



Object-Oriented Programming

The Art of Computer Programming

Pierre Senellart



12 October 2026



Outline

- + **Overview**
- + **Structures in C**
- + **Classes**
- + **Construction**
- + **Encapsulation**
- + **Operators**
- + **Inheritance**
- + **Polymorphism**
- + **Coda**

Overview



Object-oriented programming

- ★ A **programming paradigm**, along with (structured) imperative programming (lecture 2) and functional programming (lecture 12)
- ★ An **object** bundles **data** and the **operations** on that data into a single value of the language
- ★ Every object is an **instance** of a **class**, which describes, once and for all, the **properties** (the data) and the **methods** (the operations) that all objects of that class have
- ★ Classes are organized in a **hierarchy**: a class can **inherit** the properties and methods of another one, and **specialize** them
- ★ **Compare with what we have done so far**: data in one place (variables, arrays, collections) and operations in another (functions applied to that data)

The four ideas of object-oriented programming

Encapsulation the data of an object and the operations on it are **grouped together**, and part of them can be **hidden** from the rest of the program

Abstraction a class describes **what** its objects can do before, and independently of, **how** they do it

Inheritance a more specific class **reuses** the properties and methods of a more general one instead of repeating them

Polymorphism objects of different classes of the same hierarchy can be used **uniformly**, each behaving according to its own class

These four ideas run through this lecture; they are **design ideas**, not language features, and each language offers its own way (or several) of applying them

What objects are good for

Programming with objects pays off when the program manipulates **complex data that come with their own operations**:

- + real-world concepts: a bank account, a vehicle, a person, an invoice
- + elements of a graphical interface: a window, a button, a menu
- + mathematical concepts: a matrix, a polynomial, a graph, a set
- + **data structures**: a dynamic array (lecture 4), a linked list, a hash table, a search tree, a priority queue (lecture 6)
- + resources that must be **released**: a memory block (lecture 4), a file (lecture 10), a network connection

Not a universal answer: a short script, a numerical computation, or a function such as `binary_search` gains nothing from being written with classes – and a class with no data and one method is just a function in disguise

Pure and impure object-oriented programming

- ✦ In a **purely** object-oriented language (Smalltalk, Eiffel, Ruby), **everything** manipulated by the program is an object, including numbers and functions, and every operation is a method call
- ✦ In a **multi-paradigm** language (Python, C++, but also Java, JavaScript, OCaml...), objects live **together** with primitive types, plain functions, and other paradigms: the programmer chooses where objects are useful
- ✦ Some languages have **no** support for objects (C), which does not prevent applying the ideas above by hand (see further)

Python and objects

- ✦ Python is a multi-paradigm language with support for objects; it is not a pure object-oriented language, but **every value is an object** (lecture 4: even an `int` is an object on the heap, with a class)
- ✦ Consequences we have already used without saying so: `"abc".upper()`, `t.sort()`, `d.keys()` are **method calls** on objects of class `str`, `list`, `dict`
- ✦ `type(x)` gives the class of a value, and classes are themselves values:
`type(3)` is `int` is `True`
- ✦ Little syntax dedicated to objects, few guarantees enforced by the language (**no real hiding** of data, no static type checking): much rests on **conventions**

C, C++, and objects

- ★ C has **no support** for objects: no classes, no methods, no inheritance; it has **struct** to group data (next section), and the programmer applies the ideas of object-oriented programming **by hand** where they help
- ★ C++ was designed as “C with classes” and has **strong support** for objects: visibility enforced by the compiler, constructors and destructors, inheritance, virtual methods, templates (lecture 12)
- ★ It gives **control** over memory and time together with these abstractions, at the price of a **large** and intricate language: we cover the part that a first-year course needs

Structures in C



Grouping data

- + Lecture 4: a dynamic array is described by **three values** that must stay consistent
 - a pointer to the block, the number of elements in use, the capacity of the block
- + Keeping them in three separate variables is fragile: three parameters to pass to every function, three values to keep in step, and nothing says they belong together

```
void push_back(int **data, size_t *size, size_t *capacity, int x);
```

C

- + Same need for any **record**: a point (two coordinates), a date (three numbers), an employee (a name, an identifier, a salary), a node of a linked list (a value, a pointer to the next node, lecture 6)
- + C answers with the **struct**: a **new type** whose values are made of several named **fields** (or **members**)

Structures in C

C

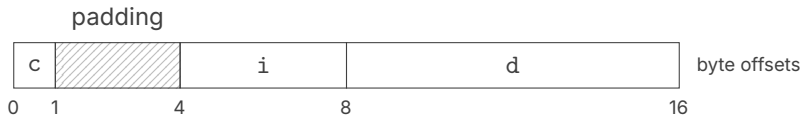
```
struct Point {           // a new type: struct Point
    double x;            // two fields, or members
    double y;
};
struct Point p = {1.5, 2.5}; // initialized in order
p.x = 3.0;                // field access, with a dot
double d = p.x * p.y;      // fields are ordinary variables
struct Point r = p;        // assignment copies every field
```

