ÉCOLE SUPÉRIEURE D'INGÉNIEURS NUMÉRIQUE ET MATÉRIAUX

Download slides on
https://github.com/dginhac/esirem-itc313

ITC313 - Lesson 02

Fundamentals of programming
Learning C++: from beginner to beyond

Dominique Ginhac

(c) 2020/2021 - dominique.ginhac@u-bourgogne.fr - @dginhac

#ESIREM

Lesson 02

User-defined types for OOP

We will begin with the basics around the variables, starting from fundamental types built into the language to user-defined types.

The main objective is to introduce the fundamentals of Object-Oriented Programming (aka OOP) with the creation of simple classes and objects on concrete examples.

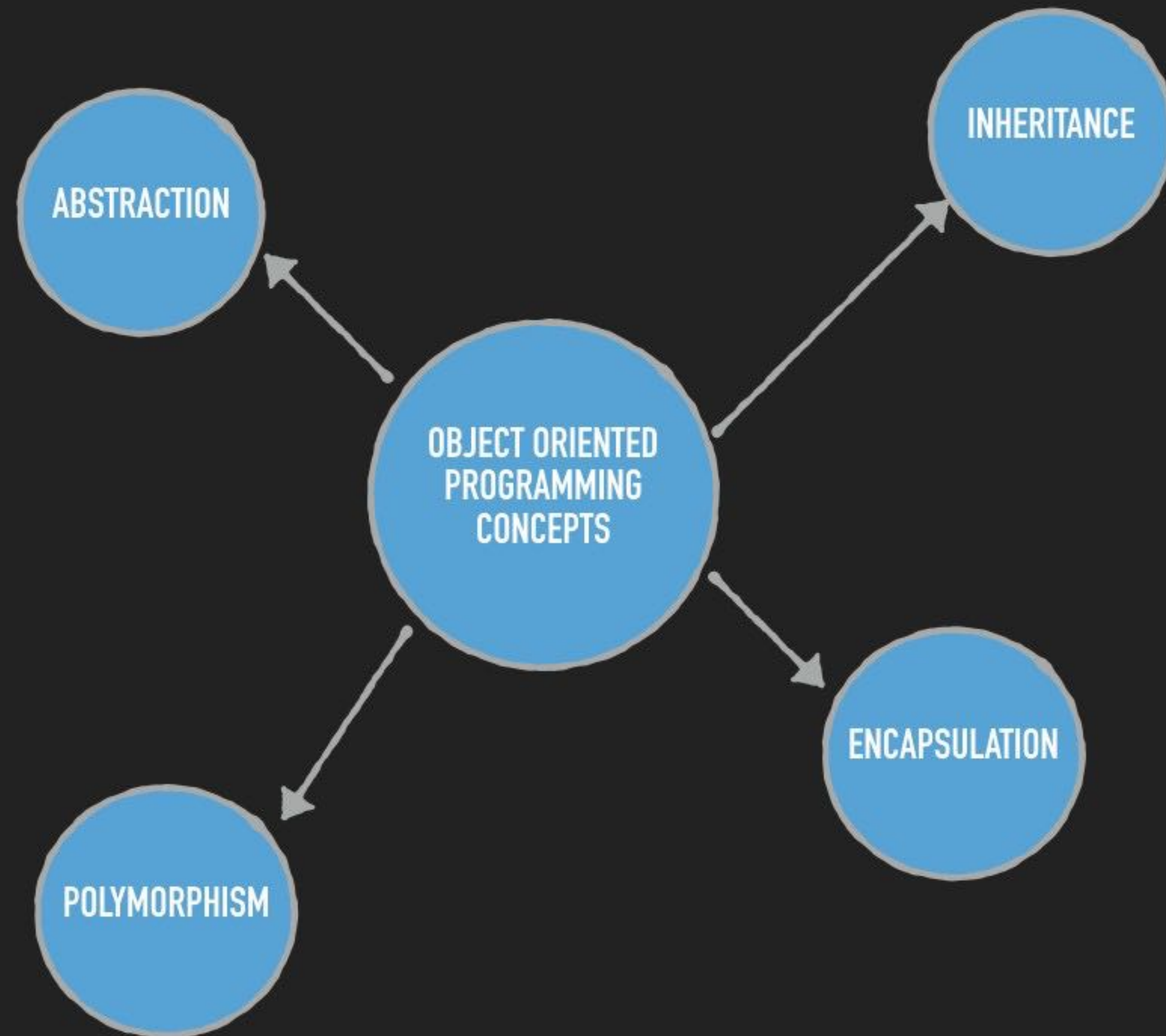
Everything you need to know about OOP

Mechanism of hiding the implementation details from the user, only the functionality will be provided to the user.

In other words, the user will have the information on what the object does instead of how it does it.

Characteristic of being able to assign a different meaning or usage to something in different contexts. Includes Overloading and Overriding.

(from the Greek meaning "having multiple forms")



Mechanism of basing an object upon another object that allows sharing of a set of properties and behaviors and implementation of new functionalities.

Inheritance is a way to reuse once written code again and again.

Also know as Data hiding, Mechanism whereby the implementation details of a class are kept hidden from the user.

Protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by a class.

Language basics - Variables

C++ is a STRONGLY TYPED language

- Variables can hold only certain types of values
- Variables must be declared before they're used and can't change type

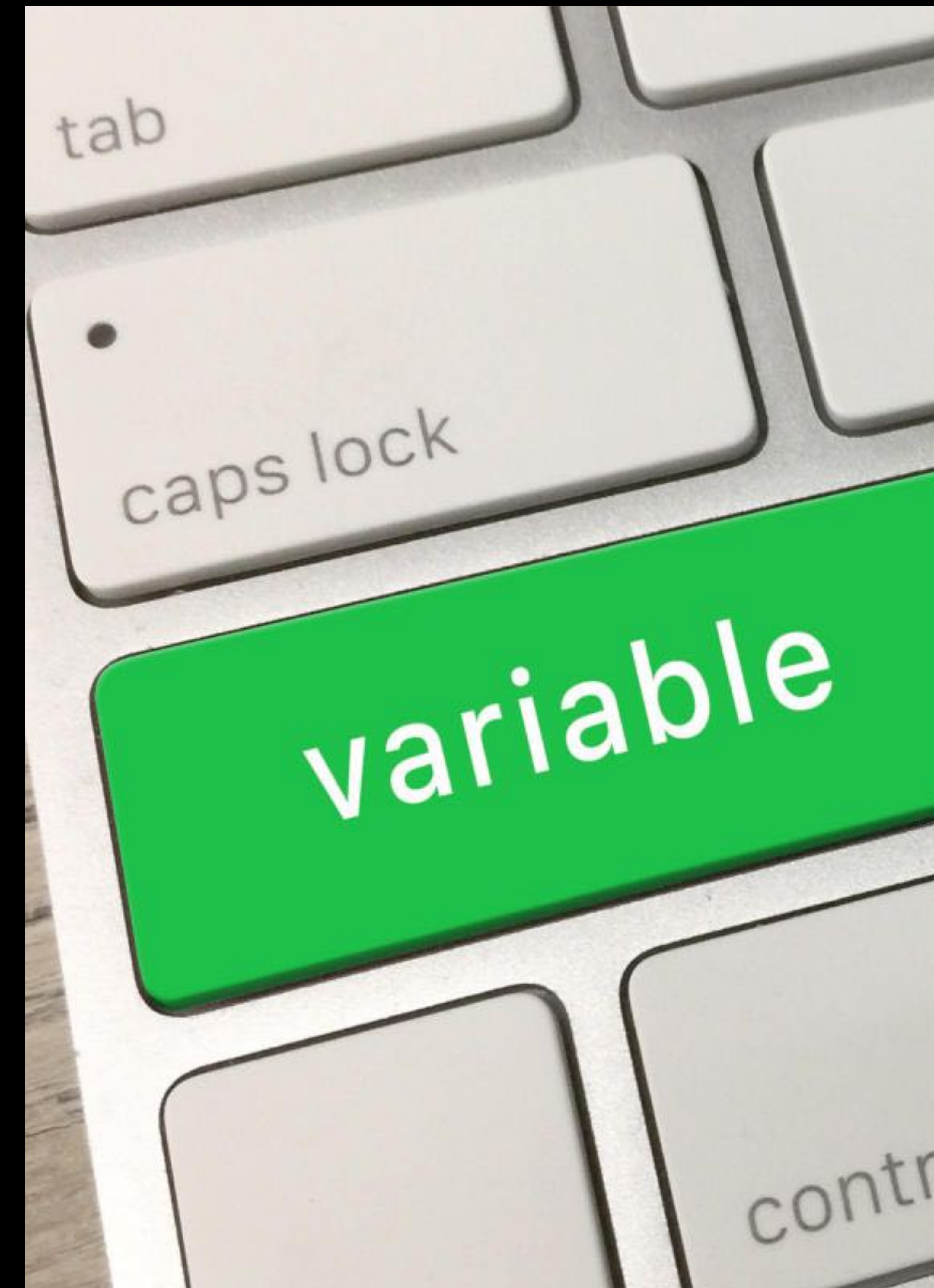
FUNDAMENTAL TYPES built into the language

- Numbers, boolean, single characters

USER-DEFINED TYPES in libraries and in your programs

- Classes, Structures, Enumerations, Unions

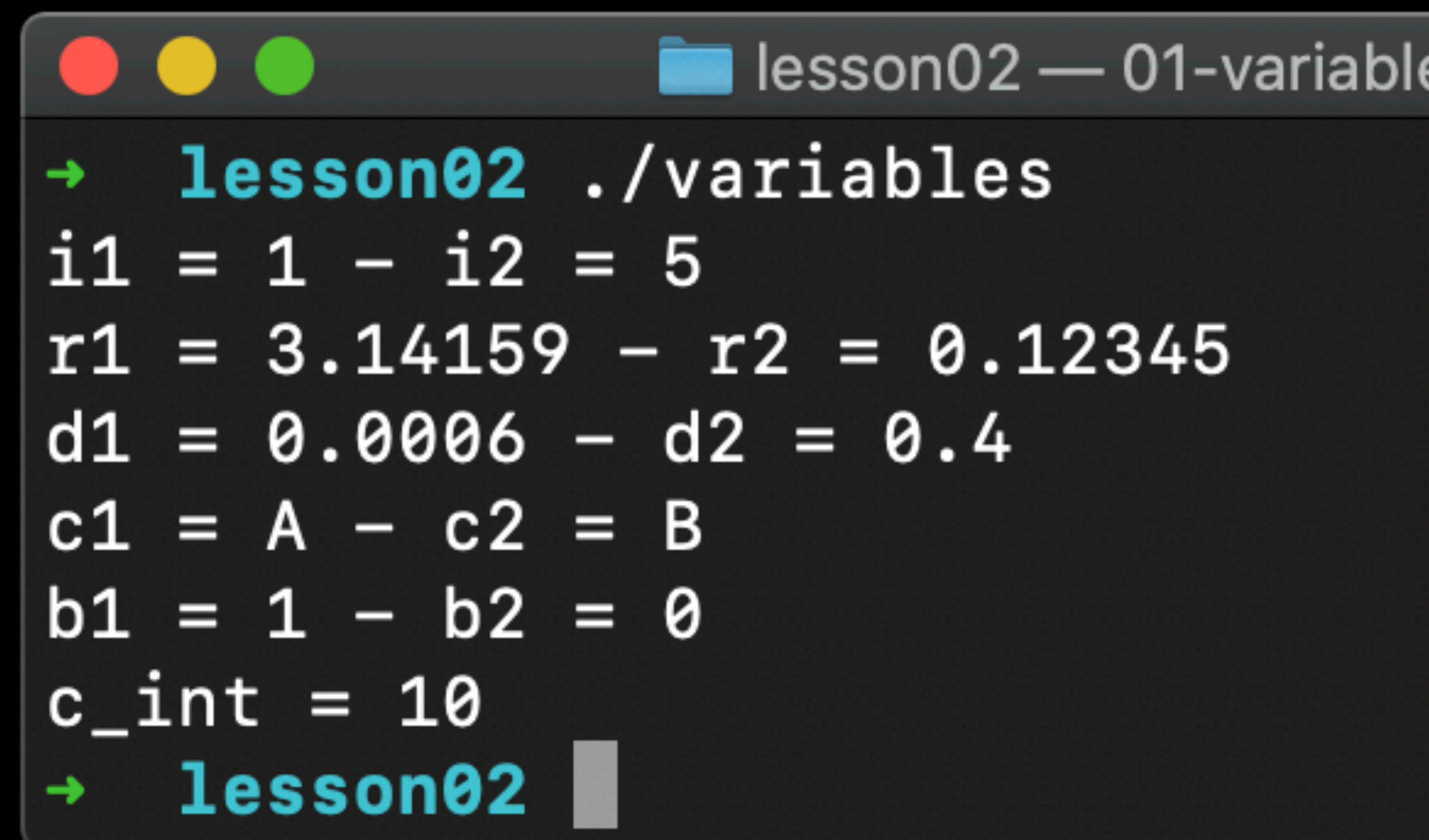
User-defined types are one of the CORE CONCEPTS of C++



