Introduction into Object - Oriented - Programming

by Borg Enders

Needed previous Knowledge:

- basic knowledge of operating system used
- basic knowledge of compiler used
- basic knowledge of programming language (preferred C++ or Java) used
- some fundamental terms of programming concepts
  - list of parameters for functions
  - global and local program constructs
  - namespaces
  - dynamic memory allocation
  - encapsulation

The used programming language was for this script C++. But also every other objectoriented programming language is suitable for comprehending the examples. The only disadvantage will be that the exemplary solutions are all written in C++.
The compiler used for this script was the gcc 2.95.2 for Windows.

Inhaltsverzeichnis

1 Establishing basic concepts

What is an Object?
An object is a for itself closed program construct existing of properties and methods, which can be used like a self declared data type for the declaration of variables.

What is a closed program construct?
A closed program construct is a closed namespace with both global and only local available properties and methods.

What is a property?
A property is a value which is part of the actual state of an instance of an object.

What is the state of an instance of an object?
The state are all properties and with it all values of an instance of an object for one well-defined moment of program execution.

What is an instance of an Object?
An instance of an object is a variable which has instead of a data type the object as its type and which has to be initialized with a constructor and deinitialized with a destructor.

What is a method?
A method is one way to change properties of an object or to communicate with an instance of an object.

What is a constructor?
A constructor is one special method of an object which is responsible for initializing all properties of one instance of an object. Especially does a constructor take care of allocating enough memory for all dynamically managed properties of an instance. Most compilers call implicit a standard constructor for each instance, but as this is not true for all compilers, it is always sensible to call a constructor explicitly for each instance.

What is a destructor?
A destructor is similar to a constructor a special method of an object which will be called, if an instance of an object is no longer needed. The job of a destructor is to free all memory which has been dynamically allocated for

this instance. In most programming languages the destructor of an instance will be called implicitly by the compiler. This happens for example in Java and C++.

What is the meaning of object-oriented?
This means that a program is build modular out of single objects. But only in sensible granularity. A granularity which is too fine would be for example an own object for each used variable.

What means programming?
Programming means to divide a given problem just formal in smaller problems and than to solve those smaller problems one by one. In the case of object-oriented-programming those smaller problems will mapped on an suitable object hierarchy.

What is an object hierarchy?
An object hierarchy is a skillful mapping of the environment of the program to objects starting with abstract objects to finally concrete instances.

What is an abstract object?
An abstract object is a basic definition for a part of the object hierarchy without offering more than some elemental functionality in itself.

2 Basics of the programming

The here explained syntax is that one of C++. Other languages will have some way different syntactic constructs.

- Object declaration:

```
class <name>
{
 public:
  name();  // standard constructor
  ~name(); // destructor
  <methods declarations>
  <property declarations>
};
```

- Property declaration:

```
<type> <name>;
```

- Method declaration:

```
<return_type> <name>(<param_list>);
```

- Method definition:

```
<return_type> <class_name>::<method_name>(<param_list of method>)
{
 \\ ...
}
```

- Access for methods and properties of an instance:

```
<instance_name>.<name of property>
<instance_name>.<name of method>(<param_list>)
```

- Creating an instance:

```
class Cexample
{
   \\ ...
};
\\ ...
  Cexample instance1=Cexample();
  \\or
  Cexapmple instance2();
\\ ...
```

Here is also to mention that an object can have other constructors with different parameters beside it's standard constructor . But every object should have a standard constructor.
Furthermore declarations are placed normally in C++ in the `header`-files and definitions in the `cpp`-files.

2.1 Exercise

For this exercise implement the following objects with described functionality:
The File `firstgraphics.h`:

```
class Cpoint
{
  public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing to do for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

   int x_coordinate;
   int y_coordinate;
};

class Cline
{
  public:
   /// creates line (x1,y1)---(x2,y2)
   Cline(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Cline();

    /// writes to cout: (x1,y1) --- (x2,y2)
   void draw();

   Cpoint start_point;
   Cpoint end_point;
};

class Clinelist
{
  public:
  /// creates list with n-entries
  Clinelist(int n=1);

  /// gives the allocated memory free
  ~Clinelist();

  /// adds one line to list if there is still an empty place
  /// returns true on success, false otherwise
```

```
  bool add(Cline line);

  /// writes to cout: ... all lines in list each followed by "\n"...
  void draw();

  Cline *list;   /// list of lines
  int size;      /// size of list
  int position;  /// position for adding of next line

};
```

For this file you have to implement the file `firstgraphics.cpp`.

The main program `main.cpp`:

```
#include "firstgraphics.h"

int main()
{
  Clinelist list1=Clinelist(3);
  Cline elem1=Cline(50,50,100,100);
  Cline elem2=Cline(100,100,100,50);
  Cline elem3=Cline(100,50,50,50);
  list1.add(elem1);
  list1.add(elem2);
  list1.add(elem3);
  list1.draw();

  return 0;
}
```

Which creates the following screen output (or at least something quiet similar):

```
 (50,50) --- (100,100)
 (100,100) --- (100,50)
 (100,50) --- (50,100)
```

2.2 Exemplary Solution

```
#include <fstream.h>
```

```
#include <iostream.h>

#include "firstgraphics.h"

/// creates point (x,y)
Cpoint::Cpoint(int x, int y)
{
  x_coordinate=x;
  y_coordinate=y;
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}
```

In C++ all output is handled via streams, where the standard output stream is named `cout`. Strings will be concatenated with the operator `<<` for outputting. This operator is especially defined for all elemental types.

