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

ITC313 - Lesson 03

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

Dominique Ginhac

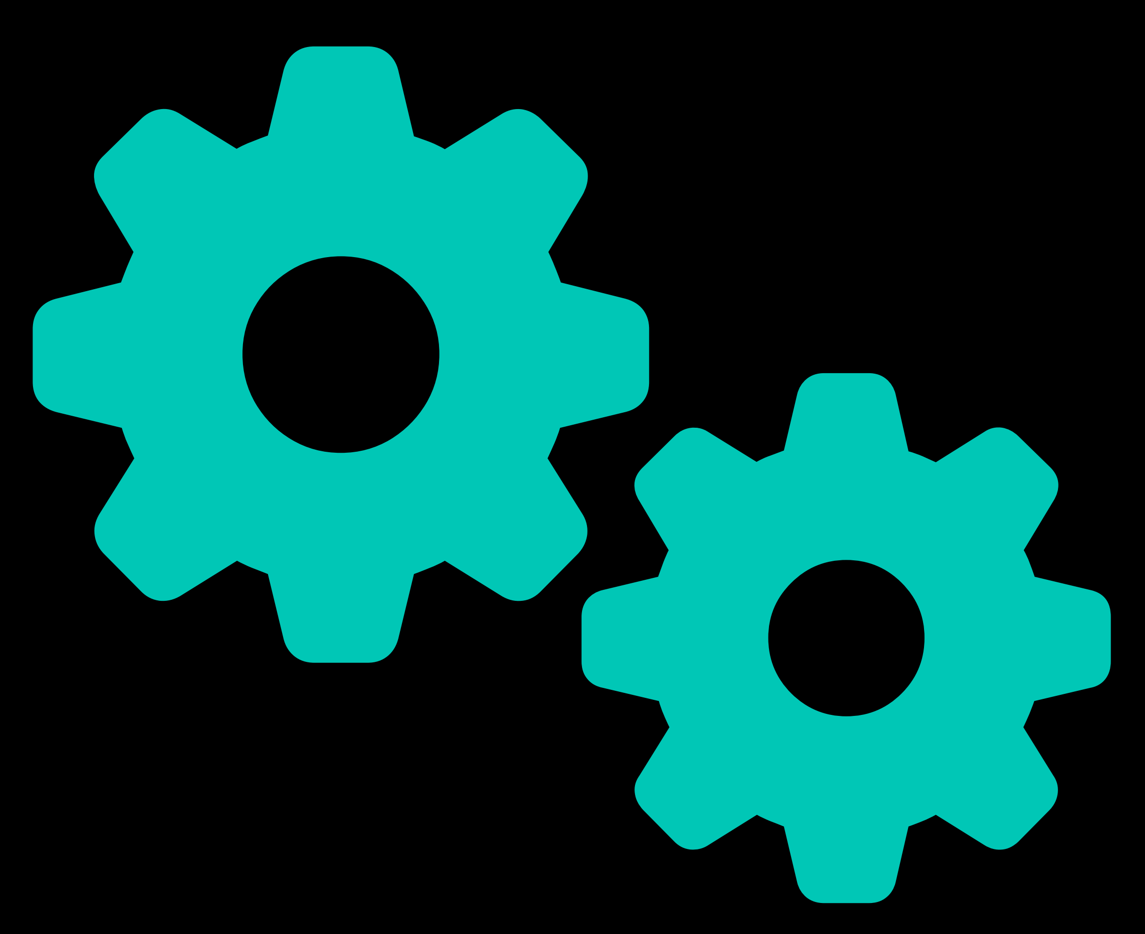
(c) 2020/2021 - dominique.ginhac@u-bourgogne.fr - @dginhac

