Yogesh Agrawal

UNIT – 4

Storage Management

Introduction → Storage Management is the act of managing computer memory. In its simpler forms, this involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed. The management of main memory is critical to the computer system.

The principal Goals of the System's Storage Management are:

- To provide memory space to enable several processes to be executed at the same time.
- To provide a satisfactory level of performance for the system users.
- To protect each program resources
- To share (if desired) memory space b/w processes
- To make the addressing of memory space as transparent as possible for the programmer.
- To provide memory space to store parameters, return address etc.

Elements requiring Storage :→ The major program and data elements requiring storage during program execution are:

1. **Code Segments for Translated User programs** :– A major block of storage in any system must be allocated to store the code segment representing the translated form of user programs.
2. **System Runtime Programs** – Another substantial block of storage during execution must be allocated to system programs that support the execution of the user programs. These may range from simple library routines, such as sine, cosine, or print string functions, to software interpreters or translators present during execution.

(3) User defined data structures & constants → Space for user data must be allocated for all data structures including constants

(4) Subprogram Return points → Subprograms have the property that they may be invoked from different points in a program. Therefore, internally generated sequence-control information, such as subprogram return points, coroutine resume points, must be allocated storage.

(5) Referencing Environments → Storage of referencing environments (identifier associations) during execution may require substantial space.

(6) Temporaries in Expression Evaluation → Expression Evaluation requires the use of system-defined temporary space for the intermediate results of evaluation. For eg, in evaluation of the expression $(x+y) \times (u+v)$, the result of the first addition may have to be stored in a temporary while the second addition is performed.

(7) Temporaries in parameter transmission → when a subprogram is called, a list of actual parameters must be evaluated & the resulting values stored in temporary storage until evaluation of the entire list is complete.

(8) Input-Output buffers → Ordinarily input & output operations work through buffers, which serve as temporary storage areas where data are stored b/w the time of the actual physical transfer of the data to and from external storage devices.

(9) Miscellaneous System data → In almost every language implementation, storage is required for various system data: tables, status information for input-output, & various miscellaneous pieces of state information (eg. reference counts or garbage-collection bits).

Besides the data & program elements requiring storage, various operations may require storage to be allocated or freed. The following are the major operations:

- Subprogram call & return operations - The allocation of storage for a subprogram activation record, the local referencing environment, and other data on call of a subprogram is often the major operation requiring storage allocation. The execution of a subprogram return operation usually requires freeing of the storage allocated during the call.
- Data structure creation & destruction operations - If the language provides operations that allow new data structures to be created at arbitrary points during program execution (eg. new in Java) than these operations ordinarily require storage allocation that is separate from that allocated on subprogram entry. The language must also provide an explicit destruction operation (such as free in C), which may free the storage space.
- Component Insertion and deletion operations - If the language provides operations that insert & delete components of data structures, storage allocation & freeing may be required to implement these operations (eg: the perl push function adds an element to an array)

Programmer & System Controlled Storage

C has become very popular because it allows extensive programmer control over storage via malloc & free, which allocate & free storage for programmer-defined data structures. On the other hand, many high-level languages allow the programmer ~~to do the same~~ no direct control. Storage management is affected only implicitly through the use of various language features.

Programmer Controlled Storage Management	System controlled Storage Management.
1) In this, programmer explicitly control over storage by a malloc and free.	In this, storage is controlled by the system implicitly itself.
2) It may place a large & often undesirable burden on the programmer.	In this, there is no burden over programmer. for eg, the programmer can hardly be expected to be concerned with storage for temporaries, subprogram return points, or other system data.
) Yet even simple allocation & freeing of storage for data structures, as in C, are likely to permit generation of garbage & dangling references. Thus, programmer-controlled storage is dangerous to the programmer because it may lead to subtle errors or loss of access to available storage.	No such error happens.
) Programmer may interfere with the necessary system controlled storage management.	No such interference of programmer is allowed.
) The programmer knows quite precisely when a particular data structure is needed or when it is no longer needed & may be freed.	It is difficult for the system to determine when storage may be most efficiently allocated & freed.

Storage Management Phases :→ The three basic aspects of storage management are:

① Initial Allocation :→ At the start of execution, each piece of storage may either be allocated for some use or free. If free initially, it is available for allocation dynamically as execution proceeds. Any storage management system requires some technique for keeping track of free storage as well as mechanisms for allocation of free storage as the need arises during execution.

② Recovery :- Storage that has been allocated and used, and that subsequently becomes available, must be recovered by the storage manager for reuse.

③ Compaction & reuse - Storage recovered may be immediately ready for reuse or compaction may be necessary to construct large blocks of free storage from small pieces.

Static Storage Management :→ The simplest form of allocation is static allocation — allocation during translation that remains fixed throughout execution.

