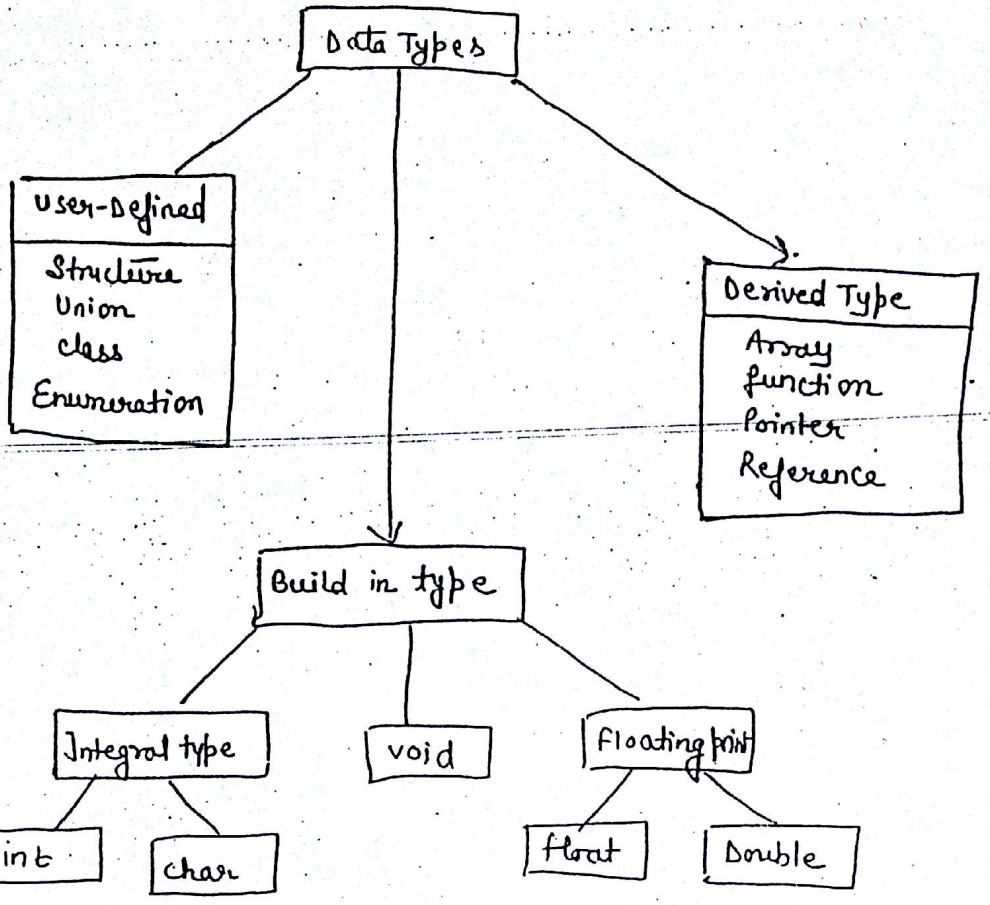
Unit-2 — Yogesh Agrawal (1) (1)

Evolution of DATA TYPES :- Data types are classified under various categories as shown in fig:



→ Build in datatypes are also known as basic Data types
→ The Basic datatypes can have various modifiers preceding them such as signed, unsigned, long & short.

Abstraction :→ Abstraction is used for hiding the unwanted information & giving revelant information.

Eg → Three set of customers are going to buy a bike. first one wants information about the style, Second one wants about the milage, Third one wants the cost & brand of the bike. So the Salesman explains about the product which customer needs what So he hiding some information & giving revelant information.

Encapsulation :- Encapsulation combines one or more information into a component.

Eg :- Capsule is mixed with one or more medicine & packed into the tube. So its related & acting in two modules.

Two concepts that go together in object-oriented approach are Encapsulation & Abstraction. (Abstraction is the representation of only the essential features of an object, while Encapsulation is the hiding of the Non-Essential features)

Think of a person driving a car. He does not need to know the internal working of the engine or the way gear changes work, to be able to drive the car (Encapsulation). Instead, he needs to know things such as how much turning the steering wheel needs, etc (Abstraction).

