Once the software is developed, it has to be subjected to tests at module (unit) level, module integration level, software/hardware integration (system) level and finally at the system level. *Module testing* focuses on individual software units or related group of units. *Module integration testing* focuses on combining software and hardware units, to evaluate the interaction among them. *System testing* focus on complete, integrated systems to evaluate compliance with requirement specification.

A module has to be tested for logical errors and computational errors while the interface is checked to see whether the interaction between the modules are proper. The techniques that have been proposed for unit testing include the following:

- Path testing: each possible path from input to output is traversed once.
- Branch testing: each path must be traversed at least once.
- Functional testing: each functional decomposition is tested at least once.
- Special values testing: testing for all values assumed to cause problems.
- Anomaly analysis: testing the program constructs that can cause problems.
- Interface analysis: testing for problems at module interfaces.

Maintenance: Once the system is put into operation, it must be maintained, which includes fixing bugs discovered during operation, adapting the system to a particular environment, and tuning it to improve performance. If some *major* changes or improvements are made to increase the functionality or performance, the system may undergo an evolution. The boundary between maintenance and evolution is fuzzy because what constitutes a major change is a subjective-opinion.

Maintenance absorbs a large fraction of the cost incurred during the software life cycle. A major portion of maintenance activity is a consequence of misinterpreted user requirements or faulty debugging during operation, which thereby introduces errors that did not exist earlier. Some of these maintenance problems could be reduced if more attention is paid to the development. If programmers have clearly understood the users' requirements, if they have documented the specification, design, and code properly, and if they have tested the system fully before its release, maintenance would not be so difficult and costly. To reduce maintenance costs, the software life cycle is divided into two fundamental phases—development and operation/maintenance. Software engineers should view these as distinct phases so that, both receive sufficient attention during the software life cycle.

20.2 Cost of Error Correction

Software development process includes analysis and generation of software requirements, software design, coding, final testing, and reliability modeling. Each one of these development phase includes verification, since it is easy to detect errors at each stage and also it will avoid error propagation from one stage to another. Further, it has been shown that the cost of correction of errors increases sharply as the development stage advances. The relative cost of correcting errors is 1% during the design phase, 3 during the coding phase, 21% during the testing phase and rises to 75% when the software is put into operation. (See Figure 20.3.)

The following are the different types of errors that may creep into the design of a software system:

- Incomplete or erroneous specification
- Intentional deviation from specification
- Violation of programming standards
- Erroneous data accessing
- Erroneous decision logic or sequencing

- Erroneous arithmetic computations
- Invalid timing
- Improper interrupt handling
- Wrong constants and data values
- Inaccurate documentation

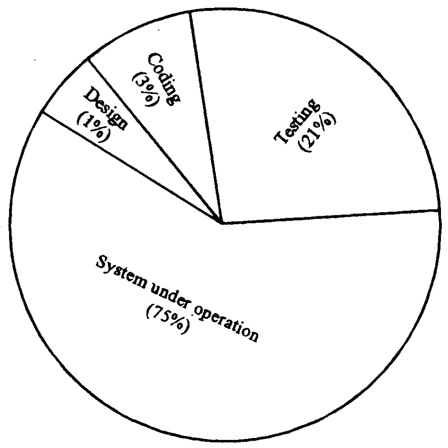

Figure 20.3: Cost of error correction Vs. development stage

20.3 Change Management

Changes to system are bound to happen many times either during the system design, or after complete implementation of the system, or during system operation. Hence, it is very essential to define change management process. The changes can be in the form of any modification to functionality during the design phase. It can also be due to any modification to agreed functionality or deliverable description in any phase. Some of the factors causing changes in a project are the following:

- Customer misunderstanding
- Inadequate specification
- New customer request
- Organization changes
- Government regulation

What is a Change ?

A change is an alteration to the project scope, deliverables, or milestones that would affect the project cost, schedule, or quality. Change is inevitable and occurs during the course of a project as shown in Figure 20.4. Once the implemented system becomes stable, many new requirements can be incorporated with minimal change to the design.The project manager is responsible for change control. Different categories of change exist: mandatory, critical, and nice to have. These changes must pass through proper channel and all documents must be updated. Before initiation of the change process, it must be first investigated and its impact on various factors must be thoroughly studied. The project manger can accept the change request, or reject the change request, or return the change request for further investigation or clarification. Once a change request is approved, it has to be incorporated appropriately at respective level or may even be carried out to all other phases. If it is improperly handled, it might even lead to the collapse of the whole system.

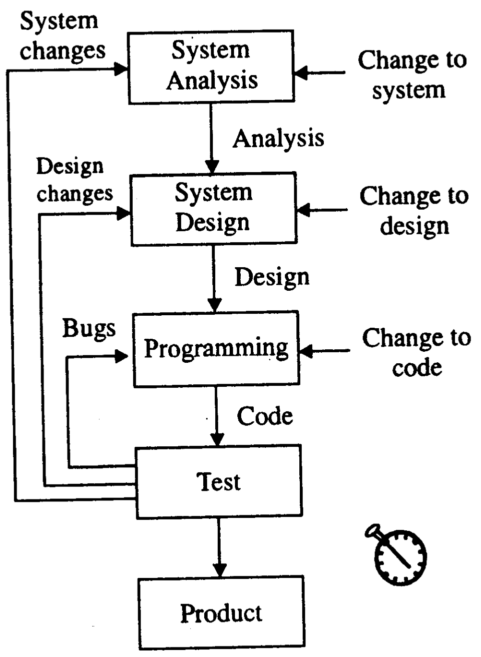

Figure 20.4: Change requests during system development

20.4 Reusable Components

Another important strategy that helps in reliable programming is to use all well proven, tested software modules without redesigning them. The usage of such well proven modules decreases the development effort and increases reliability. Though this idea is not very popular, except in scientific subroutines and some database applications, it is becoming increasingly acceptable to the software development community since the recent languages support the concept of modularization and separate compilation of those modules.

Some of the important components of reusability or levels of reusability are: code, data, design, specification, etc. The most popular level of reusability is code reusability.

Reusing Code

It can be in the form of making a call to subroutines library. Other forms of code reusability are the following:

- **Cut and paste of code**: In this method, the required portion of a code is cut and pasted in another module and necessary changes are incorporated.
- **Source-level includes**: In C++, it is performed by including the header file by using the include preprocessor directive.
- **Binary links**: Making a call to a function stored in the library in the form of executable code.
- **Runtime invocation**: In all the above three forms of source code reuse, while writing program itself the programmer has to know which component they wish to reuse. The binding of the reused components takes place at coding time, compile time, or link-time. In some cases, the flexibility of runtime

binding is essential. In C++, it is supported by virtual functions. The important point to be noted about the OO paradigm: the degree to which the OOPL supports dynamic binding may strongly influence the degree of reusability in the organization.

Reusing Data

Some of the data declared in a header file can be reused extensively by including that in a program. These can be in the form of macro constants, literals, enumerated constants, etc.

Reusing Designs

The major problem with code reuse is that coding takes place after major activity: analysis and design. It is well known that only 15 percent of project duration is used by coding phase, so any attempt to increase coding productivity (through high level languages) can have only a limited impact on overall project productivity.

Earlier major focus was given to source-level components. Today, focus is shifted to achieving significant results through reuse of the design and specification level. As pointed out by experts, code reuse typically occur at the bottom levels of a system design hierarchy whereas, design reuse occur in most of the branches of hierarchy (see Figure 20.5).

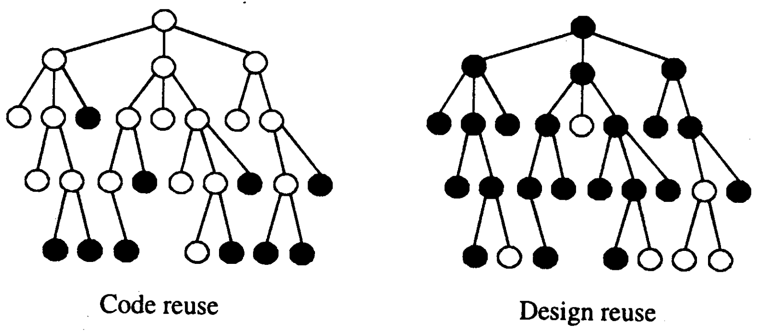

Figure 20.5: Code reuse Vs. Design reuse

Reusing Specification

Although design reuse is good, specification reuse is much better. It eliminates completely (almost) the effort needed in designing, coding, and testing an implementation of that specification.

Miscellaneous Reuse Components

While code, data, design, and specification are the most obvious candidates for reuse, they are not the only ones. Some of the possible candidates are:

- Cost-benefit calculations
- User documentation
- Feasibility studies
- Test cases, test procedures, test drivers, test stubs

Among all the entities involved in the software project, one component that cannot be reused is the people (who make up the project team). The experience, infrastructures, etc., gained by a project team during one project should be carried over, that is, reused in the next project whenever possible. This seem to be a common sense, but it is not common in software industry. It is because, teams are busted apart at the end of the project and individuals are scattered and reassigned to other projects or they might change organization (which is most common in software industry). Hence, peopleware approaches to software productivity often achieve results several times greater than technical approaches.

20.5 Software Life Cycle: Fountain-Flow Model

The conventional model requires a large amount of time to be spent in formulating the problem specifications. It delays the writing of code, and programmers may be impatient. In addition, the conventional life cycle model permits little feedback from the end user until the coding stage, which is at the end of the life cycle. At this point, if the system does not meet the specifications, the design and code becomes very expensive. It is often easier to change an existing system than to redevelop the specifications and design a new system. The conventional model fails to address software reusability, hence the existing software is not usable as a starting point.

