1

Object-Oriented Paradigm

Object-Oriented Programming popularly called *OOPs* is one of the buzzwords in the software industry. On one hand, OOP is a programming paradigm in its own right and on the other, it is a set of software engineering tools which can be used to build more reliable and reusable systems. Another kind of programming methodology which has already revealed its power in the software field, is structured programming. At present, Object-Oriented Programming is emerging from research laboratories and invading the field of industrial applications. The software industry has always been in pursuit of a methodology or philosophy, which would eliminate the problems endemic to software in one shot.The latest candidate for this role is Object Oriented methodology.

Structured programming and object-oriented programming are equally popular today although structured programming has a longer history. The current popularity of OOP and its connection to structured programming is pointed out by Tim Rentsch—*What is object oriented programming ? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip-service to it. Every programmer will practice it (differently). And no one will know just what it is.* Rentsch's predictions still hold true in the 90's.

Structured programming and Object-Oriented Programming fundamentally differ in the following way: Structured programming views the two core elements of any program—*data and functions* as *two separate entities* whereas, OOP views them as a single entity. The benefits of uniting both data and functions into a single unit, will be discussed in later sections.

Object-oriented programming as a paradigm is playing an increasingly significant role in the analysis, design, and implementation of software systems. Object-oriented analysis, design, and programming appear to be the *structured programming* of the 1990's. Proponents assert that OOP is the solution to the *software problem*. Software developed using object-oriented techniques are proclaimed as more reliable, easier to maintain, easier to reuse and enhance, and so on. The Object-Oriented Paradigm is effective in solving many of the outstanding problems in software engineering.

1.1 Why New Programming Paradigms ?

With the continuous decline of hardware cost, high speed computing systems are becoming economically feasible. Innovations in the field of computer architecture supporting complex instructions is in turn leading to the development of better programming environments, which suit the hardware architecture. More powerful tools, operating systems, and programming languages are evolving to keep up with the pace of hardware development. Software for different applications need to be developed under these environments, which is a complex process. As a result, the relative cost of software is increasing substantially when compared to the cost of the hardware of a computing system. Rate of increase in the

cost of software development and maintenance and declining hardware cost over several years is depicted in Figure 1.1. Software maintenance is the process of modifying or extending the capabilities of the existing software. It requires mastery over the understanding and modifying the existing software, and finally revalidating the modified software.

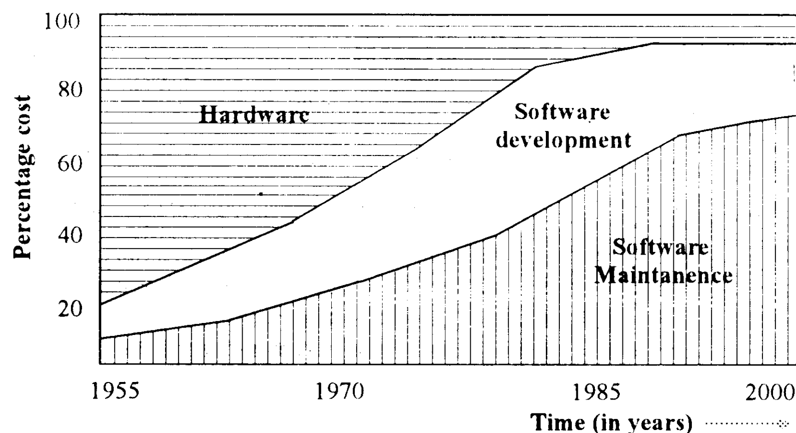


Figure 1.1: System development cost

The cost effectiveness of hardware has been growing by about three orders of magnitude every decade and simultaneously the market for computers is also expanding. This multiplies the number of applications of computers and in turn places greater demands on software. While demand for software has been growing rapidly to keep pace with the growth of hardware, the actual software development has been progressing slowly. Unfortunately, even with all the innovations in the area of languages, programming environments, software engineering concepts, etc., there has been no significant improvement in the productivity of software development, leading to *software crises*. The term "software crises" refers to the overrun of the cost of software development in terms of both budget and time-target.

The software crisis, right from the beginning, is providing an impetus for the development of software engineering principles, tools, and better programming paradigms to build more reliable and reusable systems. The state-of-the-art solution to overcome software crisis is the Object-Oriented Paradigm.

1.2 OOPs ! a New Paradigm

Object-Oriented Programming is a new way of solving problems with computers; instead of trying to mould the problem into something familiar to the computer, the computer is adapted to the problem. Object-Oriented Programming is designed around the data being operated upon as opposed to the operations themselves. Instead of making certain types of data fit to specific and rigid computer operations, these operations are designed to fit to the data. This is as it should be, because the sole purpose of a computer program is to manipulate data.

OOP languages provide the programmer the ability to create class hierarchies, instantiate co-operative objects collectively working on a problem to produce the solution and send messages between objects to process themselves. The power of object-oriented languages is that the programmer can create modular, reusable code and as a result, formulate a program by composition and modification of

the existing modules. Flexibility is gained by being able to change or replace modules without disturbing other parts of the code. Software development speed is gained, on one hand, by reusing and enhancing the existing code and, on the other hand, by having programming objects that are close in representation to the real-world objects, thus reducing the translation burden (from a real-world representation to the computer-world representation) for the programmer.

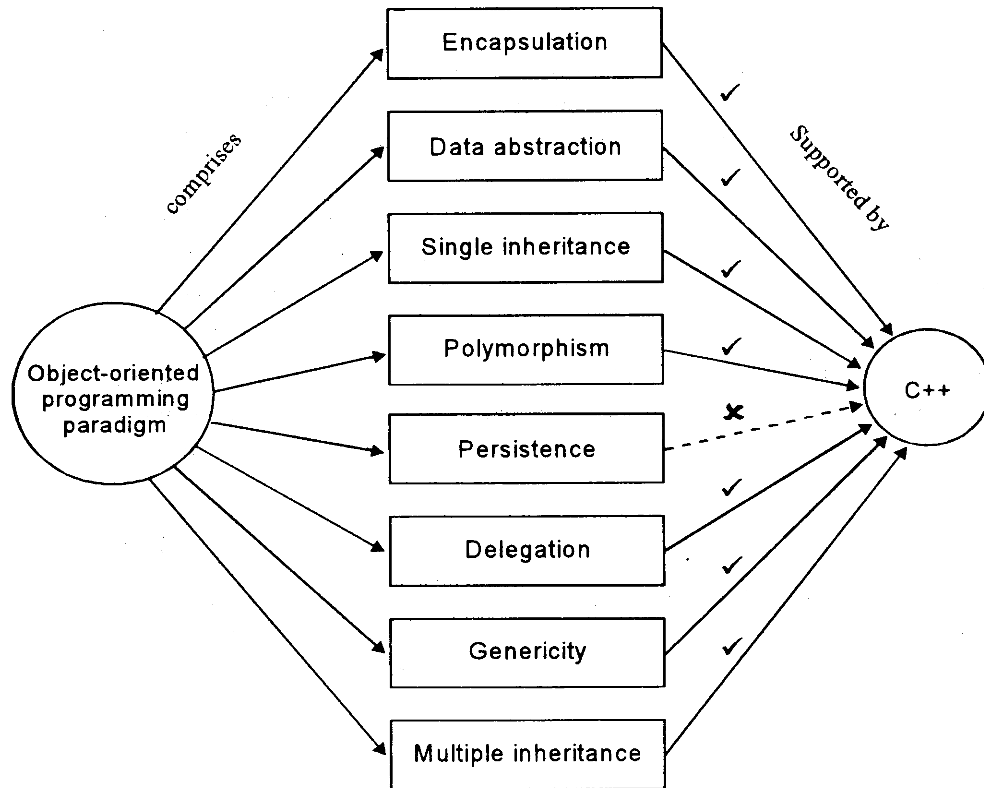


Figure 1.2: Features of object-oriented paradigm

The fundamental features of the OOPs are the following:

- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism
- Message Passing
- Extensibility
- Persistence
- Delegation
- Genericity
- Multiple Inheritance

The important features supported by the object-oriented paradigm are depicted in Figure 1.2. It also shows various features offered by C++ as a language for OOPs paradigm. OOP not only benefits programmers, but also the end-users by providing an object-oriented user interface. It provides a

consistent means of communication between analysts, designers, programmers, and end users. The following terms are most often used in the discussion of OOPs:

Encapsulation: It is a mechanism that associates the code and the data it manipulates into a single unit and keeps them safe from external interference and misuse. In C++, this is supported by a construct called `class`. An *instance* of a class is known as an *object,* which represents a real-world entity.

Data Abstraction: The technique of creating new data types that are well suited to an application to be programmed is known as data abstraction. It provides the ability to create user-defined data types, for modeling a real world object, having the properties of built-in data types and a set of permitted operators. The `class` is a construct in C++ for creating user-defined data types called abstract data types (ADTs).

Inheritance: It allows the extension and reuse of existing code without having to rewrite the code from scratch. Inheritance involves the creation of new classes (derived classes) from the existing ones (base classes), thus enabling the creation of a hierarchy of classes that simulate the class and subclass concept of the real world. The new derived class inherits the members of the base class and also adds its own. Two popular forms of inheritance are single and multiple inheritance. *Single inheritance* refers to deriving a class from a single base class—supported by C++.

Multiple Inheritance: The mechanism by which a class is derived from more than one base class is known as multiple inheritance. Instances of classes with multiple inheritance have instance variables for each of the inherited base classes. C++ supports multiple inheritance.

Polymorphism: It allows a single name/operator to be associated with different operations depending on the type of data passed to it. In C++, it is achieved by function overloading, operator overloading, and dynamic binding (virtual functions).

Message Passing: It is the process of invoking an operation on an object. In response to a message, the corresponding method (function) is executed in the object. It is supported in C++.

Extensibility: It is a feature, which allows the extension of the functionality of the existing software components. In C++, this is achieved through abstract classes and inheritance.

Persistence: The phenomenon where the object (data) outlives the program execution time and exists between executions of a program is known as persistence. All database systems support persistence. In C++, this is not supported. However, the user can build it explicitly using *file streams* in a program.

Delegation: It is an alternative to class inheritance. Delegation is a way of making object composition as powerful as inheritance. In delegation, two objects are involved in handling a request: a receiving object delegates operations to its *delegate*. This is analogous to the child classes sending requests to the parent classes. In C++, delegation is realized by using object composition. Here, new functionality is obtained by assembling or composing objects. This approach takes a view that an object can be a collection of many objects and the relationship is called the *has-a* relationship or *containership.*

Genericity: It is a technique for defining software components that have more than one interpretation depending on the data type of parameters. Thus, it allows the declaration of data items without specifying their exact data type. Such unknown data types (generic data type) are resolved at the time of their usage (function call) based on the data type of parameters. For example, a `sort` function can be parameterized by the type of elements it sorts. To invoke the parameterized `sort()`, just supply the required data type parameters to it and the compiler will take care of issues such as creation of actual function and invoking that transparently. In C++, genericity is realized through *function templates* and *class templates*.

1.3 Evolution of Programming Paradigms

As many software experts point out, *the complexity of software is an essential property, not an accidental one*. This inherent complexity is derived from the following four elements:

- The complexity of the problem domain
- The difficulty of managing the development process
- The flexibility possible through software
- The problems of characterizing the behavior of discrete systems

The sweeping trend in the evolution of high-level programming languages and the shift of focus from programming-in-the-small to programming-in-the-large has simplified the task of the software development team. It also enables them to engineer the illusion of simplicity. This shift in programming paradigm is categorized into the following:

- Monolithic Programming
- Procedural Programming
- Structured Programming
- Object Oriented Programming

Like the computer hardware, programming languages have been passing through evolutionary phases or generations. It is generally observed that most programmers work in one language and use only one programming style. They program in a paradigm enforced by the language they use. Frequently they may not have been exposed to alternate ways of solving the problem and hence, they will have difficulties in exploiting the advantages of choosing a style more appropriate to the problem at hand. Programming style is defined as a way of organizing the ideas on the basis of some conceptual model of programming and using an appropriate language to write efficient programs. Five main kinds of programming styles are listed in Table 1.1 with the different types of abstraction they employ.

Programming Style	Abstraction Employed
Procedure-oriented	Algorithms
Object-oriented	Classes and Objects
Logic-oriented	Goals, often expressed in predicate calculus
Rule-oriented	if-then-else rules
Constraint-oriented	Invariant relationship

Table 1.1: Types of programming paradigms

There is not a single programming style that is best suited for all kinds of applications. For example, procedure-oriented programming would be best suited for the design of computation-intensive problems, rule-oriented programming would be best suited for the design of a knowledge base, and logic-oriented programming would be best suited for a hypothesis derivation. The object-oriented style is best suited for a wide range of applications; indeed, this programming paradigm often serves as the architectural framework in which other paradigms are employed. Each one of these styles of programming require a different mindset and a different way of thinking about the problem, based on their own conceptual framework.

Monolithic Programming

The programs written in these languages exhibit relatively flat physical structure as shown in Figure 1.3. They consist of only global data and sequential code. Program flow control is achieved through the use of jumps and the program code is duplicated each time it is to be used, since there is no support of the subroutine concept and hence, it is suitable for developing small and simple applications. Practically, there is no support for data abstraction and it is difficult to maintain or enhance the program code.

Examples: Assembly language and BASIC

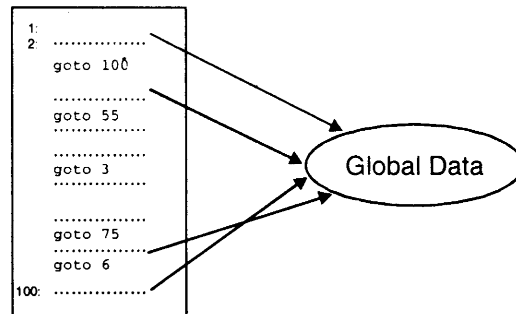


Figure 1.3: Monolithic programming

Procedural Programming

Programs were considered as important intermediate points between the problem and the computer in the mid-1960s. Initially, software abstraction achieved through procedural abstraction grew directly out of this pragmatic view of software. Subprograms were originally seen as labor-saving devices but very quickly appreciated as a way to abstract program functions as shown in Figure 1.4.

The following are the important features of procedural programming:

- Programs are organized in the form of subroutines and all data items are global
- Program controls are through jumps (gotos) and calls to subroutines
- Subroutines are abstracted to avoid repetitions
- Suitable for medium sized software applications
- Difficult to maintain and enhance the program code

Examples: FORTRAN and COBOL

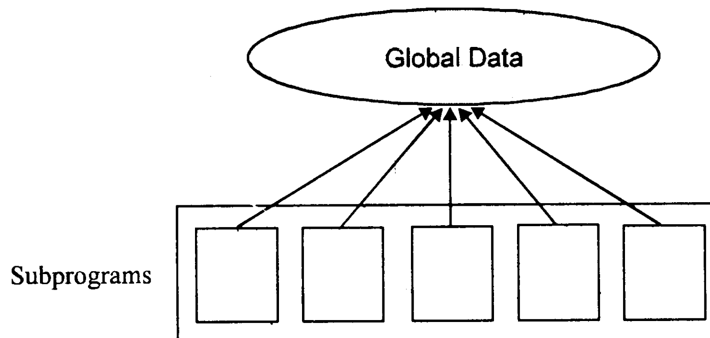


Figure 1.4: Procedural programming

Structured Programming

Structured programming has evolved as a mechanism to address the growing issues of programming-in-the-large. Larger programming projects consist of large development teams, developing different parts of the same project independently. The usage of separately compiled modules (algorithmic decomposition) was the answer for managing large development teams (see Figure 1.5). Programs consist of multiple modules and in turn, each module has a set of functions of related types.