Consider the example from Programmer's perspective who wants to allow the user to add items to a list. The user only needs to click a button to add an item (Abstraction). The mechanism of how the item is added to the list is not essential for him. (Encapsulation).

Information Hiding :- Information hiding is manifested by enabling programmers to restrict access to certain data & code. This way, a class may prevent other classes & functions from accessing its data members & some of its member functions, while allowing them to access its non-restricted members.

Eg → Suppose we have Time class that counts the time of day :

```
Class Time
{ public:
      Void display();
  Private:
      int ticks;
};
```

The Display() member function prints the current time on screen. This member function is accessible to all. It's therefore declared public. By contrast, the data member ticks is declared private. Therefore, external users can't access it. Only Time's member functions are allowed to access it. In other words, the type of ticks; its current value; and, in fact, its very existence are pieces of information that we want to hide because these are all implementation details. The main problem with exposing such details is that they're likely to change in the future. By contrast, the public members of a class are its interface. Interface are less likely to change.

Distinction b/w Abstraction & Information Hiding :-

Abstraction can be used as a technique for identifying which information should be hidden. Confusion can occur when people fail to distinguish b/w the hiding information, & a technique (Abstraction) that is used to help identify which information is to be hidden.

Abstraction, Information hiding, & encapsulation are very different, but highly-related concepts. One could argue that abstraction is a technique that helps us identify which specific information should be visible, & which information should be hidden. Encapsulation is then the technique for packaging the information in such a way as to hide what should be hidden, & make visible what is intended to be visible.

explain
Ques :→ Abstraction
← → Information Hiding
Encapsulation

Abstract Datatype :- (If for a particular collection of data only the structure of data & the functions to be performed on the data is defined but the implementation is not defined, then such a collection of data is called Abstract datatype.

foreg :- An abstract stack data structure could be defined by two operations : push that inserts some data item into the structure & pop that extracts an item from it.)

Applications that use the datatype are oblivious to the implementation : they only make use of the operations defined abstractly. In this way the application, which might be millions of lines of code, is completely isolated from the implementation of the datatype. If we wish to change implementations, all we have to do is re-implement the operations. No matter how big our application is, the cost in changing implementations is the same.

In general terms, an abstract data type is a Specification of the values & the operations that has 2 properties :

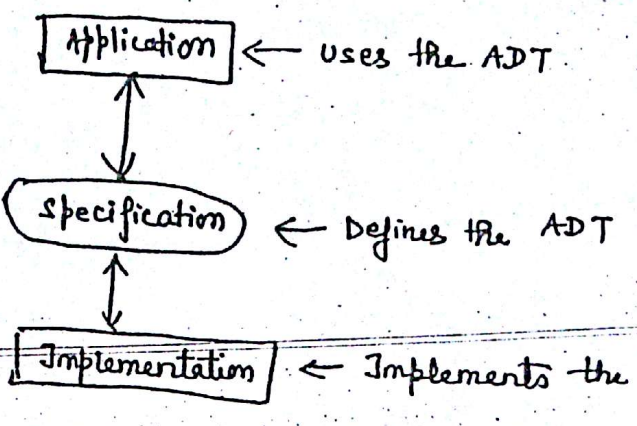
- it specifies everything you need to know in order to use the datatype
- It makes absolutely no reference to the manner in which the datatype will be implemented

When we use abstract datatypes, our programs divide into two pieces

- The Application : The part that uses abstract datatypes
- The Implementation : The part that implements the abstract datatype

