Lecture 1: why need OOP

This lecture teaches you how to program in C++, a computer language that supports *objectoriented programming* (*OOP*). Why do we need OOP? What does it do that traditional languages such as C, Pascal, and BASIC don't? What are the principles behind OOP? Two key concepts in OOP are *objects* and *classes*. What do these terms mean? What is the relationship between C++ and the older C language?

Why Do We Need Object-Oriented Programming?

Object-oriented programming was developed because limitations were discovered in earlier approaches to programming. To appreciate what OOP does, we need to understand what these limitations are and how they arose from traditional programming languages.

Procedural Languages

C, Pascal, FORTRAN, and similar languages are *procedural languages*. That is, each statement in the language tells the computer to do something: Get some input, add these numbers, divide by six, display that output. A program in a procedural language is a list of instructions.

For very small programs, no other organizing principle (often called a *paradigm*) is needed. The programmer creates the list of instructions, and the computer carries them out.

The idea of breaking a program into functions can be further extended by grouping a number of functions together into a larger entity called a *module* (which is often a file), but the principle is similar: a grouping of components that execute lists of instructions.

Dividing a program into functions and modules is one of the cornerstones of *structured programming*.

Problems with Structured Programming

As programs grow ever larger and more complex, even the structured programming approach begins to show signs of strain. You may have heard about, or been involved in, horror stories of program development. The project is too complex, the schedule slips, more programmers are added, complexity increases, costs skyrocket, the schedule slips further, and disaster ensues.

Analyzing the reasons for these failures reveals that there are weaknesses in the procedural paradigm itself. No matter how well the structured programming approach is implemented, large programs become excessively complex.
What are the reasons for these problems with procedural languages? There are two related problems. **First**, functions have unrestricted access to global data. **Second**, unrelated functions and data, the basis of the procedural paradigm, provide a poor model of the real world.

1.Unrestricted Access

There are two kinds of data. *Local data* is hidden inside a function, and is used exclusively by the function. However, when two or more functions must access the same data—and this is true of the most important data in a program—then the data must be made *global*. Global data can be accessed by *any* function in the program.

The arrangement of local and global variables in a procedural program is shown in Figure 1.

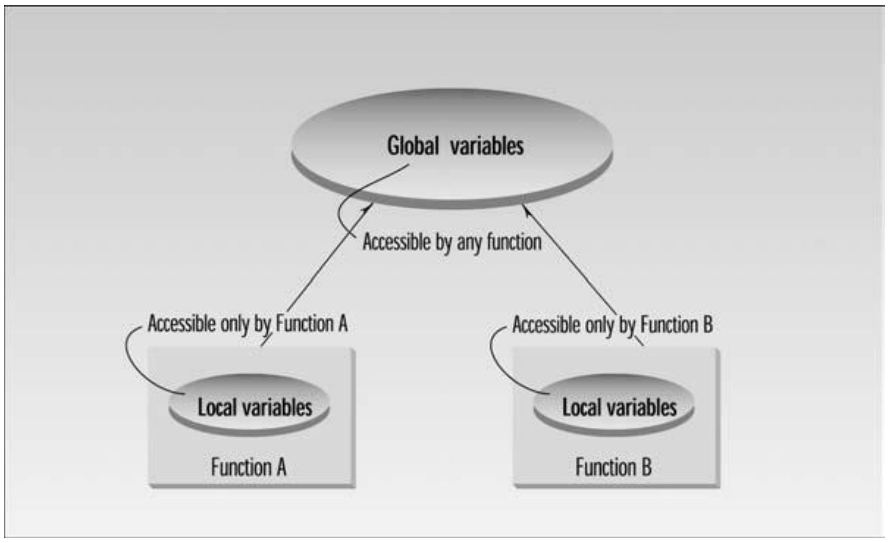
Global variables
Accessible by any function
Accessible only by Function A
Accessible only by Function B
Local variables
Function A
Local variables
Function B

FIGURE 1*Global and local variables*

In a large program, there are many functions and many global data items. The problem with the procedural paradigm is that this leads to an even larger number of potential connections between functions and data, as shown in Figure 2.