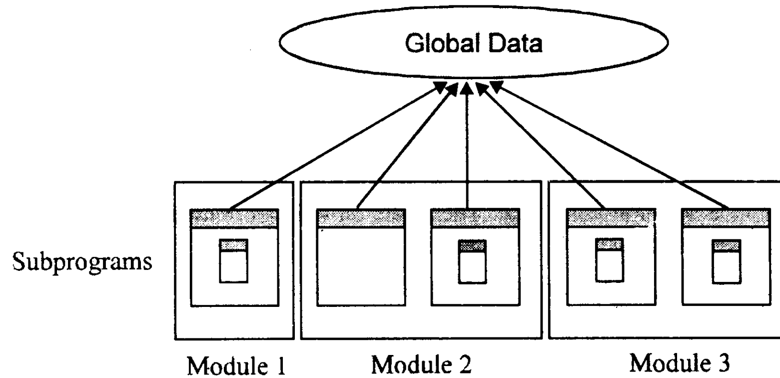


Figure 1.5: Structured programming

The following are the important features of structured programming:

- Emphasis on algorithm rather than data
- Programs are divided into individual procedures that perform discrete tasks
- Procedures are independent of each other as far as possible
- Procedures have their own local data and processing logic
- Parameter passing facility between the procedures for information communication
- Controlled scope of data
- Introduction of the concepts of user defined data types
- Support for modular programming
- Projects can be broken up into modules and programmed independently
- Scope of data items is further controlled across modules
- A rich set of control structures are available to further abstract the procedures
- Co-ordination among multiple programmers is required for handling the changes made to mutually shared data items
- Maintenance of a large software system is tedious and costly

Examples: Pascal and C

Object Oriented Programming

The easy way to master the management of complexity in the development of a software system is through the use of data abstraction. Procedure abstraction is suitable for the description of abstract operations, but it is not suitable for the description of abstract objects. This is a serious drawback in many applications since, the complexity of the data objects to be manipulated contribute substantially to the overall complexity of the problem.

The emergence of data-driven methods provides a disciplined approach to the problems of data abstractions in algorithmic oriented languages. It has resulted in the development of object-based language supporting only data abstraction. Object-based languages do not support features such as inheritance and polymorphism which will be discussed later. Depending on the object features supported, the languages are classified into two categories:

1. Object-Based Programming Languages
2. Object-Oriented Programming Languages

Object-based programming languages support encapsulation and object identity without supporting important features of OOP languages such as polymorphism, inheritance, and message based communication. Ada is one of the typical object-based programming languages.

Object-based language = Encapsulation + Object Identity

Object-oriented languages incorporate all the features of object-based programming languages along with inheritance and polymorphism. Therefore, an object-oriented programming language is defined by the following statement:

Object-oriented language = Object based features + Inheritance + Polymorphism

The topology of object-oriented programming languages is shown in Figure 1.6 for small, moderate, and large projects. The *modules* represent the physical building blocks of these languages; a module is a logical collection of classes and objects, instead of subprograms as in the earlier languages. Thus making classes and objects as the fundamental building blocks of OOPs.

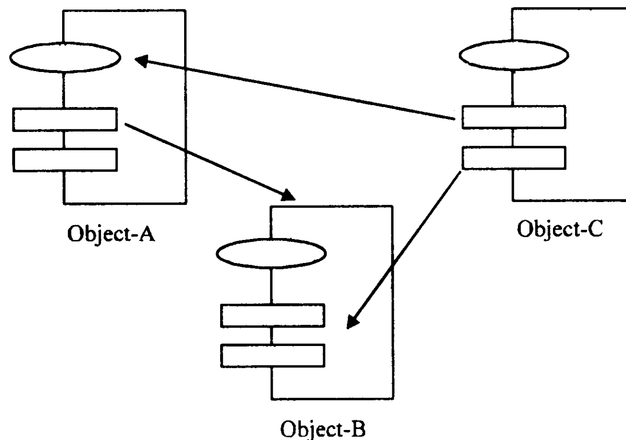


Figure 1.6: Object oriented programming

Object-oriented programming is a methodology that allows the association of data structures with operations similar to the way it is perceived in the human mind. They associate a specific set of actions with a given type of object and actions are based on these associations.

The following are the important features of object-oriented programming:

- Improvement over the structured programming paradigm
- Emphasis on data rather than algorithm
- Data abstraction is introduced in addition to procedural abstraction

- Data and associated operations are unified into a single unit,thus the objects are grouped with common attributes,operations and semantics
- Programs are designed around the data being operated, rather than operations themselves (data decomposition rather than algorithmic decomposition)
- Relationships can be created between similar, yet distinct data types

Examples: C++, Smalltalk, Eiffel, Java, etc.

1.4 Structured Versus Object-Oriented Development

Program and data are two basic elements of any computation. Among these, data plays an important role and it can exist without a program, but a program has no relevance without data. The conventional high level languages stress on the algorithms used to solve a problem. Complex procedures have been simplified by structured programming which is well established to date. Software designers and programmers have faced difficulty in the design, maintenance, and enhancement of software developed using traditional languages, and their search for a better methodology has resulted in the development of the object-oriented approach. In the conventional method, the data are defined as global and accessible to all the functions of a program without any restriction. It has reduced data security and integrity, since the entire data is available to all the functions and any function can change any data without impunity. (See Figure 1.7.)

Unlike the traditional methodology (Function-Oriented Programming -FOP), Object-Oriented Programming emphasizes on the data rather than the algorithm. In OOPs, data is compartmentalized or encapsulated with the associated functions (that operate on it) and this compartment or *capsule* is called an *object*. In the OO approach, the problem is divided into objects, whereas in FOP the problem is divided into functions. Although, both approaches adopt the same philosophy of *divide and conquer*, OOP conquers a bigger region, while FOP is content with conquering a smaller region. OOP contains FOP and so OOP can be referred to as the super set of FOP (like C++, which is a superset of C) and hence, it can be concluded that OOP has an edge over FOP.

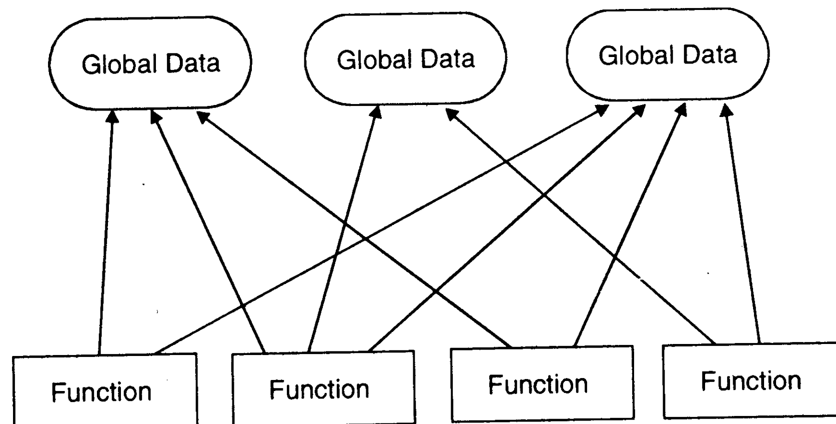


Figure 1.7: Function oriented paradigm

Unlike traditional languages, OO languages allow localization of data and code and restrict other objects from referring to its local region. OOP is centered around the concepts of objects, encapsulations, abstract data types, inheritance, polymorphism, message based communication, etc. An OO language views the data and its associated set of functions as an object and treats this combination as a single entity. Thus, an object is visualized as a combination of data and functions which manipulate them.

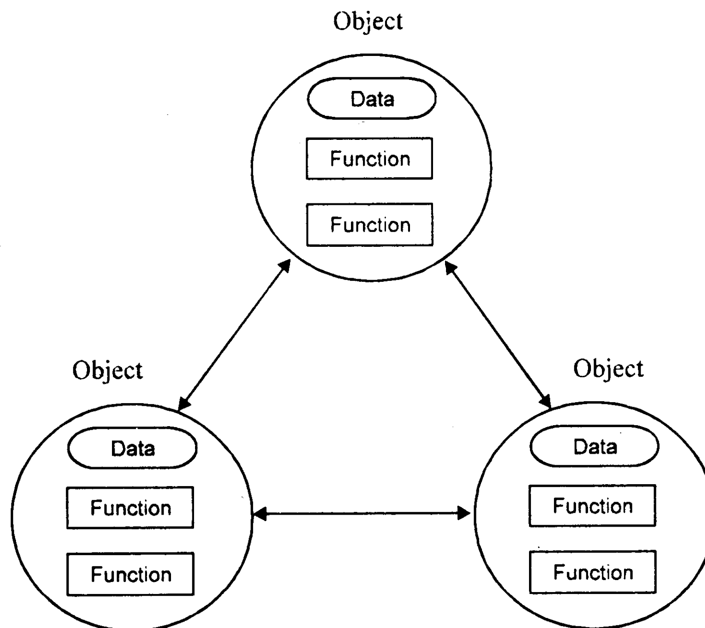


Figure 1.8: Object-oriented paradigm

During the execution of a program, the objects interact with each other by sending messages and receiving responses. For instance, in a program to perform withdrawals from an account, a *customer* object can send a withdrawal message to a *bank account* object. An object communicating with other objects need not be aware of the internal working of the objects with which it interacts. This situation is analogous to operating a television receiver, a computer, or an automobile, where one need not know the internal operations since these machines provide the user with some system controls that hide the complexity of internal structure and working. Likewise, an object can be manipulated through an interface that responds to a few messages. The object's internal structure is totally hidden from the user and this property is called *data/information hiding* or *data encapsulation*.

The external interfaces are implemented by providing a set of methods (functions), each of which accepts and responds to a particular kind of message (see Figure 1.8). The methods defined in an object's class are the same for all objects belonging to that class but, the data is unique for each object

1.5 Elements of Object-Oriented Programming

Object-Oriented Programming is centered around new concepts such as objects, classes, polymorphism, inheritance, etc. It is a well-suited paradigm for the following:

- Modeling the real-world problem as close as possible to the user's perspective.
- Interacting easily with computational environment using familiar metaphors.
- Constructing reusable software components and easily extendable libraries.
- Easily modifying and extending implementations of components without having to recode every thing from scratch.

A language's quality (and its elements) is judged by twelve important criteria. They are a well defined *syntactic and semantic structure*, *reliability*, *fast translation*, *efficient object code*, *orthogonality* (language should have only a few basic features, each of which is separately understandable), *machine independence*, *provability*, *generality*, *consistency with commonly used notations*, *subsets*, *uniformity*, and *extensibility*. The various constructs of OOP languages (such as C++) are designed to achieve these with ease.

Definition of OOP

In the 70s, the concept of the *object* became popular among researchers of programming languages. An object is a combination or collection of data and code designed to emulate a physical or abstract entity. Each object has its own identity and is distinguishable from other objects. *Programming with objects* is as efficient as programming with basic data items such as integers, floats, or arrays. Thus, it provides a direct abstraction of commonly used items and hides most of the complexity of implementation from the users.

Object-Oriented Programming is a programming methodology that associates data structures with a set of operators which act upon it. In OOPs terminology, an instance of such an entity is known as an object. It gives importance to relationships between objects rather than implementation details. Hiding the implementation details within an object results in the user being more concerned with an object's relationship to the rest of the system, than the implementation of the object's behavior. This distinction is a fundamental departure from earlier imperative languages (such as Pascal and C), in which functions and function calls are the centre of activity.

C++ Style of OOP Definition

Grady Booch, a renowned contributor to the development of object-oriented technology defines OOPs as follows: *OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united through the property called inheritance.*

Three important concepts to be noted in the above definition are: *objects, classes, and inheritance*. OOP uses objects and not algorithms as its fundamental building blocks. Each object is an instance of some class. *Classes* allow the mechanism of data abstraction for creating *new* data types. Inheritance allows building of new classes from the existing classes. Hence, if any of these elements are missing in a program, then, it is not object-oriented. In particular, a program *without inheritance* is definitely not an object oriented one; it resembles the program with abstract data types.

1.6 Objects

Initially, different parts (entities) of a problem are examined independently. These entities are chosen because they have some physical or conceptual boundaries that separate them from the rest of the problem. The entities are then represented as objects in the program. The goal is to have a clear correspondence between physical entities in the problem domain and objects in the program. A well designed object oriented program is organized according to the objects being manipulated.

Figure 1.9 shows few entities and each of them can be treated as an object. In other words, an object can be a person, a place, or a thing with which the computer must deal. Some objects may correspond to real-world entities such as students, employees, bank accounts, inventory items, etc., whereas, others may correspond to computer hardware and software components. Hardware components include a keyboard, port, video display, mouse, etc., and software components include stacks, queues, trees, etc. In an application simulating a parking lot, car, parking spaces, traffic signals, or even the persons manning the parking lot can be conceptualized as objects. Objects can be concrete such as a file system, or conceptual such as a scheduling policy in a multiprocessor operating system. Objects mainly serve the following purposes:

- Understanding of the real world and a practical base for designers.
- Decomposition of a problem into objects depends on judgement and nature of the problem.

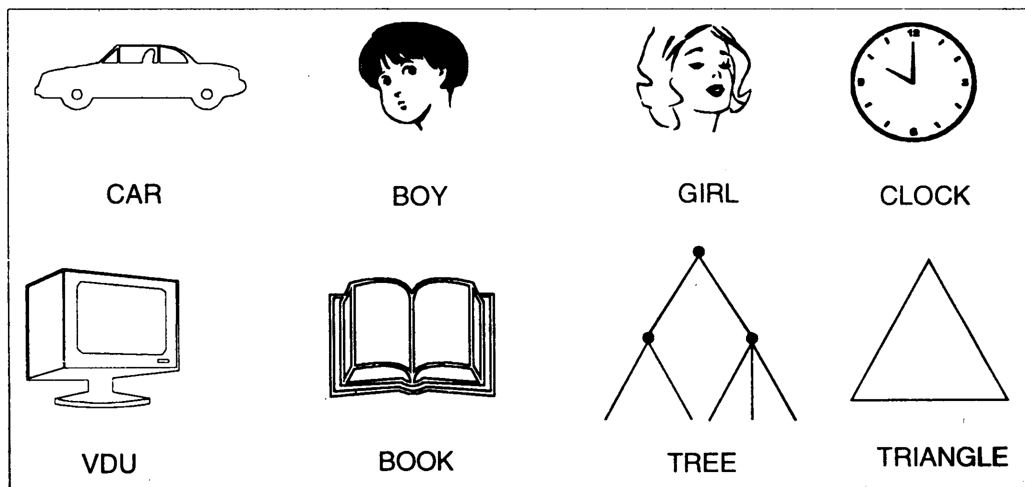


Figure 1.9: Examples of objects

Every object will have data structures called attributes and behavior called operations. The different notations of an object uniting both the data and operations, are shown in Figure 1.10.

