

B.E.

Fourth Semester Examination, May-2009

Programming Languages (CSE-204-E)

Note : Attempt any five questions.

Q. 1. (a) Discuss the various programming language translators.

Ans. Various Programming Language Translators : Language translators convert programming source code into language that the computer processor understands (machine language). Different types of translations must occur to turn programming source code into machine language, which is made up of bits of binary data. The three major types of language translators are compilers, assemblers & interpreters.

Compilers : A compiler is a special program that takes written sources code & turns it into machine language. When a compiler executes, it analyzes all of the language statements in the source code & builds the machine language object code. After a program is compiled, it is then a form that the preprocessor can execute on instruction at a time.

In some operating systems, an additional step called linking is required after compilation.

Most high level programming languages come with a compiler. However, object code is unique for each type of computer. Many different compilers exist for each language in order to translate for each type of computer. In addition, the compiler industry is quite competitive, so there are actually many compilers for each language on each type of computer.

Assembler : An assembler translates assembly language into machine language. Assembly language is one step removed from machine language. It uses computer-specific commands & structure similar to machine language, but assembly language uses names instead of numbers.

An assembler is similar to a compiler, but it is specific to translating programs written in assembly language into machine language. To do this, the assembler takes basic computer instructions from assembly language & converts them into a pattern of bits for the computer processor to use to perform its operations.

Interpreters : Many high level programming languages have the option of using an interpreter instead of a compiler. Some of these languages exclusively use an interpreter. An interpreter behaves very differently from compilers or assemblers. It converts programs into machine executable from each line of source code, in order, without looking at the entire program, an interpreter processes the program as it is being executed.

Q. 1. (b) Define programming language. Explain the syntactic and semantic rules of a programming language.

Ans. Programming Language : A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behaviour of a machine, to express algorithms precisely, or as a mode of human communication.

Many programming languages have some form of written specification of their syntax (form) & semantics (meaning). Some languages are defined by a specification document.

The earliest programming languages predate the invention of the computer & were used to direct the behaviour of machines such as Jacquard looms & player pianos.

Syntactic and Semantic Rules of a Programming Language

All programming languages have some primitive building blocks for the description of data & the processes or transformations applied to them. These primitives are defined by syntactic & semantic rules which describe their structure & meaning respectively.

Syntax : A programming language's surface is known as its syntax.

The syntax of language describes the possible combinations of symbols that form a syntactically correct problem. The meaning given to a combination of symbols is handled by semantics. Programming language syntax is usually defined using a combination of regular expressions & Back us-Naur Form.

Example based on LISP :

Expression	:	=	atom		list
atom	:	=	number		symbol
number	:	=	[+ -] ? [0' - '9'] +		
symbol	:	=	['A' - 'Z' 'a' - 'z']. *		
list	:	=	('expression *')		

This grammar specifies the following :

- (i) an expression is either an atom or a list.
- (ii) an atom is either a number or a symbol
- (iii) a number is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign.
- (iv) a symbol is a letter followed by 0 or more of any characters (excluding whitespace); &
- (v) a list is matched pair of parentheses with 0 (zero) or more expressions inside.

Semantics : The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages static semantics essentially include those semantics roles that can be checked at compile time.

Examples include checking that every identifier is declared before it is used or that the labels on the arms of a case statement & distinct etc.

Q.2. (a) Define "Abstract Data Types". What do you mean by data abstraction? Explain with example.

Ans. Abstract Data Types : When one considers the definition of a datatype in its most general form, one sees that there are several classes of attributes that may be distinguished. One class consists of those attributes which describe the representation of the objects of the data type in terms of other objects. Another describes the implementation of the operations in terms of other operations.

Abstract type is often used in the literature to refer to a class of object which are defined by a representation independent specification. Finally, if we are to have a means for data type abstraction, then it should allow us to distinguish these classes. This serves to support at least 2 goals. First, it permits authentication by checking that operations to be performed on certain objects are in fact legal. The second goal one might call security or even secrecy. Its purpose is to prevent users of the data type from writing programs which somehow → depend upon the types representation, rather than upon its abstract properties.