`Cline` uses here for logical reasons the already existing method of the object `Cpoint`, instead of accessing the properties for each point.

```
/// creates line (x1,y1)---(x2,y2)
Cline::Cline(int x1, int y1, int x2, int y2)
{
  start_point=Cpoint(x1,y1);
  end_point=Cpoint(x2,y2);
}

/// nothing todo for this class
Cline::~Cline()
{
}
```

```
/// writes to cout: (x1,y1) --- (x2,y2)
void Cline::draw()
{
  start_point.draw();
  cout<<" --- ";
  end_point.draw();
}
```

In C++ the routines `new` and `delete` are used for dynamically allocation of memory. `new` allocates the memory for the number of requested objects and `delete` frees just those memory again. Whereat the `[]` behind `delete` specify that those property is an array.

```
/// creates list with n-entries
Clinelist::Clinelist(int n)
{
  size=n;
  position=0;
  list = new Cline[size];
}

/// gives the allocated memory free
Clinelist::~Clinelist()
{
  delete[] list;
}

/// adds one line to list if there is still an empty place
/// returns true on success, false otherwise
bool Clinelist::add(Cline line)
{
  if (position<size)
  {
    list[position]=line;
    position++;
    return true;
  } else
  {
    return false;
  }
}
```

```
/// writes to cout: ... all lines in list each followed by "\n"...
void Clinelist::draw()
{
  for( int i=0; i<position;i++)
  {
    list[i].draw();
    cout<<"\n";
  }
}
```

3 Inheritance

One speciality of objects is that an object can be derived of one or more other objects. The objects from which the object will be derived are called parents and the derived itself children.
With this procedure the child inherits all methods and properties of it's parent. This means that a child can access all methods and properties of its parents and especially that all methods of the parents can be called for the child.
Furthermore a child has the possibility to define methods and properties of it's parent in a new way. Those new defined properties and methods hide than implicitly the definition in all parent classes. This method is called overloading.
One particularity for overloading of constructors and destructors has to be kept in mind. In those special methods it is needed to call always the corresponding methods of the parents.
In C++ pay attention, that the calls of the destructor happens automatically and that the call of the constructor of the parent needs to called as preset of a value of the object and not in the body of the constructor itself. If a call for a constructor of the parent is missing than will implicitly the standard constructor of that parent be called.
An other advantage of inheritance is that to each instance of a parent one instance of a child can be assigned. In this case the instance of the parent will than use the methods of the child. The usage of the methods of the child is called late bonding, because only at runtime can be decided, which method really will be called.
Unfortunately in C++ a pointer of an instance of the parent needs to be assigned the address of an instance of a child. This is one of not many points, where C++ does not support the object-oriented concept directly.
By the way the of such assignments it is often useful to define an abstract

object as basis of one branch of the hierarchy of objects. The purpose of this abstract object is to demand a given functionality for all it's children without normally define this functionality itself.
Especially can with the base object that most programming languages offer without problems the different instances of objects be managed in different data structures.

4 Programming of Inheritance

- Inheritance:

```
class <child>: public <ancestor_1>, ..., public <ancestor_n>
{
  // ...
};
```

- Late bonding:

```
class <ancestor>
{
  virtual <return_type> <method_overloadind_name>(<param_list>);

  virtual ~<ancestor>;
};
class <child>: public <ancestor>
{
  <return_type> <method_overloading_name>(<param_list>);
};
```

  As soon as one simple method is declared `virtual` also the standard destructor needs to be declared `virtual`. This garantues that with late bonding to runtime always the right destructor will be called.

- Access to a pointer of an instance of an object:

```
class Cexample
{
  //...
};
// ...
```

```
 Cexample* example;
// ...
  example=new Cexample();
//...
  example-><method_name>(<param_list>);
//...
```

- Value presetting of properties of an object by the way of a constructor

```
Cexample::Cexample(<param_list>):
  <property1>(<preset_value>),
  <property2>(<preset_value>)
{
  //...
}
```

4.1 Excersise

Implement the following objects, which are declared in the file `secondgraphics.h`:

```
class Cgraphicobject
{
  public:
    virtual ~Cgraphicobject();
    virtual void draw();
};

class Cpoint: public Cgraphicobject
{
 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Cgraphicobject
```

```
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
   virtual void draw();

   Cpoint start_point;
   Cpoint end_point;
};

class Csquare: public Crectangle
{
  public:
   /// creates square
   /// S(x1,y1)-
   /// | (x2,y2)
   Csquare(int x1=0, int y1=0, int x2=0, int y2=0);


  /// writes to cout:
  /// S(x1,y1)-
  /// | (x2,y2)
  virtual void draw();
};

class Cpicture {
 public:
  /// creates list with n-entries
  Cpicture(int n=1);

  /// gives the allocated memory free
  ~Cpicture();

  /// adds one graphicobject to list if there is still an empty place
```

```
  /// returns true on success, false otherwise
  bool add(Cgraphicobject* graphic);

  /// writes to cout: ... all graphicobjects in list each followed by "\n"...
  void draw();

  Cgraphicobject** list;   /// list of graphicobjects
  int size;      /// size of list
  int position;  /// position for adding of next line

};
```

For this the file `secondgraphics.cpp` has to be implemented. Based on the similarities to the last exercise it would be good to use the there implemented objects again.