Fundamental types

```
type variable_name = init_value; // from C-language
type variable_name(init_value); // constructor init
```

```
// integer type can be short, long, and unsigned
int i1 = 1; int i2(5);
// floating point type (simple or double precision)
float r1 = 3.14159, r2(0.12345);
double d1 = 6e-4, d2(0.4);
// character type
char c1 = 'A', c2('B');
// boolean : true or false
bool b1 = true, b2(false);
// constant can not be modified and
// must be initialised when declared
const int c_int = 10;
```

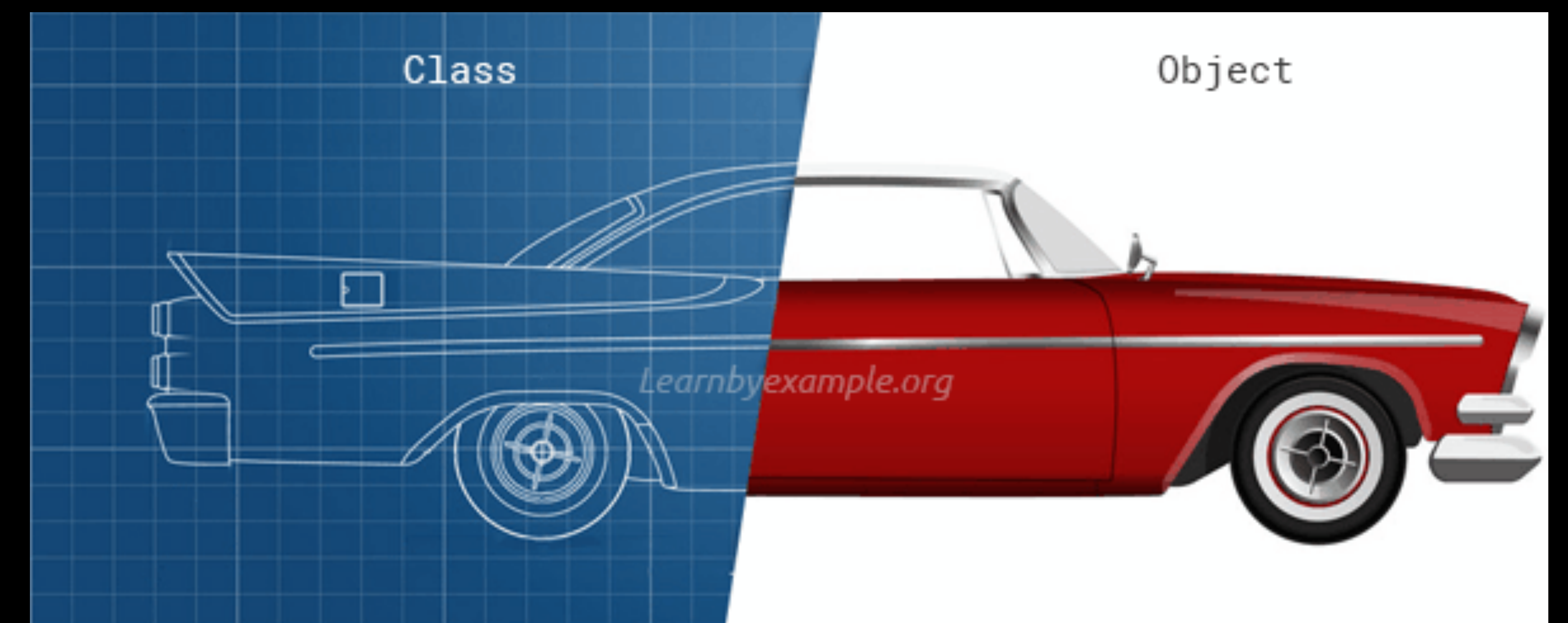


User-defined types

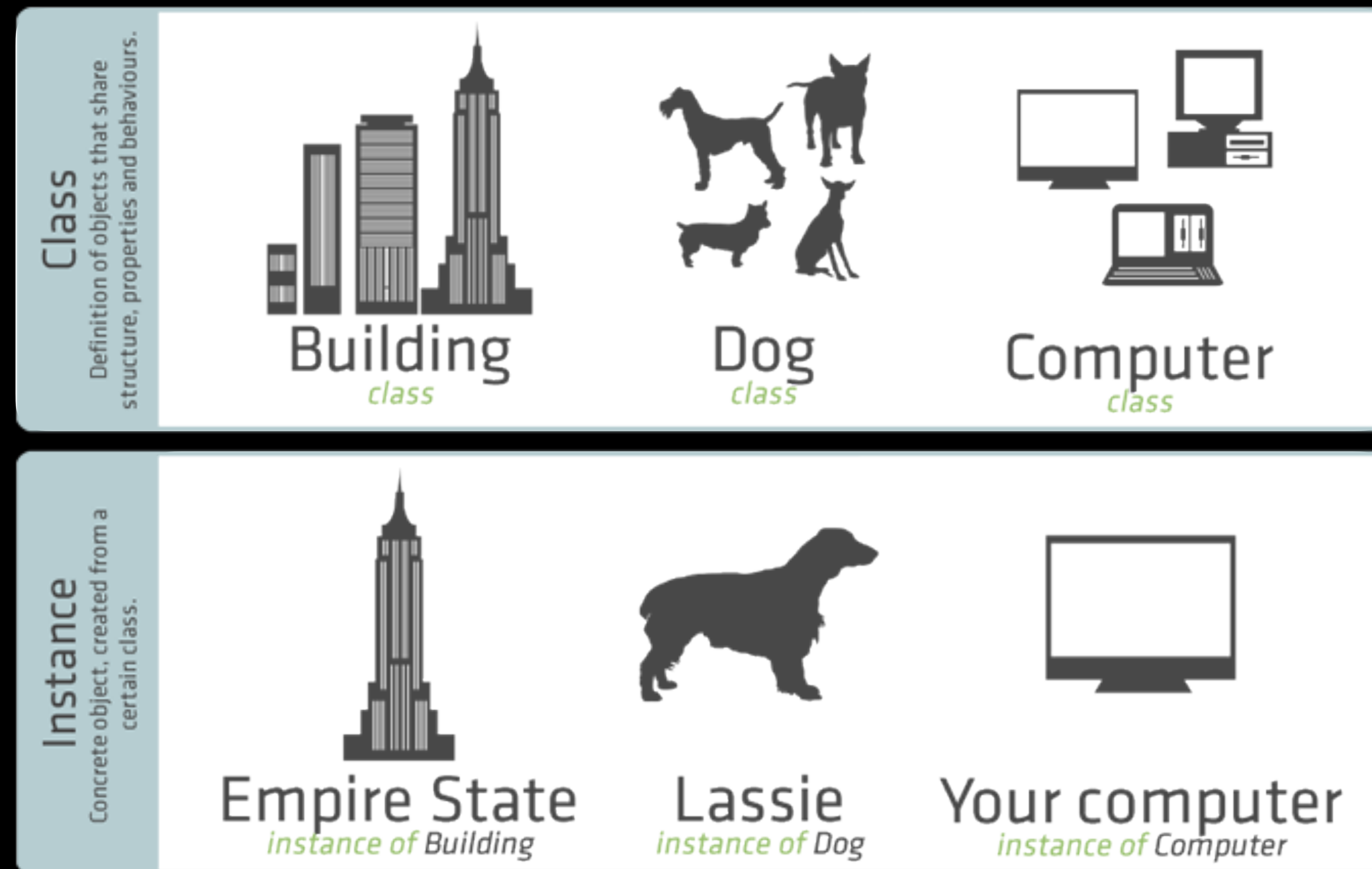
To grapple with complex problems, you need to create PRECISE REPRESENTATIONS of the data that you are talking about.

The closer these representations correspond to reality, the easier it is to write the program.

In C++, the most workable representations are CLASSES and OBJECTS.



A class = a model for a new type of objects represented by a COLLECTION OF VARIABLES combined with a SET OF RELATED FUNCTIONS.



A object = an instance of a class with own copy of member variables and member functions that operate on these member variables

A first basic example of class

Declaration

```
class Person {
public:
  std::string getFullName();

private:
  std::string firstname;
  std::string lastname;
};
```

The keyword class and the name of the class introduce the declaration

Brace brackets surround the contents
Don't forget the final semi-colon

Member variables are generally private and functions are generally public

Functions are only declared here
Definition is elsewhere

Definition

```
std::string Person::getFullName()
{
  return firstname + " " + lastname;
}
```

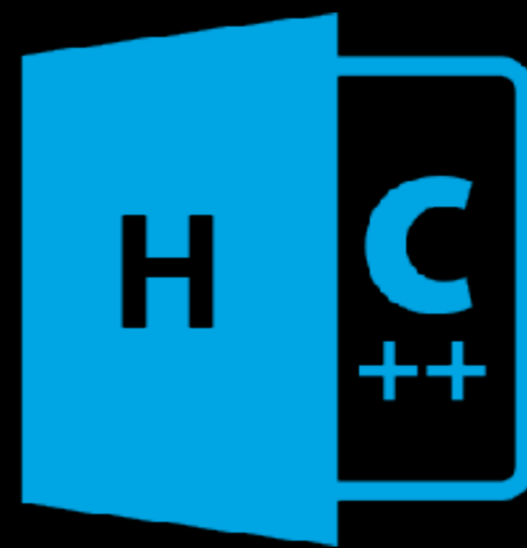
Use fully qualified name

Function access member variable with no special syntax

Interface vs. Code in C++

When writing classes, you must understand separation of

Header file



.h is #included in .cpp

Source file

Interface

Declaration of classes, functions, ...