Ex. Structure Stack;

```
new stack( ) → stack
push (stack, item) → stack
pop (stack) → stack
top (stack) → item
snew(stack) → Boolean
declare stk : stack, i; item
pop (push (stk, i) = stk
top (push (stk, i)) = i
is new (newstack) = true
```


is new(push (stk, i)) = false

restrictions

pop (newstack) = error

top (newstack) = error

* * An abstract specification of a stack

Data Abstraction : A data abstraction in a programming language is a mechanism which collects together (or encapsulates) the representation & the operations of a data type. This encapsulation forms a wall which is intended to shield the data type from improper uses. But it also provides a "window" which allows the user a well-defined means for accessing the data type. Thus, a data abstraction facility is more than just a new way of defining a data type like a record, array, set or file would be in PASCAL. In addition to defining a new type & usually allowing variables to be declared as only holding values of that type, it also shields the representation of the type, allows implementors to provide some operation to the user of the type, but possibly retain others for themselves. It usually does all this by introducing new scope rules into a programming language.

Q. 2. (b) Discuss 'Names and referencing' environment for data control in a programming language.

Ans. Names and Referencing Environment

Referencing through a Named Data Object—A data object may be given a name when it is created, & the name may then be used to designate it as an operand of an operation. Alternatively, the data object may be made a component of another data object that has a name so that the name of the larger data object may be used together with a selection operation to designate the data object as an operand.

Data transmission is used for data control within expressions, but most data control outside of expressions involves the use of names of the referencing names. The problem of the meaning of names forms the central concern in data control.

Program Elements that may be Named : Each language differs, but some general categories seen in many languages are as follows :

- (i) Variable names.
- (ii) Formal Parameter names
- (iii) Subprogram names
- (iv) Names for defined types.
- (v) Names for defined constants
- (vi) Statement labels
- (vii) Exception names
- (viii) Names for primitive operations
- (ix) Names for literal constants.

Referencing Environments : Each program or subprogram has a set of identifier associations available for use in referencing during its execution. This set of identifier associations is termed the referencing environment of the subprogram. The referencing environment of a subprogram is ordinarily invariant during its execution. It is set up when the subprogram activation is created & it remains unchanged during the lifetime of the activation. The values contained in the various data objects may change, but the associations of names with data objects & subprograms do not. The referencing environment of a subprogram may have several components :

- (i) Local Referencing Environment
- (ii) Nonlocal Referencing Environment

- (iii) Global Referencing Environment
- (iv) Predefined Referencing Environment

Referencing Operations : A referencing operation is an operation with the signature
 $\text{ref_op} : \text{id} \times \text{referencing_environment} \rightarrow \text{data_object or subprogram}$

Where ref_op , given an identifier & a referencing environment, finds the appropriate association for that identifier in the environment & returns the associated data object or subprogram definition.

Local_nonlocal, & Global References : A reference to an identifier is a local reference if the referencing operation finds the association in the local environment; it is a non-local or global reference if the association is found non-local or global environment respectively.

Q.3. (a) Write short note on variable size data structures.

Ans. Variable Size Data Structures : Data structure, which cannot be initialized previously i.e., at compile time they are called variable sized data structures.

Variable sized data structures are declared at the run time.

In this type of data structures we use variable data object which are assigned at the run time.

Structures are the collection of homogenous data i.e., different data types under one name.

It means data size can be vary according to the requirements.

Q. 3. (b) Define "arrays" and their implementation in a programming language.

Ans. Array : An array is an aggregate of homogenous data elements which are identified by their position within the aggregate.

An array is characterized by a name, a list of dimensions, a type for its elements & a type & range for its index set. Typically the index set of an array is a consecutive set of integers, but as we shall see this need not necessarily be so. If the index set is the integers, then each dimension is defined by stating an integer lower (lb) & upper (ub) bound which implies that $\text{lb} \leq \text{ub}$ & that $\text{ub} - \text{lb} + 1$ is the size along that dimension. This list of lower & upper bounds is bracketed in some way, typically by square brackets or by parentheses.

In FORTRAN, PL/I or Ada we might write $A(5,3,7)$ to denote the "5,3, 7" element of the three dimensional array whose name is A. In ALGOL 60, ALGOL 68 & Pascal square brackets are used for array dimensions, e.g., $A[5, 3, 7]$. This has the advantage that there is no confusion between an array name & a function call in an expression.

One important language design issue is whether the bounds of each dimension must be constants or can be expressions. In a language such as FORTRAN where storage is allocated at compile time, these bounds must be constant. In ALGOL 60, where storage is allocated dynamically, it was natural to allow the bounds of arrays to be determined at runtime. Thus, the lower & upper bound of a dimension can be any arithmetic expression which evaluates to an integer. The Pascal only allows constant values.