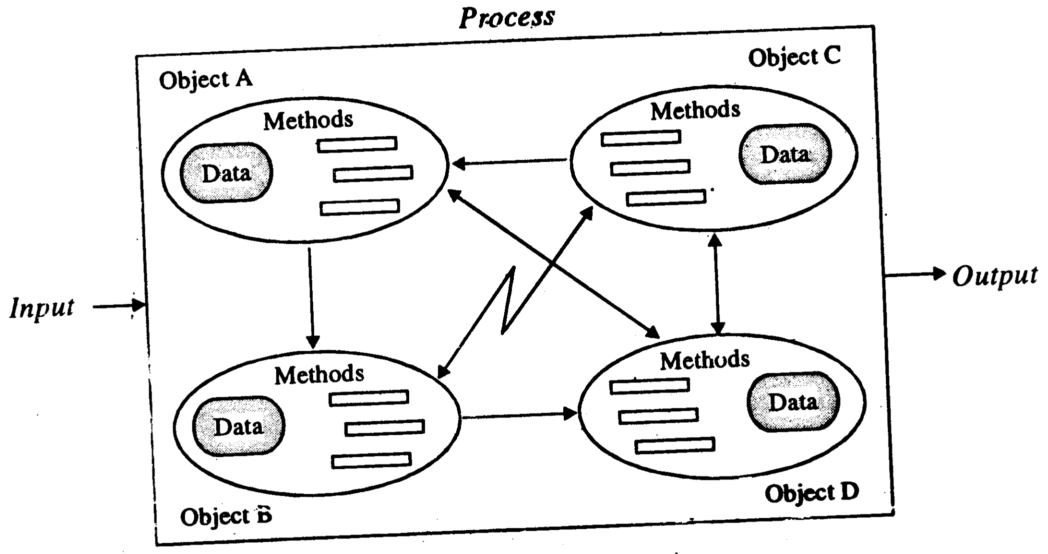

Figure 20.6: Objects interacting with each other

Objects are represented as individual entities which uniquely identify their own contents as well as the operations that may be performed on them. Thus, it is a philosophy of system design to decompose a problem into a set of abstract object types or resources in the system and a set of operations that manipulate instances in the system; and a set of operations that manipulate instances of each object type. The OO paradigm views a system as a collection of entities called objects that interact with each other to meet a specific objective. (See Figure 20.6).

OO methodology allows the end-users, analysts, designers, and programmers to view various components of the system in the same way, thus simplifying the process of mapping the customer requirement to the implementation model. The real-world entities are represented in the form of objects. Objects play the central role in all phases of the software development process. Therefore, there is a high degree of overlap and interaction among the phases. The use of *waterfall* model in the design of OO-based system does not allow overlap and interaction among the development phases. This problem can be circumvented by using a model that resembles a *fountain*. The resultant model is called *fountain* *flow* model and is shown in Figure 20.7. It allows a higher level phase to interact with its lower phase and again proceed to a higher level phase.

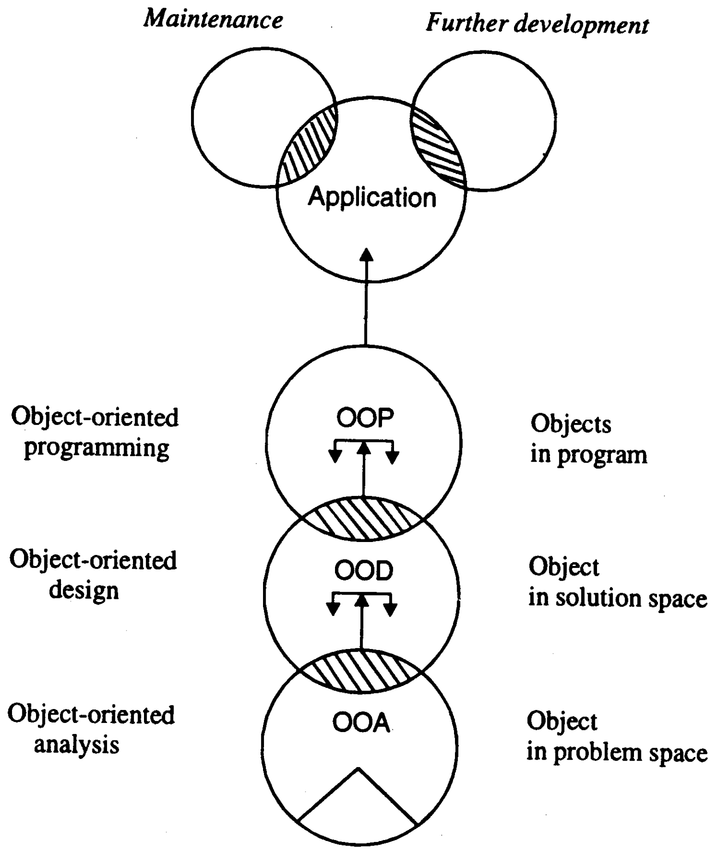

Figure 20.7: Fountain-flow model for OO system development

20.6 Object-Oriented Notations

Graphical notations play a major role while representing the design and development processes, and object-oriented design is no exception. They increase the ease with which ideas can be exchanged among the members of a project team. Object-oriented design requires notations for representing classes, objects, derived classes and their interrelationship, and interactions among objects. Unfortunately, for representing these aspects, there are no standard notations. In this book, authors have used their own notations and in addition to some of the commonly used notations, which are discussed in earlier chapters such as *Object Oriented Paradigm, Classes and Objects, Inheritance*, etc.

20.7 Object-Oriented Methodologies

Many object-oriented analysis (OOA) and object-oriented design (OOD) methodologies have emerged recently, although the concepts underlying object-orientation as a programming discipline has been developed long time ago. Object orientation certainly encompasses many novel concepts, and is popularly called as a new paradigm for software development. Object-oriented methodologies represent a radical change over conventional methodologies such as structured analysis.

Various object-oriented methodologies can be best investigated by dividing them into two camps—revolutionaries and synthesists. Revolutionaries believe that object-orientation is a radical change that renders conventional methodologies and ways of thinking (about design) obsolete. Synthesists, by contrast, view object-orientation as simply an accumulation of sound software engineering principles which adopters can graft onto their existing methodologies with relative ease.

The revolutionaries (Booch, Coad, Yourdon) state the following:

- There should be no doubt that object-oriented design is fundamentally different from traditional structured design approaches, it requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture.
- There is no doubt that one could arrive at the same results using different methods; but it is revealed from experience that the thinking process, the discovery process, and the communication between the user and analyst are fundamentally different with OOA than with structured analysis.

On the other side the synthesists (Wasserman, Pircher, Muller, Page Jones, and Weiss) state the following:

- Object-oriented structured design (OOSD) methodology is essentially an elaboration of structured design. They state that *the foundation of OOSD is structured design*, and that structured design *includes most of the necessary concepts and notations* for OOSD.
- The problems that object orientation has been widely touted as a revolutionary approach is a complete break with the past. This would be fascinating if it were true, but it is not like most engineering developments, the object oriented approach is a refinement of some of the best software engineering ideas of the past.

The leading analysis methodologies are the following:

- DeMarco structured analysis
- Yourdon modern structured analysis
- Martin information engineering analysis
- Bailin object-oriented requirements specification
- Coad and Yourdon object-oriented analysis
- Shlaer and Mellor object-oriented analysis

The leading design methodologies are the following:

- Yourdon and Constantine structured design
- Martin information engineering design
- Wasserman et al. object-oriented structured design
- Booch object-oriented design
- Wirfs-Brock et al. responsibility-driven design

Object-Oriented Analysis

Object-oriented analysis provides a simple, yet powerful mechanism for identifying objects, the building blocks of the software to be developed. It is mainly concerned with the decomposition of a problem into component parts and establishing a logical model to describe the system. The various steps involved in OOA are shown in Figure 20.8.

The two general findings about object-oriented analysis are:

1. OOA fulfills the properties of analysis, and
2. OOA has a smooth transition to design

OOA model should cover objectives, application domain knowledge, requirements of the environments, and requirements of the computer system.

- **Objectives**: These are the ultimate expectations of the users towards the entire information system (both computerized and manual). i.e., the objectives which are to be fulfilled through the interplay between the computer system and the surrounding human organization.
- **Application domain knowledge**: This defines the vocabulary of the application, its meaning, and properties.
- **Requirements of the environment**: This is a description of the behavior required from the human organization to meet the objectives.
- **Requirements of the computer system**: This is a description of the behavior required from the computer system to meet the objectives.

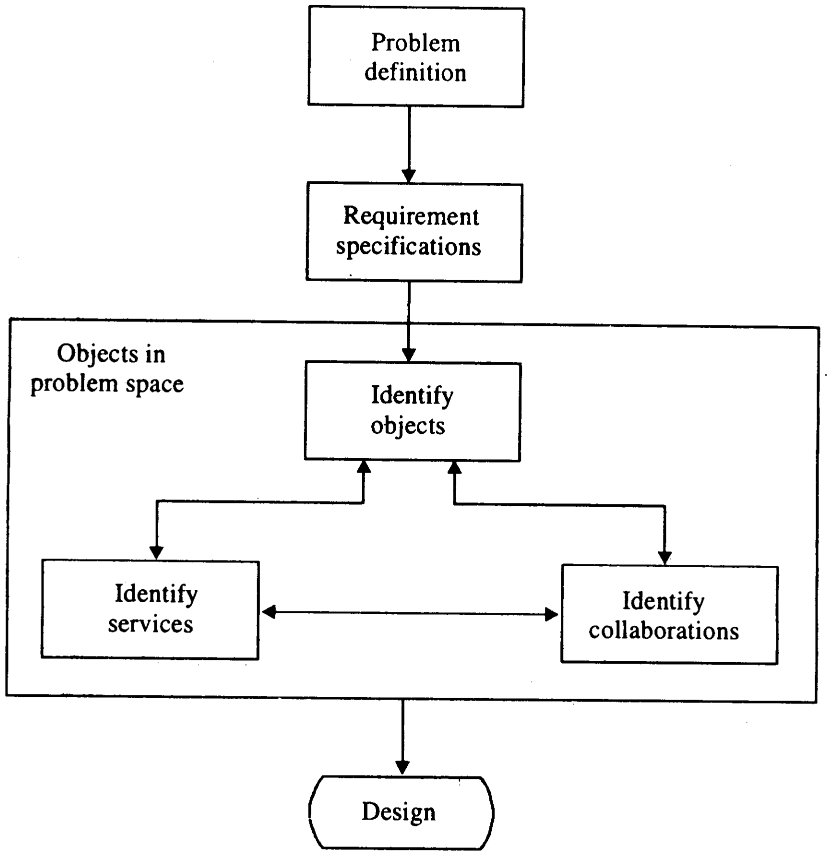

Figure 20.8: Steps in object-oriented analysis

For most OOA/OOD approaches, the difference between analysis and design is not recognized as the difference between the user requirement and the solution, but simply as the difference between "what" and "how". It is interpreted as "Analysis is aimed at describing what a target system is supposed to do to obtain an agreement with a customer bearing the expenses. While design is aimed at describing how the designed system will work...".