Implementation

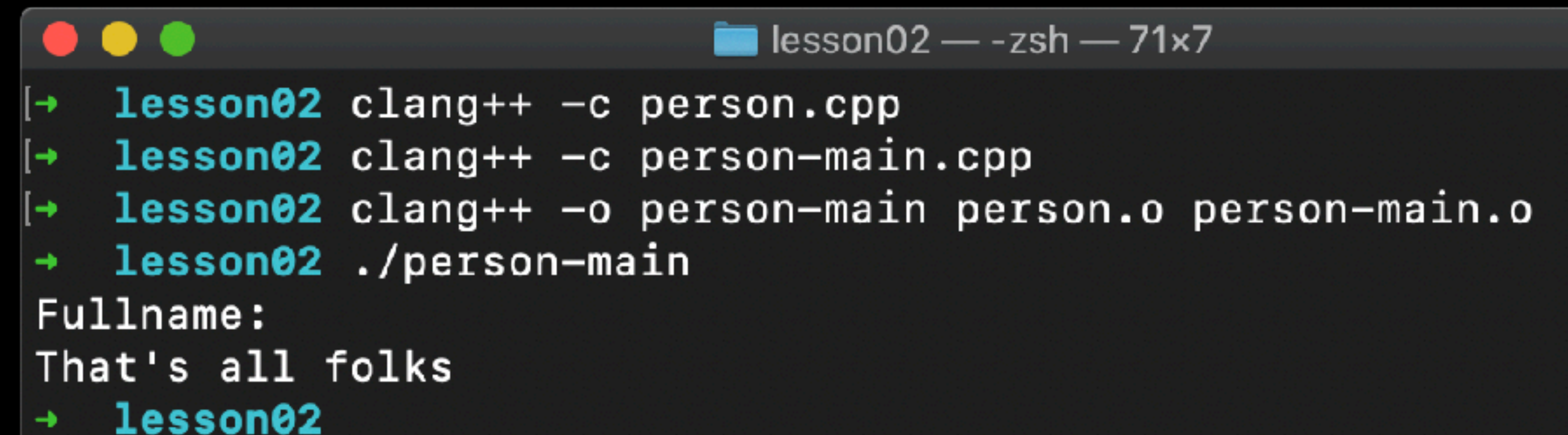
Definition of how it is implemented

So, when you define a new class Foo, you write Foo.h and Foo.cpp and you include Foo.h in each source file that uses Foo objects.

```cpp
// person.h
#include <iostream>

class Person {
public:
  std::string getFullName();

private:
  std::string firstname;
  std::string lastname;
};
```

```cpp
// person-main.cpp
#include <iostream>
#include "person.h"

int main(int argc, char const *argv[]) {
  Person p;
  std::string fullname = p.getFullName();
  std::cout << "Fullname: " << fullname << "\n";
  std::cout << "That's all folks" << std::endl;
  return 0;
}
```

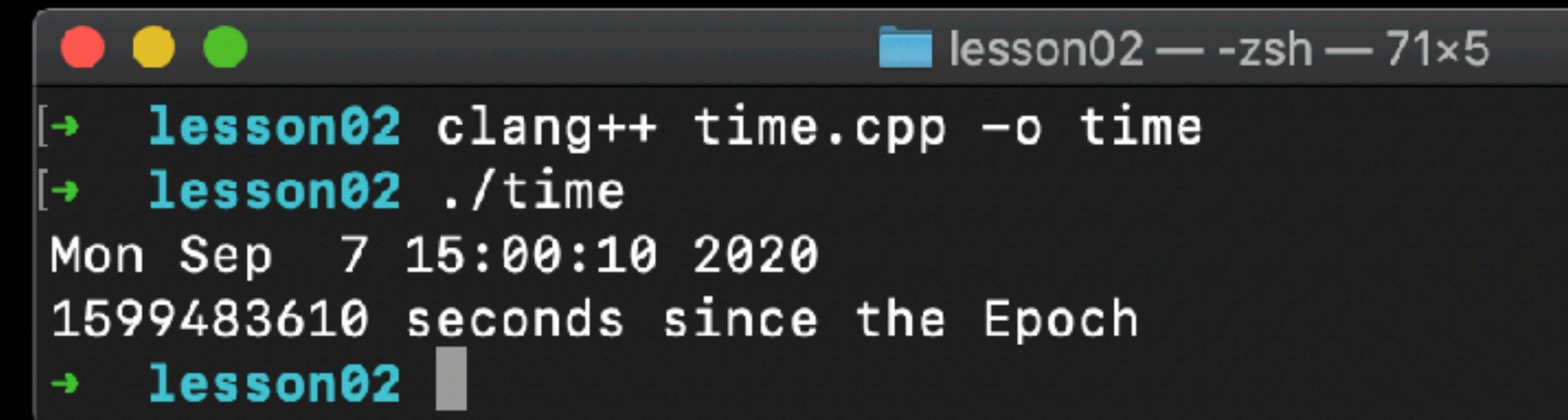
```cpp
// person.cpp
#include <iostream>
#include "person.h"

std::string Person::getFullName() {
  return firstname + " " + lastname;
}
```



A first realistic example of class

C++ does not provide Date objects, only time-related types in ctime library

```
#include <ctime>
#include <iostream>
int main()
{
    std::time_t result = std::time(nullptr);
    std::cout << std::asctime(std::localtime(&result))
              << result << " seconds since the Epoch\n";
}
```



Suppose we need dates for an application

> *What is the star wars day? May, 4 (i.e 05/04)*
> *What is you birthday? May, 26 (i.e 05/26)*
> *It is ?? Days until your next birthday.*

The Date class declaration

We need two variables month and day represented as integers:

```
int _month;                 // Use snake_case for naming variables
int _day;                   // Use “m_” or “_” prefix for member variables
```

Some methods to access and process data stored into the object

```
int getMonth();             // Use camelCase naming for functions
int getDay();               // with explicit names
Date nextDay();
std::string toString();
```

And a constructor to initialise new objects:

```
Date(int month, int day); // Constructor is a specific function dedicated
                          // to the initialization of the new objects
```

How to choose between public and private ?

Private or Public?

```
class Date {
public:
    int _month;
    int _day;
};
```

Public variables are AVAILABLE TO EVERYONE

Imagine a Date class in which everyone can code a month outside the range [1-12]

Imagine a Bank account class in which everyone can change the owner of the account or the balance

```
class Date {
private:
    int _month;
    int _day;
};
```

Private variables are ENCAPSULATED VARIABLES

Encapsulation is the first pillar of OOP, hiding the internal representation of an object from the outside (Compilation error)

Need to write PUBLIC FUNCTIONS to initialize, read and write variables.

date.h

```
/**
 * @Author: Dominique Ginhac <d0m>
 * @Date:   2020-09-07T15:15:52+02:00
 * @Email:  dginhac@u-bourgogne.fr
 * @Project: C++ Programming - ESIREM 3A IT
 * @Last modified by:   d0m
 * @Last modified time: 2020-09-07T17:16:40+02:00
 */

#ifndef DATE_H
#define DATE_H
#include <string>

class Date {
public:
   Date (int month, int day);
   int getMonth();
   int getDay();
   Date nextDay();
   std::string toString();
private:
   int _month;
   int _day;
};
#endif // DATE_H
```

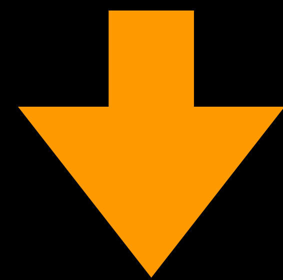
Each new file (.h or .cpp) begins with an HEADER including a detailed description in comments.

#ifndef #define #endif is an HEADER GUARD, i.e. a protection to avoid the problem of multiple inclusion in case multiple .cpp files include this .h

Public constructors, getters and setters functions must be defined to access private variables

Getters

```
int Date::getMonth() {
    return _month;
}
int Date::getDay() {
    return _day;
}
```



```
int Date::getMonth() const {
    return _month;
}
int Date::getDay() const{
    return _day;
}
```

For each member variable, a get method is defined to read its value.

Naming with Obj-C style (int Date::month()) or java style (int Date::getMonth())

No input argument and return type of the get method is the same as the one of the member variable.

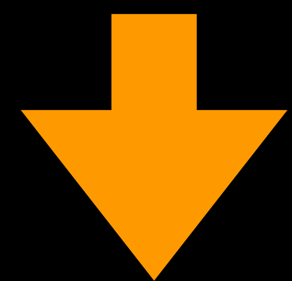
Getters should always be marked as const because they only read the member variables.

Always mark functions as const unless you can't.

It is a very useful information for the compiler which can do optimizations and for humans who read and understand your code.

Setters

```
void Date::setMonth (int month)
{
    _month = month;
}
```



```
void Date::setMonth(int month) {
    assert((month >=1) &&
           (month<=12));
    _month = month;
}
```

If a member variable needs to be modified after its initialization, a setter must be defined.

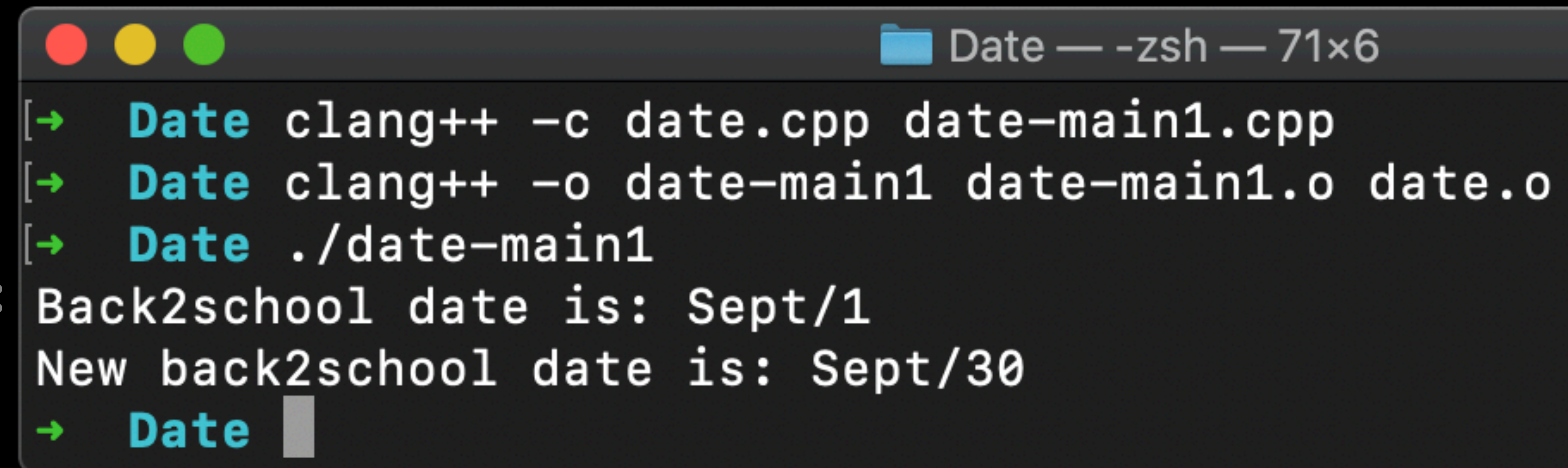
Input argument has same type than member variable, no return type.

Setters often do value checking to ensure that the new data is in the proper range.

