Instantiation of objects

Double meaning of object

In modeling

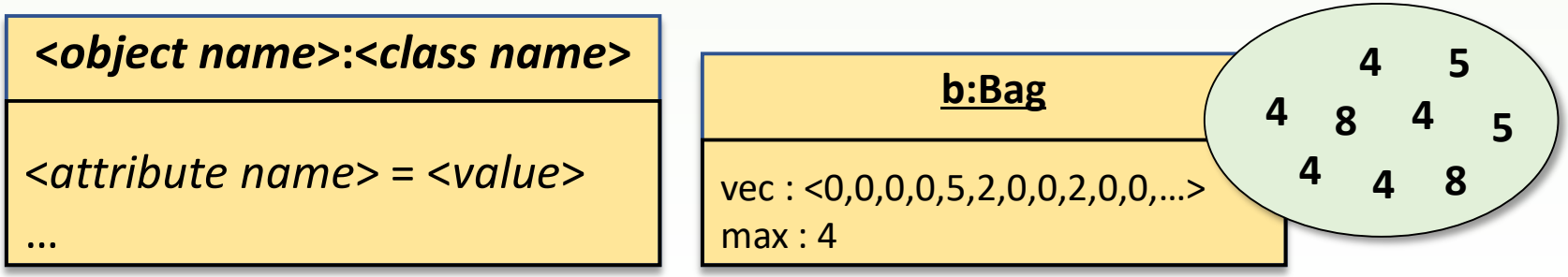
- Object is a unique part of the task to be solved.
- Object is responsible for a part of the data: it hides them and handles (reads and modifies) them exclusively through its methods.
- An object is created and later destroyed.

In programming languages

- Object is the memory allocation, where its data is stored.
- The memory area of the object is only accessable through its methods (except if the data is public).
- The memory allocation (instatiation) of the object is done by its constructor, the destruction is by its destructor.

Notation for object in UML

- An object (the unit responsible for part of the task) is specified by its
  - class, the set of objects with the same attributes and methods. It describes
    - the data (properties, attributes, fields) of the objects in name-type pairs
    - the methods (operations, member functions) that can be called on the object (to manipulate the attributes of the object).
  - name (non obligatory),
  - state (set by all the attribute name-value pairs).



Task

Let us create program that fills an array with polygons. Then it moves all of them along the same vector. Finally, the program calculates the center of the moved polygons. The coordinates of the vertices and the centers should be integers.

Objects:

- polygons
- planar points, vertices and centers
- array of polygons

Single responsibility

O

L

I

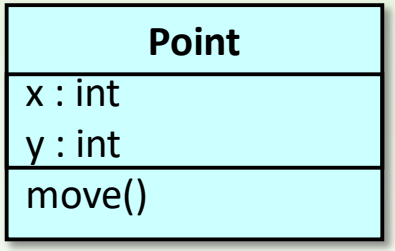
D

Responsibility of the objects:

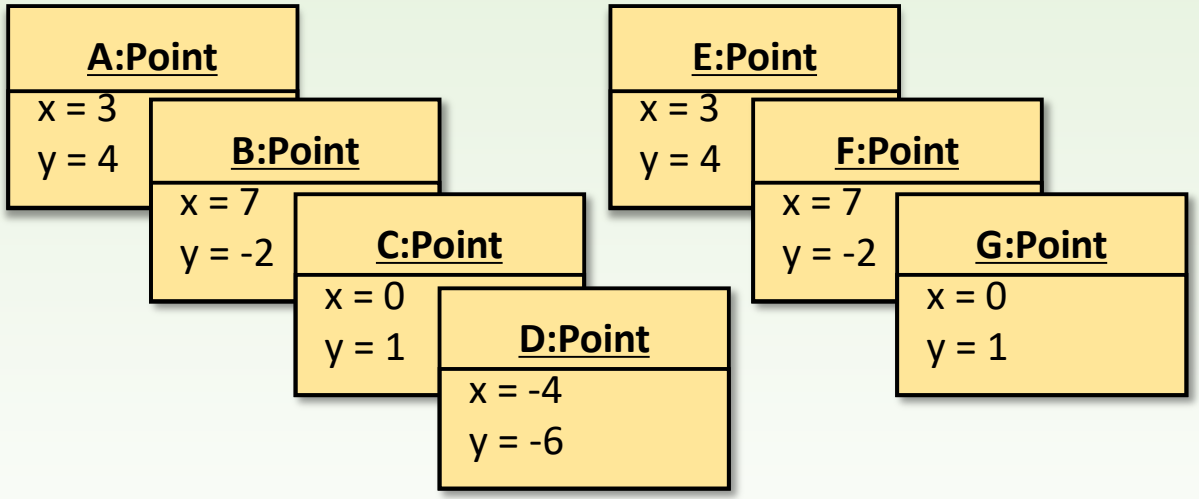
- polygons: move, center calculation, get and set 1. the number of vertices or 2. a given vertice
- planar points: move, get and set its coordinates
- array: get the element at a given index, get its size

Objects of the task

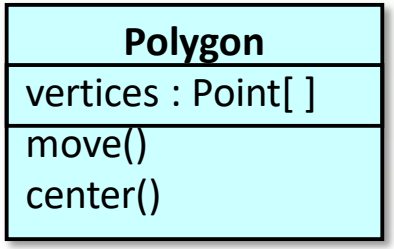
Class Point:



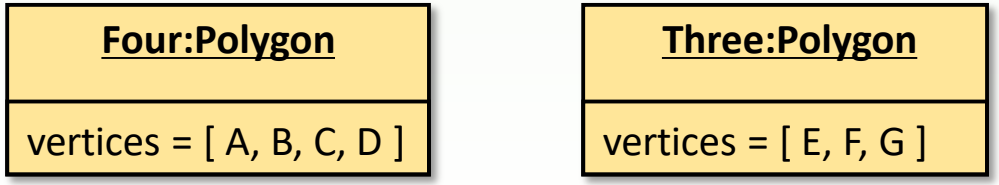
Point objects:



Class polygon:

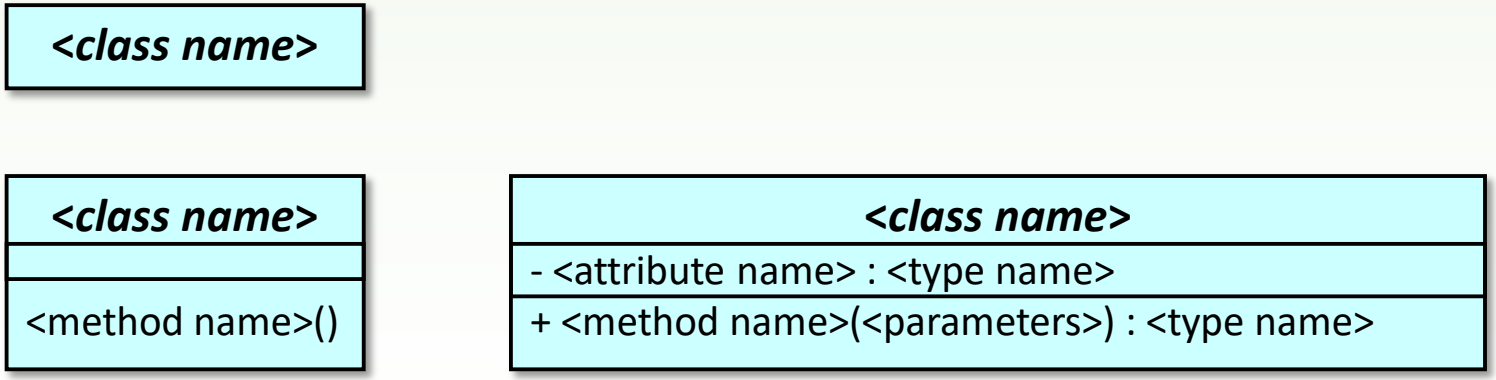


Polygon objects:



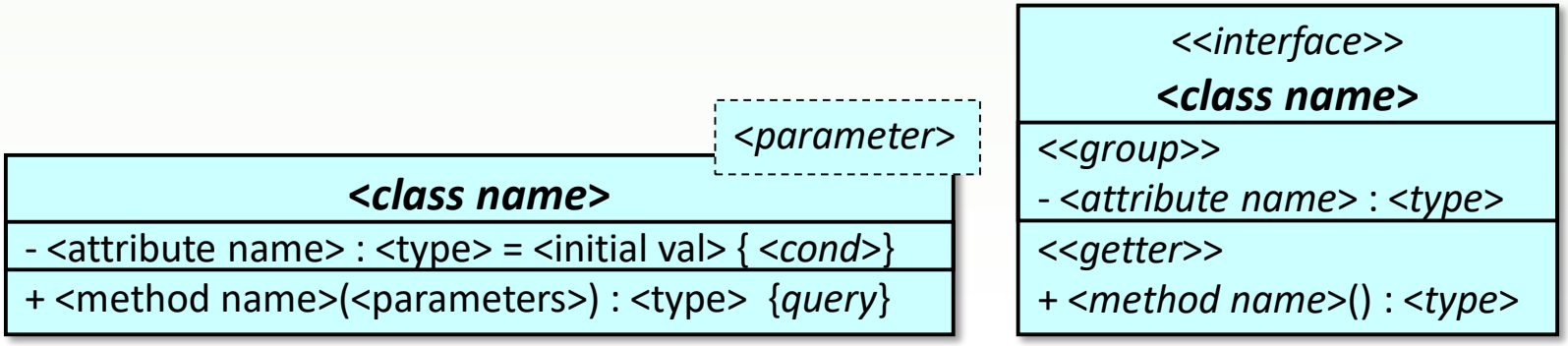
Level of detail of a class description

- Class description evolves gradually during modeling, some details might be missing on a certain level.
  - There are no attributes and/or methods.
    (For enumerations, already only values are given.)
  - There are no attribute types, method return types, method parameters.
  - There are no notations for visibility.