The main program `main.cpp`:

```
#include "secondgraphics.h"

int main() {
  Cpicture picture(2);
  Crectangle rect(50,50,150,100);
  Csquare square(200,200,300,300);
  picture.add(&rect);
  picture.add(&square);
  picture.draw();

  return 0;
}
```

Which creates the following (or a some how similar) screen output:

```
(50,50)-
| (150,100)

S(200,200)-
| (300,300)
```

4.2 Exemplary Solution

```
#include <fstream.h>
```

```
#include <iostream.h>

#include "secondgraphics.h"

Cgraphicobject::~Cgraphicobject()
{
}

void Cgraphicobject::draw()
{
  cout<<"Cgraphicobject::draw()\n";
}

/// creates point (x,y)
Cpoint::Cpoint(int x, int y):
  x_coordinate(x),
  y_coordinate(y)
{
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}

 /// creates rectangle
 /// (x1,y1)-
 /// | (x2,y2)
Crectangle::Crectangle(int x1, int y1, int x2, int y2)
{
  start_point=Cpoint(x1,y1);
  end_point=Cpoint(x2,y2);
}

/// nothing todo for this class
Crectangle::~Crectangle()
{
```

```
}

/// writes to cout:
/// (x1,y1)-
/// | (x2,y2)
void Crectangle::draw()
{
  start_point.draw();
  cout<<"-\n";
  cout<<"| ";
  end_point.draw();
  cout<<"\n";
}
```

In the correct way here has been called the constructor of `Crectangle`. For destructors the inherited will always be called as the last thing in the destructor. For constructors this causes that the calls of inherited constructors initialize all, what is not defined especially for this object, and only after this the for this object special properties will be initialized or special values assigned to inherited properties. For destructors the order of calling is just reversed, as a call of a destructor of a parent will lead to call of the destructor of the parent object of the parent itself (and so on), which will finally free the instance of this object, where by there is no further way to access any special properties of this object.

```
/// creates square
/// (x1,y1)-
/// | (x2,y2)
Csquare::Csquare(int x1, int y1, int x2, int y2):
  Crectangle(x1,y1,x2,y2)
{
}

/// writes to cout:
/// S(x1,y1)-
/// | (x2,y2)
void Csquare::draw()
{
  cout<<"S";
  start_point.draw();
  cout<<"-\n";
  cout<<"| ";
```

```
  end_point.draw();
  cout<<"\n";
}

/// creates list with n-entries
Cpicture::Cpicture(int n)
{
  size=n;
  position=0;
  list = new Cgraphicobject*[size];
}

/// gives the allocated memory free
Cpicture::~Cpicture()
{
  delete[] list;
}

/// adds one graphicobject to list if there is still an empty place
/// returns true on success, false otherwise
bool Cpicture::add(Cgraphicobject* graphic)
{
  if (position<size)
  {
    list[position]=graphic;
    position++;
    return true;
  } else
  {
    return false;
  }
}

/// writes to cout: ... all graphicobjects in list each followed by "\n"...
void Cpicture::draw()
{
  for( int i=0; i<position;i++)
  {
    list[i]->draw();
    cout<<"\n";
  }
}
```

A little extra exercise for interested: instead of using an array of pointer in `Cpicture` use a list of objects, this means replace `Cgraphicobject**` with `Cgraphicobject*` and modify the remaining code to take this into regard.

5 Scopes of properties and methods

To enforce the modularity of objects scopes for properties and methods can be explicitly defined. The two most common are the in previous chapters used scopes: `public` and `private`.
The scope public allows access from outside of the object to each method and properties declared with this scope. In opposite to this methods and properties declared as private can only be accessed from the body of methods of the object itself and all his children, but in C++ children do not have this access.
Especially it is sensible by this to declare nearly all properties of an object private and to encapsule all access to this properties via public methods. With this an instance of an object can guarantee the correctness of all it is properties through the calls of encapsuling methods and gives so the possibility of a correct error handling if not allowed values should be assigned to one property.
In some programming language there is even a third scope. This one is named `protected`. Properties and methods of an object with this scope can only access methods of the same object or as friend declared objects. For this all friendly objects for one given object must be declared explicitly.
In C++ also all children may access methods or properties which where declared protected.

