Object Oriented Programming - Lecture 3

Diana Borza - diana.borza@ubbcluj.ro

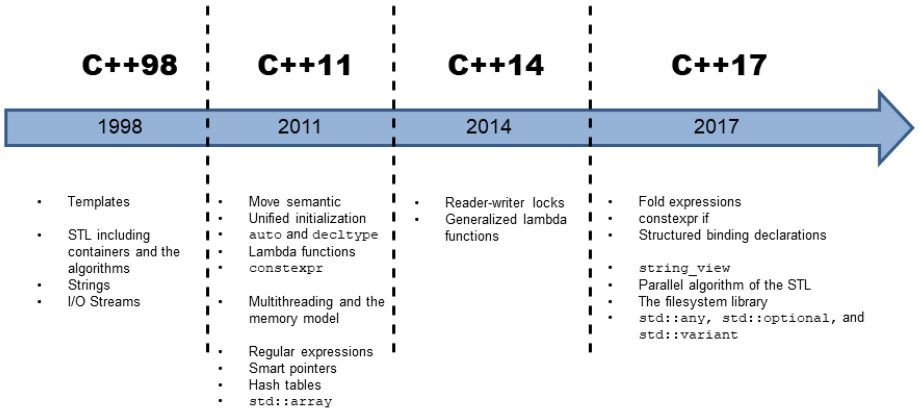
March 2021

Content

- What's new in C++?
- Programming paradigms
- OOP features
- OOP design rules
- Classes
- Constructors, destructor
- Static and friend elements

C++

- C++ was initially created by Bjarne Stroustrup as an extension to the C programming language: *C with classes*.
- C++ was first standardized in 1998. The C++ standard evolves: `https://isocpp.org/`. The current standard is C++20. It was approved on September 4, 2020 and published in ISO in December 2020. The next standard will be C++23.
- C programs are valid C++ programs.



What's new in C++?

- additional data types(bool, references)
- **classes** - object oriented programming
- namespaces
- templates, exceptions
- lambda expressions, placeholders
- type deduction
- smart pointers
- additional library facilities

C++ - References I

- A *reference variable* or *reference* is an **alias** (an alternate name) for a variable (for the same memory location).
- **Syntax:** *type_name &var_name*.
- A reference has the same memory address as the original variable.
- Parameter pass by reference: changes inside the functions are reflected after the function finishes.
- A const reference does not allow the modification of a variable.

References vs. pointers

References are similar to pointers, however there are the following notable differences:

- A reference must be initialized when it is declared. (On the other hand, pointers can be declared and not initialized or initialized with NULL or nullptr.)
- Once established to a variable, a reference cannot be changed to reference another variable. (A pointer can be made to point to a different variable than the one it was initialized with).
- There is no need to use dereferencing operator (*) or the address operator (&) with references.

C++ - Uniform initialization

- **Uniform initialization** (since C++11) - allows the usage of a consistent syntax to initialize variables and objects ranging from primitive type to aggregates.
- *syntax*
  type var_name{arg1, arg2, ...arg n}
- examples:
  - int x{}; // uninitialized integer value
  - float f{3.14f}; // initialized floating point number
  - int a[]{1, 2, 3, 4}; // aggregate initialization
  - int *ip = new int{3}; // initialized a dynamically allocated integer
  - Foo x1{}; // default constructor

Object oriented programming

Programming paradigms

1. *Imperative programming* - the programmer instructs the machine how to change its state
   - Procedural programming: instructions are grouped together into procedures
   - Object oriented programming (OOP): "objects" - *fields* + *methods*
2. *Declarative programming* - the programmer declares properties of the desired result, without describing its control flow
   - Logic programming
   - Functional programming

OOP

- Allows programmers to think in terms of the structure of the problem.
- The problem is decomposed into a set of objects.
- Object $\rightarrow$ a software component that incorporates both the attributes and the operations performed on the attributes.
- Objects interact with each other to solve the problem.
- The objects in the programming sense are designed to be closely related to the real world objects.