Positive Trends in OOA

OOA has evolved and focuses on system dynamics. Novel features of this method include:

1. It does not assume that a previously written requirement specification exists.
2. It focuses on the analysis content, including goals and objectives.
3. It considers external objects as initiators of the scenario.
4. Attention to requirement elicitation is given by creating scenarios from a structured interview process.
5. Symbolic execution can be obtained, because scripts and state transition are coupled through pre- and post- condition.

Object-Oriented Design

Object-oriented design is a radical change from both process-oriented and data-oriented methods. The OOD methodologies collectively model several important dimensions of a target system not addressed by conventional methodologies. These dimensions relate to the detailed definition of classes and inheritance, class and object relationships, encapsulated operations, and message connections. The need for adopters to acquire new competencies related to these dimensions, combined with Booch's uncontested observation that OOD uses a completely different structuring principle (based on object-oriented rather than function-oriented decomposition of system components), renders OOD as a radical change.

Object-oriented design is concerned with mapping of objects in the problem space into objects in the solution space. It creates overall architectural model and computational model of the system. In OOD, structure of the complete system is built using bottom-up approach whereas, class member functions are designed using top-down functional decomposition. It is important to construct structured hierarchies, identify abstract base classes, and simply the inter-object communication. Reusability of classes from previous design using inheritance principle, classification of objects (grouping) into subsystems providing specialized services, and determination of appropriate protocols are some of the considerations of the design stage.

Most of the object-oriented methodologies emphasize the following steps:

1. Review of objects created in the analysis phase.
2. Specification of class dependencies
3. Organization of class hierarchies using inheritance principles.
4. Design of classes.
5. Design of member functions.
6. Design of driver program.

20.8 Coad and Yourdon Object-Oriented Analysis

Coad and Yourdon OOA methodology can be viewed as *building upon the best concepts from information modeling, object-oriented programming languages, and knowledge-based systems*. OOA results in a five-layer model of the problem domain, where each layer builds on the previous layers. The layered model is constructed using a five-step procedure.

- Define objects and classes. Look for structures, other systems, devices, events, roles, operational procedures, sites and organizational units.
- Define structures. Look for relationships between classes and represent them as either general-to-specific structures (for example, employee-to-sales manager) or whole-to-part structures (for example car-to-engine).
- Define subject areas. Examine top-level objects within whole-to-part hierarchies and mark these as candidate subject areas. Refine subject areas to minimize interdependencies between subjects.
- Define attributes. Identify the atomic characteristics of object as attributes of the object. Also look for associative relationships between objects and determine the cardinality of those relationships.
- Define services. For each class and object, identify all the services it performs, either on its own behalf or for the benefit of other classes and objects.

The primary tools for Coad and Yourdon OOA are class and object diagrams and service charts. The class and object diagram has five levels, which are built incrementally during each of the five analysis steps outlined above. Service charts, which are *much similar to a (traditional) flow chart*, are used during the service definition phase to represent the internal logic of services. In addition, service charts portray state-dependent behavior such as preconditions and triggers (operations that are activated by the occurrence of a predefined event).

20.9 Booch's Object-Oriented Design

While there are many object-oriented design methodologies, one approach that reflects the essential features of object-oriented design is presented by Grady Booch. The four major steps involved in the object-oriented design (OOD) process are:

1. Identification of Classes (and Objects)
2. Identification of Semantics of Classes (and Objects)
3. Identification of Relationship between Classes (and Objects)
4. Implementation of Classes (and Objects)

Identification of Classes (and Objects)

In this step, key abstractions in the problem space are identified and labeled as potential candidates for classes and objects.

Identification of Semantics of Classes (and Objects)

In this step, the meanings of classes and objects identified in the previous step are established, which includes definition of the life cycles of each object from creation to destruction.

Identification of Relationship between Classes (and Objects)

In this step, interactions between classes and objects, such as, patterns of inheritance among classes and patterns of visibility among objects and classes (what classes and objects should be able to "see" each other) are identified.

Implementation of Classes (and Objects)

In this step, detailed internal views are constructed, including definition of methods and their behaviors. Objects and classes have to be allocated to modules (as defined in the target language environment) and resulting programs to processor (where the target environment supports multiple processors).

The primary tools used during OOD are:

- class diagrams and class templates (which emphasize class definitions and inheritance relationships)
- object diagrams and timing diagrams (which stress message definitions, visibility, and threads of control)
- state-transition diagrams (to model object states and transitions)
- operation templates (to capture definitions of services)
- module diagrams and templates (to capture physical design decisions about the assignment of objects and classes to modules)
- process diagrams and templates (to assign modules to processors in situation where a multiprocessor configuration is supported)

20.10 Class Design

Whether the design methodology chosen is Booch's OOD or any of the several other methodologies, design of classes is consistently declared to be central to the OO paradigm. Note that class design has the highest priority in OOD, and since it deals with the functional requirements of the system, it must occur before system design (mapping objects to processors/processes) and program design (reconciling of functionality using the target languages and tools etc.). Classes are developed either for building applications or for building class libraries or hierarchies. The class hierarchy is built by combining data hierarchy and procedure hierarchy as shown in Figure 20.9.

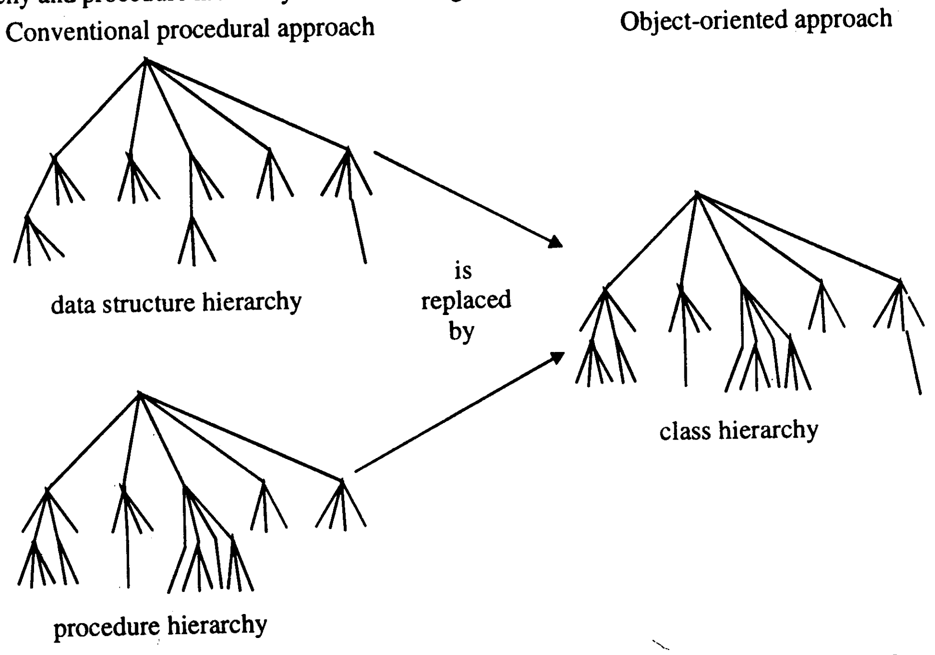

Figure 20.9: Class hierarchy combining data and procedure hierarchy

The output of the analysis phase must be transformed into a set of abstract class designs. Class design methods arrive at internal representational and algorithmic specifications that meet the declarative constraints of analysis models. The various steps involved in class development are shown in Figure 20.10. It includes class requirements, class design, testing, debugging, and finally ends with class certification. The various OOA/OOD methodologies discussed earlier have emphasized on class development.

◆ Move assumptions from a method to its callers

For example, a method might validate that the appropriated semaphore is locked by the current thread before modifying a shared resource protected by that semaphore. Instead, if all callers lock the semaphore before the call, it would no doubt be efficient, but also more dangerous and less general, to explicitly move the assumptions of lock ownership to the callers. This category of change tends to remove code from a method, and proportionally increase the number of warnings in the commentary describing the assumption made by the method.

◆ Move code from callers of a method into the method

The objective here, is to move the context of a call from the caller into the method. For example, if a caller is looping through hundreds of page table addresses in order to convert disk request to disk sector addresses, the conversion method can be augmented with a fatter interface that passes a collection of addresses, and the loop can be moved into the method. This is important in methods with protocol considerations such as lock ownership.

◆ Object pools

This technique minimizes calls to constructors in a manner analogous to memory pools minimizing the calls to operator new. The key is to reuse objects rather than constructing new ones. For example, if 80% of the fields in a page-fault object are the same for most page faults, it is possible to avoid the overhead by preconstructing page-fault objects, and adjusting the object's state via a method rather than initializing all the fields using a constructor. This is a special case of avoiding data movement.

◆ Caches

Instruction counts could sometimes be reduced by introducing caches. Note that the implied increase in data size can produce more page-faults, however, there were no tools available to predict the correlation. This issue is still being investigated.

◆ Dead code removed

Implicit C++ constructor and destructor calls provide a new variant of dead code removal. In some cases, the previous changes made to the local objects are superfluous. Removing these local objects can avoid wasting instructions. In some case, removing these has saved over 1,000 instructions along a critical path.

◆ Inlining

A function is expanded inline when the compiler replaces a traditional CALL instruction with code contained in the body of the function. In addition to eliminating the cost of setting up the stack frame, the optimizer can procedurally integrate the called function body into the caller's code by performing traditional optimization techniques across the call boundary by using techniques such as register liveness, constant propagation, and loop invariant code motion.

20.13 Software Project Management

Software project management is a complex undertaking. It requires project managers who are competent technical specialists and have some level of understanding and appreciation for the management principles as computer professionals. Knowing how to manage large projects is a critical skill for the computer professional. Many projects in the computer industry have failed to achieve their objectives due to lack of managerial skills. Consider the following circumstances:

- Project objectives are poorly defined and/or understood, even by members of the project team.
- Project deadlines are dictated by external events or imposed arbitrarily by administrators.
- Project budgets are based on naive estimates given by inexperienced managers.
- Project staffing is determined more by availability than ability.

The outcome of projects launched under such circumstances is easily predicted. Managing a well-planned and well-staffed project is challenging; with fuzzy objectives, unrealistic schedules, inadequate budget, and weak staffing, project managers would need a miracle to succeed.

Guidelines for Launching a Project

Every project is unique in its management requirements, but certain steps can be taken at the time the project is launched to improve the prospects. The following guidelines are offered for managing the project well:

- Establish a realistic project objective, setting forth in detail what will be accomplished if the project is successful.
- Appoint a competent project manager whose administrative, technical, and political skills commensurate with the task.
- Set up the project organization at an appropriate level and establish the appropriate communications links among all the elements of the organization that must play a role in the project's success.
- Staff the project with the proper mix of technical and administrative skills. Avoid, whenever possible, part-time assignments so that the individuals who are working on the project can devote their full attention to it.
- Identify key project milestones which, when achieved, will demonstrate definitive progress toward the ultimate project objective.