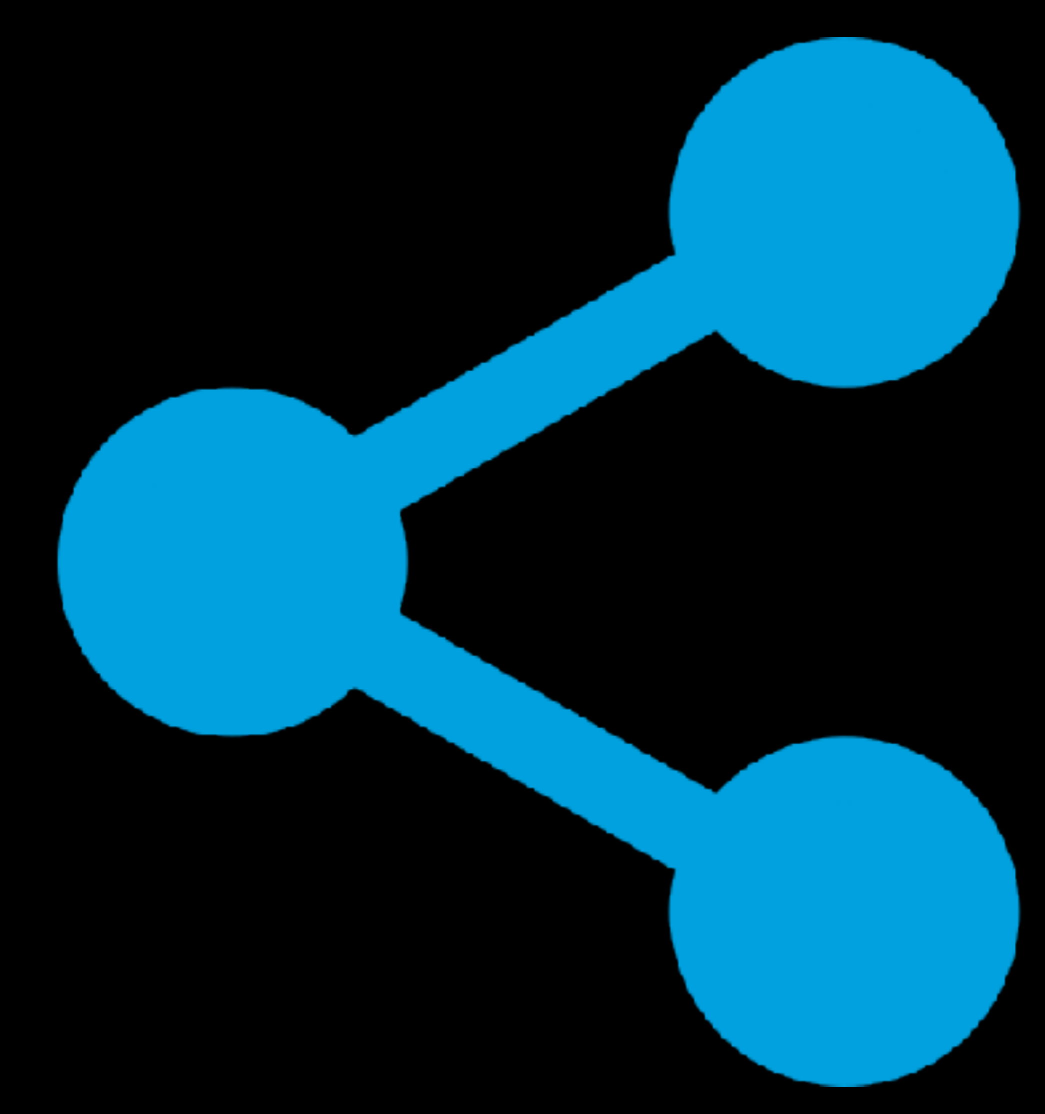
Lesson 03

Inheritance

In this lesson, students will learn how to implement inheritance, one of the most important features of Object Oriented Programming.