OOP - features

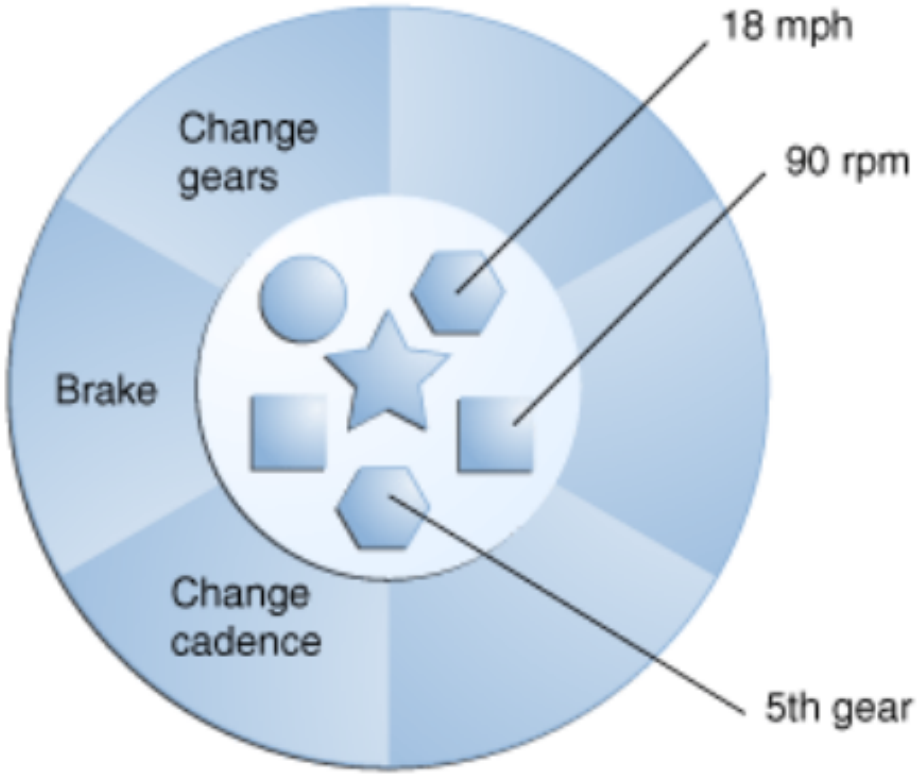
- **Abstraction**: separating an object's specification from its implementation.
- **Polymorphism**: allows an object to be one of several types, and determining at runtime how to "process" it, based on its type.
- **Inheritance**: organize classes to be arranged in a hierarchy that represents "IS A" relationships $\rightarrow$ easy re-use of the code, in addition to potentially mirroring real-world relationships in an intuitive way.
- **Encapsulation**: binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.

Real world objects

- All objects have a *state/characteristics* and a *behaviour/responsibilities* (what they can do)

Software objects

- Objects model tangible, processes or conceptual things
  - *capabilities*: exposed through methods (what they can do, how they behave)
  - *properties*: stored in fields (features that describe the objects)