  Note: This step, plus Steps 6-11 below, may require several iterations before a satisfactory plan, schedule, and budget can be developed and approved.
- Plan the project in detail, identifying all tasks that must be completed to reach each milestone.
- Assign each task to an individual or to a specific organization so that responsibility for its completion is unambiguous.
- Estimate the time required to complete each task. It is essential that the time estimate for each task be made by the individual or organization that bears the responsibility for completing it.
- Estimate the cost of completing each task (or groups of tasks); again, these estimates should be made by the responsible person.
- Produce a project schedule and time-phased budget (using critical-path or similar network techniques when the size of the project warrants).
- Distribute the plan, schedule, and budget to all concerned parties and confirm their "ownership" of the tasks assigned to them.
- Review the project schedule and budget regularly. At each review meeting, ask for reaffirmation of plans and schedules (for the forthcoming period). While managing a large and complex projects, carry out project reviews, take minutes to document key decisions and follow-up assignments.
- Update project plans and schedules after each review meeting and distribute them as noted previously.
- Manage the project!

Of course, no project management philosophy can guarantee the success of any project, no matter how noble its objectives are, or how diligently it is applied. It can, however, materially improve the prospects for success, provided all project participants accept the philosophy and it is administered in a consistent and disciplined manner.

20.14 Plan for OO Battle

After all the theory and discussions about object-oriented programming, success with OO (Object-Oriented Technology) requires a commitment, as well as a plan, for action. The software designers, who excited by the new technology, are often ready to make the commitment with no planning at all. Just to recall, *if you are not planning, you are planning for failure.* Here are a series of planning steps articulated by OO experts for the major management planning activities required for successful implementation of object-orientation:

◆ Obtain Initial Advice

It is necessary to have consultation with experienced OO consultant before embarking on the OO bandwagon, to take a decision on suitability of OO methodologies and its benefits. This must provide an insight into the key decision makers in the organization what steps are involved, how long it will take, how much it will cost, what benefits are likely to accrue, and what risks must be accepted.

◆ Obtain Management Commitment

This is a crucial issue and important for the success of the object-orientation in the organization than the technical features of OO technology or the choice of C++ over Smalltalk. If management is opposed to this, then it probably won't work-out.

◆ Conduct Pilot Projects

Similar to all new technologies, OO needs to be validated and demonstrated to the organization. This is usually demonstrated through the use of a pilot project. A pilot project should be medium-sized and within the context of the organization. It is known that the failure of a pilot project will not bankrupt the organization. A good pilot project should be staffed by enthusiastic volunteers who are well trained and well supported by expert consulting assistance. A final conclusion can be reached from the viability of the proposed new technology.

◆ Develop a Training Plan

Training for object-orientation is important before taking any initiative to switch over to OO development. It is necessary to train programmers, designers, system analysts, and project leaders. If the management cannot afford to train all of them at once, it can be done in multiple phases.

- Document Management Expectations
- Develop an OO Development Life Cycle
- Choose OOA/OOD/OOP/OOT methods
- Choose OOP Language and Compiler
- Choose OO Case Tools and Repository
- Identify OO Based Matrices
- Revise Software Development Plan

20.15 A Final Word

The activities summarized in this chapter and C++ programming issues discussed in the earlier chapters can be mastered only with hands on experience. OO is surely not suitable for managing small projects and it may appear to be very costly. OO methodology has born to stay and is all set to win. It will surely help in long term and has impact right from the system study to the system maintenance and of course, even in training the end-users.

There are many optimistic and pessimistic views on adopting this new technology. The use of latest technology has played a very significant role in the success of several (world-class) organizations and even individuals. It is well known that "future belongs to those who use latest technology", and you might as well start now; delaying the decision by a day will just add one more day to a process that is bound to take several years. If you are worried that you are not the first one in industry (state, country, or world) to adopt OO, do not worry, you are not the last person. Perhaps the best advice (drawn from the *Proceeding of the National Conference on Computers in Education and Training*, India) on adopting new technology in the rapidly changing computer world is here:

"Our initial backwardness, our late arrival on the scene, and the small investments we made in the past need not remain as our handicaps but can be turned into our most valuable advantages if we make the right decisions now, order judicious investments and march forward with determination."

Review Questions

20.1 Compare the object-oriented computational model with the structured computational model.

20.2 Explain the water-flow model of software development.

20.3 Why does the cost of error correction increase as the development phase progresses ?

20.4 What are the issues to be considered while selecting a language for software implementation ?

20.5 What is change ? Explain how change management can be handled ?

20.6 What are the different reusable components ? Explain why code reusability occurs at the bottom of hierarchy and design reuse occurs in most of the branches of hierarchy ?

20.7 Explain the fountain-flow model of software development.

20.8 Draw object-orientated notations for class, object, inheritance, delegation, etc.

20.9 Investigate object-oriented methodologies as viewed by revolutionaries and synthesists.

20.10 Explain the steps involved in object-oriented analysis.

20.11 Explain the Coad and Yourdon object-oriented analysis method.

20.12 Explain the Booch object-oriented design method.

20.13 Compare the object-oriented and traditional analysis methodologies.

20.14 Compare the object-oriented and traditional design methodologies.

20.15 What is design for reuse ? Explain the steps involved in a class design.

20.16 What is a driver function ? What are its responsibilities ?

20.17 What are the steps involved in building a reliable code ?

20.18 State and explain the guidelines for tuning performance of an OO software.

20.19 What is the software project management ? State guidelines for launching a project.

20.20 What are the steps involved in the major management planning required for successful implementation of the object-oriented system ?

Appendix A:

C++ Keywords and Operators

C++ supports a wide variety of keywords and operators to support object-oriented programming. The following sections illustrates them with syntax, description, and examples.

asm, _asm, __asm: embed assembly statements

Syntax:

```
asm <opcode> <operands> <; or newline>
_asm <opcode> <operands> <; or newline>
__asm <opcode> <operands> <; or newline>
```

Description: It allows to embed assembly language statements in between C++ statements. These assembly language statements are machine dependent; portability of a program is lost when such statements are used.

Example:

```
asm mov ax, _stklen
asm add bx, cx
asm add bx, 10
```

Any C++ statement can be replaced by the appropriate assembly language equivalent statements. In order to include a number of `asm` statements, surround them with braces by using the following format:

```
asm {
      pop ax; pop ds
       iret
 }
```

auto: define variables

Syntax: `[auto] <data definition>;`

Description: It defines variables whose resources are released as soon as they go out of scope. All the local variables are `auto` by default and hence, `auto` storage class is rarely specified explicitly.

Example:

```
int main(int argc, char **argv)
{
   auto int i;
   i = 5;
   return i;
}
```

break: pass control out of the current loop

Syntax: `break;`

Description: It causes control to pass to the statement following the innermost enclosing `while`, `do`, `for`, or `switch` statement.

Example:

```
for( i = 0; i < n; i++ )
{
   ....
   if( wants_to_terminate_loop )
      break; // transfers control to the next statement outside loop
}
```

case: specify actions when the switch expression matches with it

Syntax: `case <constant expression>:`

where `<constant expression>` must be a unique integer constant value.

Description: The list of possible branch points within `switch <statement>` is determined by the matching `case` statement within the switch body. Once a value is computed for `<expression>`, the list of possible `<constant expression>` values determined from all `case` statements is searched for a match. If a match is found, execution continues after the matching `case` statement until a `break` statement is encountered or till the end of `switch` is reached.

Example:

```
switch( figure_type )       // figure_type is character variable
{
  case 'l':
       draw_line( x1, y1, x2, y2 );
       break; // transfers to next statement to switch
  case 'c':
       draw_circle( x, y, r );
       break; // transfers to next statement to switch
  ....
  default: // execute if none of the cases match with switch expression
       cout << "invalid figure code";
       break; // it can be omitted
}
```

catch: capture exception thrown

Syntax: `catch( <exception-object> )`

Description: An exception thrown in the program is caught by the `catch` statement. It follows `try` statement and is responsible for taking corrective actions in response to an exception.

Example:

```
class div_by_zero  { };   // empty class
....
int div( int a, int b )
{
   if( b == 0 )
     throw div_by_zero(); // divide by zero error;
   return a/b;
}
....
try
{
   // read a and b value if necessary
   int c = div( a, b );
  // no exception... do other activities
}
catch( div_by_zerc )
{
   cout << "Divide by zero";
   // take necessary action
}
```

char: define character variables

Syntax: `char <var1>, .., <varn>;`

Description: It defines variable(s) of type character which is 1 byte in length. They can be signed (default) or unsigned.

Example: `char ch1, *name;`

class: encloses data and functions into a single unit

Syntax: `class <classname> [<:baselist>] { <member list> };`

- `<classname>` can be any identifier unique within its scope.
- `<baselist>` lists the base class(es) that this class derives from and it is optional.
- `<member list>` declares the class's data members and member functions.

Description: It declares C++ class which combines both the data and functions on those data into a single unit. Within a class, the data are called *data members* and the functions are called *member functions*.

Example:

```
class student        // declares class called student
{
    char *name;        // data member
    ....
    char *getname()    // member function
    {
       return name;
    }
};
```

const: define constant variable

It creates a constant variable and makes it a read-only variable.

Syntax:

```
const [data type] <variable name> [ = <value> ] ;
<function name> ( const <type> <variable name> )
```

Description: In the first version, the `const` modifier enables to assign an initial value to a variable that cannot be changed later by the program. It can be used to define constant variables of primitive and user-defined data types.

Example: `const int my_age = 25;`

Any assignments to `my_age` will result in a compiler error. Note that, a `const` variable can be indirectly modified by using a pointer as follows:

```
*(int *)&my_age = 35;
```

When the `const` modifier is used with a pointer parameter in a function's parameter list, the function cannot modify the variable that the pointer points to as follows:

```
double sqrt(const double a );
```

Here the `sqrt()` is prevented from modifying the input value passed through a variable.

`continue`: transfer control

Syntax: `continue;`

Description: It passes control to the end of the innermost enclosing `while`, do, or `for` statement, at which the loop continuation condition is evaluated.

