COMP-SCI 201L (FS17) - **Section 2**

Lab 7 (10/24/17)

Number Converter using `class` Inheritance and Polymorphism

In this Lab, you are writing a **number converter** which reads in a natural number (either in binary, octal, decimal, or hexadecimal format) and then output the number in binary, octal, decimal and hexadecimal formats. To complete this job, a parent `class Natural_Number` is already written for you. You need to create **4 child classes**, `Binary_Number`, `Octal_Number`, `Decimal_Number`, and `Hexadecimal_Number`, which inherit the `public interface` of the parent `class Natural_Number`.

The reason why in this case we use **class inheritance** and **polymorphism** is because those classes may have the same function but different implementation of the function. For example, both decimal number and hexadecimal number can have a function `to_binary()`, which converts the number to binary format. However, converting a decimal number to binary format and converting a hexadecimal number to binary format are different. Therefore, the same function `to_binary()` has different implementations in `class Decimal_Number` and `class Hexadecimal_Number`. **Polymorphism** smartly solves this issue. When we declare the pointer, we can just use the parent class. For example, we can declare a pointer to `Natural_Number` and initialize it to safe value.

```
Natural_Number* p = NULL;
```

When we need to instantiate the pointer, we can let it be a child class.

```
p = new Hexadecimal_Number("2fc");
```

Please note that **`p`** is still a pointer to `Natural_Number`. It has **polymorphism character** so that it can be instantiated as a **child `class`** of `Natural_Number`. When we need to run **`p->to_binary()`**, the compiler will know exactly which implementation of function `to_binary()` to use. If **`p`** is currently instantiated as `Octal_Number`, then the compiler will use the function implementation in the Octal_Number class. If **`p`** is currently instantiated as `Decimal_Number`, then the compiler will use the function implementation in the `Decimal_Number` class.

On the other hand, we have to admit that the important reason why we use **inheritance** is because the two `class`es have some same variables or member functions. For example, `Natural_Number` has a `string` variable called `expression`, and `Binary_Number` also has `string` variable `expression`. The only difference is that `Natural_Number`'s `expression` could be in any format, but `Binary_Number`'s `expression` can only be in binary format. Therefore, we can say that the parent `class Natural_Number` is more generalized and more abstract, and child `class Binary_Number` is more specific. There are also a lot of inheritance relationships in our everyday life. For example, animal *versus* mammal, mammal *versus* dog, bank account *versus* checking account, and so on. Actually, you already touched some inheritance and polymorphism issues in previous Labs. For example, when we overload the "`<<`" operator, we let it pass an output stream (`ostream`). In the `main()` function, we can either use output file stream (`ofstream`), or standard output stream (`cout`), because both two are different kinds of output stream.

Parent `class`

In the parent `class`, other than `private` and `public`, sometimes we need to use **`protected`**. The differences amongst the three is in Table 1.

Table 1. Different sections in the parent `class`.

Section	Accessibility
`private`	Anything defined in the `private` section can **only** be accessed within the same `class`. Child `class` **cannot** access the `private` section of its parent `class`.
`public`	Anything defined in the `public` section can be accessed everywhere.
`protected`	Anything defined in the `protected` section can be accessed in the same `class` **or in its child `classes`**. Anywhere outside the `class` or its child `classes` **cannot** access the `protected` section.

In parent class, we will learn a new keyword, **`virtual`**, in C++. In general, `virtual` means the function may (or must) have different implementation in the child `classes`. See Table 2 for details.

Table 2. Function declaration and implementation in the parent and child `classes`.

Function Declaration	Function Implementation
`[Normal Declaration];`	• The function **must and must only** be implemented in the parent `class`. • Child `classes` will use the implementation in the parent `class`.
`virtual [Normal Declaration];`	• The function **must** be implemented in the parent `class`. • Child `classes` **may** also implement function. • When calling the function, if the child `class` has its own implementation, then use it; it not, then use the implementation in the parent `class`.
`virtual [Normal Declaration] = 0;`	• The function **must not** be implemented in the parent `class`. • Each child `class` **must** implement the function. • The parent `class` is **pure abstract**, which **cannot be instantiated**.

Child `class`

The following code shows how to define a child `class`. Child `class` may have its own `private` variables, or `private`/`public` functions. However, if the parent `class` is **pure abstract**, the child `class` must implement **all the pure `virtual` functions in the parent `class`**.

```
class child_class : public parent_class {
  // All the public stuff in parent_class
  // will become public stuff in child_class
  ...
}
```

```
class child_class : private parent_class {
  // All the public stuff in parent_class
  // will become private stuff in child_class
  ...
}
```

Lab Instruction

1. Define **4 child `classes`**, `Binary_Number`, `Octal_Number`, `Decimal_Number`, and `Hexadecimal_Number`. Parent `class` `Natural_Number` is ready for you.
2. In each child `class`, you need the following.
   - `private` variable `decimal_value` of `int` type, which represents the decimal value of the number (whatever format).
   - `private` member function `get_decimal_value()`, which returns the decimal value of the number (whatever format).
   - **Constructor** which passes a `string` (default empty `string`), which sets the variable `expression` to the `string` and calculates the `decimal_value`.
   - Declare and implement **all the pure virtual functions in the parent `class`**, which return the `string` expression of the number in binary, octal, decimal, and hexadecimal formats.
3. In the `main()` function, you need to **read all the numbers** one-by-one from the input file. Each number will be one line in the input file, which contains the format (`Bin`, `Oct`, `Dec`, or `Hex`) followed by the expression. Then, you program should output each number as binary, octal, decimal, and hexadecimal forms to the output file (separated by a single space). **Sample input and output files are provided in Blackboard**.

Submission and Grading

1. Submit your code via Blackboard **no later than Thursday, 10/26/17 @ 11:59 PM**.
2. Submit **all the `.h` files and `.cpp` files except `Natural_Number.h` (unzipped)**.
3. Your code will be graded using another input file (still named `input.txt`) to see if it can generate the expected output.
4. The grader will also look at your code manually to see if the code quality meets our other requirements.
5. **You cannot leave the lab early unless you finished the Lab assignment and submitted your work via Blackboard!** You have **only one attempt** to submit your work.