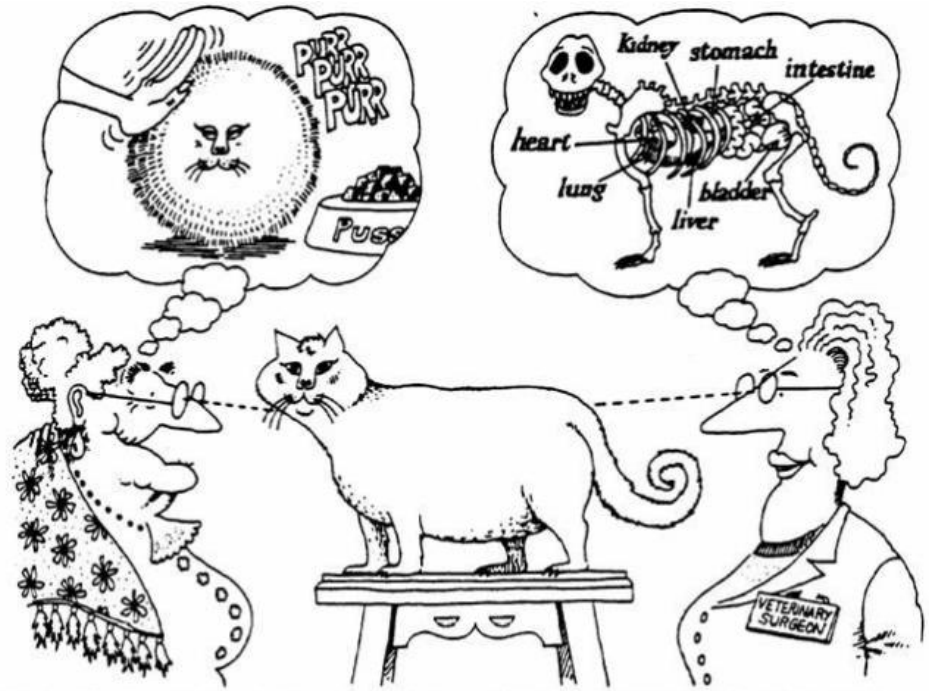


Software objects

- *Capabilities* allow objects to perform specific actions
  - **constructors**: establish initial state of object's properties;
  - **commands**: change object's properties;
  - **queries**: provide answers based on object's properties.
- *Properties* determine how an object acts
  - **attributes**: concepts that help describe an object;
  - **components**: things that are "part of" an object;
  - **associations**: things an object knows about, but are not parts of that object.

Abstraction

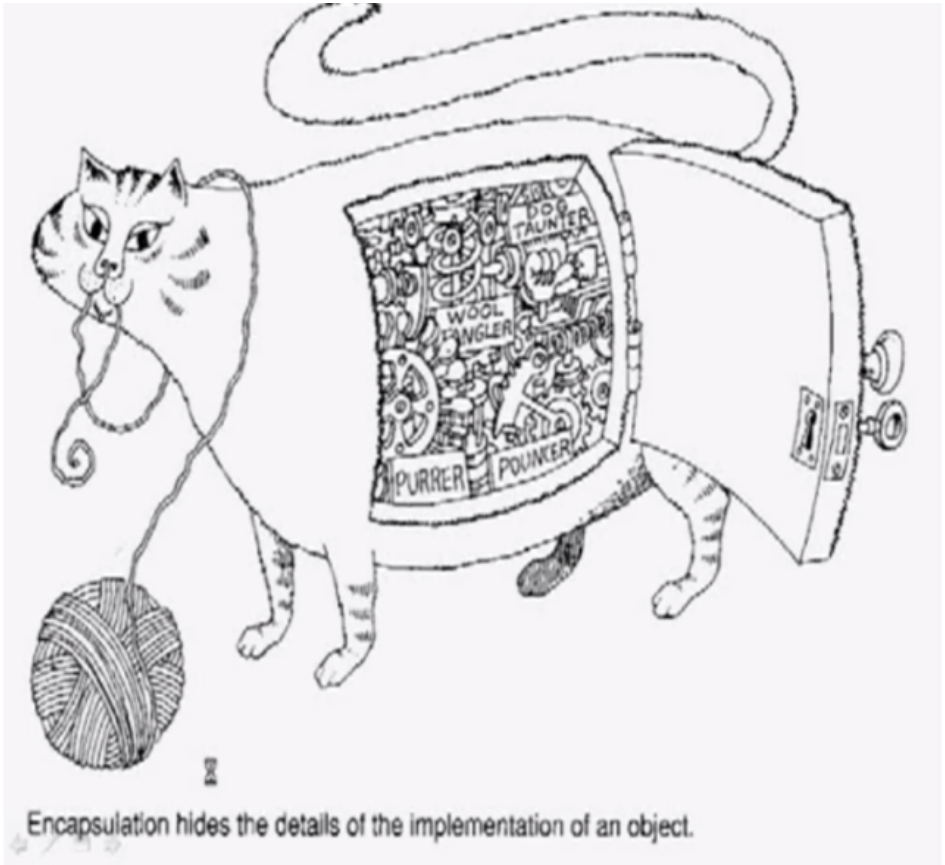
- purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure;
- filtering out/hide complexity from the user and show only relevant information;
- gives us the flexibility to change the implementation of the aspect/behaviour.