Consider the object *account* having the attributes: *AccountNumber, AccountType, Name,* and *Balance* and operations:*Deposit, Withdraw,* and *Enquire*. Its pictorial notation is shown in Figure 1.11. Each object will have its own identity though its attributes and operations are same; the objects will never become equal. In case of *person* object for instance, two persons have the same attributes like *name, age,* and *sex,* but they are not equal (technically). Objects are the basic run-time entities in an object-oriented system.

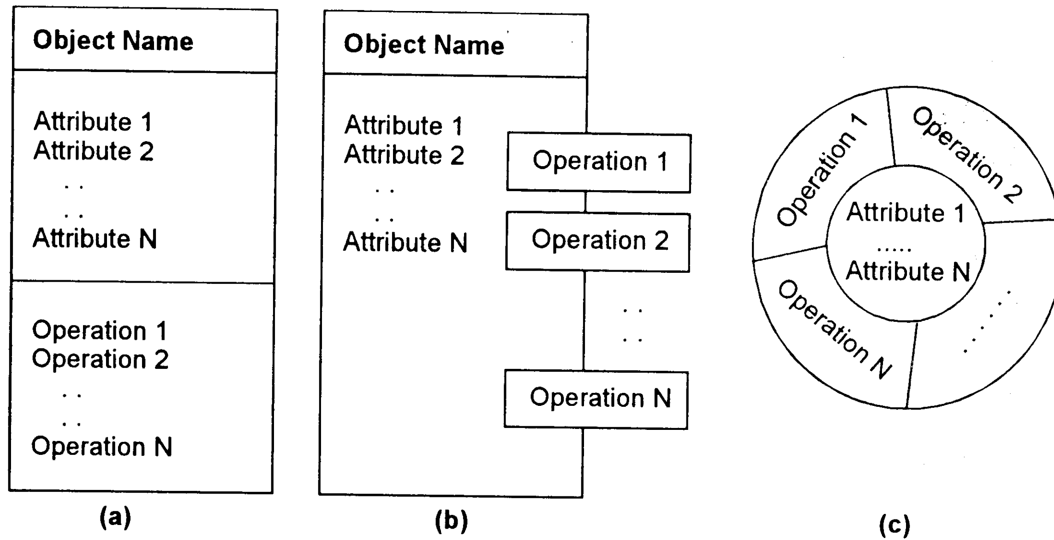


Figure 1.10: Different styles of representing an object

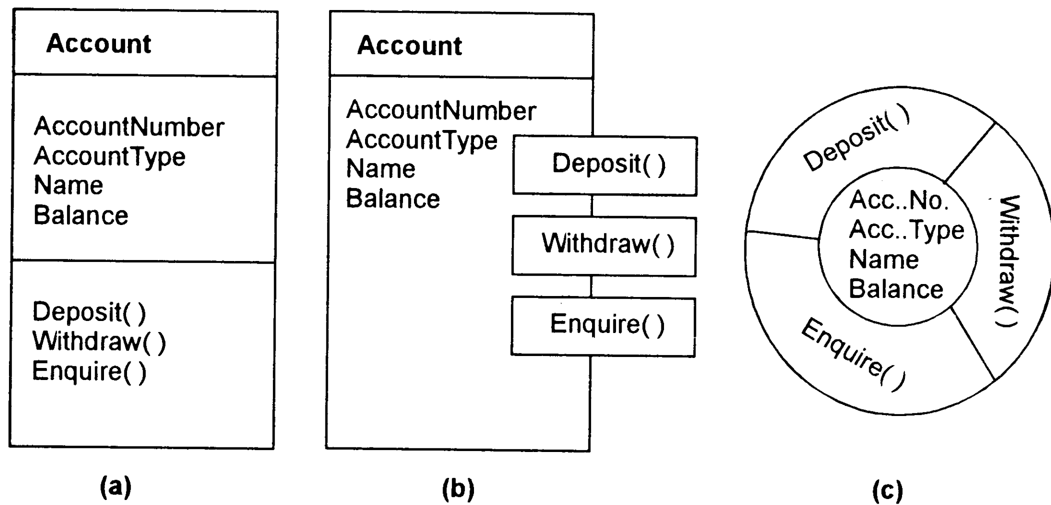


Figure 1.11: Different styles of representing the account object

1.7 Classes

The objects with the same data structure (attributes) and behavior (operations) are grouped into a *class*. All those objects possessing similar properties are grouped into the same unit. The concept of *class*-ing the real world objects is demonstrated in Figure 1.12. It consists of the *Person* class, *Vehicle* class, and *Polygon* class. In the case of *Person* class, all objects have similar attributes like *Name*, *Age*, *Sex* and similar operations like *Speak*, *Listen*, *Walk*. So *boy* and *girl* objects are grouped into the *Person* class. Similarly, other related objects such as triangle, hexagon, and so on, are grouped into the *Polygon* class.

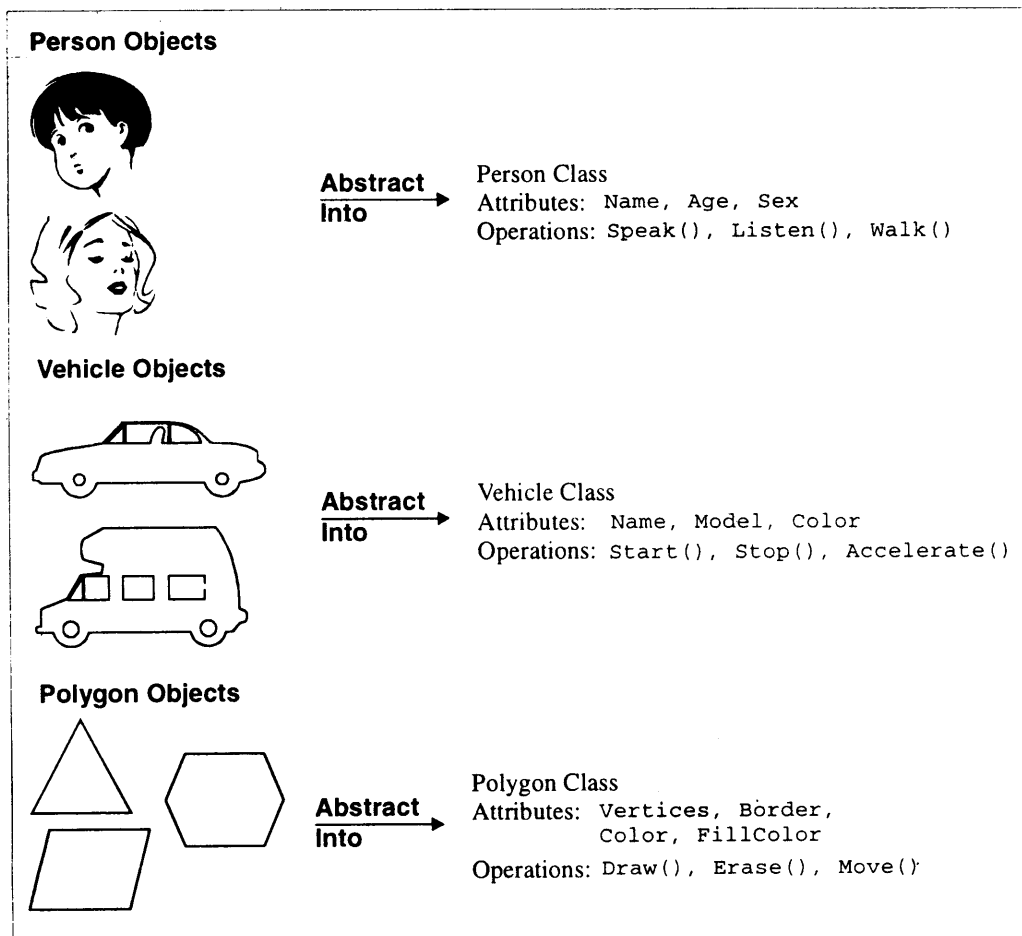


Figure 1.12: Objects and classes

Every *object* is associated with *data* and *functions* which define meaningful operations on that object. For instance, in C++, related objects exhibiting the same behavior are grouped and represented by a *class* in the following way:

```
class account
{
   private:
      char Name[20];          // data members
      int AccountType;
      int AccountNumber;
      float Balance;
   public:
      Deposit();              // member functions
      Withdraw();
      Enquire();
};
```

This declaration is similar to the structure declaration in C. It enables the creation of the *class* variables called *objects*. For example, the following statements,

```
account savings_account;
account current_account;
account FD_account;
```

create instances of the class `account`. They define `savings_account`, `current_account`, and `FD_account` as the objects of the class `account`. From this, it can be inferred that, the `account` class groups objects such as saving account, current account, etc. Thus, objects having the same structural and behavioral properties are grouped together to form a class.

Each class describes a possibly infinite set of individual objects; each object is said to be an instance of its class and each instance of the class has its own value for each attribute but shares the attribute name and operations with other instances of the class. The following points on classes can be noted:

- A class is a template that unites data and operations.
- A class is an abstraction of the real world entities with similar properties.
- A class identifies a set of similar objects.
- Ideally, the class is an implementation of abstract data type.

1.8 Multiple Views of the Same Object

A commonly accepted notion about objects is illustrated through the definition of a tree. In this classical model, a tree is defined as a class, in terms of internal state information and methods that can be applied. The designer of such an object-oriented tree, ideally works with the intrinsic properties and behavior of the tree. In the real world, properties of a tree like its height, cell-count, density, leaf-mass, etc., are intrinsic properties. Intrinsic behavior includes like growth, photosynthesis, etc., that affect the intrinsic properties. This idea of a classical model is inadequate to deal with the construction of large and growing suites of applications that manipulate the objects. Every observer (for instance, a tax-assessor, a woods man and a bird) of the tree, with different backgrounds, has his own view on the ideal model of a tree as shown in Figure 1.13.

A *tax-assessor* has his own view of the features and behavior associated with a tree. The characteristics include its contribution to the assessed value of the property on which it grows. The behavior includes the methods, by which this contribution is derived. These methods vary from *tree-type* to *tree-type*. In fact, such methods may form a part of a tax assessor's view of all objects, tree and non-tree alike. These characteristics and behaviors are extrinsic to the tree. They form the part of a tax-assessor's subjective view of the object-oriented tree.

Figure 1.13 reminds that the *tax-assessor* is merely one of a suite (type) of applications, each having its own subjective view, its own extrinsic state and behavior for the tree. The views of a woodsman and a bird on the same object,are also different compared to the tax-assessor's view. A *woodsman's* view of the tree, is in terms of sales price and time required to cut the tree with capital profit as a method. A *bird's* view of the same tree is different and its view characteristics include `FoodValue` and `ComputeFlight()`. Thus, a woodsman views the tree in terms of the amount of time required to cut the tree and the price it would fetch. The bird views it in terms of the food value and the amount of energy required to carry the food from the tree to its nest.

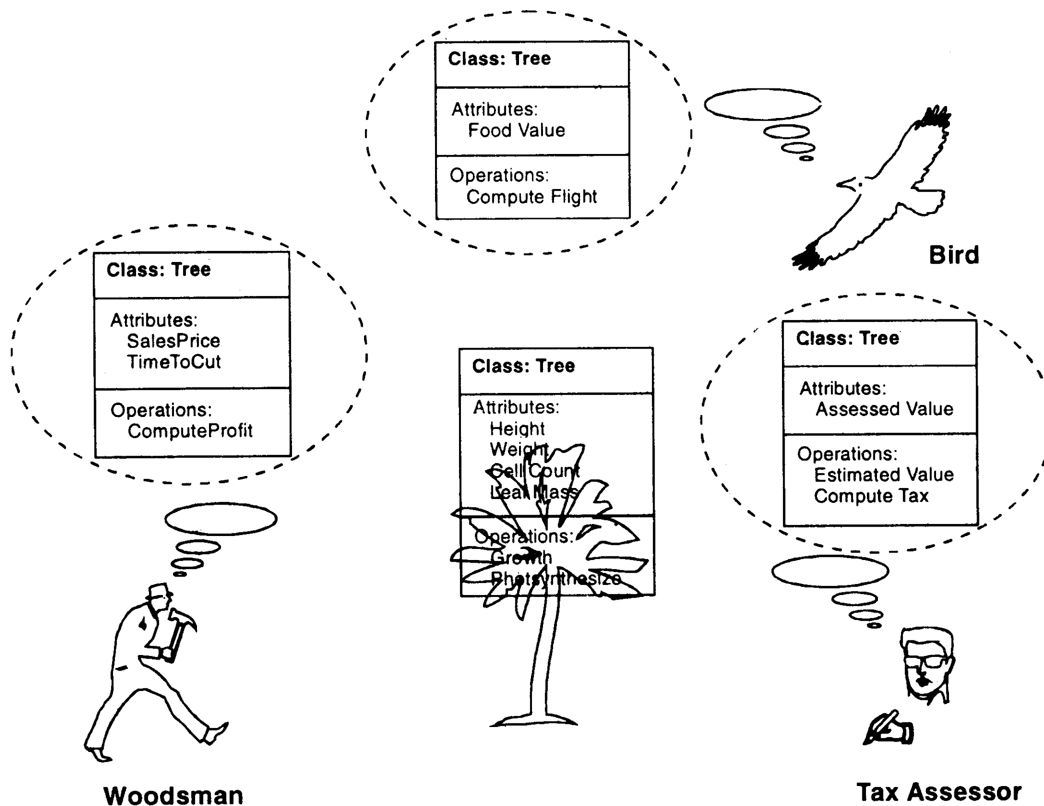


Figure 1.13: Multiple views of an object-oriented tree

1.9 Encapsulation and Data Abstraction

Encapsulation is a mechanism that associates the code and the data it manipulates and keeps them safe from external interference and misuse. Creating new data types using encapsulated-items, that are well suited to an application to be programmed, is known as *data abstraction*. The data types created by the data abstraction process are known as Abstract Data Types (ADTs). Data abstraction is a powerful technique, and its proper usage will result in optimal, more readable, and flexible programs.

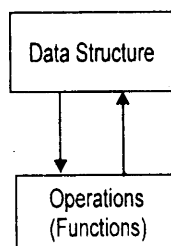


Figure 1.14: An abstract data type

Data abstraction is supported by several other modern programming languages such as Smalltalk, Ada, etc. In these languages, and in C++ as well, a programmer can define a new abstract data type by specifying a data structure, together with the operations permissible on that data structure as shown in Figure 1.14. The important feature of C++, the *class* declaration, allows encapsulation and creation of abstract data types.

The use of encapsulation in protecting the members (data and code) of a class from unauthorized access is a good programming practice; it enforces the separation between the specification and implementation of abstract data types, and it enables the debugging of programs easily.

1.10 Inheritance

Inheritance is the process, by which one object can acquire the properties of another. It allows the declaration and implementation of one class to be based on an existing class. Inheritance is the most promising concept of OOP, which helps realize the goal of constructing software systems from reusable parts, rather than hand coding every system from scratch. Inheritance not only supports reuse across systems, but also directly facilitates extensibility within a given system. Inheritance coupled with polymorphism and dynamic binding, minimizes the amount of existing code to be modified while enhancing a system.

