# ESS201: Programming-II
# Module: C++

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

Term I: 2018-19; Lecture-07 on November 14, 2018

## 1 Inheritance: Conversions between Base and Derived Classes

When `public` inheritance is used to define classes, a derived class object can be treated as a base class object. Hence, care must be taken when using object assignment as well as pointers and references to such objects.

**Conversions with object assignments:**

```
Vehicle v(1000) ;
Truck t(2000, 300, 400, "Mahindra") ;
v = t ; // permissible
t = v ; // gives compilation error
```

- *Conversion of derived class to base class is permissible.* The data members in the base class are assigned from the derived class object, and the other derived class members are ignored by the assignment. This process is called **slicing**. Slicing is inconsequential mostly, however, one must note it occurs in the following situations:
  - when passing derived class objects to functions defining base class parameters, or
  - when returning derived class objects from functions returning base class objects.

  As an aside, slicing is possible owing to the use of the *Liskov Substitution Principle* (LSP).
- *Conversion of base class object to derived class object gives compilation error.* This is because there is no information on how the data members belonging to the derived class, but not present in the base class, are going to be assigned. The governing principle is: *a derived class object is also a base class object, but a base class object is not also a derived class object."*

  However, this conversion can be done by overloading the assignment operator to allow assignment of a derived class object from a base class object. The programmer has the freedom to decide the assignment of the data members in the derived class, but are not present in the base class.

  ```
  Truck const &operator=(Truck const &other) ;
  Truck const &operator=(Vehicle const &other) ;

  Vehicle(1000) ;
  Truck t(2000, 300, 400, "Mahindra") ;
  t = v ; // permissible
  ```

**Conversion with pointer assignments**

```
Land_derv l(1000, 200) ;
Four_wheeler f(3000, 400, "Foo") ;
Truck t(6000, 4000, 300, "Bar") ;
Vehicle *vp ;

vp = &l ; // Permissible
vp = &f ; // Permissible
vp = &t ; // Permissible
```

- Assigning a base class pointer with a reference to derived class objects is permissible, as this is an implicit conversion of the derived class to a base class.
- The same holds true for references to base class. e.g., a function is defined having a base class reference parameter, the function may be passed an object of a derived class. Inside the function, the specific data members of the base class remain accessible. This analogy between pointers and references (as `const` pointers) holds true in general.
- However, if a function is overloaded in the derived class, such a conversion leads to calling the function in the base class as opposed to the derived class. This specific behavior is the consequence of the restricted behavior of conversion of derived class to base class.

  ```
  vp = &t ;
  size_t m = vp->mass() ; // gives the Vehicle mass, not the Truck mass
  ```

- When a function is called using a pointer to an object, then the type of the pointer (and not the type of the object) determines which member functions are available and which can be executed. In other words, C++ implicitly converts the type of an object reached through a pointer to the type of the pointer.

  If the actual type of the object pointed to by a pointer is known, an explicit type cast can be used to access the full set of member functions that available for the object:

  ```
  Vehicle *vp ;
  vp = &t ; // vp now points to a truck object

  Truck *tp ;
  tp = static_cast<Truck*> (vp) ; // Using static type casting for reversing
  ```

  It is to be noted that the static type casting (as with all type casting) works *only if* `vp` really points to a `Truck` object. Otherwise, it can give unexpected results.

# 2 Miscellaneous

**Conditional Operator in C++**
Using a ternary operator.

```
if (condition) var = x ;
else var = y ;
```

can be rewritten in C++ using a ternary opertor, as follows:

```
var = (condition) ? x : y ;
```

For example:

```
int var, x = 35, y = 46 ;
bool condvar = (x==y) ;
var = (condvar) ? x : y ;
```

**Core Principles of Object-Oriented Programming**
Using the mnemonic acronym, **SOLID**

- **S for Single Responsibility Principle**: A class should have only a single responsibility.
- **O for Open/Closed Principle**: Software entities should be open for extension, but closed for modification.
- **L for Liskov Substitution Principle**: Objects should be replaceable by its subtypes, without altering correctness.
- **I for Interface Segregation Principle**: Instead of a generic interface, several client-specific interfaces should be created.
- **D for Dependency Inversion Principle**: One should depend on abstractions as opposed to concretions.

**Core Principles of Programming** Using few more mnemonic acronyms.

- **DRY Principle**: Do not Repeat Yourself
- **KISS Principle**: Keep It Simple, Stupid
- **YAGNI Principle** You Ain't Going to Need It

# 3 Example Codes and Practicals

Folder specifications:

1. `v3`: Overloading assignment operator in derived class to allow conversion of base class object to derived class object.