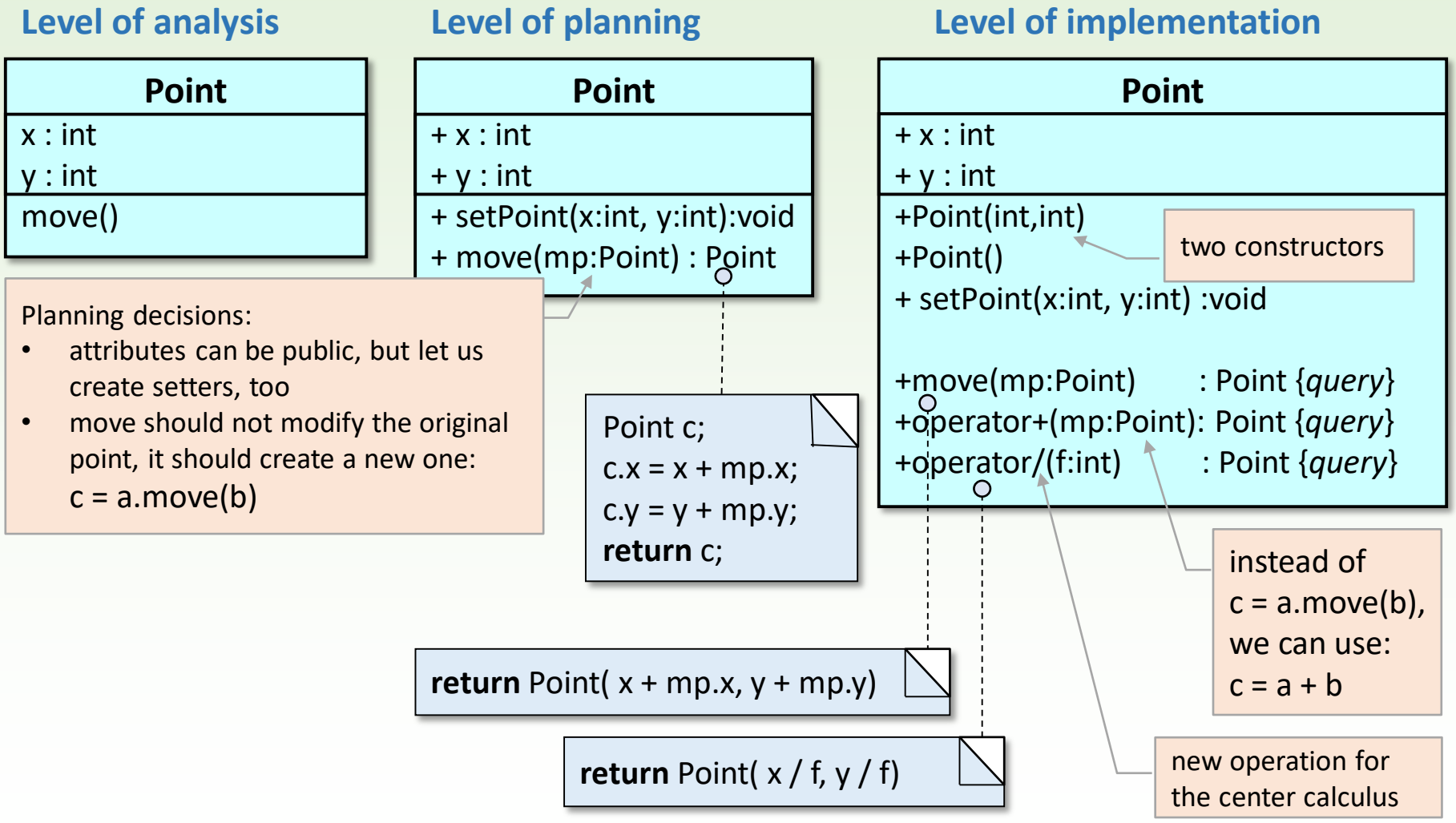


Extensions for class-description

- Constraints for the attributes and methods between {…}
  (E.g. a method that does not modify the attribues is noted by {query})
- Initial value for the attributes (set by the constructor).
- Parameter for a class.
- Stereotype for the class between<<…>> (e.g. <<interface>> , <<enumeration>>), or for a group of attributes or methods (for example <<getter>> , <<setter>>).
  - Public getters and setters are to retrieve and modify the hidden attributes in a supervised way.



Levels of modeling



C++ code for class Point

```
class Point
{
    public:
        Point(int x = 0, int y = 0): _x(x), _y(y) { }

        void setPoint(int x, int y) { _x = x; _y = y; }

        Point move(const Point &mp) const
            { return Point(_x + mp._x, _y + mp._y ); }
        Point operator+(const Point &mp) const
            { return Point(_x + mp._x, _y + mp._y);}
        Point operator/(int f) const
            { return Point(_x / f, _y / f ); }
    public:
        int _x, _y;
};
```

default values of the parameters:

```
Point a;
Point b(3);
Point c(-4, 8);
```

initializes the attributes

query

overrides the operator

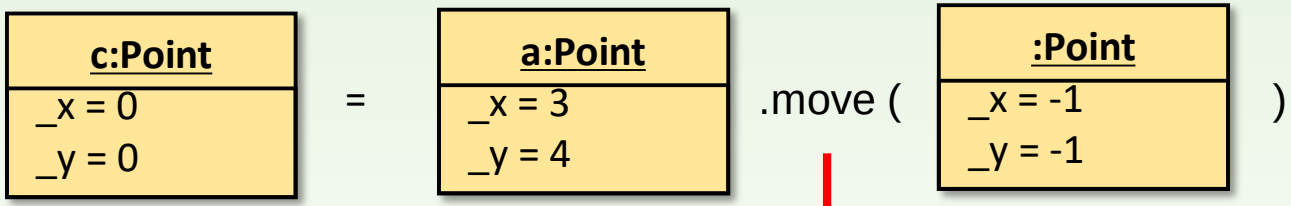
"inline" definition

Instantiatation of Point objects

```
Point a(3,4);
Point c;
c = a.move(Point(-1,-1));
```

method `move()`'s `mp` parameter refers to this point

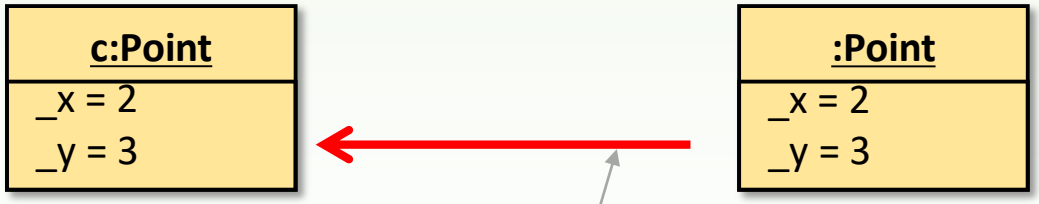
before calling move() :



during calling move() :

This is the temporal point created in `move()`. The coordinates are calculated based on `a` and `mp`.

after calling move() :



Every object has an assignment operator and a copy constructor.

```
Point move(const Point &mp) const {
    return Point(_x + mp._x, _y + mp._y );
}
```

Default pointer of an object

```
class Point{
public:
     Point(int x=0, int y=0):_x(x),_y(y) {}

     void setPoint(int x, int y) { _x = x; _y = y; }
     void setPoint(int x, int y) { this->_x = x; this->_y = y; }

     …

public:
     int _x, _y;
};
```