To understand inheritance, consider the simple example shown in Figure 1.15. When the class *Child,* inherits the class *Parent*, the class *Child* is referred to as derived class (sub-class), and the class *Parent* as a base class (super-class). In this case, the class *Child* has two parts: a derived part and an incremental part. The derived part is inherited from the class *Parent.* The incremental part is the new code written specifically for the class *Child.* In general, a feature of *Parent* may be renamed, re-implemented, duplicated, voided (nullified), have its visibility status changed or subjected to almost any other kind of transformation when it is mapped from *Parent* to *Child.* The inheritance relation is often called the *is-a* relation. This is because when the class *Child* inherits the base class *Parent*, it acquires all the properties of the *Parent* class. It can also have its own properties, in addition to those acquired from its *Parent*. This is an example of single inheritance; the child class has inherited properties from only one base class.

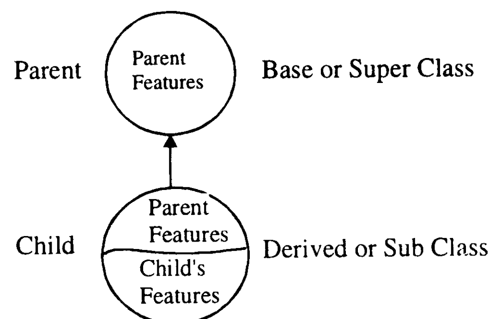


Figure 1.15: Single inheritance

The inheritance relation is often used to reflect the elements present in an application domain. For example, consider a rectangle which is a special kind of polygon as shown in Figure 1.16. This relationship is easily captured by the inheritance relation. When the *rectangle* is inherited from the *polygon*, it

gets all the features of the *polygon*. Further, the *polygon* is a *closed* figure and so, the *rectangle* inherits all the features of the *closed* figure.

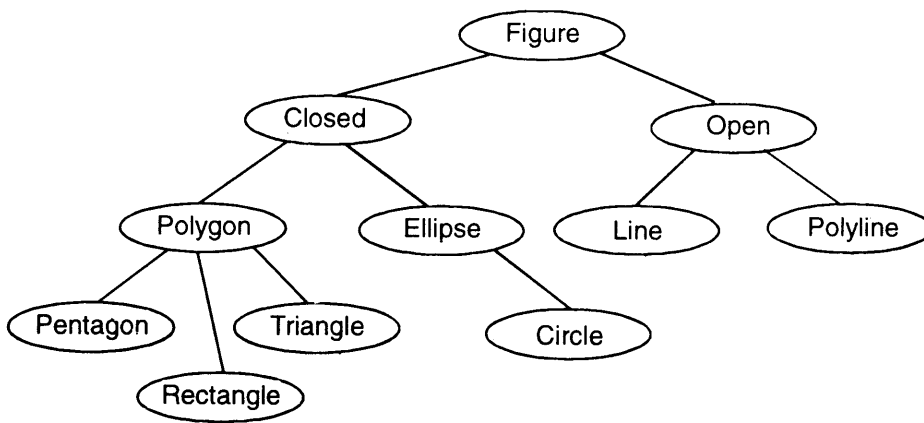


Figure 1.16: Inheritance graph (class hierarchy)

Multiple Inheritance

In the case of multiple inheritance, the derived class inherits the features of more than one base class. Consider Figure 1.17, in which the class *Child* is inherited from the base classes *Parent1* and *Parent2*. Here, the class *Child* possesses all the properties of parents classes in addition to its own.

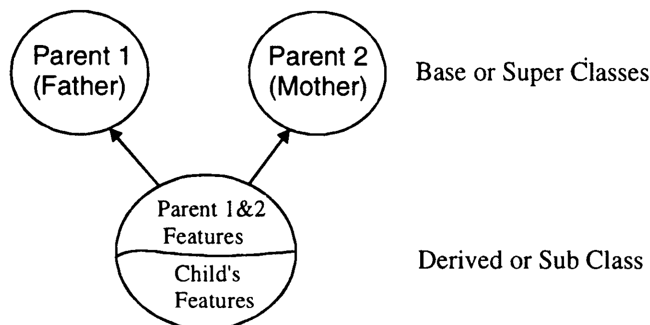


Figure 1.17: Multiple inheritance

Benefits of Inheritance

There are numerous benefits that can be derived from the proper use of inheritance, which include the following:

- The inherited code that provides the required functionalities, does not have to be rewritten. Benefits of such reusable code include, increased reliability and decreased maintenance cost because of sharing by all the users.
- Code sharing can occur at several levels. For example, at a higher level, individual or group users can use the same classes. These are referred to as software components. At a lower level, code can be shared by two or more classes within a project.

- Inheritance will permit the construction of reusable software components. Already, several such libraries are commercially available and many more are expected to come.
- When a software system can be constructed largely out of reusable components, development time can be concentrated for understanding that portion of the system which is new and unusual. Thus, software systems can be generated more quickly, and easily, by rapid prototyping.

All the above benefits of inheritance emphasize code reuse, ease of code maintenance, extension, and reduction in development time.

1.11 Delegation - Object Composition

Most people can understand concepts such as objects, interfaces, classes, and inheritance. The challenge lies in applying them to build flexible and reusable software. The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition. As explained, inheritance is a mechanism of building a new class by deriving certain properties from other classes. In inheritance, if the class D is derived from the class B, it is said that *D is a kind of B*. The new approach to object composition, takes a view that an object can be a collection of many other objects, and the relationship is called a *has-a* (D has-a B) relationship or containership.

Delegation is a way of making object composition as powerful as inheritance for reuse. In delegation, two objects are involved in handling a request: a receiving object delegates operations to its *delegate*. This is analogous to subclasses sending requests to parent classes. In certain situations, inheritance and containership relationships can serve the same purpose. For example, instead of creating a class `Window` as a derived class of `Rectangle` (because, the window happens to be rectangular), the class `Window` can reuse the behavior of Rectangle by having a `Rectangle` instance variable and delegating the `Rectangle` specific behavior to it. In other words, instead of the class Window being a Rectangle, it would have a Rectangle composed into it. Window must now forward all requests to its Rectangle instance explicitly. In inheritance, it would have inherited the same operation from the class Rectangle. The Window class delegating its Area operation to a Rectangle instance is depicted in Figure 1.18.

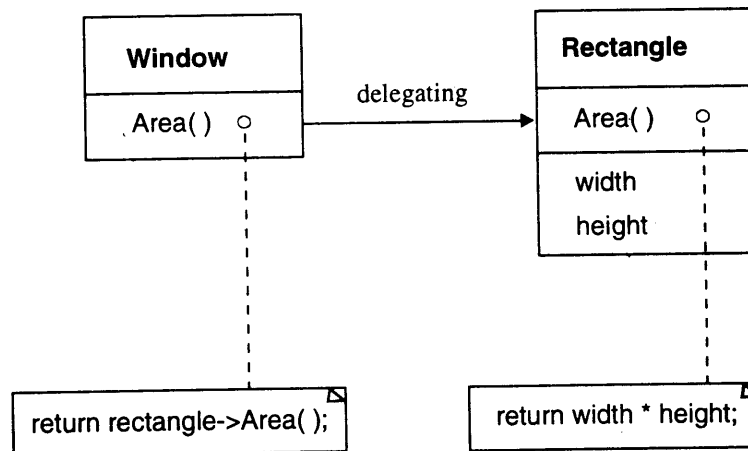


Figure 1.18: Delegation-object composition

Delegation makes it easy to compose behavior at runtime and to change the manner, they are composed. The window can become circular at runtime, simply by replacing its Rectangle instance with a Circle instance, assuming Rectangle and Circle have the same type. Thus, delegation shows that inheritance can be replaced with object composition as a mechanism for code reuse.

1.12 Polymorphism

In the real world, the meaning of an operation varies with context and the same operation may behave differently, in different situations. The *move* operation, for example, behaves differently on the class *person,* and on the class *polygon* on the screen. A specific implementation of an operation by a certain class is called a *method.* An object oriented operation, being polymorphic, may have more than one method of implementing it. The word *polymorphism* is derived from the Greek meaning *many forms.* It allows a single name to be used for more than one related purpose, which are technically different. The following are the different ways of achieving polymorphism in a C++ program:

- Function Name Overloading
- Operator Overloading
- Dynamic Binding

Polymorphism permits the programmer to generate high level reusable components that can be tailored to fit different applications, by changing their low level parts.

Dynamic Binding

Binding refers to the tie-up of a procedure call to the address code to be executed in response to the call. Dynamic binding (also called *late binding*) means that the code associated with a given procedure call is not known until its call at run-time. For example, consider a graphics application (see Figure 1.17), in which the class *Figure,* contains a procedure `draw()`. By inheritance, every graphics primitive in this diagram has a procedure `draw()`. The `draw()` algorithm is, however, unique to each graphical shape, and so the `draw()` procedure will be redefined in each class that defines a graphic primitive. To redraw the entire graphics window, the following code will suffice:

```
for i = 1 to number_of_shapes do
  ptr_to_figure[i]->draw();
```

At each pass through the loop, the code matching the dynamic type of `ptr_to_figure[i]` will be called. Even if additional kinds of shapes are added to the system, this code segment will still remain unchanged. This is, in contrast to the traditional *case/switch* statement design of a program.

Another example could be that of an operation `print` in a class `File`. Different methods could be implemented to print ASCII files, binary files, digitized picture files, etc. All these methods logically perform the same task - printing a file; thus the corresponding generic operation is `print`. However, the individual methods may each be implemented by a different code.

1.13 Message Communication

In conventional programming languages, *a function is invoked on a piece of data (function-driven communication),* whereas in an object-oriented language, *a message is sent to an object (message-driven communication).* Hence, conventional programming is based on functional abstraction whereas, object oriented programming is based on data abstraction. This is illustrated by a simple example of evaluating the square root of a number. In conventional functional programming, the function `sqrt(x)`

for different data types (x's type), will be defined with different names, which takes a number as an input and returns its square root. For each data type of x, there will be a different version of the function *sqrt*. In contrast, in an OOPL (Object-Oriented Programming language), the expression for evaluating the square root of x takes the form `x.sqrt()`, implying that the object x has sent a message to perform the square root operation on itself. Different data types of x, invoke a different function code for `sqrt`, but the expression (code) for evaluating the square root will remain the same. By its very nature, OO (Object-Oriented) computation resembles the client-server computing model.

In object-oriented programming, the process of programming involves the following steps:

- Create classes for defining objects and their behaviors.
- Define the required objects.
- Establish communication among objects through message passing.

Communication among the objects occur in the same way as people exchange messages among themselves. The concept of programming, with the message passing model, is an easy way of modeling real-world problems on computers. A message for an object is interpreted as a request for the execution of a function. A suitable function is invoked soon after receiving the message and the desired results are generated within an object. A message comprises the name of the object, name of the function and the information to be sent to the object as shown in Figure 1.19.

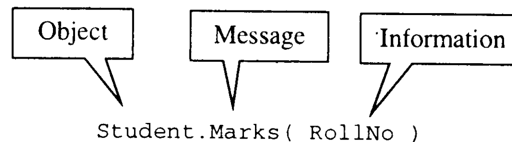


Figure 1.19: Object-oriented message communication

Like in the real world, *objects* also have a life cycle! They can be created and destroyed automatically, whenever necessary. *Communication between the objects can take place as long as they are alive*! In Figure 1.19, `Student` is treated as an object sending the message `Marks` to find the marks secured by the student with the specified `RollNo`. In this case, a function call `Marks()` is treated as a *message* and a parameter `RollNo` is treated as *information* passed to the object.

In OOPs, the correct method to execute an operation based on the name of the operation and the class of the object being operated, is automatically selected depending on the type of message received. The user of an operation need not be aware of the alternative methods available to implement a given polymorphic operation. New classes can be added without changing the existing code, but methods have to be provided for each applicable operation on the new class.

1.14 Popular OOP Languages

Every programming methodology emphasizes on some new concepts in programming. In OO programming, the attention is focused on objects. In this, data do not flow around a system; it is the messages that move around the system. By sending messages, the clients (user/application program) request objects to perform operations. The kinds of services the objects can provide are known to the clients. This, basically, represents the client-server model, where the client calls on a server, which performs some service and sends the result back to the client. The client must know the interface of the server, but the server need not know the interfaces of the clients, because all the interactions are initiated by clients using the server's interface.

Feature	C++	Smalltalk 80 *	Objective C	Simula *	Ada **	Charm ++	Eiffel *	Java
Encapsulation (Data hiding)	√	Poor	√	√	√	√	√	√
Single inheritance	√	√	√	√	✗	√	√	√
Multiple inheritance	√	✗	√	✗	✗	√	√	✗
Polymorphism	√	√	√	√	√	√	√	√
Binding (early or late)	Both	Late	Both	Both	Early	Both	Early	Late
Concurrency	Poor	Poor	Poor	√	Difficult	√	Promised	√
Garbage collection	✗	√	√	√	✗	✗	√	√
Persistent objects	✗	Promised	✗	✗	Like 3GL	✗	Limited	✗
Genericity	√	✗	✗	✗	√	√	√	✗
Class libraries	√	√	√	√	Limited	√	√	√

* Pure object-oriented languages
** Object-based languages
Others are extended conventional languages

Table 1.2: Comparing object-oriented language features

Every OO language implements the basic OO concepts in a different way. They vary in their support of some of the advanced OO concepts such as multiple inheritance, class library, memory management, templates, exceptions, etc. Some of the popular OO languages namely C++, Smalltalk, Eiffel and CLOS are discussed. The *genealogy* of different languages is shown in Table 1.2 indicating various features supported by them.

One great divide in programming exists between *exploratory programming* languages that aim at dynamism and run-time flexibility, and *software engineering* languages which have static typing and other features that aid verifiability and/or efficiency. While both languages have their applications, the latter group to which C++ belongs, is of interest for further discussion. Smalltalk is the best-known representative of the former group.

C++

Bjarne Stroustrup developed C++ at AT & T Bell laboratories as an extension of C in the year 1980. (in fact, C was also invented at the same place by Dennis Ritchie in the early 1970's). C++ was first installed outside the designer's research group in July, 1983; however, quite a few current C++ features had not been invented. Suggested advantages of C++ are the "*...previous C users can quite well upgrade*

gradually to programming in C++, in the first step just feeding their existing C code through the C++ translator and checking if some small modifications would be necessary". However, some consider this as a disadvantage. They claim that an abrupt change of paradigm is necessary to make programmers think in an object-oriented fashion.

C++ is evolved from a dialect of C known as *C with Classes* as a language for writing effective event-driven simulations. Several key ideas were borrowed from the Simula67 and ALGOL68 programming languages. The heritage of C++ is shown in Figure 1.20. Earlier version of the language, collectively known as "C with Classes" has been in use since 1980. It lacked operator overloading, references, virtual functions, and all these are overcome in C++. The name C++ (pronounced as C plus plus) was coined by Rick Mascitti in the summer of 1983. The name signifies the evolutionary nature of the changes from C. "++" is the C increment operator. The slightly short name C+ is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language is not called D, because it is an extension of C, and does not attempt to remedy the problems by removing features.