- + A **struct** declaration defines a **type**, not a variable; the type is written **struct Point** (the keyword is part of the name)
- + Values of a structure type behave like any other value: they can be **assigned**, passed to a function, and returned from one (unlike arrays, lecture 4), which **copies all the fields**
- + **But** `p == r` is **not allowed**: comparison must be written field by field

A structure in memory

C

```
struct S { char c; int i; double d; };  
printf("%zu\n", sizeof(struct S)); // 16, not 1 + 4 + 8 = 13
```



- ✦ The fields are stored **in order**, in one contiguous block: the compiler reaches a field by adding a constant **offset** to the address of the structure, in $O(1)$
- ✦ The compiler may insert **padding** between the fields, so that each starts at an address suited to its size: the size of a structure can exceed the sum of those of its fields

Naming a structure type: typedef

C

```
typedef struct {  
    int *data;  
    size_t size;  
    size_t capacity;  
} DynArray;           // DynArray: another name for this structure type  
  
DynArray a;           // no struct keyword needed any more
```

- + `typedef` gives a **new name** to an existing type; combined with a structure declaration, it lets the type be written `DynArray` instead of `struct DynArray`
- + Conventional in C for the types of a library; the name of the structure can then be omitted, as above
- + In C++, `struct Point` can always be written `Point`: no `typedef` needed

Functions on structures

C

```
double norm(struct Point p) {           // p: a copy of the argument
    return sqrt(p.x * p.x + p.y * p.y); // modifying p.x is lost
}

void translate(struct Point *p, double dx, double dy) {
    p->x += dx;                          // (*p).x += dx
    p->y += dy;                          // the caller sees the change
}

double distance(const struct Point *p, const struct Point *q);
```

- + Passing a structure **by value** copies all its fields: fine for a point, wasteful for a large record
- + Passing the **address** (lecture 2) is $O(1)$ and lets the function modify the structure; `p->x` abbreviates `(*p).x`, and a pointer the function only reads is declared `const` (lecture 4)
- + These functions are the **methods of a C structure**, with the object passed explicitly as the first parameter

A dynamic array as a structure (1/2)

C

```
typedef struct {  
    int *data; size_t size; size_t capacity;  
} DynArray;  
  
void da_init(DynArray *a) {                               // "constructor"  
    a->data = NULL; a->size = 0; a->capacity = 0;  
}  
  
void da_free(DynArray *a) {                               // "destructor"  
    free(a->data);  
    da_init(a);                                           // no dangling pointer left  
}  
  
int da_get(const DynArray *a, size_t i) { return a->data[i]; }  
size_t da_size(const DynArray *a) { return a->size; }
```

The three values of lecture 4 in one type, with the operations on it gathered in functions sharing the prefix da_

A dynamic array as a structure (2/2)

C

```
void da_push_back(DynArray *a, int x) {
    if (a->size == a->capacity) {                // full: double the capacity
        size_t new_capacity = a->capacity == 0 ? 1 : 2 * a->capacity;
        int *new_data = realloc(a->data, new_capacity * sizeof(int));
        if (new_data == NULL) { fprintf(stderr, "out of memory\n"); exit(1); }
        a->data = new_data; a->capacity = new_capacity;
    }
    a->data[a->size++] = x;                        // amortized O(1), lecture 4
}

int main(void) {
    DynArray a;
    da_init(&a);
    for (int i = 0; i < 10; i++) da_push_back(&a, i * i);
    printf("%zu %zu\n", a.size, a.capacity); // size 10, capacity 16
    da_free(&a);
}
```

What this hand-made object does not give us

- ✦ Nothing **prevents** the rest of the program from writing `a.size = 100`; and breaking the invariant $size \leq capacity$: the fields are visible to everybody (**no encapsulation**)
- ✦ Nothing **guarantees** that `da_init` is called before the first use, nor that `da_free` is called exactly once at the end (**no constructor, no destructor**); forgetting the latter is a memory leak
- ✦ The functions must all be named `da_something` by hand, and a second structure needs a second family of names (**no namespace**, no methods)
- ✦ An array of `double` instead of `int` means **copying** the whole code (**no genericity**, lecture 12)
- ✦ A sorted variant of the same array cannot reuse this one (**no inheritance**)

The C++ class answers all five; the rest of the lecture is about that, in C++ and in Python

Structures in C++

- ★ C++ keeps the C `struct`, so all of the above is valid C++, and adds to it: a `struct` can have **methods**, constructors, destructors, and can be inherited from
- ★ In C++, `struct` and `class` are **the same construct**, with one difference: what is declared in a `struct` is **public** by default, what is declared in a `class` is **private** by default

```
struct Point { double x, y; };           // both fields public
class Point { double x, y; };           // both fields private: useless
```