- Storage for all variables allocated in a static block.
- allocation can be done by the translator.
- Storage for the code segments of user & system programs is allocated statically. eg :- I/O buffers & various system data.
- Static allocation requires no run time storage management software & of course, there is no concern for recovery & reuse.
- In Fortran, all storage is allocated statically
- Each subprogram is compiled separately, with the compiler setting up the code segment containing the compiled program, its data areas, temporaries, return-point location.
- The loader allocates space in memory for these compiled blocks at load time, as well as space for system run time routines.

- During program execution, no storage management need to take place.
- This allocation is efficient because no time or space is expended for storage mgmt during execution. The translator can generate the direct l-value addresses for all data items.
- It is incompatible with recursive subprogram calls, with data structures whose size is dependent on computed or input data, & with many other desirable language features.
- FORTRAN & COBOL use static allocation.

STACK BASED STORAGE MANAGEMENT

Almost all computer runtime memory environments use a special stack (the "call stack") to hold information about procedure/function calling & nesting in order to switch to the context of the called function & restore to the caller function when the calling finishes.

- They are important way of supporting nested or recursive function calls.
- This type of stack is used implicitly by the compiler to support CALL and RETURN statements & is not directly manipulated by the programmer.
- C language use the stack to store data that is local to a procedure. Space for local data items is allocated from the stack when the procedure is entered, & is deallocated when the procedure exists.
- Because the data is added & removed in a LIFO manner, stack allocation is very simple & typically faster than heap allocation

- The advantage is that memory on the stack is automatically reclaimed when the function exists, which can be convenient for the programmer.
- Allocating more memory on the stack than is available can result in a crash due to stack overflow.
- The disadvantage is that the memory stored on the stack is automatically deallocated when the function that created it returns, & thus the function must copy the data if they should be available to other parts of the program after it returns.

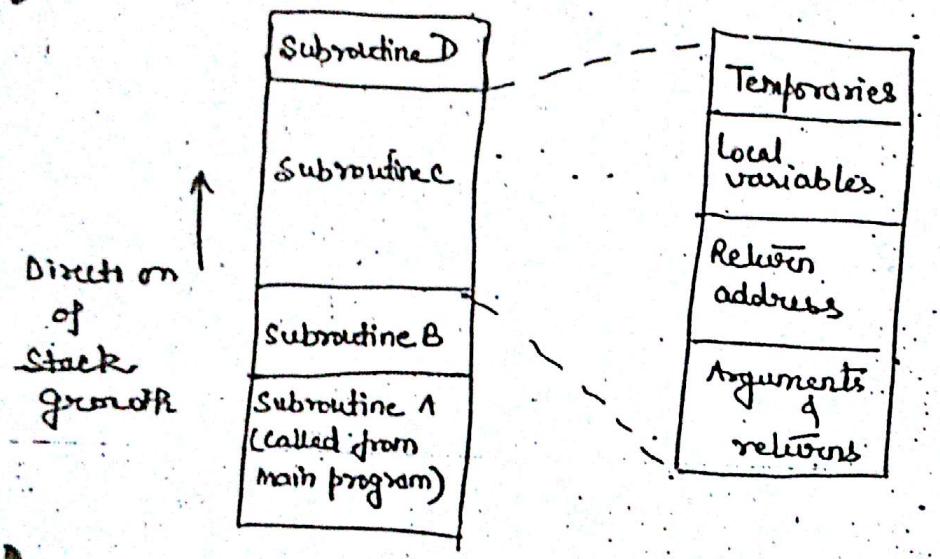


Fig → Representation of Stack based Storage Management

HEAP 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 management 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 arbitrary program points
- The Heap storage management techniques is divided into two categories, depending on whether the elements allocated are always of the same fixed size or of variable size.

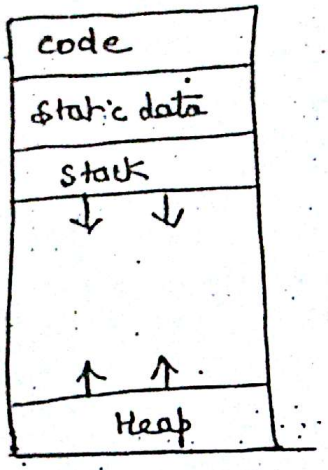


Fig → : Runtime storage in heap & stack.

- It generally used as storage for objects with an unrestricted lifetime
- Maintains a freelist — list of free space
- On allocation, memory manager finds space & marks it as used
- On deallocation, memory manager marks space as free
- Memory fragmentation — the memory fragments into small blocks over the lifetime of program
- Garbage collection — coalesce fragments, possibly moving objects

