Object-Oriented Programming

Lesson 1

Presentation of the course

Programming paradigms

Object-oriented programming

Some differences between C and C ++

Presentation of the course

Scientific area: Science and Technology of Programming

Curricula: Integrated Master Degree in Electronic Engineering and Telecommunications, Master Degree in Industrial Automation Engineering

Weekly load: 3 hours of classes

ECTS credits: 6

Code: 42531 (EET), 42521 (IAE)

The number of ECTS credit points assigned to a course does not tell you how many hours you will spend in class. It gives an indication of the total number of hours you are expected to spend on the course.

1 ECTS equals 25-30 hours of study. 6 ECTS equal 150-180 hours of study.

Work done during these hours includes: reading books, writing programs, studying for the exam and attending classes.

Evaluation

The evaluation of the course is of continuous type and incorporates several intermediate tests and assessment of performance in the classroom.

According to the Regulation of Studies of the University, at least 5 evaluation moments must occur during the semester.

Final grade = 0.1 × P1 + 0.2 × P2 + 0.2 × P3 + 0.3 × written test + 0.2 × classes

The student may choose to undertake evaluation through a final exam (this decision has to be communicated in written form before **October 1, 2014**). If the student wishes to waive his first choice he/she will have to communicate the change at least 48 hours before the first evaluation moment. All the students who do not exercise the option are automatically associated with the continuous type of assessment.

The classes are not compulsory.

Recommended bibliography

- Bjarne Stroustrup, The Programming: Principles and Practice Using C++, 2nd ed. Addison-Wesley Inc., 2014
- Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003
- Donald Knuth, The Art of Computer Programming: Volumes 1-4a, Addison Wesley, 2011
- Frank Carrano, Data Abstraction and Problem Solving with C++: Walls and Mirrors, 6th ed., Addison Wesley Inc., 2012
- Ivor Horton, Beginning Visual C++, Wrox Press, 2012

Course webpage

Summary
Announcements
Identification
Programme
Evaluation
Bibliography

Comments
Comments and suggestions are welcome by e-mail.

Today
September 14, 2014

Announcements

There will be no class on October 15, 2014. The replacement class will be given during the semester (the date and time will be announced later).

Identification

Course: Object-Oriented Programming (academic year 2014/2015, 1st semester)
Scientific area: Science and Technology of Programming
Curricula: Integrated Master Degree in Electronic Engineering and Telecommunications, Master Degree in Industrial Automation Engineering
Weekly load: 3 hours of classes

Programme

Objectives:

The goals of object-oriented programming (OOP) are to introduce the technology of object-oriented programming and to use this technology to develop programs in C++. The main objectives are:

- to introduce and discuss basic concepts of object-oriented programming, such as type abstraction, encapsulation, hierarchy, polymorphism, interface and implementation, etc.;
- to compare these properties with those used in other technologies, such as in procedural programming, in modular programming, etc.;
- to introduce tools used to develop programs based on this technology;
- to address and analyze in detail all new constructions of C++ (relative to C) and demonstrate how they support the object-oriented programming;
- to develop practical applications using object-oriented programming in the areas of computing, electronics, telecommunications and others that are related to the specialization of the students.

Programme

- Motivation: "why objects?", "what are new concepts?", "what advantages?"
- Abstract data types
- Introduction to encapsulation, inheritance, polymorphism, classes and objects, methods and data, interfaces and implementation
- Modeling with UML: basic UML components, class diagrams
- Key concepts in detail: classes and objects (encapsulation), inheritance and polymorphism
- New possibilities: data input and output, function overloading, values, pointers and references, default arguments, operators **new** and **delete**
- Classes and objects: attributes and methods; method name overloading, keywords **this**, **const** and **friend**, constructors and destructors, static data, static functions, static allocation and restricted visibility
- Inheritance: base and derived classes, virtual functions and dynamic polymorphism, virtual tables, upcasting, abstract classes, multiple inheritance, virtual base classes
- Operator overloading
- Templates
- Exception handling

Evaluation

Programming languages

All programming languages provide abstractions because complexity of the problems you are able to solve is directly related to the kind and quality of abstractions.

Assembly language is an abstraction of the underlying machine.

Imperative languages (e.g. Fortran, C) are abstractions of assembly language.

Other kinds of languages, such as Lisp or Prolog, instead of modeling the machine, try to model the problem to solve (as lists, in case of Lisp, or chains of decisions, for Prolog).