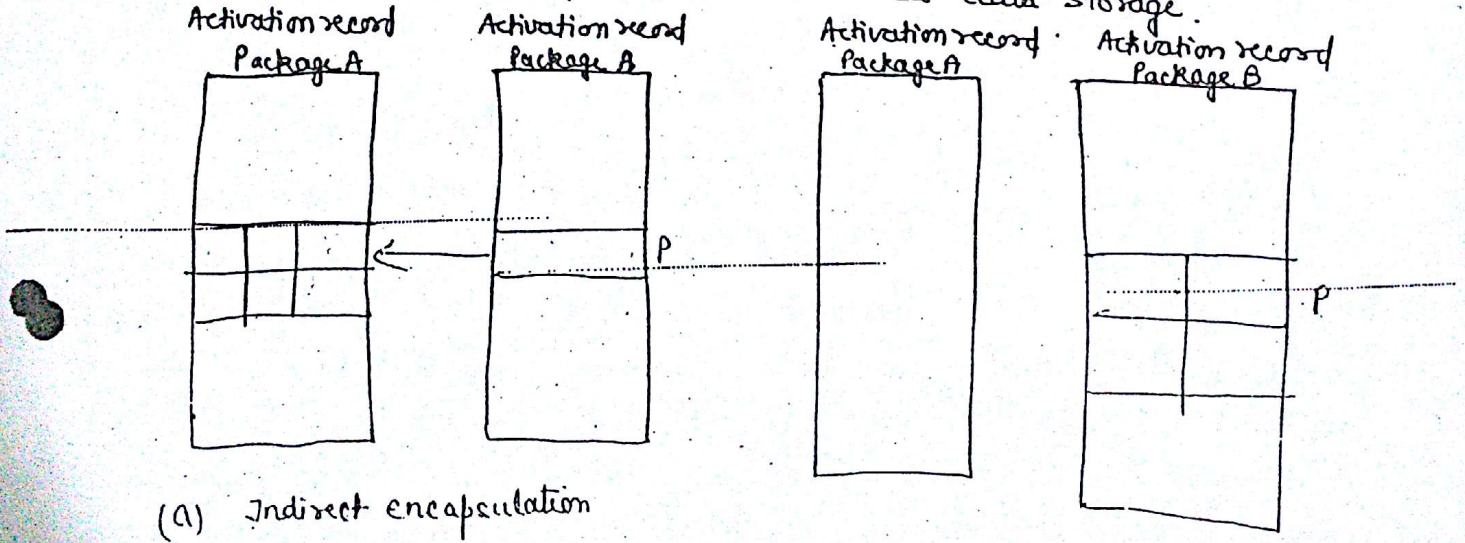
Ques :- Explain "Abstract data type".
Explain Data Abstraction with example.

These two pieces are completely independent. It should be possible to take the implementation developed for one application & use it for a completely different application with no changes.



Implementation :- A package contains 2 parts < Specification part / Implementation part

Figure presents two models for implemented encapsulated data objects. fig (a) is an example of indirect encapsulation. In this case, the structure of the abstract data type is defined by the package specification A. The actual storage for object P is maintained in an activation record for package A. In package B, which declares and uses object P, the run time activation record must contain a pointer to the actual data storage.



(a) Indirect encapsulation of object P.

(b) Direct encapsulation of object P

Fig (a) → Two Implementation models for Abstract data

An alternative implementation is given in fig (b) called the direct encapsulation. As in the indirect case, the structure of the abstract data object is defined by the specification for Package A. However, in this case the actual storage for object P is maintained within the activation record for package B.

In the Indirect case, the implementation of the abstract data type is truly independent of its use. If the structure of P changes, only Package A needs to change. Package B only needs to know that object P is a pointer & does not need knowledge of the format of the data pointed to by P. For large systems with thousands of modules, the time saved in not recompiling each module whenever the definition of P changes is significant.

The direct encapsulation case has the opposite characteristics. In this case, data object P is stored within Package B's activation record. Accessing of P's components may be faster because the standard-base-plus-offset accessing of data in a local activation record may be used; no indirection through a pointer is necessary. However, if the representation of the abstract object changes, then all instances of its use (eg Package B) must also be recompiled. This makes system changes expensive in compilation time, but more efficient in execution.

Generic Abstract Data types :- A Generic abstract type definition allows such an attribute of the type to be specified separately so that one base type definition may be given, with the attributes as parameters, & then several specialized types derived from the same base type may be created.

In C++, **Generic abstract classes are called Templates.**

Templates are those functions which can handle different datatypes without separate code for each of them.