C++

- ★ **Convention:** `struct` for a plain record whose fields are all public and free of invariants, `class` as soon as there is something to protect

Classes



Classes and objects

- + A **class** is the description of a **kind of object**; an **object** is an **instance** of a class, created at run time
- + A class specifies:
 - properties** also called **fields**, **attributes**, or **members**: the values that each object of the class carries
 - methods** the functions that can be applied to an object of the class, and that work on its properties
- + A class can also have **static** (or **class**) properties and methods, shared by all its objects rather than carried by each of them (less frequent; see the complete programs)
- + A class is a **type**: the objects are the values of that type

Example: a class of matrices

Example

Matrix is a **class** describing the **mathematical concept** of a matrix; a particular matrix, say $\begin{pmatrix} 0 & 7 & 0 \\ 0 & 0 & 0 \end{pmatrix}$, is an **object** of that class.

Properties:

_n1, _n2 the numbers of rows and of columns

_rows the entries, as a sequence of n_1 sequences of n_2 numbers

Methods:

copy a copy of the matrix

+, · the sum with another matrix of the same dimensions, the product with a scalar

This class is the running example of the lecture, in Python and in C++

Classes in Python

Python

```
class Matrix:
    def dimensions(self):          # a method: self is the object
        return (self._n1, self._n2)
    def nb_rows(self):
        return self._n1
```

- + Everything indented inside `class` belongs to it: each `def` gives a **method**
- + Every method takes the object as its **first parameter**, conventionally called `self`, **not** written at the call site: `m.dimensions()`
- + The **properties are not declared**: the property `_n1` starts to exist the moment a method assigns `self._n1`

Classes in C++

C++

```
class Matrix {  
protected:  
    unsigned n1, n2;           // properties: declared,  
private:                     // with their type  
public:  
    std::vector<std::vector<double>> rows;  
    std::pair<unsigned, unsigned> dimensions() const { return {n1, n2}; }  
    unsigned nbRows() const { return n1; }  
};
```

- ✦ Properties are **declared with their type**, methods are ordinary functions written inside the class
- ✦ A method names the properties of **its own object** directly, with no `self`

Creating and using objects

Python

```
m = Matrix(2, 3)      # an object
m[0][1] = 7
print(m.dimensions()) # (2, 3)
```

C++

```
Matrix m(2, 3); // on the stack
m[0][1] = 7;
Matrix *p = new Matrix(2, 3); // heap
delete p; // lecture 4
```

- ✦ In Python, an object is always on the heap and the variable holds its **address** (lecture 4): `m2 = m` gives **another name for the same object**
- ✦ In C++, `Matrix m(2, 3);` puts the object **on the stack**, like an `int`: it is destroyed at the end of the block, and `Matrix m2 = m;` **copies** it
- ✦ `new Matrix(2, 3)` puts it on the heap and returns its address, to be released with `delete`; needed for objects that outlive the block, and for polymorphism (see further)

this and const methods in C++

- ✦ Inside a method, `this` is a **pointer to the object** the method was called on: the property `x` of that object is `this->x`, usually written just `x`
- ✦ A method that does not modify its object must be declared `const`, after its parameters:

```
unsigned nbRows() const { return n1; }
```

C++

- the compiler **checks** that the method changes no property
- only `const` methods can be called on a `const` object, or through a `const` reference (lecture 4: `const Matrix &m` as a parameter)
- ✦ Python has no equivalent: `self` is always modifiable

Construction



Constructors and destructors

- ✦ **Constructing** an object (**instantiating** the class) means creating it and giving its properties their initial values: the job of a **constructor**, a special method called when the object is created
 - its parameters are the data needed to build the object
 - a class can have **several** constructors in C++ (different parameters); Python has one
- ✦ **Destroying** an object is the job of a **destructor**, a special method called at the end of the object's life, useful when the object holds a **resource** to release: a memory block, an open file, a connection
- ✦ **The point**: an object is **never** left half-built, and what it acquires is released **without the user of the class having to remember** – the two things the C dynamic array of the previous section could not guarantee

Constructor in Python

Python

```
class Matrix:
    def __init__(self, n1, n2, value=0):      # constructor
                                             # creates the properties
        self._n1 = n1
        self._n2 = n2
        self._rows = [[value] * n2 for _ in range(n1)]

m = Matrix(2, 3)                          # calls __init__(m, 2, 3)
n = Matrix(2, 3, value=1)                  # a default value, a keyword argument
```

- ✦ The constructor is the method named `__init__`; it is called on the freshly created object, **returns nothing**, and an object is created by **calling the class**:
`Matrix(2, 3)`
- ✦ **No destructor** to write: an object is reclaimed automatically once nothing refers to it any more (lecture 4), and a resource such as an open file is released explicitly (lecture 10)