In FORTRAN we must write:

DIMENSION A(100), B(10, 10)

where $A \rightarrow 1D$ & $B \rightarrow 2D$ Array.

In Pascal—Const number of days = 30;

var day : array [1...number of days] of real;

In Ada—

type SEQUENCE is array (Integer range < >) of FLOAT;

type SEQREF is access SEQUENCE;

P : SEQREF;

.....

P : New SEQUENCE (1..... 1000);

Where SEQUENCE \Rightarrow type

SEQREF \Rightarrow pointer

Q. 4. (a) Differentiate between static and dynamic scope with example.

Ans. Difference Between Static Scope and Dynamic Scope : A dynamic scope rule defines the dynamic scope of each association in terms of the dynamic course of program execution. Whereas static scope rule is a rule for determining the static scope of a declaration.

For example (dynamic) a typical dynamic scope rule states that the scope of an association created during an activation of subprogram P includes not only that activation but also any activation of a subprogram called by P, or called by a subprogram called by P & so on. Unless that later subprogram activation defines a new local association for the identifier that hides the original association. With this rule, the dynamic scope of an association is tied to the dynamic chain of subprogram activations.

For example, (static), In Pascal, a static scope rule is used to specify that a reference to a variable X in a subprogram P refers to the declaration of X at the beginning of P, or if not declared there, then to the declaration of X at the beginning of the subprogram Q whose declaration contains the declaration of P & so on.

A language is statically scoped if the body of a procedure is executed in the environment of the procedure's definition. Thus, we can decide at compile time which binding occurrence of an identifier corresponds to a given applied occurrence whereas a language is dynamically scoped if the body of a procedure is executed in the environment of the procedure call. The environment varies from one procedure call to another, so we can't decide unit run-time which binding occurrence of an identifier corresponds to a given applied occurrence.

With static scoping, we can determine the binding occurrence that corresponds to a given applied occurrence of identifier I, just by examining the program text. Whereas, with dynamic scoping, the binding occurrence that corresponds to a given applied occurrence of identifier I depends on the program's dynamic flow of control.

Q. 4. (b) Discuss the various parameters transmission schemes with example.

Ans. Parameter Transmission Schemes :

- (i) Call by name
- (ii) Call by reference
- (iii) Call by value
- (iv) Call by value result

Call By Name : This model of parameter transmission views a subprogram call as a substitution for the entire body of the subprogram. With this interpretation, each formal parameter stands for the actual evaluation of the particular actual parameter. Just as if the actual substitutions were made, each reference to a formal parameter requires a reevaluation of the corresponding actual parameter.

The basic call by name rule may be stated in terms of substitution. The actual parameter is to be substituted every where for the formal parameter in the body of the called program before execution of the subprogram begins.

Call By Reference : To transmit a data object as a call by reference parameter means that a pointer to the location of the data object is made available to the subprogram. At the beginning of execution of subprogram the l-values of actual parameters are used to initialize local storage locations for the formal parameters. The data object does not change position in memory.

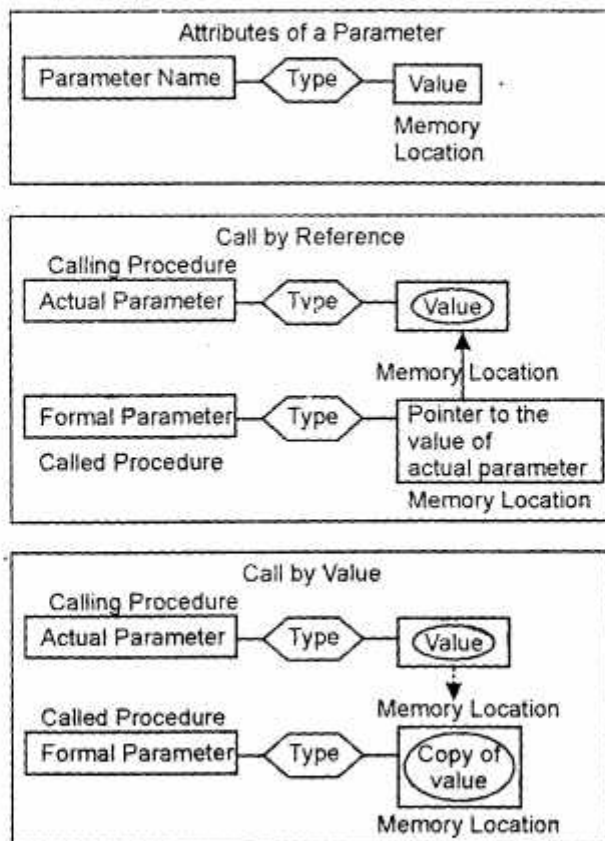
Call By Value : If a parameter is transmitted by value, the value of the actual parameter is passed to the called formal parameter. The implementation is similar to the call by reference model, except that

(i) On invoking a subprogram, a call by reference parameter passes its l-value, whereas a call by value parameter passes its r-value.