See https://bit.ly/2PxmNkr

Using functions

```
/**
 * @Author: Dominique Ginhac <d0m>
 * @Date:   2019-10-15T09:47:29+02:00
 * @Email:  dginhac@u-bourgogne.fr
 * @Project: C++ Programming - ESIREM 3A IT
 * @Last modified by:   d0m
 * @Last modified time: 2020-09-07T23:23:01+02:
 */

#include <iostream>
#include "date.h"

int main() {
   Date back2school(9,1);
   std::cout << "Back2school date is: " << back2school.toString() << std::endl;
   back2school.setDay(30);
   std::cout << "New back2school date is: " << back2school.toString() << std::endl;
   return 0;
}
```



Constructors

Special class functions

1. Allocate STORAGE to the new created objects.
2. Perform INITIALIZATION of each new object.

Automatically CALLED when a NEW OBJECT is created.

Constructors

Constructors have the same name as the class itself

They can take arguments that are used to initialize the member variables.

No return type.

When you do not specify any constructor in a class, compiler generates a default constructor and uses it with each new object.

Default Constructor

`Date d;`

Parametrized Constructor

`Date d(12,25);`

Copy Constructor

`Date d = a;`

Default Constructor

date.h

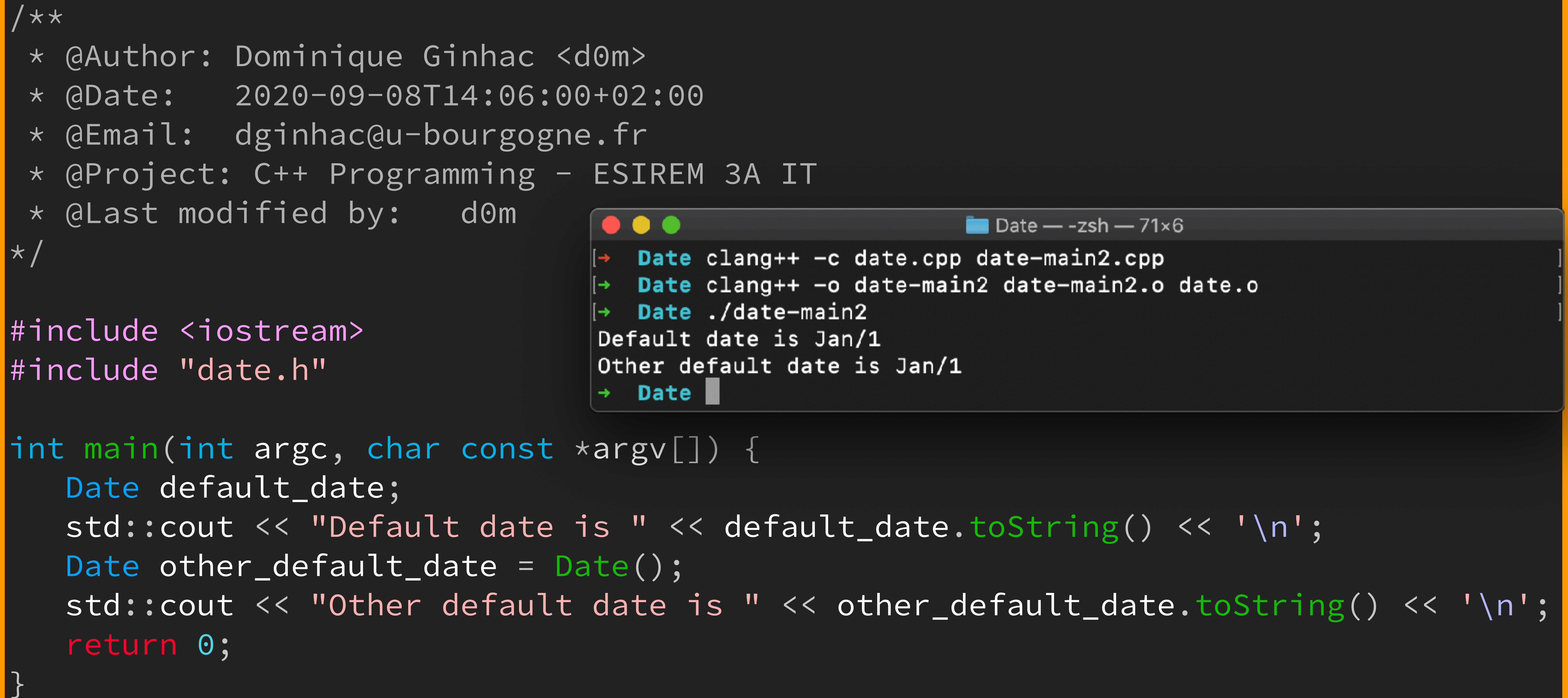
```
class Date {
public:
    Date();
  …
private:
    int _month;
    int _day;
};
```

date.cpp

```
Date::Date() {
  _month=1;
  _day=1;
}
```

The most basic constructor with
no input parameter.

Default Constructor



Default Constructor

```
Date d;
```

Parametrized Constructor

```
Date d(12,25);
```

Copy Constructor

```
Date d = a;
```

Parametrized Constructor

date.h

```
class Date {
public:
  Date(int month, int day);
   …
private:
   int _month;
   int _day;
};
```

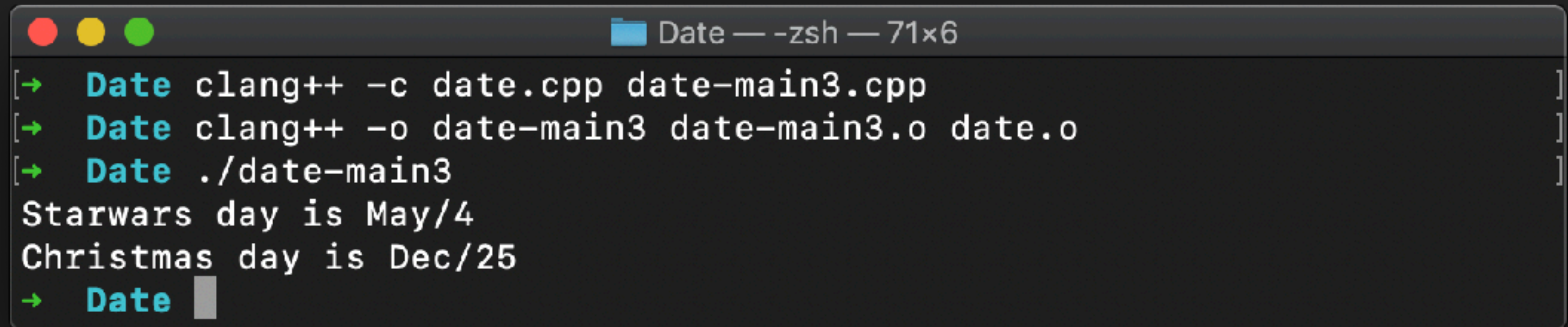
date.cpp

```
Date::Date(int month, int day) {
   _month = month;
   _day = day;
}
```

It's a customized constructor with parameters.
Each new object can be initialized with specific values passed as arguments of the constructor.

Parametrized Constructor

```cpp
/**
 * @Author: Dominique Ginhac <d0m>
 * @Date:   2020-09-08T14:06:00+02:00
 * @Email:  dginhac@u-bourgogne.fr
 * @Project: C++ Programming - ESIREM 3A IT
 * @Last modified by:   d0m
*/

#include <iostream>
#include "date.h"

int main(int argc, char const *argv[]) {
   // May the fourth be with you!
   Date starwars(5,4);
   std::cout << "Starwars day is " << starwars.toString() << std::endl;
   Date christmas = Date(12,25);
   std::cout << "Christmas day is " << christmas.toString() << std::endl;
   return 0;
}
```



Default specifier

If a class is defined with a parametrized constructor, the compiler will not generate a default constructor

If we create a new object

```
Date a_day;
```

the compiler will complain that we have no default constructor

We can force the compiler make the one for us by using the default specifier (C++11 extension)

```
class Date {
public:
    Date(int month, int day);
    Date() = default;
    …
private:
    int _month, _day;
};
```

Reducing #Constructors

```
class Date {
public:
   Date();
   Date(int month, int day);
   …
private:
   int _month, _day;
};
```

```
Date::Date() {
   _month=1;
   _day=1;
}
Date::Date(int month, int day) {
   _month = month;
   _day = day;
}
```

Constructors code is often somewhat redundant

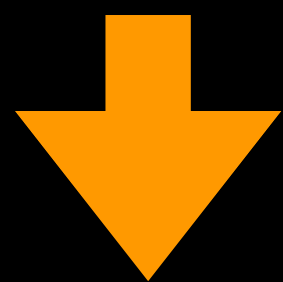
Default and parametrized constructors are overloaded functions (i.e. the same name, but different and unique parameters !)

Using default values expresses that there is really just one Constructor

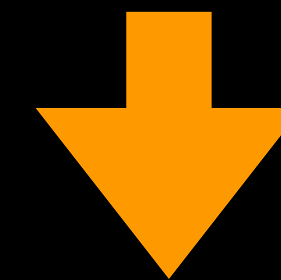
Reducing #Constructors

```
class Date {
public:
    Date();
    Date(int month, int day);
    …
private:
    int m_month, m_day;
};
```

```
Date::Date() {
    m_month=1;
    m_day=1;
}
Date::Date(int month, int day) {
    m_month = month;
    m_day = day;
}
```



Default value/s are passed to argument/s in the function prototype.



```
class Date {
 public:
    Date(int month=1, int day=1);
    …
};
```

```
Date::Date(int month, int day) {
     _month = month;
     _day = day;
 }
```

Constructor member initializer lists

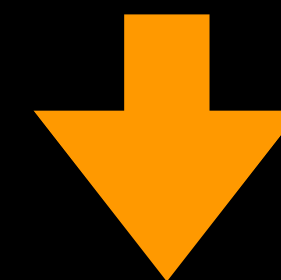
Our class member data are initialized using the assignment operator.

It works perfectly for most of the types but does not work in some specific cases including const, references, …

C++ provides “Member initializer list” starting with a colon after the list of parameters

Can be used with any type, more concise and more efficient

```
Date::Date(int month, int day) {
    _month = month;
    _day = day;
}
```



```
Date::Date(int month, int day) :
m_month(month), m_day(day) {
    //empty body
}
```

Default Constructor

`Date d;`

Parametrized Constructor

`Date d(12,25);`

Copy Constructor

`Date d = a;`

Copy Constructor

Copy Constructor is used to declare and initialize an object from another object

1. When an object is constructed based on another object of the same class.
2. When an object is returned by value.
3. When an object is passed to a function by value as an argument.