Each of these approaches are good solutions to the particular class of problem they were designed to solve, but are hardly suitable when you step outside of that domain.

The object-oriented approach provides tools for the programmer to represent elements (objects) in the problem space.

The representation is general enough that the programmer is not constrained to any particular type of problem.

Object-oriented programming

Object-oriented programming: describe the problem in terms of the problem, rather than in terms of the computer where the solution will run.

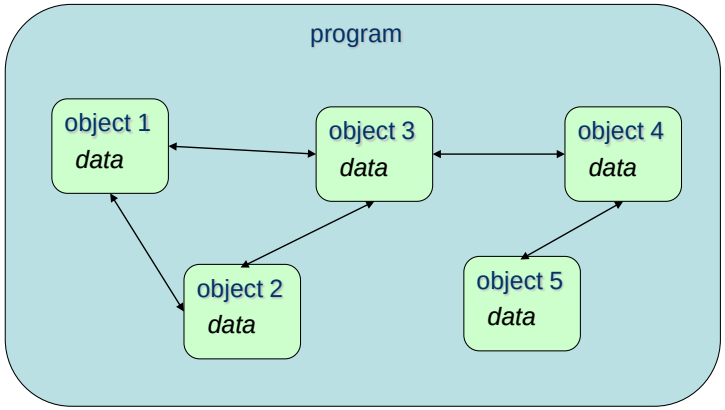
A program is a collection of objects telling each other what to do by sending messages.

Each object has its own memory made up of other objects.

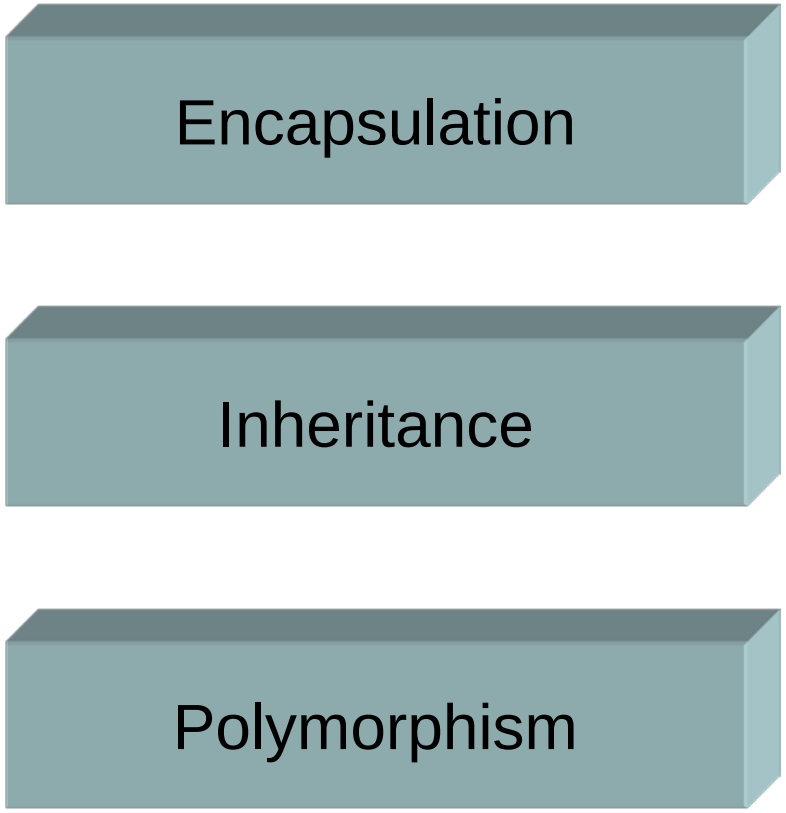
Each object has a type (class). A class defines data (attributes that describe the object) and methods (messages that the object can respond to).

Object is an instance of a class. You may create as many objects (instances of a class) as you like. Every object of a class has the same behaviour but keeps different data.

Each object is responsible for its own initialization and clean-up!