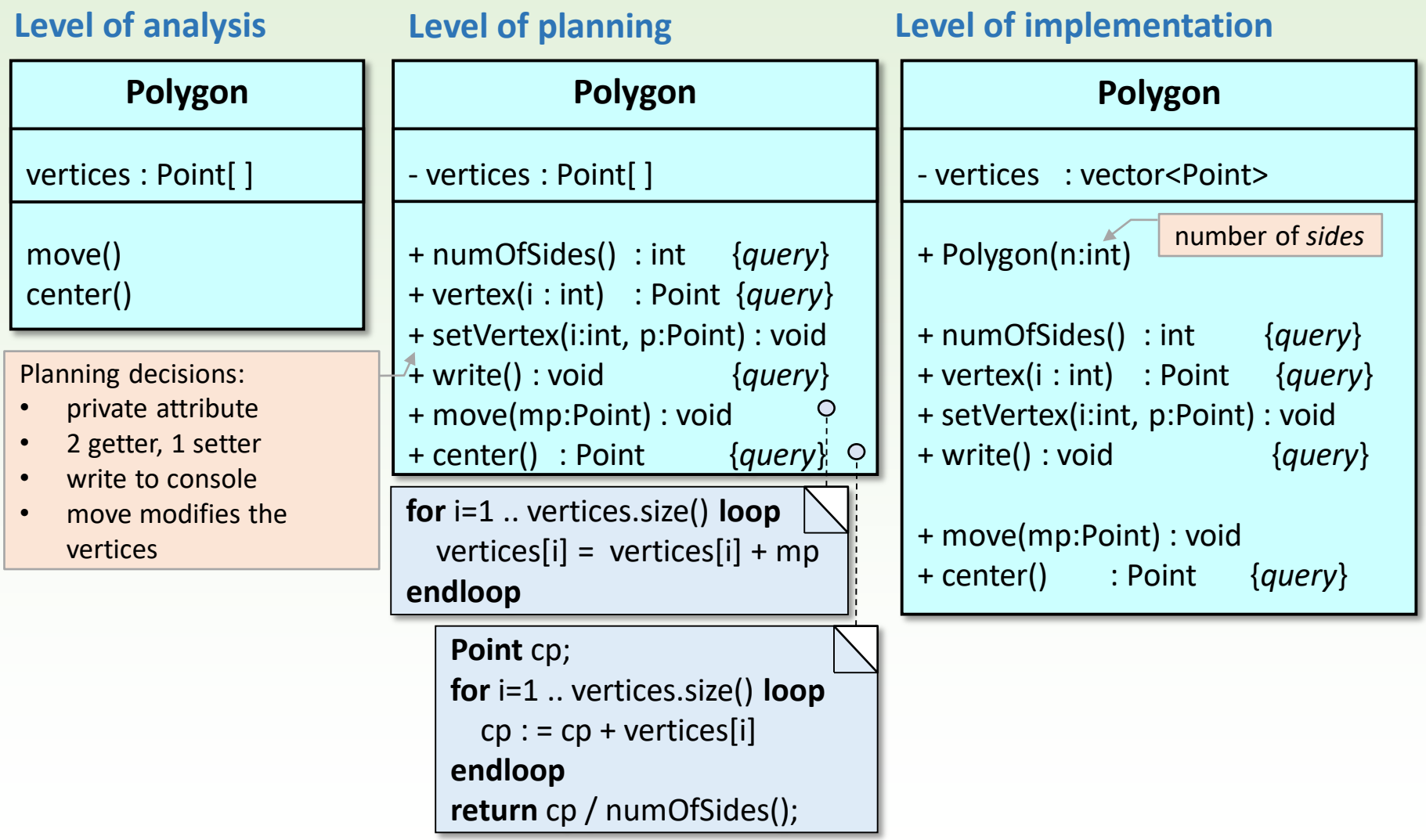
this is a default pointer variable, pointing to the object (’s memory address) on which setPoint() was called:

```
p.setPoint(3, -2)
```

