ESS201: Programming-II
Module: C++

Jaya Sreevalsan Nair
jnair@iiitb.ac.in
International Institute of Information Technology, Bangalore

Term I: 2018-19; Lecture-02 on October 10, 2018

1 Classes

- **What are C++ classes?**
  - The C `struct` bundles different data of various types together in a logical cluster, and the C `union` also defines data members of various types.
  - A C++ `class` is an extension of `struct` but with the default setting of all members being *private*, i.e. inaccessible outside of the class.
  - Thus, a `class` is the primary building block for data structures in C++, which upholds two key concepts in software engineering, namely, *data hiding* and *encapsulation*.
  - A `class` is thus roughly a set of (logically grouped) data and the functions operating on those data.
- **Relating C++ `class` with `union`**:
  - Extending `union` in C and C++ includes data fields of different class types in its version of `union` Ű these are called *unrestricted unions*.
  - Both extensions in C++ (i.e. `class` and unrestricted union) allow definition of *member* functions within the data types, unlike in C. *Member functions* are defined as functions that can only be used within the scope of the concerned data types or its instances/objects.
- **Difference between the class *interface* and its *implementation***:
  - Interface is a definition, which defines the organization of objects of that class. Unlike variable definitions which result in memory allocations, interface does not result in memory allocations. Since definition of anything happens only once in C++, as a rule, the term *interface* is used instead of definition.
  - Class interface is specified in a class *header file*. The member functions that are declared in the interface are defined in the implementation.
  - The implementation is specified in the *source file*. The separability of implementation and interface is highly recommended for: (a) better readability, and (b) easier maintenance.
- **Anatomy of a class interface**, including recommended practices for class interface setup:
  - Class member functions are called *class methods*, to differentiate from the non-member functions.

- *Constructors*, *destructors* and *copy constructors* are special member functions. They are, by default, included in the class implementation by the compiler using default functions, in case they are explicitly missing in the class interface.
- *Constructors* are special class functions which performs initialization of every object. The compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object. Constructors can be classified as *default*, *parametrized* and *copy*.
  * Default constructors do not require input parameters, whereas parametrized ones use input parameters. Copy constructors are important enough to be elaborated separately.
  * Compilers use default constructors if a constructor is not specified in the class interface.
  * A parametrized constructor can be re-used as a default one, by using default input parameters.
- *Destructors* are special class functions which destroy the concerned object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.
  * The compiler uses a default destructor, if not specified in the class interface.
  * The destructor is prefixed using a tilde $\sim$.
- *Copy constructors* are special type of constructors which takes an object as argument, and are used to copy values of data members of one object into other object.
  * The compiler uses a default copy constructor, if not specified in the class interface.
  * The customized copy constructor is for deciding what level of (i.e., shallow or deep) is required.
- *Manipulators* are member functions that allow modification of private data members. They are also referred to as *set-functions*, since by convention, names of manipulators start with `set`.
- *Accessors* are member functions that allow retrieval of private data members. They are also referred to as *get-functions*, with the accessors names starting with `get`. However, the use of get in function names is deprecated now, and simply (part of) the name of the data members themselves can be used.

All these functions can be overloaded.

- **Access modifiers/rights in the class interface**: There are three different access control rights/protocols in a C++ class (shown in Table 1).
  - `Public` class members and functions can be used from outside of a class by any function or other classes. Public data members or functions can be directly accessed by using dot operator (.) in case of object instances; and arrow operator $\rightarrow$in case of object pointers.
  - `Protected` class members and functions can be used inside its class; by friend functions and classes; and by classes derived from the class. Protected members and functions *cannot* be accessed from any other classes directly.
  - `Private` class members and functions can be used only inside of class and by friend functions and classes.

  We will revisit access control rights when we introduce *inheritance*.

  Few good practices to follow, for now:
  - The order of specification of functions in a class interface, on the basis of access rights is: `public`, `protected`, and `private`. This order provides easier readability to *users* of classes.
  - All data members are `private`, i.e. with private access rights, and are usually placed at the top of the interface.

Access Modifier	public	protected	private
Accessible within the same class	Yes	Yes	Yes
Accessible within friend classes and functions	Yes	Yes	Yes
Accessible within derived classes and functions	Yes	Yes	No
Accessible outside of the above	Yes	No	No

Table 1: Access control rights for C++ classes.

- Non-private data members *do* exist, but they must be used only as deemed necessary. e.g. Constructors, destructors, and copy constructors are by design (and hence, by default), `public` members. Copy constructors can be `private` as well.
- Default use of `public` data members is discouraged owing to three reasons: (a) violation of data encapsulation; (b) difficulty in tracking functions that change the value of the concerned member; and (c) the stability of the `public` interface, which is accessible to users of the class, is maintained.

2 Friends

Some functions have to be defined outside of a class interface. However, for some of them, according to the traditional definition of the class concept, those functions which *need not* be defined in the class interface itself should nevertheless be considered functions belonging to the class. That is, if permitted by the language syntax, they would certainly have been defined inside the class interface.

There are two ways to implement such functions.

1. Implementing those functions using available public member functions.
2. Applying the definition of the class concept to those functions, one can state those functions in fact belong to the class. Hence, they should be given direct access to the data members of objects. This is accomplished by the `friend` keyword.

3 Example Codes and Practicals

We shall use several examples to articulate various aspects of C++ programming.

1. Strict type-checking in C++ unlike C:
```
$ gcc typechecking.c; ./a.out
$ g++ typechecking.cpp ; ./a.out
$ g++ typechecking_corrected.cpp ; ./a.out
```
2. Function overloading; using g++ for C code; use of headers in C:
```
$ gcc function_overloading.c; ./a.out
$ g++ function_overloading.c ; ./a.out
$ g++ function_overloading.cpp ; ./a.out
```
3. Default function arguments in C++; using multiple files:
```
$ g++ default_func_parameters.cpp; ./a.out
$ g++ -c second_file.cpp default_func_parameters_variant1.cpp ; g++ -o output *.o ; ./output
$ g++ -c second_file.cpp default_func_parameters_variant2.cpp ; g++ -o output *.o ; ./output
$ g++ -c second_file.cpp default_func_parameters_variant3.cpp ; g++ -o output *.o ; ./output
```

Run this segment by deleting all object codes and executables between each experiment for the first round; and without deleting the same for the second round, to show how `second_file.o` *can be reused.*

Using `a.out`:

```
$ g++ default_func_parameters_variant1.cpp ; ./a.out
$ g++ second_file.cpp default_func_parameters_variant1.cpp ; ./a.out
$ g++ default_func_parameters_variant1.cpp second_file.cpp ; ./a.out
$ g++ default_func_parameters_variant2.cpp second_file.cpp ; ./a.out
$ g++ -c default_func_parameters_variant2.cpp second_file.cpp
```

4. Define local variables:

```
$ g++ local_variables.cpp ; ./a.out
$ g++ local_global_variables.cpp ; ./a.out
```

5. Writing a C++ class for `Matrix`:

   - `class-v0`: Setting up constructor, destructor, copy constructor and get functions; and demonstrating access to private data members.
   - `class-v1`: Setting up set functions; specifying scope (Matrix::) in the definition and implementation.
   - `class-v2`: Setting up friend functions.