# Advanced Programming Tutorial
# Worksheet 7: Object-Oriented Programming (Part 2)

**Keywords:** inheritance, `protected`, `virtual`, runtime polymorphism

## Prepare!

Before coming to the tutorial, and so that the time is enough to solve the exercise, revise the slides of the lecture unit "Object-Oriented Programming (pt II)" and answer the following questions:

- ☐ If a class `Cat` defines a private member `_gravity` and a class `SmellyCat` inherits from `Cat`, can `SmellyCat` use the value of `_gravity`?
- ☐ Where are virtual functions useful?
- ☐ What is slicing and when does it happen?
- ☐ What is moving? What steps does a move constructor typically execute?
- ☐ Remember `OnlineMUT` from the previous worksheet and look at the new skeleton. What new classes are there?

We only have one exercise this week, because we want to build a larger codebase with multiple classes. You can definitely solve the exercise by yourself, but you need some time to think how to implement everything.

## Exercise 1: One needs Mut[1] to look into all these classes

Let's continue our ambitious project to build `OnlineMUT`, an Online system for Managing University Tables. Read the comments in the `main.cpp` to understand the goal and then implement the following tasks:

1. Different courses have different properties of interest: while a lecture has a final exam, a practical course has worksheets, and a seminar has a paper and a presentation. We can use inheritance to give our courses different meaning, apart from only the `CourseType` label.
   (a) Have a look at the files `Lecture.h` and `Lecture.cpp`. Why can the constructor of `Lecture` see and print the `_id`? How do we initialize the underlying `Course` object?
   (b) Implement similarly the `Practical` class to represent a practical course that does not have a final exam, but has a number of worksheets.
2. To get more information about a course, we want that every course type contains a `void describe()` method.
   (a) How can we enforce that every course implements such a method? Apply your idea.
   (b) Implement the `describe()` method in both `Lecture` and `Practical`.
   (c) Declare your destructor as `virtual`. When do we really need to do this?

[1] "Mut" in German: "courage"

3. Now that we have concrete types, we don't need the `CourseType` anymore. Remove it from everywhere and implement `type()` similarly to the `describe()`.

4. Even though we have different course types, we sometimes want to treat them in the same way: we don't want a list of lectures, a list of seminars, and a list of practical courses, we only want a list of courses.

   (a) What do we need to achieve runtime polymorphism?

   (b) In `main.cpp`, define a function `void describe_course(std::shared_ptr<const Course>)` that accepts any kind of course and calls its `describe()` method.

5. We don't want that every student stores a separate copy of a course, but we want that they only maintain a list of pointers to courses they are registered for. Courses should be managed by study programs (curricula). Have a look at the files `Curriculum.h` and `Curriculum.cpp`:

   (a) Implement the constructor of `Curriculum`, which populates the list of courses with at least two (hard-coded for now) courses of different types and prints the available courses. For simplification, we assume that all the study programs have the same courses.

   (b) We also want to be able to print the list of courses after constructing a study program. Move the printing functionality to a method `void print_courses()`.

   (c) Implement a method `std::shared_ptr<const Course> request(std::string id)` which searches the available courses for a course of the given `id` and returns a pointer to that course, if found, or throws an exception, if not. See the "some new concepts" section below.

   (d) The `Student` class should now store a vector of pointers to `const` courses, instead of a vector of courses. What kind of pointer (raw, unique, shared) do we need here and why? See the section "some new concepts" for an alternative.

   (e) The student's `register_for()` method now accepts a course id instead of a course. Implement it so that it uses the `request` method and handles any expected exceptions.

6. **Idea:** Use Sourcetrail to visualize and navigate the class hierarchies in your project. This amazing project is unfortunately being discontinued early, but maybe there will be some successor.

7. **Idea:** Use Doxygen to generate documentation and a clickable structure of your project.

## Some new concepts

`try{ ... } catch() { ... }`

There are several ways to handle errors. C++ offers the concept of *exceptions*. We can *try* to execute code that can run into (specific or not) types of errors and then *catch* these errors and handle them, without exiting the program.

For example:

```
try {
    // We are trying to access the third point of a line,
    // but a line only has two points
    std::cout << line._vertices.at(2) << std::endl;
}
catch (out_of_range& err) {
    // Instead of exiting unexpectedly,
    // we show the error message and continue.
    std::cerr << err.what() << std::endl;
}
```

There are multiple exception types[2], and we should try to catch the most specific type that represents what we expect to happen (e.g. division by zero, out of range, etc). There are also more general types, such as `std::runtime_error`: `throw(std::runtime_error("oops!"));`.

### Different kinds of pointers and `std::weak_ptr`

We have already seen raw pointers (∗) and two types of smart pointers (`std::unique_ptr`, `std::shared_ptr`). The main reason we introduced these two smart pointers was to ensure that dynamic resources we may be using are automatically destructed/released when the objects that own them (the smart pointers) go out of scope.

There are still cases where a raw pointer is perfectly fine: for example, when we only want to *observe* a resource that is managed by an external object (such as a smart pointer). We can access the raw pointer contained in a smart pointer with the `.get()` method:

```
std::shared_ptr<int> sp = std::make_shared<int>();
int * p = sp.get();
```

However, we should only be using raw pointers when we are certain that the object still exists before we try to access it. There is a third type of smart pointer that implements such a check: the `std::weak_ptr`:

```
std::shared_ptr<int> sp = std::make_shared<int>();
std::weak_ptr<int> wp = sp;
```

As the `std::weak_ptr` is only meant to be pointing to existing objects, there is no “make weak” method. The way we use it is also a bit different (we first need to `.lock()` it to access the object it points to, which returns a `std::shared_ptr`).

[2]See `https://en.cppreference.com/w/cpp/error/exception` for a list of exception types.

## Advice / C++ Core Guidelines

The C++ Core Guidelines have a large section C: Classes and class hierarchies. Here are the most interesting guidelines, to our view:

- C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently
- C.9: Minimize exposure of members
- C.10: Prefer concrete types over class hierarchies
- C.20: If you can avoid defining default operations, do ("rule of zero")
- C.21: If you define or `=delete` any copy, move, or destructor function, define or `=delete` them all ("rule of five")
- C.31: All resources acquired by a class must be released by the class's destructor
- C.35: A base class destructor should be either public+virtual, or protected+non-virtual
- C.41: A constructor should create a fully initialized object
- C.45: Don't define a default constructor that only initializes data members; use in-class member initializers instead
- C.49: Prefer initialization to assignment in constructors
- C.51: Use delegating constructors to represent common actions for all constructors
- C.67: A polymorphic class should suppress copying
- C.82: Don't call virtual functions in constructors and destructors
- C.128: Virtual functions should specify exactly one of `virtual`, `override`, or `final`
- C.132: Don't make a function virtual without reason
- C.152: Never assign a pointer to an array of derived class objects to a pointer to its base
- C.153: Prefer virtual function to casting

## Epilogue

Use cases of runtime polymorphism include choosing a numerical solver (e.g., direct or iterative), the back-end for some computation (e.g., running on the CPU or the GPU, depending on what is available), or the database system to use in your next web-based project. Of course, it quickly becomes difficult to understand the relation of all classes in a codebase. Not understanding how parts of the code are connected is very common and sometimes takes years of experience in a project to comfortably find your way through everything. Knowing how to read code is as useful and as important as knowing how to write it.

*Chair of Scientific Computing in Computer Science – Technical University of Munich*
*Last updated on November 2, 2023*