

The *type* must be a pointer or a reference to a defined class type. The *argument* object must be expression that resolves to a pointer or reference. The use of the operator `dynamic_cast()` is also called a *type-safe downcast*.

The typeid Operator

We can use the **typeid** operator to obtain the types of unknown objects, such as their class name at runtime. For example, the statement

```
char *objectType = typeid(object).name();
```

will assign the type of "object" to the character array **objectType** which can be printed out, if necessary. To do this, it uses the **name()** member function of the **type_info** class. The object may be of type **int**, **float**, etc. or of any class.

We must include `<typeinfo>` header file to use the operators **dynamic_cast** and **typeid** which provide run-time type information (RTTI).

16.4 Class Implementation

ANSI C++ Standard adds two unusual keywords, **explicit** and **mutable**, for use with class members.

The explicit Keyword

The **explicit** keyword is used to declare class constructors to be "explicit" constructors. We have seen earlier, while discussing constructors, that any constructor called with one argument performs *implicit conversion* in which the type received by the constructor is converted to an object of the class in which the constructor is defined. Since the conversion is automatic, we need not apply any casting. In case, we do not want such automatic conversion to take place, we may do so by declaring the one-argument constructor as explicit as shown below:

```
class ABC
{
    int m;
public:
    explicit ABC (int i)    // constructor
    {
        m = i;
    }
    // .....
    // .....
};
```

Here, objects of ABC class can be created using only the following form:

```
ABC abc1(100);
```

The automatic conversion form

```
ABC abc1 = 100;
```

is not allowed and illegal. Remember, this form is permitted when the keyword **explicit** is not applied to the conversion.

The mutable Keyword

We know that a class object or a member function may be declared as **const** thus making their member data not modifiable. However, a situation may arise where we want to create a **const** object (or function) but we would like to modify a particular data item only. In such situations we can make that particular data item modifiable by declaring the item as **mutable**. Example:

```
mutable int m;
```

Although a function(or class) that contains **m** is declared **const**, the value of **m** may be modified. Program 16.2 demonstrates the use of a **mutable** member.

USE OF KEYWORD **mutable**

```
#include <iostream>
using namespace std;

class ABC
{
private:
    mutable int m; // mutable member
public:
    explicit ABC(int x = 0)
    {
        m = x;
    }
    void change() const // const function
    {
        m = m+10;
    }
    int display() const // const function
    {
        return m;
    }
};
```

(Contd)

```
int main()
{
    const ABC abc(100);           // const object
    cout << "abc contains: " << abc.display();

    abc.change();                // changes mutable data
    cout << "\nabc now contains: " << abc.display();
    cout << "\n";

    return 0;
}
```

PROGRAM 16.2

The output of Program 16.2 would be:

```
abc contains: 100
abc now contains: 110
```

note

Although the function **change()** has been declared constant, the value of **m** has been modified. Try to execute the program after deleting the keyword **mutable** in the program.

16.5 Namespace Scope

We have been defining variables in different scopes in C++ programs, such as classes, functions, blocks, etc. ANSI C++ Standard has added a new keyword **namespace** to define a scope that could hold global identifiers. The best example of namespace scope is the C++ Standard Library. All classes, functions and templates are declared within the namespace named **std**. That is why we have been using the directive

```
using namespace std;
```

in our programs that uses the standard library. The **using namespace** statement specifies that the members defined in **std** namespace will be used frequently throughout the program.

Defining a Namespace

We can define our own namespaces in our programs. The syntax for defining a namespace is similar to the syntax for defining a class. The general form of namespace is:

```
namespace namespace_name
{
    // Declaration of
    // variables, functions, classes, etc.
}
```

note

There is one difference between a class definition and a namespace definition. The namespace is concluded with a closing brace but no terminating semicolon.

Example:

```
namespace TestSpace
{
    int m;
    void display(int n)
    {
        cout << n;
    }
} // No semicolon here
```

Here, the variable **m** and the function **display** are inside the scope defined by the **TestSpace** namespace. If we want to assign a value to **m**, we must use the scope resolution operator as shown below.

```
TestSpace::m = 100;
```

Note that **m** is qualified using the namespace name.

This approach becomes cumbersome if the members of a namespace are frequently used. In such cases, we can use a **using** directive to simplify their access. This can be done in two ways:

```
using namespace namespace_name; // using directive
using namespace_name::member_name; // using declaration
```

In the first form, all the members declared within the specified namespace may be accessed without using qualification. In the second form, we can access only the specified member in the program. Example:

```
using namespace TestSpace;
m = 100; // OK
display(200); // OK

using TestSpace::m;
m = 100; // OK
display(200); // Not ok, display not visible
```

Nesting of Namespaces

A namespace can be nested within another namespace. Example:

```
namespace NS1
{
```

```

.....
.....
namespace NS2
{
    int m = 100;
}
.....
.....
}

```

To access the variable **m**, we must qualify the variable with both the enclosing namespace names. The statement

```
cout << NS1::NS2::m;
```

will display the value of **m**. Alternatively, we can write

```
using namespace NS1;
cout << NS2::m;
```

Unnamed Namespaces

An unnamed namespace is one that does not have a name. Unnamed namespace members occupy global scope and are accessible in all scopes following the declaration in the file. We can access them without using any qualification.

A common use of unnamed namespaces is to shield global data from potential name clashes between files. Every file has its own, unique unnamed namespace.

Program 16.3 demonstrates how namespaces are defined, how they are nested and how an unnamed namespace is created. It also illustrates how the members in various namespaces are accessed.

USING NAMESPACE SCOPE WITH NESTING

```

#include <iostream>

using namespace std;

// Defining a namespace
namespace Name1
{
    double x = 4.56;
    int m = 100;

    namespace Name2 // Nesting namespaces
    {
        double y = 1.23;
    }
}

```

(Contd)

```

    }
}

namespace                // Unnamed namespace
{
    int m = 200;
}

int main()
{
    cout << "x = " << Name1::x << "\n";           // x is qualified
    cout << "m = " << Name1::m << "\n";
    cout << "y = " << Name1::Name2::y << "\n";     // y is fully qualified
    cout << "m = " << m << "\n";                 // m is global

    return 0;
}

```

PROGRAM 16.3

The output of Program 16.3 is:

```

x = 4.56
m = 100
y = 1.23
m = 200

```

note

We have used the variable **m** in two different scopes.

Program 16.4 shows the application of both the **using** directive and **using** declaration.

ILLUSTRATING THE **using** KEYWORD

```

#include <iostream>

using namespace std;

// Defining a namespace
namespace Name1
{
    double x = 4.56;
    int m = 100;

    namespace Name2                // Nesting namespaces
    {

```

(Contd)

```

        double y = 1.23;
    }
}

namespace Name3
{
    int m = 200;
    int n = 300;
}

int main()
{
    using namespace Name1;           // bring members of Name1
                                     // to current scope
    cout << "x = " << x << "\n";     // x is not qualified
    cout << "m = " << m << "\n";
    cout << "y = " << Name2::y << "\n"; // y is qualified
    using Name3::n;                  // bring n to current scope
    cout << "m = " << Name3::m << "\n"; // m is qualified
    cout << "n = " << n << "\n";       // n is not qualified

    return 0;
}

```

PROGRAM 16.4

The output of Program 16.4 would be:

```

x = 4.56
m = 100
y = 1.23
m = 200
n = 300

```

note

Understand how the data members are qualified when they are accessed.

Program 16.5 uses functions in a namespace scope.

PROGRAM 16.5: FUNCTIONS IN NAMESPACE SCOPE

```

#include <iostream>

using namespace std;

namespace Functions
{

```

(Contd)

```
int divide(int x,int y) // definition
{
    return(x/y);
}

int prod(int x,int y); // declaration only
}

int Functions::prod(int x,int y) // qualified
{
    return(x*y);
}

int main()
{
    using namespace Functions;

    cout << "Division: " << divide(20,10) << "\n";
    cout << "Multiplication: " << prod(20,10) << "\n";

    return 0;
}
```

PROGRAM 16.5

The output of Program 16.5 would be:

```
Division: 2
Multiplication: 200
```

note

When a function that is declared inside a namespace is defined outside, it should be qualified.

Program 16.6 demonstrates the use of classes inside a namespace.

USING CLASSES IN NAMESPACE SCOPE

```
include <iostream>

using namespace std;

namespace Classes
{
    class Test
    {
```

(Contd)


```
private:
    int m;

public:
    Test(int a)
    {
        m = a;
    }

    void display()
    {
        cout << "m = " << m << "\n";
    }
};

int main()
{
    // using scope resolution
    Classes::Test T1(200);
    T1.display();

    // using directive
    using namespace Classes;
    Test T2(400);
    T2.display();

    return 0;
}
```

PROGRAM 16.6

The output of Program 16.6 would be:

```
m = 200
m = 400
```

16.6 Operator Keywords

The ANSI C++ Standard proposes keywords for several C++ operators. These keywords, listed in Table 16.1, can be used in place of operator symbols in expressions. For example, the expression

```
x > y && m != 100
```

may be written as

```
x > y and m not_eq 100
```

Operator keywords not only enhance the readability of logical expressions but are also useful in situations where keyboards do not support certain special characters such as &, ^ and ~.

Table 16.1 Operator keywords

<i>Operator</i>	<i>Operator keyword</i>	<i>Description</i>
&&	and	logical AND
	or	logical OR
!	not	logical NOT
!=	not_eq	inequality
&	bitand	bitwise AND
	bitor	bitwise inclusive OR
^	xor	bitwise exclusive OR
~	compl	bitwise complement
&=	and_eq	bitwise AND assignment
=	or_eq	bitwise inclusive OR assignment
^=	xor_eq	bitwise exclusive OR assignment

16.7 New Keywords

ANSI C++ has added several new keywords to support the new features. Now, C++ contains 64 keywords, including **main**. They are listed in Table 16.2. The new keywords are boldfaced.