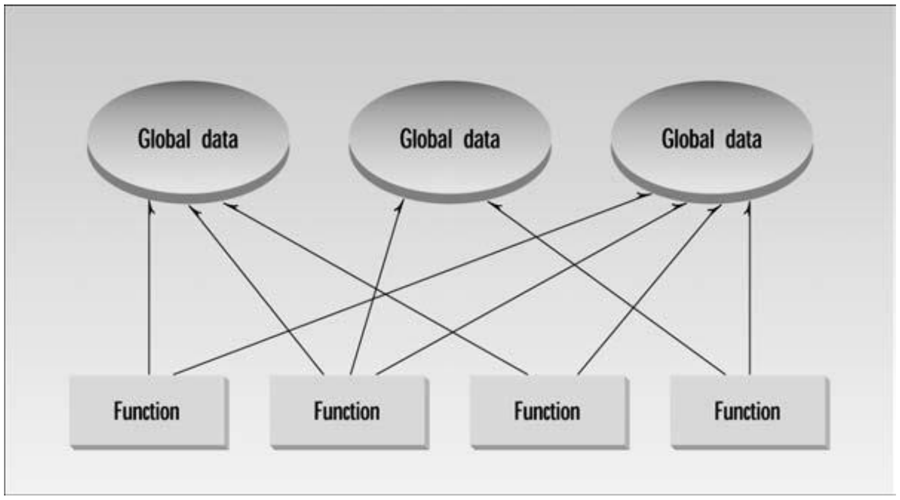

FIGURE 2 *The procedural paradigm*

This large number of connections causes problems in several ways. **First**, it makes a program's structure difficult to conceptualize. **Second**, it makes the program difficult to modify.
A change made in a global data item may necessitate rewriting all the functions that access that item.

2.unrelated functions and data

The second—and more important—problem with the procedural paradigm is that its arrangement of separate data and functions does a poor job of modeling things in the real world. In the physical world we deal with objects such as people and cars. Such objects aren't like data and they aren't like functions. Complex real-world objects have both *attributes* and *behavior*.
Attributes in the real world are equivalent to data in a program: they have a certain specific values, such as blue (for eye color) or four (for the number of doors).
Behavior is like a function: you call a function to do something and it does it.
So neither data nor functions, by themselves, model real-world objects effectively.

The Object-Oriented Approach

The fundamental idea behind object-oriented languages is to combine into a single unit both *data* and the *functions that operate on that data*. Such a unit is called an *object*.
An object's functions, called *member functions* in C++, typically provide the only way to access its data. The data is *hidden*, so it is safe from accidental alteration. Data and its functions are said to be *encapsulated* into a single entity. *Data encapsulation* and *data hiding* are key terms in the description of object-oriented languages.
A C++ program typically consists of a number of objects, which communicate with each other

by calling one another's member functions. The organization of a C++ program is shown in Figure 3.