6 Programming with scopes

If no scope is used explicitly C++ uses implicitly the scope private. Additional can also a scope be set for the inheritance of a child from a parent. This scope influences than the scope of the methods and properties of the parent for just this child.

```
<child_class>: <public/protected/private> <ancestor_class>
{
  friend class <friend_class>;

  public:
    /// all methods and properties, which should be public

  protected:
    /// all methods and properties, which should be protected
```

```
  private:
    /// all methods and properties, which should be private

};
```

Scopes in children depending on the set scope for the inheritance:

Scope in Parent	Scope of Inheritance		
	private	protected	pubic
public	private	protected	public
protected	private	protected	protected
private	-	-	-

6.1 Exercise

The following objects declared in file `thirdgraphics.h` have to be implemented:

```
class Cgraphicobject
{
  public:
    virtual ~Cgraphicobject();
    virtual void draw();
};

class Cpoint: public Cgraphicobject
{
 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

 private:
   int x_coordinate;
```

```
   int y_coordinate;
};

class Crectangle : public Cgraphicobject
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
   virtual void draw();

  protected:
   Cpoint start_point;
   Cpoint end_point;
};

class Csquare: public Crectangle
{
  public:
   /// creates square
   /// S(x1,y1)-
   /// | (x2,y2)
   /// and controls that x2-x1=y2-y1 is
   /// and if not decreases the larger side
   /// to the size of the smaller side
   Csquare(int x1=0, int y1=0, int x2=0, int y2=0);

  /// writes to cout:
  /// S(x1,y1)-
  /// | (x2,y2)
  virtual void draw();
};

class Cpicture {
 public:
```

```
  /// creates list with n-entries
  Cpicture(int n=1);

  /// gives the allocated memory free
  ~Cpicture();

  /// adds one graphicobject to list if there is still an empty place
  /// returns true on success, false otherwise
  bool add(Cgraphicobject* graphic);

  /// writes to cout: ... all graphicobjects in list each followed by "\n"...
  void draw();

 private:
  Cgraphicobject** list;   /// list of graphicobjects
  int size;      /// size of list
  int position;  /// position for adding of next line
};
```

For this the file `thirdgraphics.cpp` has to be implemented. On the base of the similarity to the last exercise it would really be wise, to reuse the there implemented objects.

The main program `main.cpp`:

```
#include "thirdgraphics.h"

int main() {
  Cpicture picture=Cpicture(2);
  Crectangle rect=Crectangle(50,50,150,100);
  Csquare square=Csquare(200,200,300,300);
  picture.add(&rect);
  picture.add(&square);
  picture.draw();

  return 0;
}
```

Which should generate the following output (or something quite similar):

```
WARNING
```

```
S(200,200)-
| (250,300)
is not a square!
correcting endpoint to (250,250)


(50,50)-
| (150,100)

S(200,200)-
| (250,250)
```

6.2 Exemplary Solution

Here only the changes to the sample solution `secondgraphics.cpp`.

```
// ...
#include "thirdgraphics.h"
// ...

/// creates square
/// (x1,y1)-
/// | (x2,y2)
/// and controls that x2-x1=y2-y1 is
/// and if not decreases the larger side
/// to the size of the smaller side
Csquare::Csquare(int x1, int y1, int x2, int y2):
  Crectangle(x1,y1,x2,y2)
{
    int dx=x2-x1;
    int dy=y2-y1;

    if (dx!=dy)
    {
      cout<<"WARNING"<<"\n";
      draw();
      cout<<"is not a square!"<<"\n";

      if (dx<dy)
      {
        y2=y1+dx;
      } else
      {
```

```
        x2=x1+dy;
      }
      end_point=Cpoint(x2,y2);

      cout<<"correcting endpoint to ";
      end_point.draw();
      cout<<"\n\n\n";
    }
}
// ...
```

A small additional exercise for interested: declare the properties `start_point` and `end_point` of the object `Crectangle` as `private` and change the rest of the code accordingly.

7 Design Patterns

Design patterns are solution approaches for often recurring problems of programming.
Here only four of those patterns should be introduced. But there are a lot more Design Patterns, for starting you should take a look at the book "Design Patterns: Elements Of Reusable Object-Oriented Software" (Addison-Wesley) by Erich Gamma.
The here introduced patterns are:

Composite Pattern: This patterns composes one object on the base of some similar objects, for example this often the case for graphical user interfaces.

Template Method Pattern: This patterns defines with an abstract Object a basic algorithm and abstract methods for all needed methods of this algorithm. Than all children of this object can use this algorithm. This pattern can for example be used for sorting algorithms.

Factory Pattern: Here can one of many objects be the result of one method. The calling objects decides, which object is needed. This can be used for example, for defining similar objects with same methods but different properties.

Chain Of Responsibility: With this pattern a chain of objects will be created, which are responsible for a special event while the program is executed. For example this can be used for the handling of mouse events by a graphical interface.