Table 16.2 ANSI C++ keywords

<i>asm</i>	<i>else</i>	<i>namespace</i>	<i>template</i>
auto	enum	new	this
bool	explicit	operator	throw
break	export	private	true
case	extern	protected	try
catch	false	public	typedef
char	float	register	typeid
class	for	reinterpret_cast	typename
const	friend	return	union
const_cast	goto	short	unsigned
continue	if	signed	using
default	inline	sizeof	virtual
delete	int	static	void
do	long	static_cast	volatile
double	main	struct	wchar_t
dynamic_cast	mutable	switch	while

16.8 New Headers

The ANSI C++ Standard has defined a new way to specify header files. They do not use **.h** extension to filenames. Example:

```
#include <iostream>
#include <fstream>
```

However, the traditional style **<iostream.h>**, **<fstream.h>**, etc. is still fully supported. Some old header files are renamed as shown below:

Old style	New style
<assert.h>	<cassert>
<ctype.h>	<cctype>
<float.h>	<float>
<limits.h>	<climits>
<math.h>	<cmath>
<stdio.h>	<cstdio>
<stdlib.h>	<cstdlib>
<string.h>	<cstring>
<time.h>	<ctime>

SUMMARY

- ⇔ ANSI C++ Standard committee has added many new features to the original C++ language specifications.
- ⇔ Two new data types **bool** and **wchar_t** have been added to enhance the range of data types available in C++.
- ⇔ The **bool** type can hold Boolean values, **true** and **false**.
- ⇔ The **wchar_t** type is meant to hold 16-bit character literals.
- ⇔ Four new cast operators have been added: **static_cast**, **const_cast**, **reinterpret_cast** and **dynamic_cast**.
- ⇔ The **static_cast** operator is used for any standard conversion of data types.
- ⇔ The **const_cast** operator may be used to explicitly change the **const** or **volatile** attributes of objects.
- ⇔ We can change the data type of an object into a fundamentally different type using the **reinterpret_cast** operator.
- ⇔ Casting of an object at run time can be achieved by the **dynamic_cast** operator.
- ⇔ Another new operator known as **typeid** can provide us run time type information about objects.
- ⇔ A constructor may be declared **explicit** to make the conversion explicit.
- ⇔ We can make a data item of a **const** object or function modifiable by declaring it **mutable**.

- ⇔ ANSI C++ permits us to define our own **namespace** scope in our program to overcome certain name conflict situations.
- ⇔ Namespaces may be nested.
- ⇔ Members of **namespace** scope may be accessed using either **using** declaration or **using** directive.
- ⇔ ANSI C++ proposes keywords that may be used in place of symbols for certain operators.
- ⇔ In new standard, the header files should be specified without **.h** extension and the **using** directive

```
using namespace std;
```

should be added in every program.

- ⇔ Some old style header files are renamed in the new standard. For example **math.h** file is known as **cmath**.

Key Terms

- **and**
- **and_eq**
- ANSI C++
- **bitand**
- **bitor**
- **bool**
- Boolean
- C_type casting
- C++ standard
- C++ type casting
- casts
- **compl**
- **const**
- **const** function
- **const** object
- **const_cast**
- constant casts
- current scope
- downcast
- **dynamic_cast**
- dynamic casts
- **explicit** constructor
- **false** value
- global identifiers
- **header file**
- implicit conversion
- **mutable** member
- **name()** function
- **namespace** scope
- nesting namespaces
- **not**
- **not_eq**
- operator keywords
- **or**
- **or_eq**
- polymorphic objects
- reinterpret casts
- **reinterpret_cast**

(Contd)

- RTTI
- source type
- standard library
- static casts
- **static_cast**
- **std** namespace
- target type
- **true** value
- type casts
- **type_info** class
- **type_safe** casting
- **typeid**
- **typeinfo** header
- unnamed namespaces
- **using** declaration
- **using** directive
- **using namespace**
- **volatile**
- **wchar_t**
- **wide_character** literal
- **xor**
- **xor_eq**

Review Questions

- 16.1 List the two data types added by the ANSI C++ standard committee.
- 16.2 What is the application of **bool** type variables?
- 16.3 What is the need for **wchar_t** character type?
- 16.4 List the new operators added by the ANSI C++ standard committee.
- 16.5 What is the application of **const_cast** operator?
- 16.6 Why do we need the operator **static_cast** while the old style cast does the same job?
- 16.7 How does the **reinterpret_cast** differ from the **static_cast**?
- 16.8 What is dynamic casting?. How is it achieved in C++?
- 16.9 What is **typeid** operator?. When is it used?
- 16.10 What is **explicit conversion**?. How is it achieved?
- 16.11 When do we use the keyword **mutable**?
- 16.12 What is a namespace conflict? How is it handled in C++?
- 16.13 How do we access the variables declared in a named namespace?
- 16.14 What is the difference between using the **using namespace** directive and using the **using** declaration for accessing **namespace** members?
- 16.15 What is wrong with the following code segment?

```
const int m = 100;
int *ptr = &m;
```

16.16 What is the problem with the following statements?

```
const int m = 100;
double *ptr = const_cast<double*>(&m);
```

16.17 What will be the output of the following program?

```
#include<iostream.h>
class Person
{
    // .....
}
int main()
{
    Person John;
    cout << " John is a ";
    cout << typeid(John).name() << "\n";
}
```

16.18 What is wrong with the following namespace definition?

```
namespace Main
{
    int main()
    {
        // .....
    }
}
```

Debugging Exercises

16.1 Identify the error in the following program.

```
#include <iostream>
class A
{
public:
    A()
    {
    }

    A(int i)
    {
    }
}
```

```
};

class B
{
public:
    B()
    {
    }

    explicit B(int)
    {
    }
};

void main()
{
    A a1=12;
    A a2;
    A a3=a1;

    B b1 = 12;
}
```

16.2 Identify the error in the following program.

```
#include <iostream.h>

class A
{
protected:
    int i;
public:
    A()
    {
        i = 10;
    }

    int getI()
    {
```

```
        return i;
    }
};

class B: public A
{
public:
    B()
    {
    }

    int getI()
    {
        return i + i;
    }
};

void main()
{
    A *a = new A();
    B *b = static_cast<B*>(a);
    cout << b->getI();
}
```

16.3 Identify the error in the following program.

```
#include <iostream.h>

namespace A
{
    int i;
    void dispI()
    {
        cout << i;
    }
}

void main()
{
    namespace Inside
```



```
{
    int insideI;
    void dispInsideI()
    {
        cout << insideI;
    }
}

A::i = 10;
cout << A::i;
A::dispI();

Inside::insideI = 20;
cout << Inside::insideI;
Inside::dispInsideI();
}
```

Programming Exercises

- 16.1 Write a program to demonstrate the use of ***reinterpret_cast*** operator.
- 16.2 Define a namespace named **Constants** that contains declarations of some constants. Write a program that uses the constants defined in the namespace **Constants**.

17

Object-Oriented Systems Development

Key Concepts

- Software development components
- Procedure-oriented development tools
- Object-oriented paradigm
- OOP notations and graphs
- Data flow diagrams
- Object-oriented design
- Top-down decomposition
- System implementation
- Procedure-oriented paradigm
- Classic software development life cycle
- Fountain model
- Object-oriented analysis
- Textual analysis
- Class hierarchies
- Structured design
- Prototyping paradigm

17.1 Introduction

Software engineers have been trying various *tools*, *methods*, and *procedures* to control the process of software development in order to build high-quality software with improved productivity. The methods provide "how to's" for building the software while the tools provide automated or semi-automated support for the methods. They are used in all the stages of software development process, namely, planning, analysis, design, development and maintenance. The software development procedures integrate the methods and tools together and enable rational and timely development of software systems (Fig.17.1). They provide guideines as to how to apply the methods and tools, how to produce the deliverables at each stage, what controls to apply, and what milestones to use to assess the progress.

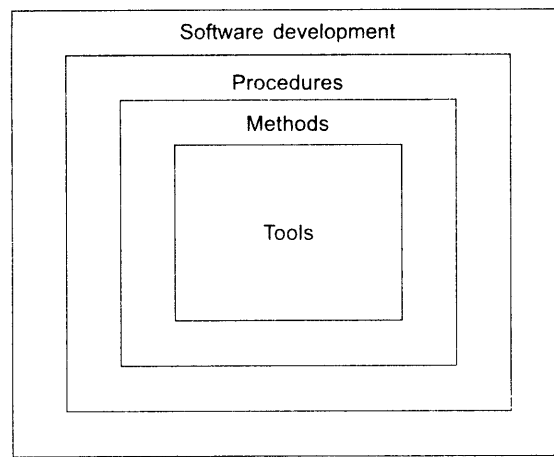


Fig. 17.1 ⇔ *Software development components*

There exist a number of software development paradigms, each using a different set of methods and tools. The selection of a particular paradigm depends on the nature of the application, the programming language used, and the controls and deliverables required. The development of a successful system depends not only on the use of the appropriate methods and techniques but also on the developer's commitment to the objectives of the system. A successful system must:

1. satisfy the user requirements,
2. be easy to understand by the users and operators,
3. be easy to operate,
4. have a good user interface,
5. be easy to modify,
6. be expandable,
7. have adequate security controls against misuse of data,
8. handle the errors and exceptions satisfactorily, and
9. be delivered on schedule within the budget.

In this chapter, we shall review some of the conventional approaches that are being widely used in software development and then discuss some of the current ideas that are applicable to the object-oriented software development.

17.2 Procedure-Oriented Paradigms

Software development is usually characterized by a series of stages depicting the various tasks involved in the development process. Figure 17.2 illustrates the classic software life cycle that is most widely used for the procedure-oriented development. The classic life cycle is based on an underlying model, commonly referred to as the "water-fall" model. This model attempts to break up the identifiable activities into series of actions, each of which must be

completed before the next begins. The activities include problem definition, requirement analysis, design, coding, testing, and maintenance. Further refinements to this model include iteration back to the previous stages in order to incorporate any changes or missing links.

Problem Definition: This activity requires a precise definition of the problem in user terms. A clear statement of the problem is crucial to the success of the software. It helps not only the developer but also the user to understand the problem better.

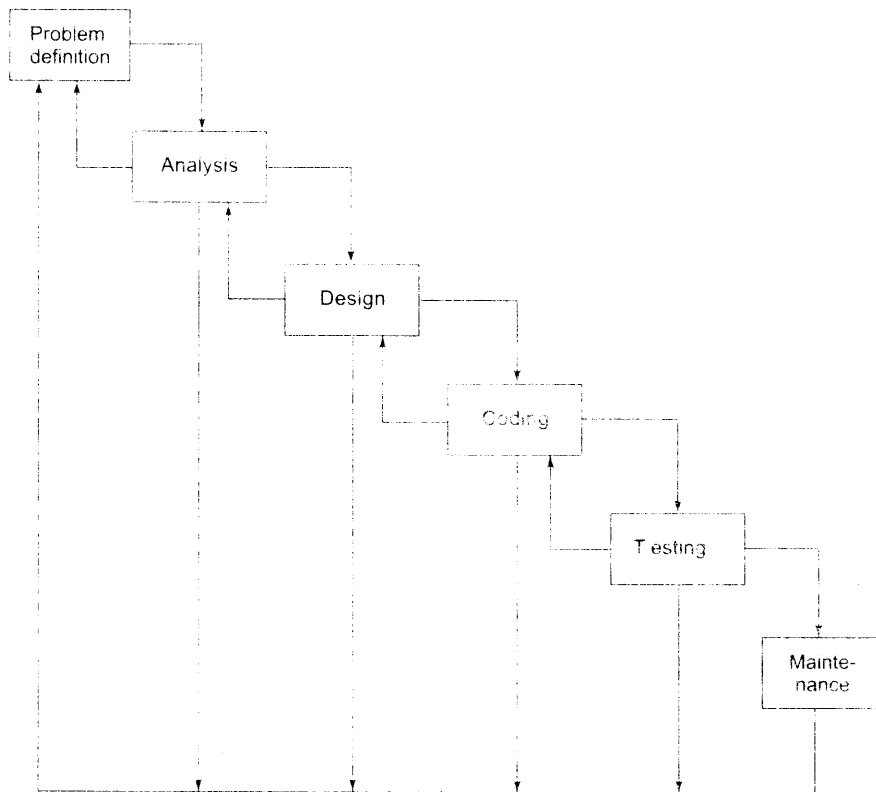


Fig. 17.2 ↔ Classic software development life cycle (Embedded 'water-fall' mode)

Analysis: This covers a detailed study of the requirements of both the user and the software. This activity is basically concerned with what of the system such as

- what are the inputs to the system?
- what are the processes required?
- what are the outputs expected?
- what are the constraints?