Constructors in C++

C++

```
class Matrix {  
    ...  
public:  
    Matrix(unsigned n1, unsigned n2, double value = 0.)  
        : n1(n1), n2(n2), rows(n1, std::vector<double>(n2, value)) {}  
};  
  
Matrix m(2, 3);           // on the stack  
Matrix *p = new Matrix(2, 3); // on the heap
```

- ✦ A constructor is a method with **the name of the class** and **no return type**; several constructors can coexist, as long as their parameters differ
- ✦ The **initializer list**, between : and the body, initializes each property by calling **its own** constructor: the preferred form
- ✦ A class with **no** declared constructor gets a **default** one taking no argument

Destructors in C++

C++

```
class Matrix {  
public:  
    ~Matrix() {                               // destructor  
        std::cout << "Matrix " << n1 << "x" << n2 << " destroyed\n";  
    }  
};  
{ Matrix m(2, 3); ... } // end of the block: ~Matrix() runs here, always
```

- ✦ A destructor is named after the class, prefixed with ~, and takes no argument; there is only one, and a class holding no resource needs none (the default one destroys each property)
- ✦ It runs at a **predictable moment**: the end of the block for an object on the stack, the `delete` for one on the heap

A class that owns memory

C++

```
class DynArray {  
    int *data;                // private: the invariant is ours to keep  
    std::size_t size_, capacity;    // size_: size() is the method  
public:  
    DynArray() : data(nullptr), size_(0), capacity(0) {}  
    ~DynArray() { delete[] data; }    // released, whatever happens  
    void push_back(int x);            // doubling, as in lecture 4  
    std::size_t size() const { return size_; }  
    int &operator[](std::size_t i) { return data[i]; }  
};  
void f() {  
    DynArray a;  
    for (int i = 0; i < 10; i++) a.push_back(i * i);  
}                                     // no free to write: ~DynArray() runs here
```

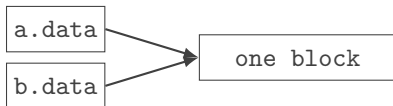
The same dynamic array as in C, with the invariant **protected** and the release **automatic**: the pattern behind `std::vector` and every container of the standard library

Copying an object

- C++ **copies** objects on assignment, when passing by value, and when returning by value; without instructions, it copies **each property**: exactly right for Matrix (whose vectors copy themselves), wrong for DynArray

```
DynArray a;  
a.push_back(1);  
DynArray b = a;    // copies the pointer  
b[0] = -1;          // also changes a[0]!  
                  // then both destructors  
                  // delete the same block
```

with the default copy



- Copying the fields of an object that **points** to data shares the data: a **shallow copy**; copying the data as well is a **deep copy**
- Here the shallow copy gives a **double free** and two objects modifying each other: two of the memory errors of lecture 4

The rule of three

C++

```
DynArray(const DynArray &other)           // copy constructor: new
    : data(new int[other.capacity]), size_(other.size_), // object, copy of another
    capacity(other.capacity) {
    for (std::size_t i = 0; i < size_; i++) data[i] = other.data[i];
}
```

- ★ The **copy constructor** builds a new object from an existing one: it runs for `DynArray b = a;`, and when an object is passed or returned by value
- ★ **Rule of three**: a class that needs a **destructor** needs a **copy constructor** and a **copy assignment** (`operator=`, written likewise) too – and a class that holds only standard containers needs **none of the three**

Copying an object in Python

Python

```
import copy
m = Matrix(2, 3)
n = m                                # the same object, under two names
shallow = copy.copy(m)              # a new object, sharing the rows
deep = copy.deepcopy(m)             # a new object, with its own rows
shallow._rows[0][0] = 1             # also changes m
deep._rows[0][0] = 2                # changes only deep
```

- ✦ In Python, **nothing is ever copied implicitly**: assignment, argument passing, and returning all pass the **address** of the object (lecture 4)
- ✦ `copy.copy` makes a **shallow** copy (the properties, shared), `copy.deepcopy` a **deep** one (the properties, copied recursively)
- ✦ Our `Matrix.copy` calls `deepcopy`, so that `3 * m` leaves `m` intact
- ✦ **The same question in both languages**: does a copy of this object share anything with the original? The answer belongs in the documentation of the class

Encapsulation



Encapsulation

- ✦ A class groups the data of an object and the operations on it; **encapsulation** is the decision to let the rest of the program use **only part** of them
- ✦ What is hidden can be changed without touching any other file: the class promises its **methods**, not its **storage** (DynArray can switch from doubling to another growth factor, DiagonalMatrix can store only its diagonal)
- ✦ What is hidden also cannot be broken from outside: the class is the only place where the **invariants** of its objects can be violated (*size* $\leq$ *capacity*, the dimensions matching the number of rows stored)
- ✦ Three usual degrees of **visibility**:
 - public** usable by everybody
 - protected** usable inside the class and its subclasses
 - private** usable inside the class only