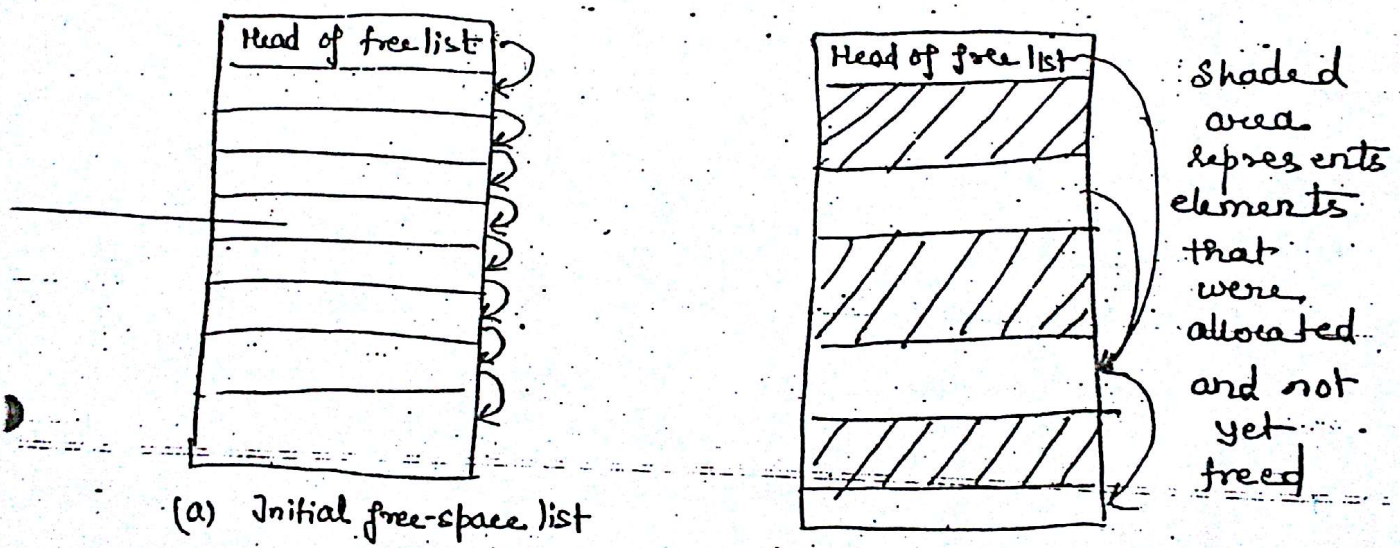
Q. Discuss Heap storage management

Heap storage Management for fixed size Elements

(5)

Here the heap occupies a contiguous block of memory. Whenever an element is needed, one of these blocks is allocated from the heap. Whenever an element is freed, it must be one of these original heap elements

Here the elements, initially are linked together to form a free-space list. To allocate an element, the first element on the free-space list is removed from the list & a pointer to it is returned to the operation requesting the storage. When an element is freed, it is simply linked back in at the head of the free-space list. fig shows such an initial free list, as well as the list after allocation and freeing of a No: of elements



(a) Initial free-space list

(b) Free space after execution

fig :- free space list structure

Recovery :→ In the process of identification and recovery of freed storage elements in fixed size heap storage, many problems occur. Those problems can be solved in the following way:

① Explicit return by programmer or system :- Here when an element that has been in use becomes available for use, it must be explicitly identified as free & returned to the free-space list. But it is not always feasible because of the following two reasons :

(a) Dangling References :- In context of heap storage management, a dynamic reference is a pointer to an element that has been returned to free space list means 'dangling pointers arise when an object is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. As the system may reallocate the previously freed memory to another process, if the original program then dereferences the dangling pointers, unpredictable behavior may result, as the memory may now contain completely different data.

Eg :→ int *p, *q; /* p and q are pointers to integers */

p = malloc (sizeof (int)); /* allocate an int and set l-value in p */

q = p; /* save l-value of allocated space in q */

free (p); /* dangling reference still in q */

(b) Garbage :- In opposite to dangling references discussed above, if the last access path to a structure is destroyed without the structure itself being destroyed & the storage recovered, then the structure becomes garbage.

A garbage element is one that is available for reuse but not on the free-space list, & thus it has become inaccessible. If garbage accumulates, available storage is gradually reduced until the program may be unable to continue for lack of known free space.

```
int *p, *q;          /* p and q are pointers
                        to integers */
p = malloc(sizeof(int));   /* allocate an int &
                              set lvalue in p */
p = q;
                     /* l-value previously in
                        p now lost */
```

Alternative solution which solves the problems (dangling references & garbage) due to explicit return by programmer or system are reference count & garbage collection.

(2) Reference count :- The most straightforward way to recognize garbage & make its space reusable for new cells is to use reference counts. Within each element in the heap some extra space is provided for a reference counter. The reference counter contains the reference count indicating the No: of pointers to that element that exist. When an element is initially allocated from the free-space list, its reference count is set to 1. Each time a new pointer to the element is created, its reference count is increased by 1. Each time a pointer is destroyed, the reference count is decreased by 1. When the reference count of an element reaches zero, the element is free & may be returned to the free-space list.

- Reference counts allow both garbage & dangling references to be avoided in most situations.