3

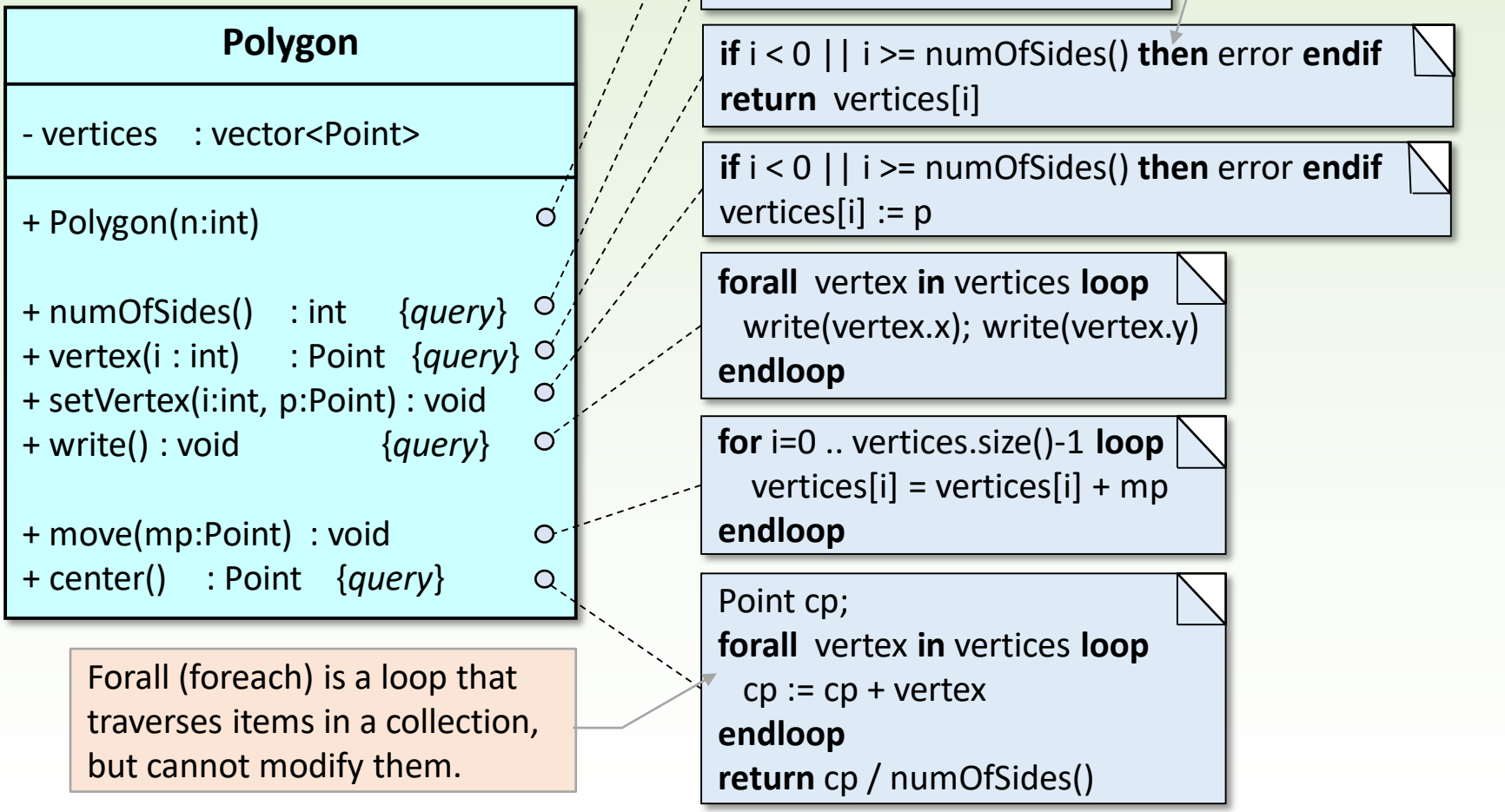
Another criterion of object orientation is open recursion:
the object can always see itself and access its methods and attributes.

Planning of class Polygon



Polygon class

Level of implementation



Code of class Polygon

```
class Polygon
{
public:
     enum Errors{FEW_VERTICES, INDEX_OVERLOADING};

     Polygon(int n) : _vertices(n) {
          if (n < 3) throw FEW_VERTICES;
     }
     unsigned int  sides() const { return _vertices.size(); }

     Point vertex(unsigned int i) const {
          if (i < 0 || i >=sides()) throw INDEX_OVERLOADING;
          return _vertices[i];
     }
     void setVertex(unsigned int i, const Point &p) {
          if (i < 0 || i >=sides()) throw INDEX_OVERLOADING;
          _vertices[i] = p;
     }
     void write() const;
     void  move(const Point &mp);
     Point center() const;
private:
     std::vector<Point> _vertices;
};
```

error cases

Subscript operator

```
class Polygon
{
public:
    …
    Point vertex(unsigned int i) const { … }

    Point Polygon::operator[](unsigned int i) const {
        if (i < 0 || i >=sides()) throw INDEX_OVERLOADING;
        return _vertices[i];
    }

    void setVertex(unsigned int i, const Point &p) { … }

    Point& Polygon::operator[](unsigned int i) {
        if (i < 0 || i >=sides()) throw INDEX_OVERLOADING;
        return _vertices[i];
    }
    …
};
```

Polygon t(3);
Point p = t.vertex(1); // can be replaced by the following:
Point p = t[1];

Polygon t(3);
t.setVertex(1, Point(2,2)); // can be replaced by the following:
t[1] = Point(2, 2);