Encapsulation in C++

C++

```
class Matrix {  
private:  
    std::vector<std::vector<double>> rows;    // ours only  
protected:  
    unsigned n1, n2;                          // also for the subclasses  
public:  
    unsigned nbRows() const { return n1; }    // read-only access  
    void resize(unsigned n1, unsigned n2);    // changes both, consistently  
};
```

- + **public, protected, private** introduce a section of the class; the compiler **refuses** any access from outside to what is not public (a compilation error, not a convention)
- + Default: **private** in a **class**, **public** in a **struct**
- + **Usual practice**: properties private or protected, methods public as far as they are part of what the class offers; `n1` public would allow `m.n1 = 100;`, and nothing else would follow

Encapsulation in Python

- + Python enforces almost nothing, and relies on a **naming convention** (except for the special methods `__init__`, `__str__`...):
 - public** a name starting with a letter: `dimensions`
 - protected** a name starting with **one underscore**: `_rows`
 - private** a name starting with **two underscores**: `__secret`
- + A single underscore is **only** a message to the reader: "this is an implementation detail, do not use it from outside"; the language allows the access
- + Two underscores make the name harder to reach from outside (the language **mangles** it): a protection against **accidents**, not against a determined user
- + **In practice**: one underscore for everything internal, which is what our `Matrix` does

Accessors

- + An **accessor** is a method giving controlled access to a property: a **getter** to read it, a **setter** to modify it, checking what must be checked – the only way in C++ once a property is private
- + In Python, a property can become a pair of accessors **without changing the code that uses it**, thanks to the decorator `@property`:

Python

```
class Matrix:
    @property
    def label(self):
        return self._label
    @label.setter
    def label(self, text):
        if not text:
            raise ValueError("the label cannot be empty")
        self._label = text
```

Hence the Python habit: a **plain public property** until a check is needed, then `@property`

Operators



Specializing behavior

- ★ A class is more pleasant to use when its objects behave like the values of the language: printing a matrix with `print(m)`, adding two matrices with `m + n`, reading an entry with `m[i][j]`, comparing with `m == n`
- ★ Both languages allow the class to **define what the operators and the built-in operations mean** for its objects:
 - in Python, by defining **special methods** with reserved names, `__add__`, `__str__`... (also called **magic methods**, or **dunder** methods, for *double underscore*)
 - in C++, by defining or overloading the **operator functions** `operator+`, `operator<<`...
- ★ **With restraint**: an operator must mean for the class what it means everywhere else. `+` on matrices, yes; `+` to insert into a tree, no; a method with a name is clearer

The main special methods in Python

Method	Called by	Meaning
<code>__init__</code>	<code>Matrix(2, 3)</code>	construction
<code>__str__</code>	<code>print(m)</code> , <code>str(m)</code>	readable text
<code>__len__</code>	<code>len(m)</code>	number of elements
<code>__getitem__</code> , <code>__setitem__</code>	<code>m[i]</code> , <code>m[i] = x</code>	indexing
<code>__eq__</code> , <code>__lt__</code>	<code>m == n</code> , <code>m < n</code>	comparison
<code>__add__</code> , <code>__mul__</code>	<code>m + n</code> , <code>m * n</code>	arithmetic, object on the left
<code>__radd__</code> , <code>__rmul__</code>	<code>3 * m</code>	arithmetic, object on the right

- ★ `3 * m` first tries `(3).__mul__(m)`, which fails since `int` knows nothing of matrices, then `m.__rmul__(3)`: hence the “reflected” variants

Special methods: example

Python

```
class Matrix:
    def __getitem__(self, i):          # m[i]: the i-th row
        return self._rows[i]         # so m[i][j] reads an entry
    def __str__(self):                # print(m)
        return "\n".join(" ".join(str(x) for x in self[i])
                           for i in range(self._n1))
    def __eq__(self, other):          # m == n, up to a tolerance
        if self.dimensions() != other.dimensions():
            return False
        return all(abs(self[i][j] - other[i][j]) <= 1e-9
                    for i in range(self._n1) for j in range(self._n2))
```

- + `m[2][3] = 7` works with `__getitem__` alone: it reads row 2, a list, then modifies that list
- + Inside the class, `self[i]` goes through `__getitem__` too

Special methods: arithmetic

Python

```
class Matrix:
    def __add__(self, other):
        # m + n
        if self.dimensions() != other.dimensions():
            raise ValueError("matrices of different dimensions")
        result = self.copy()
        for i in range(self._n1):
            for j in range(self._n2):
                result[i][j] += other[i][j]
        return result
    def __rmul__(self, scalar):
        # 3 * m
        result = self.copy()
        for i in range(self._n1):
            for j in range(self._n2):
                result[i][j] *= scalar
        return result
```