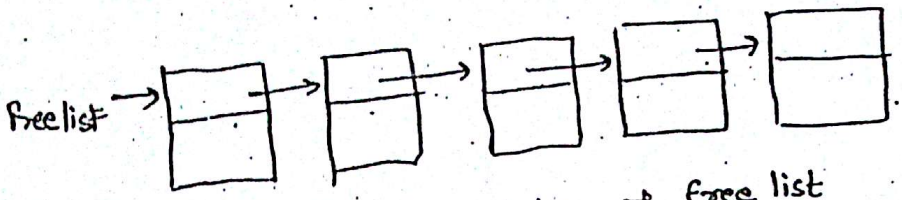


fig: Representation of free list

- Where the programmer is allowed an explicit free or erase statement, reference counts also provide protection. A free statement decrements the reference count of the structure by 1. Only if the count then is zero is the structure actually returned to the free-space list. A non-zero reference count indicates that the structure is still accessible & the free command should be ignored.

Advantages
① conceptually simple
② Immediate reclamation of storage.

Disadvantages
① Extra space for storing reference counter.
② Extra time (every pointer assignment has to change/check count).
③ can't collect "cyclic garbage".
④ cost of maintaining reference counter (In terms of storage)
⑤ Decrease in execution efficiency because testing, incrementing & decrementing occur continously through out execution.

8) Garbage Collection :- The basic philosophy behind garbage collection is simply to allow garbage to be generated to avoid dangling references. When the free-space list is entirely exhausted & more storage is needed; the computation is suspended temporarily and an extraordinary procedure instituted - a garbage collection - which identifies garbage elements in the heap & returns them to the free-space list. The original computation is then resumed, and garbage again accumulates until the free-space list is exhausted; at which time another garbage collection is initiated, & so on.

Because garbage collection is done only rarely (when the free-space list is exhausted), it is allowable for the procedure to be fairly costly. Two stages are involved :

(a) Mark :- In the first stage, each element in the heap that is active (i.e that is part of an accessible data structure) must be marked. Each element must contain a garbage-collection bit set initially to on. The marking algorithm sets the garbage-collection bit of each active element off.

(b) Sweep :- Once the marking algorithm has marked active elements, all those remaining whose garbage collection bit is on are garbage and may be returned to the free-space list. A simple sequential scan of the heap is sufficient. The garbage collection bit of each element is checked as it is encountered in the scan. If off, the element is passed over; if on, the element is linked into the free-space list. All garbage-collection bits are reset to on during the scan (to prepare for a later garbage collection)

Advantages :→ Garbage collection frees the programmer from manually dealing with memory allocation & deallocation. As a result, certain categories of bugs are eliminated or substantially reduced.

Disadvantages → It adds an overhead that can affect program performance. This activity will likely require more CPU time than would have been required if the program explicitly freed unnecessary memory.

Q Discuss heap storage management for variable size elements

Heap storage Management for variable size Elements

Heap storage management - where the programmer controls the allocation & recovery of variable-size elements — is more difficult than with fixed-size elements. Variable-size elements arise in many situations. for eg → if space is being used for programmer-defined data structures stored sequentially, such as arrays, then variable-size blocks of space are required or activation records for tasks might be located in a heap in sequential blocks of varying sizes.

Phases of variable size Elements Heap storage Management

① Initial allocation & reuse :- In variable size elements heap storage management, we maintain free space in blocks of as large a size as possible. Hence, we consider the heap as simply one large block of free storage. A heap pointer is appropriate for initial allocation. When a block of N words is requested, the heap pointer is

advanced by N & the original heap pointer value returned as a pointer to the newly allocated element. As storage is freed behind the advancing heap pointer, it may be collected into a free-space list. Eventually the heap pointer reaches the end of the heap block. Some of the free space back in the heap must now be reused. Two possibilities for reuse are:

(a) Use the free-space list for allocation, searching the list for an appropriate size block & returning any leftover space to the free list after the allocation

(b) Compact the free space by moving all the active elements to one end of the heap, leaving the free space as a single block at the end & resetting the heap pointer to the beginning of this block.

Let us look at these two possibilities in turn

Reuse directly from a Free-space List

The simplest approach, when a request for an N-word element is received, is to scan the free-space list for a block of N or more words. A block of N words can allocated directly. A block of more than N words must be split into two blocks, an N-word block, which is immediately allocated, & the remainder block, which is returned to the free-space list. A No: of particular techniques for managing allocation directly from such a free-space list are used:

(a) First-fit method :- When an N-word block is needed, the free-space list is scanned for the first block of N or more words, which is then split into an N-word block, & the remainder, which is returned to the free-space list.

(b) Best fit Method :- when an N-word block is needed, the free space list is scanned for the block with the minimum No. of words greater than or equal to N. This block is allocated as a unit, if it has exactly N words, or is split & the remainder returned to the free-space.

Advantage :→ By keeping elements in the free space list in size order, it makes allocation fairly efficient. You only have to scan the list until you find the appropriate size needed.