8 Composite Pattern

The composite pattern represents a recursive part-whole-relation between objects and enables with this an uniform handling of the atomic parts of the object.
For this an abstract base class demands the following functionality:

- Definition of all needed methods of derived objects
- adding of components
- removing of components
- access to any added components

The public part of the object definition looks than like this:

```
class CComponent
{
  public:
   // definition of needed methods
```

```
   virtual void example_method();

   virtual int add(CComponent* c);

   virtual int remove(CComponet* c);

   virtual CComponent* get_child(int index);

   virtual ~CComponent();
};
```

Base on this object other composed object or atomic objects can be derived, which public parts looks like this:

```
class CComposite: public CComponent
{
  public:
   // definition of needed methods
   void example_method();

   int add(CComponent* c);

   int remove(CComponet* c);

   CComponent* get_child(int index);
};

class Catom: public CComponent
{
  public:
   // definition of needed methods
   void example_method();
};
```

8.1 Exercise

The following objects have to be implemented, declared in the file `fourthgraphics.h`:

```
class Cgraphicobject
{
  public:
    virtual void draw();
```

```
    virtual int add(Cgraphicobject g);

    virtual Cgraphicobject* get_child(int index);

    virtual ~Cgraphicobject();
};

class Cpoint: public Cgraphicobject
{
 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

 private:
   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Cgraphicobject
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
   virtual void draw();

 protected:
   Cpoint start_point;
   Cpoint end_point;
```

```
};

class Cpicture : public Cgraphicobject
{
 public:
  /// creates list with n-entries
  Cpicture(int n=1);

  /// gives the allocated memory free
  ~Cpicture();

  /// adds one graphicobject to list if there is still an empty place
  /// returns true on success, false otherwise
  int add(Cgraphicobject graphic);

  /// writes to cout: ... all graphicobjects in list each followed by "\n"...
  void draw();

  // result the element at position index in list
  Cgraphicobject* get_child(int index);

 private:
  Cgraphicobject* list;   /// list of graphicobjects
  int size;      /// size of list
  int position;  /// position for adding of next line
};
```

Here was gone without `remove` of `Cgraphicobject`, as for this an equality test of objects would have been needed.
For this the file `fourthgraphics.cpp` is to be implemented. Based on the similarity to the last exercise it is recommended to reuse the already for it implemented objects.

The main program `main.cpp`:

```
#include "fourthgraphics.h"

int main() {
  Cpicture picture=Cpicture(2);
  Crectangle rect=Crectangle(50,50,150,100);
  Crectangle square=Crectangle(200,200,300,300);
  picture.add(&rect);
```

```
  picture.add(&square);
  picture.add(&rect);
  picture.draw();

  return 0;
}
```

Which should produce the following (or a quite similar) output:

```
Error: list is full

(50,50)-
| (150,100)

(200,200)-
| (300,300)
```

8.2 Exemplary solution

Here only the changes beyond the sample solution `thirdgraphics.cpp`.

```
//...
#include "fourthgraphics.h"
//...

int Cgraphicobject::add(Cgraphicobject* g)
{
  cout<<"Cgraphicobject::add";
  return -1;
}

Cgraphicobject* Cgraphicobject::get_child(int index)
{
  cout<<"Cgraphicobject::get_child(int index)";
  return NULL;
}
//...

/// adds one graphicobject to list if there is still an empty place
/// returns true on success, false otherwise
int Cpicture::add(Cgraphicobject* graphic)
```

```
{
  if (position<size)
  {
    list[position]=graphic;
    position++;
    return position-1;
  } else
  {
    cerr<<"Error: list is full"<<"\n\n";
    return -1;
  }
}
//...
```

In c++ commonly all error messages are written to the predefined stream `cerr`. These stream is handled completly in the same way as standard output stream-

```
// result the element at position index in list
Cgraphicobject* Cpicture::get_child(int index)
{
  if (index<position)
  {
    return list[index];
  } else
  {
    return NULL;
  }
}
```

A small exercise for interested: change the main program in a way, that no longer the method `draw` of `picture` will be called, but each element of `picture` will be acquired via `get_child` and only than `draw` of this element will be called.

9 Template Method Pattern

With the help of the pattern Template Method can a useful algorithm be implemented for an abstract object. For this the template object just defines the algorithm itself and abstract methods for all needed methods of the algorithm. This abstract methods have than to be implemented by the concrete children of the template object which want to use the algorithm.

The public part of a template object looks like the following code:

```
class Ctemplate
{
  public:
    // nice algorithm
    void templateMethod();

    // for templateMethod needed methods
    virtual void example_method1();

    virtual void example_method2();

    virtual ~Ctemplate();
};
```

The public part of a concrete children looks than like the following:

```
class Cconcrete: public Ctemplate
{
  public:

    // for templateMethod needed methods
    void example_method1();

    void example_method2();
};
```

9.1 Exercise

The following objects need implementing, declared in the file `fifthgraphics.h`:

```
class Cpoint
{
  friend class Clist;

 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();
```

```
   /// writes to cout: (x,y)
   void draw();

 protected:
   int x_coordinate;
   int y_coordinate;
};

class Clisttemplate
{
  public:
    /// algorithm sorts points into list
    /// depending on distance to (0,0)
    void insert(Cpoint p);

    /// needed operations for algorithm

    ///adds one point to a pointlist
    virtual void add(Cpoint p);

    /// swaps the points with index from and to
    virtual void swap(int from, int to);

    /// size of list
    virtual int size();

    /// distance of point i to (0,0)
    virtual int distance(int i);

    virtual ~Clisttemplate();
};

class Clist : public Clisttemplate
{
 public:
  /// creates list with size n
  Clist(int n);

  /// gives the allocated memory free
  ~Clist();
```

```
  /// writes to cout: ... all points in list each followed by "\n"...
  void draw();

  ///adds one point to a pointlist if there is still en ampty place
  void add(Cpoint p);

  /// swaps the points with index from and to
  void swap(int from, int to);

  /// size of list
  int size();

   /// distance of point i to (0,0)
  int distance(int i);

 private:
  Cpoint* list;   /// list of points
  int max_size;       /// size of list
  int position;   /// position for next empty place in list
};
```

