# Introduction to Programming

## Lecture 6 — Objects in your programs

**Nicolas Gry**

Université de Saint-Étienne | Year 2026/2027

digicrea digital creativity
Université Jean Monnet Saint-Étienne

# 01 Recap’

# 02 Assignment review

# 03 Defining your own types

# An example of `struct`

Let’s walk through an example.

# Custom types and member variables

You can create a type with `struct`.

A type can contain **member variables**. When you instanciate a variable of a custom type, you are implicitly instanciating all those member variables.

# Custom types and member variables

You can create a type with `struct`.
A type can contain **member variables**. When you instanciate a variable of a custom type, you are implicitly instanciating all those member variables.

```
// Definition -> no instance in memory yet
struct Pixel {
    int r,g,b;
}; // Don't forget the semicolon here !

// Instanciation, p exists in memory
Pixel p;
p.r = 42;
```

# 04 Member functions

# Another cool example

You know the drill: `https://godbolt.org/z/8Khhqocc3`.

# Declaring and defining member functions

A custom type can have **member functions**. You declare them in your structure body; you can put the definition either in the structure body or outside of it. In the latter case, the function's name has to be preceded by a double colon "`::`".

# Declaring and defining member functions

A custom type can have **member functions**. You declare them in your structure body; you can put the definition either in the structure body or outside of it. In the latter case, the function's name has to be preceded by a double colon ":".

```
struct Foo{
    void a_function() {}
};
```

```
struct Foo{
    void a_function();
};
void Foo::a_function() {}
```

# Calling a member function

Once you have an instance of a custom type, you can call the member function with that instance.

# Calling a member function

Once you have an instance of a custom type, you can call the member function with that instance.

```
struct Number {
    int val;
    void increment() { val++; }
};
Number a {1};
Number b {10};
a.increment(); // a.val == 2
b.increment(); // b.val == 10
```

# Const functions 1/2

You can declare `const` functions to ensure that this function **cannot** alter the state of your object, i.e it cannot alter the member variables values.

# Const functions 1/2

You can declare `const` functions to ensure that this function **cannot** alter the state of your object, i.e it cannot alter the member variables values.

```
struct Foo{
    void func() const {} // func is const
};
```

# Const functions 2/2

When passing an object as a `const` argument to a function, only `const` member functions can be called in that function

# Const functions 2/2

When passing an object as a `const` argument to a function, only `const` member functions can be called in that function

```
struct Song{
    std::string name;
    std::string name () const { return name; }
    void rename(const std::string new_name) { name = new_name; }
};

void display(const Song& song){
    std::cout << song.name() << std::endl; // ok
    song.rename("hello"); // not ok, rename is not const
}
```

# Overloads

Just like *free* functions — regular functions —, you can *overload* member functions following the same rules as before.

```
struct RGBImage {
    std::vector<std::vector<Pixel>> pixels;

    void set_pixel(int x, int y, int r, int g, int b);
    void set_pixel(int x, int y, Pixel pixel);
};
```

# 05 Public vs Private members

# Abstraction needs

You may want to hide some information contained in your type. Things that should be available only inside your structure, but not outside of it should be `private`. Things available to everyone will be `public`.

```
struct MyType{
public:
    int everyone_can_see_me;
    int compute() {
        return everyone_can_see_me
            + private_part;
    }
private:
    int private_part;
};
```

# Abstraction needs

You may want to hide some information contained in your type. Things that should be available only inside your structure, but not outside of it should be `private`. Things available to everyone will be `public`.

```
struct MyType{
public:
    int everyone_can_see_me;
    int compute() {
        return everyone_can_see_me
            + private_part;
    }
private:
    int private_part;
};
```

```
MyType var;
var.compute(); // ok
var.private_part = 7; // not ok,
    won't compile
```

# `class` vs `struct`

Another keyword to define a type is `class`.

# `class` **vs** `struct`

Another keyword to define a type is `class`.

Members of a `struct` are `public` by default (strUct → pUblic).

Members of a `class` are `private` by default (clAss → privAte).

# `class` vs `struct`

Another keyword to define a type is `class`.

Members of a `struct` are `public` by default (strUct → pUblic).