DisAdvantage → There is a corresponding cost of adding entries into the free-space list by having to search the list looking for the appropriate place to add the new entry.

2. Recovery with Variable-Size Blocks :→

Explicit return of freed space to a free-space list is similar to fixed size elements. The problems of garbage and dangling references are also present but dealt in different manner. (Reference counts may be used in the similar manner)

(Garbage Collection is also similar but sweep phase of fixed size element is replaced by collecting phase). In sweep phase of fixed size elements, we collected by a simple sequential scan of memory, testing each elements garbage collection bit; (If the bit was "on", the element was returned to the free space list; if "off", it

was still active and was passed over). All these steps are same in collecting phase also but now there is a problem in determining the boundaries b/w the elements. where does one element end & the next begin ? without this information the garbage cannot be collected.

The simplest solution is to maintain along with the garbage-collection bit in the first word of each block, active or not, an integer length indicator specifying the length of the block. With the explicit length indicators present, a sequential scan of memory is again possible, looking only at the first word of each block. During this scan, adjacent free blocks may also be compacted into single blocks before being returned to the free-space list, thus eliminating the partial-compaction problem. Garbage collection may also be effectively combined with full compaction to eliminate the need for a free-space list altogether.

(3.) Compaction and the Memory Management Problem

The problem that any heap storage management system using variable size elements faces is that of memory fragmentation. One begins with a single large block of free space. As computation proceeds, this block is progressively fragmented into smaller pieces through allocation, recovery & reuse. If only the simple first-fit or best-fit allocation technique of used, it is apparent that free space blocks continue to split into ever smaller pieces. Ultimately one reaches a point where a storage allocator cannot honor a request for a block of N words because

no sufficiently large block exists, even though the free-space lists contains in total far more than N words) without some compaction of free blocks into larger blocks, execution will be halted by a lack of free storage faster than necessary. Depending on whether active blocks within the Heap may be shifted in position, (one of two approaches to compaction possible :

(A) Partial compaction :- If active blocks cannot be shifted (or if it is too expensive to do so), then only adjacent free blocks on the free-space list may be compacted.

(B) Full compaction :- If active blocks can be shifted, then all active blocks may be shifted to one end of the Heap, leaving all free space at the other in a contiguous block. Full compaction requires that when an active block is shifted, all pointers to that block be modified to point to the new location.

Procedural Language

- Procedural programming specifies a list of operations that the program must complete to reach the desired state.
- Each program has a starting state, a list of operations to complete, and an ending point
- This approach is known as imperative programming
- Procedures, also known as routines, subroutines, methods or functions simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.
- A procedural programming language provides a programmer a means to define precisely each step in the performance of a task. The programmer knows what is to be accomplished & provides through the language step-by-step instructions on how the task is to be done.
- By splitting the programmatic tasks into small pieces, procedural programming allows a section of code to be reused in the program without making multiple copies.
- It also makes it easier for programmers to understand and maintain program structure.
- Eg. of procedural programming languages are FORTRAN, C AND BASIC.

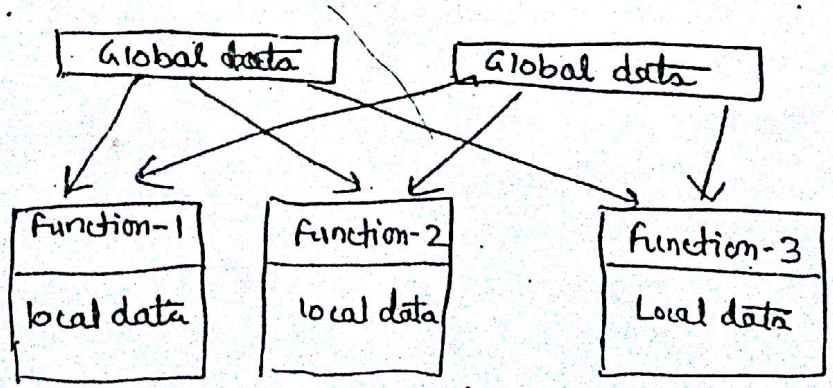
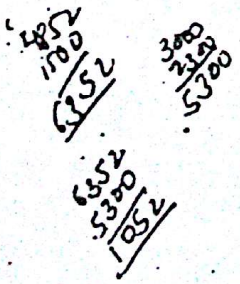


Fig :- Relationship of data & functions in Procedural programming

- Here Emphasis is on doing things (algorithms)
- Larger programs are divided into smaller programs known as functions
- Most of the functions share global data
- Data move openly around the system from function to function
- An easier way to keep track of program flow than a collection of "GOTO" or "JUMP" statements.
- The ability to be strongly modular or structured.
- The main drawback of procedural programming is that it breaks down when the problems get very large. There are limits to the amount of detail one can cope with.
- Another serious drawback is that it doesn't model real world problem very well. This is because functions are action-oriented & don't really corresponds to the element of the problem.