Design: The design phase deals with various concepts of system design such as data structure, software architecture, and algorithms. This phase translates the requirements into a representation of the software. This stage answers the questions of *how*.

Coding: Coding refers to the translation of the design into machine-readable form. The more detailed the design, the easier is the coding, and better its reliability.

Testing: Once the code is written, it should be tested rigorously for correctness of the code and results. Testing may involve the individual units and the whole system. It requires a detailed plan as to what, when and how to test.

Maintenance: After the software has been installed, it may undergo some changes. This may occur due to a change in the user's requirement, a change in the operating environment, or an error in the software that has not been fixed during the testing. Maintenance ensures that these changes are incorporated wherever necessary.

Each phase of the life cycle has its own goals and outputs. The output of one phase acts as an input to the next phase. Table 17.1 shows typical outputs that could be generated for each phase of the life cycle.

Table 17.1 Outputs of classic software life cycle

Phase	Output
Problem definition (why)	<ul style="list-style-type: none"> • Problem statement sheet • Project request
Analysis (what)	<ul style="list-style-type: none"> • Requirements document • Feasibility report • Specifications document • Acceptance test criteria
Design (how)	<ul style="list-style-type: none"> • Design document • Test class design
Coding (how)	<ul style="list-style-type: none"> • Code document (program) • Test plan • User manual
Testing (what and how)	<ul style="list-style-type: none"> • Tested code • Test results • System manual
Maintenance	<ul style="list-style-type: none"> • Maintenance log sheets • Version documents

The software life cycle, as described above, is often implemented using the *functional decomposition technique*, popularly known as *top-down, modular* approach. The functional decomposition technique is based on the interpretation of the problem space and its translation into the solution space as an inter-dependent set of functions. The functions are decomposed into a sequence of progressively simpler functions that are eventually implemented. The final system is seen as a set of functions that are organized in a top-down hierarchical structure.

There are several flaws in the top-down, functional decomposition approach. They include:

1. It does not allow evolutionary changes in the software.
2. The system is characterized by a single function at the top which is not always true. In fact many systems have no top.

3. Data is not given the importance that it deserves.
4. It does not encourage reusability of the code.

17.3 Procedure-Oriented Development Tools

A large number of tools are used in the analysis and design of the systems. It is important to note that the process of systems development has been undergoing changes over the years due to continuous changes in the computer technology. Consequently, there has been an evolution of new system development tools and techniques. These tools and techniques provide answers to the *how* questions of the system development.

The development tools available today may be classified as the *first generation*, *second generation*, and *third generation* tools. The first generation tools developed in the 1960's and 1970's are called the traditional tools. The second generation tools introduced in the late 1970's and early 1980's are meant for the structured systems analysis and design and therefore they are known as the structured tools. The recent tools are the third generation ones evolved since late 1980's to suit the *object-oriented* analysis and design.

Table 17.2 shows some of the popular tools used for various development processes under the three categories. Although this categorization is questionable, it gives a fair idea of the growth of the tools during the last three decades.

Table 17.2 System development tools

Process	First generation	Second generation	Third generation
Physical processes	System flowcharts	Context diagrams	Inheritance graphs Object-relationship charts
Data representation	Layout forms Grid charts	Data dictionary	Objects object dictionary
Logical processes	Playscript English narrative	Decision tables & trees Data flow diagrams	Inheritance graphs Data flow diagrams
Program representation	Program flowcharts I/O layouts	Structure charts Warnier /Orr diagrams	State change diagrams Ptech diagrams Coad/Yourdon charts

This section gives an overview of some of the most frequently used first and second generation tools. Object-oriented development tools will be discussed later in this chapter (as and when they are required).

System flowcharts: A graphical representation of the important inputs, outputs, and data flow among the key points in the system.

Program flowcharts: A graphical representation of the program logic.

Playscripts: A narrative description of executing a procedure.

Layout forms: A format designed for putting the input data or displaying results.

Grid charts: A chart showing the relationship between different modules of a system.

Context diagrams: A diagram showing the inputs and their sources and the outputs and their destinations. A context diagram basically outlines the system boundary.

Data flow diagrams: They describe the flow of data between the various components of a system. It is a network representation of the system which includes processes and data files.

Data dictionary: A structured repository of data about data. It contains a list of terms and their definitions for all the data items and data stores.

Structure chart: A graphical representation of the control logic of functions (modules) representing a system.

Decision table: A table of contingencies for defining a problem and the actions to be taken. It presents the logic that tells us what action to take when a given condition is true or otherwise.

Decision tree: A graphic representation of the conditions and outcomes that resemble the branches of a tree.

Warnier/Orr diagrams: A horizontal hierarchy chart using nested sets of braces, pseudocodes, and logic symbols to indicate the program structure.

17.4 Object-Oriented Paradigm

The object-oriented paradigm draws heavily on the general systems theory as a conceptual background. A system can be viewed as a collection of *entities* that interact together to accomplish certain objectives (Fig. 17.3). Entities may represent physical objects such as equipment and people, and abstract concepts such as data files and functions. In object-oriented analysis, the entities are called *objects*.

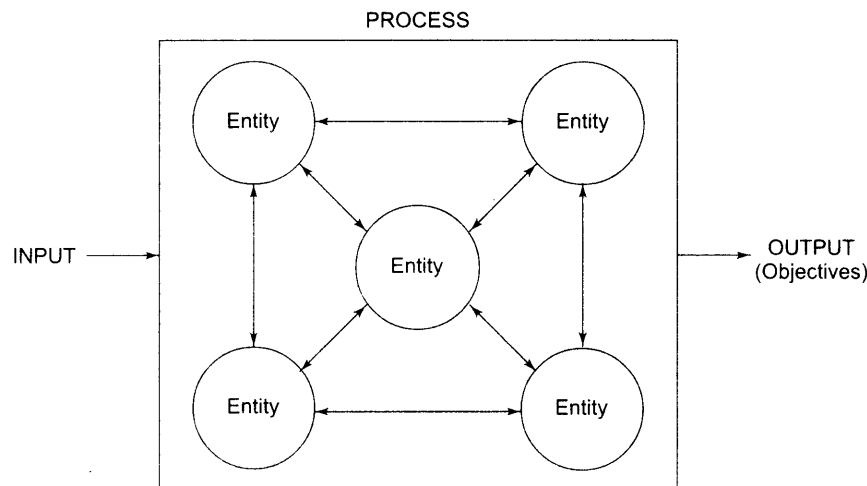


Fig. 17.3 ⇔ A system showing inter-relationship of entities

As the name indicates, the object-oriented paradigm places greater emphasis on the objects that encapsulate data and procedures. They play the central role in all the stages of the software development and, therefore, there exists a high degree of overlap and iteration between the stages. The entire development process becomes evolutionary in nature. Any

graphical representation of the object-oriented version of the software development life cycle must, therefore, take into account these two aspects of overlap and iteration. The result is a “fountain model” in place of the classic “water-fall” model as shown in Fig. 17.4. This model depicts that the development reaches a higher level only to fall back to a previous level and then again climbing up.

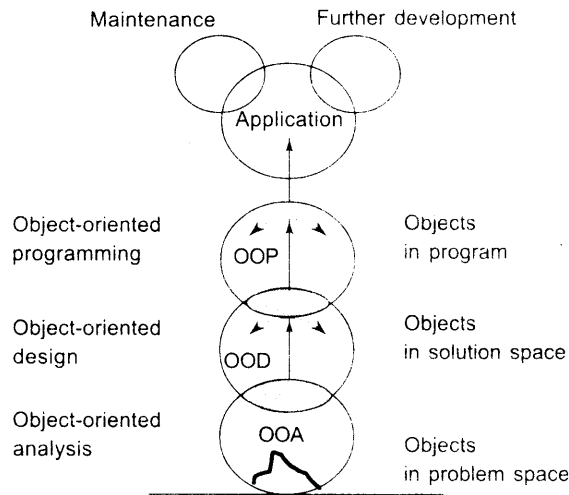


Fig. 17.4 ⇔ *Fountain model of object-oriented software development*

Object-oriented analysis (OOA) refers to the methods of specifying requirements of the software in terms of real-world objects, their behaviour, and their interactions. *Object-oriented design* (OOD), on the other hand, turns the software requirements into specifications for objects and derives class hierarchies from which the objects can be created. Finally, *object-oriented programming* (OOP) refers to the implementation of the program using objects, in an object-oriented programming language such as C++.

By developing specifications of the objects found in the problem space, a clear and well-organized statement of the problem is actually built into the application. These objects form a high-level layer of definitions that are written in terms of the problem space. During the refinement of the definitions and the implementation of the application objects, other objects are identified. Figure 17.5 illustrates the layers of the object specifications that result from this process.

All the phases in the object-oriented approach work more closely together because of the commonality of the object model. In one phase, the problem domain objects are identified, while in the next phase additional objects required for a particular solution are specified. The design process is repeated for these implementation-level objects.

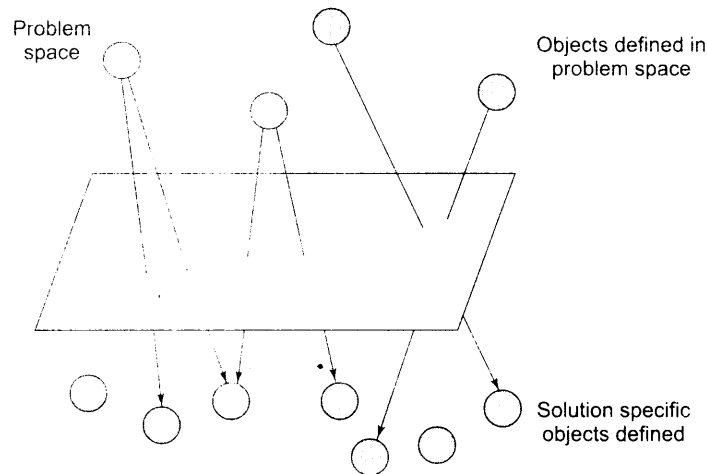


Fig. 17.5 ↔ *Layers of object specifications*

In contrast to the traditional top-down functional decomposition approach, the object-oriented approach has many attributes of both the top-down and bottom-up designs. The top functional decomposition techniques can be applied to the design of individual classes, while the final system can be constructed with the help of class modules using the bottom-up approach.