Members of a `class` are `private` by default (clAss → privAte).

```
struct Foo{
    int var; // public
};
```

```
class Foo{
    int var; // private
};
```

# 06 Some special functions: constructors and destructors

# Constructors through an example

Let's dive in here: `https://godbolt.org/z/TbPvevjfc`

# Constructors 1/2

A **constructor** is a function that is called whenever a variable is instanciated.
The constructor that takes no argument is called the *default* constructor.

```
struct Pixel {
    int red, green, blue;
};
Pixel p1; // default constructor called
```

# Constructors 2/2

You can overload the constructor with one that takes argument. If you still want the default constructor to be available, you have to define it.

# Constructors 2/2

You can overload the constructor with one that takes argument. If you still want the default constructor to be available, you have to define it.

```
struct Pixel {
    int red, green, blue;
    Pixel () = default;
    Pixel(int r, int g, int b)
        : red{r}, green{g}, blue {b} {}
};
Pixel p1; // default constructor called
Pixel p2 {111, 22, 3}; // custom constructor
```

# Destructors 1/2

Conversly, a **destructor** is a function that is called whenever a variable is *destroyed* (i.e it goes out of scope).
The destructor never takes any argument.

# Destructors 2/2

```
struct AudioFile {
    AudioFile(const std::string filename) {
        m_file = sf_open(filename.c_str(), SF_READ, &m_info);
    }
    ~AudioFile(){
        sf_close(m_file);
    }
private:
    SF_INFO m_info;
    SNDFILE * m_file;
};
```

# 07 Abstraction

# Why do we need our own types?

It’s a way to group things that go together.
It’s easier to manipulate variables and functions that all relate to the same thing when they’re grouped in the same place.

# How do we design types?

Think about how the object will be used. Think in terms of functionality.
The *public interface* should expose *what you can do* with an object; *private members* contain *how the object implements it*.
This is called **abstraction**.

# Abstraction example

Without abstraction

```
std::vector<std::vector<int>>
    bw_image = {
     {12, 125}, {45, 242}};
int height = bw_image.size();
int width = bw_image[0].size();
bw_image[0][1] = 43;
```

With abstraction

```
BWImage image;
image.height();
image.width();
image.set_pixel(0, 1, 43);
```

# One class = One responsability

You don’t want to overcrowd a class code. Think in terms of what your class is responsible for, what it should represent.
In the case of an image, you split the definition of a pixel and the definition of an image.

# One class = One responsability

You don’t want to overcrowd a class code. Think in terms of what your class is responsible for, what it should represent.
In the case of an image, you split the definition of a pixel and the definition of an image.
This makes your code more *maintainable*. The more modular your code, the easier it is to modify a piece without breaking every other ones.

# Member function vs Free function

| Member functions | Free function |
| --- | --- |
| Operate on private members / internals. | Whenever possible, make a function a free function. It could be used similarly on other types. |

# Practise time !

# 08 An introduction to generic programming

# Making a function work on many types

An example.

# Generic functions

A **generic function** is a function that can operate on values of different types within a single definition.

```
template<typename ImageType>
void print_dimensions(const ImageType& image) {
    std::cout << "width: " << image.width()
        << std::endl;
    std::cout << "height: " << image.height()
        << std::endl;
}
```

# Type deduction 1/2

Type deduction happens the parameter of a generic function can be inferred by the call of the function. Type is usually deduced from passed arguments' type.

# Type deduction 1/2

Type deduction happens the parameter of a generic function can be inferred by the call of the function. Type is usually deduced from passed arguments' type.

```
template<typename T>
T add(T a, T b) { return a+b; }
add(1, 2); // T is int
add("hello ", "world"); // T is char *
```

# Type deduction 2/2

A function parameter cannot be deduced from the return type alone. When type deduction is not possible, you have to specify the parameter's type yourself. Example.

# Type deduction 2/2

A function parameter cannot be deduced from the return type alone. When type deduction is not possible, you have to specify the parameter's type yourself. Example.

```
template<typename Image>
Image create_image(int w, int h){
    return Image(w,h);
}
RGBImage image = create<RGBImage>(50, 50);
```

# Generic functions and multiple files

You **cannot** forward declare a generic function.

If you want to share a generic function using different files, its definition will have to go into a header file.

# Practise time!

# See you next time!