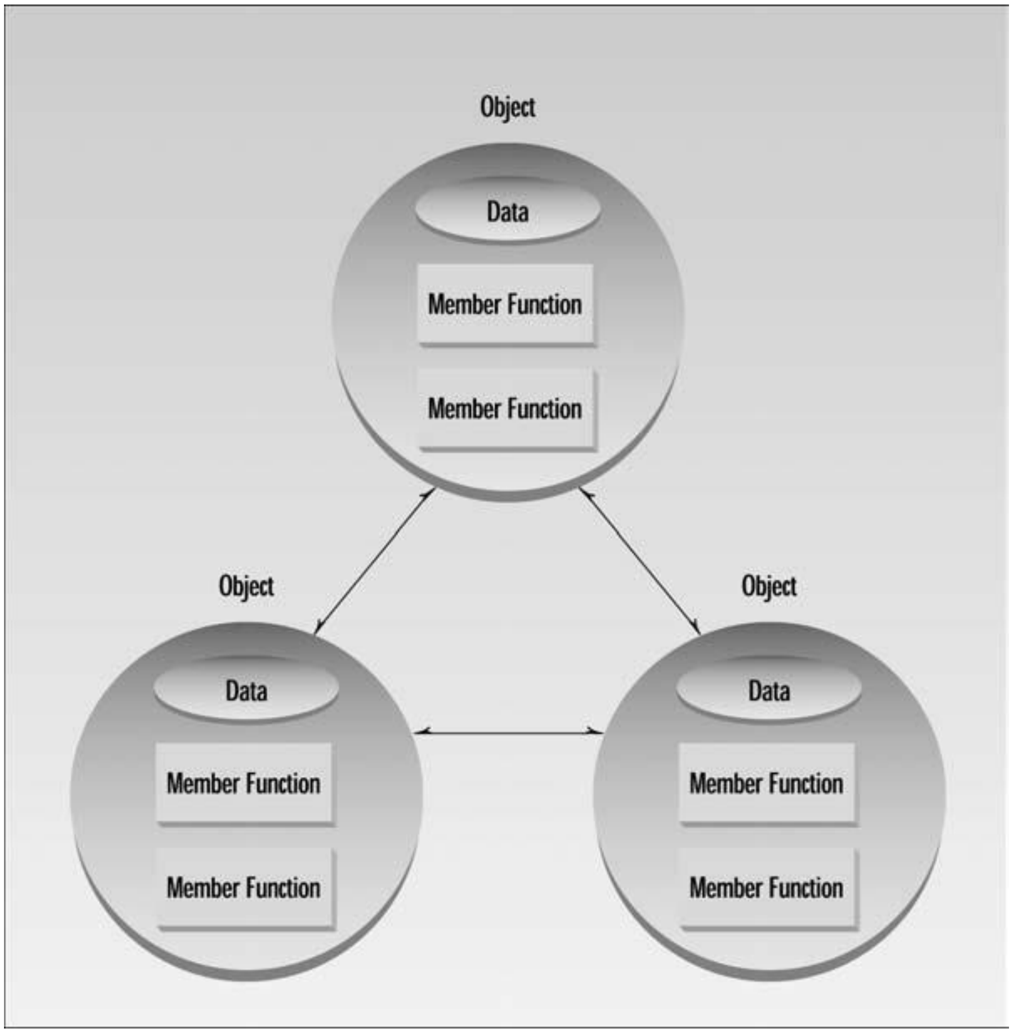

FIGURE 3 *The object-oriented paradigm*

Characteristics of Object-Oriented Languages

Let's briefly examine a few of the major elements of object-oriented languages in general, and C++ in particular.

<u>Objects</u>

When you approach a programming problem in an object-oriented language, you no longer ask how the problem will be divided into functions, but how it will be divided into objects. Thinking in terms of objects, rather than functions, has a surprisingly helpful effect on how easily programs can be designed. This results

from the close match between objects in the programming sense and objects in the real world.
The match between programming objects and real-world objects is the happy result of combining data and functions: The resulting objects offer a revolution in program design. No such close match between programming constructs and the items being modeled exists in a procedural language.

What kinds of things become objects in object-oriented programs? here are some typical categories to start you thinking:
• **Physical objects**
Automobiles in a traffic-flow simulation
Electrical components in a circuit-design program
Aircraft in an air traffic control system
• **Elements of the computer-user environment**
Windows, Menus, or Graphics objects (lines, rectangles, circles)
• **Data-storage constructs**
Customized arrays, Stacks, Linked lists, or Binary trees
• **Human entities**
Employees, Students, Customers, or Sales people
• **Collections of data**
An inventory, A personnel file, or A dictionary
• **User-defined data types**
Time, Angles, or Complex numbers
• **Components in computer games**
Cars in an auto race
Positions in a board game (chess, checkers)
Animals in an ecological simulation
Opponents and friends in adventure games

Classes

In OOP we say that objects are members of *classes*. What does this mean? Almost all computer languages have built-in data types. For instance, a data type int, meaning integer, is predefined in C++. You can declare as many variables of type int as you need in your program:

```
int day;
int count;
int divisor;
int answer;
```

In a similar way, you can define many objects of the same class, as shown in Figure 1.5. A class serves as a plan, or blueprint