17.5 Object-Oriented Notations and Graphs

Graphical notations are an essential part of any design and development process, and object-oriented design is no exception. We need notations to represent classes, objects, subclasses, and their inter-relationships. Unfortunately, there are no standard notations for representing the objects and their interactions. Authors and researchers have used their own notations. Some of them are used more frequently while others are not. Figures 17.6 through 17.14 show some of the commonly used notations to represent the following:

1. Classes and objects.
2. Instances of objects.
3. Message communication between objects.
4. Inheritance relationship.
5. Classification relationship.
6. Composition relationship.
7. Hierarchical chart.
8. Client-server relationship.
9. Process layering.

We must use these notations and graphs wherever possible. They improve not only the clarity of the processes but also the productivity of the software developers.

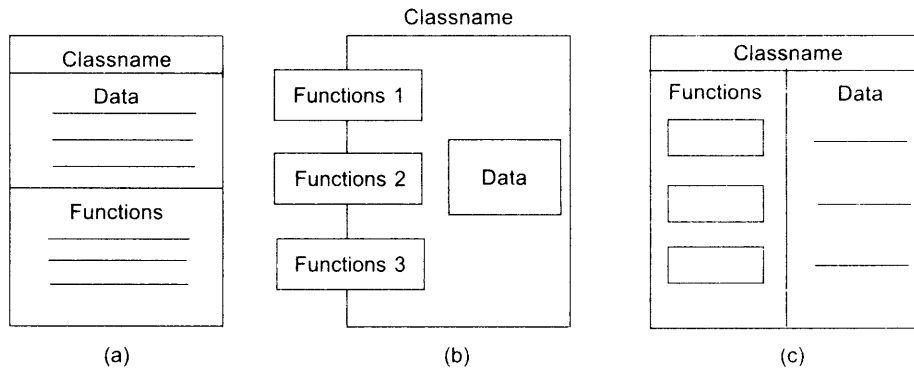


Fig. 17.6 ⇔ *Various forms of representation of classes/objects*

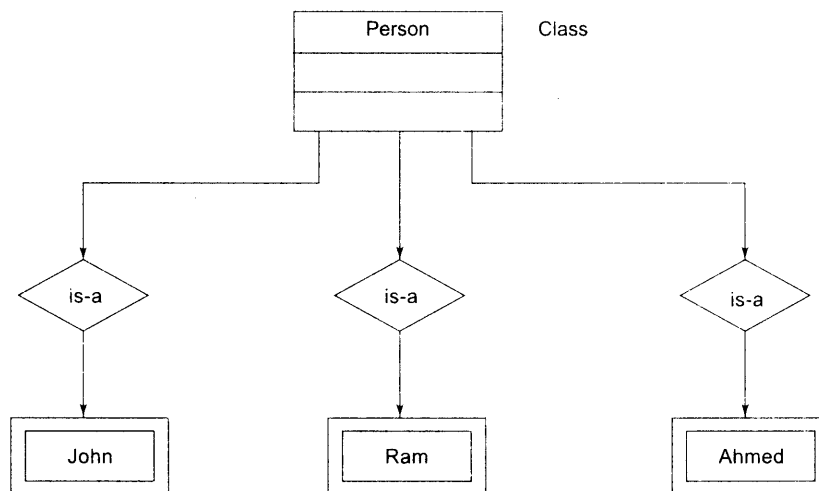


Fig. 17.7 ⇔ *Instances of objects*

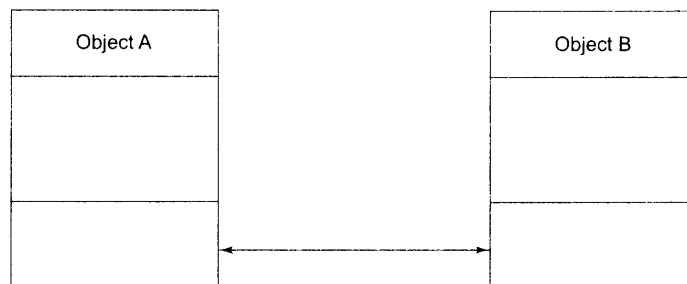


Fig. 17.8 ⇔ *Message communication between objects*

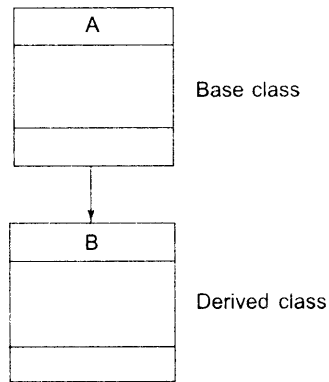


Fig. 17.9 ⇔ *Inheritance relationship*

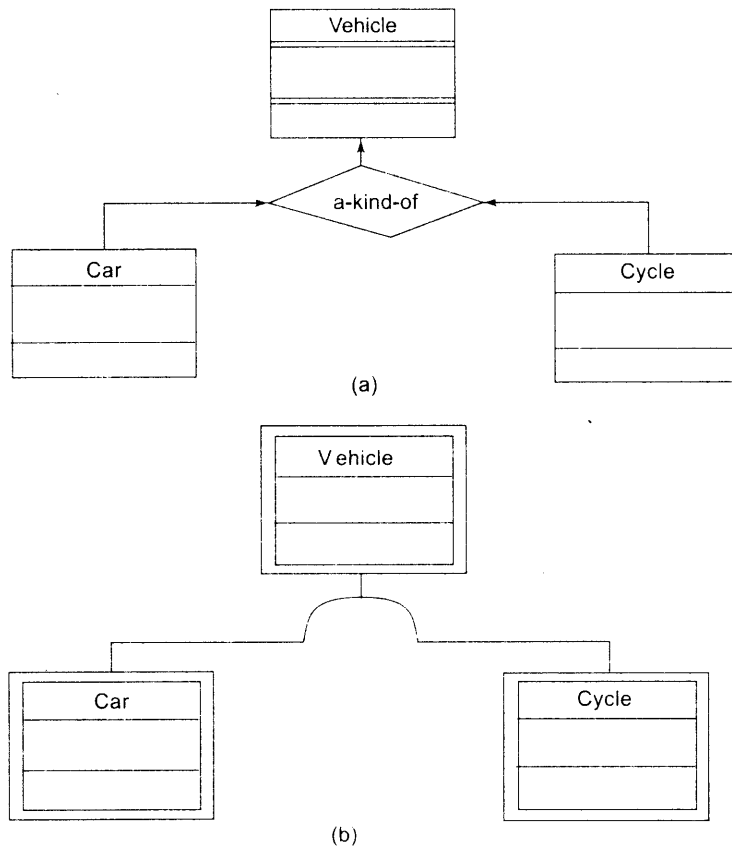


Fig. 17.10 ⇔ *Classification relationship*

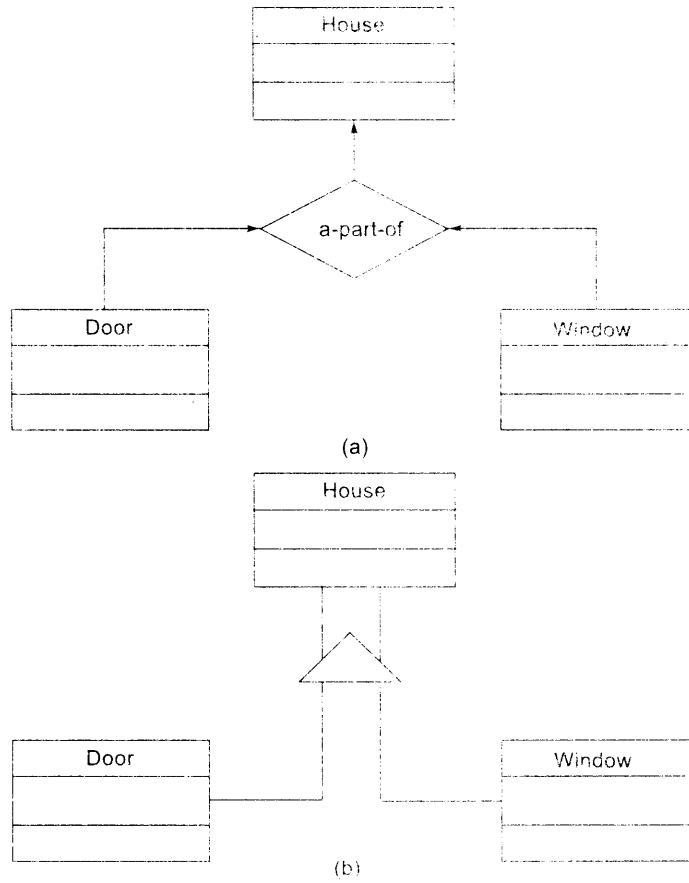


Fig. 17.11 ⇔ Composition relationship

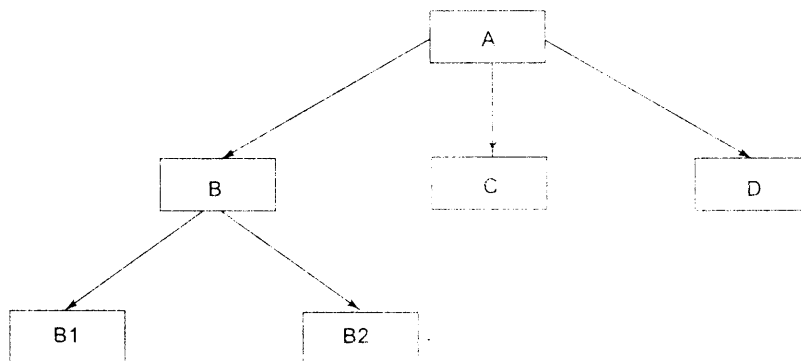


Fig. 17.12 ⇔ Hierarchical chart

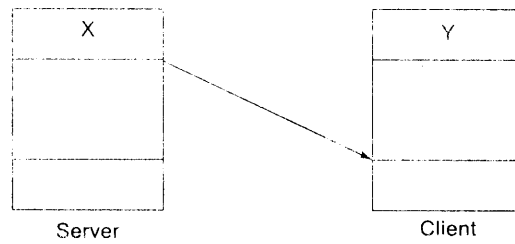


Fig. 17.13 ⇔ *Client-server relationship*

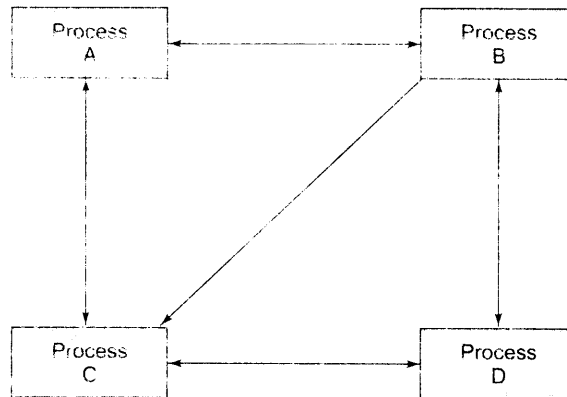


Fig. 17.14 ⇔ *Process layering (A process may have typically five to seven objects)*

17.6 Steps in Object-Oriented Analysis

Object-oriented analysis provides us with a simple, yet powerful, mechanism for identifying objects, the building block of the software to be developed. The analysis is basically concerned with the decomposition of a problem into its component parts and establishing a logical model to describe the system functions.

