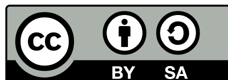
Object-Oriented Programming

The Art of Computer Programming

Pierre Senellart



20 October 2025

Outline

Overview of OOP

OOP: Concepts and Python/C++ Implementation

Structured Data in C

Object-Oriented Programming (OOP)

- Specific programming paradigm, along with (structured) imperative programming, functional programming, etc.
- An object is an instance of a well-defined class, which specifies its properties and methods: all objects of the same class have the same properties and methods
- Classes are organized into a hierarchy of super- and subclasses, the subclasses inheriting the properties and methods of the superclasses
- Classes allow one to abstract certain implementation details
- A class encapsulates the data and the functions manipulating that data, which allows certain data to be hidden from the user of the class
- We can use instances of objects belonging to different subclasses of the same class uniformly; they behave polymorphically

Pure and Impure OOP

- In purely object-oriented programming languages (Smalltalk, Eiffel, Scala), everything that is manipulated (in particular, all language values) is an object, with its class, etc.
- In more general-purpose languages supporting OOP, including Python and C++ (but also Java, JavaScript, etc.), objects are used together with primitive types, functions, etc.

Usefulness of OOP

OOP is particularly well suited to programming on complex data that come with their own properties and methods:

- real-world concepts (vehicle, factory, person, bank account, etc.)
- graphical interface elements (button, menu, etc.)
- mathematical concepts (set, relation, graph, etc.)
- algorithmic data structures (linked list, arrays, hash table, search tree, heap, stack, priority queue, etc.)

Python and OOP

- Python is a multi-paradigm language with support for OOP
- Python is not a pure object-oriented language
- Even compared with other general-purpose multi-paradigm languages, OOP in Python is not strongly emphasized and not very rich (little syntactic sugar)
- The fact that Python lacks strong typing makes it difficult to apply certain OOP concepts
- That said: OOP remains very useful (and widely used) in Python, and the Python standard library enriches the language with additional features dedicated to OOP

C and OOP

- C is not an object-oriented language, with no native support for classes, objects, inheritance, or polymorphism
- Using OOP in C requires manual design of objects and their behavior
- But because of the usefulness of OOP, large software projects that use C (e.g., Linux kernel, PostgreSQL) do develop their own way of implementing OOP
- Also common to find software projects mainly written in C, with some C++ code where OOP is the most useful

C++ and OOP

- C++ is a multi-paradigm language with strong support for OOP
- C++ is not a pure object-oriented language
- Provides both low-level control (like C) and high-level abstractions
- Good language to apply OOP concepts rigorously while paying attention to memory and performance implications
- Predates and a big source of inspiration for other languages with focus on OOP (e.g., Java, JavaScript, C#)
- But also a very complex programming language as a whole – we will not cover every aspect of OOP in C++

Outline

Overview of OOP

OOP: Concepts and Python/C++ Implementation

Structured Data in C

Classes and Objects: Concepts

- A class is the computer description of a type of object
- An object is defined as an instance of a class
- A class indicates in particular:
  - the properties (or fields, or attributes) of objects of this class: values that will be associated with each object of the class
  - the methods of objects of this class: functions that can be applied to the objects of the class manipulating the properties
- A class can also have class (or static) properties or methods: values or functions associated with the entire class, shared by all objects of the class and not applied to a particular object (less useful and less important)

Classes and Objects: Example

Example

Matrix can be a class describing the mathematical concept of a matrix. An object of type Matrix, also called an instance of the class Matrix, is the computer representation of a particular matrix.

A matrix may for example have the following properties:

- **n1** first dimension (a natural number)
- **n2** second dimension (a natural number)
- **rows** the rows of the matrix (each an array of numbers)

A matrix may for example have the following methods:

- **copy** to produce a copy of the matrix
- **add** to compute the result of adding the matrix to another matrix of the same dimension

Classes and Objects: In Python

- A class is defined as follows:

Python
```
class MyClass:
    ...
```

- Inside the class (indented), we put:
  - function definitions: the methods and static methods of the class
  - variable definitions: the static properties of the class
- Methods take an extra first argument, conventionally called `self`, which refers to the object to which the method is applied
- The properties of objects of the class are not explicitly declared, but the property `foobar` is automatically created in a method by assigning a value to `self.foobar`
- Static methods and properties are referred to outside of the class by prefixing them with `MyClass.`