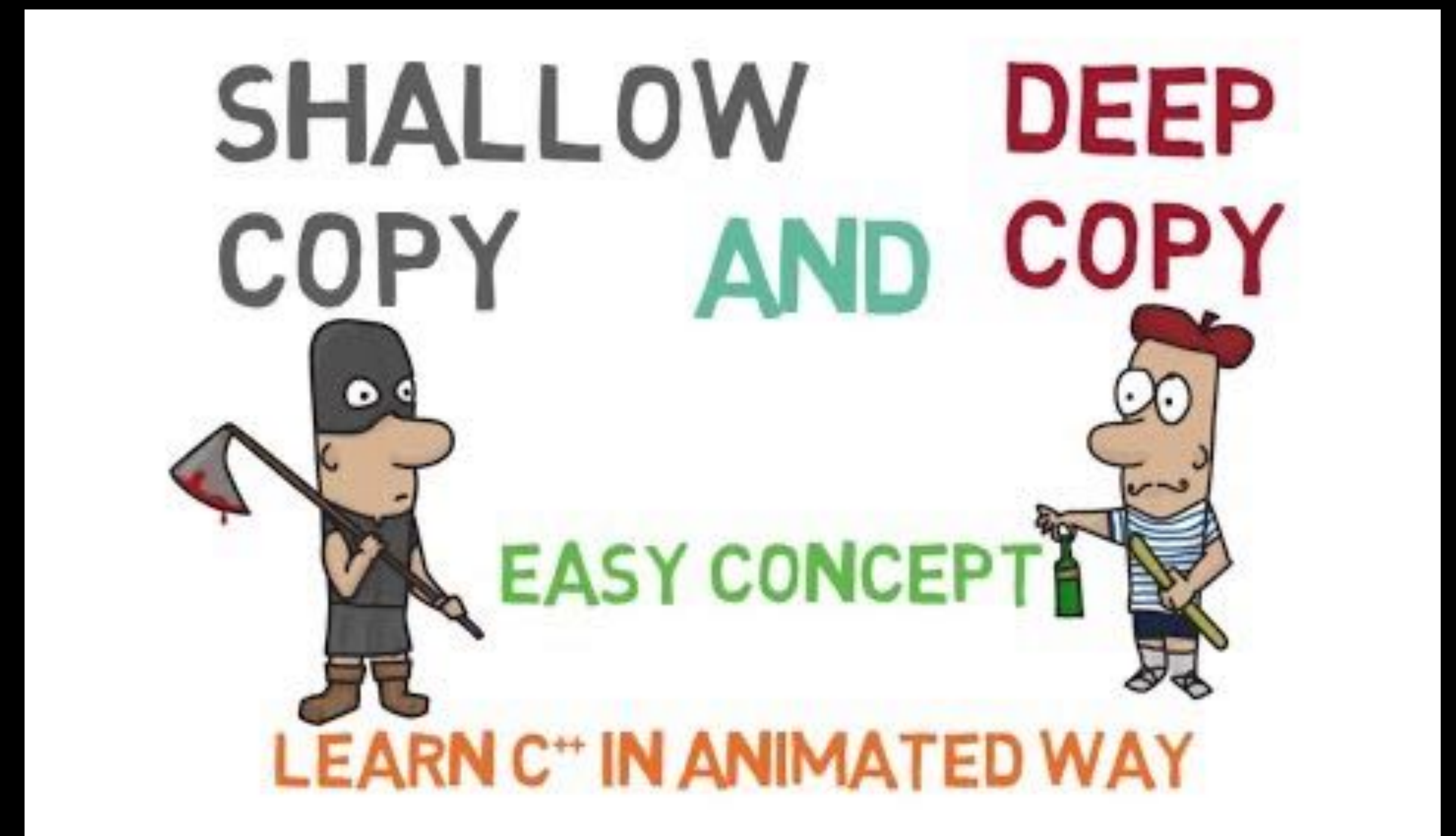
```
Date starwars(5,4);
Date other_starwars_day = starwars;
Date another_starwars(starwars);

Date tomorrow = today.nextDay();

Date pi_day(3,14);
bool b = starwars.before(pi_day);
```

Do we need writing specific code for Copy constructor?

When is used-defined copy constructor needed?

If copy constructor is not defined, the C++ compiler creates a default copy constructor making a memberwise copy (aka a shallow copy).

We only need to define our own copy constructor (aka a deep copy) only if an object has pointers or any runtime allocation of resources.



Upcoming lesson

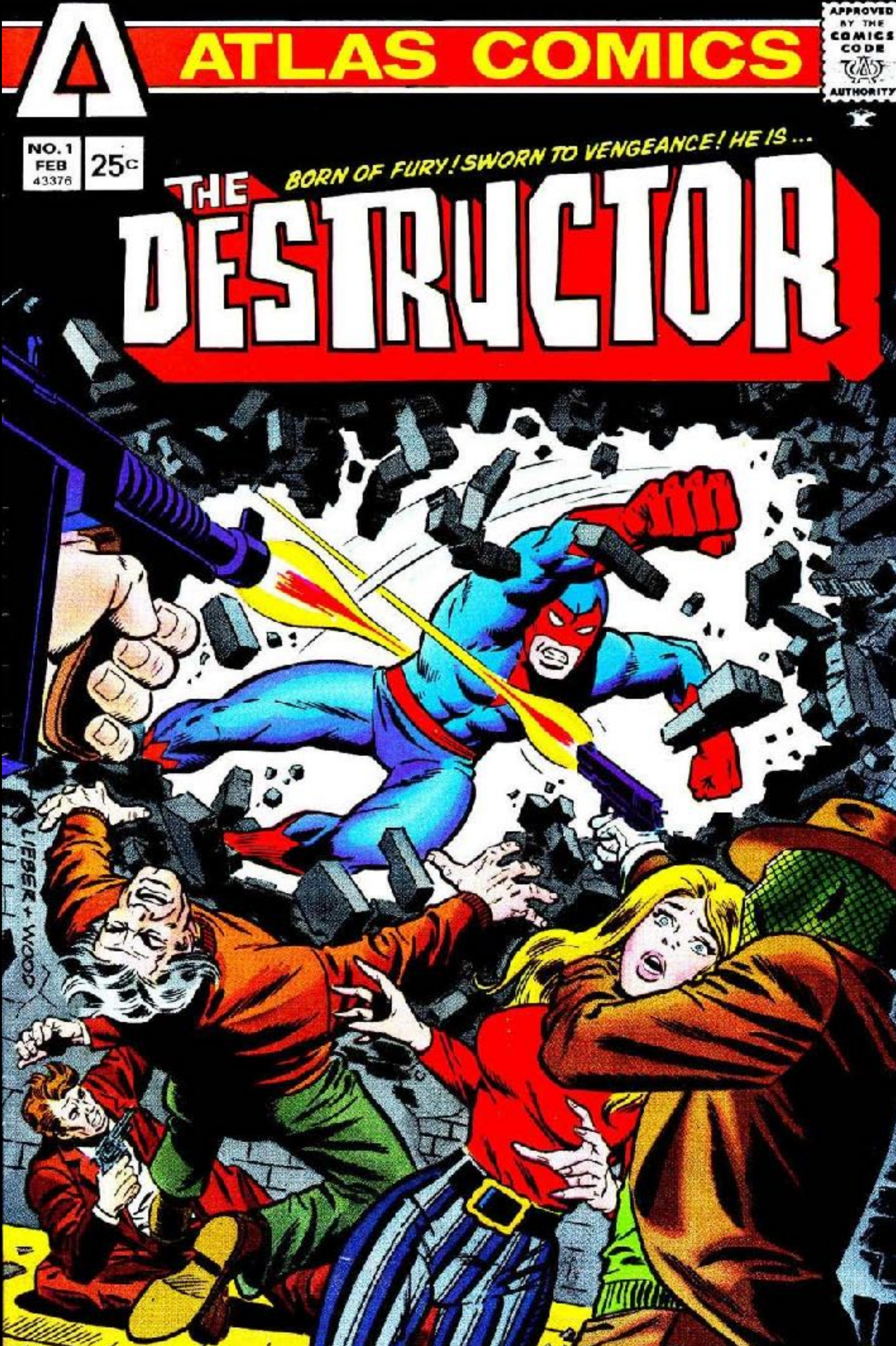
https://www.youtube.com/watch?v=uEz8ZLkw44o&list=PLrKBFf87Cy9BVBAZ1Y-6OIzz0KTU4Qw6L

What about Destructors?

Special MEMBER FUNCTION that works just opposite to constructor

Useful for releasing resources

Automatically called when objects become out of scope



Destructors rules

Destructor name should begin with tilde sign(~) and must match class name.

Only ONE destructor in a class.

No parameter, unlike constructors

No return type, just like constructors.

When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.

Destructor of Date

DECLARING ~Date() in date.h

```
class Date {
public:
    Date (int month=1, int day=1);
    ~Date();
private:
    int _month;
    int _day;
```

IMPLEMENTING ~Date() in date.cpp

```
Date::~Date() {
    std::cout << "Destructor" << '\n';
}
```

Using destructors

Destructors are automatically called when objects become out of scope. The scope of a variable is defined as the code within which the variable is declared or worked with

```
void scopeFunc(Date d) { // Free function - out of any class
    std::cout << "func input: " << d.toString() << '\n';
    d.setMonth(8);
    std::cout << "func output: " << d.toString() << '\n';
}
int main() {
    Date d(3,14);
    std::cout << "main 1: " << d.toString()
    scopeFunc(d);
    std::cout << "main 2: " << d.toString()
    {
        Date d(5,4);
        std::cout << "block: " << d.toString(
    }
    std::cout << "main 3: " << d.toString()
}
```

Date — -zsh — 80×13

```
→ Date clang++ -c date.cpp date-out-of-scope.cpp
→ Date clang++ -o date-out-of-scope date.o date-out
→ Date ./date-out-of-scope
main 1: March/14
func input: March/14
func output: Aug/14
  Destructor: Aug/14
main 2: March/14
block: May/4
  Destructor: May/4
main 3: March/14
  Destructor: March/14
→ Date
```

The final date class

```cpp
#ifndef DATE_H
#define DATE_H
#include <string>

class Date {
public:
   Date (int month=1, int day=1);
   ~Date();
   int getMonth() const;
   int getDay() const;
   void setMonth(int month);
   void setDay(int day);
   Date nextDay();
   std::string toString();
private:
   int _month;
   int _day;
};
#endif // DATE_H
```

```cpp
#include “date.h"
#include <iostream>

Date::Date(int month, int day) {
   _month = month;
   _day = day;
}
Date::~Date() {
   std::cout << "  Destructor: " << toString();
}
int Date::getMonth() const {
   return _month;
}
int Date::getDay() const {
   return _day;
}
void Date::setMonth(int month) {
    _month = month;
}
```

Minimal version of the class with no value checking

Check a date

```
bool Date::checkDate(int month, int day) const {
    bool status=true;

    if ((month == 1 || month == 3 || month == 5 || month == 7
    || month == 8 || month == 10 || month == 12) && ( day>31 || day<1) ) {
        status = false;
    }
    else if ((month == 4 || month == 6 || month == 9 || month == 11)
    && (day>30 || day<1) ) {
        status = false;
    }
    else if ((month == 2) && (day>28 || day<1) ) {
        status = false;
    }
    if ((month < 1) || (month > 12)) {
        status = false;
    }
    return status;
}
```

|| = OR

&& = AND

Safer methods

```
Date::Date(int month, int day) {
    bool status = checkDate(month, day);
    assert(status==true && "Date is not valid");
    _month = month;
    _day = day;
}

void Date::setMonth(int month) {
    assert((month >=1) && (month<=12) && “Month
must be between 1 and 12");
    _month = month;
}

void Date::setDay(int day) {
    bool status = checkDate(_month, day);
    assert(status == true && "Day is not valid");
    _day = day;
}
```

Constructor and Setters do VALUE CHECKING to ensure that the data is in the proper range.

