Classes. Objects. Important Methods. Operators.

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2024

Overview

C++ programming language I

- C++ was initially created by Bjarne Stroustrup and first standardized in 1998.
- The C++ standard evolves: https://isocpp.org/. The current standard is C++23. The committee started working on C++26.
- C programs are valid C++ programs.

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off". (Bjarne Stroustrup)

C++ programming language II

In addition to the facilities provided by C, C++ provides:

- additional data types (bool, reference);
- classes;
- templates;
- exceptions;
- namespaces;
- operator overloading;
- function overloading;
- free store management operators;
- additional library facilities.

I/O Library

- The C++ I/O operations are defined in the header <iostream>
- cin - corresponds to the standard input (stdin).
- cout - corresponds to the standard output (stdout).
- The writing operation is achieved using the *insertion operator* <<.
- The reading operation is achieved using the *extraction operator* >>.

References in C++ I

- A *reference variable* or *reference* is an *alias* or an *alternate name* for a variable (for the same memory location).
- They are particularly useful for function parameter passing by reference (changes inside the functions are reflected after the function finishes).
- A reference has the same memory address as the original variable.
- A const reference does not allow the modification of a variable.

References in C++ II

- 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.

DEMO

References. (*Lecture_3/c_plus.cpp*).

Variable initialization

- Initialize variables using the curly brackets {} (C++ uniform initialization): `int a{2};`
- The `auto` specifier specifies that the type of the variable that is being declared will be automatically deduced from its initializer.
- `auto` is useful for:
  - avoiding writing long typenames;
  - avoiding repetitions;
  - getting the correct type (and no implicit conversions).

DEMO

References. (*Lecture_3/c_plus.cpp*).

Namespaces

- Namespaces provide a method for grouping items together, for preventing name conflicts and for organizing code.
- A namespace is a declarative region that provides a scope to the identifiers inside it.
- A namespace can contain functions, variables, classes.
- The elements inside a namespace are accesible only by using:
  - the fully qualified name (including the **scope resolution operator ::**).
  - a using directive. This directive **should not** be used in header files!

DEMO

Namespaces. (*Lecture_3/c_plus.cpp*).

Ranged-based for loop

- Is a more readable equivalent for the *for loop*, for iterating a container.

DEMO

Ranged-based for loop. (*Lecture_3/c_plus.cpp*).

Classes

Figure source: https://devrant.com/rants/77251/c-vs-c

Object-oriented programming I

- Allows programmers to think in terms of the structure of the problem.
- The problem is decomposed into a set of objects.
- Objects interact with each other to solve the problem.
- New types of objects are created to model elements from the problem space.
- The objects in the programming sense are designed to be closely related to the real world objects.

Object-oriented programming II

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

Object-oriented programming III

Primary OOP features

- **Abstraction**: separating an object's *specification* from its *implementation*.
- **Encapsulation**: grouping related data and functions together as objects and defining an interface to those objects.
- **Inheritance**: allowing code to be reused between related types.
- **Polymorphism**: allowing an object to be one of several types, and determining at runtime how to "process" it, based on its type.

Real world objects

- they all have: a *state/characteristics* and a *behaviour/responsibilities* (what they can do).

Software objects

- are conceptually similar to real world objects;
- the *state* - is stored in *fields* (data/attributes);
- the *behaviour* - is exposed through *methods* (functions).