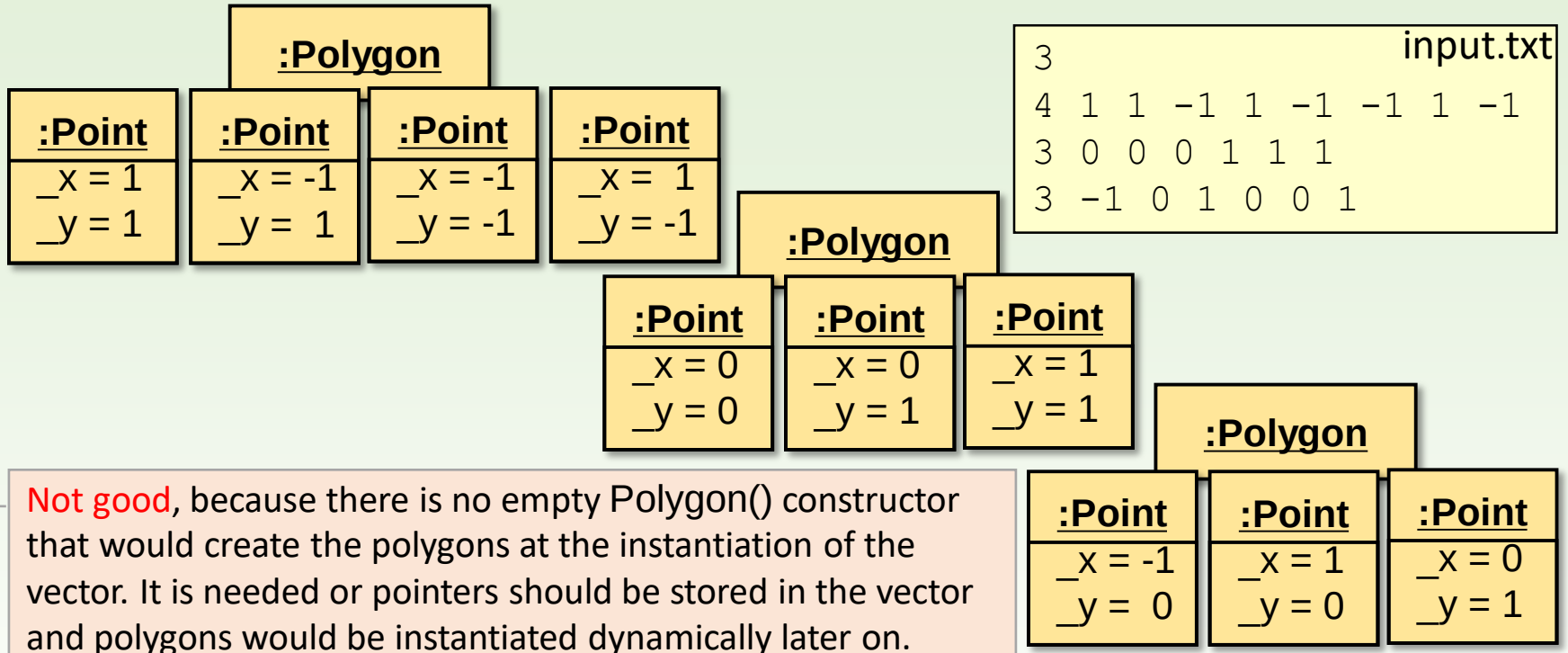
Rest of the methods of Polygon

```
void Polygon::move(const Point &mp)
{
    for(unsigned int i=0; i<_vertices.size(); ++i) {
        _vertices[i] += mp;
    }
}

Point Polygon::center() const
{
    Point center;
    for(Point pvertex : _vertices) {
        center += pvertex;
    }
    return center / sides();
}

void Polygon::write() const
{
    cout << "<";
    for( Point pvertex : _vertices ){
        cout << "(" << pvertex._x
             << "," << pvertex._y << ")";
    }
    cout << ">\n";
}
```

forall (foreach) loop instead of the following:

```
for(unsigned int i=0;
    i<_vertices.size();
    ++i)
{
    center += _vertices[i];
}
```

Population of the task



input.txt

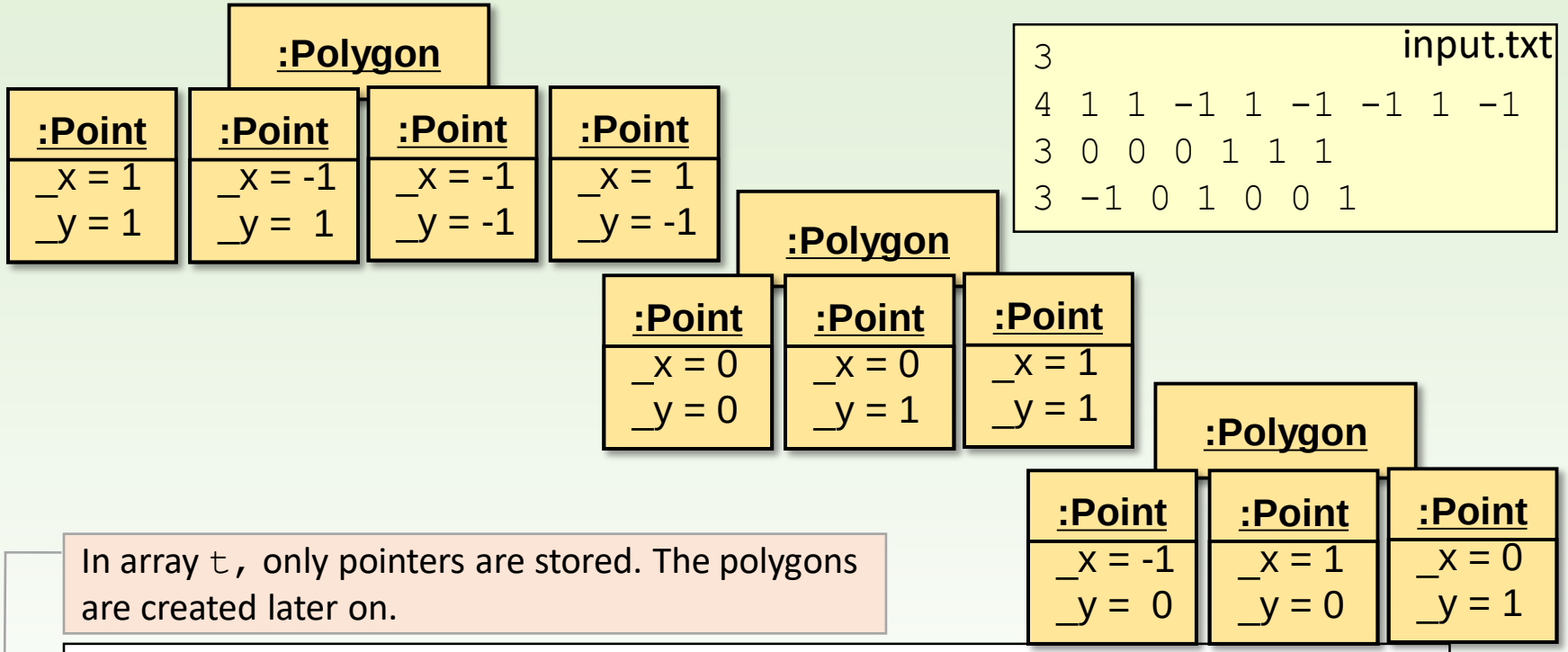
```
3
4 1 1 -1 1 -1 -1 1 -1
3 0 0 0 1 1 1
3 -1 0 1 0 0 1
```

