This chapter introduces the notion of base, derived, abstract and template classes. This leads us to present two new kind of functions called virtual and pure virtual functions. The concepts of inheritance and polymorphism are also discussed. The last section shows how to resolve circular dependency issue when two classes are using each other.

1. Inheritance: Base and Derived classes

Object oriented programming languages, like C++, introduce the concept of inheritance. It allows programmers to define class, called derived class, based on another class refers as base class. Such feature comes is handy when programming classes that present similitude (like sharing a large number of variables or methods).

1.1. Definition and properties. A derived class, named name-derived_class, is defined with respect to a base class, named name-base_class, with the following instructions:

```cpp
class name-derived_class : set_inheritance name-base_class{
  // instructions
}
```

where set_inheritance is a access specifier. It can be equal to public, protected or private. If not present, it is considered to be equal to private. This specifier tells the programs how the members of the base class should be considered by the derived class. We note that a derived class has only access to the public and protected members of a base class. Depending of the value of the access specifier set_inheritance, the access specifier of the base class members's may be changed. Here is a description of the action of the three different access specifiers:

- **public.** The public, respectively protected or private, members of the base class remains public, respectively protected or private, for the derived class.
- **protected.** All the public members of the base class becomes protected for the derived class. The other members access are unchanged.
- **private.** All the members of the base class becomes private for the derived class.

A derived class inherits from all the public/protected variables and methods members of the base class except the following methods:

- constructors.
- destructors.
- assignment operators (=).
- friend functions (or operators).
We remind that the access specifier can modify the access of certain members of the base class such that public variables are not accessible if set_inheritance is set to private. Although the constructors and destructors of the base class are not inherited by the derived class, they are still used during the creation of an object of the derived class. Indeed each call to a constructor of a derived class is preceded by a call to the default constructor of the base class (to ensure every members are allocated even the private one). Similarly a call to the destructor of an object of a derived class is followed by a call to the destructor of the base class. The example described in the next section shows how the constructor of a derived class can be defined to call a specific constructor of the base class (and not the default one).

It is possible to define derived class based on multiple base class (multiple inheritance). Such classes are defined as follows:

```cpp
class derived_class : set_inhe1 base_class1, set_inhe2 base_class2{
    // instructions
};
```

1.2. Example. The example ex32 derived class.tar.gz, available on CAAM 510's webpage, introduces a base class named Box_2D and two derived class named Rectangle and Triangle defined as follows.

```cpp
class Box_2D{
    protected:
        double width;
        double height;
    public:
        void Set_Width(double);
        void Set_Height(double);
        double Get_Width(void);
        double Get_Height(void);
};
class Rectangle: public Box_2D{
    public:
        double Get_Area(void);
};
class Triangle: protected Box_2D{
    public:
        Triangle(double, double);
        double Get_Area(void);
};
```

The example focuses on the use of different access specifiers and the call of base/derived class’s constructors. We refer to the class for more details.

2. Polymorphism: virtual functions and virtual inheritance

The concept of polymorphism lies on the idea that one interface can be used to refer to multiple entities of different types. There is three main types of polymorphism in C++. The first one is named Ad hoc polymorphism. It consists of using the same name for functions that present
different arguments and instructions. It was introduced earlier in the course with the overloading of functions. The second type of polymorphism is called parametric. It consists of implementing a part of a code (like a class or function) without setting a specific type for the variables used as members or arguments. This type of polymorphism appears when creating template functions or template classes that are introduced in the section. The last type of polymorphism is called subtype. It allows the use of an object of a derived class as a reference of an object of the base class. In addition, a pointer to a derived class is compatible with a pointer of the base class. For instance, a function using a reference to an object of a base class A can also be called with an object of a derived class of A. Similarly, a pointer of the base class A can be used to point to an object of a derived class of A.

