POLYTECH
DIJON

2024/2025

ITC313

Mastering Object-Oriented Programming in C++: From Fundamentals to Best Practices

Pr. Dominique Ginhac
dginhac@u-bourgogne.fr

Last updated on October, 22, 2024 by dG

Lecture #02

Inheritance

http://ginhac.com/ITC313/02-inheritance.pdf

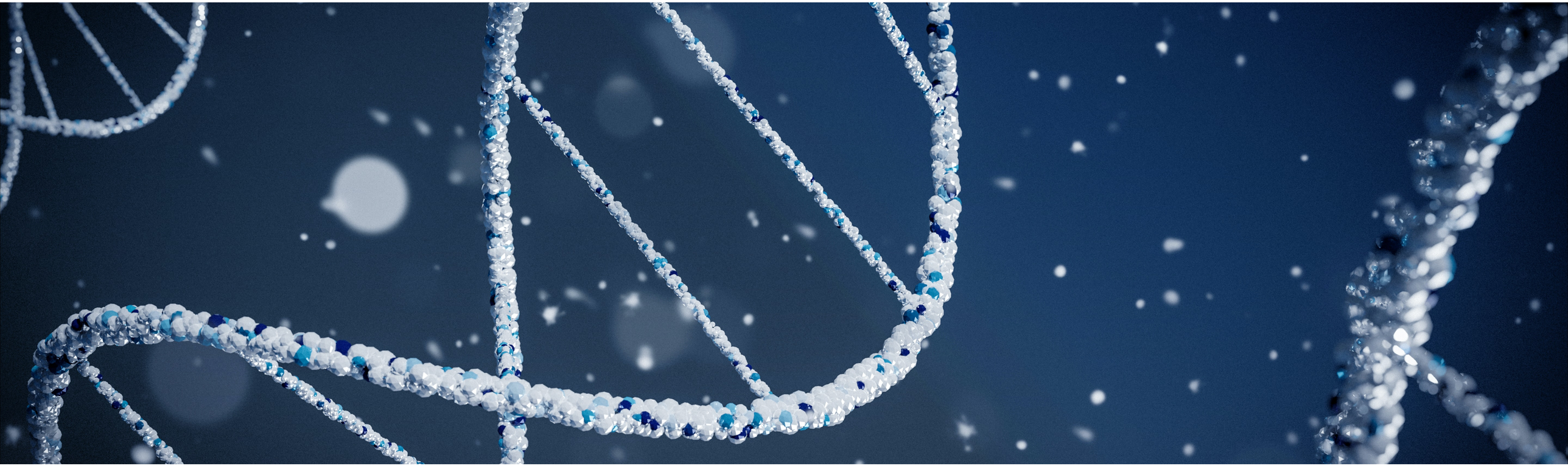


Photo by Braňo on Unsplash

Introduce inheritance one of the most important pillars of OOP with concrete examples