The main program `main.cpp`:

```
#include "fifthgraphics.h"

int main() {
   Clist list(5);
   Cpoint p1=Cpoint(10,20);
   list.insert(p1);
   Cpoint p2=Cpoint(5,15);
   list.insert(p2);
   Cpoint p3=Cpoint(50,50);
   list.insert(p3);
   Cpoint p4=Cpoint(10,10);
   list.insert(p4);
   Cpoint p5=Cpoint(12,13);
   list.insert(p5);
   list.draw();

   return 0;
}
```

Which should produce the following or a quite similar output:

```
(10,10)
(5,15)
(12,13)
(10,20)
(50,50)
```

9.2 Exemplary Solution

Here the exemplary solution:

```
#include <fstream.h>
#include <iostream.h>

#include "fifthgraphics.h"

/// creates point (x,y)
Cpoint::Cpoint(int x, int y)
{
  x_coordinate=x;
  y_coordinate=y;
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}

Clisttemplate::~Clisttemplate()
{
}

/// algorithm sorts points into list
/// depending on distance to (0,0)
void Clisttemplate::insert(Cpoint p)
{
```

```
  add(p);

  for(int i=size()-1;i>0;i--)
  {
    if (distance(i)<distance(i-1))
      swap(i-1,i);
  }
}
```

This algorithm is a simple form of the insertion-sort-algorithm. The advantage of this algorithm is that it only needs linear time for inserting elements into a sorted list.

```
///adds one point to a pointlist
void Clisttemplate::add(Cpoint p)
{
 cout<<"Clisttemplate::add(Cpoint p)"<<"\n";
}

/// swaps the points with index from and to
void Clisttemplate::swap(int from, int to)
{
  cout<<"Clisttemplate::swap(int from, int to)\n";
}

int Clisttemplate::size()
{
  cout<<"Clisttemplate::size()\n";
  return 0;
}

int Clisttemplate::distance(int i)
{
  cout<<"Clisttemplate::distance(int i)\n";
  return 0;
}

/// creates list with size n
Clist::Clist(int n)
{
  max_size=n;
```

```
  position=0;
  list = new Cpoint[n];
}

/// gives the allocated memory free
Clist::~Clist()
{
  delete[] list;
}

/// adds one point to the end of the list
void Clist::add(Cpoint p)
{
  if (position<max_size)
  {
    list[position]=p;
    position++;
    return;
  } else
  {
    cerr<<"Error: list is full"<<"\n\n";
    return;
  }
}

void Clist::swap(int from, int to)
{
  if ((from>=0) && (from<position) &&
      (to>=0)   && (to<position)    )
  {
    Cpoint h=list[from];
    list[from]=list[to];
    list[to]=h;
  } else
  {
    cerr<<"Error: list has size "<<size()<<"\n";
    cerr<<"swap("<<from<<","<<to<<") out of index\n";
  }
}

/// writes to cout: ... all graphicobjects in list each followed by "\n"...
void Clist::draw()
{
```

```
  for( int i=0; i<position;i++)
  {
    list[i].draw();
    cout<<"\n";
  }
}

int Clist::size()
{
  return position;
}

int Clist::distance(int i)
{
  return list[i].x_coordinate*list[i].x_coordinate+
         list[i].y_coordinate*list[i].y_coordinate;
}
```

10 Factory Pattern

The factory pattern defines a homogenous interface for the creation of instances, where the instances can be created from different objects. The calling instance decides, which object should be used for creating an instance.
The public part of for the needed objects looks like the following:

```
class Ccreator
{
  public:
    Cproduct* factoryMethod(<params>);

  //...
};

class Cproduct
{
  //...
};

class Cproduct1: public Cproduct
{
  //...
};
```

```
class Cproduct2: public Cproduct
{
 //...
};
```

This could than be called like in the following example:

```
//...
  Ccreator factory();
//...
  Cproduct1* example=(Cproduct1*)factory.factoryMethod(<value for
                                                        product1 as result>);
//...
```

Here is `(Cproduct1*)` a so called `typecast`. This means that a variable of a given typed should be used in a way as if the variable would have the type between the brackets.