Structured Programming Languages

- Structured programming is a special type of procedural programming.
- It provides additional tools to manage the problems that larger programs were creating.
- It requires that programmers break program structure into small piece of code that are easily understood.
- It also frowns upon the use of global variables & instead uses variables local to each subroutine.
- One of the well known features of structured programming is that it does not allow the use of GOTO statement.
- (It is often associated with "top-down" approach to design.) The top down approach begins with an initial overview of the system that contains minimal details about the different parts. Subsequent design iterations then add increasing detail to the components until the design is complete.
- Structured programming is written with only the following code structures :
  - Sequence of sequentially executed statements
  - Conditional execution of statements
  - Looping
  - Structured subroutine calls (eg. 'gosub' but not goto)
  - Stepwise Refinement

The following concepts are forbidden :

- Goto Statements
- Break & continue out of the middle of loops
- Multiple exit points to a function / procedure / subroutine (multiple "return" statements)
- Multiple entry points to a function / Procedure / subroutine

- Structured programming is more flexible than procedural programming. It is designed for programmer flexibility.
- Such languages have activation records but employ scope rules & nested block structure for storing and accessing data.
- (Parameters are transmitted either by value or reference). Subprograms may be passed as parameters.
- (During translation, static type checking for almost all operations is possible so that little dynamic checking is required)
- During execution, a central stack is used for subprogram activation records, with a heap storage area for data objects created directly for use with pointer variables, and a static area for subprogram code segments & runtime support routines.
- (Ada, pascal, Algol) and PL/I are structured programming languages.

Ques: Explain the concept of procedural & structured programming.

Non Procedural Progeamming

- It is one in which we specifies WHAT needs to be computed but not HOW it is to be done. That is one specifies :
  - the set of objects involved in the computation
  - the relationships which hold between them
  - the constraints which must hold for the problem to be solved.

and leaves it up the language interpreter or compiler to decide How to satisfy the constraints.

Programs in such languages do not state exactly how a result is to be computed but rather describe the form of the result. The difference is that we assume the computer system can somehow determine how the result is to be gotten.

- They are also known as fourth generation languages
- In Non procedural languages, the order of program instructions is not important. The importance is given only to "what is to be done".
- With Non procedural Language, the user (programmer) writes English like instructions to retreive data from databases using non-procedural Languages.
- These languages provide the user-friendly program development tools to write instructions. The programmers have to not spent much time for coding the program.
- Eg. are SQL & RPG

SQL stands for Structured Query Language. SQL is very popular database access language & is specially used to access & to manipulate the data of databases. SQL can be used to create tables, add data, delete data, update data of database tables etc

RPG stands for Report Program Generator. This language was introduced by IBM to generate business reports.

Applications of Non Procedural Programming.

① Relational Data base Management Systems :- RDBMS store data in the form of tables. Queries on such databases are often stated in relational calculus, which is a form of symbolic logic. The query languages of these systems are nonprocedural in the same sense that logic programming is nonprocedural. The user does not describe how to retrieve the answer; rather, He or She only describes the characteristics of the answer. Logic programming thus a natural match to the needs of implementing an RDBMS

② Expert System → Prolog can and has been used to construct expert systems. It can easily fulfill the basic needs of expert systems, using resolution as the basis for query processing, using its ability to add facts & rules to provide the learning capability, & using its trace facility to inform the user of the reasoning behind a given result

③ Natural language Processing → Certain kinds of Natural-language processing can be done with logic programming. In particular, natural language interfaces to computer-software systems, such as intelligent databases & other intelligent knowledge-based systems can be conveniently done with logic programming.

④ Education → Researchers claim a Number of advantages in teaching prolog to young people. First, It is possible to introduce computing using this approach. It also has the side effect of teaching logic, which can result in clearer thinking & expression. This can help students learning a variety of subjects, such as solving equations in mathematics, dealing with grammars for natural languages, & understanding the rule & order of the physical world.

Functional Programming

1, 4, 6, 7

- Functional programming is so called because a program consists entirely of functions.
- The main program itself is written as a function which receives the program's input as its argument & delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives.
- Functional programming is based on expression rather than sequence of statement.

In an Imperative language, an expression is evaluated & the result is stored in a memory location which is represented as a variable in program. A purely functional programming language does not use variable or assignment statements, This frees the programmer from concerns about the memory cells of the computer on which the program is executed without variables, iterative constructs are not possible, for they are controlled by variables.

- Repetition must be done by recursion.

Example → Consider the task of calculating the sum of Integers from 1 to 10.

Imperative language like C will express it as:

```
total = 0 ;
for (i=1 ; i<=10 ; i++)
    total += i ;
```

In functional language, the same program would be expressed as

```
Sum [1..10]
```

Here [1..10] can be used to calculate the sum of 10 numbers using sum function. Hence Functional language is a expression based language.