Key concepts



Encapsulation

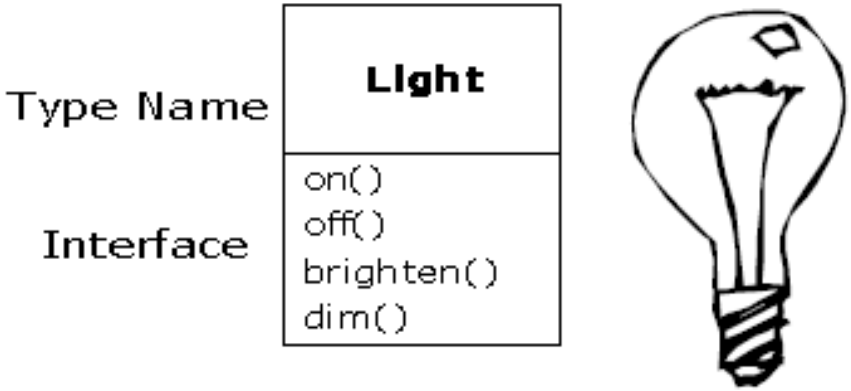
Encapsulation – the ability to package data with functions creating a new data type (class).

C++ allows a user to directly define types that behave in the same way as built-in types. Such a type is often called an abstract data type, or user-defined type.

The requests that you can make for an object are defined by its interface.

Code that satisfies requests comprises the implementation.

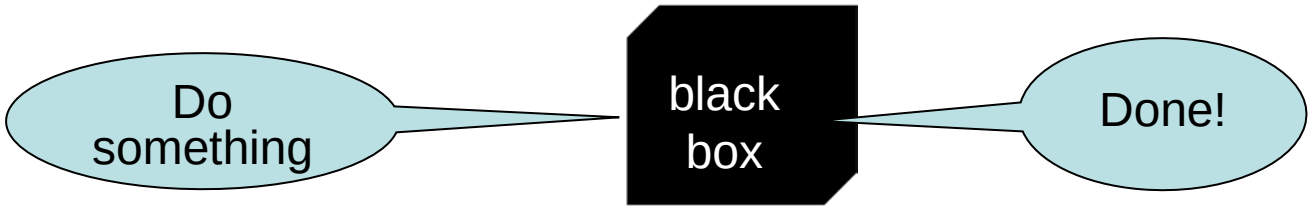
Unified Modeling Language (UML) – permits to represent graphically classes and their relationships in a program.



Encapsulation (cont.)

In C++ interface of an object is usually defined in a header file (**.h*) and the implementation – in the **.cpp* file.

To use an object you do not need to know has it is implemented.



object

This is a service to users because they can easily see what's important to them and what they can ignore. Besides, the designer may change the internal workings of the class without worrying about how it will affect the client programmer.

=> Hide as much information as you can with access control!

Encapsulation (cont.)

Example:

C

```
a = 1;
b = 2;
c = a + b;
```

Take *a*, with value 1, and *b*, with value 2, add them using built-in capability of *C* and assign the result to the variable *c*.

C++

```
a = 1;
b = 2;
c = a + b;
```

Send to the object *a*, with value 1, a message "+" with argument *b*, with value 2. The object *a* receives the message, executes the requested action, creates a new object, initializes it with the result and assigns the object to *c*.

What for such complexity?

To be able to process complex data!!!

Encapsulation (cont.)

Example:

C

```
char s1[] = “This does not ";
char s2[] = “work!";

s1 + s2;
```

Does not work because the C compiler only “knows” how to add numbers, while *s1* and *s2* are not numbers!

C++

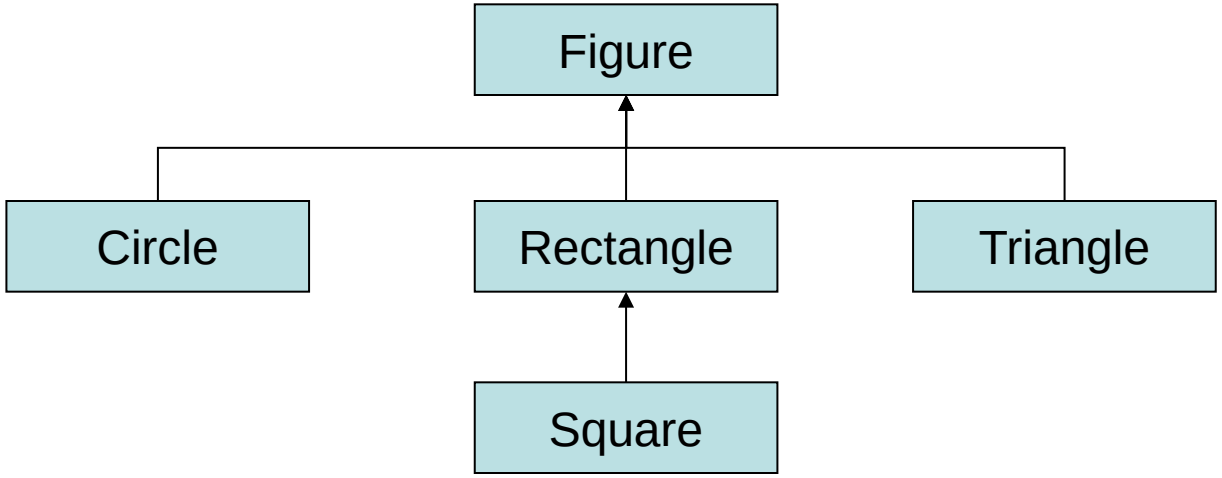
```
string s1 = “This does ";
string s2 = “work!";

s1 + s2;
```