Enjoy! 😉

Lecture #00
Course Introduction

Lecture #01
User-defined Data Types

Today

Lecture #02
Inheritance

Lecture #03
Polymorphism

Lecture #04
STL Containers

Lecture #05
Templates

Lecture #06
Exceptions

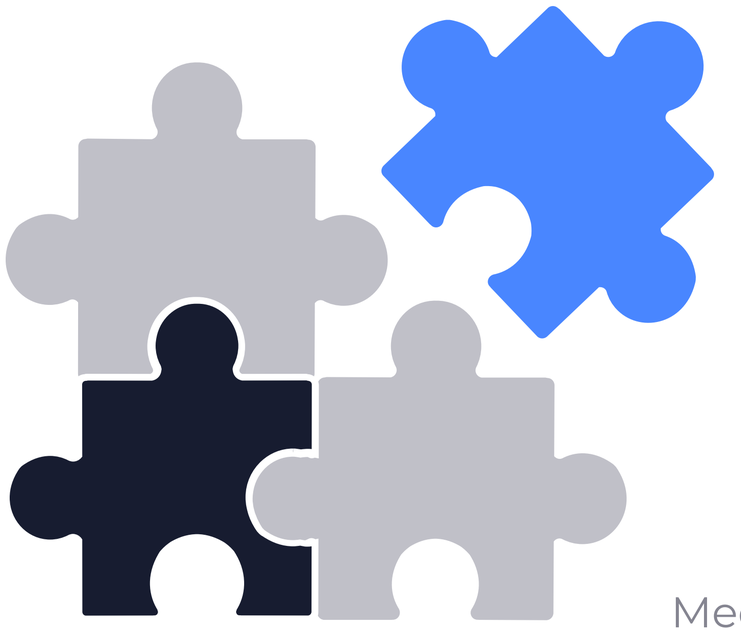
AGENDA

01 – What is Inheritance ?

02 – The todo class

03 – Multiple inheritance

The four pillars of OOP



Abstraction

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

Encapsulation

Mechanism, also known as Data hiding, that refers to the bundling of data/methods into a single coherent unit.

Inheritance

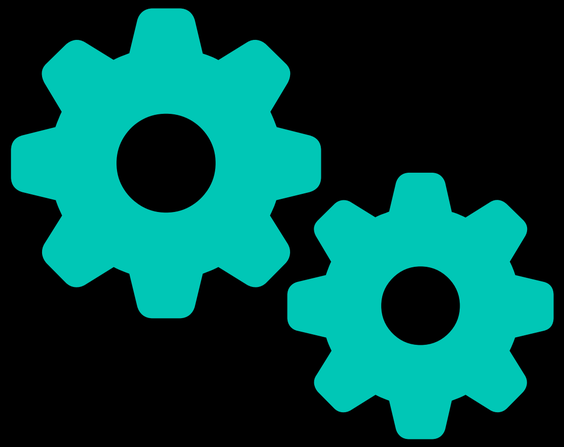
Mechanism of basing an object upon another object that allows sharing of properties and behaviors and implementation of new functionalities.

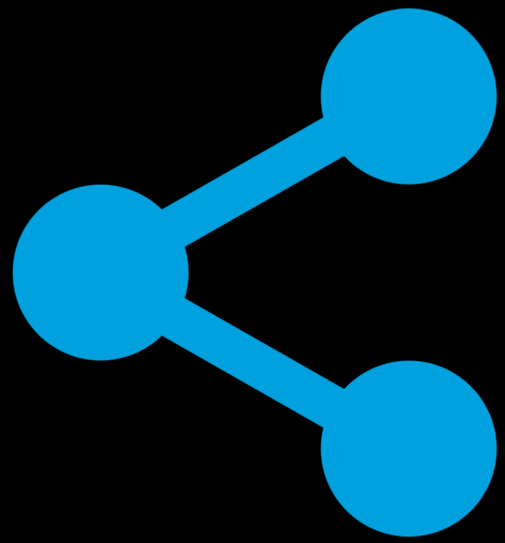
Polymorphism

Mechanism of assigning a different meaning or usage to something in different contexts.

Includes Overloading and Overriding.