Example:

```
for( i = 0; i < 20; i++ )
{
   if(array[i] == 0)
      continue;  // skips this iteration
   array[i] = 1/array[i];
}
```

`default`: default operation when all cases fail

Syntax: `default:`

Description: In a `switch` statement, if a case-match is not found and the `default:` prefix is found within the switch body, control is transferred to that point, otherwise, the switch body is skipped entirely.

Example: (see case)

`delete`: deallocate memory

Syntax: `delete <pointer_to_name>;`

Description: It destroys an object by releasing all the resources allocated to it by the `new` operator. The `delete` operator destroys the object <name> by deallocating `sizeof( <name> )` bytes (pointed to by `<pointer_to_name>`). The storage duration of the new object is from the point of creation until the operator `delete` deallocates its memory, or until the end of the program.

Example:

```
int *p;              // pointer to integer
....
p = new int[100];  // allocate memory for 100 integer elements
....
delete p;        // deallocate memory allocated to p using new operator
```

do: do..while loop

Syntax: `do <statement> while (expression>;`

Description: The `<statement>` enclosed within the body of a loop is executed repeatedly as long as the value of `<expression>` remains nonzero. Irrespective of the value of a `<expression>`, this loop executes its body atleast once.

Example:

```
i = 1; factorial = 1;
do
{
   factorial *= i;
   i++;
} while (i <= n );
```

double: define double precision real variable

Syntax: `double <var1>, ...<varn>;`

Description: It defines variables of type real type which is 8 bytes in length. Use of `double` or `float` requires linking in the floating-point math package if numeric coprocessor does not exist in the system. Most of the compilers include math package automatically if floating point numbers are used in a program.

Example: `double a, b;   // a and b are double type variables`

else: actions when the if condition fails

Syntax:

```
if( condition )
   statement1;              // if condition is true
else
   statement2;       // if condition is false
```

Description: It specifies the alternate statement to be executed when the if condition fails

Example:

```
if( boy_age > girl_age )
   cout << "boy is elder than girl";
else
   cout << "girl is elder than boy";
```

enum: declare enumerated constants

Syntax: `enum [<type_tag>] {<constant_name> [= <value>], ...} [var_list];`

Description: It declares a set of constants of type `int`. A `<type_tag>` is an optional and is used to name the set. `<constant_name>` is the name of a constant that can optionally be assigned the value of `<value>`. Note that, `<value>` must be an integer. If `<value>` is missing, it is assumed to be `<prev> + 1` where `<prev>` is the value of the previous integer constant in the list. For the first integer constant in the list, the default value is 0. `<var_list>` is an optional variable list that can follow the type declaration. It assigns variables to the `enum` type.

Example: `enum modes { LASTMODE = -1, BW40 = 0, C40, BW80, C80, MONO = 7 };`

In the above declaration, `modes` is the type tag, `LASTMODE,BW40,C40`, etc. are the enumerated constant names. The value of C40 is `1 (BW40 + 1)` and `BW80 = 2 (C40 + 1)`, etc.

extern: specify variable/function type which is defined elsewhere

Syntax:
```
extern <data definition>;
extern <function prototype>;
```

Description: It declares variables/functions and indicates that the actual storage and initial value of a variable or the body of a function, is defined elsewhere, usually in a separate source code module. The keyword `extern` is optional for a function prototype.

The `extern` variables cannot be initialized at the point of declaration and if they are not defined a linker error *Undefined symbol 'symbol-name' in module 'module-name'* is generated.

Example:
```
extern int _fmode;
extern void factorial(int n);
```

float: define float variables

Syntax: `float <var1>, ...<varn>;`

Description: It defines variables of `float` data type, which are 4 bytes in length. Use of `double` or `float` requires linking in the floating-point math package. Most of the compilers including Borland C++ will do this automatically, if floating point numbers are used in a program.

Example: `float a, b;`

for: loop

Syntax: `for ( [<expr1>] ; [<expr2>] ; [<expr3>] ) <statement>`

Description: The `<statement>` enclosed with the body of a loop is executed repeatedly as long as the value of `<expr2>` remains nonzero. The `<statement>` is executed repeatedly until the value of `<expr2>` is 0. The `<expr1>` is evaluated before the first iteration and is usually used to initialize variables of the `for` loop. The `<expr2>` is evaluated before entering the loop statement. After each iteration of the loop, `<expr3>` is evaluated, and is usually used to increment a loop counter.

In C++, `<expr1>` can have an expression or variable definition. The scope of any identifier defined in `<expr1>` is extended to outside its loop and those defined within the loop body is limited to that loop iteration. All the expressions are optional. If `<expr2>` is left out, it is assumed to be 1.

Example:
```
for( i=0; i < 100; i++ )
   cout << "i = " << i << endl;
```

friend: allow other function/class to access private members of a class

Syntax: `friend <identifier>;`

Description: A friend of a class can be a function or a class. Friend function or friend class is allowed to access `private` or `protected` members of a class. A class which wants other class or function to be its friend, should explicitly declare it as its friend. Friend function or class cannot access members of a class to which it is a friend directly; it has to access them using class objects.

Example:

```
class stars
{
   private:
      int magnitude;
   ...
   friend galaxy;  // galaxy is friend class
};
class galaxy
{
   ...
   void func()
   {
       stars s1;
       s1.magnitude = 100;  // valid since galaxy is a friend of stars
       ...
    }
};
```

The above declaration states that, the class `galaxy` can access all the members of the class `stars` but not vice-versa.

goto: transfer control

Syntax: `goto <identifier> ;`

Description: It transfers control to the specified location. Control is unconditionally transferred to the location of a local label specified by `<identifier>`.

Example:

```
Again:
...
...
goto Again;
```

Note that Labels must be followed by a statement.

if: actions when the if condition succeeds

Syntax:

```
if( <expression> )  // if statement
   <statement1>;
if( <expression> )  // if-else statement
   <statement1>;
else
   <statement2>;
```

Description: It transfers control conditionally to a required statement based on the conditional result. If `<expression>` is nonzero when evaluated, `<statement1>` is executed. In the second case, `<statement2>` is executed if the `<expression>` is zero. An optional `else` can follow an `if` statement, but no statements can come between an `if` statement and an `else`; however, multiple statements can be enclosed within flower brackets.

Examples:

```
if(count < 50)
  count++;
if(x < y)
  small = x;
else
  small = y;
```

The `#if` and `#else` preprocessor statements (directives) look similar to the `if` and `else` statements, but have very different effects and their effect can be seen only at compile time. They decide which source file lines are to be compiled and which are to be ignored.

`inline`: substitute the function body at the point of call

Syntax:

```
inline <datatype> <function>(<parameters>) { <statements>; }
inline <datatype> <class>::<function> (<parameters>) { <statements>; }
```

Description: It declares/defines C++ inline functions. The compiler substitutes function call by the body of a function so that program execution speed increases. Member functions defined within the body of a class are treated as `inline` functions by default.

The first syntax declares an `inline` function by default. This syntax can be used to define normal functions or member functions as inline function. The second syntax declares an `inline` function explicitly and such definitions need not fall within the class definition.

`Inline` functions are best reserved for small, frequently used functions, and any normal function can also be made as `inline`.

Example:

```
// Implicit inline statement
int num;      // global num
class cat
{
   public:
    char* func(void) { return num; } // inline function implicitly
     char* num;
}
// Explicit inline statement
inline char* cat::func(void) { return num; }
```

Any C++ function can be declared `inline` as follows:

```
inline swap( int *a, int *b )
{
  // swap without using temporary variable
  *a = *a +  *b;
  *b = *a - *b;        // *b = (*a + *b) - *b = *a
  *a = *a - *b;        // *a = (*a + *b) - *a = *b
}
```

int: define integer variable

Syntax: `int <var1>, .., <varn>;`

Description: It defines variables of integer data type which is one word in length. They can be signed (default) or unsigned. It is represented by 2 bytes under 16-bit operating system (e.g., MS-DOS) and 16-bit compiler (Borland C++) and 4 bytes under 32-bit OS and compilers (e.g., Under UNIX).

Examples:

```
int  i, j;
long x;              // int is implied
signed int i;    // signed is default
unsigned long int l;. // int OK, not needed
```

new: allocate memory

Syntax:

```
<pointer_to_name> = new <name> [ count ];
<pointer_to_name> = new <name> ( init_value );
```

Description: The new operator creates an object <name> by allocating `sizeof(<name>)*count` bytes from the *heap*. The storage duration of the new object is from the point of creation until the operator `delete` deallocates its memory, or until the end of the program.

Example: (see delete)

```
int *iptr = new int[ 15 ];     // allocates 15 integer memory
int *a = new ( 10 );           // allocates a integer and assigns 10
```

operator: overload operator

Syntax:

```
operator <operator symbol>( <parameters> )
{
   <statements>;
}
```

Description: It allows to define a new action for the existing C++ overloadable operators to operate on user defined data types. The keyword `operator` followed by an operator symbol, defines a new (overloaded) action for the given operator.

Example:

```
complex operator +(complex c1, complex c2)
{
   return complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

private: specify class members access scope

Syntax:

```
private: <declarations>
```

Description: It explicitly declares members of a class to have private privilege. If a member is `private`, it can be accessed only by member functions or friends of the class. Members of a class are private by default unless otherwise specified explicitly.

Example:

```
class Abc
{
        int a;          // private by default
        ...
        public:
           int c;
           ...
        private:                // private by explicit
          int b;
          ...
        protected:              // protected by declaration
          int c;
          ...
     public:            // public by declaration
        ...
};
```

`protected`: specify class members access scope

Syntax: `protected: <declarations>`

Description: It explicitly declares members of a class to have protected privilege so that they are inheritable to derived classes similar to `public` members. They can either have `private` or `protected` status in derived classes depending on type of derivation. Note that protected members have the same privilege as private member except that they are inheritable.

Example: (see `private`)

`public`: members accessible to all users

Syntax: `public: <declarations>`

Description: It explicitly declares members of a class to have public privilege and they are accessible to all the users. If a member is `public`, it can be used by any function. In C++, members of a `struct` or `union` are `public` by default.

Example: (see `private`)

`register`: allocate a register for the variable

Syntax: `register <data definition>;`

Description: It informs the compiler to allocate a CPU register if possible for the variable to speedup data access.

Example: `register int i;`

`return`: transfer control to the caller

Syntax: `return [ <expression> ] ;`

Description: Returns control immediately from the currently executing function to the calling routine, optionally returning a value.

template: declare generic functions or classes

Syntax: `template < template-argument-list > declaration`

Description: It constructs a family of related functions or classes. It can be used to declare function templates and class templates.

Examples:

```
template < class T >
swap( T & a, T & b ) // function template
{
   T temp;
   temp = a;
   a = b;
   b = temp;
}
template <class T>   // class template
class myclass
{
   T a;
   ....
};
```

this: pointer to current object

Syntax:

```
this               // address of the class in which it is referenced
this->member  // access a member
```

Description: It is a predefined pointer variable within every class and points to the current object. `this` is passed as a hidden argument in all calls to non-static member functions. The keyword `this` is a local variable available in the body of any nonstatic member function. The keyword `this` does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references.

Example:

```
class date
{
   int day;
   ...
   void init()
   {
       this->day = 1; // same as day = 1
   }
};
```

If `x.func(y)` is called, where `y` is a member of `x`, the variable `this` is set to `&x` and `y` is set to `this->y`, which is equivalent to `x.y`.

throw: raise an exception

Syntax: `throw object;        // object of a class`

Description: It allows to raise an exception when an error is generated during computation. It normally raises exception using temporary object of a empty class.

Example: (see `catch`)

`try`: enclose a code raising an exception

Syntax:
```
try {
   .... // code raising exception
}
```

Description: A code raising an exception or exceptions must be enclosed within try-block. It indicates that the program is prepared to test for the existence of an exception if it occurs within the scope of the try-block. The catch-block following the try-block will actually take appropriate action for all those exceptions raised.

Example: (see `catch`)

`typedef`: enhance existing data type

Syntax: `typedef <type definition> <identifier>;`

Description: It assigns the symbol name `<identifier>` to the data type definition `<type definition>`. It helps in declaring a convenient name for the existing data type and thus simplifies representation of complicated statements.

Examples:
```
typedef unsigned char byte;  // a new data type called byte is created
typedef struct
{
   double re, im;
} complex;
typedef int * array_t;    // array_t p; is same as int *p;
```

The definition such as
```
byte a, b;
```
is actually treated as
```
unsigned char a, b;
```

`union`: all members share the same memory

Syntax:
```
union [<union type name>]
{
    <type> <variable names> ;
    ...
} [<union variables>] ;
```

Description: It is similar to a `struct`, except that its members share the same storage space.

Example:
```
union int_or_long
{
   int i;
```

```
        long l;
    } a_number;
```

The compiler will allocate enough storage in `a_number` to accommodate the largest element in the `union`. Unlike a `struct`, the variables `a_number.i` and `a_number.l` occupy the same location in memory. Thus, writing into one, will overwrite the other. Elements of a `union` are accessed in the same manner as a `struct`.

virtual: declares virtual function or class

Syntax:

```
class classname
{
    ....
    virtual int myfunc()=0;
  };
```

Description: It can be used to make a function or class virtual. *Virtual function* allows derived classes to provide different versions of a base class function, which is declared as virtual function. *Virtual class* allows to inherit only one copy of a base class indirectly from more than one immediate base classes.

Examples:

Virtual function:

```
class figure
{
       virtual void draw()  = 0; // definition in derived class
};
class line: public figure
{
   ...
   draw()      // implements virtual function declared in base class
   {
      // draw line
   }
};
....
figure *fig;   // can point to its derived class objects also
line l1;
....
fig = &l1;
fig->draw(); // invoke draw() defined in the class line
```

Virtual class:

```
class B { ...};
class D : B, B { ... };  // illegal
```

However, a base class can be indirectly passed to the derived class more than once:

```
class X : public B { ... };
class Y : public B { ... };
class Z : public X, public Y { ... };  // Error
```

In this case, each object of class `Z` will have two sub-objects of `class B`.

If this causes problems, the keyword `virtual` can be added to a base class specifier. For Example,

```
class X : virtual public B { ... };
class Y : virtual public B { ... };
class Z : public X, public Y { ... };
```

B is now a virtual base class, and class Z has only one sub-object of the class B.

void: empty data type

Syntax: `void var1, var2, ..., varn;`
`void funcname(..);`

Description: It can be used to define variables or declare functions which return nothing. When used as a function return type, `void` means that the function does not return a value.

Example:

The function definition returning no data to a caller is as follows:

```
void hello(char *name)
{
   cout << "Hello, " << name;
}
```

The function that does not take any parameters is indicated by `void`, for instance, `int init(void)`

Void pointers cannot be dereferenced without explicit type casting. This is because the compiler cannot determine the size of the object the pointer points to. For Example,

```
int x;  float r;
void *p = &x;           /* p points to x */
int main (void)
{
   *(int *) p = 2;
   p = &r;              /* p points to r */
   *(float *)p = 1.1;
}
```

volatile: update memory when the variable is assigned to register

Syntax: `volatile <data definition> ;`

Description: It indicates that a variable can be changed by a background routine. Every reference to the variable will reload the contents from memory rather than take advantage of situations where a register is allocated to the variable for efficiency purpose. Note that, C++ allows `volatile` to be applied to objects.

Example: `volatile int i;`

while: while loop, repeats execution

Syntax: `while ( <expression> ) <statement>`

Description: The `<statement>` is executed repeatedly as long as the value of `<expression>` remains nonzero. The test takes place before each execution of the `<statement>`.

Example:

```
i = 1; factorial = 1;
```

```
while( i <= n )
{
   factorial *= i;
   i++;
}
```

C++ Operators

Some of the operators such as new, delete, etc. have been discussed in the previous section. In addition to them, C supports many other operators which are summarized in Table A.1. Every operator has *precedence* and *associativity* associated with them. Precedence specifies the operator to be evaluated first when an expression is of type mixed-mode, whereas, associativity specifies the order in which operands associated with each operator are to be evaluated.

Operator Summary		
`::`	Scope resolution	`ClassName :: member`
`::`	global	`:: name`
	member selection	`object . member`
`->`	member selection	`pointer -> member`
`[ ]`	subscripting	`pointer [expr]`
`( )`	function call	`expr (expr_list)`
`( )`	value construction	`type (expr_list)`
`++`	post increment	`lvalue ++`
`--`	post decrement	`lvalue --`
`sizeof`	Size of object	`size of expr`
`sizeof`	size of type	`sizeof (type)`
`++`	pre increment	`++ lvalue`
`--`	pre decrement	`-- lvalue`
`~`	complement	`~ exor`
`!`	not	`! expr`
`-`	unary minus	`- expr`
		`- expr`
`+`	unary plus	`+ expr`
`&`	address of	`& lvalue`
`*`	dereference	`* expr`
`new`	create (allocate)	`new type`
`delete`	destory (de-allocate)	`delete pointer`
`delete []`	destroy array	`delete [] pointer`
`()`	cast (type conversion)	`(type) expr`
`.*`	member selection	`object .* pointer-to-member`
`->*`	member selection	`pointer ->* pointer-t0-member`
`*`	multiply	`expr * expr`
`/`	divide	`expr / expr`
`%`	modulo (remainder)	`expr % expr`
`+`	add (plus)	`expr + expr`
`-`	subtract (minus)	`expr - expr`

Table A.1: C++ operators

(*Continued*)

Operator Summary (*Continued*)		

<<
>>
	shift left
	shift right
		`expr << expr`
		`expr >> expr`

<
<=
>
>=
	less than
	less than or equal
	greater than
	greater than or equal
		`expr < expr`
		`expr <= expr`
		`expr > expr`
		`expr >= expr`

==
!=
	equal
	not equal
		`expr == expr`
		`expr != expr`

&	bitwise AND	`expr & expr`
^	bitwise exclusive OR	`expr ^ expr`
|	bitwise inclusive OR	`expr | expr`
&&	logical AND	`expr && expr`
||	logical inclusive OR	`expr || expr`
? :	conditional expression	`expr ? expr : expr`

=
*=
/=
%=
+=
-=
<<=
>>=
&=
|=
^=
	simple assignment
	multiply and assign
	divide and assign
	modulo and assign
	add and assign
	subtract and assign
	shift left and assign
	AND and assign
	inclusive OR and assign
	exclusive OR and assign
		`lvalue = expr`
		`lvalue *= expr`
		`lvalue /= expr`
		`lvalue %= expr`
		`lvalue += expr`
		`lvalue -= expr`
		`lvalue <<= expr`
		`lvalue >>= expr`
		`lvalue &= expr`
		`lvalue |= expr`
		`lvalue ^= expr`

`throw`	throw exception	`throw expr`
	comma (sequencing)	`expr    expr`

Table A.1: C++ operators

Appendix B: C++ Library Functions

Function	Description	Include File
`abort()`	abnormally terminates a program	`stdlib.h`
`abs()`	returns the absolute value of an integer	`math.h`
`acos()`	calculates the arc cosine	`math.h`
`asctime()`	converts date and time to ASCII	`time.h`
`asin()`	calculates the arc sine	`math.h`
`assert()`	tests a condition and possibly aborts	`assert.h`
`atan()`	calculates the arc tangent	`math.h`
`atan2()`	calculates the arc tangent of y/x	`math.h`
`atexit()`	registers an exit function	`stdlib.h`
`atof()`	converts a string to a floating-point number	`math.h`
`atoi()`	converts a string to an integer	`stdlib.h`
`atol()`	converts a string to a long integer	`stdlib.h`
`bsearch()`	binary search of an array	`stdlib.h`
`calloc()`	allocates main memory	`stdlib.h`
`ceil()`	rounds up	`math.h`
`clearerr()`	resets error indication	`stdio.h`
`clock()`	determines processor time	`time.h`
`cos()`	calculates the cosine of a value	`math.h`
`cosh()`	calculates the hyperbolic cosine of a value	`math.h`
`ctime()`	converts date and time to a string	`time.h`
`exit()`	terminates program	`stdlib.h`
`fabs()`	returns the absolute value of a floating-point number	`math.h`
`fclose()`	closes a stream	`stdio.h`
`feof()`	detects end-of-file on a stream	`stdio.h`
`ferror()`	detects errors in a stream	`stdio.h`
`fflush()`	flushes a stream	`stdio.h`
`fgetc()`	gets character from stream	`stdio.h`
`fgetpos()`	gets the current file pointer	`stdio.h`
`fgets()`	gets a string from a stream	`stdio.h`
`floor()`	rounds down	`math.h`
`fmod()`	calculates x modulo y, the remainder of x/y	`math.h`
`fopen()`	opens a stream	`stdio.h`
`fprintf()`	writes formatted output to a stream	`stdio.h`
`fputc()`	puts a character on a stream	`stdio.h`
`fputs()`	outputs a string on a stream	`stdio.h`
`fread()`	reads data from a stream	`stdio.h`
`free()`	free allocated block	`alloc.h`
`freopen()`	associates a new file with an open stream	`stdio.h`
`frexp()`	splits a double number into its mantissa and exponent	`math.h`
`fscanf()`	scans and formats input from a stream	`stdio.h`
`fseek()`	repositions a file pointer on a stream	`stdio.h`
`fsetpos()`	positions the file pointer of a stream	`stdio.h`
`fstat()`	gets file statistics	`sys\stat.h`
`ftell()`	returns the current file pointer	`stdio.h`
`fwrite()`	writes to a stream	`stdio.h`
`getc()`	gets a character from a stream	`stdio.h`
`getchar()`	gets a character from stdin	`stdio.h`
`gets()`	gets a string from stdin	`stdio.h`
`isalnum()`	character classification macro	`ctype.h`
`isalpha()`	character classification macro	`ctype.h`
`isascii()`	character classification macro	`ctype.h`
`iscntrl()`	character classification macro	`ctype.h`
`isdigit()`	character classification macro	`ctype.h`
`isgraph()`	character classification macro	`ctype.h`
`islower()`	character classification macro	`ctype.h`
`isprint()`	character classification macro	`ctype.h`
`ispunct()`	character classification macro	`ctype.h`
`isspace()`	character classification macro	`ctype.h`
`isupper()`	character classification macro	`ctype.h`
`isxdigit()`	character classification macro	`ctype.h`

Function	Description	Include File
labs()	gives long absolute value	math.h
ldexp()	calculates x * 2^{exp}	math.h
ldiv()	divides two longs, returning quotient and remainder	stdlib.h
log()	calculates the natural logarithm of x	math.h
log10()	calculates $\log_{10}(x)$	math.h
malloc()	allocates main memory	stdlib.h
memchr()	searches n bytes for character c	mem.h
memcmp()	compares two blocks upto a length of exactly n bytes	mem.h
memcpy()	copies a block of n bytes	mem.h
memset()	set n bytes of a block of memory to byte c	mem.h
mktime()	converts time to calendar format	time.h
perror()	prints a system error message	stdio.h
pow()	calculates x to the power of y	math.h
printf()	writes formatted output to stdout	stdio.h
putc()	outputs a character to a stream	stdio.h
putchar()	outputs character on stdout	stdio.h
puts()	outputs a string to stdout	stdio.h
raise()	sends a software signal to the executing program	signal.h
rand()	random number generator	stdlib.h
realloc()	reallocates main memory	stdlib.h
remove()	removes a file	stdio.h
rename()	renames a file	stdio.h
rmdir()	remove a file directory	dir.h
scanf()	scans and formats input from the stdin stream	stdio.h
setbuf()	assigns a buffer to a stream	stdio.h
setjmp()	set up for non-local goto	setjmp.h
signal()	specifies signal-handling actions	signal.h
sin()	calculates sine	math.h
sinh()	calculates hyperbolic sine	math.h
sprintf()	writes formatted ouput to a string	stdio.h
srand()	initializes random number generator	stdlib.h
sscanf()	scans and formats input from a string	stdio.h
stat()	gets information about a file	sys\stat.h
strcat()	appends one string to another	string.h
strchr()	scans a string for the first occurrence of a given character	string.h
strcmp()	compares one string with another	string.h
strcoll()	compares two strings	string.h
strcpy()	copies one string into another	string.h
strcspn()	scans a string for initial segment	string.h
strdup()	copies a string into a newly created location	string.h
strerror()	returns a pointer to an error message string	string.h
strftime()	formats time for output	time.h
strlen()	calculates the length of a string	string.h
strncat()	appends a portion of one string to another	string.h
strncmp()	compares a portion of one string to a portion of another	string.h
strncpy()	copies a given number of bytes from one string into another	stdio.h
strrchr()	scans a string for the last occurrence of a given character	string.h
strspn()	scans a string for the first segment (a subset of a given string)	string.h
strstr()	scans a string for the occurrence of a given substring	string.h
strtod()	converts a string to a double value	stdlib.h
strtok()	searches one string for tokens	string.h
strtoul()	converts a string to an unsigned long in the given radix	stdlib.h
strxfrm()	transforms a portion of a string	string.h
system()	invokes the shell in order to execute a command	stdlib.h
tanh()	calculates the hyperbolic tangent	math.h
time()	gets the time of the day	time.h
tmpnam()	creates a unique file name	stdio.h
tolower()	translates characters to lowercase	ctype.h
toupper()	translates characters to uppercase	ctype.h
ungetc()	pushes a character back into the input stream	stdio.h
vfprintf()	writes formatted output to a stream	stdio.h
vprintf()	writes formatted output to stdout	stdarg.h
vsprintf()	writes formatted output to a string	stdarg.h

Appendix C: Glossory

abstract class It acts as a frame work for creating new classes. It appears normally as the root of a class hierarchy. Its instances cannot be created.

abstract data type It is a data type whose internal representation is fully transparent to the user. They are popularly called ADTs (Abstract Data Types).

access operations They allow access to the internal state of objects without modifying them.

actor A model of concurrent computation in distributed systems. Computations are carried out in response to the communications sent to the actor system.

alias A different name given to a variable. Variable aliasing allows to access the same data with different names.

attributes Data members of an object.

base class A class from which new classes can be created.

callee A function which is called. It is also known as called function.

caller A function which calls. It is also known as calling function.

class It is the basic language construct in C++ for creating user-defined data types. It unites both the data and functions that operates on data.

class hierarchy The set of superclasses and subclasses derived from the superclasses can be arranged in a tree-like structure, with the superclasses on top of all classes derived from them. Such an arrangement is called a hierarchy of classes.

class object A variable whose data type is a class.

client An object which request services of other objects.

constructor A special member function of a class, which is invoked automatically whenever an instance of a class is created. It has the same name as its class.

container class A class that can store objects of other classes. Normally data structure classes act as container classes.

copy constructor A constructor which receives objects of the same class as argument. Object parameters to copy constructors must be passed either by reference or as pointers.

CORBA It is an acronym for Common Object Request Broker Architecture. Object Management Group (OMG) developed standards for connecting and integrating object applications running in heterogeneous, distributed computing environments. Defines the request protocol used by objects in communicating across platform and machine boundaries.

data abstraction It refers to creation of new data types that are well suited to an application to be programmed. 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.

data flow diagram A diagram that shows the flow of data through a system. It can have nodes to process those data also.

data hiding It hides data from rest of the program. Internal representation of hidden data is unknown to its users. However, it can be accessed by using interface functions.

data member A variable that is defined in a class declaration.

default parameter A parameter whose value is specified at the function declaration and is used if the corresponding actual parameter is missing in a call to that function.

delegation It is an alternative to class inheritance. 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.
derived class A class that inherits properties of other classes (base classes).
destructor A special member function of a class, which is invoked automatically whenever an object goes out of scope. It has the same name as its class with a tilde character prefixed.
dynamic binding It postpones the binding of a function call to a function until runtime. This is also known as late or runtime biding.
dynamic memory allocation It allows to allocate the requested amount of primary memory at runtime.
dynamic objects A class can be instantiated at runtime and objects created by such instantiation are called dynamic objects.
early binding The binding of a function call to a function is done during compile time. This is also known as static or compile-time biding.
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.
exception It refers to any unusual condition in a program. It is used to notify error to a caller.
exception handling It provides a way of transferring control and information to an unspecified caller that has expressed willingness to handle exceptions of a given type. Exception handling can be used to support notions of error handling and fault tolerant computing.