- Functional programming is highly abstract
- The resultant code is shorter than imperative Language.
- The execution of a function always produce the same result, when given the same parameters. This is called referential transparency.

  eg → F(3 * n + 1) if n is odd, = 1, 3 . .
- Even though there are no variables, there are identifiers bound to values, just as n is used in previous example.
- Here Heap storage & list structures become the natural process of storage management rather than the traditional activation record mechanism in procedural languages.
- Eg. of functional language are LISP, PROLOG & ML
- Since functional programs contain no assignment statements, so given a value, never change
- The time it takes to develop code, & even more importantly, to modify programs, is substantially faster for functional programs than for procedural programs.
- An interesting property of functional languages is that they can be reasoned about mathematically. Since a functional language is simply an implementation of a formal system, all mathematical operations that could be done on paper still apply to the ~~papers~~ programs written in that language.
- Functional programming focus on 'what' rather than 'how'

Some of the Concepts of functional Programming are:

① Higher-order functions :→ Functions are higher-order when they can take other functions as arguments and return them as results. It is a technique in which a function is applied to its arguments one at a time, with each application returning a new (higher-order) function that accepts the next argument.

② Pure functions – functional programming involves writing programs using pure functions. Unlike conventional programming side-effects are completely eradicated. This allows a very clean, very-high level, very concise programming model, which is also:

- quick to write.
- easy to reason about
- easy to modify/debug
- Easy to parallelize

③ Recursion → In recursion, function calls itself again & again until some specified condition is satisfied.

Recursion may require maintaining a stack.

④ Strict, Non-strict & lazy Evaluation

Functional languages can be categorized by whether they use strict or non-strict evaluation, concepts that refer to how function arguments are processed when an expression is being evaluated. To illustrate, consider the following two functions f & g:

$$f(x) := x^2 + x + 1$$

$$g(x, y) := x + y$$

The following expression can be evaluated in one of two ways.

f(g(1,4))

Evaluate the innermost function g first :

f(g(1,4)) → f(1+4) → f(5) → 5^2 + 5 + 1 → 31

Or Evaluate the outermost function f first :

f(g(1,4)) → g(1,4)^2 + g(1,4) + 1

→ (1+4)^2 + (1+4) + 1 → 5^2 + 5 + 1 → 31

The first case is an instance of strict evaluation : arguments to a function are evaluated before the function call; while the second case is an instance of non-strict evaluation where arguments are passed to the function unevaluated & the calling function determines when the arguments are to be evaluated.

Strict evaluation has efficiency advantages. An argument is evaluated once with strict evaluation, while it may be evaluated multiple times with non-strict evaluation, as can be seen in the above example where g(1,4) is evaluated twice. Also, strict evaluation is easier to implement since the arguments passed to a function are data values, whereas with non-strict evaluation arguments may be expressions, requiring some notion of closure.

Object-Oriented programming

- In object-oriented programs, the designer specifies both the data structures & the type of operations that can be applied to those data structures. This pairing of a piece of data with the operations that can be performed on it is known as an object.
- The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development & does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions.
- Here Emphasis is on data, rather than procedure.
- Data is hidden & cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data & functions can be easily added whenever necessary.

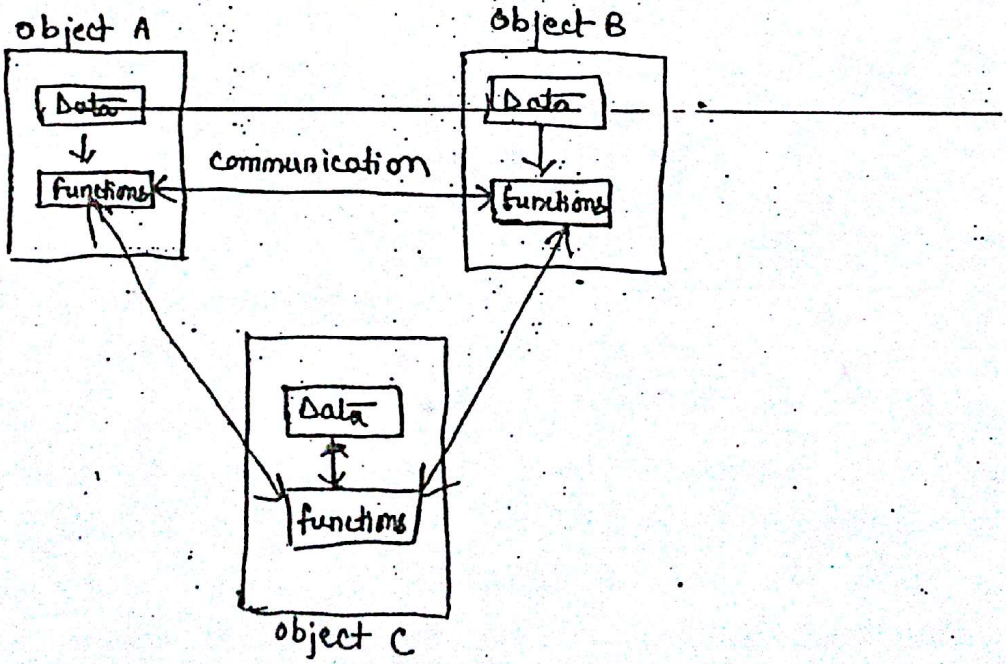


fig → Organization of data & functions in OOP

Explain various features of OOPS.