Works because the class string includes a method for processing the message “+”!

“A well-designed user-defined type differs from a built-in type only in the way it is defined, not in the way it is used.”
(Bjarne Stroustrup, The C++ Programming Language, special edition, 2000, Addison-Wesley Inc., p.8)

Inheritance

Inheritance – possibility to take the existing class, clone it, and then make additions and modifications.



In UML diagrams arrows point from the derived classes to the base class.

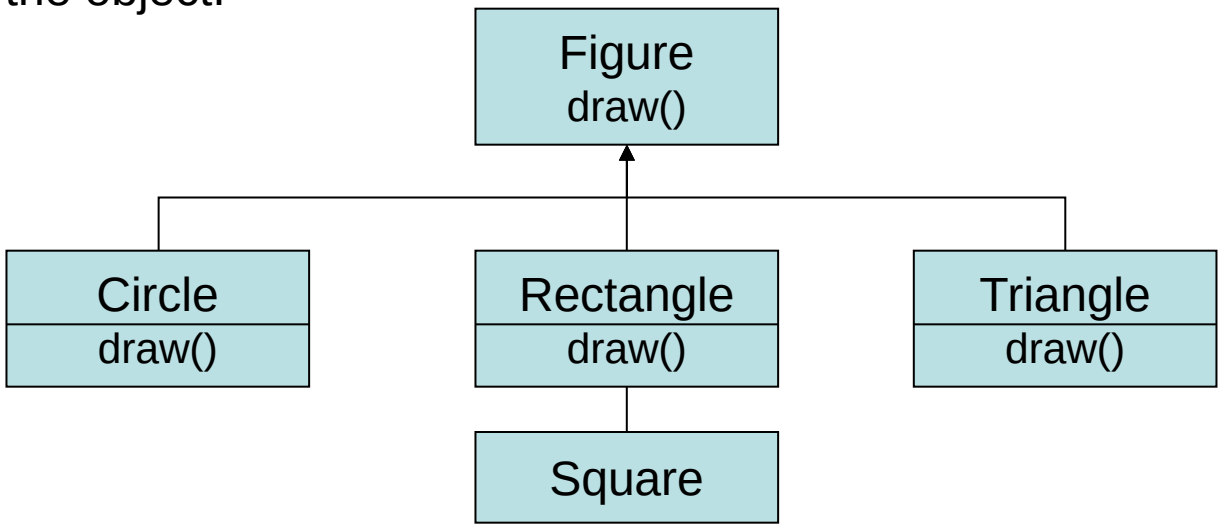
The original class is called the base or super or parent class.

The modified clone is called the derived or inherited or sub or child class.

Inheritance promotes code reutilization.

Polymorphism

A non-OOP compiler executes early binding. It means the compiler generates a call to a specific function name, and the linker resolves this call to the absolute address of the code to be executed.
In OOP, the concept of late binding is applied. When you send a message to an object, the code being called isn't determined until runtime. The compiler does ensure that the function exists and performs type checking on the arguments and return value, but it doesn't know the exact code to execute.
To perform late binding, the C++ compiler inserts a special bit of code in lieu of the absolute call. This code calculates the address of the function body, using information stored in the object.



Brief history of C++

1960s – Simula-67 – designed in the Norwegian Computing Center in Oslo *(classes, objects, inheritance).*

1970s – Algol 68 *(operator overloading*).

1970-1980s – Smalltalk – developed at Xerox PARC (*possibility to create and modify classes dynamically*).

1980s – Ada (*exception handling, generics*).

1980s – C – subset of C++.