Exceptions and Assertions

Upcoming lesson

Testing invalid Date objects

```
int main() {
    Date not_valid(4,31);
    std::cout << “date  is " << not_valid.toString() << std::endl;
    return 0;
}
```

```
Date — -zsh — 80×6
→ Date clang++ -std=c++14 date.cpp date-main4.cpp -o date-main4
→ Date ./date-main4
Assertion failed: (status==true && "Date is not valid"), function Date, file dat
e.cpp, line 34.
[1]    88077 abort      ./date-main4
→ Date
```

```
int main() {
    Date valid(4,30);
    std::cout << "date is " << valid.toString() << std::endl;
    valid.setDay(31);
    std::cout << "date is " << valid.toString() << std::endl;
    return 0;
}
```

```
Date — -zsh — 80×7
→ Date clang++ -std=c++14 date.cpp date-main5.cpp -o date-main5
→ Date ./date-main5
date is April/30
Assertion failed: (status == true && "Day is not valid"), function setDay, file
date.cpp, line 67.
[1]    88117 abort      ./date-main5
→ Date
```

#02

Take Home Message

ENCAPSULATION and ABSTRACTION are the two first key concepts of OOP.

By declaring class attributes as private, ENCAPSULATION ensures that "sensitive" data is hidden from users.

Data ABSTRACTION refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Other User-defined types

C++ does not only offer user-defined types
as classes but also

A first example of struct

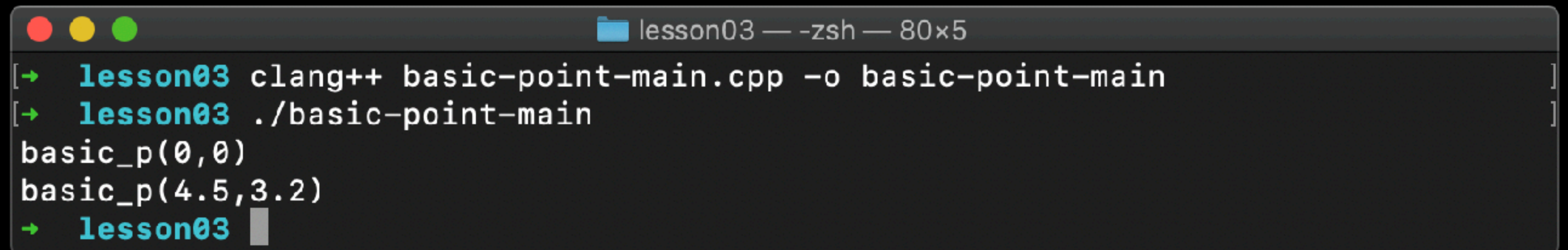
Used mainly for plain old data (i.e. a BUNDLE that just stores data with little logic)

Default access is public for backwards compatibility with C whereas default access is private for class

```
#ifndef POINT_H
#define POINT_H

struct Point {
   float x; // x and y are public
   float y; // No need to write getters/setters
  };
#endif // POINT_H
```

Using struct Point

```
#include <iostream>
#include "point.h"

int main(int argc, char const *argv[]) {
    // declaration using struct Point as in C language
    // In C++, We can also declare: Point basic_p
    struct Point basic_p;
    std::cout << "basic_p(" << basic_p.x << "," << basic_p.y << ")"<< '\n';
    basic_p.x = 4.5;       // x is public
    basic_p.y = 3.2;       // x is public
    std::cout << "basic_p(" << basic_p.x << "," << basic_p.y << ")"<< '\n';
    return 0;
}
```



A less basic Struct

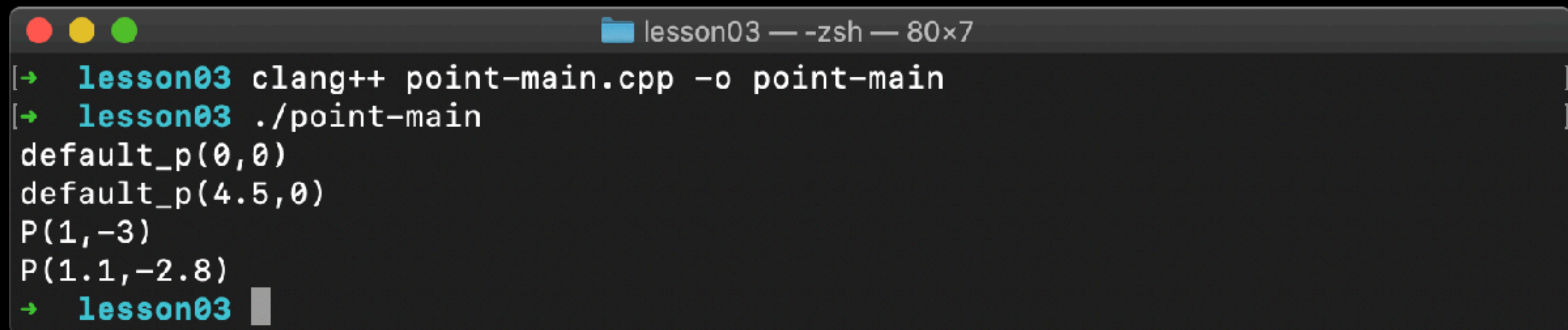
Struct can also have constructors, functions and destructor as classes

```
#ifndef POINT_H
#define POINT_H

struct Point {
   float x; // x and y are public
   float y; // No need to write getters/setters
   Point(float x=0.0, float y=0.0) : x(x), y(y) {}
   void move(float dx, float dy) {
      x+= dx;
      y+= dy;
   }
};
#endif // POINT_H
```

NO real DIFFERENCE between class and structure

Using struct Point

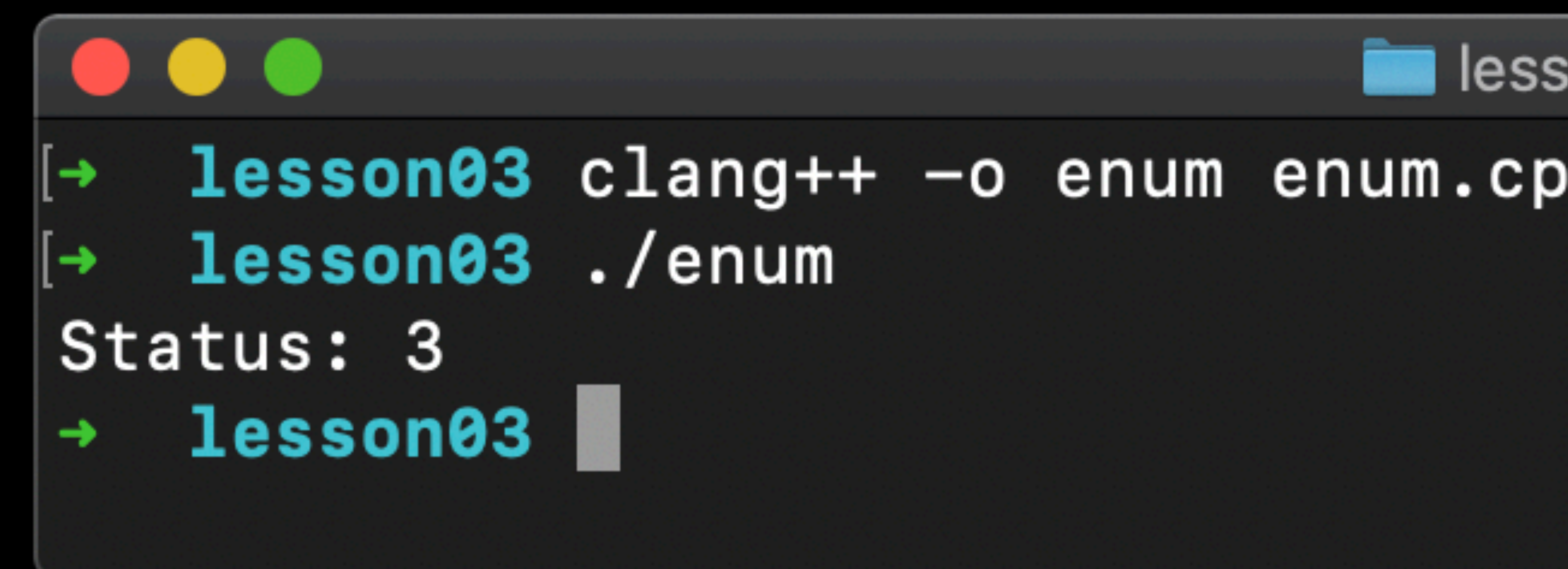
```
#include <iostream>
#include "point.h"

int main(int argc, char const *argv[]) {
    struct Point default_p; //declaration using struct Point as in C language
    std::cout << "default_p(" << default_p.x << "," << default_p.y << ")"<< '\n';
    default_p.x = 4.5;       // x is public
    std::cout << "default_p(" << default_p.x << "," << default_p.y << ")"<< '\n';
    Point p(1.0, -3.0);      // declaration using Point, only valid in C++
    std::cout << "P(" << p.x << "," << p.y << ")"<< '\n';
    p.move(0.1, 0.2);        // call of the move function
    std::cout << "P(" << p.x << "," << p.y << ")"<< '\n';
    return 0;
}
```



Enum

User-defined data type which can be assigned some limited values that are defined by the programmer at the time of declaring the enumerated type.

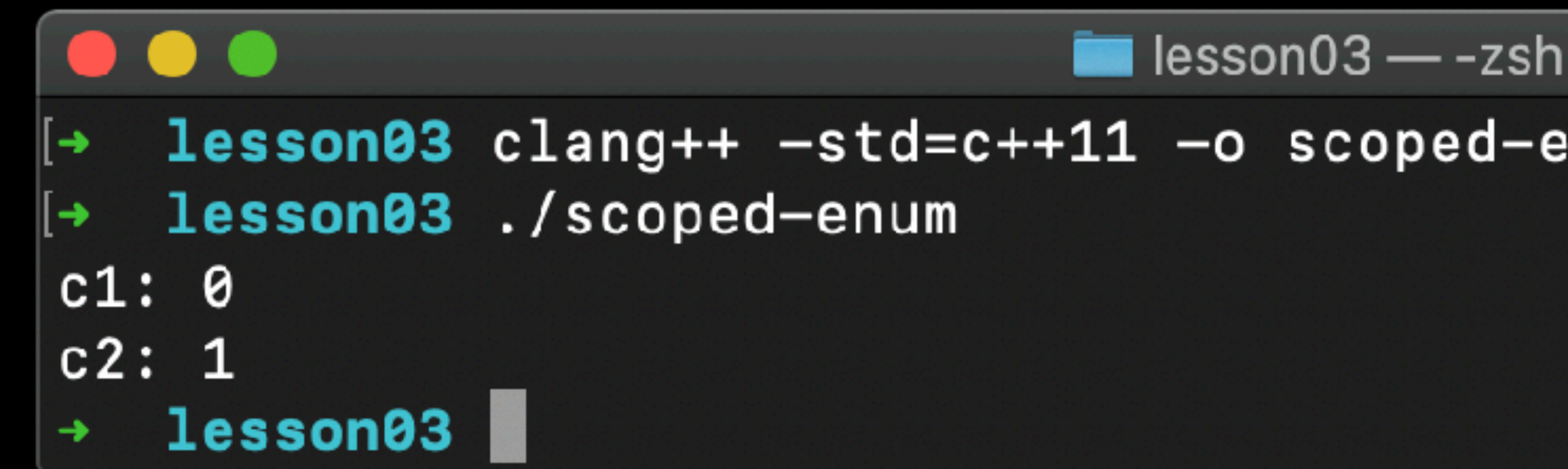
```
enum Status { // Braces surround the entries
    Pending,    // a comma-separated
    Urgent,     // set of constants
    Delayed,    // (integers) with unique names
    Cancelled,
    Done        // No comma after the last one
}; // Do not forget the semi-colon

int main(int argc, char const *argv[]) {
    Status status; // or enum Status status;
    status = Cancelled;
    std::cout << "Status: " << status << '\n';
    return 0;
}
```



Scoped Enum

C++11 has introduced enum classes (also called scoped enum), that makes enumerations both strongly typed and strongly scoped.

