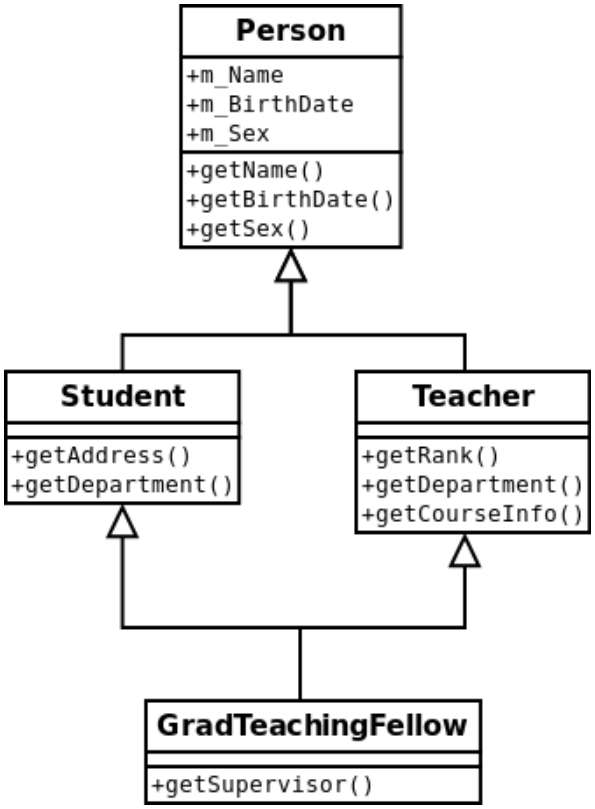
C++ lab 4

Recap of few concepts you used during the fourth lab:

1. Inheritance
   - Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application (because you can reuse the code).
   - A class can be derived from more than one classes, which means it can inherit data and methods from multiple base classes.
   - To define a derived class, you need to use a class derivation list to specify the base class(es) and the access specifiers:

```
class Person {
    protected:
        std::string name, birth_date, sex;
    public:
        std::string getName() const;
        std::string getBirthDate() const;
        std::string getSex() const;
};
class Student : public Person {
    protected:
        std::string address, departement;
    public:
        std::string getAdress() const;
        std::string getDepartement() const;
};
class Teacher : public Person {
    protected:
        int rank;
        std::string course_info, departement;
    public:
        int getRank() const;
        std::string getDepartment() const;
        std::string getCourseInfo() const;
};
class GradTeachingFellow final:
        public Student, public Teacher {
    private:
        std::string supervisor;
    public:
        std::string getSupervisor() const;
};
```



- If you add the `final` keyword, this means that you cannot inherit from your class anymore.
  For example if you try to do the following you will get an error from the compiler:

```
class A : public GradTeachingFellow {
    /* error: cannot derive from 'final' base 'GradTeachingFellow' */
}
```

- Access specifiers for inheritance are:
  - **public mode:** When deriving a subclass from a public class, the public members of the base class remain public and the protected members of the base class remains protected in the derived class.
  - **protected mode:** When deriving a subclass from a protected class, both the public and protected members of the base class become protected in the derived class.
  - **private mode:** When deriving a subclass from a private class, both the public and the protected members of the base class become private in the derived class.
  - If not specified the **default access specifier** is `private` if the derived type is defined as a `class` and `public` if the derived type is defined as a `struct`.

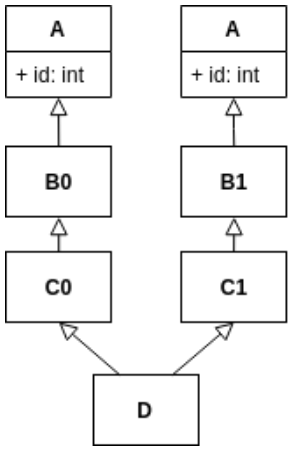
- You can access members of any base type with the scope resolution operator "`::`".

```
struct A {
    void print() const { std::cout << "A::print()  "; }
};
struct B: A {
    void print() const { std::cout << "B::print()  "; }
};
int main() {
    A a; a.print(); a.A::print();                  /* A::print()  A::print()          */
    B b; b.print(); b.B::print(); b.A::print();    /* B::print()  B::print()  A::print */
}
```

- To disambiguate base types in the case of multiple inheritance of the same type you can use the `static_cast` operator to cast the variable to a reference to one of the base type:

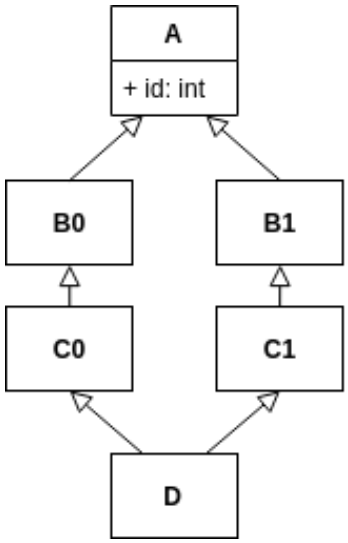
```
struct A {
    int id;
    A(int id): id(id) {}
    void print() const { std::cout << 'A' << id << "::print()  "; }
};
struct B0: A {
    B0(): A(0) {};
    void print() const { std::cout << "B0::print()  "; }
};
struct B1: A {
    B1(): A(1) {};
    void print() const { std::cout << "B1::print()  "; }
};
struct C0: B0 {
    void print() const { std::cout << "C0::print()  "; }
};
struct C1: B1 {
    void print() const { std::cout << "C1::print()  "; }
};
struct D: C0,C1 {
    void print() const { std::cout << " D::print()  "; }
};
int main() {
    D d;
    d.print();                          /* D::print()                          */
    d.C0::print(), d.C1::print();       /* C0::print()  C1::print()            */
    d.B0::print(), d.B1::print();       /* B0::print()  B1::print()            */
    //d.A::print()                      /* error 'A' is an ambiguous base of 'D' */
    static_cast<C0&>(d).A::print();     /* A0::print()                         */
    static_cast<C1&>(d).A::print();     /* A1::print()                         */
}
```

- The access `d.A::print` is ambiguous because the any object of type `D` will contain two instances of type `A`.



- If you want to implement a type `D` that only contain one `A` object, you need to rely on `virtual inheritance`. This problem is known as the diamond problem. Once virtual inheritance is used, the constructor of any virtual base class has to be called explicitly in every constructors of each derived types (but will only be called one time from constructor of the most derived object).

```
struct A {
    int id;
    A(int id): id(id) {}
    void print() const {
     std::cout << 'A' << id << "::print()  ";
    }
};
struct B0: virtual A {  B0(): A(0) {}  };
struct B1: virtual A {  B1(): A(1) {}  };
struct C0: B0        {  C0(): A(2) {}  };
struct C1: B1        {  C1(): A(3) {}  };
struct D: C0,C1      {   D(): A(4) {}  };
int main() {
    B0 b0; b0.print();              /* A0::print() */
    B1 b1; b1.print();              /* A1::print() */
    C0 c0; c0.print();              /* A2::print() */
    C1 c1; c1.print();              /* A3::print() */
    D   d; d.print();               /* A4::print(), not ambiguous anymore */
    static_cast<C0&>(d).print();    /* A4::print() */
}
```



- You can implicitely upcast a class to any of its parent classes.
  You may explicitely downcast a class to any of its compatible child classes.

```
#include <algorithm> /* std::fill_n, fill an array with a constant */
struct A    {
    char i[1024];
    A(char a=0) { std::fill_n(i, 1024, a); }; // just fill array i with the value of a
    void print() const { std::cout << "A" << int(i[1023]) << ' '; }
};
struct B: A {
    char j[1024];
    B(char a=1, char b=2) : A(a) { std::fill_n(j, 1024, b); };
    void print() const { std::cout << "B" << int(j[1023]) << ' '; }
};
int main() {
    std::cout << "Size of type A is " << sizeof(A) << '.' << std::endl;  // 1024 bytes
    std::cout << "Size of type B is " << sizeof(B) << '.' << std::endl;  // 2048 bytes

    A   a0;         a0.print();     // A0
    A& ra0 =  a0;   ra0.print();    // A0
    A* pa0 = &a0;   pa0->print();   // A0

    B   b0;         b0.print();     // B2
    B& rb0 =  b0;   rb0.print();    // B2
    B* pb0 = &b0;   pb0->print();   // B2

    /* example of implicit upcasts from B to A, B& to A& and B* to A* */
    A  a1  =  b0;   a1.print();     // A1
    A& ra1 =  b0;   ra1.print();    // A1 (static dispatch of print())
    A* pa1 = &b0;   pa1->print();   // A1 (static dispatch of print())

    /* example of valid explicit downcasts A& to B& and A* to B* */
    B& rb1 = static_cast<B&>(ra1);  rb1.print();  // B2, ok because ra1 references b0 of type B.
    B* pb1 = static_cast<B*>(pa1);  pb1->print(); // B2, ok because pa1 points to b0 of type B.

    /* example of bad explicit downcasts from A& to B& and A* to B* */
    B& rb2 = static_cast<B&>(ra0);  rb2.print();  // B?, may segfault because ra0 references a0 of type A.
    B* pb2 = static_cast<B*>(pa0);  pb2->print(); // B?, may segfault because pa0 points to a0 of type A.
}
```