1990s – C++ – designed by Bjarne Stroustrup at Bell Labs. C+ is a language in which new and different features are built on top of an existing syntax – the C language.

ISO/IEC 14882 – the first international standard appeared in 1998.

The standard was revised in 2003, 2007 e 2011 (C++11).

Eiffel, Delphi, Objective-C, Oberon, Actor, Object Pascal, Java, C#, etc...

Some differences between C and C++

Namespaces – are used to avoid name conflicts.

```
using namespace std;
```

Comments – end of line comment // is implemented in C++ (in addition to /*...*/).

```
// this is a comment
```

NULL pointers – in C++ all zero values are coded as 0.

New syntax for casts – C++ introduces new cast constructions to convert one type to another.

```
double d = 5.6;
int i = static_cast<int>(d);
```

Void parameter list – in C++ an empty parameter list is interpreted as the absence of any parameter.

```
void f();
void f(void);
```

Some differences between C and C++

Header files – for standard C++ libraries the file extension.h is not used. The libraries that have been inherited from C are used by prepending a “c” before the name.

```
#include <iostream>
#include <cmath>
```

Definition of local variables – in C++ local variables can be created at any position in the code.

```
for (int i = 0; i < 10; i++)
```

Function overloading – in C++ it is possible to define functions having identical names provided the function differ in their parameter lists.

```
double sum (double a, double b);
int sum (int a, int b);
```

Default function arguments – in C++ it is possible to provide default arguments when defining a function.

```
int func (int a = 3, int b = 5);
```

```
func(10, 10);
func(8);
func();
```

Some differences between C and C++

Typedef – is still allowed in C++ but no longer necessary when used as a prefix in union, struct or enum definitions.

```
struct my_struct
{       int a;
        double b;
};
my_struct s;
```

Structs with functions – in C++ it is allowed to define functions as part of a struct.

```
struct my_struct
{       int a;
        double b;
        double calc();
};
```

Type bool – C++ defines a new built-in data type which can have only two values: true or false.

```
bool flag = false;
```

Classes

Each object has a type.

Each object is an instance of a class.

Class = Type.

Definition of a class:

```
class CBook
{
public:
       //public data and methods

private:
        //private data and methods
};
```

Hiding the implementation

Class creator

Client programmer

Class creator must be able to change the hidden portion at will without worrying about the impact to anyone else.

Access specifiers – determine who can use the definitions that follow:

public – means the following definitions are available to everyone.

private – means that no one can access those definitions except you, the creator of the type, inside member functions of that type.

protected – will be explained later.

By defauld, all class members are private.

"Hiding the implementation reduces program bugs."
(Bruce Eckel, Thinking in C++, 2nd edition, 2000, MindView, Inc.)

Constructors

In C++, initialization is too important to leave to the client programmer.

The class designer can guarantee initialization of every object by providing a special function called the constructor.

If a class has a constructor, the compiler automatically calls that constructor at the point an object is created.

The class constructor has the same name as the name of the class. Like any function, the constructor can have arguments to allow you to specify how an object is created and give it initialization values.

Constructors return nothing.

Interface of a class

```
class CBook
{
public:
        CBook(std::string title, unsigned year);
        ~CBook(void);

        void Print();
private:
        unsigned m_uYear;
        std::string m_sTitle;
};
```

Implementation of a class

```
Cbook::CBook(std::string title, unsigned year)
{
	m_uYear = year;
	m_sTitle = title;
}

CBook::~CBook(void)
{
}

void CBook::Print()
{
	using namespace std;
	cout << "Title: " << m_sTitle << "; Year: " <<
		m_uYear << endl;
}
```

Destructors

Cleanup is as important as initialization and is therefore guaranteed with the destructor.

The syntax for the destructor is similar to that for the constructor: the class name is used for the name of the function, preceded by a leading tilde (~). In addition, the destructor never has any arguments.

The destructor is called automatically by the compiler when the object goes out of scope.

```
CBook::~CBook(void)
{
        using namespace std;
        cout << "Destructor" << endl;
}
```

Summary

- Everything is an object.
- Computations are performed by objects that exchange messages among themselves and perform actions.
- Each object has its own memory that consists of other objects.
- Each object is an instance of a class.
- A class defines the type and behaviour associated with objects.
- All objects of a class are constructed in the same manner and perform the same actions.
- Classes may be arranged in a class hierarchy in which derived classes inherit data and behaviour of the base classes.
- All classes have at least one constructor.
- Every class has a destructor.