Inheritance



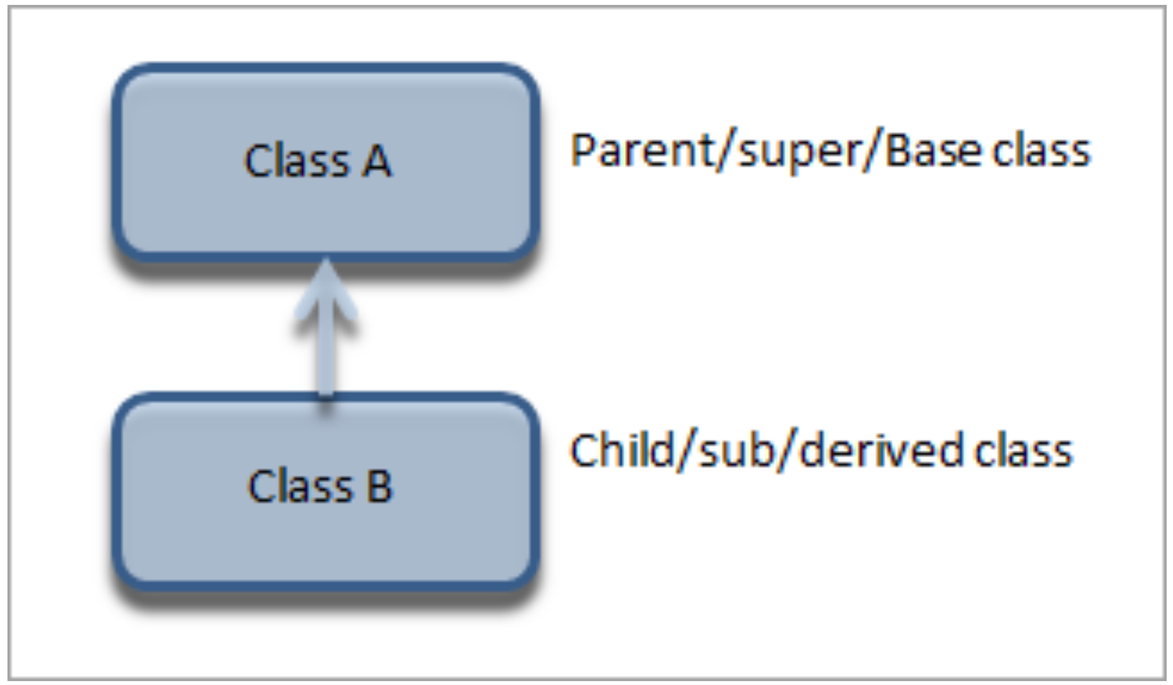


INHERITANCE is one of the most important key concepts of OOP

A DERIVED Class is a class that inherits the features (var and functions) of another class

Inheritance ADDS or OVERRIDES functionality of the base class

The syntax




```
class A {
    // Declaration of the base class
};

class B : Public A {
    // Declaration of the
    // first derived class
};
```

AGENDA

01 – What is Inheritance ?

02 – The todo class

03 – Multiple inheritance

Let’s take an example

Imagine you want to code a todo list application.

What are the minimal MEMBER VARIABLES of a generic Todo?

- title (std::string)
- category (enum)
- due date (Date)
- priority (int)
- status (bool)

The Todo class

todo.h

```
namespace todo {
  class Todo {
    public:
      Todo(std::string title, Category category, int priority,
           date::Date due_date, bool completed=false);
      std::string title() const;
      Category category() const;
      int priority() const;
      date::Date dueDate() const;
      bool completed() const;
      void updateTitle(std::string title);
      void updateCategory(Category category);
      void updatePriority(int priority);
      void updateDueDate(date::Date due_date);
      void setCompleted(bool completed);
    private:
      std::string _title;
      Category _category;
      int _priority;
      date::Date _due_date;
      bool _completed;
  };
}
```

```
enum class Category {
    Personal, Work
};
```

The Todo and Date classes are defined into specific namespaces (todo and date)

Using the Todo class

main.cpp

Need to use the fully qualified name todo:: and date:: to access the user-defined types defined into the namespaces

Use the fully qualified name to access the different values of the enum class Category

The parameter priority has been declared with #define directives

```
#define HIGH 10
#define NORMAL 5
#define LOW 1
```

```
-zsh — d0m                                    ⌥⌘1
→  basic-todo make
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app todo.o date.o main.o
→  basic-todo ./app
TODO: Buy Beer (Normal priority - 5/10)
TODO: Buy Beer (High priority - 11/10)
DONE: Buy Beer (11/10)
→  basic-todo
```

```cpp
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(10,5);
    Category category = Category::Personal;
    int priority = HIGH;
    todo::Todo todo1(title,
                     category,
                     priority,
                     due_date);
    display(todo1);
    // Update due date and priority
    todo1.updateDuedate(date::Date(10,11));
    todo1.updatePriority(HIGH);
    display(todo1);
    // Go To the Brewery and Buy Beer
    todo1.setCompleted(true);
    display(todo1);
   return 0;
}
```