Convention: an arithmetic operator returns a **new object** and leaves its operands **unchanged**, as + does on numbers; hence the copy at the beginning

Operator overloading in C++

- ★ An operator is a function named `operator` followed by the symbol; it can be a **method** (the object is the left operand) or a **free function** (both operands are parameters)

C++

```
class Matrix {  
public:  
    bool operator==(const Matrix &other) const;    // m == n  
    Matrix operator+(const Matrix &other) const;    // m + n  
    std::vector<double> &operator[](unsigned i);    // m[i]: a row  
};  
Matrix operator*(double scalar, const Matrix &m); // 3 * m: outside  
std::ostream &operator<<(std::ostream &os, const Matrix &m); // cout << m
```

- ★ A free function is the only possibility when the left operand is **not** of our class:
`3 * m, std::cout << m` (lecture 10), which plays the role of Python's `__str__`

Operator overloading: example

C++

```
class Matrix {
public:
    std::vector<double> &operator[](unsigned i) { return rows[i]; } // a row
    virtual void print(std::ostream &os) const;
};

Matrix operator*(double scalar, Matrix m) { // m: a copy, by value
    for (unsigned i = 0; i < m.nbRows(); ++i)
        for (unsigned j = 0; j < m.nbColumns(); ++j)
            m[i][j] *= scalar;
    return m; // the copy, modified
}

std::ostream &operator<<(std::ostream &os, const Matrix &m) {
    m.print(os); // printing is a method...
    return os; // ... so it can be virtual
}
```

Operator overloading: use

C++

```
Matrix m(2, 3);  
m[0][1] = 7;  
std::cout << m;  
std::cout << 3 * m;  
std::cout << (m + m == 2 * m) << "\n";
```

Output

```
0 7 0  
0 0 0  
0 21 0  
0 0 0  
1
```

Python

```
m = Matrix(2, 3)  
m[0][1] = 7  
print(m)  
print(3 * m)  
print(m + m == 2 * m)
```

Output

```
0 7 0  
0 0 0  
0 21 0  
0 0 0  
True
```

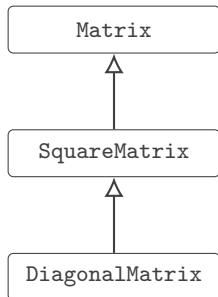
The same program in both languages, and in both the operators of the class are read exactly like those of the built-in types; C++ prints a `bool` as 1 or 0

Inheritance



Inheritance

- A **subclass** (or **derived class**) describes a **more specific** kind of object than its **superclass** (or **base class**)
- It **inherits** the properties and methods of the superclass, can **add** its own, and can **specialize** (redefine, override) the inherited ones
- **Is-a** relation: every square matrix **is a** matrix, so anything written for matrices applies to square ones
- Several subclasses can share a superclass, and a subclass can have subclasses of its own: a **hierarchy**



A square matrix has one dimension instead of two, and a determinant; a diagonal matrix has a faster one

Inheritance in Python

Python

```
class SquareMatrix(Matrix):           # superclass in parentheses
    def __init__(self, n, value=0):
        super().__init__(n, n, value)  # the constructor of Matrix

    @property
    def n(self):
        return self._n1

    def determinant(self):             # a new method
        ...
```

- ★ The superclasses are listed after the name of the class; with none, the class derives from object
- ★ `super()` designates the superclass: `super().method(args)` calls its method of that name, the **usual** way to reach an overridden method
- ★ `isinstance(x, Matrix)` tests whether `x` is an instance of `Matrix` **or of any subclass**

Inheritance in Python: specializing

Python

```
class DiagonalMatrix(SquareMatrix):  
    def __init__(self, n, value=0):  
        super().__init__(n)           # an n × n matrix of zeros  
        for i in range(n):  
            self[i][i] = value  
    def __rmul__(self, scalar):         # redefined: O(n) instead of O(n²)  
        result = self.copy()  
        for i in range(self.n):  
            result[i][i] *= scalar  
        return result
```

- ★ Specializing (**overriding**) a method is redefining it in the subclass: no keyword, nothing to declare in the superclass
- ★ Everything else (`__str__`, `__add__`, dimensions, `n...`) is inherited unchanged; the subclass can still call the overridden method with `super().__rmul__(scalar)`

Inheritance in C++

C++

```
class SquareMatrix : public Matrix {           // public inheritance
public:
    SquareMatrix(unsigned n, double value = 0.)
        : Matrix(n, n, value) {}               // constructor of the base

    unsigned n() const { return n1; }           // n1: protected in Matrix

    virtual double determinant() const;         // may be specialized
};
```

- + `: public` Base after the name of the class: `public` inheritance is the **is-a** relation
- + The base constructor is called in the **initializer list**, before the properties of the subclass
- + A subclass reaches the `protected` members of its base, not its `private` ones; an overridden method is reached explicitly with `Matrix::print(os)`
- + A method a subclass may specialize is declared `virtual` in the base class and `override` in the subclass