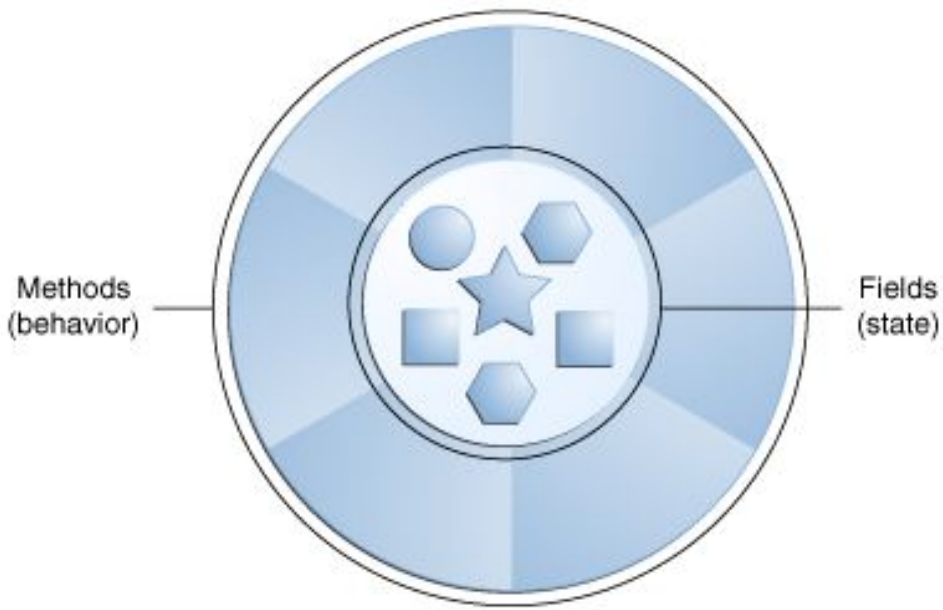


Figure source: Oracle: What is an object?

Example I

Write a program to simulate how a bacteria population grows inside a human host and how it spreads within a community, over time.
The bacterium multiplies with a certain growth rate inside the human body.
If the people undergo drug treatment, the bacterium growth is reduced.
Because of too many antibiotics, some bacteria mutate and become resistant to treatment.
The bacteria can spread by air, whenever two persons meet.
According to their level of immunity, some persons might be healed without drugs after a certain interval of time.

Example II

- What objects can be identified?
- What are their characteristics and responsibilities?

- **Bacterium**
  - *Characteristics*: growth rate, antibiotic resistance.
  - *Responsibilities*: reproduce, mutate.
- **Person**
  - *Characteristics*: bacteria population, immunity level.
  - *Responsibilities*: undergo treatment, meet other person.

Classes I

- Classes enable us to create new types.
- A class:
  - 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.

Classes II

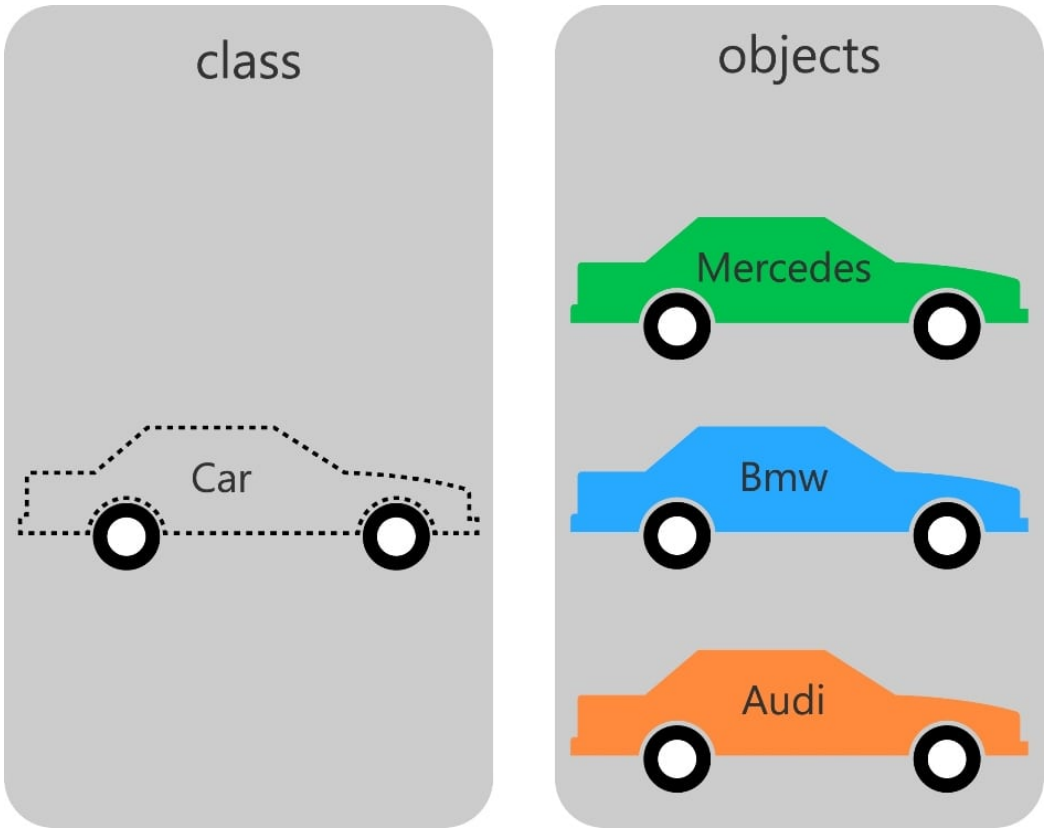


Figure source: http://www.expertphp.in/article/php-classes-and-object

Example - Vector in a plane (2D Vector)