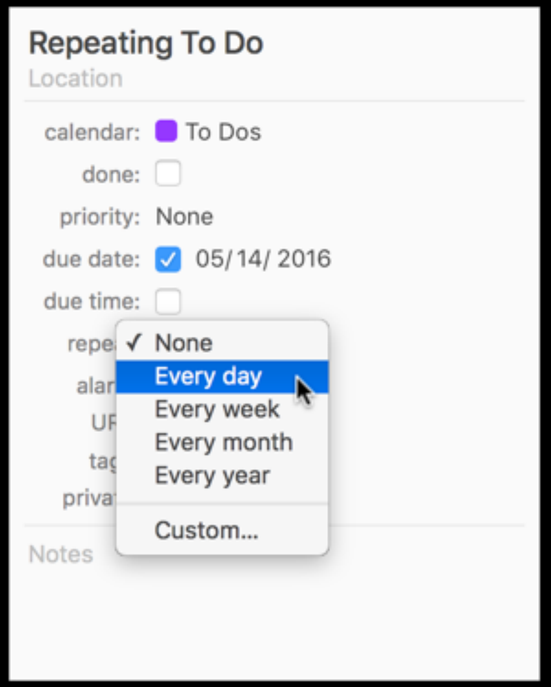
Recurring todo ?

A todo that happens on a REGULAR basis.

When you mark a todo complete, the next todo is automatically generated based on the repeating pattern you set up.

SIMILAR to a standard todo (title, category, priority, due date, completed)

But DIFFERENT from a standard todo (end date, period)



A RecurringTodo class

todo.h

Declared into the namespace todo

Defined as a PUBLIC derived class

Need only two other member variables
(period, end_date)
SHARE all other variables
with the Todo class

Need new getters/setters
for period and end_date

Share all other functions,
except setCompleted and display

RecurringTodo does not want to simply
inherit these two functions but will
REDEFINE them with specific code

```
namespace todo {
  class RecurringTodo : public Todo {
    public:
      RecurringTodo(std::string title,
                    Category category,
                    int priority,
                    date::Date due_date,
                    int period,
                    date::Date end_date,
                    bool completed=false);
      int period() const;
      date::Date endDate() const;
      void updatePeriod(int period);
      void updateEndDate(date::Date end_date);
      void setCompleted(bool completed);
   private:
      int _period;
      date::Date _end_date;
    };
}
```

Public inheritance?

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

Public inheritance is by far the most commonly used type of inheritance.

In practice, private inheritance is rarely used (‘has a’ relationship) and protected inheritance is very rare.

So, how to access Todo private variables into RecurringTodo ?

Update Todo using protected?

With a PRIVATE attribute in the base class, derived classes can not access the variables directly and need using PUBLIC accessors

With a PROTECTED attribute, derived classes can access members directly. But if you later change any protected member, you'll need to change the base class AND the derived classes. Moreover, it is up to every derived todo to manipulate the protected data correctly.

Making your members private gives you BETTER ENCAPSULATION and insulates derived classes from changes to the base class. But at the cost to build all the access methods that the derived classes need.

todo.h

```
namespace todo {
  class Todo {
    public:
      // Same declarations
      // of member functions
    private: // or protected
      std::string _title;
      Category _category;
      int _priority;
      date::Date _due_date;
      bool _completed;
  };
}
```

RecurringTodo Constructor

todo.cpp

```
RecurringTodo::RecurringTodo(std::string title, Category category,
    int priority, date::Date due_date, int period,
    date::Date end_date, bool completed)
:
    Todo(title, category, priority, due_date, completed),
    _period(period), _end_date(end_date) {
}
```

Only the base class constructor can properly initialize the base class members

So, calling the Todo constructor in the RecurringTodo INITIALIZER is required.