Inheritance in C++: specializing

C++

```
class DiagonalMatrix : public SquareMatrix {  
public:  
    DiagonalMatrix(unsigned n, double value = 0.) : SquareMatrix(n) {  
        for (unsigned i = 0; i < n; ++i) (*this)[i][i] = value;  
    }  
    // The determinant of a diagonal matrix: O(n) instead of O(n!)  
    double determinant() const override {  
        double product = 1.;  
        for (unsigned i = 0; i < n(); ++i) product *= (*this)[i][i];  
        return product;  
    }  
};
```

- + `override` is optional but recommended: the compiler then **checks** that the base class has a method of that signature, catching the typo that **adds** a method
- + This `DiagonalMatrix` keeps the n^2 entries of its base: storing only the diagonal needs an **abstract** base class (next section)

Inheritance or composition?

- ✦ Inheritance is the **most visible** tool of object-oriented programming, and the **most often misused**: it ties the subclass to the internals of its superclass, and the tie cannot be undone later
- ✦ Before writing `class B : public A`, check that every B really **is an** A – that it can be used wherever an A is expected, and respects what the methods of A promise
- ✦ The alternative is **composition**: B **has a** A as one of its properties, and offers the methods it chooses. A Matrix has a `std::vector` of rows; it is not a `std::vector`
- ✦ **Rule of thumb**: composition by default, inheritance when several classes must be **usable through a common interface** – which is what the next section is about

Polymorphism



Abstract classes

- ✦ An **abstract** class describes what its objects can do without saying how: it is meant to be **inherited from**, not instantiated
- ✦ It declares **abstract methods** – a name, parameters, a meaning, no implementation – that its **concrete** subclasses must define
- ✦ Such a class is an **interface**: a contract between the code that uses the objects and the classes that implement them
- ✦ **Example**: `MathematicalObject` with an abstract method `size`; `Matrix` and `Set` implement it, each in its own way. Whoever writes `print(o.size())` needs to know nothing else
- ✦ **Example**: a class of matrices with an abstract `at(i, j)`, implemented by a dense subclass storing n^2 numbers and by a diagonal subclass storing n – the design our `DiagonalMatrix` lacks

Abstract classes in Python

Python

```
from abc import ABC, abstractmethod

class MathematicalObject(ABC):           # ABC: abstract base class
    @abstractmethod
    def size(self):
        ...                               # no implementation
class Matrix(MathematicalObject):        # concrete subclass
    def size(self):
        return self._n1 * self._n2

MathematicalObject() # TypeError: Can't instantiate abstract class ...
```

- + No keyword for this: the module `abc` of the standard library provides the base class `ABC` and `@abstractmethod`
- + Instantiating a class with an abstract method left undefined raises a `TypeError` (lecture 8)

Abstract classes in C++

C++

```
class MathematicalObject {  
public:  
    virtual unsigned size() const = 0;    // = 0: pure virtual, no definition  
};  
class Matrix : public MathematicalObject {  
public:  
    unsigned size() const override { return n1 * n2; }  
};  
  
MathematicalObject o;    // error: cannot declare variable of abstract type
```

- ✦ A method declared **virtual** with `= 0` instead of a body is **pure virtual**; a class with at least one is **abstract** and cannot be instantiated, and a subclass leaving one undefined is abstract too
- ✦ A class with only pure virtual methods is the C++ form of an **interface**, like Python's ABC

Polymorphism

- ✦ **Polymorphism**: objects of different classes, sharing a superclass or an interface, are manipulated **uniformly**, and each behaves **according to its own class**
- ✦ The caller writes one piece of code; which method actually runs is decided by the **class of the object**, possibly only at run time (**dynamic dispatch**)
- ✦ **Example**: every square matrix has a determinant. Computing it by expansion costs $O(n!)$; for a diagonal matrix it is the product of the diagonal, $O(n)$. Both classes offer determinant, and `m.determinant()` calls the **right one**
- ✦ The gain is in the code that **uses** the objects: a function computing the determinant of each matrix of a list works for any present or future subclass, with no test on the class and no modification

Polymorphism in Python

Python

```
class SquareMatrix(Matrix):  
    def determinant(self):  
        ...  
        # expansion along the first row,  
        # in  $O(n!)$   
  
class DiagonalMatrix(SquareMatrix):  
    def determinant(self):  
        # the product of the diagonal,  
        return prod(self[i][i] for i in range(self.n)) # in  $O(n)$ 
```

- + Nothing to declare in SquareMatrix for its method to be specialized, and nothing to write in the code calling `m.determinant()`
- + A method call looks for the method **in the class of the object**, then up the hierarchy: the behavior is that of the actual class