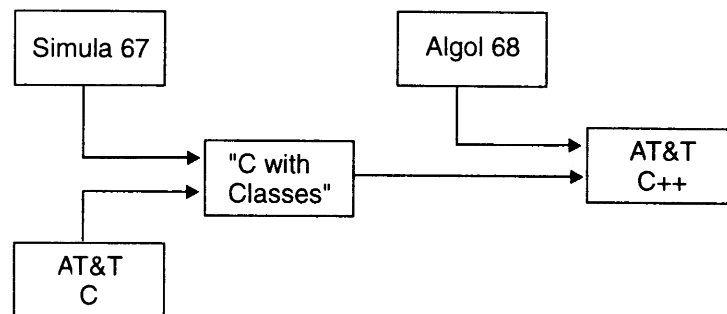


Figure 1.20: Heritage of C++

The C++ language corrects most of the deficiencies of C by offering improved compile-time type checking and support for modular and object-oriented programming. Some of the most prominent features of C++ are *classes, operator and function overloading, free store management, constant types, references, inline functions, inheritance, virtual functions, streams for console and file manipulation, templates, and exception handling.*

In C++, both attributes (data) and methods (functions) are members of a class. The members must be declared either as *private* or *public*. Public members can be accessed by any function; private members can only be accessed by methods of the same class. C++ has a special *constructor function* to initialize new instances and a *destructor function* to perform necessary cleanup when an object is to be destroyed. C++ provides three kinds of memory allocation for objects: *static* (preallocated by the compiler in fixed global memory); *automatic* (allocated on the stack) and *dynamic* (allocated from a heap). Static storage is obtained by defining a variable outside any function using the `static` keyword . Local variables within functions normally use automatic storage. Dynamic storage is allocated from a heap on an explicit request from the programmer and it must be explicitly released since, standard implementations of C++ do not have a garbage collector.

The superclass of a class is specified as a part of class declarations. A superclass is known as *base class* and a subclass is known as *derived class*. Attributes once declared in the superclass, which are inherited by its subclasses, need not be repeated. They can be accessed from any subclass unless they

are declared *private*. Only the methods of a class can access its private attributes: Attributes declared *protected*, are accessible to subclasses, but not to a direct client object like private members. Methods declared in a superclass are also inherited. If a method can be overridden by the subclass, then it must be declared virtual in its first appearance in a superclass. Thus, the need to override the method must be anticipated and written into the base class itself. C++ does not support the concept of dynamic binding in a thorough sense and hence it is (some times) considered as a poor OOP language.

Smalltalk

Smalltalk is the first popular OO language developed at Xerox's Palo Alto Research Center (PARC). Apart from being a language, it has a development environment. Smalltalk programs are normally entered using the Smalltalk browser. Objects are called instance variables. All Smalltalk objects are dynamic, and are allocated from a heap. Smalltalk offers fully automatic garbage collection and deallocation is performed by a built-in garbage collector. All variables are untyped and can hold objects of any class. New objects are created using the same message passing mechanism used for operations on objects. All attributes are private to the class. There is no way to restrict the operations of a class. All operations are public.

Inheritance is achieved by supplying the name of the superclass. All attributes of the superclass are available to all its descendants. All methods can be overridden. The standard implementation of Smalltalk does not support multiple inheritance. Smalltalk is weakly typed, so errors are more likely to appear at runtime. It provides a highly interactive environment, which permits rapid development of programs. It has a rich class library designed to be extended and adapted by adding subclasses to meet the needs of a specific application.

Charm ++

Charm++ is a portable, concurrent, object-oriented system based on C++. It is an extension of C++ and provides a clear separation between sequential and parallel objects. The execution model of Charm++ is message driven, which helps the programmer to write programs that are latency-tolerant. The language supports multiple inheritance, dynamic binding, overloading, strong typing, and reuse of parallel objects. Charm++ provides specific modes for sharing information between parallel objects. It is based on the Charm parallel processing system and its runtime system implementation reuses most of the runtime system of Charm. Extensive dynamic load balancing strategies are provided. Charm++ has been implemented to run on different parallel systems, including shared memory machines (e.g., Sequent Symmetry), non-shared machines (e.g., nCUBE/2), uniprocessor, and network of workstations.

Java

The Java programming language is the result of several years of research and development at SUN (Stanford University Net) Microsystems, Inc., USA. SUN defines Java as follows: Java is a *new, simple, object-oriented, distributed, portable, architecture neutral, robust, secure, multi-threaded, interpreted,* and *high-performance programming language*. Java is mainly intended for the development of object-oriented network based software for *Internet* applications. Its syntax is similar to C and C++, but it omits semantic features that make C and C++ complex, confusing, and insecure. It does not support some of the more difficult to use features of C++ such as pointers. It also features built-in safety mechanisms (like absence of pointers) which provide some level of security on network. Hence, Java as a logical successor to C++ can also be called as C++--++ (C-plus-plus-minus-minus-plus-plus i.e., remove some difficult to use features of C++ and add some good features).

Java is the first language to provide a comprehensive, robust, platform-independent solution to the

challenges of programming for the Internet and other complex networks. Java features portability, security and advanced networking without compromising on performance. Sun Microsystems' traditional family of SPARC processors, as well as processors of other architectures, will run Java software. By optimizing the new Java processor family for Java-only applications, an unprecedented level of price versus performance will be reached. Java was initially designed to address the problems of building software for small distributed systems to embed in consumer devices. As such it is designed for heterogeneous networks, multiple host architectures, and secure delivery. To meet these requirements, compiled Java code had to survive transport across networks, operate on any client, and assure the client that it is safe to run.

Java's future is promising. It is robust, object-oriented, and portable (source and byte code-executable) i.e., Java's application byte code runs on any platform without any modification or re-compilation; Java byte codes are interpreted by Java Virtual Machine (JVM) running on a local machine. Java integrates the flexibility of interpreted languages and power of compiler languages. Java comes bundled with a suite of classes for GUI (Graphical User Interface), multithreading, networking, file I/O, and the like. To add to this, APIs (Application Program Interface) for database access (Java Database Connectivity), more robust multimedia processing, and remote object access are in the development.

1.15 Merits and Demerits of OO Methodology

OOP systems are sold on the promise of improved productivity through object reuse and high level of code modularity. These aspects precisely lead to their greatest benefit, namely improved software quality, considering the objective of OO design is to *mirror the real world objects* in the software systems. OO languages have many advantages over traditional procedure-oriented languages.

Advantages

We perceive the world around us as being made up of objects and the brain arranges this information into chunks (groups). OO design uses objects in a programming language, which aids in trapping an existing pattern of human thought into programming.

Since the objects are autonomous entities and share their responsibilities only by executing methods relevant to the received messages, each object lends itself to greater modularity. Cooperation among different objects to achieve the system operation is done through exchange of messages. The independence of each object eases development and maintenance of the program.

Information hiding and data abstraction increase reliability and help decouple the procedural and representational specification from its implementation. Dynamic binding increases flexibility by permitting the addition of a new class of objects without having to modify the existing code. Inheritance coupled with dynamic binding enhances the reusability of a code, thus increasing the productivity of a programmer.

Many OO languages provide a standard class library that can be extended by the users, thus saving a lot of coding and debugging effort. Reducing the amount of code simplifies understanding and thus allows to build reliable programs. Code reuse is possible in conventional languages as well, but OO languages greatly enhance the possibility of reuse.

Object-oriented design involves the identification and implementation of different classes of objects and their behavior. The objects of the system closely correspond and relate in a one-to-one manner to the objects in the real world. Thus, it is easier to design and implement the system consisting of objects, as observed and understood by the brain.

Object orientation provides many other advantages in the production and maintenance of software: shorter development times, high degree of code sharing and malleability (can be moulded to any shape). These advantages make OOPs an important technology for building complex software systems.

Disadvantages

The runtime cost of dynamic binding mechanism is the major disadvantage of object oriented languages. The following were the demerits of adopting object-orientation in software developments in the early days of computing (some remain forever):

- Compiler overhead
- Runtime overhead
- Re-orientation of software developer to object-oriented thinking
- Requires the mastery over the following areas:
  - Software Engineering
  - Programming Methodologies
- Benefits only in long run while managing large software projects, atleast moderately large ones.

Object oriented concepts are becoming important in many areas of computer science, including programming, graphics, CAD systems, databases, user interfaces, application integration platforms, distributed systems and network management architectures. OO technology is more than just a way of programming. It is a way of thinking abstractly about a problem using real world concepts rather than computer concepts.

Although object orientation has been around for many years, it is only recently that it has received major attention from vendors and methodologists. OO programming is gradually picking up as an important technology for building complex software systems. For any programming language to succeed, it must be easy to learn i.e., programmers must be able to master language constructs easily; they must be able to reuse code written by them earlier without much modifications in a new software project; and above all, the programming language should be received well by application and system software developers. The following sections (OO Learning Curve, Software Reuse, and Objects Hold the Key) discuss these issues by taking object-oriented methodologies into consideration.

1.16 OO Learning Curve

The transition from an early linear programming language, BASIC, to the latest structured programming language, C, is easy as long as an `if` statement is an `if` statement, and a *function* is a *function* regardless of the language. While using function oriented methodology, the programmers need not think in terms of a specific language, because the individual syntax and capabilities are generally equivalent.

Programming in an object oriented paradigm, is different from programming in function oriented paradigm. Object-oriented programs should be structurally different from function oriented programs. Whereas a function-oriented program is organized around the actions being performed, a well designed object-oriented program is organized according to the objects being manipulated. This shift in perspective causes trouble for function-oriented programmers stepping into an object-oriented programming environment. Obviously, they have to *unlearn* known concepts while switching to object-oriented programming. (The communication between subroutines takes place through an explicit call to a required subroutine in the functional languages; whereas in OO languages, it takes place through message communication.)

Object-oriented techniques have promised to produce faster, smaller, and easier-to-maintain programs. The difference between function-oriented and object-oriented programming is that the program-

mer must switch from designing programs based on actions to designing programs around data types and their interactions.

The designer of C++, Bjarne Stroustrup, recommends that the shift from C to C++ should be a gradual one, with programmers learning the improvements a small step at a time. With C++, quite often, people, as a first exercise, write a string class and as a second exercise, try to implement a graphics system. That is very challenging and might be good for a professional programmer, but it's not the best way of teaching an average student programmer. What we need is an environment that has a very good string class that you can take apart and look at one which has a very nice graphics system, so that you never care about MS-windows or X-windows again, unless you absolutely want to. So, the two components needed to start OO programming are an *environment* and a *library* supporting resuability.

1.17 Software Reuse

Programmers have to write code from scratch, when a new software is being developed, using traditional languages, because there is hardly any reuse of the existing components. Software systemis have become so complex that even *coding is considered as a liability today.* Reusing existing software components is treated as a key element in improving software development productivity. It facilitates the use of existing well tested and proven software code as a base module and then develop on it, instead of developing from scratch. The simplest approach in this direction involves the development and use of libraries of software components.

Once a class has been developed, implemented, and tested, it can be distributed to other programmers for use in their programs (called reusability). It is similar to the way library functions are used in different programs. However, in OOP, the concept of inheritance provides an important extension to the idea of reusability. A programmer can use an existing class without modifying it and add new additional features and capabilities to build a new class. A newly created derived class has all the inherited features of the old one with additional features of its own. The ease with which the existing software can be reused is a major benefit of OOP.

Reuse is becoming one of the key areas in dealing with the cost and quality of the software systems. The basis for reuse is the reliability of the components intended for reuse and gains achieved through its application. The components developed for reuse must have a quality stamp, for example, concerned with reliability and performance. Object-Oriented techniques make it possible to develop components in general, and to develop reusable components in particular.

One of the important problems of the software component reuse consists of their localization and retrieval from a large collection. In fact, reuse implies the following three actions: (i) Retrieve needed component, (ii) Understand them, and (iii) Use them.

A method to reduce the effort of reusable components' search, comprehension, and adaptation consists of developing a reuse, strategy which defines a component classification, a component structure, and search-and-use mechanism. The OO concepts such as classes and inheritance provide a better mechanism for grouping related entities and simplifying the identification of reusable components.

Reuse through Inheritance and its Quantification

Inheritance is considered as an excellent way to organize abstraction, and as a tool to support reuse. The use of inheritance does have some trade-offs (costs) - inheritance increases the complexity of the system and the coupling between classes. Booch recommends that inheritance hierarchies be built as balanced lattices and that the maximum number of levels and their width be limited to 7 ± 2 classes.

A study of inheritance was conducted on nineteen C++ software systems ranging from language tools, Graphical User Interfaces and toolkits, applications, thread packages from public domain to proprietary systems implemented using C++. It revealed that, only 37% of the systems have a median class inheritance depth greater than 1. However, an individual inheritance tree can be deep.

The inheritance depth varies from system to system depending on the application domain. Software systems that have been designed as applications also differ notably from the reuse libraries. The Graphical User Interface (GUI) applications tend to have greater reuse through inheritance. GUI software are more suitable for design with inheritance. The reuse of classes in a reusable software library is more than in an application system. Developers put more effort into the design of reusable libraries than application software. Therefore, the reuse software library developer can take greater advantage of inheritance. Experiments have revealed that, a lot of code and standard structures are common in many applications and a great improvement in programmers' productivity can be achieved by code reusability. Before the use of software components become an established methodology (code reuse), major efforts are needed in the area of reusable data, reusable architecture, and reusable design.

Reusable Data: The concept of reusable data implies a standard data interchange format. However there is no universal format to allow easy transport of data from one system to another.

Reusable Architecture: The architecture of reusable components should have the following attributes:

- all data descriptions should be external to the programs or modules intended for reuse
- all literals and constants should be external to the programs or modules intended for reuse
- all input/output controls should be external to the programs or modules intended for reuse
- the programs or modules intended for reuse should consist primarily of application logic

Reusable Design: A factor affecting the software reusability is the non-availability of good design principles for major application types. OO software components can be designed in a consistent way and can become a defacto standard for further development.

Reuse and Porting

Software reuse refers to the usage of existing software knowledge or artifacts to build new software artifacts. It is sometimes confused with porting. Reuse and porting are distinguished as follows: *Reuse* refers to using an asset in different systems; *Porting* is moving a system across different environments (moving software from DOS to UNIX operating system) or platforms (moving software from x86 to SUN's UltraSPARC processor). For example, in Figure 1.21, a component in System A is used again in System B, which is an example of reuse. System A, developed for Environment 1, is moved into Environment 2, which is an example of porting.