Classes and Objects: Python Example

Python

```python
class Matrix:
    # dimensions is a method
    # _n1 and _n2 are two properties
    def dimensions(self):
        return [self._n1, self._n2]
    # copy is a method
    def copy(self):
        from copy import deepcopy
        return deepcopy(self)

    # sum is a static method
    def sum(a, b): return a+b
    # description is a static property
    description = "Matrix"

print(Matrix.sum(35, 7))
print(Matrix.description)
```

Classes and Objects: In C++

- A class is defined as follows:

C++

```cpp
class MyClass {
public:
  int property;
  void method() { ... }
};
```

- Inside the class, we define:
  - methods (member functions) – and static methods using the `static` keyword
  - properties (member variables) – and static properties using the `static` keyword
- Access modifiers (`public`, `protected`, `private`) control visibility – see later
- Static methods and properties are referred to outside of the class by prefixing them with `MyClass::`

Classes and Objects: C++ Example (1/2)

C++

```
#include <vector>
#include <string>
#include <iostream>
using namespace std;

class Matrix {
public:
  unsigned n1, n2;
  vector<vector<double> > rows;

  Matrix copy() const
  {
    return *this;
  }
```

Classes and Objects: C++ Example (2/2)

C++

```cpp
  static int sum(int a, int b) {
    return a+b;
  }

  static string description;
};

string Matrix::description = "Hello";

int main() {
  cout << Matrix::sum(35, 7) << endl;
  cout << Matrix::description << endl;
  return 0;
}
```

Objects and Methods: this and const in C++

- Inside a non-static method, the pointer `this` points to the object the method was called on (which can therefore be referred to as `*this`)
- `p->x` is a shortcut for `(*p).x`: accessing the property `x` of object `p`

C++
```
void setValue(int x) { this->value = x; }
```

- Within a method, `this->x` can be abbreviated to `x` (unless there is another local variable `x`)
- Methods that do not modify the object should be marked `const`, meaning that `this` is a pointer to a constant value:

C++
```
int getValue() const { return value; }
```

  - Ensures the method cannot change any non-mutable member
  - Allows calling the method on const objects

Declaration vs Definition in C and C++

- In C, functions and global variables are sometimes declared before being defined (by specifying the name and type, followed by a semi-colon, without a value or a function body, and sometimes the keyword `extern`)
- In C++, this extends to classes, methods (static and non-static), and static properties: they can be declared before they are defined
- On the other hand, local variables, static variables, and non-static properties are always defined at the same time they are declared
- Declaration introduces the name and type, but does not fully define its value or implementation
- Definition allocates memory (for variables and static properties) or provides the implementation (for functions and classes)
- Use: A variable, function, etc., must be declared before being used – but definition can be given later
- Static member variables must be defined outside the class
- Methods (non-static and static) can be defined outside the class (typically, when they become too long)

Constructor and Destructor: Concepts

- When we construct an object of a class (we say we instantiate the class), we can use a constructor to specify the initial values of the object's properties
- When an object is destroyed (because it is no longer used), we can use a destructor to specify operations to perform at the end of the object's life

Constructor and Destructor: In Python

- The constructor is a special method named `__init__`. It can take an arbitrary number of arguments, which will be used to construct the object.
- To construct an object, use the syntax `object = Class()`. Any constructor parameters go inside the parentheses.
- The destructor is a special method named `__del__` (sometimes called a *finalizer*). But it is rarely useful in Python because there is no guarantee when it will be called (depends on memory management)

Constructor and Destructor: Python Example

Python

```
class Matrix:
    # Constructor
    # Initialize properties _n1, _n2, and _rows
    # with constructor arguments
    def __init__(self, n1, n2, value=0):
        self._n1 = n1
        self._n2 = n2
        self._rows = [[value]*n2 for i in range(n1)]

    # Destructor (or finalizer)
    def __del__(self):
        print(f"Matrix {self._n1}×{self._n2} destroyed")

# Create a 5x5 zero matrix
m = Matrix(5, 5)
# Destructor may be called at the end of the program
```

Constructor and Destructor: In C++

- The constructor is a special method with the same name as the class and no return type. It can take an arbitrary number of arguments, which will be used to construct the object.