Inheritance





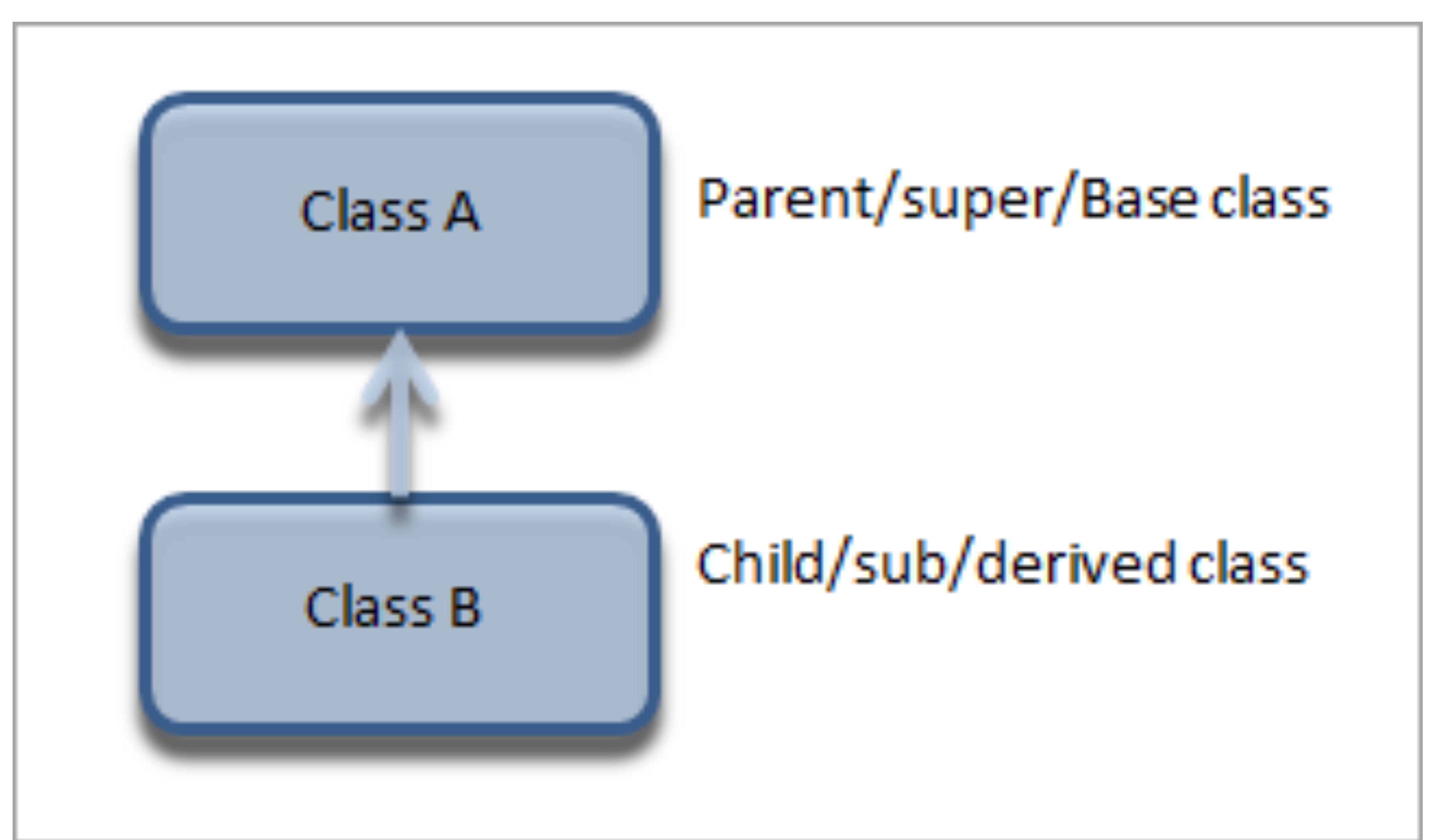
INHERITANCE is the third key concepts of OOP

(After Encapsulation and Abstraction)

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
};
```

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

```cpp
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);
      void display() const;
    private:
      std::string _title;
      Category _category;
      int _priority;
      date::Date _due_date;
      bool _completed;
  };
}
```

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

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

Using the Todo class

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
```

```
lesson03 — -zsh — 78×9
➜  lesson03 make
clang++ -Wall  -std=c++14 -MMD -c  date.cpp
clang++ -Wall  -std=c++14 -MMD -c  todo.cpp
clang++ -Wall  -std=c++14 -MMD -c  todo-main.cpp
clang++ -Wall  -std=c++14  -o todo-main date.o todo.o todo-main.o
➜  lesson03 ./todo-main
TODO: Buy Beer (2020/Oct/1)
DONE: Buy Beer (2020/Oct/1)
➜  lesson03
```

```
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(2020,10,1);
    Category category = Category::Personal;
    int priority = HIGH;
    // bool completed = false;

    todo::Todo todo1(title,
                     category,
                     priority,
                     due_date);
    todo1.display();
    // Go To the Brewery and Buy Beer
    todo1.setCompleted(true);
    todo1.display();
  return 0;
}
```