The object-oriented analysis (OOA) approach consists of the following steps:

1. Understanding the problem.
2. Drawing the specifications of requirements of the user and the software.
3. Identifying the objects and their attributes.
4. Identifying the services that each object is expected to provide (interface).
5. Establishing inter-connections (collaborations) between the objects in terms of services required and services rendered.

Although we have shown the above tasks as a series of discrete steps, the last three activities are carried out inter-dependently as shown in Fig. 17.15.

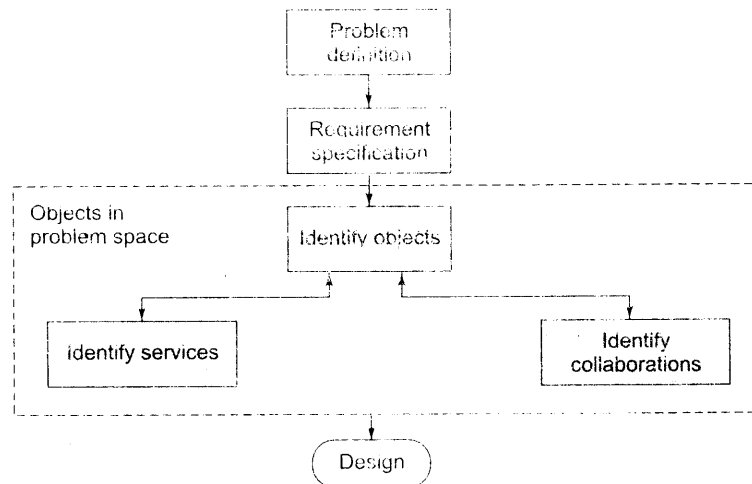


Fig. 17.15 ⇔ *Activities of object-oriented analysis*

Problem Understanding

The first step in the analysis process is to understand the problem of the user. The problem statement should be refined and redefined in terms of computer system engineering that could suggest a computer-based solution. The problem statement should be stated, as far as possible, in a single, grammatically correct sentence. This will enable the software engineers to have a highly focussed attention on the solution of the problem. The problem statement provides the basis for drawing the requirements specification of both the user and the software.

Requirements Specification

Once the problem is clearly defined, the next step is to understand what the proposed system is required to do. It is important at this stage to generate a list of user requirements. A clear understanding should exist between the user and the developer of what is required. Based on the user requirements, the specifications for the software should be drawn. The developer should state clearly

- What outputs are required.
- What processes are involved to produce these outputs.
- What inputs are necessary.
- What resources are required.

These specifications often serve as a reference to test the final product for its performance of the intended tasks.

Identification of Objects

Objects can often be identified in terms of the real-world objects as well as the abstract objects. Therefore, the best place to look for objects is the application itself. The application may be analyzed by using one of the following two approaches:

1. Data flow diagrams (DFD)
2. Textual analysis (TA)

Data Flow Diagram

The application can be represented in the form of a data flow diagram indicating how the data moves from one point to another in the system. The boxes and *data stores* in the data flow diagram are good candidates for the objects. The process *bubbles* correspond to the procedures. Figure 17.16 illustrates a typical data flow diagram. It is also known as a *data flow graph* or a *bubble chart*.

A DFD can be used to represent a system at any level of abstraction. For example, the DFD shown in Fig. 17.16 may be expanded to include more information (such as payment details) or condensed as illustrated in Fig. 17.17 to show only one bubble.

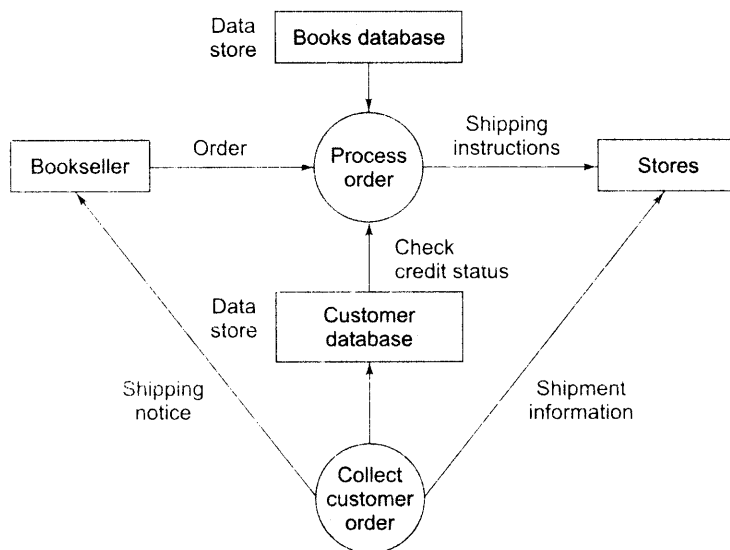


Fig. 17.16 ⇔ *Data flow diagram for order processing and shipping for a publishing company*

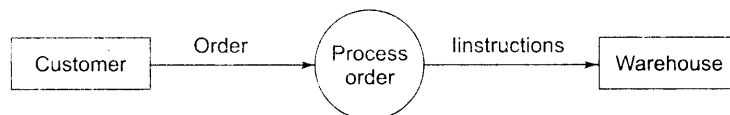


Fig. 17.17 ⇔ *Fundamental data flow diagram*

Textual Analysis

This approach is based on the textual description of the problem or proposed solution. The description may be of one or two sentences or one or two paragraphs depending on the type and complexity of the problem. The nouns are good indicators of the objects. The names can further be classified as *proper nouns*, *common nouns*, and *mass or abstract nouns*. Table 17.3 shows the various types of nouns and their meaning.

Table 17.3 *Types of nouns*

<i>Type of noun</i>	<i>Meaning</i>	<i>Example</i>
Common noun	Describe classes of things (entites)	Vehicle, customer income, deduction
Proper noun	Names of specific things	Maruti car, John, ABC company
Mass or abstract noun	Describe a quality. Quantity or an activity associated with a noun	Salary-income house-learn feet, traffic

It is important to note that the context and semantics must be used to determine the noun categories. A particular word may mean a common noun in one context and a mass or abstract noun in another.

These approaches are only a guide and not the ultimate tools. Creative perception and intuition of the experienced developers play an important role in identifying the objects.

Using one of the above approaches, prepare a list of objects for the application problem. This might include the following tasks:

1. Prepare an object table.
2. Identify the objects that belong to the solution space and those which belong to the problem space only. The problem space objects are outside the software boundary.
3. Identify the attributes of the solution space objects.

Remember that not all the nouns will be of interest to the final realization of the solution. Consider the following requirement statements of a system:

Identification of Services

Once the objects in the solution space have been identified, the next step is to identify a set of services that each object should offer. Services are identified by examining all the verbs and verb phrases in the problem description statement. Verbs which can note actions or occurrences may be classified as shown in Table 17.4.

Doing verbs and *compare verbs* usually give rise to services (which we call as functions in C++). *Being verbs* indicate the existence of the classification structure while *having verbs* give rise to the composition structures.

Table 17.4 Classification of verbs

<i>Types of verb</i>	<i>Meaning</i>	<i>Examples</i>
Doing verbs	operations	read , get, display, buy
Being verbs	classifications	is an, belongs to
Having verbs	composition	has an, is part of
Compare verbs	operations	is less than, is equal to
Stative verbs	invariance-condition	to be present, are owned

Establishing Interconnections

This step identifies the services that objects provide and receive. We may use an information flow diagram (IFD) or an entity-relationship (ER) diagram to enlist this information. Here, we must establish a correspondence between the services and the actual information (messages) that are being communicated.

17.7 Steps in Object-Oriented Design

Design is concerned with the mapping of objects in the problem space into objects in the solution space, and creating an overall structure (architectural model) and computational models of the system. This stage normally uses the bottom-up approach to build the structure of the system and the top-down functional decomposition approach to design the class member functions that provide services. It is particularly important to construct structured hierarchies, to identify abstract classes, and to simplify the inter-object communications. Reusability of classes from the previous designs, classification of the objects into subsystems and determination of appropriate protocols are some of the considerations of the design stage. The object oriented design (OOD) approach may involve the following steps:

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

Review of Problem Space Objects

An exercise to review the objects identified in the problem space is undertaken as a first step in the design stage. The main objective of this review exercise is to refine the objects in terms of their attributes and operations and to identify other objects that are solution specific. Some guidelines that might help the review process are:

1. If only one object is necessary for a service (or operation), then it operates only on that object.
2. If two or more objects are required for an operation to occur, then it is necessary to identify which object's private part should be known to the operation.
3. If an operation requires knowledge of more than one type of objects, then the operation is not functionally cohesive and should be rejected.

Applying these guidelines will help us refine the services of the objects. Further, the redundant or extraneous objects are removed, synonymous services are combined and the names of the operations (functions) are improved to denote clearly the kind of processing involved.

Class Dependencies

Analysis of relationships between the classes is central to the structure of a system. Therefore, it is important to identify appropriate classes to represent the objects in the solution space and establish their relationships. The major relationships that are important in the context of design are:

1. Inheritance relationships.
2. Containment relationships.
3. Use relationships.

Inheritance relationship is the highest relationship that can be represented in C++. It is a powerful way of representing a hierarchical relationship directly. The real appeal and power of the inheritance mechanism is that it allows us to reuse a class that is almost, but not exactly, what we want and to tailor the class in a way that it does not introduce any unwanted side effects into the rest of the class. We must review the attributes and operations of the classes and prepare a *inheritance relationship* table as shown in Table 17.5.

Table 17.5 *Inheritance relationship table*

<i>Class</i>	<i>Depends on</i>
A
B	A
C	A
D	B
B1	B
B2	B

Containment relationship means the use of an object of a class as a member of another class. This is an alternative and complimentary technique to use the class inheritance. But, it is often a tricky issue to choose between the two techniques. Normally, if there is a need to override attributes or functions, then the inheritance is the best choice. On the other hand, if we want to represent a property by a variety of types, then the containment relationship is the right method to follow. Another place where we need to use an object as a member is when we need to pass an attribute of a class as an argument to the constructor of another class. The “another” class must have a member object that represents the argument. The inheritance represents *is_a* relationship and the containment represents *has_a* relationship.

Use relationship gives information such as the various classes a class uses and the way it uses them. For example, a class **A** can use classes **B** and **C** in several ways:

- **A** reads a member of **B**.
- **A** calls a member of **C**.
- **A** creates **B** using new operator.