(ii) On reference in the subprogram, a call by reference parameter uses the l-value stored in the formal parameter to access the actual data object, whereas in call by value, the formal parameter contains the value that is used.

Call By Value Result : In this, the formal parameter is a local variable (data object) of the same data type as the actual parameter. The value of actual parameter is copied into the formal parameter data object at the time of call so that the effect is same as if an explicit assignment of actual parameter to formal parameter were executed.

Value result parameter transmission developed from ALGOL-W, a language developed by Nicklaus Wirth.



Q.5. (a) Define sequence control. What do you mean by sequence control within expressions?

Ans. Sequence Control : The control of the order of execution of the operations (primitive, user defined).

Sequence Control Structures

- (i) Structures used in expressions (and thus within statements) such as precedence rules, parentheses.
- (ii) Structures used between statements or groups of statements, such as conditional, iteration.
- (iii) Structures used between subprograms such as subprograms calls, co-routines.

Implicit Control Vs. Explicit Control

Implicit Control : Defined by the language & not modified until user redefine it.

Explicit Control : Programmer uses to redefine implicit control sequence (parenthesis)

SEQUENCE WITH ARITHMETIC EXPRESSION

Prefix Notation :

** Operators come first, then the operands

e.g., $(a+b) * (c-a) \Rightarrow * a \text{ } ab-ca$

** No ambiguity & no parentheses needed

** No. of arguments for an operator must be known a priori.

** Relatively easy to decode using stack mechanism.

Postfix Notation :

** An operator follows its operands e.g., $(a+b) * (c-a) \Rightarrow ab + ca - *$

** No ambiguity & no parenthesis needed

** Relatively easy to decode using stack mechanism.

Infix Notation :

** Only suitable for binary operations

** For more than one infix operator, it is inherently ambiguous

** Parentheses is used to explicitly unnotate the order.

Implicit Control Rules :

** Hierarchy of operations (precedence) e.g., Ada

\Rightarrow **, abs, not : Exponential absolute negation

\Rightarrow */mod : Multiplication, division,

\Rightarrow + - : Unary addition & subtraction

\Rightarrow = < > : Relational

\Rightarrow and or xor : Boolean Operation

* Associative :

** left_right associativity (+, -, others)

** right_left associativity (exponential)

** Issues in Evaluating Expressions

** Uniform Evaluation rules (eager & lazy)

Eager \Rightarrow evaluates the operands as soon as they appear.

Lazy \Rightarrow delay evaluation of operands as late as possible.

** Side Effects

$\rightarrow a * \text{foo}(x) + a$; say $a = 1$; $\text{foo}(x)$

generates 3

\rightarrow if each term is evaluated $1 * 3 + 2 = 5$

\rightarrow if a is evaluated only once $1 * 3 + 1 = 4$

\rightarrow if evaluate $\text{foo}(x)$ first $2 * 3 + 2 = 8$

* Error Condition \rightarrow No solution other than exceptional

* Short Circuit Expression \rightarrow Use the characteristics of "and" or "or" operation.

Q. 5. (b) Write short note on :

(i) Recursive subprograms

(ii) Co-routines

Ans. (i) Recursive Subprograms : A recursive subprogram is an independent unit of code that can be used recursively from different parts of a program. In other words, a subprogram is like a function in C & a recursive means a function calling itself.

A recursive subprogram must satisfy the following properties :

- (i) It must not modify any code instructions.
- (ii) It must not modify global data. All variables are stored on the stack.

Advantages to writing recursive code :

- (i) Can be called recursively.
- (ii) Can be shared by multiple processes.
- (iii) Recursive subprograms work much better in multithreaded.