C++
```cpp
Matrix(unsigned n1, unsigned n2); // constructor declaration
```

- To construct an object, use the syntax `Class object();` for a value on the stack, `Class *p = new Class();` for a value on the heap, or `Class()` for a temporary value. Any constructor parameters go inside the parentheses.
- The destructor is a special method with the name of the class prefixed with ~ and no return type; should be declared `virtual` if the class may have derived classes

C++
```cpp
virtual ~Matrix(); // destructor declaration
```

- If no constructor is provided, a default one without argument is used

Constructor and Destructor: C++ Example

C++

```cpp
Matrix(unsigned n1, unsigned n2, double value=0.)
{
  this->n1 = n1;
  this->n2 = n2;
  rows = vector<vector<double> >(n1);
  for(unsigned i=0; i<n1; ++i) {
    rows[i] = vector<double>(n2, value);
  }

virtual ~Matrix() {
  cout << "Matrix " << n1 << "x" << n2 << " destroyed" << endl;
}
```

C++

```cpp
Matrix m(5,5);
```

Copy Constructors in C++

- A copy constructor is a special constructor in C++ that initializes a new object as a copy of an existing object.
- Syntax:

C++
```cpp
ClassName(const ClassName& other);
```

- When a copy constructor is called:
  - Passing an object by value to a function
  - Returning an object by value from a function
  - Explicitly creating a copy: `ClassName b(a);`
- If no constructor is defined, C++ provides a default copy constructor
- Default copy copies member values directly; problematic for dynamic memory

Copy Constructors in C++: Example (1/2)

C++

```
#include <iostream>
#include <cstring>
class String {
  char* data;
public:
  String(const char* s) {
    data = new char[strlen(s)+1];
    strcpy(data, s);
  }
  String(const String& other) { // Copy constructor
    data = new char[strlen(other.data)+1];
    strcpy(data, other.data);
  }
  ~String() { delete[] data; }
  void print() { std::cout << data << "\n"; }
};
```

Copy Constructors in C++: Example (2/2)

C++

```
int main() {
  String s1("Hello");
  String s2 = s1; // copy constructor called
  s2.print();
  return 0;
}
```

Specialization, Redefinition: Concepts

- In a class, one can redefine or specialize pre-existing methods or methods defined in a superclass (see later) to have appropriate behavior for the class's objects
- One can also redefine the behavior of programming language operators (+, *, **, <, ==, etc.) to use them with objects of the class

Specialization, Redefinition: In Python

- To specialize a method, simply redefine it in the class like any other method
- Python has magic methods automatically called in certain contexts; their name is always of the form `__method__`. We've already seen `__init__`, `__del__`, but there is also `__str__` called when an object is converted to a string (e.g., for printing)
- Operator behavior can also be redefined with magic methods: `__add__` for addition (left), `__rmul__` for multiplication (right), `__getitem__` for indexing (`[]`), etc.

Specialization, Redefinition: Python Example (1/2)

Python

```
class Matrix:
    def __getitem__(self, i):
        return self._rows[i]

    def __str__(self):
        result = ""
        for i in range(self._n1):
            for j in range(self._n2):
                result += str(self[i][j]) + " "
            if i != self._n1 - 1:
                result += "\n"
        return result

# Modify element 2,3 of matrix m
m[2][3] = 7
# Print matrix m
print(m)
```

Specialization, Redefinition: Python Example (2/2)

Python

```
class Matrix:
    def __rmul__(self, scalar):
        copy = self.copy()
        for i in range(self._n1):
            for j in range(self._n2):
                copy[i][j] *= scalar
        return copy

# Multiply matrix m by 3 and store in n
n = 3*m
# Print matrix n
print(n)
```

Specialization, Redefinition: In C++

- Redefine methods in a class or a derived class using the same signature; methods that are specialized in derived classes should be declared with the keyword `virtual` in base and derived classes, and `override` in derived classes (see further)
- Operators can be overloaded as methods or external functions with a name formed of `operator` followed by the operator:

C++

```cpp
class Matrix {
  ...
  // Adding two matrices
  Matrix operator+(const Matrix& other) const;
}

// Multiplying a scalar with a matrix
Matrix operator*(double scalar, const Matrix& m);
```

Specialization, Redefinition: C++ Example (1/2)

C++

```
class Matrix {
  ...
  auto &operator[](unsigned i)
  {
    return rows[i];
  }

  virtual void print() {
    for(unsigned i=0; i<n1; ++i) {
      for(unsigned j=0; j<n2; ++j)
        cout << rows[i][j] << " ";
      cout << endl;
    }
  }
};
```

Specialization, Redefinition: C++ Example (2/2)

C++

```
Matrix operator*(double scalar, Matrix m)
{
  for(unsigned i=0; i<m.n1; ++i)
    for(unsigned j=0; j<m.n2; ++j)
      m[i][j]*=scalar;
  return m;
}
```

