C++

OBJECT-ORIENTED PROGRAMMING FOR C DEVELOPERS

C++

- C++ was first released in 1985
- Compiled language providing fine control on resource usage
- Any code written in C can be compiled and used in C++

Note: Python was first released in 1991 and is coded in C

Somehow, Python is an "interpreted cousin" of C++

IMPERATIVE PROGRAMMING

- Math-inspired paradigm
- Treat data using functions
- Procedural, iterative, instruction lists...
- Best when dealing with primitive data types (especially in large arrays)

C is such a language.

As well as C++, Python, Julia, JavaScript, *etc*.

OBJECT-ORIENTED PROGRAMMING

- Data is no longer "bare", but encapsulated in objects
- An object contains both data *and code*
    - *Data* as **attributes**
    - *Code* as **methods** (much like functions but exclusive to an object)
- Interacting with an object is done through its **methods**

Paradigm in C++, Java, Python, Julia, JavaScript, *etc.*

SOME TERMINOLOGY

- A **class** is the blueprint of an object, attributes and methods are defined in a class
- An **object** is an **instance of a class**, there can be many objects of the same class
- An **attribute** is to a class what a **field** is to a struct
- A **method** is a class-specific function having access to:
  - `this` (*i.e.* a pointer to an object of this class)
  - all attributes and methods of *this* object

HOW TO IMPLEMENT IT IN C ?

Start with a struct

```c
struct Object {
    int anInt;  // fields in C -> attributes in OOP
    float aFloat;
    char someText[32];
};
```

Initialise an object

```c
struct Object initObject(int i, float f, char text[]) {
    struct Object object = {i, f, ""};
    strncpy(object.someText, text, 32);
    object.someText[31] = '\0';
    return object;
}
```

HOW TO IMPLEMENT IT IN C ?

Write some *methods* as functions

```c
void printObject(struct Object self) {
    printf("self.anInt : %i\n", self.anInt);
    printf("self.aFloat : %f\n", self.aFloat);
}
```

```c
void doubleObject(struct Object* self) {
    self->anInt *= 2;
    self->aFloat *= 2;
}
```

Noticed a difference ?

`struct Object self`

`struct Object* self`

HOW TO IMPLEMENT IT IN C ?

Use it in a program

```c
int main(int argc, char* argv[]) {
  srand(42);  // seeding the rng
  for (int i=0; i+1 < argc; ++i) {
    float num = (float)rand();
    float den = (float)rand();
    while (den == 0.f) den = (float)rand();
    struct Object object = initObject(i, num/den, argv[i+1]);
    printObject(object);
    doubleObject(&object);
    printObject(object);
    printf("self.someText : %s\n", object.someText);
    printf("\n");
  }
}
```

```
$ ./cOOP this is a test veryLongTextThatIsGoingBeyond32Characters
```

NOTE ON PYTHON

Have you noticed the "self" in the code ?

See what I'm doing here ?

This is basically how the Python interpreter is implemented.

At C level, `object` is in fact a pointer to a struct called `PyObject`.

C++ SYNTAX BASICS

REFERENCES AND POINTERS

- `Type*` : a *pointer* to an instance of Type
- `&arg` : returns the *address* of arg
- `*ptr` : *dereferences* a pointer
- `Type&` : a *reference* to an instance of Type

References are not present in C, but are of utmost importance in C++

REFERENCES AND POINTERS

```
Object obj;  // This is an instance of the class Object
Object* objPtr;  // This is a pointer to an Object
objPtr->print();  // Will segfault !!
objPtr = &obj;  // Store the address of obj into a pointer
objPtr->print();  // Call the print() method of obj
obj.print();  // Exactly equivalent to the previous line
```

In C++ you can use references in addition to pointers

```
Object obj;  // This is an instance of Object
Object& objRef(obj);  // This is a reference to obj
objRef.print();  // Call the print() method of obj
obj.print();  // Exactly equivalent to the previous line
```

REFERENCES AND POINTERS

```
Object* objPtr;  // This is a pointer to an object
objPtr = new Object;  // Store the address of a new Object
Object& objRef(*objPtr);  // Reference to a dynamic Object
Object obj(*objPtr); // (static) copy of a dynamic Object
objPtr->print();  // Call the print() method
objRef.print();  // Exactly equivalent to the previous line
delete objPtr;  // Destroy Object and free the memory
objPtr->print();  // Will segfault !!
objRef.print();  // Will segfault !!
obj.print();  // Call the print() method of the copy
```

NOTION OF "CONSTNESS"

You can code in C++ without using the keyword `const`

But should you ?