(ii) Co-routines : The co-routine relationship between two procedures is one of mutual control. Procedure P may invoke procedure Q while suspending its processing. At some later point, Q continues P at precisely the point where P invoked Q. Now Q is suspended, but it remembers its state. P & Q may call each other alternately & each time the procedure begins at the place it last was suspended. Only the original task, in this case P, can delete Q & reallocate its resources. A more general situation permits the original procedure to generate many co-routines. The co-routines may all continue each other & the original procedure as well. When a coroutine instance is created, its activation record is placed on the stack. In SIMULA for example, co-routines are executed immediately as they are created while in SL 5 the user must first create the co-routine & later on provide a command which begins its execution.

Q.6. (a) Discuss Heap storage management with examples.

Ans. Heap Storage Management : This is a type of storage management. A heap is a block of storage within which pieces are allocated & freed in some relatively unstructured manner. Here the problems of storage allocation, recovery, compaction & reuse may be severe. There is no single heap storage management technique, but rather a collection of techniques for handling various aspects of managing this memory.

The need for heap storage arises when a language permits storage to be allocated & freed at arbitrary points during program execution, as when a language allows creation, destruction or extension of programmer data structures at an arbitrary point. For example, in ML, two lists may be concatenated to create a new list at any arbitrary point during execution; or the programmer may dynamically define a new type. In LISP, a new element may be added to an existing list structure at any point, again requiring storage to be allocated. In both ML & LISP, storage may also be freed at unpredictable points during execution.

It is convenient to divide heap storage management techniques in 2 categories depending on whether the elements allocated are always of the same fixed size or of variable size. Where fixed-size elements are used, management techniques may be considerably simplified compaction, in particular, is not a problem because all available elements are the same size.

Heap Storage Problems

Get Storage – allocate(x)

Free storage – free(x)

Problems :

Dangling reference

allocate(x) ; y = x;

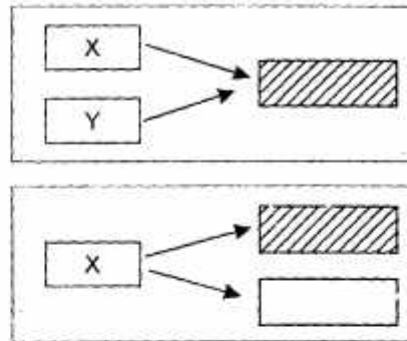
free(x);

⇒ y still points to allocated

Inaccessible storage :

allocate(x);

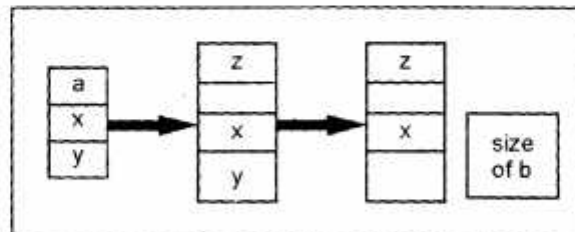

```
allocate(x);  
⇒ first allocation to x now lost.  
Memory fragmentation... (next slide)
```



Memory Fragmentation Occurs with Variable Sized Blocks

Memory fragmentation :

```
allocated(a) ;  
allocated(x);  
allocate(y);  
free(a);  
allocate(z);  
free(y);  
allocate(b);  
⇒ No contiguous space for b
```



Q. 6. (b) Differentiate between system controlled and programmer controlled storage management with suitable example.

Ans. Difference Between System Controlled and Programmer Controlled Storage Management :

The advantage of allowing programmer control of storage management is that it is often extremely difficult for the system to determine when storage may be most effectively allocated & freed.

The difficulty with programmer controlled storage management is twofold : It may place a large & often undesirable burden on the programmer & it may also interfere with the necessary system controlled storage management.

Programmer controlled storage management is dangerous to the programmer because it may lead to subtle errors or loss of access to available storage.

Programmer controlled storage management also may interfere with system controlled storage management in that special storage areas and storage management routines may be required for programmer controlled storage, allowing less efficient use of storage overall.

For example, the programmer can hardly be expected to be concerned with storage for temporaries, subprogram return points or other system data.

At best a programmer might control storage management for local data. Yet even simple allocation & freeing of storage for data structures, as in C, are likely to permit generation of garbage & dangling references.

Q. 7. (a) Compare procedural languages with object oriented programming language using suitable example.

Ans. Procedural languages enforce sequential processing of instructions object oriented languages may implement event driven processing.

Procedural languages store all data as global while OPPs language support data encapsulation—all related data is stored inside one object & only relevant data is shown to the user.

Facilities like function overloading & operator overloading allow you to use same names & provide different functionality which avoids personalism in naming conventions. These overload versions are easy to use & remember.