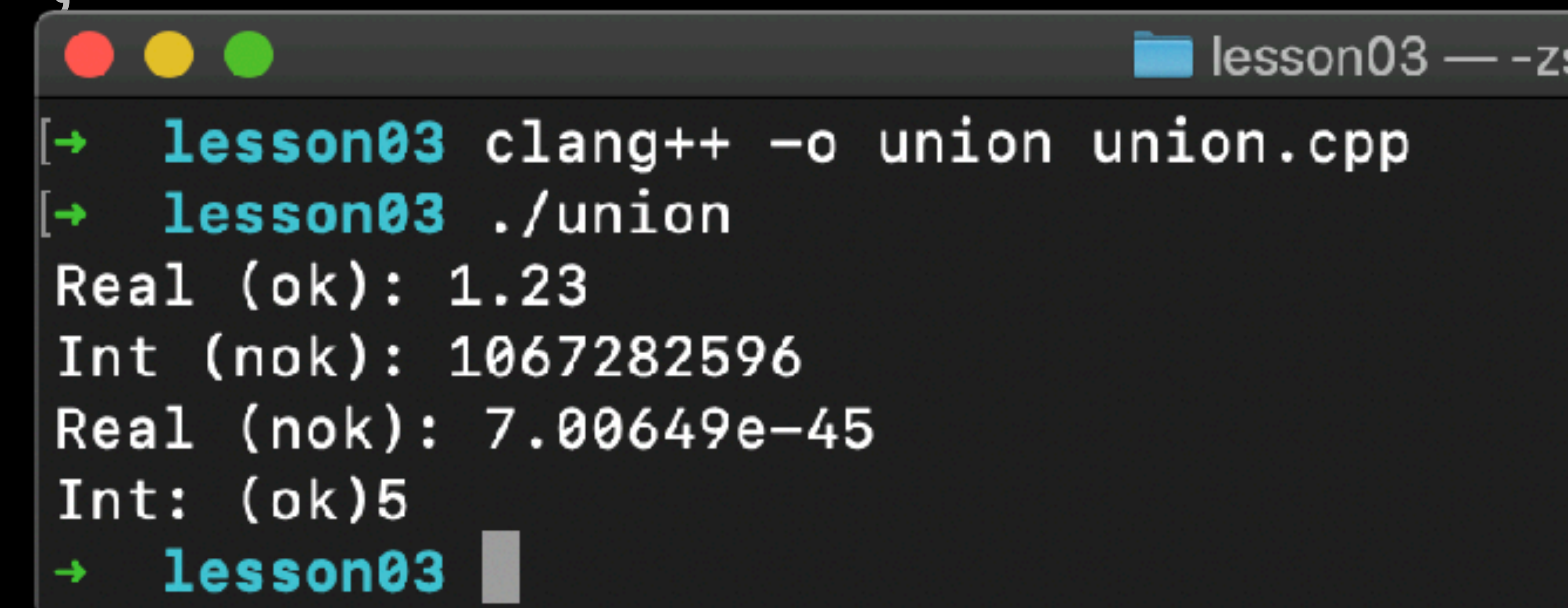
```
enum class Colorset1 { // Definition is similar to standard enum
    Red, Green, Blue     // Use enum class instead
};
enum class Colorset2 {
    Orange, Red, Yellow  // Names don't have to be unique
};
int main(int argc, char const *argv[]) {
    Colorset1 c1 = Colorset1::Red; // Use fully qualified name
    Colorset2 c2 = Colorset2::Red;
    // No implicit conversion to integer
    std::cout << "c1: " << static_cast<int>(c1)
    std::cout << "c2: " << static_cast<int>(c2)
    return 0;
}
```



Union

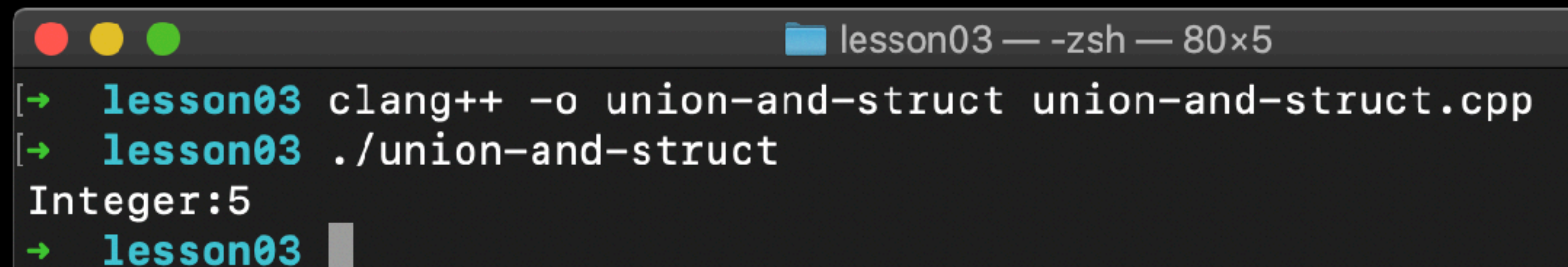
A union is a special class type that can hold only one of its non-static data members at a time.

```
union Data {          // The purpose of union is to save memory
      float real;     // by using the same memory region for storing
      int integer;    // different objects at different times.
};
int main() {
   Data d;
   d.real = 1.23;
   std::cout << "Real (ok): " << d.real << '\n';
   std::cout << "Int (nok): " << d.integer << '\n';
   d.integer=5;
   std::cout << "Real (nok): " << d.real << '\n';
   std::cout << "Int: (ok)" << d.integer << '\n';
   return 0;
}
```



```cpp
#define REAL 1
#define INTEGER 2
#define CHAR 3

union Data {
    float real;
    int integer;
    char letter;
};
struct FullData {
   short type;
   Data data;
   void display();
};

int main(int argc, char const *argv[]) {
   FullData my_var;
   my_var.type = INTEGER;
   my_var.data.integer = 5;
   my_var.display();
   return 0;
}
```

```cpp
void FullData::display() {
   switch (type) {
      case REAL:
         std::cout << "Real: " << data.real << '\n';
      break;
      case INTEGER:
         std::cout << "Integer: " << data.integer <<
      break;
      case CHAR:
         std::cout << "Char: " << data.letter << '\n
      break;
   }
}
```



Namespaces

Imagine an application written by different programmers,

The same names may be used fr different things (classes, functions, ...)

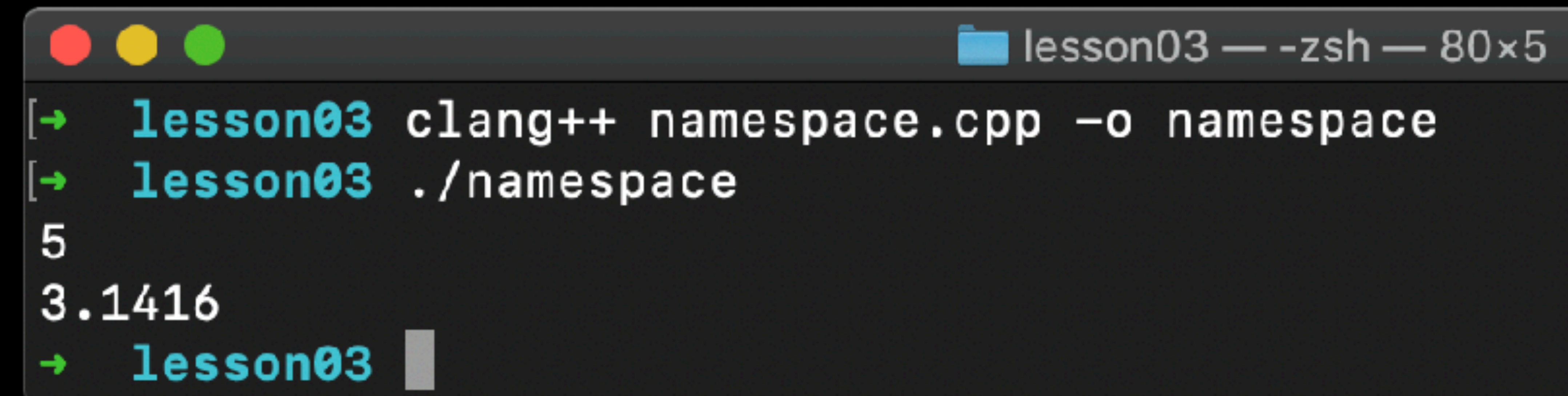
NAMESPACES provide an elegant method for preventing NAME CONFLICTS in large projects

Namespaces

A namespace is a CONTAINER for identifiers.

It puts the names of its members in a DISTINCT SPACE so that they don't conflict with the names in other namespaces or global namespace

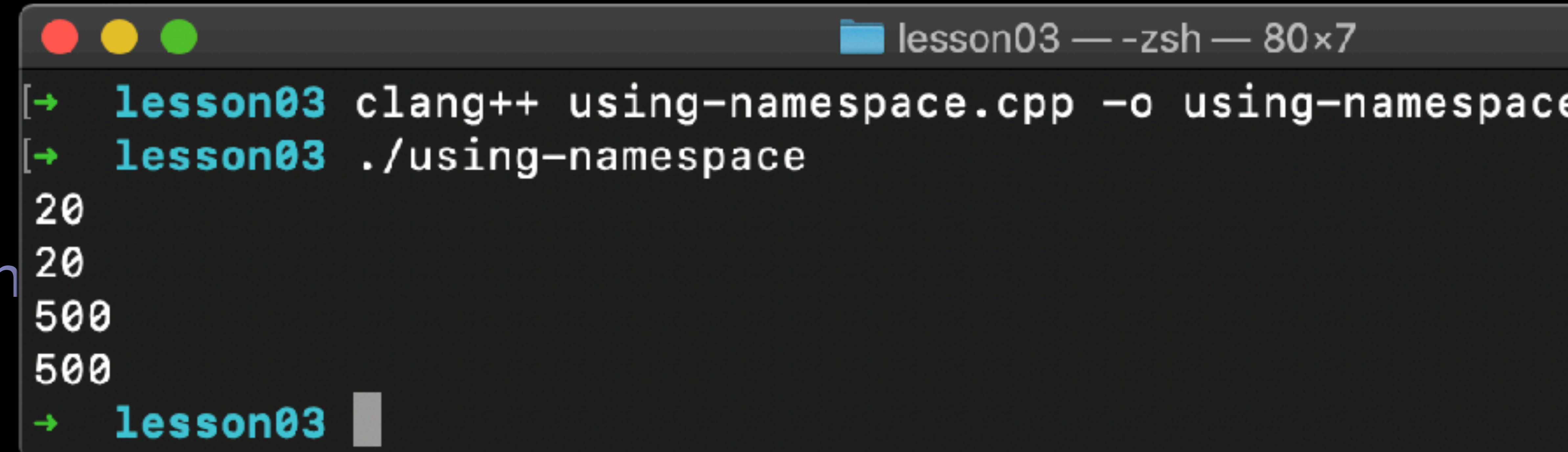
```
namespace first { // Namespace names are all lower-case
  int var = 5;
}
namespace second {
  double var = 3.1416;
}

int main () {
  std::cout << first::var << '\n';
  std::cout << second::var << '\n';
  return 0;
}
```