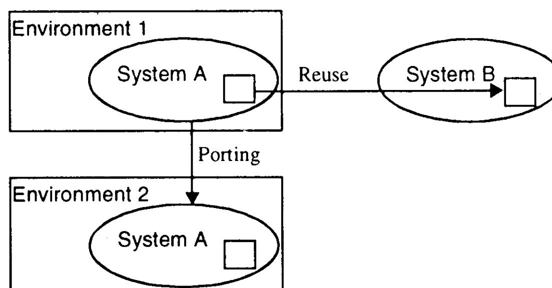
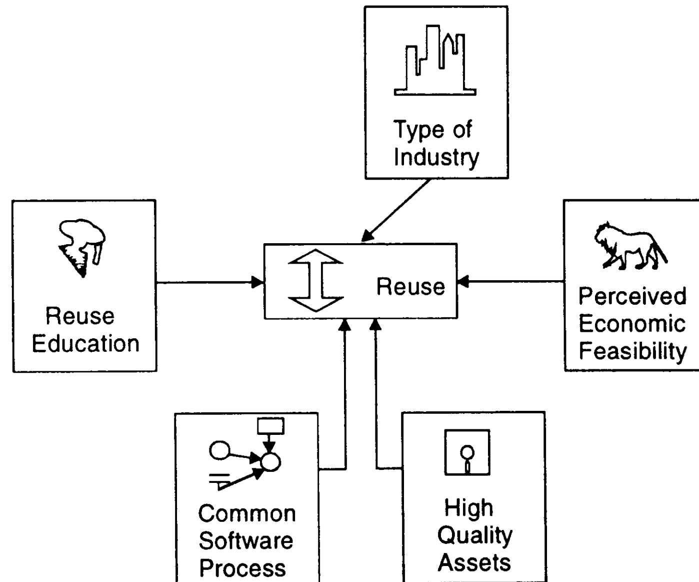


Figure 1.21: Reuse versus porting

Factors Influencing Reuse

An organization trying to improve systematic reuse, should concentrate on educating developers about reuse so as to improve their understanding of the economic feasibility of reuse, instituting a common development process, and making high-quality assets available to developers (see Figure 1.22a). The other factors (see Figure 1.22b), do not seem to be important, inspite of conventional wisdom. It should be understood, however, that these conclusions are based on data gathered from the industries; the salient factors of a particular organization may be different. The best course is to investigate the factors affecting reuse in the target organization (through surveys, case studies, or other techniques), and take action based on those results.



(a) Factors Affecting Reuse

(b) Factors Not Affecting Reuse

Figure 1.22: Effects on systematic reuse of the factors

1.18 Objects Hold the Key

Popularity of OOPs in the development of most software systems with ease, has created a great deal of excitement and interest among software communities. OOP finds its application from design of database systems to the future generation operating systems, which have *computing*, *communication*, and *imaging* capabilities built into it. Today, OOP is used extensively in the design of Graphics User Interfaces on systems such as Windows. Some promising applications of OOP include the following:

- Object-Oriented Database Systems
- Object-Oriented Operating Systems
- Graphical User Interfaces
- Window based Operating System Design
- Simulation and Modeling Studies
- Multimedia Applications
- Design Support Systems
- Office Automation Systems
- Real-Time Systems
- Computer Aided Design/Manufacturing (CAD/CAM) systems
- Computer-Based Training and Educational Systems

The object-oriented paradigm, which initially started with the introduction of OO programming languages, has moved into design, and recently even into analysis. Thus, new object technologies such as object-oriented analysis and object-oriented design have emerged and are getting mature. OO technology not only increases the productivity of the developer, but also increases the quality of the software systems. A software designer will think, analyze, design, implement, and even maintain future software systems in terms of object-oriented technology.

OOP-based computing solutions are expected to hold the key in the development of application and system software. Operating systems (OSs') of the future will be OOPs-based and compatibility and interoperability will no longer be a critical issue. *OOPs is to tomorrow's OSs' what C means to UNIX* in the form of portability. In fact, UNIX and C are a made-for-each-other couple. Sophisticated features of today's operating systems like Networking, Internet Connectivity, Multimedia, Database management, etc., will all be represented as objects. Spreadsheets can look up data by automatically retrieving it from a database. Object-based Internet connectivity feature can automatically locate information on the World Wide Web (WWW) and load this data into the local database. It would lead to fewer bugs and the burden on virtual memory would be reduced by a large degree, since the code would be smaller. Instead of using swap files the way most applications do today, tomorrow's programs will communicate by passing messages through data structures in memory. A background program will monitor and continually clear up the stack, heap, and other critical data structures, thus reducing chances of a system crash and making them *stable and reliable* computing entities. Objects no longer in use will be automatically cleaned up by making use of destructors and the RAM made available dynamically.

The features discussed above resembles Plug-and-Play, which allows a call to any object and get the job done anywhere (local or remote computing); and there will be no linking of applications (applications will be dynamically linked when they are called upon to perform a particular task). System down time due to reinstallation will just disappear. New objects will be automatically added and made available to any program that needs them, thereby eliminating the redundancy of code. OOPs is an indispensable part of the future, and it calls for an unconditional restructuring of today's methodologies. These features will automatically migrate to tomorrow's operating systems.

The usage of OO concepts in the development of futuristic operating systems sounds impossible yet fascinating. An OO-based operating system, Oberon, has already been implemented by Nicklaus Wirth, the chief proponent of Pascal and Modula-2. Another implementation of Object-Oriented OS is *Cronus*. Cronus is a distributed operating system developed at BBN Laboratories Inc., Massachusetts, to interconnect cluster of heterogeneous computers on high-speed LANs (Local Area Networks). It supports three types of objects: *primal objects* (bound forever to the host that created them), *migrating objects* (basis for system reconfiguration-load balancing to improve performance), and *replicated objects* (to achieve survivability).

Object-Oriented Programming has made long lasting changes in programming methodology. The old style of programming referred to as structured programming is now dead. OOP has emerged as the winner. All new operating systems and development tools will support OOPs and make the life of the programmer easier and the life of the program longer. Revolutionary features of modern operating systems such as Object Linking and Embedding (OLE) in Microsoft Windows have given rise to the Common Object Model (COM), which is expected to become a standard and leading Object-Oriented Operating System.

Review Questions

1.1 What is a software crisis ? Justify the need for a new programming paradigm. Explain how object-oriented paradigm overcomes this software crisis.

1.2 What is object-oriented paradigm ? Explain the various features of OO paradigm.

1.3 Define the following terms related to OO paradigm:
a) Encapsulation **b)** Data abstraction **c)** Inheritance **d)** Multiple Inheritance **e)** Polymorphism **f)** Message Passing **g)** Extensibility **h)** Persistence **i)** Delegation **j)** Containership **k)** Genericity **l)** Abstract Data Types **m)** Objects **n)** Classes

1.4 What are the programming paradigms currently available ? Explain their features with programming languages supporting them.

1.5 Compare structured and OO Programming paradigms.

1.6 What are the elements of Object-Oriented Programming ? Explain its key components such as objects and classes with examples.

1.7 Write an object representation (pictorial) of `Student` class.

1.8 Explain multiple views of an object with a suitable example.

1.9 What is the difference between inheritance and delegation ? Illustrate with examples.

1.10 List different methods of realizing polymorphism and explain them with examples.

1.11 What are the steps involved in OO Programming ? Explain its message communication model.

1.12 List some popular OOP Languages and compare their object-oriented features.

1.13 Which is the first object-oriented language ? Explain the heritage of C++.

1.14 What is Java ? Why is this language gaining popularity now-a-days ?

1.15 Discuss the merits and demerits of object-oriented methodologies.

1.16 What is software reuse ? What is the difference between reuse and porting ? What are the factors influencing the software reuse ?

1.17 Identify reusable components in software and discuss how OOPs helps in managing them.

1.18 Justify "Objects hold the key." List some promising areas of applications of OOPs. Discuss how object-oriented paradigm affects different elements of computing such as hardware architectures, operating systems, programming environments, and applications ?

2

Moving from C to C++

2.1 Introduction

C++ has borrowed many features from other programming languages. It includes the commenting style from BCPL, the class concept with derived classes and virtual functions from Simula 67. It owes the concept of operator overloading and freedom to place definitions wherever necessary, to Algol 108, while the template facility and inline functions were borrowed from Ada. The concept of parametrized modules is borrowed from Clu programming language.

This chapter is a guideline for C programmers to transit from C to C++ programming without really bothering about C++'s OOP features. Mastering *non-class* features of C++ will provide impetus to the user to appreciate the influence of object oriented concepts over the conventional style of programming. Even if the programmers are not interested in OO programming, the other benefits, which are essential for structured programming with C, can be found in a more powerful form in C++. For instance, features such as strict prototyping as demanded by the compiler and others such as function overloading, single-line comment, function templates, etc., greatly improve productivity of the programmer. The various non-OOP features supported in C++ have greater role to play while writing OOP based programs.

2.2 Hello World

Similar to C, C++ programs must contain a function called `main()`, from which execution of the program starts. The function `main()` is designated as the starting point of the program execution and it is defined by the user. It cannot be overloaded and its syntax type is implementation dependent. Therefore, the number of arguments and their data-type is dependent on the compiler. The most popularly used format for defining the function `main()` is shown below:

```
void main()
{
   ....
 // Program Body
  ....
}
```

The traditional beginner's C program, usually called *Hello World,* is listed in `hello.c`. It has one of the heavily used header file `stdio.h`, included for supporting standard I/O operations. The `printf` statement outputs the string message `Hello World` on the console. The function body consists of statements for creating data storage variables called *local variable* and executable statements. Note that although the program execution starts from the `main()`, the data variables defined by it are not visible to any other function. With all the pieces of the program in place, a *driver* is needed to initialize and start things. The function `main()` serves as a driver function.

```
/* hello.c: printing Hello World message */
#include <stdio.h>
void main()
{
   printf( "Hello World" );
}
```

Run:

```
Hello World
```

The standard C library function `printf()` sends characters to the standard output device. The *Hello World* program will also work in C++, since it supports the ANSI-C function library. However, the program could be rewritten using C++ streams. The C++ equivalent of the Hello World program is listed in the program `hello.cpp`.

```
// hello.cpp: printing Hello World message
#include <iostream.h>
void main()
{
   cout << "Hello World";
}
```

Run:

```
Hello World
```

The header file `iostream.h` supports streams programming features by including predefined stream objects. The C++'s stream insertion operator, << sends the message `"Hello World"` to the predefined console object, `cout`, which in turn prints on the console. The Hello World program in C++ is shown in Figure 2.1 for the purpose of comparative analysis.

Code	Label
`1: // hello.cpp: printing Hello World message`	comment
`2: #include <iostream.h>`	preprocessor directive
`3: void main( )`	function declarator
`4: {`	function begin
`5:    cout << "Hello World";`	body of the function main
`6: }`	function end

Figure 2.1: Hello World program in C++

The various components of the program `hello.cpp`, shown in Figure 2.1, are discussed in the following section:

First Line - Comment Line

The statement which starts with symbols // (i.e., two slash characters one after another without a space) is treated as comment. Hence the compiler ignores the complete line starting from the // character pair.

Although comments do not contribute to the runtime of a program, when used properly, they are the most valuable part of a piece of source code.

The word cpp, in the program hello.cpp, is an acronym for CPlusPlus (C++). The compiler will recognize program as a C++ program only when it has an extension cpp. (However, the extension is compiler dependent and most of the compilers assume cpp as default extension. Some C++ compilers such as GNU under UNIX system, expect program files to have cc as an extension).

Second Line - Preprocessor Directive

The second line is a preprocessor directive. The preprocessor directive

```
#include <iostream.h>
```

includes all the statements of the header file iostream.h. It contains instructions and predefined constants that will be used in the program. It plays a role similar to that of the header file stdio.h of C. The header file iostream.h contains declarations that are needed by the cout and cin stream objects. There are a number of such preprocessor directives provided by the C++ library, and they have to be included depending on the built-in functions used in the program. In addition, the users can also write preprocessor directives and declare them in the beginning of the program (usually, but they can be declared anywhere in the program). In effect, these directives are processed before any other executable statements in the source file of the program by the compiler.

Third Line - Function Declarator

The third line in the program is

```
void main()
```

Similar to a C program, the C++ program also consists of a set of functions. Every C++ program must have one function with name main, from where the execution of the program begins. The name main is a special word (not a reserved word) and must not be invoked anywhere by the user. The names of the functions (except main) are coined by the programmer. The function name is followed by a pair of parentheses which may or may not contain arguments. In this case, there are no arguments, but still the parentheses pair is mandatory. Every function is supposed to return a value, but the function in this example does not return any value. Such function names must be preceded by the reserved word void.

Fourth Line - Function Begin

The function body in a C/C++ program, is enclosed between two flower brackets. The opening flower bracket ({) marks the beginning of a function. All the statements in a function, which are listed after this brace can either be executable or non-executable statements.

Fifth Line - Function Body

The function body contains a statement to display the message Hello World. The output statement cout is pronounced as C-out (meaning Console Output). It plays a role similar to that of the printf() in C. The first statement in the main() body (of course it is the last statement in the main() body in this case)

```
cout << "Hello World";
```

prints the message "Hello World" on the standard console output device (VDU, video display unit by default). It plays the role of the statement

```
printf( "Hello World" );
```

as in the hello.c program.

Sixth Line - Function End

The end of a function body in a C/C++ program is marked by the closing flower bracket (}). When the compiler encounters this bracket, it is replaced by the statement,

```
return;
```

which transfers control to a caller. In this program, the last line actually marks the end of program and control is transferred to the operating system on termination of the program.

Compilation Process

The C++ program `hello.cpp`, can be entered into the system using any available text editor. Some of the most commonly available editors are Norton editor (`ne`), `edline`, `edit`, `vi` (most popular editor in UNIX environment). The program coded by the programmer is called the *source code*. This source code is supplied to the compiler for converting it into the *machine code*.

C++ programs make use of libraries. A library contains the object code of standard functions. The object code of all functions used in the program have to be combined with the program written by the programmer. In addition, some *start-up code* is required to produce an executable version of the program. This process of combining all the required object codes and the start-up code is called *linking* and the final product is called the *executable code*.

Most of the modern compilers support sophisticated features such as multiple window editing, mouse support, on-line help, project management support, etc. One such compiler is Borland C++. It can be invoked through command-line or integrated development environment (refer to Borland C++ developers guide).

Command Line Compilation

Most of the compilers support the command line compilation of a program. All the required arguments are passed to the compiler from the command line. For the purpose of discussion, consider the Borland C++ compiler. (However this process is implementation dependent. For more details, refer to the manual supplied by the compiler vendor.)

The command-line compiler is invoked by issuing the command:

```
tcc filename.cpp  (in the case of Turbo C++)
bcc filename.cpp  (in the case of Borland C++)
```

at the DOS prompt. It creates an object file `filename.obj`, and an executable file `filename.exe`. In the case of multiple file compilation, they must be compiled through `-c` option to create only the object file as follows:

```
tcc/bcc -c filename.cpp
```

The linker is invoked to link multiple object files and to create an executable file through the explicit issue of the linking command:

```
tlink filename1.obj  filename2.obj <library name>
```

The library file can also be passed as a parameter to the linker for binding functions defined in it. To create the executable of `hello.cpp`, issue the command `bcc hello.cpp` at the MS-DOS prompt.

2.3 Streams Based I/O

C++ supports a rich set of functions for performing input and output operations. The syntax of using these I/O functions is totally consistent, irrespective of the device with which I/O operations are

performed. C++'s new features for handling I/O operations are called streams. Streams are abstractions that refer to data flow. Streams in C++ are classified into

- Output Streams
- Input Streams

Output Streams

The output streams allow to perform write operations on output devices such as screen, disk, etc. Output on the standard stream is performed using the `cout` object. C++ uses the bit-wise left-shift operator for performing console output operation. The syntax for the standard output stream operation is as follows:

```
cout << variable;
```

The word `cout` is followed by the symbol <<, called the insertion or put-to operator, and then with the items (variables/constants/expressions) that are to be output. Variables can be of any basic data type. The use of `cout` to perform an output operation is shown in Figure 2.2.