COPY VS REFERENCE

```
// Here, a copy is made
void someFct(Object object) {
  object.modifyAttributes();  // The attributes changed
  // No changes persist outside of this function
  // since the changes are made to a copy
}
```

```
// Here, no copy is made
void someFct(Object& object) {
  object.modifyAttributes();  // The attributes changed
  // The changes are made to the original object (no copy)
}
```

What if you want to ensure the argument (object) is not modified, but you also want to avoid a copy being made ?

CONST ARGUMENTS

```
// Here, no copy is made
void someFct(const Object& object) {
  object.modifyAttributes();  // This will not be allowed
  int i = object.anInt();  // Is this allowed ?
}
```

That depends

```
class Object {
[...]
public:

  int anInt() { return 21; }
  int anInt() const { return 42; }
};
```

CONST ARGUMENTS

```cpp
class Object {
[...]
public:
  // Only allowed in a non-const context
  int anInt() { return 21; }

  // Used in a const context (avail. in non-const if needed)
  int anInt() const { return 42; }
};
```

```cpp
void testNoConst(Object& obj) {
  std::cout << "testNoConst : " << obj.anInt() << std::endl;
}  // testNoConst : 21

void testConst(const Object& obj) {
  std::cout << "testConst : " << obj.anInt() << std::endl;
}  // testConst : 42
```

CLASS BASICS

CLASS BASICS

Let's start with some code:

```
class Object {

  int anInt;
  float aFloat;
  char someText[32];  // -> This is C

public:
  // This is the "C way"
  Object(int i, float f, char text[]) :
        anInt(i), aFloat(f), someText{} {
    strncpy(someText, text, 32);
    someText[31] = '\0';
  }
};
```

CLASS BASICS

Use the C++ standard library:

```
class Object {

  int anInt;
  float aFloat;
  std::string someText;  // -> This is C++

public:

  Object(int i, float f, char text[]) :
        anInt(i), aFloat(f), someText(text) {}
};
```

CLASS BASICS

```
class Object {
[...]
  void print() {
    std::cout << "this->anInt : " << anInt << std::endl;
    std::cout << "this->aFloat : " << aFloat << std::endl;
  }

  std::string getSomeText() { return someText; }

  void doubleValues() {
    anInt = anInt * 2;  // eq. to anInt *= 2;
    aFloat *= 2;  // eq. to aFloat = aFloat * 2;
  }
};
```

PRIVATE AND PUBLIC

In a struct, all *fields* are public.

In a class, all *attributes* are private by default. (But they can be declared *protected* or *public*)

OOP is all about using the *interface* of a class

```
class Object {

  int _anInt;  // private

public:

  void setAnInt(int i) { _anInt = i; }  // setter
  int anInt() { return _anInt; }  // getter

  void anInt(int i) { _anInt = i; }  // setter
};
```

INTERLUDE : NAMING CONVENTIONS

When starting a project, choose a naming convention ***and stick to it*** *e.g.*:

- Classes start with a capital letter, instances do not
- Private attributes start with an underscore
- camelCaseLooksLikeThis
- PascalCaseLikeThisNoticeTheCapitalFirstLetter
- and_snake_case_like_this

OPERATORS OVERLOADING

```
class Object {

  int _anInt;
  float _aFloat;

public:

  Object& operator=(int i) {
    _anInt = i; return *this;
  }

  Object& operator=(float f) {
    _aFloat = f; return *this;
  }
};
```

OPERATORS OVERLOADING

```
class Object {
[...]
public:

  Object& operator*=(float f) {
    _anInt *= f;  // Will be "floored" to an int
    _aFloat *= f;
    return *this;
  }

  Object& operator*=(int f) {...}
  Object& operator*=(double f) {...}
};
```

OPERATORS OVERLOADING

```
class Object {

  int _anInt;
  float _aFloat;
  std::string _someText;

public:
  // Make the compiler generate overloads
  template <typename NumericType>
  Object& operator*=(NumericType n) {
    _anInt *= n;  // eq. to this->_anInt *= n;
    this->_aFloat *= n;  // eq. to _aFloat *= n;
    return *this;
  }
};
```

COPY OPERATOR/CONSTRUCTOR

```
class Object {
[...]
public:
  //Object(const Object& other) : _anInt(other._anInt),
  //    _aFloat(other._aFloat), _someText(other._someText) {}
  Object(const Object& other) = default;

  Object& operator=(const Object& other) {
    _anInt = other._anInt;
    _aFloat = other._aFloat;
    _someText = other._someText;
    return *this;
  }
};
```

IMPLICIT VS EXPLICIT

```
class Object {
[...]
public:
  // Conversion :
  operator int() { return _anInt; }  // implicit operator
  explicit operator float() { return _aFloat; }
  // Assignation :
  Object& operator=(int i) { _anInt = i; return *this; }
  Object& operator=(float f) { _aFloat = f; return *this; }
};
```

```
int main() {
  Object object(42, 32.5, "blah");
  int i;
  float f;

  i = object;  // i = 42
  f = object;  // f = 42.f
  f = static_cast<float>(object);  // f = 32.5f
}
```

SO FAR...