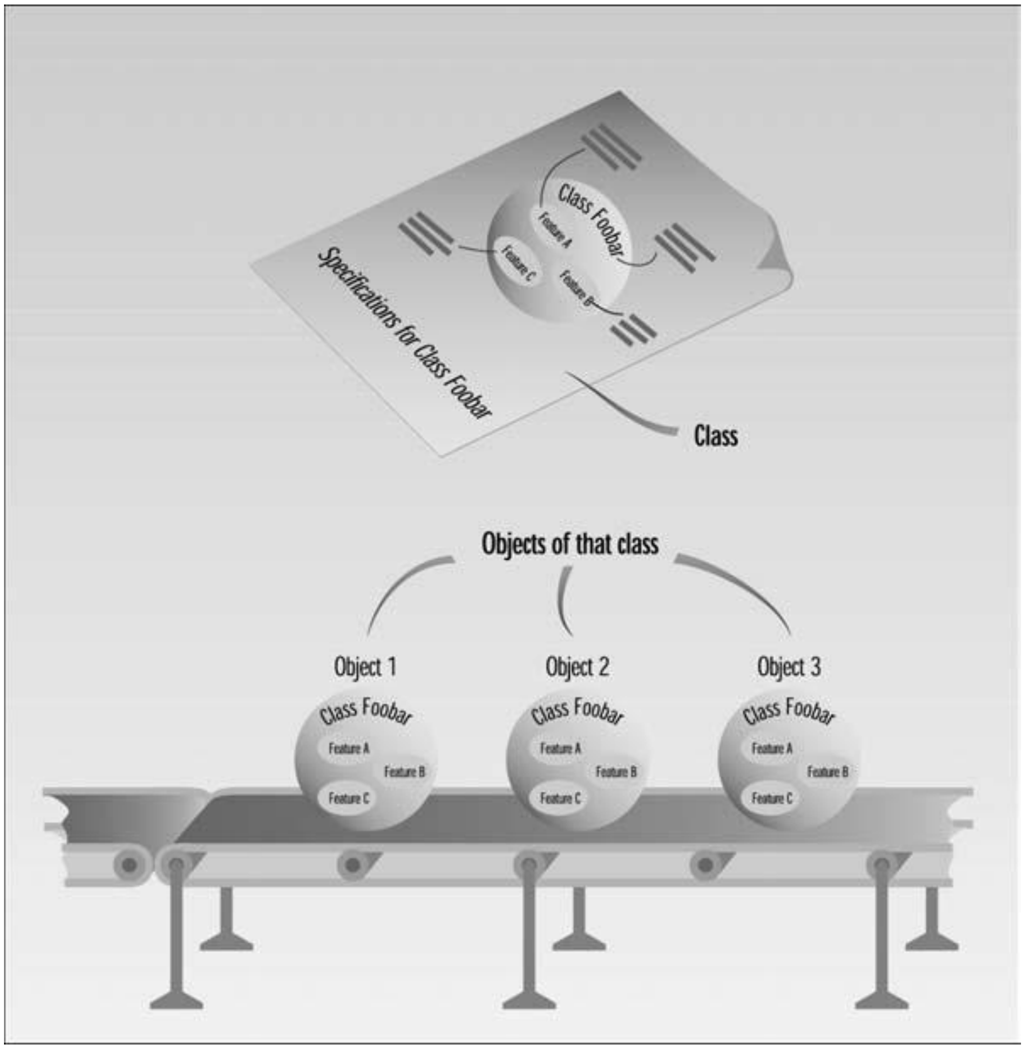

FIGURE 5 *A class and its objects*

Inheritance

The idea of classes leads to the idea of *inheritance*. In our daily lives, we use the concept of classes divided into subclasses. The principle in this sort of division is that each subclass shares common characteristics with the class from which it's

derived. In addition to the characteristics shared with other members of the class, each subclass also has its own particular characteristics. This idea is shown in Figure 6.
In a similar way, an OOP class can become a parent of several subclasses. In C++ the original class is called the *base class*; other classes can be defined that share its characteristics, but add their own as well. These are called *derived classes*.