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Encapsulation

- *grouping* the data (object fields, attributes) with the methods that operate on that data and *restricting direct access* to some of an object's components;
- internal complexity, data and operation details are hidden;
- provides data security;
- you don't need to know how an object works to send messages to it, but you need to know what messages it can understand (what capabilities it has)



Encapsulation hides the details of the implementation of an object.

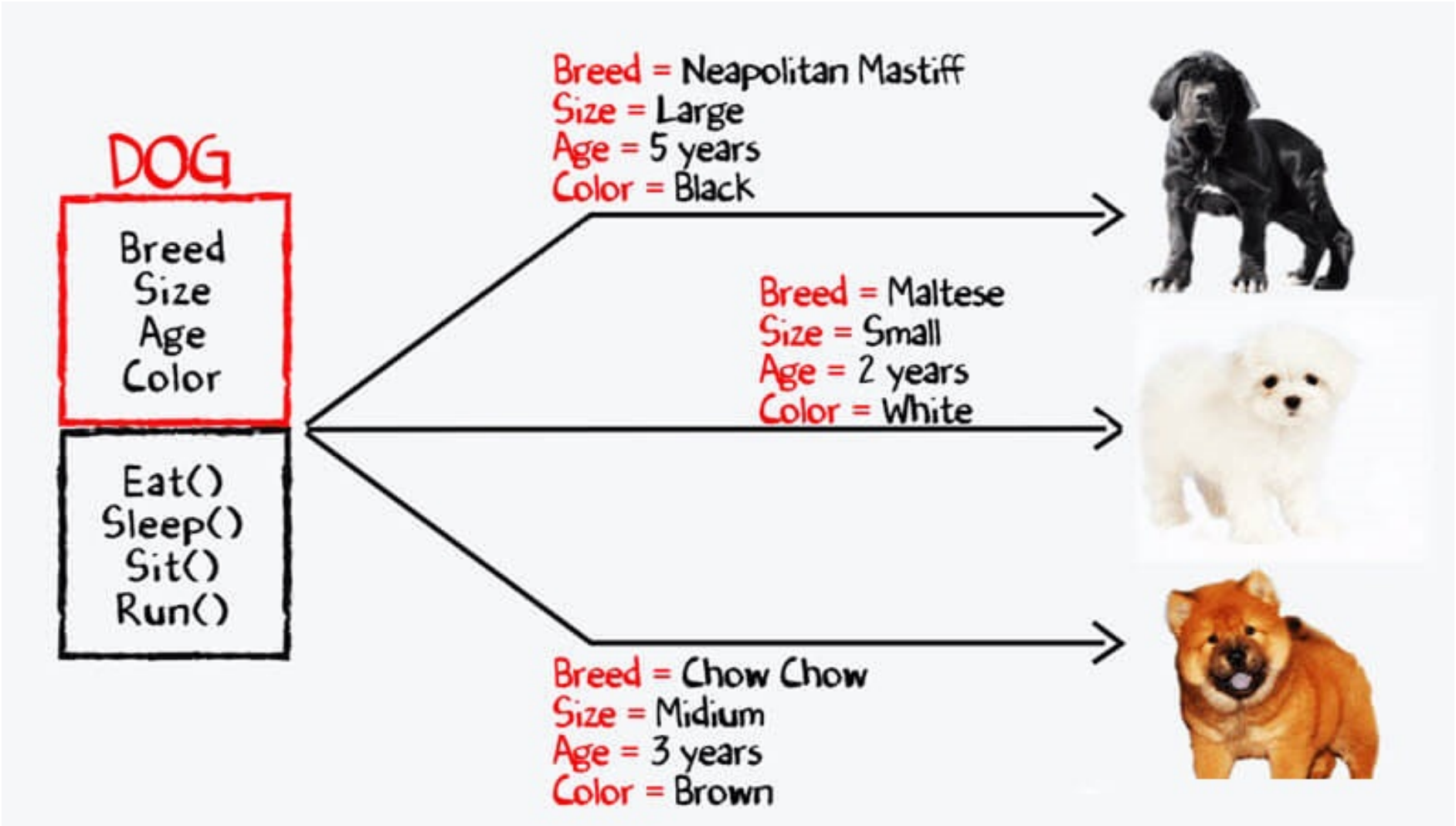
Classes - declaration vs definition

- **Declaration**
  - inside header files (.h, .hpp);
  - list of functions and fields;
  - like a "contract": includes functions that the class promises to its client.
- **Definition**
  - inside source files (.cpp);
  - actual implementation of the methods.

Classes and objects I

- **Classes** enable us to create new types.
  - is a user defined data type;
  - is a template/blueprint from which individual objects are created;
  - specifies what data and what functions will be included in objects of that type.
- **Objects**
  - made from class template (*instantiating* an object)
  - one class may represent an indefinite number of object instances.

Classes and objects II



Access modifiers

- Access modifiers define where the classes' fields and methods can be accessed from.
  1. **public** fields/methods can be accessed from any other object of any other class.
  2. **protected** fields/methods can only be accessed within the class or from child/derived classes.
  3. **private** fields/methods can only be accessed within the class (and from friend functions).
- The default access mode for classes if private.
- Minimize the public parts.
- **Getters** can be used to allow read-only access (from outside the class) to private fields.
- **Setters** can be used to modify private fields (from outside the class).

Access modifiers

- Access modifiers define *from where* we can access the members of a class (fields or methods).
  - public: public members can be accessed from anywhere.
  - private: private members can be accessed from within the class or from friend functions or classes.
  - protected: protected members can be accessed from within the derived classes; protected acts just like private, except that inheriting classes have access to protected members, but not to private members. Friend functions or classes can access protected members.

Access modifiers

Access	Public	Protected	Private
Class	YES	YES	YES
Derived class	YES	YES	NO
Client code	YES	NO	NO

Friend elements I

- A non-member function can access the private and protected members of a class if it is declared a friend of that class.