C++

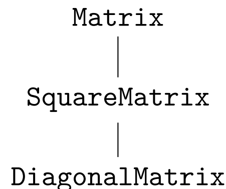
```
m[2][3] = 7;
m.print();
Matrix n = 3*m;
n.print();
```

Inheritance: Concepts

- A subclass (or derived class) of a class is a class describing a more specific type of object than the type described by the superclass (or base class)
- A subclass inherits the properties and methods of its superclass
- A subclass can specialize the methods of the superclass
- Multiple subclasses can inherit from the same superclass
- A subclass can inherit from multiple superclasses (but this makes inheritance more complex)

Inheritance: Concepts

- A subclass (or derived class) of a class is a class describing a more specific type of object than the type described by the superclass (or base class)
- A subclass inherits the properties and methods of its superclass
- A subclass can specialize the methods of the superclass
- Multiple subclasses can inherit from the same superclass
- A subclass can inherit from multiple superclasses (but this makes inheritance more complex)



Inheritance: In Python

- Specify base class(es) when defining the subclass:

Python
```
class MyClass(SuperClass1, SuperClass2):
```

- Specializing methods works the same way as redefining default methods or operators
- When redefining a method `my_method`, you can use `super().my_method` to call the method of the same name in the superclass; also works for magic methods. With multiple inheritance, it works but is more complex to determine which method is called.
- You can use `isinstance(my_object, MyClass)` to test if `my_object` has class `MyClass` or a subclass below MyClass in the hierarchy

Inheritance: Python Example (1/2)

Python

```
class SquareMatrix(Matrix):
    def __init__(self, n, value=0):
        self._n = n
        super().__init__(n, n, value)

a = SquareMatrix(5, 2)
print(3*a)
```

Inheritance: Python Example (2/2)

Python

```
class DiagonalMatrix(SquareMatrix):
    def __init__(self, n, value=0):
        self._n1 = n
        self._n2 = n
        self._n = n
        self._diagonal = [value] * n

    def __getitem__(self, i):
        return [0] * i + [self._diagonal[i]] \
                     + [0] * (self._n - i - 1)

    def __rmul__(self, scalar):
        copy = self.copy()
        for i in range(self._n):
            copy._diagonal[i] *= scalar
        return copy
```

Inheritance: In C++

- Subclass inherits from superclass using the `: public` syntax

C++
```cpp
class SquareMatrix : public Matrix {
};
```

- Methods of superclass can be overridden using `virtual` keyword in the base class and `override` keyword in the derived class
- The superclass constructor can be called using the following syntax:

C++
```cpp
DerivedClass(int n) : BaseClass(n, n) {
  ...
}
```

- Access to superclass methods can be done via `BaseClass::method()`

Inheritance: C++ Example

C++

```cpp
class SquareMatrix : public Matrix {
public:
  SquareMatrix(unsigned n, double value=0.) : Matrix(n, n, value) {}
};

int main() {
  SquareMatrix sm(5, 1.);
  (3*sm).print();
  return 0;
}
```

Encapsulation: Concepts

- Classes can be used to encapsulate the properties of an object and ways to access and manipulate these properties
- It can be useful to hide some properties or methods from the user of the class, to avoid exposing implementation details or incorrect usage
- For example, we don't want a user of the `Matrix` class to change the dimensions of a matrix without modifying the real number of rows/columns
- Three degrees of visibility for a property or method:

public	accessible without restriction from outside the class
protected	accessible only inside the class or its subclasses
private	accessible only inside the class

Encapsulation: In Python

- No real encapsulation enforcement in Python compared to other languages (C++, Java)
- But there is a convention to follow when naming methods and properties (except magic methods!):

public	name starting with a letter
protected	name starting with a single underscore (_)
private	name starting with two underscores (__)

- Same for static methods and properties
- Private names are truly hidden (actually renamed to `_Class__method`) outside the class; no protection for protected names, just a convention

Encapsulation: Additional Remark

- One way to enforce encapsulation is to forbid access to all properties and force access through special (accessor) methods for reading or modifying properties — common in languages like Java
- Useful if constraints must be checked before allowing modifications
- In Python, one can also do this (by declaring properties as private or protected and defining public methods) — but not always appropriate
- Can also be done almost automatically with the `@property` decorator

Encapsulation: Python Example

Python

```python
class Matrix:
    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, new_color):
        if new_color == 'pink':
            new_color = 'red'
        self._color = new_color

m = Matrix()
m.color = 'pink'
print(m.color)
```

Encapsulation: In C++

- Use public, protected, private to control visibility (default is private)