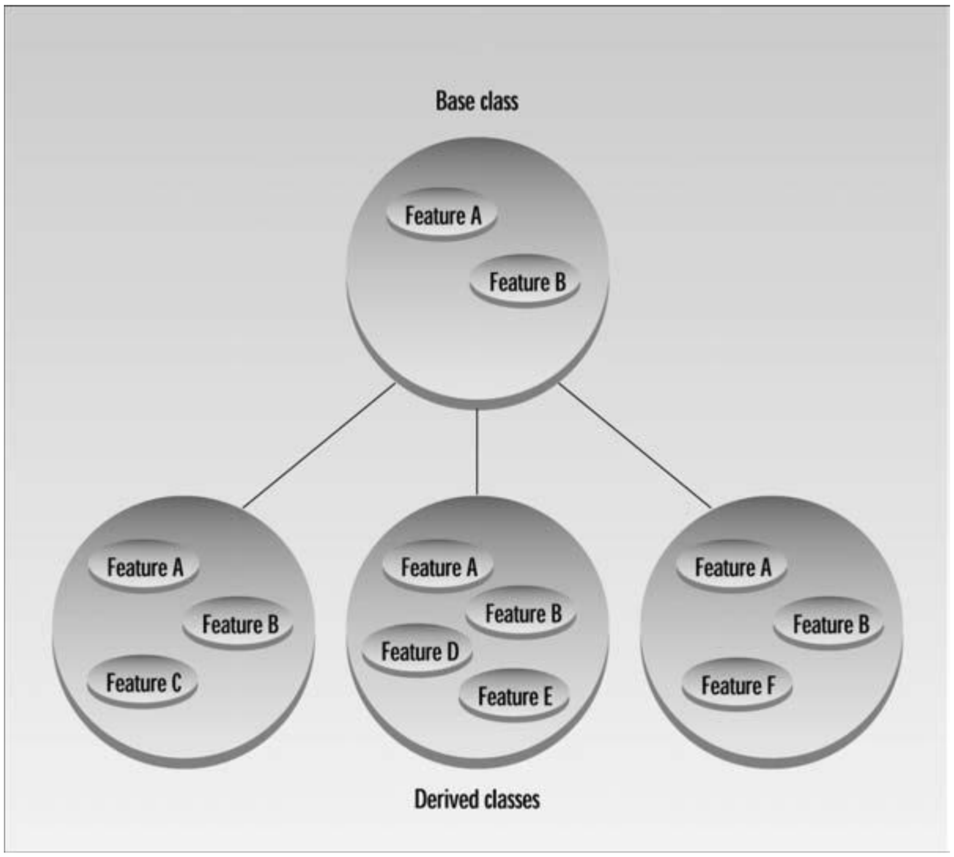

FIGURE 6 *Inheritance.*

Reusability
Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called *reusability*. It is similar to the way a library of functions in a procedural language can be incorporated into different programs.

However, in OOP, the concept of inheritance provides an important extension to the idea of reusability. A programmer can take an existing class and, without modifying it, add additional features and capabilities to it. This is done by deriving a new class from the existing one. The new class will inherit the capabilities of the old one, but is free to add new features of its own.

Creating New Data Types

One of the benefits of objects is that they give the programmer a convenient way to construct new data types. Suppose you work with two-dimensional positions (such as x and y coordinates) in your program. You would like to express operations on these positional values with normal arithmetic operations, such as

```
position1 = position2 + origin
```

where the variables position1, position2, and origin each represent a pair of independent numerical quantities. By creating a class that incorporates these two values, and declaring position1, position2, and origin to be objects of this class, we can, in effect, create a new data type.

Polymorphism and Overloading

Note that the = (equal) and + (plus) operators, used in the position arithmetic shown above, don’t act the same way they do in operations on built-in types such as int.

Using operators or functions in different ways, depending on what they are operating on, is called *polymorphism* (one thing with several distinct forms). When an existing operator, such as + or =, is given the capability to operate on a new data type, it is said to be *overloaded*. Overloading is a kind of polymorphism; it is also an important feature of OOP.