- Friend function: the declaration of this external function is placed within the class and it is preceded with the keyword friend.
- The friend keyword does not need to be used when defining the function.

this keyword

- this - a pointer to the current instance.
- The this pointer is implicitly passed to every method, to have a reference to the current instance.
- this is useful if there is a method parameter that has the same name as a class field.

Constructor I

- is a special function that is called automatically when an instance of a class is declared;
- does **not** return anything;
- must always have exactly the **same name** as the class;
- may have 0 or more parameters;
- a constructor with no parameters is called a *default constructor*;
- is generally public;
- it is impossible to create an object without a constructor being called $\rightarrow$ if you don't declare a constructor, an implicit constructor is automatically created.

Default constructor

- A **default constructor**
  - can be invoked with no arguments;
  - has no arguments or
  - defaults all its arguments.
- A class should have only one default constructor.
- The compiler automatically generates a default constructor if none is available.
- Defining *any* user defined constructor will prevent the compiler from implicitly declaring a default constructor.

Copy constructor I

- Copy constructors are invoked when a copy of the current object is needed:
  - when assigning one class instance to another;
  - when passing object as arguments (pass by value);
  - when returning a value from a function.
- The input parameter must be a (const) reference to an object of the same type.
- ? Why does it need to be a reference ?

Copy constructor II

- The compiler automatically generates a copy constructor if none is defined.
- The automatically generated copy constructor simply copies the contents of the original into the new object (byte by byte copy) $\rightarrow$ shallow copies for pointer variables.

Destructors

- The destructor is a special member function that is called when the lifetime of an object ends:
  - the object goes out of scope;
  - delete is called on a dynamically allocated object;
  - program termination - for objects with static storage duration
- The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.
- Syntax: ~class_name().
- A class can have a single destructor.

Constructor and destructor invocation

Constructors are invoked:

- when a new stack-allocated variable is declared;
- if we allocate instance using new (on the heap);
- when a copy of the instance is required (copy constructor):
  - assignment;
  - argument passing by value;
  - return an object from a function (by value).