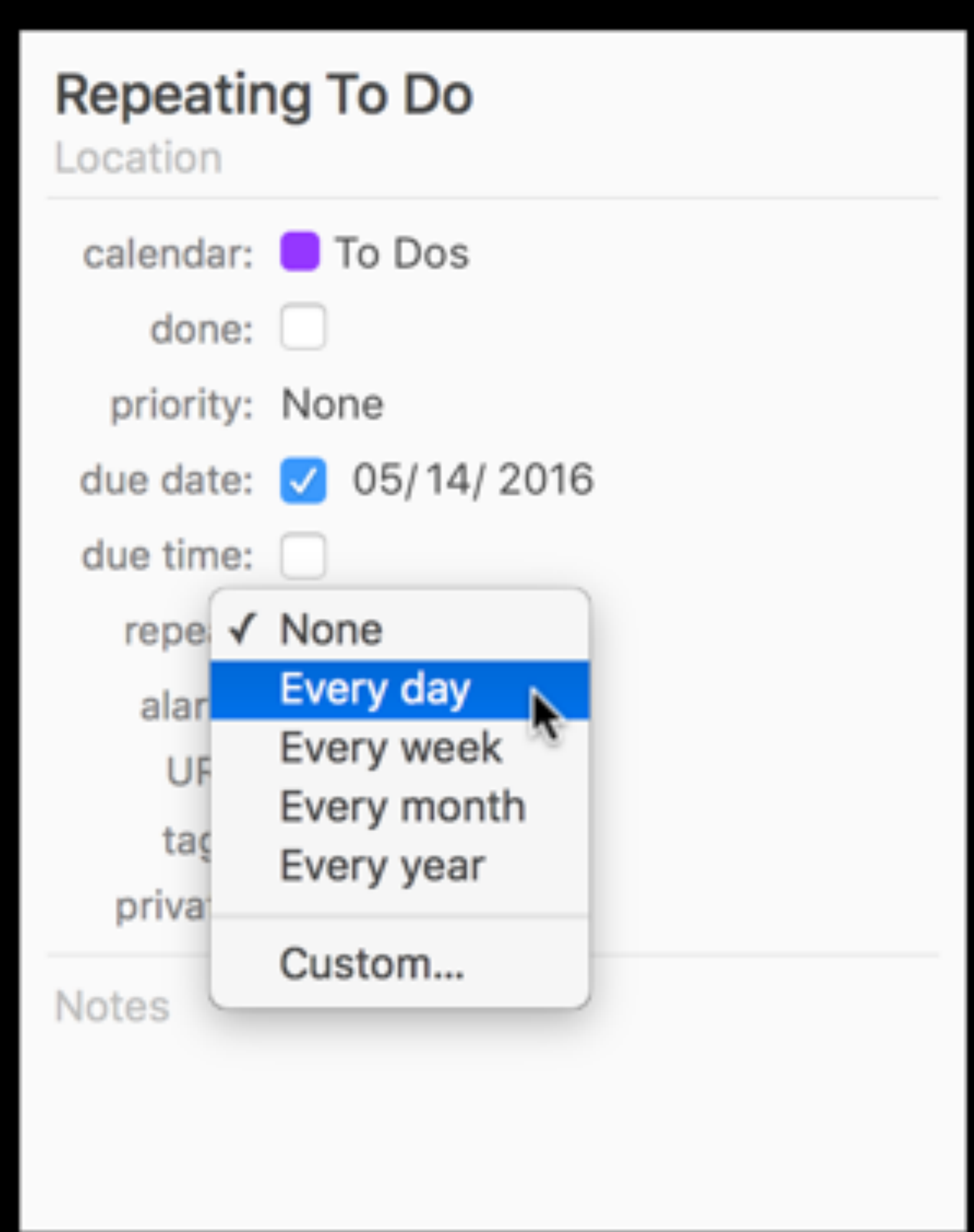
Recurring todo ?

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

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

Share all other functions, except display and setCompleted

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);
      void display() const;
    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.

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.

```
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;
   };
}
```

Using the RecurringTodo class

Using a base class or
a derived class is similar

There is 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

The display function
of the base class is REDEFINED
into the RecurringTodo class

```
→  lesson03 make
clang++ -Wall  -std=c++14 -MMD -c  date.cpp
clang++ -Wall  -std=c++14 -MMD -c  todo.cpp
clang++ -Wall  -std=c++14 -MMD -c  todo-main.cpp
clang++ -Wall  -std=c++14  -o todo-main date.o todo.o todo-main.o
→  lesson03 ./todo-main
TODO: Buy Beer (2020/Oct/1)
TODO: Play Piano (2020/Oct/1) - every 2 days until 2020/Nov/3
→  lesson03
```

```cpp
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(2020,10,1);
    Category category = Category::Personal;
    int priority = HIGH;
    todo::Todo todo1(title, category,
                     priority, due_date);
    todo1.display();
    title = "Play Piano";
    date::Date end_date(2099,1,1);
    int period = 2; // every 2 days
    todo::RecurringTodo r_todo1(title,
                                category,
                                priority,
                                due_date,
                                period,
                                end_date);
    r_todo1.display();
}
```