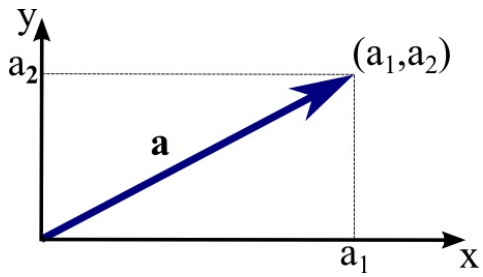


Figure source: http://mathinsight.org/vectors_cartesian_coordinates_2d_3d

- *Characteristics*: **x** and **y** coordinates/components of the 2D vector (data members).
- *Behaviour* (function members/methods):
  - 2D vectors can be added;
  - 2D vectors can be subtracted;
  - 2D vectors can be multiplied by a scalar value;
  - 2D vectors can be rotated;

Class declaration and definition

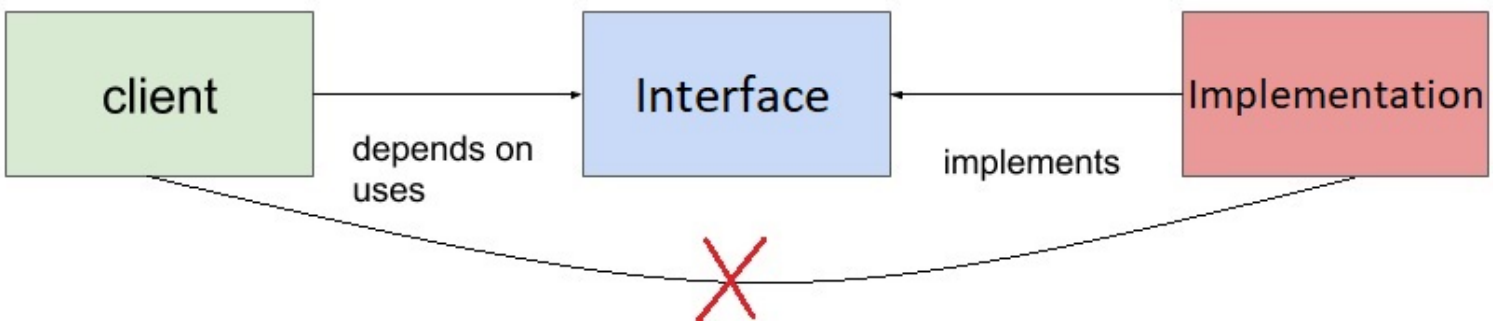


Figure inspired from: MIT OpenCourseWare: Introduction To C Memory Management And C++ Object-Oriented Programming

Class declaration

A class is declared in a header file: it will contain fields and function declarations:

```
class Vector2D
{
private:
    double xCoordinate;
    double yCoordinate;

public:
    /*
        Rotates the current 2D vector.
        Input: angle - real value
        Output: the current 2D vector is rotated with the given angle.
    */

    void rotate(double angle);

    /*
        Multiplies the current 2D vector with a scalar value.
        Input: scalarValue - real number
        Output: the current 2D vector is multiplied by the given value.
    */
    void multiplyByScalar(double scalarValue);

    // other methods
};
```

Method definition I

- In a separate cpp file we define the methods declared in the class.
- Include the header file containing the class declaration.
- Use the **scope resolution operator ::** to indicate that the method is part of the class.

```
#include "Vector2D.h"
#include <cmath>

void Vector2D::add(Vector2D v)
{
    xCoordinate += v.xCoordinate;
    yCoordinate += v.yCoordinate;
}

void Vector2D::rotate(double angle)
{
    xCoordinate = xCoordinate * cos(angle) - yCoordinate * sin(angle);
    yCoordinate = xCoordinate * sin(angle) + yCoordinate * cos(angle);
}
```

Method definition II

- Methods can also be defined in the class declaration (header file).
- These **must be** *inline* methods.
- The *inline* specifier usually serves as a hint for the compiler to perform optimisations (e.g. replace the function call with the actual code from the function).
- Inlining is best suited to short functions.

Access modifiers I

- Access modifiers define where the classes fields and methods can be accessed from.
- **public** fields/methods can be accessed from anywhere.
- **private** fields/methods can only be accessed within the class (and from friend functions).
- **protected** fields/methods can only be accessed within the class or from child/derived classes.
- The default access mode for classes if **private**.
- **?** Why control access to class members?