10.1 Exercise

The following objects are to implement, which are declared in the file `sixthgraphics.h`:

```
class Cgraphicobject
{
  public:
    virtual void draw();
    virtual ~Cgraphicobject();
};

class Cpoint: public Cgraphicobject
{
 public:
   /// creates (x,y)
   Cpoint(int x=0,int y=0);

   /// creates point (x,y)
   Cpoint(const Cpoint& init);

   /// nothing todo for this class
   ~Cpoint();
```

```
   /// writes to cout: (x,y)
   void draw();

 private:
   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Cgraphicobject
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(Cpoint start,Cpoint end);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
    void draw();

 private:
   Cpoint start_point;
   Cpoint end_point;
};

class Ctriangle : public Cgraphicobject
{
 public:
   /// creates triangle
   /// (x1,y1)-(x2,y2)
   /// | (x3,y3)
   Ctriangle(Cpoint a, Cpoint b, Cpoint c);

   /// nothing todo for this class
   ~Ctriangle();

   /// writes to cout:
   /// (x1,y1)-(x2,y2)
   /// | (x3,y3)
```

```
   void draw();

 private:
   Cpoint a_point;
   Cpoint b_point;
   Cpoint c_point;
};

class Clist
{
 public:
  /// creates list with size n
  Clist(int n);

  /// gives the allocated memory free
  ~Clist();

  ///adds one point to a pointlist if there is still en ampty place
  void add(Cpoint p);

  /// creates a Ctriangle for type==0 and a Crectangle for type==1

 Cgraphicobject* factoryMethod(int type);

 private:
  Cpoint* list;   /// list of points
  int max_size;       /// size of list
  int position;   /// position for next empty place in list
};
```

The main program `main.cpp`:

```
#include "sixthgraphics.h"

int main() {
   Clist list(3);
   Cpoint p1=Cpoint(10,20);
   list.add(p1);
   Cpoint p2=Cpoint(5,15);
   list.add(p2);
   Cpoint p3=Cpoint(50,50);
   list.add(p3);
```

```
   Ctriangle* t=(Ctriangle*)list.factoryMethod(0);
   t->draw();

   Crectangle* r=(Crectangle*)list.factoryMethod(1);
   r->draw();

   return 0;
}
```

Which should produce the following output (or something quite similar):

```
(10,20)-(5,15)
| (50,50)
(10,20)-
| (5,15)
```

10.2 Exemplary Solution

Here the exemplary solution:

```
#include <fstream.h>
#include <iostream.h>

#include "sixthgraphics.h"

Cgraphicobject::~Cgraphicobject()
{
}

void Cgraphicobject::draw()
{
  cout<<"Cgraphicobject::draw()";
}

///creates (x,y)
Cpoint::Cpoint(int x,int y)
{
  x_coordinate=x;
  y_coordinate=y;
}
```

```
/// creates point (x,y)
Cpoint::Cpoint(const Cpoint& init)
{
  x_coordinate=init.x_coordinate;
  y_coordinate=init.y_coordinate;
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}

 /// creates rectangle
 /// (x1,y1)-
 /// | (x2,y2)
Crectangle::Crectangle(Cpoint start,Cpoint end)
{
  start_point=Cpoint(start);
  end_point=Cpoint(end);
}

/// nothing todo for this class
Crectangle::~Crectangle()
{
}

/// writes to cout:
/// (x1,y1)-
/// | (x2,y2)
void Crectangle::draw()
{
  start_point.draw();
  cout<<"-\n";
  cout<<"| ";
  end_point.draw();
  cout<<"\n";
}
```

```
 /// creates triangle
 /// (x1,y1)-(x2,y2)
 /// | (x3,y3)
Ctriangle::Ctriangle(Cpoint a,Cpoint b,Cpoint c)
{
  a_point=Cpoint(a);
  b_point=Cpoint(b);
  c_point=Cpoint(c);
}

/// nothing todo for this class
Ctriangle::~Ctriangle()
{
}

/// writes to cout:
/// (x1,y1)-(x2,y2)
/// | (x3,y3)
void Ctriangle::draw()
{
  a_point.draw();
  cout<<"-";
  b_point.draw();
  cout<<"\n";
  cout<<"| ";
  c_point.draw();
  cout<<"\n";
}

/// creates list with size n
Clist::Clist(int n)
{
  max_size=n;
  position=0;
  list = new Cpoint[n];
}

/// gives the allocated memory free
Clist::~Clist()
{
  delete[] list;
}
```

```
/// adds one point to the end of the list
void Clist::add(Cpoint p)
{
  if (position<max_size)
  {
    list[position]=p;
    position++;
    return;
  } else
  {
    cerr<<"Error: list is full"<<"\n\n";
    return;
  }
}

Cgraphicobject* Clist::factoryMethod(int type)
{
   if (type==0)
   {
      if (position>=3)
      {
        Ctriangle* t=new Ctriangle(list[0],list[1],list[2]);
        return t;
      } else
      {
        cerr<<"Not enough points to create triangle\n";
        Cpoint p=Cpoint();
        Ctriangle* t=new Ctriangle(p,p,p);
        return t;
      }
   }

   if (type==1)
   {
      if (position>=1)
      {
        Crectangle* r=new Crectangle(list[0],list[1]);
        return r;
      } else
      {
        cerr<<"Not enough points to create rectangle\n";
        Cpoint p=Cpoint();
```

```
        Crectangle* r=new Crectangle(p,p);
        return r;
      }
   }

  return NULL;
}
```

Here the operator `new` is used for explicitly reserving memory. With this the instance are now longer local and will not be destroyed on leaving the method. The destruction of the objects should be happen explicitly.