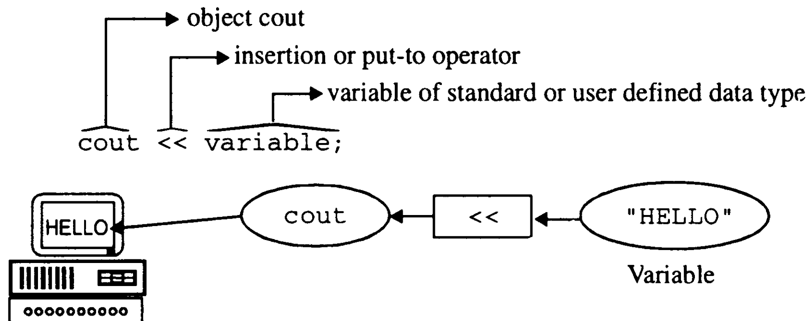


Figure 2.2: Output with cout operator

The following are examples of stream output operations:

```
1.   cout << "Hello World";
2.   int age;
     cout << age;
3.   float weight;
     cout << weight;
4,   double area;
     cout << area;
5.   char code;
     cout << code;
```

More than one item can be displayed using a single `cout` output stream object. Such output operations in C++ are called *cascaded output operations*. For example, output of the age of a person along with some message can be performed by `cout` as follows:

```
cout << "Age = " << age;
```

The `cout` object will display all the items from left to right. Hence, in the above case, it prints the message string `"Age = "` first, and then prints the value of the variable `age`. C++ does not enforce any restrictions on the maximum number of items to be output. The complete syntax of the standard

output streams operation is as follows:

```
cout << variable1 << variable2 << .. << variableN;
```

The object `cout` must be associated with at least one argument. Like `printf`, a constant value can also be sent as an argument to the `cout` object. Following are some valid output statements

```
cout << 'H';
cout << "Hello";
cout << 420;
cout << 90.25;
cout << 1234567;
cout << " ";  // will display blank
cout << "\n"; // prints new line
cout << x << " " << y;
```

The last output statement prints the value of the variable x followed by a blank character, and then the value of the variable y.

The program `output.cpp` demonstrates the various methods of using `cout` for performing output operation.

```
// output.cpp: display contents of variables of different data types
#include <iostream.h>
void main()
{
   char sex;
   char *msg = "C++ cout object";
   int age;
   float number;
   sex = 'M';
   age = 24;
   number = 420.5;
   cout << sex;
   cout << " " << age << " " << number;
   cout << "\n" << msg << endl;
   cout << 1 << 2 << 3 << endl;
   cout << number+1;
   cout << "\n" << 99.99;
}
```

Run

```
M 24 420.5
C++ cout object
123
421.5
99.99
```

The item `endl` in the statement

```
cout << "\n" << msg << endl;
```

serves the same purpose as "\n" (linefeed and carriage return) and is known as a *manipulator*. It may be noticed that there is no mention of the data types in the I/O statements as in C. Hence, I/O statements of C++ are easier to code and use. C++, as a superset of C, supports all functions of C, however, they are not used in the above C++ program.

Input Streams

The input streams allow to perform read operation with input devices such as keyboard, disk, etc. Input from the standard stream is performed using the `cin` object. C++ uses the bit-wise right-shift operator for performing console input operation. The syntax for standard input streams operation is as follows:

```
cin >> variable;
```

The word `cin` is followed by the symbol >> (extraction operator) and then with the variable, into which the input data is to be stored. The use of `cin` in performing an input operation is shown in Figure 2.3.

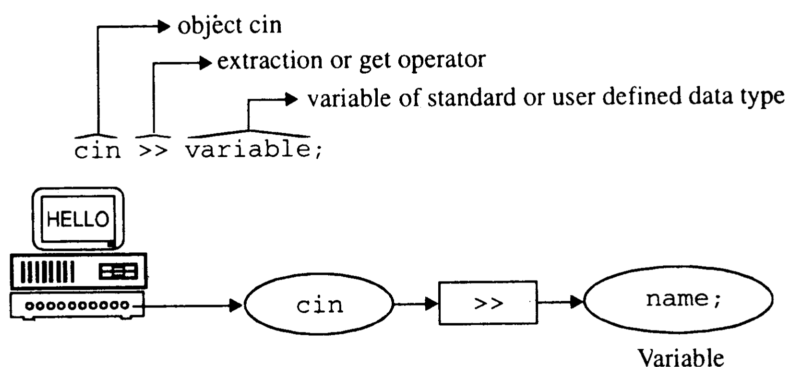


Figure 2.3: Input with cin operator

The following are examples of stream input operations:

1.
```
int age;
cin >> age;
```
2.
```
float weight;
cin >> weight;
```
3.
```
double area;
cin >> area;
```
4.
```
char code;
cout >> code;
```
5.
```
char name[20];
cin >> name;
```

Input of more than one item can also be performed using the `cin` input stream object. Such input operations in C++ are called *cascaded input operations*. For example, reading the name of a person followed by the age, can be performed by the `cin` as follows:

```
cin >> name >> age;
```

The `cin` object will read all the items from left to right, Hence, in the above case, it reads the name of the person as a string (until first blank) first, and then the age of person into the variable `age`. C++ does not impose any restrictions on the number of items to be read. The complete syntax of the standard input streams operation is as follows:

```
cin >> variable1 >> variable2 >> .. >> variableN;
```

The object `cin`, must be associated with at least one argument. Like `scanf()`, constant values cannot be sent as an argument to the `cin` object. Following are some valid input statements:

```
    cin >> i >> j >> k;
    cin >> name >> age >> address;
```

The program `read.cpp` demonstrates the various methods of using `cin` for performing input operation.

```
// read.cpp: data input through cin object
#include <iostream.h>
void main()
{
   char name[25];
   int age;
   char address[25];
   // read data
   cout << "Enter Name: ";
   cin >> name;
   cout << "Enter Age: ";
   cin >> age;
   cout << "Enter Address: ";
   cin >> address;
   // output data
   cout << "The data entered are:" << endl;
   cout << "Name = " << name << endl;
   cout << "Age = " << age << endl;
   cout << "Address = " << address;
}
```

Run

```
Enter Name: Rajkumar
Enter Age: 24
Enter Address: C-DAC-Bangalore
The data entered are:
Name = Rajkumar
Age = 24
Address = C-DAC-Bangalore
```

Performing I/O operations through the `cout` and `cin` are analogous to the `printf` and `scanf` of the C language, but with different syntax specifications. The following are two important points to be noted about the stream operations.

- Streams do not require explicit data type specification in I/O statement.
- Streams do not require explicit address operator prior to the variable in the input statement.

In `scanf` and `printf` functions, format strings are necessary, while in the `cin` stream format specification is not necessary, and in the `cout` stream format, specification is optional. Format-free input and output are special features of C++, which make I/O operations comfortable for beginners. The input stream `cin` accepts both numbers and characters, when the variables are given in the normal form. The function `scanf` requires ampersand (`&`) symbol to be prefixed to a numeric or a character variable, (whereas, the string variables can be given as they are). One must, therefore, carefully follow the syntax requirements in coding the different statements.

Another point to be noticed is that, the operator << , is the same as the left-shift bit-wise operator and the operator >>, is the same as the right-shift bit-wise operator used in C and also in C++. In C++, operators can be overloaded, i.e., the same operator can perform different activities depending on the context (types of data-items with which they are associated). The `cout` is a predefined object in C++, which corresponds to the output stream, and `cin` is an object in the input stream. Different objects are instructed to do specified jobs.

2.4 Single Line Comment

C++ has borrowed the new commenting style from Basic Computer Programming Language (BCPL), the predecessor of the C language. In C, comment(s) is/are enclosed between /* and */ character pairs. It can be either used for single line comment or multiple line comment.

Single line comment runs across only one line in a source program. The statement below is an example of single line comment:

```
/* I am a single line comment */
```

Multiple line comment runs across two or more lines in a source program. The statement below is an example of multiple line comment.

```
/* I am a multiple line comment.
   Hope you got it. */
```

Apart from the above style of commenting, C++ supports a new style of commenting. It starts with two forward slashes i.e., // (without separation by spaces) and ends with the end-of-line character. The syntax for the new style of C++ comment is shown in Figure 2.4.

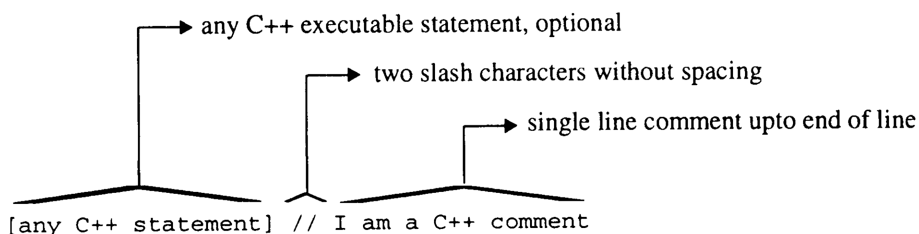


Figure 2.4: Syntax of single line comment

The following examples illustrate the syntax of C++ comments:

```
int acc;            // Account Number
acc = acc + 1;      // adding new account number for new customer
```

In C, the above two statements are written as

```
int acc;            /* Account Number */
acc = acc + 1;      /* adding new account number for new customer */
```

The above examples of comments indicate that, C++ commenting style is easy and quicker for single line commenting. Although, C++ supports C style of commenting, it is advisable to use the C style for commenting multiple lines and the C++ style for commenting a single line.

Some typical examples of commenting are listed below:

1. `// this is a new style of comment in C++`
2. `/* this is an old style of comment in C++ */`
3. `// style of comment runs to the end of a line`
4. `/* runs to any number of lines but hard to type and takes up more space and coding time also. */`
5. (i) `/* Here is a comment followed by an executable statement */ a = 100;`
   (ii) `// Here is a comment followed by a non-executable statement a = 100;`

The statement (i) has a comment followed by an executable statement `a = 100;` but, the statement (ii) is entirely treated as a commented line.

Large programs become hard to understand even by the original author (programmer), after some time has passed. Even a few well-placed comments which explain *why* and *what* of a variable, expression, statement, or block, help tremendously. Comments that simply restate the nature of a line of code, obviously do not add much value, but comments which explain the algorithm are the mark of a good programmer.

Comments are integral part of any program and they help in program coding and maintenance. The compiler completely ignores comments, therefore, they do not slow down the execution speed, nor do they increase the size of the executable program. Comments should be used liberally in a program and they should be written during the program development, but not as an after-thought activity.

The program `simpint.cpp` for computing the simple interest demonstrates how comments aid in the understanding and improving redability of the source code.

```
// simpint.cpp: Simple interest computation
#include <iostream.h>
void main()
{
   // data structure definition
   int principle;     // principle amount
   int time;          // time in years
   int rate;          // rate of interest
   int SimpInt;       // Simple interest
   int total;         // total amount to be paid back after 'time' years
   // read all the data required to compute simple interest
   cout << "Enter Principle Amount: ";
   cin >> principle;
   cout << "Enter Time (in years): ";
   cin >> time;
   cout << "Enter Rate of Interest: ";
   cin >> rate;
   // compute simple interest and display the results
   SimpInt = (principle * time * rate) / 100;
   cout << "Simple Interest = ";
   cout << SimpInt;
   // total amount = principle amount + simple interest
   total = principle + SimpInt;
   cout << "\nTotal Amount = ";
   cout << total;
}
```

Run

```
Enter Principle Amount: 1000
Enter Time (in years): 2
Enter Rate of Interest: 5
Simple Interest = 100
Total Amount = 1100
```

2.5 Literals–Constant Qualifiers

Literals are constants, to which symbolic names are associated for the purpose of readability and ease of handling standard constant values. C++ provides the following three ways of defining constants:

- `# define` preprocessor directive
- enumerated data types
- `const` keyword

The variables in C can be created and initialized with a constant value at the point of its definition. For instance, the statement

```
float PI = 3.1452;
```

defines a variable named PI, and is assigned with the floating-point numeric constant value 3.1452. It is known that the constant value does not change. In the above case, the variable PI is considered as a constant, whose value does not change throughout the life of the program (complete execution-time). However, an accidental change of the value of the variable PI is not restricted by C. C++ overcomes this by supporting a new constant qualifier for defining a variable, whose value cannot be changed once it is assigned with a value at the time of variable definition. The qualifier used in C++ to define such variables is the `const` qualifier. The syntax of defining variables with the constant qualifier is shown in Figure 2.5. Note that if `DataType` is ommitted, it is considered as `int` by default.

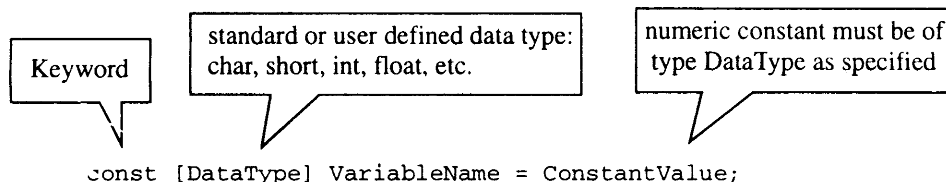


Figure 2.5: Syntax of constant variable definition

The following examples illustrate the declaration of the constant variables:

- `const float PI = 3.1452;`
- `const int TRUE = 1;`
- `const int FALSE = 0;`
- `const char *book_name = "OOPs with C++";`

The program `area.cpp`, illustrates the declaration and the use of constant variables.

```
// area.cpp: area of a circle
#include <iostream.h>
const float PI = 3.1452;
```

```
void main()
{
   float radius;
   float area;
   cout << "Enter Radius of Circle: ";
   cin >> radius;
   area = PI * radius * radius;
   cout << "Area of Circle = " << area;
}
```

Run

```
Enter Radius of Circle: 2
Area of Circle = 12.5808
```

In the above program, the use of the statement such as

```
PI = 2.3;
```

to modify a constant type variable leads to the compilation error: *Cannot modify a const object*

Thus, the keyword `const`, can be used before a type to indicate that the variable declared is constant, and may therefore not appear on the left side of the assignment (=) operator. In C++,the `const` qualifier can be used to indicate the parameters that are to be treated as read-only in the function body.

Consider the C program `disp.c`, having the function to display any string passed to it.

```
/* disp.c: display message in C */
#include <stdio.h>
#include <string.h>
void display( char *msg )
{
   printf( "%s", msg );
   /* modify the message */
   strcpy( msg, "Misuse" );
}
void main()
{
   char string[15];
   strcpy( string, "Hello World" );
   display( string );
   printf( "\n%s", string );
}
```

Run

```
Hello World
Misuse
```

The function `display()`, is supposed to output the input string argument passed to it onto the console. But accidental use of a statement such as

```
strcpy( msg, "Misuse" );
```

in `display()` modifies the input argument. This modification is also reflected in the calling function; (see the second message in the output) the string argument is a pointer type and any modification in function will also be reflected in the calling function. Such accidental errors can be avoided by defining

the input parameter with the `const` qualifier. The C++ program `disp.cpp` illustrates the mechanism of overcoming the problem of modifying constant variables.

```
// disp.cpp: display message in C++
#include <stdio.h>
#include <string.h>
void display( const char *msg )
{
   cout << msg;
   /* modify the message */
   // strcpy( msg, "Misuse" ); this produces a compilation error
}
void main()
{
   char string[15];
   strcpy( string, "Hello World" );
   display( string );
   cout << endl << string;
}
```

Run

```
Hello World
Hello World
```

The use of a statement such as,

```
strcpy( msg, "Misuse" );
```

in `display()` leads to a compilation error. Thus, reminding the programmer regarding the accidental modification of read-only type variables will protect from common programming errors.