Access modifiers II

- Member fields should be kept private.
- The amount of public parts should be minimized.
- Getters can be used to allow read-only access (from outside the class) to private fields.

```
    double getXCoordinate() { return this->xCoordinate;  }
    double getYCoordinate() { return this->yCoordinate;  }
```

- Setters can be used to modify private fields (from outside the class).

```
void Vector2D::setXCoordinate(double newX)
{
    if (newX < 0)
        throw std::invalid_argument{"New value must be > 0!"};
    this->xCoordinate = newX;
}
```

The **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.
- **?** Why use $this \rightarrow xCoordinate$ instead of $this.xCoordinate$?

The use of **const**

- const can be used to indicate that an object should not be changed;

```
Vector2D(const Vector2D& v);
```

- the **const** restrictions are verified at compile time;
- const can be used in a method to indicate that it is not changing the state of the object; in this case, const is part of the function's signature.
- a *non-const* method cannot be called for a **const** object.

```
double getXCoordinate() const { return this->xCoordinate; }
double getYCoordinate() const { return this->yCoordinate; }
```

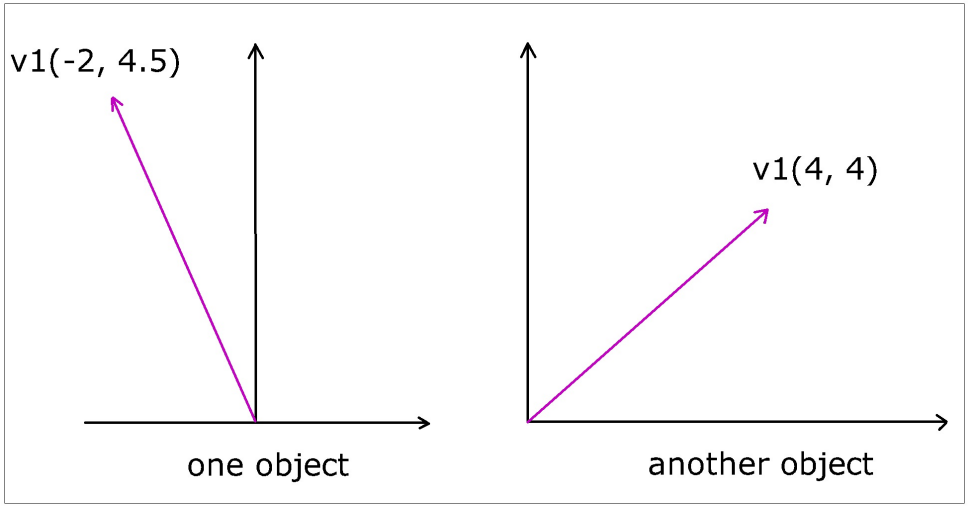
DEMO

Const methods. (*Lecture_3/vector_ex2.cpp*).

Object declaration and initialization I

- The template/blueprint (data type) for 2D vectors is created $\Rightarrow$ objects can be created with this template.
- An **object** is an *instance* of a class, a particular value of the defined type.
- Different instances can have different sets of values in their fields.

Object declaration and initialization II



Object declaration and initialization III

Object declaration

<class_name> <identifier>;

- Memory is allocated to store the object (store every attribute value).
- Object values should be initialized.

DEMO

Class creation and object initialization. (*Lecture_3/vector_ex2.cpp*).

Initialization - Constructors I

A constructor

- 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.

Initialization - Constructors II

- It is impossible to create an object without a constructor being called.
- A class must have at least one constructor function (if you don't declare one, an implicit constructor is automatically created).

Default constructors I

A default constructor

- can be invoked with no arguments;
- has no arguments or
- defaults all its arguments.

- Classes can have more than one default constructors, but should not (ambiguity).

DEMO

Default constructors. (*Lecture_3/vector_ex2.cpp*).

Default constructors II

- When making an array of objects, the default constructor is invoked on each element.
- 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.

Constructors with parameters

- A class can have multiple constructors (constructors can be overloaded), with different number of parameters and/or parameters of different types.

Member initialization

- insert a colon (:) before the constructor's body and then a list of initializations for class members:

```
Vector2D::Vector2D(double x, double y) : xCoordinate{x}, yCoordinate{y} {}
```

Copy constructors 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.

```
Vector2D(const Vector2D& v);
```

Copy constructors 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.
- **If the class has pointer variables and has some dynamic memory allocations, then one must explicitly create a copy constructor.** Why **?**

DEMO

Copy constructors. (*Lecture_3/vector_ex2.cpp*).