It is the only way to construct the Todo Object and initialize its inherited data

The display functions

todo.cpp

```
void display(const  Todo& todo) {
    if (todo.completed()) {
        std::cout << "DONE: " << todo.title() << " (" << toString(todo.due_date());
    }
    else {
        std::cout << "TODO: " << todo.title() << " (" << toString(todo.due_date();
    }
}
```

```
void display(const RecurringTodo& todo) {
    if (todo.completed()) {
        std::cout << "DONE: " << todo.title() << '\n';
    }
    else {
        std::cout << "TODO: " << todo.title() << " ("
                  << toString(todo.dueDate())
                  << ") - every “ << todo.period() << " days until "
                  << toString(todo.endDate()) << '\n';
    }
}
```

The setCompleted functions

todo.cpp

```
void Todo::setCompleted(bool completed) {
    _completed = completed;
}
```

```
void RecurringTodo::setCompleted(bool completed) {
    if (completed) {
        updateDueDate(dueDate()+_period);
        if (dueDate()>_end_date) {
            Todo::setCompleted(completed);
        }
    }
    else {
        Todo::setCompleted(false);
    }
}
```

No access to private members of Todo

Require using getters and setters of base class

Use base class setCompleted function with fully qualified name

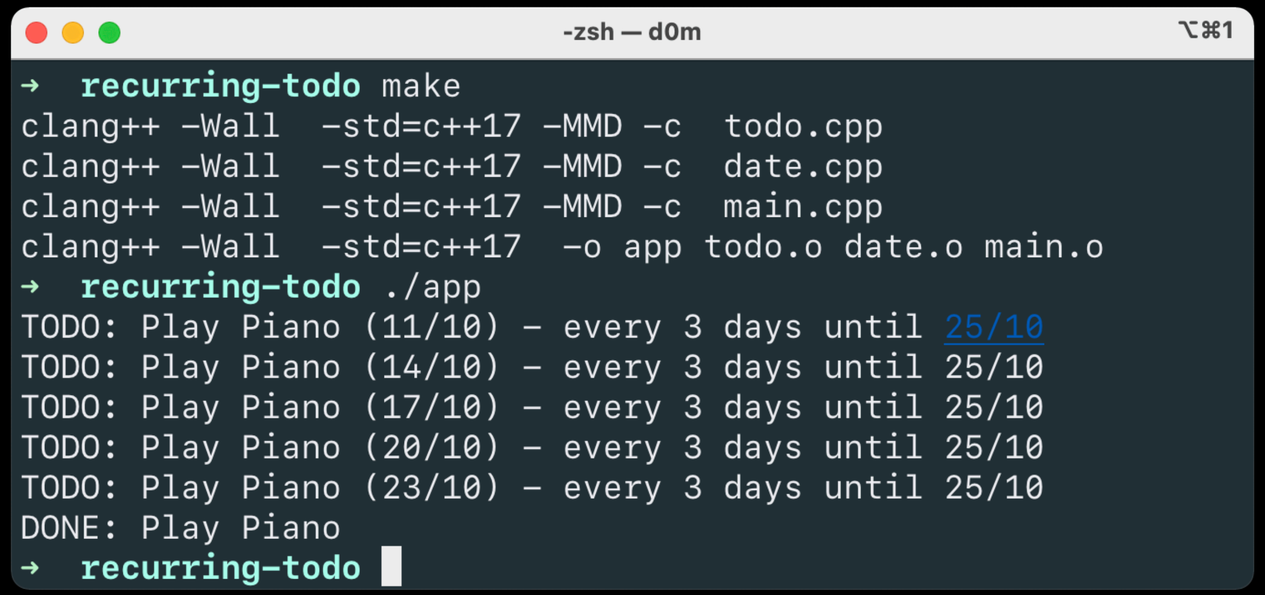
Comparison and other arithmetic operations
`dueDate()+_period`
`dueDate()>_end_date`