- Encapsulation
- Overloading (operators and methods)
- Overloading (const and no-const)
- Explicit/Implicit conversions

VIRTUALITY & INHERITANCE

POLYMORPHISM

OOP is about a lot more than "code in a struct".

What if we wanted several different classes sharing (part of) the same *interface* ?

A bit like how we can mix up numeric types (int, float...) and let the compiler handle the casting

This is called **polymorphism**. That is, behaving the same (to a point), while being different classes.

POLYMORPHISM

Again, how to do it in C ?

```c
struct Number {
  void* attributes;
  struct Number(*copy)(struct Number*);  // copy
  void(*print)(struct Number*);

  float(*re)(struct Number*);  // ptr to a function
  float(*im)(struct Number*);  // those are getters
  float(*norm)(struct Number*);

  void(*imul)(struct Number*, struct Number*);  // op. *=
  void(*iadd)(struct Number*, struct Number*);  // op. +=
  struct Number(*mul)(struct Number*, struct Number*);  // *
  struct Number(*add)(struct Number*, struct Number*);  // +
};
```

POLYMORPHISM

Don't get confused, there is no code in a struct, but pointers to functions.

The idea here is to create a *table* that maps a **signature** to a function meant for a type of Number.

This *table* is the interface. Common to all *instances* of Number.

But the programmer can initialise each instance of Number with pointers to different custom functions.

POLYMORPHISM

Let's say we want *Real* numbers and *Complex* numbers:

```
struct Number initComplex(float re, float im);
struct Number initReal(float re);
```

And operators that work for all types of number :

```
float re(struct Number* a) { return a->re(a); }
float im(struct Number* a) { return a->im(a); }
float norm(struct Number* a) { return a->norm(a); }
void print(struct Number* a) { a->print(a); }
struct Number copy(struct Number* a) { return a->copy(a); }
struct Number sum(struct Number* a, struct Number* b) { ... }
struct Number prod(struct Number* a, struct Number* b) { ... }
void iadd(struct Number* a, struct Number* b) { ... }
void imul(struct Number* a, struct Number* b) { ... }
```

POLYMORPHISM

Let's again have a peek on the Python interpreter textbook :

Duck typing : "If it walks like a duck, and it quacks like a duck, then it must be a duck"

If we restrict our *"interface"* to how they walk and the noise they make, several birds might be mistaken for ducks.

To us, they are polymorphs

Very much like our Numbers

POLYMORPHISM

In-place operators :

```
void iaddComplex(struct Number* self, struct Number* other) {
  float* att = ((float*)self->attributes);
  att[0] = re(self) + re(other);
  att[1] = im(self) + im(other);
}

void imulComplex(struct Number* self, struct Number* other) {
  float* att = ((float*)self->attributes);
  att[0] = re(self) * re(other) - im(self) * im(other);
  att[1] = re(self) * im(other) + im(self) * re(other);
}
```

```
void iaddReal(struct Number* self, struct Number* other) {
  *((float*)self->attributes) = re(self) + re(other);
}

void imulReal(struct Number* self, struct Number* other) {
  *((float*)self->attributes) = re(self) * re(other);
}
```

POLYMORPHISM

add and mul methods :

```c
struct Number addNumber(struct Number* self,
                        struct Number* other) {
  struct Number res = copy(self);
  iadd(&res, other);
  return res;
}

struct Number mulNumber(struct Number* self,
                        struct Number* other) {
  struct Number res = copy(self);
  imul(&res, other);
  return res;
}
```

See ? These can be the same for both Real and Complex.

POLYMORPHISM

Complex :

```c
struct Number initComplex(float re, float im) {
    struct Number self = { malloc(2*sizeof(float)) };
    float* att = ((float*)self.attributes);
    att[0] = re;
    att[1] = im;
    [...]
    self.iadd = &iaddComplex;  // <-- see this ?
    self.imul = &imulComplex;  // <-- see this ?
    self.add = &addNumber;  // <-- and this ?
    self.mul = &mulNumber;  // <-- and this ?
    [...]
    return self;
}
```

```c
void destroyNumber(struct Number* self) {
  free(self->attributes);
  self->attributes = NULL;
}
```

POLYMORPHISM

```
struct Number initReal(float re) {
    struct Number self = { malloc(sizeof(float)) };
    float* att = ((float*)self.attributes);
    *att = re;
    [...]
    self.iadd = &iaddReal;  // <-- see this ?
    self.imul = &imulReal;  // <-- see this ?
    self.add = &addNumber;  // <-- and this ?
    self.mul = &mulNumber;  // <-- and this ?
    [...]
    return self;
}
```

```
void destroyNumber(struct Number* self) {
  free(self->attributes);
  self->attributes = NULL;
}
```