Destructors I

- A destructor is a special member function called when the class instance is deallocated:
  - if the instance was dynamically allocated (with new) - the destructor is called when delete is called.
  - if the instance was statically allocated - the destructor is called when it goes out of scope.
- The destructor must have the same name as the class, prefixed with tilde($\sim$).
- It does not return anything and does not have any parameters.

Destructors II

- **If the class has pointer variables and has some dynamic memory allocations, then one must explicitly create a destructor.**

DEMO

Destructors. (*Lecture_3/vector_ex2.cpp*).

Allocating and deallocating instances

- new - can be used to allocate a class instance in the free-store.
- delete - must be used for deallocation.

DEMO

Dynamic allocation and deallocation of objects. (*Lecture_3/vector_ex2.cpp*).

Constructors and destructors 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 new;
- when an instance allocated on the stack goes out of scope.

Operator overloading I

- The built-in operators available in C++ can be overloaded for user-defined types.
- Operator overloading makes the program easier to write, read and understand.
- It is just another way of calling a function.
- Almost all operators can be overloaded; see http://www.tutorialspoint.com/cplusplus/cpp_overloading.htm for the list of overloadable/non-overloadable operators.

Operator overloading II

Definition

- 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.

```
/*
    Overloading the + operator to add 2 2D vectors.
    Input: v - Vector2D
    Output: a 2D vector representing the sum of the current 2D vector
        and the parameter v.
*/
Vector2D operator+(const Vector2D& v);

/*
    Overloading the * operator to multiply a 2D vector with a scalar
        value.
    Input: scalarValue - double
    Output: a 2D vector representing the product of the current 2D
        vector and the given scalar value.
*/
Vector2D operator*(double scalarValue);
```

Operator overloading III

Using operator overloading

```
Vector2D v3 = v1 + v2;   // <=> Vector2D v3 = v1.operator+(v2);
Vector2D v4 = v1 * 3;    // <=> Vector2D v3 = v1.operator*(3);
```

? Will the following line work? Why/why not?

```
Vector2D v5 = 3 * v1;
```

DEMO

Operator overloading. (*Lecture_3/vector_ex3.cpp*).

Overloading the assignment operator (=) 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 memberwise assignment.
  - It makes shallow copies.
- **If the class has pointer variables and has some dynamic memory allocations, then one must explicitly create an assignment operator.**

Overloading the assignment operator (=) II

Return value of the assignment operator

- The return value cannot be void (chain assignment $a = b = c$ would then be impossible).
- It must return a reference to the object that called the operator function.

```
Vector2D& operator=(const Vector2D& v);
```

Copy constructor vs. assignment operator

```
Vector2D v1{ -1, 1 };
Vector2D v2{2, 3};
Vector2D v7 = v1;   // copy constructor is called (a new object is
    created and data is copied into it)
Vector2D v8;
v8 = v2;        // assignment operator is called (the object already
    exists, data is copied into it)
```

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.

Source: Microsoft: General Rules for 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." (http://en.cppreference.com/w/cpp/language/rule_of_three)

If a class is responsible to manage a resource (heap memory, file, database connection, etc) we need to define:

- copy constructor;
- assignment operator;
- destructor.

Static elements I

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 (::)**.

DEMO

Static elements. (*Lecture_3/vector_ex3.cpp*).

Static elements II

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 (::)**.

Friend elements I

- Friend functions are used when one wants to allow a function that is not a member of a class to access all private and protected members of the class.
- The prototype of the function must be placed inside the class, preceded by the keyword friend.

```
class Vector2D
{
// ...
public:
    // ...

    // friend function
    friend void printVectorData(const Vector2D& v);
};
```

Friend elements II

- A class can also be a friend of another class: the entire class and all its members are friend of the initial class.

```
class Vector2D
{
// ...
public:
    // ...

    // friend class
    friend class Graphics;
};

class Graphics
{
    // ...
};
```

DEMO

Friend elements. (*Lecture_3/vector_ex3.cpp*).

Summary I

Object Oriented Programming

- Decomposing the problem in a set of objects having characteristics and responsibilities and that interact.
- OOP features: abstraction, encapsulation, inheritance, polymorphism.

Classes and objects

- **Class** - template, **object** - instance.
- The class interface is exposed (header file), the class implementation is hidden (.cpp file).

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.