Such features are very appreciable when programming. However, the process of converting an identifier (like a variable or a function call) into a memory address that contains the value/implementation of the identifier, process known as binding, may rise issues during the compilation of the executable. For instance to support Ad hoc polymorphism, C++ uses a specific name mangling that allows the compiler to tell the difference between two overload functions. When using subtype polymorphism, programmers face an other issue as C++ uses a static binding by default. Static binding is done during the compilation of the executable. Such binding is not compatible with subtype polymorphism when the base and derived classes contain methods (functions) with the same signature. For instance, it makes a pointer of the base class to always use the methods as defined in the base class even if it points to an object of a derived class that contains a different definition of the methods. To face this issue C++ allows to use dynamic binding, done during the execution of a program, for the methods of a class by introducing the concept of virtual functions.

The following describes how to declare and use virtual functions. A description of the concept of virtual inheritance is also provided. An example that enhances the difference between a static and dynamic binding is available on CAAM 519 webpage (see ex33_virtual_functions.tar.gz).

2.1. **Virtual function.** To enforce a dynamic binding, a function member of a class can be declared as a virtual function by adding the specifier virtual as the beginning of its declaration as follows.

```cpp
class base_class{
public:
    virtual type_output name_func(type_inputs);
};
```

The keyword virtual should not be added in the function definition if the definition is done outside the class definition, meaning in a source file and not the header file that contains the class. A virtual function can only be a member of a class.

2.2. **On the use of virtual functions.** As mentioned earlier, the declaration of a class's function as virtual enforces a dynamic binding. It allows to call the correct function of a derived class when subtype polymorphism is involved. For instance, let us introduce a class B derived from a class A that both contain a function called "print" with no input and output. If the function print is not declared as virtual in the class A, the following instructions:

```cpp
A * pt_A;
B b;
pt_A=&b;
pt_A->print();
```
are going to execute the function print as it is defined in the class A and not the class B. As a consequence, a programmer would declare the function print as virtual in the class A such that the correct function is executed when running the program.

Since the standard C++11, two specifiers connected to virtual function were introduced. They are named final and override. Here is a description of their operation.

- **final.** It can be added at the end of a virtual function declaration as follows:

  ```cpp
  virtual type_output name_func(type_input) final;
  ```

  It prevent a function to be overwritten in derived class.

- **override.** This specifier can be used to check that a function of a derived class is indeed overriding a virtual function of the base class. It is used by adding the specifier override at the end of the function’s declaration as follows:

  ```cpp
  virtual type_output name_func(type_input) override;
  ```

  If no virtual function have the same signature (name and outputs/inputs) in the base class, a compilation error is generated.

We refer to the example ex33_virtual_functions.tar.gz for more details.

**Remark 1:** The same way than virtual function can be declared final to not be overwritten by derived class’s instructions, a class can also be declared as final. Such classes can not be used as a base class and are defined as follows.

```cpp
class name_class final{
  // instructions
};
```

**Remark 2:** The destructor of a base class should be declared as virtual. It avoid to get undefined behavior when allocating and deallocating dynamic memory to store object of a derived class with the operator new and delete. These operators will be introduced in the next chapter.

### 2.3. Virtual inheritance.

The virtual inheritance is a feature of C++ that allows a grandchild of a base class, meaning a class derived from a derived class of the base class, to inherit the members of the base class without ambiguity. We consider the following set of classes to see what problems may arise.

```cpp
class A{
  public:
    int a;
};
class B: public A{
};
class C: public A{
};
class D: public A{
};
```

When trying to access the member "a" of the class D as follows:
```cpp
int main(void){
  D d;
  d.a = 1.0
  return 0;
}
```

A compilation error appears (ambiguous definition). It is due to the fact the object D does not know if it should use the definition of the integer member \"a\" inherited from the class B or the one inherited from the class C. To face this issue, programmers use virtual inheritance. This feature allows derived class of derived class to have access to the members of a base class without ambiguous definition. It is done by adding the specifier virtual after the inheritance specifier in the definition of the classes B and C as follows:

```cpp
class B: public virtual A{
};
class C: public virtual A{
};
```

The base class A is then often referred as virtual base class. We refer to the class and the example ex34_virtual_inheritance.cc for more details.

### 3. Abstract classes and interfaces. Pure virtual functions

Abstract and interface classes are two kind of classes that can not be used to create object, i.e. they cannot be instantiated. They are only used to create a blueprint for multiple derived class. Theses classes involve the concept of pure virtual functions introduced in the following.

