C++ AND OBJECT ORIENTED PROGRAMMING
2. C++ AND OBJECT ORIENTED PROGRAMMING

2.1 EVOLUTION OF C++

When a program's size increases, it becomes hard to manage because it is difficult to grasp as a totality. In 1980, while working at Bell Laboratories at Murray Hill, New Jersey, Bjarne Stroustrup addressed this problem by adding several extensions to the "C" language. Initially Called "C with Classes", the name was changed to C++ in 1983.

Most additions made to C by Stroustrup support Object-Oriented Programming, sometimes referred to as OOP. Stroustrup states that some of C++'s Object-Oriented features were inspired by another object-oriented language, SIMULA 67. Therefore C++ represents the blending of two powerful programming methods.

The work on what eventually became C++ started with an attempt to analyze the Unix kernel to determine how it could be distributed over a network of computers connected by a local area network.

The name C++ was coined by Rick Mascitti in the summer of 1983. The name signifies the evolutionary nature of the changes from C. The language is not called D, because it is an extension of C and does not attempt to remedy problems by removing features [3]. It has had two major revisions since
it was invented, once in 1985 and again in 1989. Borland began development of C++ in 1988, and released its product in May 1990 [18].

2.2 OBJECT ORIENTED APPROACH

The single most important feature of an Object Oriented language is the OBJECT. It is a logical entity containing data and code that manipulates that data. The concept of class is static, because its definition is written in the code of the program. The concept of Object, instead, is dynamic - because it exists only at run time and denotes a location in the computer memory.

Scot Robert Ladd [47] states that classes allow abstraction of complex areas of the program, so that we are not forced to be concerned with details. The class contains data members and member functions. It is by means of this construct that C++ realizes the strong coupling between data and functions, which is one of the strong traits of the Object Oriented languages.

Data encapsulation offers a higher degree of security, forbidding direct access to private data of a class. When software design is understood as operational modelling, object-oriented design is a natural approach: the world being modelled is made of objects and it is appropriate to organize the model around computer representations of these objects.
Nowhere perhaps is this view of software as inescapable as in the area of simulation [1]. It seems hard to devise a better structure for simulation programs than one which is directly patterned after the objects whose behaviour is being simulated.

AT & T is rewriting Unix in C++ because C++ improves the reliability, maintainability, and reusability of code. Apple is using C++ to develop system software for its Macintosh line for the same reason and because OOP techniques are a natural match to program features such as windows and dialog boxes [50].

Among the member functions of a class, two have a particular role: Constructor[s] and the Destructor of the class. The Constructors, which have the same name as the class are the functions called to perform the initialization of the objects. The opposite of a Constructor, a destructor is a function that is automatically called at the end of the "life" of the object, i.e., when the object is no longer needed in the system. This cleans up the memory used by the object.

The class data members and functions belong to any one of the following three categories:

Private, Protected, Public,
If the class members cannot be accessed or called from outside the class, then they are said to be private. On the other hand if they can be accessed or called from outside the class, they are Public. If the data members and functions are visible and can be accessed or called only from within those classes that are derived from the class where they have been declared then they are said to be Protected. The class construct has by default, Private members.

An object provides a significant level of protection against accidental modification or incorrect use. The linkage of code and data in this way is generally called as **ENCAPSULATION**.

### 2.2.1 VIRTUAL FUNCTIONS

Virtual means existing in effect but not in reality [28]. A virtual function is one that does not exist but appears real to some parts of a program. Virtual functions overcome the problems with the type-field solution by allowing the programmer to declare functions in a base class that can be redefined in each derived class. The compiler and loader will guarantee the correct correspondence between objects and functions applied to them. The keyword **virtual** indicated that the function can have different versions for different derived classes and that it is the task of the compiler to find the appropriate one for each call of the function.
2.2.1.1 STATIC BINDING AND DYNAMIC BINDING

Binding generally denotes the connection between an entity and its properties. Limited to functions, the main connection is between the function call and the code that its call executes. In static binding, which is available in most procedural languages, the compiler and the linker directly define the fixed location of the code to be executed on every function call. In dynamic binding also known as late binding, the compiler does not define the location called, instead waits until Run Time to decide which of a list of locations is the actual one. Only the execution of the program will determine the effective code set that is to be executed.

2.2.2 POLYMORPHISM

OOPs support polymorphism, which allows one name to be used for several slightly different purposes. More than one function can exist with the same name, and depending on the number or type of the parameters the respective function will be executed accordingly.

2.2.3 INHERITANCE

One of the powerful intellectual tools for managing complexity is hierarchical ordering, that is, organising related concepts into a tree structure with the most general concept at the root. In C++, derived classes represent such structures [2].
INHERITANCE is the process by which one object can acquire the properties of another object. Through the property of inheritance it is possible to create new classes called derived classes, from the existing base classes. The derived classes inherit all the capabilities of the base class, and can add data members and properties of its own. The base class is unchanged by this process.

Inheritance has the important advantage of code reusability. Once a base class is written and debugged, it need not be touched again but can nevertheless be adapted to work in different situations. It can also help in the original conceptualization of a programming problem, and in the overall design of the program.

Member functions can be used in a derived class that has the same name as those in the base class. When the same function exists in both the base class and the derived class, the function in the derived class will be executed.

The program development process, as practiced for decades by programmers everywhere is being fundamentally altered by Object Oriented Programming. This is due not only to the use of classes in OOP but also to inheritance as well [28].

C++ does not have built-in complex number, string, or matrix types, or direct support for concurrency, persistence, distributed computing, pattern matching, or file system
2.3 OBJECT ORIENTED APPROACH AND SIMULATION ENVIRONMENT

A variety of techniques can be used to represent and manipulate knowledge. Three techniques that are of particular relevance to simulation are

* Object-Oriented Programming
* Frames and Inheritance
* Graphics-based Editing

Frames provide the structure that supports Object-Oriented Programming [4]. The Frame concept fits very naturally into the structure of discrete-event simulation, offering the modeller a number of capabilities like

* Library development - New entities can be specified as members of a particular class with certain differences.
* Construction via component assembly - Systems can be assembled by selecting components from a library and then connecting those entities with various relations.
* Component representation - can be used to effectively manipulate sub components.