The knowledge of such relationships is important to the design of a program.

Organization of Class Hierarchies

In the previous step, we examined the inheritance relationships. We must re-examine them and create a class hierarchy so that we can reuse as much data and/or functions that have been designed already. Organization of the class hierarchies involves identification of common attributes and functions among a group of related classes and then combining them to form a new class. The new class will serve as the super class and the others as subordinate classes (which derive attributes from the super class). The new class may or may not have the meaning of an object by itself. If the object is created purely to combine the common attributes, it is called an *abstract class*.

This process may be repeated at different levels of abstraction with the sole objective of extending the classes. As hierarchy structure becomes progressively higher, the amount of specification and implementation inherited by the lower level classes increases. We may repeat the process until we are sure that no new class can be formed. Figure 17.18 illustrates a two-level iteration process.

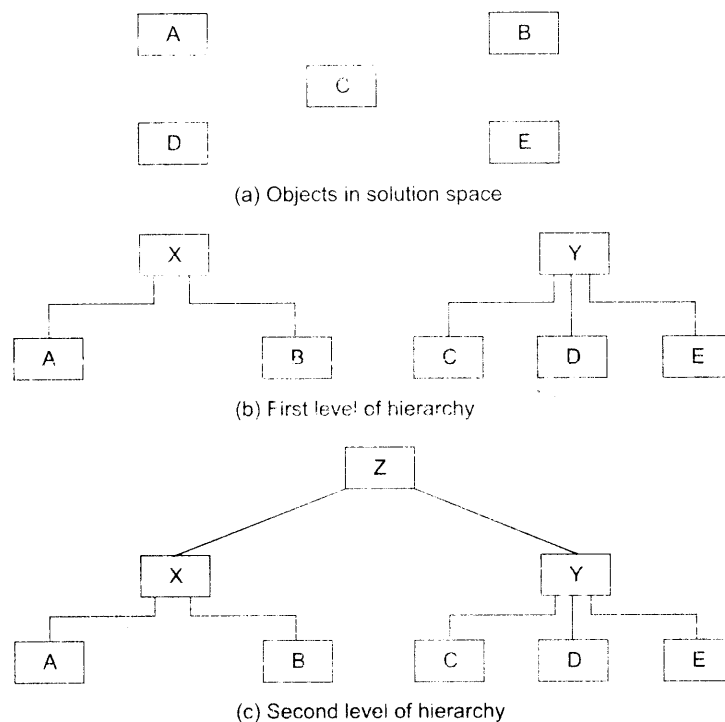
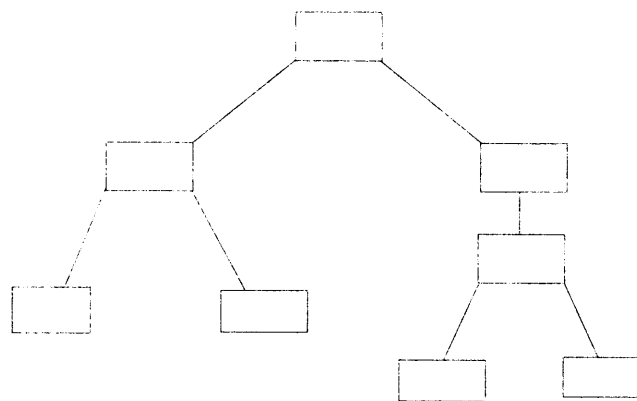
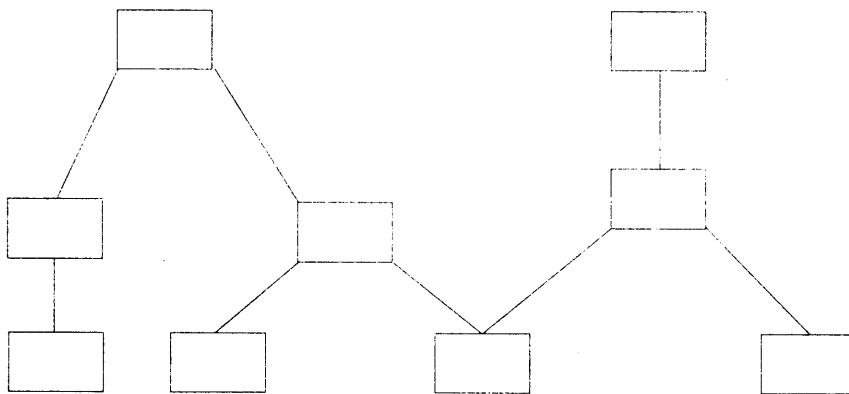


Fig. 17.18 ⇔ Level of class hierarchies

The process of a class organization may finally result in a *single-tree model* as shown in Fig. 17.19(a) or *forest model* as shown in Fig. 17.19(b).



(a) Single-tree model



(b) Forest model

Fig. 17.19 ⇔ *Organisation of classes*

Design of Classes

We have identified classes, their attributes, and *minimal* set of operations required by the concept a class is representing. Now we must look at the complete details that each class represents. The important issue is to decide what functions are to be provided. For a class to be useful, it must contain the following functions, in addition to the service functions:

1. Class management functions.
 - How an object is created?
 - How an object is destroyed?
2. Class implementation functions.
What operations are performed on the data type of the class?
3. Class access functions.
How do we get information about the internal variables of the class?
4. Class utility functions.
How do we handle errors?

Other issues that are to be considered are:

1. What kind of access controls are required for the base classes?
2. Which functions can be made virtual?
3. What library classes are expected to be used in a class?
4. The design of the individual classes has a major impact on the overall quality of the software.

Given below are some guidelines which should be considered while designing a class:

1. The public interface of a class should have only functions of the class.
2. An object of one class should not send a message directly to a member of another class.
3. A function should be declared public only when it is required to be used by the objects of the class.
4. Each function either accesses or modifies some data of the class it represents.
5. A class should be dependent on as few (other) classes as possible.
6. Interactions between two classes must be explicit.
7. Each subordinate class should be designed as a specialization of the base class with the sole purpose of adding additional features.
8. The top class of a structure should represent the abstract model of the target concept.

Design of Member Functions

We have so far identified

1. classes and objects,
2. data members,
3. interfaces,
4. dependencies, and
5. class hierarchy (structure).

It is time now to consider the design of the member functions. The member functions define the operations that are performed on the object's data. These functions behave like any other C function and therefore we can use the top-down functional decomposition technique to design them as shown in Fig. 17.20.

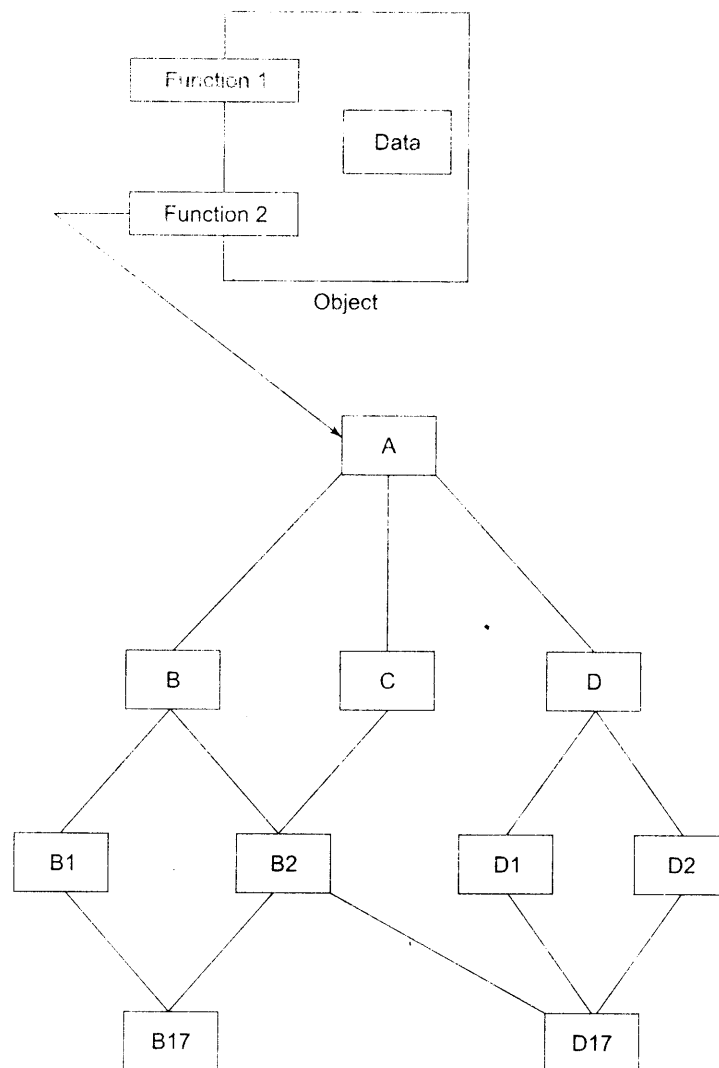


Fig. 17.20 ⇔ *Top-down design of functions*

We may also apply the *structured design* techniques to each module. The basic structured techniques supported by C++ are shown in Fig. 17.21. We may implement them in any combination and in any order using one-entry, one-exit concept.

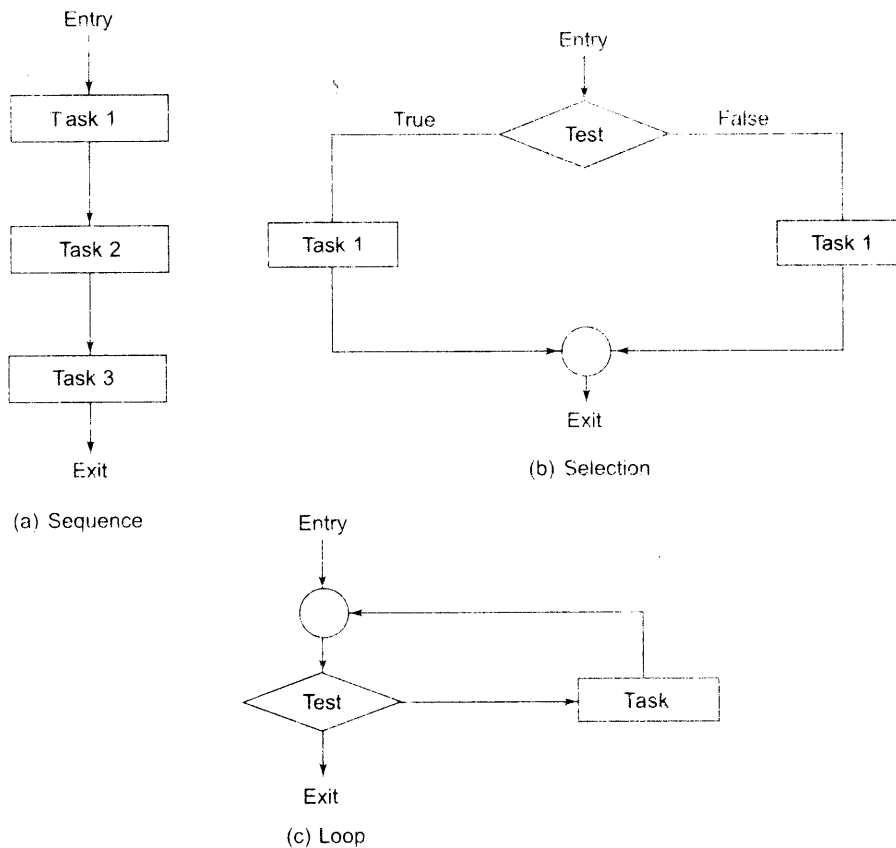


Fig. 17.21 ⇔ Structure design techniques

Design of the Driver Program

Every C++ program must contain a **main()** function code known as the driver program. The execution of the program begins and ends here. The driver program is mainly responsible for:

1. receiving data values from the user,
2. creating objects from the class definitions,
3. arranging communication between the objects as a sequence of messages for invoking the member functions, and
4. displaying output results in the form required by the user.