(12)

Features of OOPS

→ **Class** → A class is a collection of variables & methods. These methods can be accessible to all other classes (public methods) or can have restricted access (private methods). New classes can be derived from a parent class. So a class defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) & the thing's behaviors (the thing it can do or methods, operations or features) foreg, the class Dog would consist of traits shared by all dogs such as breed & fur color (characteristics), & the ability to bark and sit (behaviors)

Object → objects are basic run-time entities in an object-oriented system : objects take up space in the memory. When a program is executed, the objects interact by sending messages to one another.

Data Encapsulation → The wrapping up of data & functions into a single unit (called class) is known as encapsulation. Data Encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data & the program. Encapsulation refers to how the implementation details of a particular class are hidden from all objects outside of the class.

Data Abstraction → Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction & are defined as a list of abstract attributes such as size, weight & cost, & functions to operate on these attributes. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

Polymorphism → Poly means many & morphism means forms, so one thing in many form is known as polymorphism. In OOPS, polymorphism is of two types

- Compile time or static polymorphism
- Runtime or Dynamic polymorphism.

Compile time polymorphism is implemented with

- Function overloading
- Operator overloading

In function overloading, function name is same but No: of arguments & type of arguments varies (different signatures.) for eg.

```
class Test
{
    Void fun()      // fun with No arguments
    {
    }
    void fun(int a)     // fun with one argument of
    {                       Integer type
    }
    void fun(int a, float b)  // fun with one argument of
    {                             Integer & other of
    }                             float type
}
```

Fig → Program to show function overloading

In operator overloading, operators are overloaded with extra meanings, so that they can be used with objects. In normal, operators are used with operands only for example

a+b ('+' operator used to add two operands 'a' & 'b')
a*b ('*' " " " multiply two values)

With the help of operator overloading operator can be used to work with objects also. for example,

ob1 + ob2 (If '+' operator is overloaded, it can add two objects)

ob1 * ob2 (If '*' operator is overloaded, it can multiply two objects)

Run time polymorphism can be implemented with the help of virtual functions in C++.

Runtime Polymorphism → Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at runtime.

```
#include<iostream.h>
Class Base
{ public:
        virtual void show (void)
          { cout << endl << "in base"; }
};
class der1 : public Base
{ public:
        void show ()
          { cout << endl<< "in derived-1" ; }
};
class der2 : public base
{ public:
        Void show (void)
          { cout << endl << "in derived-2"; }
};
int main()
{   base *bp ;    // base class pointer
    base bo; der1 do1;  der2 do2 ;   //objects
    bp = &bo ;  // pointer pointing to objects of base class
    bp -> show() ; // show() of base called
    bp = &do1 ;  // pointer pointing to objects of der1 class
    bp -> show() ; // show() of der1 called
    bp = &do2 ;  // pointer pointing to objects of der2 class
    bp -> show() ;  // show() of der2 called
    return 0;
}
```

Inheritance :→ Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. In OOPs, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

Comparison of C & C++ Imp Explain comparision b/w C and C++

1. C is a procedural language while C++ is a object-oriented language.
2. In C, there's only one major memory allocation function - malloc. You use it to allocate both single elements & arrays & you always release the memory in the same way. While In C++, however, memory allocation for arrays is somewhat different than for single objects; you use the new[] operator, and you must match calls to new[] with calls to delete[].
3. Inline functions are not available in C while Inline functions are available in C++
4. C doesnot have reference variables & namespaces while C++ does have reference variables & namespaces
5. You cannot overload a function in C while you can overload a function in C++
6. In C++ you are free to leave off the statement 'return 0' at the end of main; it will be provided automatically but in 'C' you must manually add it.
7. In C variable cannot be initialized at run time while In C++ Variables can be initialized at run time.
8. In 'C', global version of a variable cannot be accessed from within the inner block while In C++, this problem can be resolved by introducing a new operator :: called scope resolution operator.

(9) In C we use #include <stdio.h> as inclusion file while in C++, we use #include <iostream.h> as inclusion file.

(10) C gives importance to procedure that is functions rather than data while C++ gives importance to object that is data.

(11) C++ supports polymorphism, inheritance, but in C, these features are not supported.

(12) In C++, function prototying is compulsory if the definition is not placed before the function call whereas in C, it is optional.

(13) In case of C++, functions can be used inside a structure while structures cannot contain functions in C.

(14) C uses printf and scanf functions whereas C++ uses cin>> and cout<< as input & output functions.