#### 3.1. Pure virtual functions

A pure virtual function is a virtual function of a class that is not defined for the class. It is done by adding the instructions \"=0\" at the end of the virtual function as follows:

```cpp
virtual type_output name_function(type_input)=0;
```

A class that contains a pure virtual function can not be instantiated, i.e. used to create object. Any of its derived class has to provide a definition of the pure virtual function or declare it as a pure virtual function as above.

Although a class with a pure virtual function consider that this function is not defined in the class, it is possible to provide an implementation (\"definition\") of the function. This implementation has to be done outside the class in a source file. Derived classes can then use this implementation in their definition of the pure virtual function as follows:

```cpp
type_output name_function(type_inputs inputs){
  return name_base_class::name_function(inputs);
}
```

where the keyword return can be removed in type_output is void.
3.2. **Abstract classes.** An abstract class is a class that:

- contains at least one pure virtual function.
- can contain data members like integer or double.
- can contain methods and virtual functions.
- can provide an implementation of pure virtual function outside the class in a source file. This implementation can only be used by derived classes when providing a definition of the pure virtual function.

For instance, we can consider the base class Box_2D of the example 32 and turns in into an abstract class by adding the following pure virtual function. We refer to the example 35 for more details.

```cpp
class Box_2D{
public:
    virtual double Get_Area(void)=0;
};
```

Abstract classes are mainly used to create a blueprint for a set of classes that present a lot of similarities like containing data members of the same type or functions members with the same signature.

3.3. **Interface classes.** An interface class is a class that contains only pure virtual functions and a virtual destructor. The destructor should not contain any instructions. It has to be virtual for compatibility between an interface’s pointer and the operator delete that will be introduced later in class. Unlike abstract classes, interfaces do not contain data members or methods (functions or virtual function). Moreover, they can’t contain implementations of the pure virtual functions outside the class. Such classes are mainly used to define a blueprint of specific features that may be common to different kind of classes. The name of an interface class usually starts with the capital letter I for interface.

4. **Template of functions and classes**

The C++ programming language allows the definition of functions and classes that use template parameters. These parameters represents a generic type that can be used by the template function/class to perform instructions or to declare the class’s members. Such feature allows to avoid the definition of multiple overload functions or nearly identical classes. The definition of a function and a class template has to be as independent as possible of the type of variables it involves such that it can be used with various types.

4.1. **Function template.** The prototype of a template function, that uses a template parameter named T, is declared using one of the following ways:

```cpp
template <typename T> output_type name_function(type_inputs);
```

or

```cpp
template <class T> output_type name_function(type_inputs);
```

The name T could be changed to any identifier like MyType. The use of the keyword typename or class is equivalent in this frame. Note that the type of the inputs and output of the function
may be of the type T. A template function can involve more than one template parameters. The different template parameters are then separated by coma.

```
template <typename T1, typename T2> type_output function(inputs);
```

Here is a short example on how to define a function add that adds two variables of various types without involving overload functions.

```
template <typename T>
T add(T a, T b){
    return a+b;
};
```

The function can then be called with different types like integer or double.

```
int i = add<int>(2.3,3.9);  //returns 5
double d = add<double>(2.3,3.9); //returns 6.2
```

4.2. Class template. The same way that C++ allows template function, a programmer can define a template class with the following instructions.

```
template <typename T>
class name_class{
    //declaration data members
    //definition method members
};
```

Once again "typename T" can be changed with "class T" where T represent the template parameter. The creation of objects of the class name_class required to set the template parameter to a specific type. Here is an example:

```
name_class<int> obj1;
name_class<double> obj2;
```

where we assume that the definition of the class name_class is relevant when the template type T is replaced by int or double.