Polymorphism in Python: duck typing

Python

```
def total_determinant(matrices):          # works on anything with
    return sum(m.determinant() for m in matrices)  # a determinant

print(total_determinant([SquareMatrix(3), DiagonalMatrix(3, 2)]))
```

- + Python is **dynamically typed** (lecture 1): a variable has no declared type, and a function imposes none on its arguments
- + A function therefore accepts **any** object having the methods it uses, related to the others by inheritance **or not**: "if it walks like a duck and quacks like a duck, it is a duck" (**duck typing**)
- + Polymorphism in Python is thus wider than inheritance – and checked **only when the method is called**: a missing method is an `AttributeError` at run time (lecture 8), where C++ refuses to compile
- + **Consequence**: an abstract base class **documents** an interface and catches mistakes early; the language does not require one

Polymorphism in C++

C++

```
DiagonalMatrix d(3, 2);  
SquareMatrix *p = &d;           // a pointer to the base class  
std::cout << p->determinant(); // 8: DiagonalMatrix::determinant  
SquareMatrix copy = d;           // by value: only the SquareMatrix part  
std::cout << copy.determinant(); // 8 as well, but in O(n!)  
void show(const Matrix &m) { std::cout << m; } // by reference: virtual
```

- + A method declared **virtual** is chosen **according to the class of the object**, at run time, through a pointer or a reference to the base class; a non-virtual method is chosen according to the **declared** type, when compiling
- + **Trap**: an object assigned or passed **by value** to a variable of the base class is **truncated** to its base part (**slicing**) and loses its class: polymorphism in C++ needs a **&** or a *****
- + Likewise, a class meant to be inherited from declares its destructor **virtual**, so that **delete** through a base-class pointer runs the **actual** class's

Polymorphism in C++: a collection of matrices

C++

```
std::vector<SquareMatrix *> matrices;
matrices.push_back(new SquareMatrix(3));
matrices.push_back(new DiagonalMatrix(3, 2));
double total = 0.;
for (const auto *m : matrices)
    total += m->determinant();           // the method of the actual class
std::cout << total << "\n";
for (auto *m : matrices) delete m;      // lecture 4: each new, one delete
```

- + A collection of objects of **different** classes of the same hierarchy holds **pointers**: the objects have different sizes, and a `std::vector<SquareMatrix>` would slice them. The Python list of the previous slide is exactly this, with the pointers and the releases hidden (lecture 4)

What does an object cost?

		Memory	Time
C++	<code>struct</code> or class, no virtual method	16 bytes	0.9 ns
C++	class with a virtual method, through a pointer	24 bytes + pointer	2.3 ns
Python	object	200 bytes	67 ns

- + A point with two `double` coordinates and a method returning $x^2 + y^2$: memory per point, including its two numbers, and time per call over 10^7 points
- + A C++ object is **exactly its properties**, with no header: a class with no virtual method costs **nothing** over the `struct`, and its methods are **inlined**; one virtual method adds a pointer per object, an indirection per call
- + A Python object carries the header of lecture 4 and a **dictionary** of its properties, looked up by name at each access
- + **Reading**: the language costs two orders of magnitude here, a virtual method a factor of 2 – design for clarity, and measure (lecture 3) before trading it away

Coda



Wednesday's tutorial and references

- ★ Wednesday's tutorial: bundling related values into a `struct` in C and writing the functions that work on it, then the same data as a class in C++ (constructor, destructor, visibility) and in Python; operators and special methods on a class of your own; a small hierarchy with a specialized method, and a function using its objects polymorphically
- ★ The complete programs: `matrix.py`, `matrix.cpp`, `dynarray.c`, `dynarray.cpp`
- ★ Next lecture: the **data structures** behind the collections of lectures 3 and 4 (linked lists, hash tables, search trees), written as classes, and how to split a program into **several files**
- ★ References: King [2008], chapter 16 (structures); for Python, the part on classes of Lutz [2025] and Python Software Foundation [2026] for the special methods; for C++, chapters 4 and 5 (classes, essential operations) of Stroustrup [2022], and `cppreference.com` [2026]

Licensing

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



Bibliography I

- cppreference.com. Classes. <https://en.cppreference.com/w/cpp/language/classes>, 2026.
- K. N. King. *C Programming: A Modern Approach*. W.W. Norton & Company, 2 edition, 2008. ISBN 0393979504. URL <http://knking.com/books/c2/index.html>.
- Mark Lutz. *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media, 6 edition, 2025. ISBN 1098171306. URL <https://www.oreilly.com/library/view/learning-python-6th/9781098171301/>.
- Python Software Foundation. The python language reference: Data model. <https://docs.python.org/3/reference/datamodel.html>, 2026.
- Bjarne Stroustrup. *A Tour of C++*. Addison-Wesley, 3 edition, 2022. ISBN 978-0-13-681648-5. URL <https://www.stroustrup.com/tour3.html>.