Object oriented programming is a methodology for modelling the real world or at least the problem being solved, by decomposing the problem into smaller discrete pieces called objects. Whereas procedural language is a methodology for modelling the real world or the problem being solved, by determining the steps & the order of those steps that must be followed in order to reach a desired outcome or specific program state.

Procedural language is dependent upon the procedures whereas the object oriented programming language is dependent upon the objects & classes.

Object oriented language follows OOPs concept whereas procedural language does not follows OOPs concept.

Q.7. (b) Write short note on declaration and type checking of data structure.

Ans. Declaration and Type Checking of Data Structure : The basic concepts & concerns surrounding declarations & type checking for data structures are similar to those discussed for elementary data objects. However, structures are ordinarily more complex because there are more attributes to specify. For example, the C declaration `float A[20];` at the beginning of a subprogram P specifies the following attributes of Array A :

- (i) Data type is an array.
- (ii) No. of dimensions is one.
- (iii) No. of components is 20.
- (iv) Subscripts naming the rows are the integers from 0 to 19.
- (v) Data type of each component is float.

Declaration of these attributes allows a sequential storage representation for A & the appropriate accessing formula for selecting any component `A[I]` of A to be determined at compile time, despite that A is not created until entry to subprogram P at runtime. Without the declaration, the attributes of A would have to be determined dynamically at run time, with the result that the storage representation & component accessing would be much less efficient.

Type checking is somewhat more complex for data structures because component selection operations must be taken into account. There are two main problems :

(i) Existence of a Selected Component : The arguments to a selection operation may be of the right-types, but the component designated may not exist in the data structure.

(ii) **Type of a Selected Component** : A selection sequence may define a complex path through a data structure to the desired component.

To perform static type checking it must be possible to determine at compile time the type of component selected by any valid composite, selector of this sort.

Q. 8. Write short notes on the following :

- (a) Scalar data types
- (b) Implicit and explicit sequence control
- (c) Information Hiding
- (d) Exception and exception handlers

Ans. (a) Scalar Data Types : These are the data types available in the openCL C programming language used to create kernels that are executed on openCL device(s). The openCL C programming language is based on the ISO/IEC 9899 : 1999 C language specification with specific extensions & restrictions must built in scalar data types are also declared as appropriate types in the openCL API (& header files) that can be used by an application.

Type in Open CL Language	Description
Bool	A conditional data type which is either true or false
Char	A signed two's compliment—8-bit integer
Unsigned Char	
An unsigned 8 bit integer short	A signed two's compliment 16-bit integer
Unsigned short,	
An Unsigned 16 bit integer	
int	Signed 2's compliment 32-bit
unsigned int	
An unsigned 32 bit integer	
long	64-bit integer
unsigned long, float etc.	

(b) Implicit and Explicit Sequence Control :

Implicit sequence control	—	Defined by the language & not modified until user redefine it.
Explicit sequence control	—	Programmer uses to redefine implicit control sequence (parentheses)

**** Sequence Control**

⇒ Expressions	* Precedence rules (Priority)
	* Associativity (L-R, R-L)
⇒ Statements	* Sequence (One Way)
	* Conditionals (Two Way)
	* Iterations (Repetition)
⇒ Subprograms	(Functions or Modular Programming)
⇒ Declarative Programming	
*Functional	
*Logic Programming	

(c) Information Hiding : Information hiding in computer science is the principle of segregation of design decisions in a computer that are most likely to change, thus, protecting other parts of the problem from extensive modification if the design decision is changed. The protection involves providing a stable interface

which protects the remainder of the program from the implementation (the details that are most likely to change.)

The concept of information hiding was first dominated in a paper by David Parnas, "On the criteria to be used in decomposing systems into modules." Published in the communications of the ACM in December 1972. Before then, modularity was discussed by Richard Gauthier & Stephen Pont in their 1970 book titled Designing Systems Programs although modular programming itself had been used at many commercial sites for many years previously, especially in I/O subsystems & software libraries—without acquiring the 'information hiding' tag but for similar reasons.

(d) Exception and Exception Handlers : Exceptions provide a way to react to exceptional circumstances in our program by transferring control to special functions called handlers.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a try block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally & all handlers are ignored. An exception is thrown by throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block. A throw expression accepts one parameter, which is passed as an argument to the exceptional handler.

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

          int main() {
            try { throw 20; }
            catch(int e) {
              cout << "An exception" << e << endl;
            } return 0;}
```

Output— An exception 20