Using namespace directive

The keyword ‘using’ introduces a name from a namespace into the current region or even to introduce an entire namespace.

```
namespace operations  {
    int val = 500;
    int mult(int a, int b) {return a * b;}
}
using operations::mult; // Only mult
using operations::val;  // Only val
// Or
using namespace operations; // All the namespace

int main() {
    std::cout << operations::mult(4,5)
    std::cout << mult(4,5) << '\n';
    std::cout << operations::val << '\n
    std::cout << val << '\n';
    return 0;
}
```

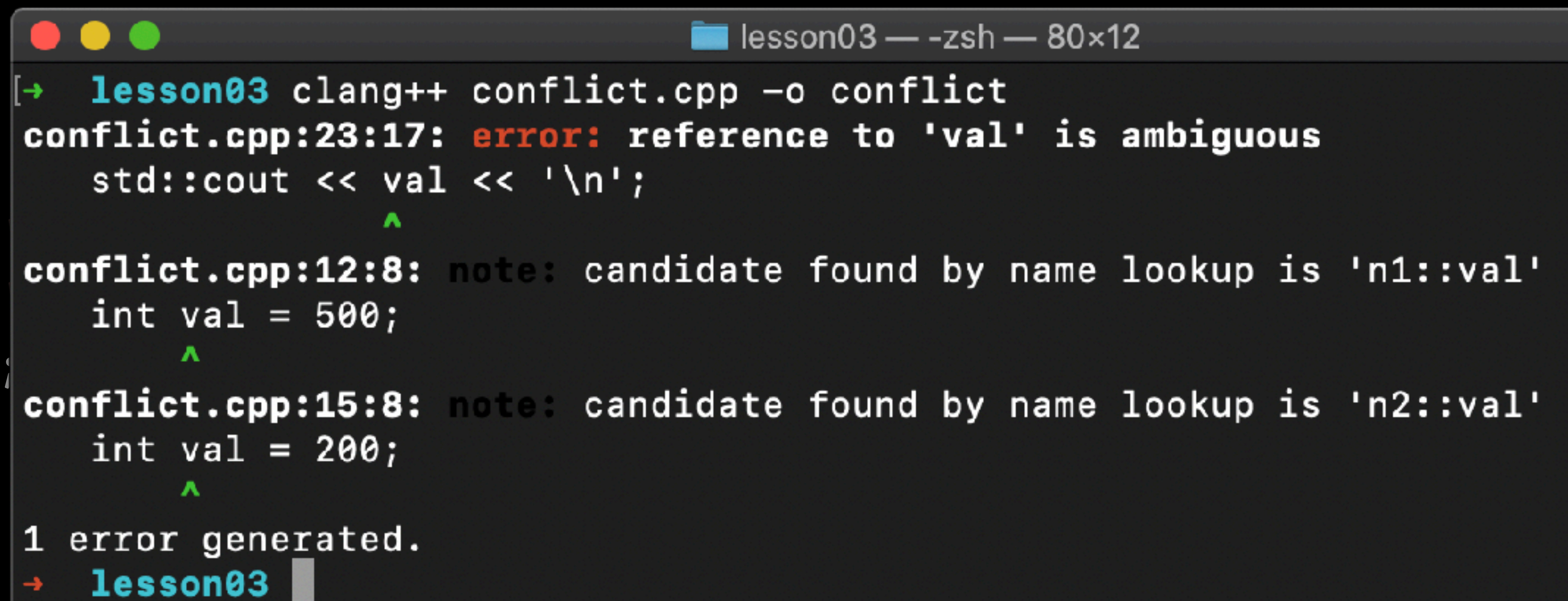


Using namespace directive

Using namespace directive is considered BAD PRACTICE

It's not safe because there may be ambiguity between elements from different namespaces that can have same name.

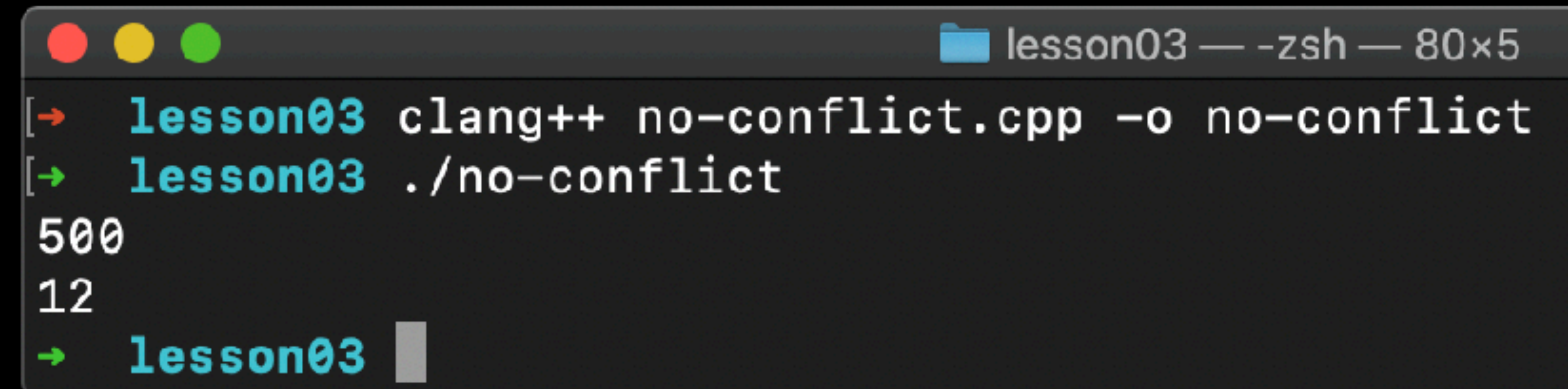
```
namespace n1  {  int val = 500; }
namespace n2 { int val = 200; }
using namespace n1;
using namespace n2;

int main() {
    std::cout << n1::val <<
    std::cout << n2::val <<
    std::cout << val << '\n'
    return 0;
}
```



Using directive

So, do not use ‘using namespace’ directive and eventually limit to use ‘using’ directive to import specific identifiers

```
namespace operations  {
    int val = 500;
    int mult(int a, int b) {return a * b;}
}
using operations::mult; // Only import mult function
using std::cout; // only import cout

int main() {
    cout << operations::val << '\n';
    std::cout << mult(3,4) << '\n';
    return 0;
}
```



if you still want to import entire namespaces, try to do so inside LIMITED SCOPE and not in global scope. Do not import namespace in .h file !

#02bis

Take Home Message

C++ mainly relies on USER-DEFINED TYPES that are collections of data describing an object's attributes and state

Class are the main user-defined types and struct, enum and union are marginally employed

User-defined namespaces organize code into logical groups and prevent name collisions.

Next Lectures

~~Lesson 00: Introduction~~

~~Lesson 01: Hello, world!~~

~~Lesson 02: User-defined types~~

Lesson 03: Inheritance

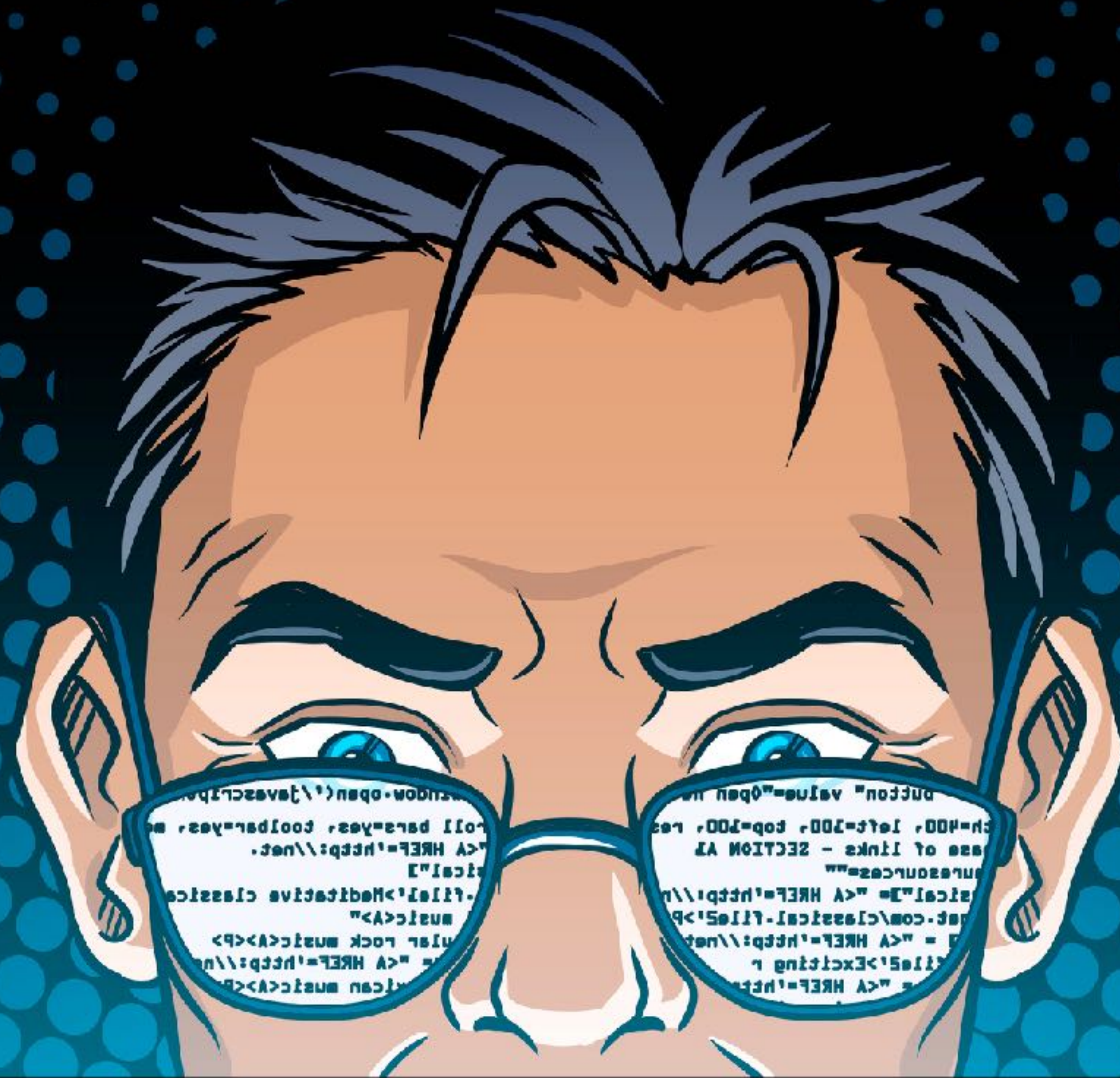
Lesson 04: Polymorphism

Lesson 05: STL Containers

Lesson 06: Indirection

Lesson 07: Templates

Lesson 08: Exceptions



So let’s go on C++ training with ❤

(c) 2020/2021 - dominique.ginhac@u-bourgogne.fr - @dginhac