extensibility It is a feature which allows to extend the functionality of existing software components. In C++, this is achieved through abstract classes and inheritance.
extraction operator The operator >> which is used to read data from input stream object.
free store A pool of memory from which storage space of objects or variables is allocated. This is also know as heap.
friend A function which has authorization to access the private members of a class though it is not a member of the class.
friend class A class that can access private members of another class. That is, all member functions of a friend class are friend functions.
function overloading It allows multiple functions to assume the same name as long as they differ in terms of number of parameters or their data type.
function prototype It just specifies function return type and its arguments data type with function implementation. It is also know as function declarator.
genericity It is a technique for defining software components that have more than one interpretation depending on the parameters data type. 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.
header file A file containing declaration of new data types, macros, and function prototypes. For example, iostream.h is a header file.
indirection operator The * operator prefixed to a pointer variable. It is used to access the contents of the memory pointed to by a pointer variable.
inheritance It allows the extension and reuse of the existing code without having to rewrite the code from scratch. Inheritance involves derivation of new classes from existing ones, thus enabling the creation of a hierarchy of classes that simulates the class and subclass concept of the real world. A new

class created using existing classes (base classes) is called the derived class. This phenomenon is called inheritance. The derived class inherits the members - both data and functions of the base class.
inheritance path A series of classes that provide a path along which inheritance can takes place.
inline function A function whose body is substituted at the place of its call.
insertion operator The operator << which is used to send data to output stream object.
instance A variable or an object of a class is known as instance of a class.
instantiation The process of creation of objects of a class is called class instantiation.
interface Member functions that allow to access data members of a class.
late binding Refer to dynamic binding.
lifetime It is the interval of time an object exists by occupying memory.
manipulator A data object that is used with stream operators.
member Data and functions defined with a class are called members except friend functions.
member functions Functions which are members of a class are known as member functions.
message It is a request sent to an object.
message passing It is the process of invoking an operation on an object. In response to a message, the corresponding method (procedure) is executed in the object.
method A member function is also called as method.
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.
NULL The character that is used to indicate the end of the string.
NULL pointer A pointer that does not hold the address of any object.
object It is an instance of a class.
ODMG It is the acronym for Object Database Management Group. Small consortium, loosely affiliated with OMG, established to define a standard for data model and language interfaces to object-oriented database management systems.
OMG It is the acronym for Object Management Group. Consortium of OO software vendors, developers, and users promoting the use of objects for the development of distributed computing systems. World-Wide-Web (WWW) home page located at http://www.omg.org.
OO It is the acronym for Object-Oriented. It is an adjective (modifier) indicating that the associated noun has features to support role-oriented decomposition, modeling, or construction.
OOA It is the acronym for Object-Oriented Analysis. Use of role-oriented decomposition techniques to model a system.
OOBE It is the acronym for Object-Oriented Business Engineering. Application of object concepts to the design or restructuring of business processes or enterprise architecture.
OOD It is the acronym for Object-Oriented Design. Application of object concepts to the design of software.
OODB It is the acronym for Object-Oriented Database. A database where units of information are defined and managed as objects.
OOP It is the acronym for Object-Oriented Programming. An application of object concepts to the implementation of software, employing an OOPL.
OOPL It is the acronym for Object-Oriented Programming Languages. Programming language that includes features to support objects, such as data abstraction, encapsulation, sub-classing, inheritance, and polymorphism; examples include C++, Smalltalk, Self, Eiffel. May be a hybrid (incremented)

language that extends an otherwise non-OO base language through the addition of OO constructs (e.g., C++, Objective-C, Object Pascal, Ada).

OOPSLA A conference called Object-Oriented Programming, Systems, Languages, and Applications.

operator overloading It allows to extend functionality of a existing operator to operate on user-defined data type also.

pass by pointer The address of an actual parameter is explicitly passed to a function.

pass by reference The address of an actual parameter is implicitly passed to a function.

pass by value A copy of the actual parameter value is passed to a function.

persistence The phenomenon where 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.

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

preprocessor A part of the compiler that processes header files, macros, and escape sequences with the designated character.

private member A class member which is accessible to only members of a class or friend functions.

protected member A class member whose scope is the same as private except that it is inheritable.

public member A class member which is accessible to external users through dot operator.

pure virtual function A function whose declaration exist in a base class and implementation in derived classes. A class having pure virtual member functions cannot be instantiated and hence, such classes are called abstract classes.

reusability A feature which allows to build new classes from existing classes.

scope The region of code in which an item is visible.

scope resolution operator It 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.

server An object which services the client's requests.

static binding Refer to early binding.

static member A class member which is declared as static. A static data member of a class is shared by all the instances of the class. A static member functions cannot access auto members of a class.

stream A sequence of characters is called stream. It can be an input stream or an output stream.

structured programming Software development methodology which employs functional decomposition and a top-down design approach for developing modular software (traditional programming technique of breaking a task into modular subtasks).

sub-class Another name for derived class.

super-class Another name for base class.

templates See genericity.

this pointer It is a pointer (named as `this`) to the current object.

type conversion A conversion of a value from one type to another.

virtual base classes A class which gets inherited to a derived class more than once has to be declared as virtual. Such base classes are called virtual base classes.

virtual functions A member function prefixed with the keyword virtual. It allows to achieve dynamic binding.

Appendix D: ASCII Character Set

Character	Decimal	Character	Decimal
(NUL)	00	#	35
☺ (SOH)	01	$	36
☻ (STX)	02	%	37
♥ (ETX)	03	&	38
◆ (EOT)	04	'	39
♣ (ENQ)	05	(	40
♠ (ACK)	06	)	41
● (BEL)	07	*	42
● (BS)	08	+	43
(HT)	09	,	44
(LF)	10	-	45
♂ (VT)	11	.	46
♀ (FF)	12	/	47
♪ (CR)	13	0	48
♫ (SO)	14	1	49
☼ (SI)	15	2	50
► (DLE)	16	3	51
◄ (DC1)	17	4	52
↕ (DC2)	18	5	53
‼ (DC3)	19	6	54
¶ (DC4)	20	7	55
§ (NAK)	21	8	56
▬ (SYN)	22	9	57
↨ (ETB)	23	:	58
↑ (CAN)	24	;	59
↓ (EM)	25	<	60
→ (SUB)	26	=	61
← (ESC)	27	>	62
(cursor right) (FS)	28	?	63
(cursor left) (GS)	29	@	64
(cursor up) (RS)	30	A	65
(cursor down)(US)	31	B	66
(SP)	32	C	67
!	33	D	68
"	34	E	69

Continued ... ►

Character	Decimal	Character	Decimal
F	70	i	105
G	71	j	106
H	72	k	107
I	73	l	108
J	74	m	109
K	75	n	110
L	76	o	111
M	77	p	112
N	78	q	113
0	79	r	114
P	80	s	115
Q	81	t	116
R	82	u	117
S	83	v	118
T	84	w	119
U	85	x	120
V	86	y	121
W	87	z	122
X	88	{	123
Y	89	|	124
Z	90	}	125
[	91	~	126
\	92	(DEL)	127
]	93	Ç	128
^	94	Ü	129
_	95	é	130
`	96	â	131
a	97	ä	132
b	98	à	133
c	99	å	134
d	100	ç	135
e	101	ê	136
f	102	ë	137
g	103	è	138
h	104	ï	139

Continued ... ▶

Character	Decimal	Character	Decimal
î	140	»	175
ì	141	░	176
Ä	142	▒	177
Å	143	█	178
É	144	│	179
æ	145	┤	180
Æ	146	╡	181
Ô	147	╢	182
ö	148	╖	183
Ò	149	╕	184
û	150	╣	185
ù	151	║	186
ÿ	152	╗	187
Ö	153	╝	188
ü	154	╜	189
¢	155	╛	190
£	156	┐	191
¥	157	└	192
₧	158	┴	193
ƒ	159	┬	194
á	160	├	195
í	161	─	196
ó	162	┼	197
Ú	163	╞	198
^	164	╟	199
Ñ	165	╚	200
ª	166	╔	201
º	167	╩	202
¿	168	╦	203
⌐	169	╠	204
¬	170	═	205
½	171	╬	206
¼	172	╧	207
¡	173	╨	208
«	174	╤	209

Continued ... ▶

Character	Decimal	Character	Decimal
╥	210	θ	233
╙	211	Ω	234
╘	212	δ	235
╒	213	∞	236
╓	214	∅	237
╫	215	∈	238
╪	216	∩	239
┘	217	≡	240
┌	218	±	241
█	219	≥	242
▄	220	≤	243
▌	221	⌠	244
▐	222	⌡	245
▀	223	÷	246
α	224	≈	247
β	225	°	248
Γ	226	•	249
π	227	·	250
Σ	228	√	251
σ	229	η	252
μ	230	z	253
γ	231	▪	254
ϕ	232	(SP)	255

THE ASCII SYMBOLS

NUL	- Null	DLE	- Data Link Escape
SOH	- Start of Heading	DC	- Device Control
STX	- Start of Text	NAK	- Negative Acknowledge
ETX	- End of Text	SYN	- Synchronous Idle
EOT	- End of Transmission	ETB	- End of Transmission Block
ENQ	- Enquiry	CAN	- Cancel
ACK	- Acknowledge	EM	- End of Medium
BEL	- Bell	SUB	- Substitute
BS	- Backspace	ESC	- Escape
HT	- Horizontal Tabulation	FS	- File Separator
LF	- Line Feed	GS	- Group Separator
VT	- Vertical Tabulation	RS	- Record Separator
FF	- Form Feed	US	- Unit Separator
CR	- Carriage Return	SP	- Space (Blank)
SO	- Shift Out	DEL	- Delete
SI	- Shift In		

Appendix E: Bibliography

[1] Alan Joch, *Nine ways to make your code more reliable — How Software Doesn't Work ?*, Byte Magazine 49-60p, October 1995.

[2] Bernd Muller, *Is Object-Oriented Programming Structured Programming ?*, ACM SIGPLAN Notices, Volume 28, No. 9, September 1993.

[3] Bjarne Stroustrup, *The C++ Programming Language* (2nd Edition), Addison Wesley, 1991.

[4] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison Wesley, 1994.

[5] Bruce Eckel, *Using C++*, Osborne McGraw Hill, 1989.

[6] Capper, Colgate, Hunter, and James, *The impact of object-oriented technology on software quality: Three case histories*, IBM Systems Journal, Volume 33, No. 1, 1994.

[7] D. Dechanpeaur et al, *The Process of Object Oriented Design*, Seventh Annual Conference on Object-Oriented Programming, System, Language, and Applications (OOPSLA), 1992.

[8] Data Quest Magazine, *OOP - New Software Paradigm*, 1-15 April, 1995, India.

[9] E Balagurusamy, *Object-Oriented Programming with C++*, Tata McGraw Hill Publications, 1996.

[10] Edmund X Dejesus, *Big OOP, No Oops*, Byte, August 1995.

[11] Edward Yourdon, *Object-Oriented Systems Design*, Prentice Hall Inc, 1994.

[12] Harald M Muller, *Ten Rules for Handling Exceptions Handling Successfully*, C++ Report, Jan.1996.

[13] Henda Hodjami, *A Reuse approach based on Object-Orientation*, Software Engineering Notes, Proceedings of the Symposium on Software Reusability, August 1995.

[14] James and Josephine, *Reuse Through Inheritance*, Software Engineering Notes, Proceedings of the Symposium on Software Reusability, August 1995.

[15] Keith Gorlen, C++ *Under UNIX*, UNIX Papers, Waite Groups.

[16] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990, ISBN 0-201-51459-1.

[17] Markku Sakkinen, *The Darker Side of C++ Revisited*, Department of Computer Science and Information Systems, University of Jyvaskyla, Finland.

[18] Nicholas Wilt, *Templates in C++*, Supplement to Dr. Dobb's Journal, December 1992.

[19] Rajkumar, *Fault Tolerant Computing*, A Seminar Report, Bangalore University, 1995.

[20] Randall Hyde, *Object-Oriented Programming in Assembly Language*, Dr. Dobb's Journal, Mar.1990.

[21] Tim Rentsch, *Object-Oriented Programming*, SIGPLAN Notices, September, 1992.

[22] Robert G Fichman and Chris F Kemerer, *Object-Oriented and Conventional Analysus and Design Methodologies*, IEEE Computer, 1992.

[23] Robert Lafore, *Object-Oriented Programming in Turbo C++*, Waite Group, 1992.

[24] Steven J, *A Technique for Tracing Memory Leaks in C++*, NCR Microelectronics, Colorado.

[25] SunSoft, *C++ Selected Reading, Object-Oriented Programming*, August 1994.

[26] Turbo C++, *Library Reference Manual*, Borland International Inc., 1990

[27] Venugopal K R and Vimala H S, *Programming with Fortran*, Tata McGraw Hill, India, 1994.

[28] Venugopal K R and Vimala H S, *Programming with Pascal and C*, Tata McGraw Hill, India, 1994.

[29] Venugopal K R and Rajkumar, *Microprocessor x86 Programming*, BPB Publications, India, 1995.

[30] Venugopal K R and Maya, *Programming with Pascal*, Tata McGraw Hill, India, 1997.

[31] Venugopal K R and Sudeep, *Programming with C*, Tata McGraw Hill, India, 1997.