All activities, including processing during the execution of the program, result from the mutual interactions of the objects. One major design decision to be made is the logical order of the message passing.

The driver program is the gateway to the users. Therefore, the design of user-system interface (USI) should be given due consideration in the design of the driver program. The system should be designed to be user-friendly so that users can operate in a natural and comfortable way.

17.8 Implementation

Implementation includes coding and testing. Coding includes writing codes for classes, member functions and the **main** program that acts as a driver in the program. Coding becomes easy once a detailed design has been done with care.

No program works correctly the first time. So testing the program before using is an essential part of the software development process. A detailed test plan should be drawn as to what, when and how to test. The class interfaces and class dependencies are important aspects for testing. The final goal of testing is to see that the system performs its intended job satisfactorily.

17.9 Prototyping Paradigm

Most often the real-world application problems are complex in nature and therefore the structure of the system becomes too large to work out the precise requirements at the beginning. Some particulars become known and clear only when we build and test the system. After a large system is completed, incorporation of any feature that has been identified as "missing" at the testing or application stage might be too expensive and time consuming. One way of understanding the system design and its ramifications before a complete system is built is to build and test a working model of the proposed system. The model system is popularly known as a *prototype*, and the process is called *prototyping*. Since the object-oriented analysis and design approach is evolutionary, it is best suited for *prototyping paradigm* which is illustrated in Fig. 17.22.

A prototype is a scaled down version of the system and may not have stringent performance criteria and resource requirements. Developer and customer agree upon certain "outline specifications" of the system and a prototype design is proposed with the outline requirements and available resources. The prototype is built and evaluated. The major interest is not in the prototype itself but in its performance which is used to refine the requirement specifications. Prototypes provide an opportunity to experiment and analyze various aspects of the system such as system structure, internal design, hardware requirements and the final system requirements. The benefits of using the prototype approach are:

- We can produce understandable specifications which are correct and complete as far as possible.
- The user can understand what is being offered.
- Maintenance changes that are required when a system is installed, are minimized.
- Development engineers can work from a set of specifications which have been tested and approved.

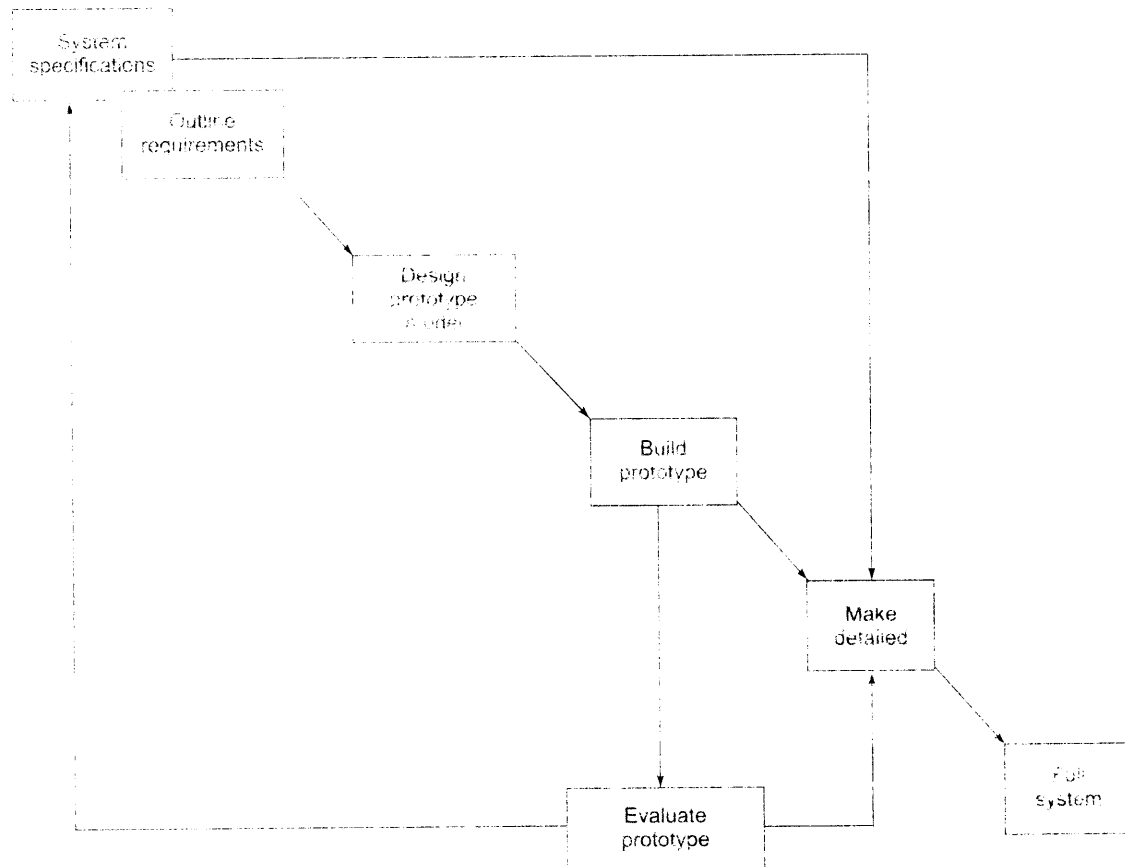


Fig. 17.22 ⇔ *Prototype paradigm*

Prototype is meant for experimenting. Most often it cannot be tuned into a product. However, occasionally, it may be possible to tune a prototype into a final product if proper care is taken in redesigning the prototype. The best approach is to throw away the prototype after use.

17.10 Wrapping Up

We have discussed various aspects of the object-oriented analysis and design. Remember, there is no one approach that is always right. You must consider the ideas presented here as only guidelines and use your experience, innovation and creativity wherever possible.

Following are some points for your thought and innovation:

1. Set clear goals and tangible objectives.
2. Try to use existing systems as examples or models to analyze your system.

3. Use classes to represent concepts.
4. Keep in mind that the proposed system must be flexible, portable, and extendable.
5. Keep a clear documentation of everything that goes into the system.
6. Try to reuse the existing functions and classes.
7. Keep functions strongly typed wherever possible.
8. Use prototypes wherever possible.
9. Match design and programming style.
10. Keep the system clean, simple, small and efficient as far as possible.

SUMMARY

- ⇔ The classic system development life cycle most widely used for procedure oriented development consists of following steps.
 - Problem definition
 - Analysis
 - Design
 - Coding
 - Testing
 - Maintenance
- ⇔ In object oriented paradigm, a system can be viewed as a collection of entities that interact together to accomplish certain objectives.
- ⇔ In object oriented analysis, the entities are called objects. Object oriented analysis (OOA) refers to the methods of specifying requirements of the software in terms of real world objects, their behaviour and their interactions with each other.
- ⇔ Object oriented design (OOD) translates the software requirements into specifications for objects, and derives class hierarchies from which the objects can be created.
- ⇔ Object oriented programming (OOP) refers to the implementation of the program using objects, with the help of object oriented programming language such as C++.
- ⇔ The object oriented analysis (OOA) approach consists of the following steps:
 - Defining the problem.
 - Estimating requirements of the user and the software.
 - Identifying the objects and their attributes.
 - Identifying the interface services that each object is supposed to provide.
 - Establishing interconnections between the objects in terms of services required and services rendered.
- ⇔ The object oriented design (OOD) approach involves the following steps:
 - Review of objects created in the analysis phase.
 - Specification of class dependencies.

- Organization of class hierarchies.
 - Design of classes.
 - Design of member functions.
 - Design of driver program.
- ⇔ One way of understanding the system design and its ramifications before a complete system is built is to build and test a working model of the proposed system. This model system is called the prototype, and the process is called prototyping.
- ⇔ The benefits of using the prototype approach are:
- You can produce understandable specifications which are correct and complete as far as possible.
 - The user can understand what is being offered.
 - Maintenance changes that are required when a system is installed are minimized.
 - Development engineers can work from a set of specifications, which have been tested and approved.

Key Terms

- | | |
|-------------------------------|-------------------------------|
| ➤ abstract class | ➤ loop |
| ➤ abstract nouns | ➤ maintenance |
| ➤ a-kind-of | ➤ mass nouns |
| ➤ analysis | ➤ member functions |
| ➤ a-part-of | ➤ message communications |
| ➤ being verbs | ➤ methods |
| ➤ bubble chart | ➤ modular approach |
| ➤ class hierarchies | ➤ object-oriented analysis |
| ➤ classes | ➤ object-oriented design |
| ➤ classic life cycle | ➤ object-oriented paradigm |
| ➤ classification relationship | ➤ object-oriented programming |
| ➤ client-server relationship | ➤ objects |
| ➤ coding | ➤ playscripts |
| ➤ collaborations | ➤ problem definition |
| ➤ common nouns | ➤ problem space |
| ➤ compare verbs | ➤ procedure-oriented paradigm |
| ➤ composition relationship | ➤ procedures |
| ➤ containment relationship | ➤ process layering |
| ➤ context diagrams | ➤ program flowcharts |

(Contd)

- data dictionary
- data flow diagrams
- decision table
- decision tree
- design
- development tools
- doing verbs
- driver program
- entities
- entity relationship diagram
- entity-relationship
- fist generation
- flowcharts
- forest model
- fountain model
- functional decomposition
- grid charts
- has-a relationship
- having verbs
- hierarchical chart
- information flow diagram
- inheritance relationship
- instances of objects
- is-a relationship
- layout forms
- proper nouns
- prototype
- prototyping
- prototyping paradigm
- second generation
- selection
- sequence
- single-tree model
- software life cycle
- solution space
- stative verbs
- structure chart
- structured design
- structured tools
- system flowcharts
- testing
- textual analysis
- third generation
- tools
- top-down approach
- traditional tools
- use relationship
- Warnier diagrams
- water-fall model

