Terms and C++ keywords used in Object-Oriented Programming (OOP)

Abstraction is the separation of the essential logical properties of an object or action from the implementation details of that object or action. Informally, you could define it as *defer the details*.

Data Abstraction is the separation of a data type's logical properties from its implementation details (e.g., a class).

Control Abstraction is the separation of an action's (or task's) logical properties from its implementation details (e.g., a function).

A **Data Type** is the specification of the properties (domain and operations) of a variable or other data object of that type.

An **Abstract Data Type** (ADT) is a data type whose properties (domain and operations) are specified independently from any particular implementation.

The **domain** of a data type (abstract or not) is the set of legal values for that data type.

The **operations** of a data type (abstract or not) is the set of possible actions on the values in the domain.

The **Data Representation** of an ADT is the concrete form of data used to represent the abstract values in the domain of the ADT.

A data type is **structured** if its elements are arranged in a fixed manner in memory, either physically (e.g., an array) or logically (e.g., linked).

A data type is **unstructured** if its elements are not required to be arranged in a fixed manner in memory (e.g., the system heap).

A **class** is a structured data type in a programming language that is used to represent an ADT.

(A class may informally be called an ADT. More precisely, it is the implementation of an ADT.)

A **class member** is a component of a class. Members may be either data or functions (methods).

An **object**, **class object**, or **class instance** is a variable of a class type.

Objects have **state**, **behavior** and **identity**.

The **state** of an object is the current set of values in its data members.

The **behavior** of an object is the set of methods available in the object.

Identity is that property of any object that differentiates it from all other objects of the same type. For a class object, its identity will normally be its address in RAM.

Encapsulation (or **Information Hiding**) is the deliberate hiding of the implementation details of an abstraction, thus preventing the user of that abstraction from depending upon or incorrectly manipulating those details.

An **Abstraction Barrier** is an "invisible wall" around an object or a function that encapsulates the object's or function's implementation details. The wall can be breached only through the public interface (the contractual interface).

A **client** is software that declares and manipulates objects of a particular class. The client of a class may be an application or another class.

There are five categories of ADT operations (implemented as class methods or friend functions):

A **constructor** is an operation that creates a new instance of a class object.

A **destructor** is an operation that destroys an instance of a class object.

A **mutator** (**transformer**, **manipulator** **or** **setter**) is an operation that builds a new value (state) of a class object, given one or more previous values of the same type.

An **observer** (**inspector**, **accessor** **or** **getter**) is an operation that allows us to observe the state of a class object without changing it.

An **iterator** (not all ADTs have these) is an operation that allows us to process, one at a time (one per call), all of the components of a class object.

The **public interface** (**contractual interface**) of a class is the set of public methods (of all five types) by which the client can manipulate objects of that class.

A **black box** is an electrical, mechanical or logical device whose inner workings are hidden from view.

A **white box** is an electrical, mechanical or logical device whose inner workings are exposed.

Aggregation is the process of using a class object (of a class defined outside the client class) as a data member of another (client) class. Aggregation implements the **has-a** relationship. Members of the **enclosed class** are normally accessible within the **enclosing class** only via the public interface of the enclosed class. A similar process, **composition**, is the use of a class, defined within another class, to create a data member of the enclosing class.

Inheritance is the process of bringing forward all the members of a class (data and methods) into a new class as members of that new class. The inherited members do not include the constructors, destructor or friends. Inheritance implements the **is-a** relationship. In **public** inheritance, members of the **base class** are accessible in the **derived class** if the base class members are protected, but not if they are private.

Polymorphism (from Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form. In OOP, a pointer or reference to a base class may be type-compatible with any derived class.

Access keywords for classes:

public denotes members (data or methods) that may be accessed whenever a client program has access to an object of that class type.

private denotes members (data or methods) that may be accessed only by member functions and friend functions.

friend denotes other classes, methods from other classes, or external standalone functions that have permission to access private data or methods in a class object, provided they have access to that object (e.g., as a reference parameter).

Friendship may only be granted by a class, it cannot be imposed on the class from outside.

protected denotes members (data or methods) that may be accessed only by member functions and friend functions in a base class, and that may also be accessed by member functions and friend functions in a derived class. If a class with private members is used as a base class in public inheritance, the derived class will not be able to access the private members it inherits from the base class, so *protected* preserves encapsulation in the derived class, but allows the derived class to operate on the members as though they were actually declared as members of the derived class.

virtual denotes member functions in a base class that must be redefined in a derived class.

An **abstract class** is a class containing a virtual method assigned the value 0. You cannot implement an abstract class. It must therefore be used as a base class. For example:

```
class ABSTRACT
{
   public:
   virtual int foo() = 0;  // foo() is an abstract method
              // so ABSTRACT cannot be implemented as an object
};
```

This may be used to specify the common behavior (public interface) of a set of derived classes.

The three inheritance access modifiers public, protected and private are analogous to the access modifiers used for class members.

public - When a base class is specified as public; i.e.: `class C : public base {...};` the base class can be seen by anyone who has access to the derived class. That is, any members inherited from the class base can be seen by code accessing the class `C`.

protected - When a base class is specified as protected; i.e.: `class C : protected base {...};` the base class can only be seen by `C` or subclasses of `C`, not from the client of `C`.

private - When a base class is specified as private; i.e.: `class C : private base {...};` the base class can only be seen by the class `C` itself, not even from derived classes of `C`.

Normally, you use public inheritance, except under special circumstances.