The destructor is invoked:

- when delete is used to deallocate an instance allocated with delete;
- when an instance allocated on the stack goes out of scope.

Static data members

- The variables declared as static are characteristic to the class, they do not represent object state.
- They are "global" for all objects of the class, shared by all objects.
- The reference to the variable is performed using the class name and the **scope resolution operator** (**::**).

Static function members

- A static function member is characteristic to the class, does not depend on individual objects.
- It can be called even if no instances of the class exist.
- A static function can only access other static data members or functions, as well as functions outside the class.
- The static functions **DO NOT** have acces to the this pointer.
- Static functions are accessed using the class name and the **scope resolution operator** (**::**).

Operator overloading

- **function overloading** $\rightarrow$ two or more functions can have the same name but different parameters.
- `https://en.cppreference.com/w/cpp/language/operators`
- **Syntax**: Use the keyword operator followed by the symbol for the operator being defined.
- Like any other function definition, it must have parameters and a return type.

Operator overloading

- Operators that CANNOT be overloaded:

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Assignment operator overloading I

- The assignment operator is used to copy the values from one object to another *already existing object*.
- The compiler will generate an assignment operator, if none was defined.
  - Its default behaviour is member wise assignment.
  - It makes shallow copies.
- ? When is the copy contructor invoked and when is the assignment operator invoked ?

Rules for operator overloading

- Overloaded operators must either be a nonstatic class member function or a global function.
- The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked.
- Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
- Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.
- Overloaded operators cannot have default arguments.
- https://docs.microsoft.com/en-us/cpp/cpp/general-rules-for-operator-overloading?redirectedfrom=MSDN&view=vs-2019

Rule of three

"If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three."

`https://en.cppreference.com/w/cpp/language/rule_of_three`

OOP design guidelines

1. First, objects must be identified.
2. Objects' internals (*attributes*) and behaviour (*actions*) must be defined.
3. The manner in which the objects interact must be described (functions).
4. OOP includes and combines the advantages of modularity and reusability.

OOP design example I

Write a program that simulates the growth of virus population in humans over time. Each virus cell reproduces itself at some time interval. Patients may undergo drug treatment to inhibit the reproduction process, and clear the virus cells from their body. However, some of the cells are resistant to drugs and may survive.

- What are the objects?
- Characteristics?
- Responsibilities?

OOP design example II

Patient

- Characteristics
  - virus population
  - immunity to virus (%)
- Responsibilities
  - take medicine

Virus

- Characteristics
  - reproduction rate (%)
  - resistance (%)
- Responsibilities
  - reproduce
  - survive

OOP - strengths

- Development is easier: Objects generally have physical counterparts and this simplifies modeling (a key aspect of OOP)
- Promotes code reuse.
- Well-designed objects are independent units.
  - Everything that relates to the real-world object being modeled is in the object — *encapsulation*;
  - Communication is achieved by sending messages.
- Provides facilities associated to class hierarchies.

Summary I

- OOP - decomposes the problem into a set of objects (all having a state and certain capabilities);
- Objects interact with each other through messages to solve the problem;
- Which are the features of OOP?
  - Abstraction
  - Polymorphism
  - Inheritance
  - Encapsulation

Summary II

- A class has fields (data, attributes) and methods.
- Objects are initialized via constructors (default, with parameters, copy constructors).
- Objects are destoyed via destructors.
- C++ allows operator overloading.
- *Rule of three*: If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.

Homework

Write a simple class for rectangle type. The class should have the following:

- private fields *x*, *y* - the x an y coordinates of the top left corner
- private fields *width*, *height* - the width and height of the rectangle
- methods:
  - bool isEmpty() - checks if the rectangle is empty (i.e. width $= 0$, height $= 0$);
  - int area(); - computes the area of the rectangle
  - getter and setter methods for all its private fields
  - assignment operator overloading
  - destructor
  - overload the following operators:
    - ==: rect == rect1(rectangle comparison)
    - &: rect = rect1 **&** rect2 (rectangle intersection)