IN ACTION

```c
int main() {
  struct Number real = initReal(2.5f);
  struct Number complex = initComplex(2, 2);

  struct Number prodComplex = prod(&complex, &real);
  print(&prodComplex);  printf("\n");
  struct Number sumReal = sum(&real, &complex);
  print(&sumReal);  printf("\n");
  imul(&real, &complex);
  print(&real);  printf("\n");
  iadd(&complex, &complex);
  print(&complex);  printf("\n");

  destroyNumber(&real);  destroyNumber(&complex);
}
```

SUMMING UP

We can implement **polymorphism** at C level by associating *signatures* to custom functions.

The programmer needs to be careful to not mix up those functions and to keep track of the actual type of a Number (Real/Complex).

One can easily override a "method" at runtime.

In C++, you do not need to do all that bookkeeping work yourself.

VIRTUALITY & INHERITANCE IN C++

PARENT CLASS

```cpp
class Number {

    float _re;

public:
[...]
    float re() const { return _re; }
    virtual float im() const { return 0.f; }
    virtual float norm() const { ... }
[...]
};
```

VIRTUAL METHODS

Methods marked as virtual create an entry in the **virtual table**.

Child classes can override an entry in the virtual table, like in C with pointers to custom functions.

You can then manipulate objects of type `Number&` or `Number*` even if the actual object is a Real or Complex

PURE VIRTUAL

```cpp
class Number {
[...]
public:
    // This is a "pure virtual"
    virtual explicit operator std::string() =0;
    // This is a virtual with default implementation
    virtual Number& operator+=(const Number& other) {
        _re += other.re(); return *this;
    }
    // This is a virtual that throws an error
    virtual Number& operator*=(const Number&) {
        throw std::runtime_error(
        "Number& operator*=(const Number&) : Not implemented");
    }
};
```

Create an entry in the VTable, but do not populate it.

Child classes **have to** override pure virtual methods.

CHILD CLASSES

```
class Real : Number {
[...]
public:

    explicit operator std::string() override {
        return std::to_string(re());
    }

    Number& operator*=(const Number&) override {
        setRe(re()*other.re());
        return *this;
    }
[...]
};
```

Here, we want to override the pure virtual and the "error: not implemented" methods.

ABSTRACT CLASS

A class containing a pure virtual method cannot be instanciated. Such classes are called **abstract**

Abstract classes are very useful to polymorphism because you delegate the actual implementation to the child classes.

A class can inherit from several base classes.

INTERFACE DESIGN PATTERN

An interface is:

- An abstract class
- With no attributes
- All methods are virtual

You can see it as a building block of the interface of a child class

INTERFACE DESIGN PATTERN

```
class IPrintable {
public:
    virtual operator std::string() const =0;
    virtual void print() const {
        std::cout << static_cast<std::string>(*this)
                  << std::endl;
    }
};
```

```
class IBuffer {
public:
    virtual std::string read() const =0;
    virtual void write(const std::string&) =0;
};
```

INTERFACE DESIGN PATTERN

```
class Buffer : IBuffer, IPrintable {
    std::string _buff;
public:
    operator std::string() const override {
        return _buff;
    }
    std::string read() const override {
        return _buff;
    }
    void write(const std::string& text) override {
        _buff += text;
    }
};
```

INTERFACE DESIGN PATTERN

```
class Console : IBuffer, IPrintable {
    std::string _history;
public:
    operator std::string() const override {
        return _history;
    }
[...]
```

```
[...]
    std::string read() const override {
        std::string res;
        std::cin >> res;
        _history += "\n" + res;
        return res;
    }
    void write(const std::string& text) override {
        _history += "\n" + text;
        std::cout << text << std::endl;
    }
};
```

INTERFACE DESIGN PATTERN

```cpp
int main() {
    Console console;
    Buffer buffer;
    std::vector<IBuffer*>(2) buffers;
    buffers[0] = &console;
    buffers[1] = &buffer;
    for (auto* item: buffers)
        item->write("This is a test");
    for (IPrintable* item: {&console, &buffer})
        item->print();
}
```

We can use pointers to abstracts even if we cannot instanciate them.

This is a powerful mechanism to completely ignore the actual object's type while still interacting with it.

LAST RECOMMENDATIONS

- Use the standard library
- Avoid new/delete if you can
- Use meaningful names, avoid cryptic code
- Class declarations in .h files
- Non-template implementations in .cpp files
- Template implementations in .hpp files

LIBRARIES OF INTEREST

- Boost
- ArrayFire
- Eigen3
- Tensorflow
- libtorch
- OpenMP
- OpenMPI
- Qt
- Matplot++