Not good, because there is no empty Polygon() constructor that would create the polygons at the instantiation of the vector. It is needed or pointers should be stored in the vector and polygons would be instantiated dynamically later on.

```
cout << "file name: "; string fn; cin>>fn;
ifstream inp(fn.c_str());
if(inp.fail()) { cout << "Wrong file name!\n"; exit(1); }

int n; inp >> n;
vector<Polygon> t(n);
for( unsigned int i=0; i<n; ++i ) t[i] = set(inp);
```

it would set the i^{th} polygon based on the next line in the file

Population of the task again



input.txt

```
3
4 1 1 -1 1 -1 -1 1 -1
3 0 0 0 1 1 1
3 -1 0 1 0 0 1
```

In array `t`, only pointers are stored. The polygons are created later on.

```
cout << "file name: "; string fn; cin>>fn;
ifstream inp(fn.c_str());
if(inp.fail()) { cout << "Wrong file name!\n"; exit(1); }

unsigned int n; inp >> n;
vector<Polygon*> t(n);
for( unsigned int i=0; i<n; ++i ) t[i] = create(inp);
```

Polygon is created by dynamic memory allocation based on the next line in the file.

```
for( Polygon* p : t ) delete p;
```

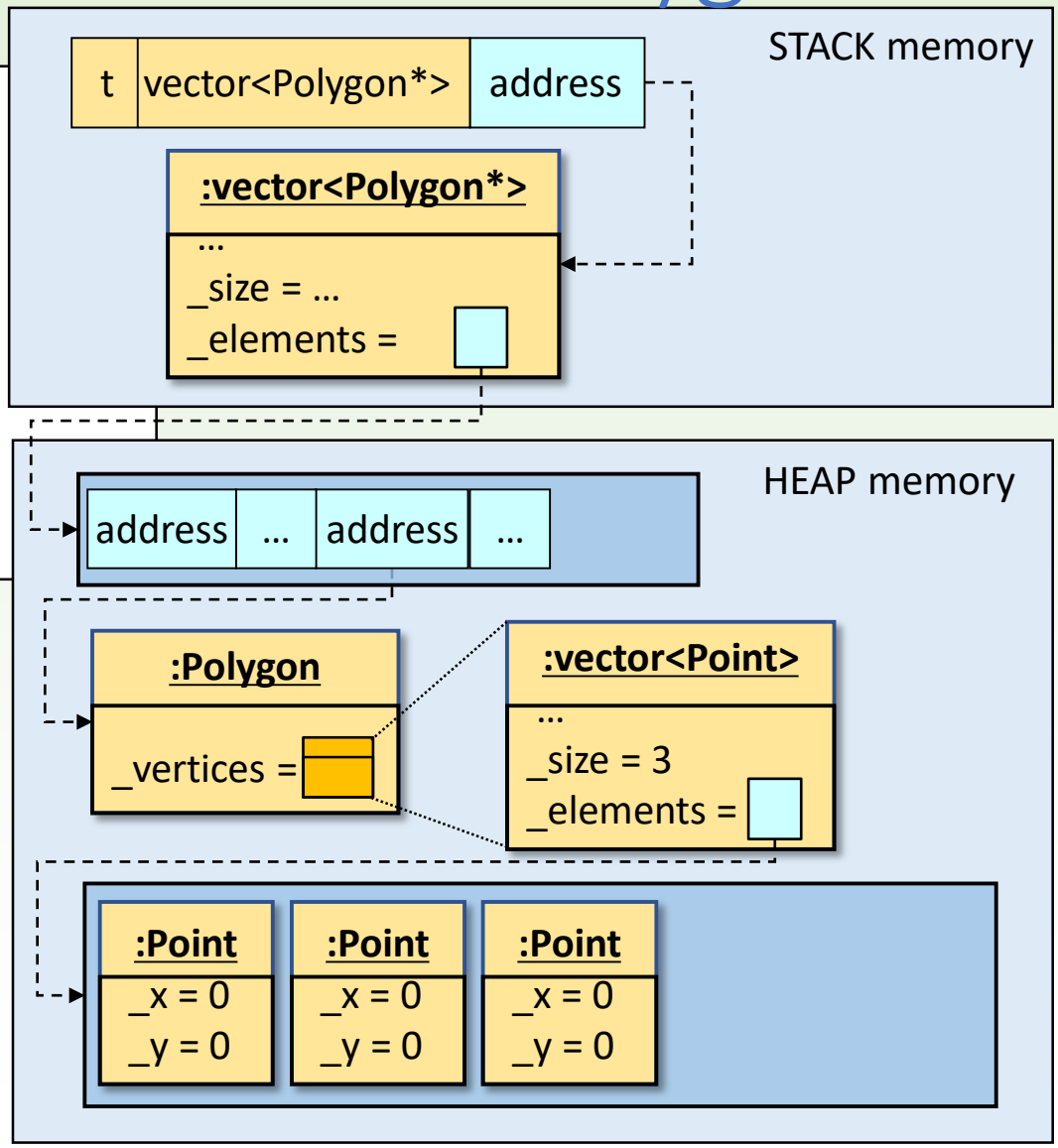
At the end, polygons, the address of which are stored in vector `t`, have to be destroyed.

Dynamic instantiation of Polygon

```
class Polygon {
public:
     Polygon(int n) : _vetices(n)
     {
          if (n < 3) throw
                         FEW_VERTICES;
     }
     …
private:
     vector<Point> _vertices;
};
```

```
t[i] = new Polygon(3);
```

```
int c = t[i]->center();
```



Creating a Polygon

input.txt

```
3
4 1 1 -1 1 -1 -1 1 -1
3 0 0 0 1 1 1
3 -1 0 1 0 0 1
```

```
Polygon* create(ifstream &inp) {
    Polygon *p;
    try{
        int sides;
        inp >> sides;
        p = new Polygon(sides);
        for(int i=0; i < sides; ++i){
            inp >> (*p)[i]._x >> (*p)[i]._y;
        }
    } catch(Polygon::Errors e){
        if(e==Polygon::FEW_VERTICES) cout << " … ";
    }
    return p;
}
```

```
int x, y;
inp >> x >> y;
p->_vertices[i].setPoint(x,y)
```

would be more telling, but as `_vertices` is private and `create()` does not belong to class `Polygon`, it is not accessable.