C++

```
class Matrix {
protected:
    unsigned n1, n2; // hidden
public:
    void setSize(unsigned n1, unsigned n2);
    unsigned getN1() const;
};
```

- Ensures safe access to object properties
- Usually properties are protected: can be accessed in the class or its derived class, but not outside of the class

Abstraction: Concepts

- Sometimes it is useful to define abstract classes: classes that we do not want an object to instantiate, but can have concrete subclasses
- Abstract classes can have abstract methods: methods without content (or with default content), which cannot be directly called, and must be specialized in concrete subclasses
- Useful to specify interfaces: functionalities that derived classes should implement without specifying how

Abstraction: In Python

- No built-in support for abstract classes, but support exists via the `abc` module (*abstract base class*)
- After doing

Python
```python
from abc import ABC, abstractmethod
```

define the base class of the abstract class as `ABC`, and decorate abstract methods with `@abstractmethod`

Abstraction: Python Example

Python

```python
from abc import ABC, abstractmethod

class MathematicalObject(ABC):
    @abstractmethod
    def size(self):
        pass

class Matrix(MathematicalObject):
    def size(self):
        return self._n1*self._n2

class Set(MathematicalObject):
    def size(self):
        return self._nb_elements

    ...
```

Abstraction: C++

- Abstract classes: declare at least one pure virtual method by specifying `= 0;` in lieu of a definition

C++

```cpp
class Shape {
public:
  virtual double area() const = 0; // pure virtual
};
```

- Derived classes must implement the pure virtual methods or stay themselves abstract

Polymorphism: Concepts

- Objects instantiating a class derived from `MyClass`, even if abstract, can be treated as instances of `MyClass` (e.g., by calling `MyClass` methods redefined in derived classes)
- Depending on the actual class of the object, behavior may differ: we call this polymorphic behavior
- For example, any square matrix has a determinant we can compute; if the matrix is diagonal (or triangular), its determinant is much easier to compute

Polymorphism: In Python

- In statically-typed languages (like Java), polymorphism means one can manipulate an object of type `DerivedClass` as if it were of type `BaseClass`, but its behavior will be that of `DerivedClass`
- Python being a dynamically typed language, an object is always treated as its type — but one can, for example, pass it to a function that does not care about its exact type, as long as it can be used like an object of another type (even beyond inheritance hierarchies, *duck typing*)

Polymorphism: Python Example (1/2)

Python

```
class SquareMatrix(Matrix):
    def determinant(self):
        if self._n == 2 and len(self[0]) == 2:
            val = self[0][0] * self[1][1] - self[1][0] * self[0][1]
            return val
        total = 0
        for c in range(self._n):
            copy = self.copy()
            copy._n -= 1
            copy._n1 -= 1
            copy._n2 -= 1
            copy._rows = copy._rows[1:]
            for i in range(self._n - 1):
                copy._rows[i] = copy[i][0:c] + copy[i][c+1:]
            sign = (-1) ** (c % 2)
            sub_determinant = copy.determinant()
            total += sign * self[0][c] * sub_determinant
        return total
```

Polymorphism: Python Example (2/2)

Python

```
class DiagonalMatrix(SquareMatrix):
    def determinant(self):
        from functools import reduce
        from operator import mul
        return reduce(mul, self._diagonal, 1)

a = SquareMatrix(5,1)
b = DiagonalMatrix(5, 2)
print(a.determinant())
print(b.determinant())
```

Polymorphism: C++

- Virtual functions enable runtime polymorphism

C++

```
Matrix* m = new SquareMatrix(5);
m->print(); // calls SquareMatrix::print if overridden
```

- Behavior depends on actual object type, not of declared type!

Outline

Overview of OOP

OOP: Concepts and Python/C++ Implementation

Structured Data in C

Structured Data in C: struct (1/2)

- In C, a `struct` is used to group related data together, forming a structured type
- A `struct` is similar to a `class` in C++, but with important differences:
  - All members are public
  - No static members
  - No methods (functions) are associated directly with a `struct`
  - No inheritance or polymorphism
  - No constructors or destructors (manual initialization and cleanup are required)

C

```c
struct Point {
  double x;
  double y;
};

struct Point p1 = {10, 20};
p1.x = 15;
```

Structured Data in C: struct (2/2)

- To emulate methods, functions take a pointer to the `struct`

C

```
void move_point(struct Point* p, double dx, double dy) {
  p->x += dx;
  p->y += dy;
}
```

- In C++, a `struct` is identical to a `class` except that all members are public by default

Licensing

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.