2.6 Scope Resolution Operator ::

C++ supports a mechanism to access a global variable from a function in which a local variable is defined with the same name as a global variable. It is achieved using the *scope resolution operator*. The syntax for accessing a global variable using the scope resolution operator is shown in Figure 2.6.

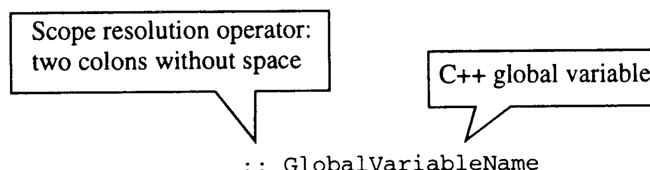


Figure 2.6: Syntax of global variable access

The global variable to be accessed must be preceded by the scope resolution operator. It directs the compiler to access a global variable, instead of one defined as a local variable. The program `global.cpp` illustrates the access mechanism to the global variable `num` from the function `main()`, which has a local variable by the same name. Thus, *the scope resolution operator permits a program to reference an identifier in the global scope that has been hidden by another identifier with the same name in the local scope.*

```
// global.cpp: global variables access through scope resolution operator
#include <iostream.h>
int num = 20;
void main()
{
   int num = 10;
   cout << "Local = " << num;          // local variable
   cout << "\nGlobal = " << ::num;     // global variable
   cout << "\nGlobal+Local = " << ::num+num;  // both local & global use
}
```

Run

```
Local = 10
Global = 20
Global+Local = 30
```

The program `loop.cpp` illustrates the accessing of local and global variables within a `for` loop. It also shows mixing of the single-line comment statement within a single executable statement.

```
// loop.cpp: local and global variables in a loop
#include <iostream.h>
int counter = 50;                 // global variable
int main ()
{
    register int  counter;        // local variable
    for(counter = 1;              // this refers to the
        counter < 10;             // local variable
        counter++)
    {
        cout << endl <<           // print new line followed by
                ::counter         // global variable
                /                 // divided by
                counter;          // local variable
    }
    return( 0 );
}
```

Run

```
50
25
16
12
10
8
7
6
5
```

2.7 Variable Definition at the Point of Use

In C, local variables can only be defined at the top of a function, or at the beginning of a nested block. In C++, local variables can be created at any position in the code, even between statements. Further-

more, local variables can be defined in some statements, just prior to their usage. The program `var1.cpp` defines the variable in the `for` statement and its scope continues even after the `for` statement.

```
// var1.cpp: defining variables at the point of use
#include <iostream.h>
int main()
{
   // variable i cannot be referred before 'for' statement
   for ( int i = 0; i < 5; i++ ) // variable i is defined and used here
      cout << i << endl;
   cout << i;   // i visible after the 'for' statement also
   return( 0 );
}
```

Run

```
0
1
2
3
4
5
```

In `main()`, the statement

```
for ( int i = 0; i < 5; i++ )
```

creates the variable `i` inside the `for` statement. The variable does not exist prior to the statement, but continues to be available as a local integer variable even after the block scope of the `for` statement. The statement outside the `for` loop

```
cout << i;
```

refers to the variable created in the `for` loop.

The program `def2.cpp` illustrates the scope of variables and the usage of scope resolution operator.

```
// def2.cpp: Variable scope demonstration
#include <iostream.h>
int a = 10; // global variable
void main()
{
   cout << a << "\n";      // uses global variable
   int a = 20;
   {
      int a = 30;
      cout << a << "\n";   // uses locally defined variable within a block
      cout << ::a << "\n"; // uses global variable
   } // variable a defined within a block goes out of scope here
   cout << a << "\n";      // uses local variable a defined near main()
   cout << ::a << "\n";    // uses global variable
}
```

Run

```
10
30
10
20
10
```

The definition of variables at any position in the code can reduce code readablity. Therefore local variables should be defined at the beginning of a function, following the first {, or they should be created at *intuitively right* places.

2.8 Variable Aliases–Reference Variables

C++ supports one more type of variable called reference variable, in addition to the value variable and pointer variables of C. Value variables are used to hold some numeric values; pointer variables are used to hold the address of (pointer to) some other value variables. Reference variable behaves similar to both, a value variable and a pointer variable. In the program code, it is used similar to that of a value variable, but has an action of a pointer variable. In other words, a reference variable acts as an alias (alternative name) for the other value variables. Thus, *the reference variable enjoys the simplicity of value variable and power of the pointer variable.* It does not provide the flexibility supported by the pointer variable. Unlike pointer variable, when a reference is bound to a variable, then its binding cannot be changed. All the accesses made to the reference variable are same as the access to the variable, to which it is bound. The general format of declaring the reference variable is shown in Figure 2.7.

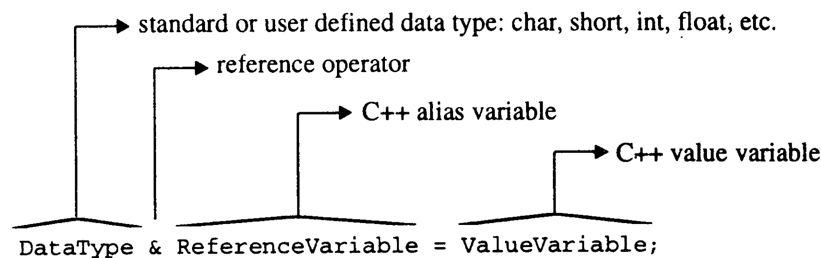


Figure 2.7: Syntax of reference variable declaration

The reference variable must be initialized to some variable only at the point of its declaration. Initialization of reference variable after its declaration causes compilation error. Hence, reference variables allow to create alias (another name) of existing variables. The following examples illustrate the concept of reference variables.

```
1.    char & ch1 = ch;        // ch1 is an alias of char ch
2.    int & a = b;            // a is an alias of int b
3.    float & x = y;
4.    double & height = length;
5.    int &x = y[100];        // x is an alias of y[100] element
6.    int n;
      int *p = &n;
      int &m = *p;
```

These declarations cause m to refer n, which is pointed to by the pointer variable p.

7. `int &num = 100;        // invalid`

This statement causes compilation error; constants cannot be made to be pointed to by a reference variable. Hence the rule, *no alias for constant value.*

Reference variables are not bounded to a new memory location, but to the variables to which they are aliases. For instance, the reference variable `height` is bound to the same memory location to which the value variable `length` is bound. The program `refvar.cpp`, illustrates the use of reference variables.

```
// refvar.cpp: reference variable for aliasing
#include <iostream.h>
void main()
{
  int a = 1, b = 2, c = 3;
  int &z = a;   // variable z becomes alias of a
  cout << "a=" << a << " b=" << b << " c=" << c << " z=" << z << endl;
  z = b;        // changes value of a to the value of b
  cout << "a=" << a << " b=" << b << " c=" << c << " z=" << z << endl;
  z = c;        // changes value of a to the value of c
  cout << "a=" << a << " b=" << b << " c=" << c << " z=" << z << endl;
  cout<<"&a=" << &a << " &b=" <<&b << "&c=" << &c << " &z=" << &z << endl;
}
```

Run

```
a=1 b=2 c=3 z=1
a=2 b=2 c=3 z=2
a=3 b=2 c=3 z=3
&a=0xfff4 &b=0xfff2 &c=0xfff0 &z=0xfff4
```

In `main()`, the statements

```
z = b;
z = c;
```

assign the value of variables b and c to the variable a since, the reference variable z is its alias variable. It can be observed that, in the last line of the above program output, the memory addresses of the variables a and z are same. The reference variables are bound to memory locations at compile time only. Consider the following statements:

```
int n;
int *p = &n;
int &m = *p;
```

Here m refers to n, which is pointed to by the variable p. The compiler actually binds the variable m to n but not to the pointer. If pointer p is bound to some other variable at runtime, it does not affect the value referenced by m and n. It is illustrated in the program `reftest.cpp`.

```
// reftest.cpp: testing of reference binding
#include <iostream.h>
void main()
{
   int n = 100;
   int *p = &n;
```

```
    int &m = *p;     // m is bound to n
    cout << "n = " << n << " m = " << m << " *p = " << *p << endl;
    int k = 5;
    p = &k;  // pointer value is changed
    k = 200;
    // is there change in m value ?
    cout << "n = " << n << " m = " << m << " *p = " << *p << endl;
}
```

Run

```
n = 100 m = 100 *p = 100
n = 100 m = 100 *p = 200
```

In `main()`, the statement

```
p = &k;  // pointer value changed
```

changes the pointer value of `p`, but does not effect the reference variable `m` and the variable `n`.

2.9 Strict Type Checking

C++ is a strongly-typed language and it uses very strict type checking. A prototype must be known for each function which is called, and the call must match the prototype. The prototype provides information of the type and number of arguments passed and it also specifies the return type (if any) of the function. In C++, function prototyping is compulsory if the definition is not placed before the function call whereas, in C, it is optional. The program `max.cpp` for computing the maximum of two numbers illustrates the need for the function prototype.

```
// max.cpp: maximum of two numbers
#include <iostream.h>
int main ()
{
    int x, y;
    cout << "Enter two integers: ";
    cin >> x >> y;
    cout << "Maximum = " << max( x, y ); // Error max.cpp 11:...
    return 0;
}
int max( int a, int b )
{
    if( a > b )
        return a;
    else
        return b;
}
```

Compilation of the above program produces the following errors:

```
Error max.cpp 11: Function 'max' should have a prototype in function main()
```

C++ checks all the parameters passed to a function against its prototype declaration during compilation. It produces errors if there is a mismatch in argument types and this can be overcome by placing the prototype of the function `max()` before it is invoked. The modified program of `max.cpp` is listed in `newmax.cpp`, which is compiled without any errors.

```
// newmax.cpp: maximum of two numbers
#include <iostream.h>
int max( int a, int b );      // prototype of max
void main ()
{
   int x, y;
   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << "Maximum = " << max( x, y );
}
int max( int a, int b )
{
   if( a > b )
      return a;
   else
      return b;
}
```

Run

```
Enter two integers: 10 20
Maximum = 20
```

The advantages of strict type checking is that the compiler warns the users if a function is called with improper data types. It helps the user to identify errors in a function call and increases the reliability of a program. The program `swap_err.cpp` shows notification of the compiler, when improper data type parameters are passed to the function. The program `swap_err.cpp` illustrates the detection of the statement calling the function with improper data items.

```
// swap_err.cpp: swap integer values by reference
#include <iostream.h>
void swap( int * x, int * y )
{
   int t;   // temporarily used in swapping
   t = *x;
   *x = *y;
   *y = t;
}
void main()
{
   int a, b;
   swap( &a, &b );      // OK
   float c, d;
   swap( &c, &d );      // Errors
}
```

The compilation of the above program produces the following errors:

Error swap_err.cpp 20: Cannot convert 'float *' to 'int *' in function main()
Error swap_err.cpp 20: Type mismatch in parameter 'x' in call to 'swap(int *,int *)' in function main()
Error swap_err.cpp 20: Cannot convert 'float *' to 'int *' in function main()
Error swap_err.cpp 20: Type mismatch in parameter 'y' in call to 'swap(int *,int *)' in function main()

The above errors are produced due to the following statement in `main()`

```
swap( &c, &d );      // Compilation Errors
```

Because the expressions `&c` and `&d` passed to `swap()` are not pointers to integer data type. When a call to a function is made, the C++ compiler checks its parameters against the parameter types declared in the function prototype. The compiler flags errors if improper arguments are passed.

2.10 Parameters Passing by Reference

A function in C++ can take arguments passed by value, by pointer, or by reference. The arguments passed by reference is an enhancement over C. A copy of the actual parameters in the function call is assigned to the formal parameters in the case of pass-by-value, whereas the address of the actual parameters is passed in the case of pass-by-pointer. In the case of pass-by-reference, an alias (reference) of the actual parameters is passed. Mechanism of parameter linkage is shown in Figure 2.8.

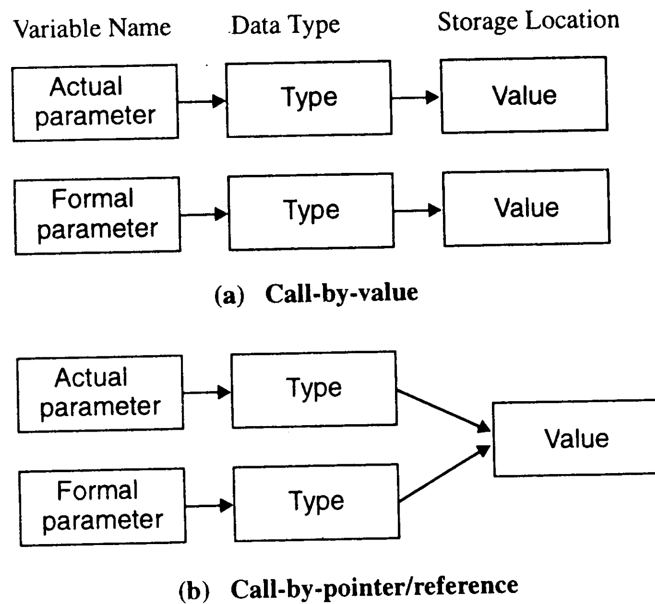


Figure 2.8: Parameter passing mechanism

Consider an example of swapping two numbers to illustrate the mechanism of parameter passing by reference. The function definition with pointer type parameters is listed below:

```
void swap( int * p, int * q ) // by pointers
{
   int t;
   t = *p;
   *p = *q;
   *q = t;
}
```

A call to the function `swap()`

```
swap( &x, &y )
```

has effect on the values of x and y i.e, it exchanges the contents of variables x and y. The above swap(..) function can be redefined by using a new parameter passing scheme, call by reference, as follows:

```
void swap( int & x, int & y ) // by reference
{
   int t;
   t = x;
   x = y;
   y = t;
}
```

A call to the function swap()

```
swap( x, y );
```

with integer variables x and y, has effect on the values of x and y variables. It exchanges the contents of the variables x and y. The body and the call to the function swap appears same as that of call-by-value case, but has an effect of call-by-pointer. Thus, call by reference combines the flexibility (ease of programming) of call by value and the power of call by pointer.

The complete program having swap(..) function with call-by-reference mechanism for parameter passing is listed in swap.cpp.

```
// swap.cpp: swap integer values by reference
#include <iostream.h>
void swap( int & x, int & y ) // by reference
{
   int t;   // temporary variable used in swapping
   t = x;
   x = y;
   y = t;
}
void main()
{
   int a, b;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   swap( a, b );
   cout << "On swapping <a, b>: " << a << " " << b;
}
```

Run

```
Enter two integers <a, b>: 2 3
On swapping <a, b>: 3 2
```

In main(), the statement

```
swap( a, b );
```

is translated into

```
swap( & a, & b );
```

internally during compilation; the prototype of the function

```
void swap( int & x, int & y ) // by reference
```

indicates that the formal parameters are of reference type and hence, they must be bound to the memory