Upcoming lesson

Using the RecurringTodo class

Using a base class or a derived class is similar. NO DIFFERENCE when declaring a variable of the base class (Todo) or of the derived class (RecurringTodo)

Only the parameters passed to the CONSTRUCTOR must be specific




```cpp
int main(int argc, char const *argv[]) {
    std::string title = "Play Piano";
    date::Date due_date(10,11);
    date::Date end_date(10,25);
    Category category = Category::Personal;
    int priority = NORMAL;
    int period = 3; // every 3 days
    todo::RecurringTodo r_todo1(title,
                                category,
                                priority,
                                due_date,
                                period,
                                end_date);
    while (!r_todo1.completed()) {
        display(r_todo1);
        r_todo1.setCompleted(true);
    }
    display(r_todo1);
}
```

main.cpp

Questions

AGENDA

01 – What is Inheritance ?

02 – The todo class

03 – Multiple inheritance

Multiple inheritance

C++ provides the ability to do MULTIPLE inheritance.

Multiple inheritance enables a derived class to inherit members from MORE THAN ONE base class.

univ.h

```
class Professor : public Member, public Teacher, public Researcher {
   public:
      Professor(std::string firstname, std::string lastname,
date::Date birthday, std::string faculty, std::string lab, double
salary);
      double salary() const;
      void updateSalary(double salary);
 private:
      double _salary;
   };
```

univ.h

```
class Member {
   public:
      Member(std::string firstname, std::string lastname, date::Date birthday);
   private:
      int _id;
      std::string _firstname;
      std::string _lastname;
      date::Date _birthday;
   };
   class Teacher {
   public:
      Teacher(std::string faculty);
   private:
      std::string _faculty;
   };
   class Researcher {
   public:
      Researcher(std::string lab);
   private:
      std::string _lab;
   };
```

Code for getters/ setters is not presented here

univ.cpp

```cpp
Professor::Professor(std::string firstname, std::string lastname, date::Date
birthday, std::string faculty, std::string lab, double salary) :
    Member(firstname, lastname, birthday), Teacher(faculty),
    Researcher(lab), _salary(salary) {
}

void display(Professor& p) {
  std::cout << p.firstname() << " " << p.lastname() << " (" << p.id() << ")\n";
}
```

main.cpp

```cpp
#include “univ.h"

int main() {
    Univ::Professor me("Dom",“Ginhac",date::Date(05,26),“ESIREM”,"IMVIA",10000);
    display(me);
    return 0;
}
```

```
-zsh — d0m                                                    ⌥⌘1
→  professor make
clang++ -Wall  -std=c++17 -MMD -c  univ.cpp
clang++ -Wall  -std=c++17 -MMD -c  date.cpp
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app univ.o date.o main.o
→  professor ./app
Dom Ginhac (897180041)
→  professor
```

Multiple inheritance

Multiple inheritance is not a simple extension of single inheritance. MI introduces a lot of ISSUES that can increase the complexity of programs and make them a maintenance nightmare (diamond problem).

Do we really need Multiple inheritance? Not really. We can do WITHOUT multiple inheritance because it is prone to compilation error.

The only case where multiple inheritance can be interesting is when using INTERFACES (i.e. classes with all the functions defined as pure virtual) to derive a class.

Questions

#04

Take Home Message

INHERITANCE is 1 of the 4 core concepts of generic programming in OOP with abstraction, encapsulation and polymorphism

Inheritance allows us to derive a new class from a base class, inheriting the features from the base class while providing its own additional features

Inheritance offers opportunities to reuse code and to enable faster implementation time

Photo by Alexander Andrews on Unsplash

Contacts

Pr. Dominique Ginhac

dginhac@u-bourgogne.fr

Come visit us at

https://github.com/dginhac/esirem-itc313