11 Chain of Responsibilty

The pattern Chain of Responsibiity is motivated by creating messages or events in one object, which should be handled by instances of other objects. To guarantee, that each event or message is handled by the correct instance a chain of responsibility has to be created. Through this chain will than the events and messages be propagated and each instance handles the events or messages, which are relevant for it.
For this a definition is first needed of an object for handling events or messages and an object which represents events or messages. The public part of this objects look like the following code:

```
class Cevent
{
  //...
};

class Chandler
{
  public:
    virtual void handle(Cevent e);

    void setsuccessor(Chandler* h);

    virtual ~Chandler();
};
```

In the private part of `Chandler` there is the property `Chandler* successor`, which is used for the creation of the chain.
At the end of the method `handle` has to be the same method of the instance,

which has been set with the method `setsuccesor`, called to propagate the event in the chain.
With `Chandler` as base concrete objects for handling events can now be derived. The public code of those looks like the following:

```
class Chandlerspec: public Chandler
{
  public:
    void handle(Cevent e);
};
```

11.1 Exercise

To implement are the following objects, declared in the file `seventhgraphics.h`:

```
class Cevent
{
   public:
     Cevent(int t=-1);

     ~Cevent();

     // type==0
     bool isDrawEvent();

   private:
     int type;
};

class Chandler
{
  public:
    Chandler();

    virtual ~Chandler();

    virtual void handle(Cevent e);

    void setsuccessor(Chandler* h);

  private:
    Chandler* succ;
};
```

```
class Cpoint
{
 public:
   /// creates point (x,y)
   Cpoint(int x=0, int y=0);

   /// nothing todo for this class
   ~Cpoint();

   /// writes to cout: (x,y)
   void draw();

 private:
   int x_coordinate;
   int y_coordinate;
};

class Crectangle : public Chandler
{
 public:
   /// creates rectangle
   /// (x1,y1)-
   /// | (x2,y2)
   Crectangle(int x1=0, int y1=0, int x2=0, int y2=0);

   /// nothing todo for this class
   ~Crectangle();

    /// writes to cout:
    /// (x1,y1)-
    /// | (x2,y2)
    /// if e is drawevent
   void handle(Cevent e);

 protected:
   Cpoint start_point;
   Cpoint end_point;
};

class Cpicture : public Chandler
{
 public:
  Cpicture();
```

```
  /// nothing to do
  ~Cpicture();

  void handle(Cevent e);

  /// creates draw event
  void draw();
};
```

The main program `main.cpp`:

```
#include "seventhgraphics.h"

int main() {
  Cpicture picture=Cpicture();
  Crectangle rect=Crectangle(50,50,150,100);
  picture.setsuccessor(&rect);
  Crectangle rect2=Crectangle(200,200,250,300);
  rect.setsuccessor(&rect2);
  picture.draw();

  return 0;
}
```

Which will create the following output (or something quite similar):

```
Drawevent created
(50,50)-
| (150,100)
(200,200)-
| (250,300)
Event finished
```

11.2 Exemplary Solution

Here the exemplary solution:

```
#include <fstream.h>
#include <iostream.h>
```

```
#include "seventhgraphics.h"

Cevent::Cevent(int t=-1)
{
  type=t;
}

Cevent::~Cevent()
{
}

bool Cevent::isDrawEvent()
{
  return type==0;
}

Chandler::Chandler()
{
  succ=NULL;
}

Chandler::~Chandler()
{
  delete succ;
}

void Chandler::handle(Cevent e)
{
  if (succ!=NULL)
  {
    succ->handle(e);
  } else
  {
    cout<<"Event finished\n";
  }
}

void Chandler::setsuccessor(Chandler* h)
{
  succ=h;
}

/// creates point (x,y)
```

```
Cpoint::Cpoint(int x, int y)
{
  x_coordinate=x;
  y_coordinate=y;
}

/// nothing todo for this class
Cpoint::~Cpoint()
{
}

/// writes to cout: (x,y)
void Cpoint::draw()
{
  cout<<"("<<x_coordinate<<","<<y_coordinate<<")";
}

 /// creates rectangle
 /// (x1,y1)-
 /// | (x2,y2)
Crectangle::Crectangle(int x1, int y1, int x2, int y2)
{
  start_point=Cpoint(x1,y1);
  end_point=Cpoint(x2,y2);
}

/// nothing todo for this class
Crectangle::~Crectangle()
{
}

/// writes to cout:
/// (x1,y1)-
/// | (x2,y2)
void Crectangle::handle(Cevent e)
{
  if (e.isDrawEvent())
  {
     start_point.draw();
     cout<<"-\n";
     cout<<"| ";
     end_point.draw();
     cout<<"\n";
```

```
  }

  Chandler::handle(e);
}

Cpicture::Cpicture()
{
}

/// nothing to do
Cpicture::~Cpicture()
{
}

void Cpicture::handle(Cevent e)
{
  Chandler::handle(e);
}

/// creates drawEvent
void Cpicture::draw()
{
  Cevent e=Cevent(0);
  cout<<"Drawevent created\n";
  handle(e);
}
```