Review Questions

- 17.1 *List five most important features, in your opinion, that a software developer should keep in mind while designing a system.*
- 17.2 *Describe why the testing of software is important.*
- 17.3 *What do you mean by maintenance of software? How and when is it done?*
- 17.4 *Who are the major players in each stage of the systems development life cycle?*
- 17.5 *Is it necessary to study the existing system during the analysis stage? If yes, why? If no, why not?*
- 17.6 *What are the limitations of the classic software development life cycle?*
- 17.7 *“Software development process is an iterative process”. Discuss.*

- 17.8 Distinguish between the “water-fall” model and the “fountain” model.
- 17.9 Distinguish between object-oriented systems analysis and systems design. Which of the two requires more creative talents of the system developer?
- 17.10 Distinguish between the following.
- (a) Classification relationship and composition relationship.
 - (b) Inheritance relationship and client-server relationship.
 - (c) Objects in problem space and objects in solution space.
 - (d) Data flow diagrams and hierarchical charts.
- 17.11 Discuss the application of structured design techniques in object-oriented programming.
- 17.12 What are the critical issues that are to be considered while designing the driver program? Why?
- 17.13 “No program works correctly first time.” Comment.
- 17.14 What is prototyping? How does it help improve the system design?
- 17.15 State whether the following statements are TRUE or FALSE.
- (a) An important consideration when designing a new system is the end-user.
 - (b) The user has no role in the analysis and design of a system.
 - (c) A decision table is a pictorial representation of data flow.
 - (d) The only useful purpose of data flow diagrams is documentation.
 - (e) Data flow diagrams stress logical flow of data versus physical flow of data.
 - (f) Computer outputs can be designed in such a way that it is a humanizing force.
 - (g) Structured programming techniques have no place in the object-oriented program design.
 - (h) A prototype cannot be improved into a final product.

Appendix A

Projects

A.1 Memory Game

Learning Objectives

The designing of the Memory Game project enable the students to:

- Create a simple C++ game application
- Generate random values and explore the use of random functions
- Handle arrays with pointers efficiently in the scope-based scenario
- Use the gotoxy function to locate the cursor point effectively
- Identify the levels of the project that map to the requirement standards

Understanding the Memory Game

The game is about finding the matching pairs hidden in the 5×4 matrix. The matrix elements are composed of numbers and characters. The matrix consists of 20 elements, of which 10 of them are unique. The other 10 elements are the repetitions of the unique ones to match the elements in the cells.

The initial screen displays the 5×4 matrix with all the elements being hidden under the '@' symbol. The rows are named as 'A', 'B', 'C', 'D', 'E' and the columns are named as '0', '1', '2', '3'.

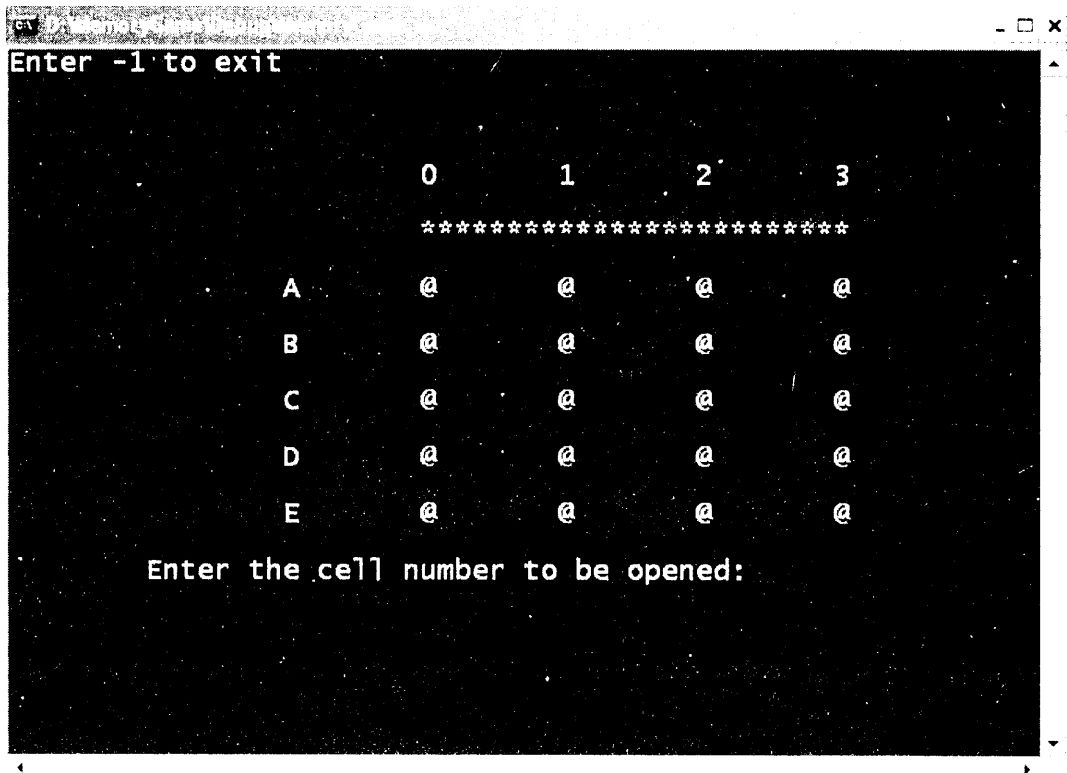
The user needs to enter the cell number to open the respective cell. For example,

'D3' opens 'Fourth row Fourth column element'.

and so on. If the two consecutive turned out cells do not match, then they are turned back to the default symbol ('@') when the third cell number is entered. On the other hand, if the

turned out cells match, then the cell values remain visible. We have to find the next pair in the similar manner.

When all the cells are turned out with the matching values, the GAME is OVER. A screen showing the number of tries taken to complete the matching pairs will be displayed. If we want to continue the game, we can press 'Y' to go to the initial screen to continue the game. We can enter "-1" to terminate the game at any point of time. The user interface of the memory game project is shown below:



```
Enter -1 to exit

          0      1      2      3
          *****
          A      @      @      @      @
          B      @      @      @      @
          C      @      @      @      @
          D      @      @      @      @
          E      @      @      @      @

Enter the cell number to be opened:
```

Fig. A.1

Creating the Memory Game

To create the memory game application, a C++ source file and a C++ header file is created. The source file is named as "**MemoryGame.cpp**" and the header file is named as "**RandomGeneration.h**"

Creating the Source File (MemoryGame.cpp)

We need to create a class, called **Game** in the source file (MemoryGame.cpp) that does the following tasks.

- Displaying a 5×4 matrix containing the default symbol ('@').
- Creating a 5×4 matrix containing the respective cell values.
- Changing the cell values in the 5×4 matrix as per user entry.
- Retaining the entries in the cells if the cell values are identical.
- Identifying the location of the cells to place the matrix elements.
- Counting the number of tries taken by the user to complete the game.

To implement the above tasks, we need to create the **Game** class with appropriate member variables and member functions as specified below.

Member Variables

Variable	Description
boardDefault[5][4]	Specifies the matrix, whose elements are filled with default '@' symbol.
boardSet[5][4]	Specifies the matrix, whose elements are filled with respective values in each cells.
boardCurrent[5][4]	Specifies the matrix, whose elements are filled with values as per the user entry.
Tries	Counts the number of attempts taken by the user to complete the game.
cmp1 , cmp2	Used for comparing the values of two consecutive entries in the matrix.
t1,t2,t3,t4	Used to retain the cell number of two consecutive cells entered by the user.

Member Functions

Functions	Description
CreateBoard	Used to display the game board with the default '@' symbol.
CurrentMatrix	Used to store and display the values in the game board when the user had started playing the game.
FillBoard	Used to fill a matrix with the values, that should be made visible when a cell is turned out.
ChangeCell	Used to fill the turned out cell with the value generated by FillBoard function.
IdentifyLocation	Used to find the location on the screen to place the turned out cell value. It takes the cell number as its parameter.
ReplaceWithDefault	Used to replace the turned out cell value with the default '@' symbol if two consecutive turned out cells are not identical. It takes the cell number as its parameter.

Creating the Header File (RandomGeneration.h)

The **RandomGeneration.h** header file contains the class called RandomGeneration, which is responsible for generating a random number. The generated random number is used to create differing values in the cells each time when the application is executed.

The class has two member variables “Low” and “High with the default access. The member functions used in the class along with their description is tabulated below:

Member Functions

<i>Functions</i>	<i>Description</i>
DrawRandomNumber	Generates the Random number and returns an integer type.
SetTimerSeed	Sets the seed to generate the psuedo-random numbers.
GetHigh and GetLow	Properties of the private members of the class to read their values. Their return type is int.

Working with the Memory Game

As we know the structure of the classes and the purpose of the member functions, the concentration is on defining and utilizing the member functions.

Identifying the cell locations

To display the matrix in the center of the screen, gotoxy function is used. There is no default gotoxy function in C++. One of the functions to achieve this is with the use of “windows.h” header file.

The code listing for the gotoxy function is specified below:

```
void gotoxy(short x, short y)
{
HANDLE hCon = GetStdHandle(STD_OUTPUT_HANDLE);
COORD pos;
pos.X=x-1;
pos.Y=y-1;
SetConsoleCursorPosition(hCon, pos);
}
```

Displaying the Default Matrix

When the application is executed, a game board (with dimension 5×4) is displayed which contains default ‘@’ symbol in all 20 cells.

For each element in the 5×4 matrix, the placeholder for those elements is identified using this gotoxy function. The matrix boardDefault[5][4] stores the default ‘@’ symbol for all cells. The position for placing these cell values is identified using gotoxy function.

Validating User Entry

If a particular cell is to be opened, we have to enter the respective cell number. When a cell number is entered, we have to address the following issues:

- Whether the entered cell number is a valid cell number, that is, it falls within the specified range.
- If the cell number entered is the repeated, then proper error messages needs to be supplied.
- If the entry is -1 then the user has the option to quit the game and proper messages needs to be specified for starting the game again.

Identifying the Cell Numbers

The cell number is of type `char[2]` as it is a combination of alphabets and numbers (for example A0, E3, etc). But the matrix indices are of type integers. Hence, we need to convert each character in the cell number to its corresponding integer type.

The code listing for identifying the first row elements is specified below:
(`cellno` variable contains the cell number entered by the user. Ex: If the user enters, say "A0")

```
switch(cellno[0])
{
    case 'A':
    case 'a':
        cell_1=0; break;
    case 'B':
    case 'b':
        cell_1=1; break;
    case 'C':
    case 'c':
        cell_1=2; break;
    case 'D':
    case 'd':
        cell_1=3; break;
    case 'E':
    case 'e':
        cell_1=4; break;
    default :
        {
            gotoxy(1,22+cnt);
            cout<<"\nEnter valid cell no..";
        }
}
```

The above code checks whether `cellno[0]` falls in the specified range. In the same manner, the `cellno[1]` is also converted to integer.