RecurringTodo Constructor

```
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

```
void Todo::display() const {
    if (_completed) {
        std::cout << "DONE: " << _title << " (" << _due_date.toString() << ")\n";
    }
    else {
        std::cout << "TODO: " << _title << " (" << _due_date.toString() << ")\n";
    }
}
```

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

By redefining the display() function (same name and parameters), each RecurringTodo object will call AUTOMATICALLY the derived function

No access to private members of Todo

Require getters

Using the RecurringTodo class

```
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";    date::Date due_date(2020,10,1);
    Category category = Category::Personal;     int priority = HIGH;
    todo::Todo todo1(title, category, priority, due_date);
    todo1.display();
    todo1.setCompleted(true); // Go To the Brewery and Buy Beer
    todo1.display();
    title = "Play Piano";  date::Date end_date(2020,10,20);
    int period = 2; // every two days
    todo::RecurringTodo r_todo1(title, category,
    priority, due_date, period, end_date);
    while (!r_todo1.completed()) {
        r_todo1.display();
        r_todo1.setCompleted(true);
    }
    r_todo1.display();
    return 0;
}
```

```
lesson03 — -zsh — 78×20
→  lesson03 make
clang++ -Wall  -std=c++14 -MMD -c  date.cpp
clang++ -Wall  -std=c++14 -MMD -c  todo.cpp
clang++ -Wall  -std=c++14 -MMD -c  todo-main.cpp
clang++ -Wall  -std=c++14  -o todo-main date.o todo.o todo-main.o
→  lesson03 ./todo-main
TODO: Buy Beer (2020/Oct/1)
DONE: Buy Beer (2020/Oct/1)
TODO: Play Piano (2020/Oct/1) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/3) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/5) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/7) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/9) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/11) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/13) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/15) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/17) - every 2 days until 2020/Oct/20
TODO: Play Piano (2020/Oct/19) - every 2 days until 2020/Oct/20
DONE: Play Piano
→  lesson03
```

The setCompleted functions

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

```
void RecurringTodo::setCompleted(bool completed) {
    if (completed) {
        for (int i=0; i<_period; i++) {
            date::Date next = dueDate();
            next.nextDay();
            updateDueDate(next);
        }
        if (dueDate()>_end_date) {
            Todo::setCompleted(completed);
        }
    }
    else {
        Todo::setCompleted(false);
    }
}
```

No access to private members of Todo

Require using getters and setters

Use base class setCompleted function with fully qualified name

Comparison between objects `dueDate()>_end_date`

Upcoming lesson

To sum up, Todo and RecurringTodo share variables and functions but have also some specificities:

- Additional variables for RecurringTodo
- Different behavior for functions such as display, setCompleted, …

Can't we create a sufficiently generic class and use it to derive classes both for a normal task and a recurring task?

Abstract class

An ABSTRACT class is a class that has at least ONE PURE VIRTUAL function, i.e. a function that has only a declaration (.h) and no definition (.cpp)

A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes

```
virtual void setCompleted(bool completed) = 0; //=0 => PURE virtual
virtual void display() const = 0; // No need to define it into .cpp file
```

An abstract class can't be instantiated because of the virtual function

Abstract classes are essential to provide ABSTRACTION to make code REUSABLE and EXTENDABLE

GenericTodo class

```
class GenericTodo {
  public:
    GenericTodo(std::string title, Category category, int priority, bool completed
    virtual void setCompleted(bool completed) = 0;
    virtual void display() const = 0;
  protected:
    std::string _title;
    Category _category;
    int _priority;
    bool _completed;
  };
```

GenericTodo uses protected variables for reasons of SIMPLICITY

Only COMMON variables are provided

Declared with only 1 constructor and 2 pure virtual functions

```
GenericTodo::GenericTodo(std::string title, Category category,
                         int priority, bool completed) :
                         _title(title), _category(category),
                         _priority(priority), _completed(completed) {}
```

The new Todo class

The new Todo class inherits from GenericTodo, provides its constructor, adds a private due date variable with its getter/setter and the two functions that are virtual in the base class

```
class Todo : public GenericTodo {
   public:
      Todo(std::string title, Category category, int priority,
           date::Date due_date, bool completed=false);
      date::Date dueDate() const;
      void updateDueDate(date::Date due_date);
      void setCompleted(bool completed);
      void display() const;
   private:
      date::Date _due_date;
   };
```

The new RecurringTodo class