Factory method

```
Polygon* Polygon::create(ifstream &inp)
{
     Polygon *p;
     try{
          int sides;
          inp >> sides;
          p = new Polygon(sides);
          for(int i=0; i < sides; ++i){
               int x, y;
               inp >> x >> y;
               p->_vertices[i].setPoint(x,y)
          }
     }catch(Polygon::Err
          if( e==Polygon:
     }
     return p;
}
```

It should belong to class Polygon, but it cannot be called on a Polygon, as it should create the Polygon itself.

```
class Polygon {
public:
     enum Errors { … };
     Polygon(int n);
     …
     static Polygon* create(std::ifstream &inp);
private:
     vector<Point*> _vertices;
};
```

class-level method

calling a class-level method

```
t[i] = Polygon::create(inp);
```

Main program

Moving polygons in a sequence (c++ vector) along the same vector, then center calculation.

$A: t : Polygon^n,\ mp : Point,\ cout : Point^n$

$Pre: t = t_0 \wedge mp = mp_0$

notation for concatenation

$Post: mp = mp_0 \wedge t = \bigoplus_{i=1}^{n} < t_0[i].move(mp) > \wedge$

$\wedge\ cout = \bigoplus_{i=1}^{n} < t[i].center() >$

cout := <>

i = 1 .. n

t[i].move(mp)

cout:=cout ⊕ t[i].center()

Two summations (copy):

i∈[m..n]	~	i∈[1 .. n]	i∈[1 .. n]
s	~	t	cout
f(i)	~	$< t_0[i].move(mp) >$	$< t[i].center() >$
H,+,0	~	$Polygon^n, \oplus, <>$	$Point^n, \oplus, <>$

```
for( Polygon *p : t ){
    p->move(Point(20,20));
    p->write();
    Point cp = p->center();
    cout << "(" << cp._x << "," << cp._y << ")\n";
}
```

Type-oriented solution

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <vector>
#include "polygon.h"
#include "point.h"
using namespace std;

int main()
{
    cout << "file name: "; string fn; cin>> fn;
    ifstream inp(fn.c_str());
    if(inp.fail()) { cout << "Wrong file name!\n"; exit(1);}
    int n; inp >> n;
    vector<Polygon*> t(n);
    for (unsigned int i=0; i<n; ++i ) t[i] = Polygon::create(inp);
    for ( Polygon* p : t ){
        p->move(Point(20,20)); p->write();
        Point cp = p->center();
        cout << "(" << cp._x << "," << cp._y << ")\n";
    }
    for ( Polygon* p : t ) delete p;
    return 0;
}
```

population

calculus

destruction

Object-oriented solution

```
int main(){
     Application a;
     a.run();
     return 0;
}
```

```
class Application{
public:
     Application();
     void run();
     ~Application();
private:
     std::vector<Polygon*> t;
};
```

population

calculus

destruction

```
Application::Application(){
     cout << "file name: "; string fn; cin>> fn;
     ifstream inp(fn.c_str());
     if(inp.fail()) {
          cout << "Wrong file name!\n"; exit(1);
     }
     unsigned int n; inp >> n;
     t.resize(n);
     for(unsigned int i=0; i<n; ++i)
          t[i] = Polygon::create(inp);
}
```

```
void Application::run(){
     for ( Polygon* p : t ){
          p->move(Point(20,20)); p->write();
          Point cp = p->center();
          cout << "(" << cp._x << ","
                       << cp._y << ")\n";
     }
}
```

```
Application::~Application(){
     for ( Polygon* p : t ) delete p;
}
```

Menu-controlled object-oriented solution

```
int main()
{
     Menu a;
     a.run();
     return 0;
}
```

```
void Menu::run()
{
     int v = 0;
     do{
          menuWrite();
          cin >> a;  // validation!
          switch(a){
               case 1: case1(); break;
               case 2: case2(); break;
               case 3: case3(); break;
               case 4: case4(); break;
          }
     }while(a != 0);
}
```

```
class Menu{
public:
     Menu(){s = nullptr;}
     void run();
     ~Menu(){ if(s!=nullptr) delete s;}
private:
     Polygon* s;

     void menuWrite();
     void case1();
     void case2();
     void case3();
     void case4();
};
```

```
void Menu::menuWrite(){
     cout << "0 - exit\n";
     cout << "1 - create\n";
     cout << "2 - write\n";
     cout << "3 - move\n";
     cout << "4 - center\n";
}
```

methods would be: create, write, move, center

Menu items

input1.txt

```
4 1 1 -1 1 -1 -1 -1 1
```

input2.txt

```
3 0 0 -1 0 0 -1
```

```
void Menu::case1(){
    if(s!=nullptr) delete s;
    cout << "file name: "; string fn; cin>> fn;
    ifstream inp(fn.c_str());
    if(inp.fail()){ cout << "Wrong file name!\n"; return;}
    s = Polygon::create(inp);
}
```

```
void Menu::case2(){
    if(s==nullptr){ cout << "There is no polygon!\n"; return;}
    s->write();
}
```

```
void Menu::case3(){
    if(s==nullptr){ cout << "There is no polygon!\n"; return;}
    s->move(Point(20,20));
}
```

```
void Menu::case4(){
    if(s==nullptr){ cout << "There is no polygon!\n"; return;}
    Point sp = s->center();
    cout << "(" << cp._x << "," << cp._y << ")\n";
}
```