Unlike other type of classes (base, derived, abstract), the methods of a template classes should be defined in the header file where the class is defined and not in a source file. If they are included in a source file, you may get compilation error during the generation of your executable. It is due to the fact that when instantiating an object of a template class, the compiler needs to have access to a definition of the class’s methods with the the correct type. If this definition is in an other source file using template type, the compiler will have trouble to find a function that matches. The easiest method to fix this issue is to put the definition of a template class's methods in the same header file. When a method of the template class name_class is defined outside the class definition, but still in the same header file, its definition is implemented with the following structure.

```
template <typename T>
type_output name_class<T>::name_function(type_inputs){
    //instructions
}
```
We note there is ways to include the method’s implementation of a template class in an other file. One could include the methods’s definitions in an other kind of file (.tcc .tpp) and include it at the end of the header file that contains the template class. An other way uses source file but requires more information on which types will be used to instantiate the template class. For instance if the programmer knows that a template class will only be instantiated with the types int and double, the implementation of the template class’s methods can still be done in a source file by adding the following instructions at the end of the source file.

```cpp
template class name_class<int>

template class name_class<double>
```

As the above two methods either introduce a new type of file or are user dependent (limited to a predefined set of template types), I advise to keep the implementation of the methods of a template class in the header file. We refer to the class and example ex36_template.tar.gz for more information.

4.3. Non type template parameter. In addition to using type template parameters, a function or a class template can also be defined with a non type template parameters as follows:

```cpp
template <typename T, int N>
T add_N(T a)
{
    return a+N;
}
```

The above template function can be called as follows:

```cpp
double d = add_N<double, 5>(3.1);
```

If a programmer want to use a variable to set N, the variable has to a const.

```cpp
const int n = 5;
double d = add_N<double, n>(3.1);
```

Since the standard C++11, give the option ”-std=c++11” to the compiler, it is possible to provide a default value for a non type template parameter. Following the previous example, a programmer would set a default value of N to zero with the following instructions.

```cpp
template <typename T, int N=0>
T add_N(T a)
{
    return a+N;
}
```

If no value of N is given when calling add_N, the compiler will assume that N is equal to zero.

5. Note on circular dependency

This section focuses on the compilations error that can be generated when two different classes need to have access to object or pointer of the other class.
5.1. **Problem set up.** Let’s consider two classes A and B that both contain an object of the other class as members. A programmer could try to define such classes in two header files `my_class_A.hh` and `my_class_B.hh` as follows:

- `my_class_A.hh`
  ```cpp
  #include "my_class_B.hh"
  class A{
    public:
      B b;
  }
  ```

- `my_class_B.hh`
  ```cpp
  #include "my_class_A.hh"
  class B{
    public:
      A a;
  }
  ```

The above implementation presents two problems. Firstly, if the compiler tries to allocate memory to store an object of A, it then needs to allocate memory for an object of B and so on. As a consequence, a programmer would include pointer of object as members and not an object itself. On the other hand, trying to compile a program that uses these header files leads to a compilation error as when the first header (class A) is included, it is going to include the next one (class B) and try to read its content. However, to define the class B the compiler needs to know what an element of a class A is, which is not known yet. Such problem can be faced by using forward declaration as shown in the next section.

5.2. **Use of forward declaration.** A solution of the above is to only include the header file in a source file `main.cc` and add forward declarations of the class B, respectively class A, in the header that defines the class A, respectively B. To do so, the previous headers would be modified as follows:

- `my_class_A.hh`
  ```cpp
  ifndef MY_CLASS_A_HH
  define MY_CLASS_A_HH
  class B; // Forward declaration
  class A{
    public:
      B* pt_b;
  }
  endif
  ```

- `my_class_B.hh`
  ```cpp
  ifndef MY_CLASS_B_HH
  define MY_CLASS_B_HH
  class A; // Forward declaration
  class B{
    public:
  }
  ```
The source file main.cc can then define objects of class A and B as shown here.

```c
#include "my_class_A.hh"
#include "my_class_B.hh"
int main(void){
    A a; B b;
    a.pt_b=&b; b.pt_a=&a;
    return 0;
}
```

Is it still possible to include the my_class_B.hh in the header my_class_A.hh and vice versa. However this is not required as the source file main.cc include both of the header files. This technique of forward declaration also allows to declare reference of a class’s object as member of an other class. However the reference needs to be initialized in the constructor. We refer to the example ex_37_circular_dependencies.tar.gz for more details.