopes.

Outline

Homework 6 (20 points)

Matrix class

Available at `http://info.ippt.pan.pl/~ljank`.

E-mail the answer and the source code (all three files) to `ljank@ippt.pan.pl`.