Introduction to Modern C++

Lab Session 5
Const, Virtual, and Move Semantics

1 Introduction

We wish to construct some model of the (economic) world. To that end, we need a representation of countries: We will create a `class` called `Country` that contains relevant information, for instance at least: Name of country (in English), Area (in km^2), Population, GDP (nominal, in USD).

Task 1. Create a header file called `country.h` and containing *only the signature* of your `class Country`. Your class should at least have the following members:

- Public fields: `name`, `area`, `population`, `gdp`
- Public methods: `Country()` (default constructor)

Hint: You may want to include <`string`> to use `std::string`.

Task 2. Create a file `main.cpp` that contains these lines:

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

int main() {
  Country Germany ();     // Create a country 'Germany'

  Germany.name = "Germany";
  Germany.area = 357168;

  // ... use Wikipedia to fill the rest

  std::cout << Germany.name
            << " has "
            << Germany.population
            << " inhabitants." << std::endl;
}
```

Test your code. Create similarly a new `Country` instance called `Lesotho`, and yet another one called `Colombia`. Use Wikipedia to input correct information about these countries.

Task 3. Add a method `double getDensity()` to your `Country` class, that returns the population density of the country (i.e. population divided by area). Which of your three countries is the most densely populated? The least?

2 Const-safety: The `const` keyword

Now that we have countries, we can think about interesting things to do with them. Here is a very silly demographic prediction (probably not true): In 2020, each country's population will be multiplied by 110%. This can be implemented by the following code:

```
double getPopulationIn2020(Country& myCountry) {
  return myCountry.population * 1.10;
}
```

Task 4. Test that this function works on `Germany`. Why do we use `Country&` and what does it mean? What happens if we use the following code?

```
double getPopulationIn2020(Country& myCountry) {
  myCountry.name = "Catland";
  return myCountry.population * 1.10;
}
```

The dilemma is as follows: Either we copy everything (poor performance, high memory usage) and our objects cannot be directly modified; Or we pass by reference (high performance, low memory usage), but then it seems that our objects can be modified. The `const` keyword solves this issue:

Task 5. Check that the following code *does not compile correctly*. Describe the effect of the `const` keyword.

```
double getPopulationIn2020(const Country& myCountry) {
  myCountry.name = "Catland";
  return myCountry.population * 1.10;
}
```

3 Dynamic dispatch: The `virtual` keyword

Let's now focus on humans.

Task 6. Create a `class Person` that has only one method: `sayHello()` of type `void`. Implement the function `Person::sayHello()`.

Task 7. Create a `class GermanPerson` that *inherits* from `Person`. Implement `GermanPerson::sayHello()`. Create also a `LesothoPerson` and a `ColombiaPerson`. Implement their respective `LesothoPerson::sayHello()` and `ColombiaPerson::sayHello()`. Test it with the following code:

```
int main() {
  GermanPerson   Hans;
  LesothoPerson  Nombeko;
  ColombiaPerson Raul;

  Hans.sayHello();
  Nombeko.sayHello();
  Raul.sayHello();
}
```

Hint: In Sotho (the language of Lesotho), "hello" is said "dumela". In Colombian Spanish, you can choose between ¡Hola!, Buenas, Buenos días, ¡Quibo!, ¿Qué más?, ¿Cómo vas?, ¿Cómo van las cosas?, ¿Qué cuentas?, ¿Qué has hecho?, Cómo me le va?, ¿Bien o qué?, etc. etc.

Task 8. We now wish to have a generic function `greet` that takes a person as input and makes that person say hello:

```
void greet(Person& human) {
  human.sayHello();
}
```

What does `Person&` mean? Try to call this function with a `GermanPerson`: What happens?

Task 8. To solve this issue, we need to specify that `sayHello` is re-defined by child classes — otherwise, the mother class will always prevail. This is done by using the `virtual` keyword (only in the class definition):

```
class Person {
  public:
    virtual void sayHello();
}
```

● **Task 9.** How to use `const Person& human` instead of `Person& human`? Hint: To specify that a function is const you must add the `const` keyword *after its signature*.

4 ● Move Semantics

Important remark: To deal with this section you must use a compiler compatible with C++11 (e.g. g++ with the --std=c++11 compiler option).

Consider the following problem: We have an object BigObject1 and we want to move it to BigObject2. After the move operation, BigObject2 should have all that was in BigObject1. In this particular scenario, we don't care about BigObject1 anymore.

Traditionally, we could (1) copy all the contents of BigObject1 into BigObject2, then (2) erase information from BigObject1.

C++11 offers another way to achieve this, called *move semantics*. "Moving" simply transfers state (or ownership) to another object.

```
#include <iostream>  // std::cout, std::endl
#include <algorithm> // std::move
#include <list>      // std::list

class ListHolder {
  public:
    std::list<std::string> m_tokens;

    // Traditional: Copy the list
    void setTokens(const std::list<std::string>& toks) {
        std::cout << "Using *copy* semantics" << std::endl;
        m_tokens = toks;
    }

    // Move semantics: Just take ownership
    void setTokens(std::list<std::string>&& toks) {
        std::cout << "Using *move* semantics" << std::endl;
        m_tokens = std::move(toks);
    }
};

int main() {
    // Create a list
    std::list<std::string> tokens;

    tokens.push_back("there");
    tokens.push_back("is");
    tokens.push_back("a-town");
    tokens.push_back("in-new-orleans");

    // Create a ListHolder
    ListHolder p;

    // Now, move the strings to the ListHolder
    p.setTokens( std::move(tokens) );

    // Copy instead
    // p.setTokens( tokens );

    // Print the first element
    std::cout << p.m_tokens.front() << std::endl;

    // Verify that the old list is now empty
    std::cout << tokens.size() << std::endl;
}
```

Explain: Why is moving better than copying? What are the difference between copy and move semantics? Are we violating the Rule of 3/0/5?