```
class RecurringTodo: public GenericTodo {
   public:
      RecurringTodo(std::string title, Category category, int priority,
                    date::Date next_date, int period, date::Date end_date,
                    bool completed=false);
      date::Date nextDate() const;
      date::Date endDate() const;
      void updateNextDate(date::Date next_date);
      void updateEndDate(date::Date end_date);
      void setCompleted(bool completed);
      int period() const;
      void updatePeriod(int period);
      void display() const;
   private:
      date::Date _next_date;
      date::Date _end_date;
      int _period;
   };
```

Provides its CONSTRUCTOR

adds PRIVATE VARIABLES
(next date, end date, period)
with getter/setter

Provides also the
IMPLEMENTATION of the two
virtual functions

The setCompleted function

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

void RecurringTodo::setCompleted(bool completed) {
      if (completed) {
         for (int i=0; i<_period; i++) {
            _next_date.nextDay();
         }
         if (_next_date>_end_date) {
            _completed = completed;
         }
      }
      else {
         _completed = false;
      }

   }
```

With protected variables in the base class, writing functions in the derived class is easier.

But be careful not to change the name of the variables in the base class!

Using the two classes

Since GenericTodo is an abstract class, it's not possible to declare GenericTodo variables.

```
lesson03 — -zsh — 70×19
→  lesson03 make -f Makefile.abstract
clang++ -Wall  -std=c++14 -MMD -c  date.cpp
clang++ -Wall  -std=c++14 -MMD -c  abstract-todo.cpp
clang++ -Wall  -std=c++14 -MMD -c  abstract-todo-main.cpp
abstract-todo-main.cpp:25:22: error: variable type 'todo::GenericTodo'
      is an abstract class
   todo::GenericTodo g_todo(title, category, priority, due_date);
                     ^
./abstract-todo.h:25:20: note: unimplemented pure virtual method
      'setCompleted' in 'GenericTodo'
      virtual void setCompleted(bool completed) = 0;
                   ^
./abstract-todo.h:26:20: note: unimplemented pure virtual method
      'display' in 'GenericTodo'
      virtual void display() const = 0;
                   ^
1 error generated.
make: *** [abstract-todo-main.o] Error 1
→  lesson03
```

```cpp
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(2020,10,25);
    Category category = Category::Personal;
    int priority = HIGH;

    todo::GenericTodo g_todo(title,
                             category,
                             priority,
                             due_date);
    g_todo.display();

    return 0;
}
```

```cpp
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(2020,10,25);
    Category category = Category::Personal;
    int priority = HIGH;
    todo::Todo todo1(title, category,
                     priority, due_date);
    todo1.display();
    todo1.setCompleted(true);
    todo1.display();

    title = "Play Piano";
    date::Date next_date(2020,10,20);
    date::Date end_date(2020,11,4);
    int period = 2; // 2 days
    todo::RecurringTodo r_todo1(title, category, priority,
                                next_date, period, end_date);
    do {
        r_todo1.setCompleted(true);
        r_todo1.display();
    } while (!r_todo1.completed());
    return 0;
}
```

lesson03 — -zsh — 62×18

```
→  lesson03 make -f Makefile.abstract
clang++ -Wall  -std=c++14 -MMD -c  date.cpp
clang++ -Wall  -std=c++14 -MMD -c  abstract-todo.cpp
clang++ -Wall  -std=c++14 -MMD -c  abstract-todo-main.cpp
clang++ -Wall  -std=c++14  -o abstract-todo-main date.o abstra
ct-todo.o abstract-todo-main.o
→  lesson03 ./abstract-todo-main
TODO: Buy Beer (2020/Oct/25)
DONE: Buy Beer (2020/Oct/25)
TODO: Play Piano (2020/Oct/22) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Oct/24) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Oct/26) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Oct/28) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Oct/30) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Nov/1) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Nov/3) every 2 days until 2020/Nov/4
DONE: Play Piano
→  lesson03
```

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.

```
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);
        void display();
   private:
        double _salary;
    };
```

```
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

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

```cpp
#include "ub.h"

int main() {
   ub::Professor me("Dom",“Ginhac",date::Date(1972,05,26),“ESIREM”,"IMVIA",10000);
   me.display();
   return 0;
}
```

lesson03 — -zsh — 76×5

```
→  lesson03 clang++ -std=c++14 date.cpp ub.cpp ub-main.cpp -o ub-main
→  lesson03 ./ub-main
Dom Ginhac (1325924463)
→  lesson03
```

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

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

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