Example :- Suppose we have two function

```
int Add (int a, int b)
  {
    return (a+b);
  }
float Add (float a, float b)
  {
    return (a+b);
  }
```

But if we use a C++ function template, the code will be

```
template < class T >
T add (T a, T b)    // C++ function Template
  {
    return (a + b);
  }
```

Here T is the typename. This is dynamically determined by the compiler according to the parameter passed

⇒ Generic Data types are datatypes where the operations are defined but the types of the items being manipulated are not; that is, the set of operations is defined but the set of values is not.

Ques: What do you mean by subprogram? (5) (5)

Ans :- A subprogram is an abstract operation defined by the programmer.

A subprogram definition has two parts < specification / Implementation

Specification of a subprogram :- It includes

- The name of the subprogram
- The signature (also called prototype) of the subprogram giving the number of arguments, their order, & the datatype of each, as well as the number of results, their order, & the datatype of each; and
- The action performed by the subprogram (i.e, a description of the function it computes).

If a subprogram returns a single-result data object explicitly, it is usually termed a function subprogram. (or simply function). The syntax in C is

Float FN (Float X, int Y) which specifies the signature

FN : real x integer → real

If a subprogram returns more than one result or if it modifies its arguments rather than returning results explicitly, it is usually termed as a procedure or subroutine & its syntax is

Void Sub (Float X, int Y, Float *Z, int *W);

In this specification

Void → indicates a null function, a subprogram that does not return a value

The formal parameter name preceded by * may indicate a result value or an argument that may be modified.

⇒ The tags in, out, & inout distinguish the three ways to invoke arguments to a subprogram.

In → It indicates an argument that is not modified by the subprogram

INOUT → It designates an argument that may be modified.
OUT → It designates a result.

Some problems arise in attempting to describe precisely the function computed:

① A subprogram may have implicit arguments in the form of nonlocal variables that it references.

② A subprogram may have Implicit results (side results) returned as changes to nonlocal variables or as changes in its in-out arguments.

③ A subprogram may not be defined for some possible arguments so that it does not complete execution in the ordinary way if given those arguments, but instead transfers control to some external exception handler or terminates execution of the entire program abruptly.

④ A subprogram may be History sensitive so that its results depend on the arguments given over the entire past history of its calls & not just on the arguments given in a single call. History sensitivity may be due to the subprogram's retaining local data b/w invocations.

Implementation of a subprogram :- A subprogram is implemented using the data structures & operations provided by the Programming language. The Implementation is defined by the subprogram body which consists of local data declarations defining the data structures used by the subprogram and statements defining the actions to be taken when the subprogram is executed. The declarations & statements are usually encapsulated so that neither the local data nor the statements are accessible separately to the user of the subprogram; the user may only invoke the subprogram with a particular set of arguments & receive the computed results.

Ques: Explain implementation of subprogram.

The C syntax for the body of a subprogram is typical :

```
Float FN (float X , int Y)   - Signature of subprogram
{ Float M(10) ; int N ;      - Declarations of local data objects
  ...
  ...                        - Sequence of statements defining
                               the actions of the subprogram.
}
```

— Typechecking must be done for each invocation of a subprogram that they receives proper types of arguments.

Subprogram Definitions & Subprogram Activations :

A subprogram definition is a static property of a Program. During execution of the program, if the subprogram is called (or invoked), an activation of the subprogram is created. When execution of the subprogram is complete, the activation is destroyed. If another call is made, a new activation is created. From a single subprogram definition, many activations may be created during program execution.

The distinction b/w definitions & activations of subprograms is important. A definition is what is present in the program as written & is the only information available during translation (eg, the type of subprogram variables is known, but not their value or location [r-value or l-value]). Subprogram activations exist only during program execution. During execution, code to access a variable's l-value or r-value can be executed, but the type of a variable may not be available unless the translator saved such information in a variable's descriptor.

Implementation of Subprogram Definition & Invocation :-

Consider C definition :

```
Float FN (Float x, int Y)
{
    Const initval = 2 ;   # define final val 10
    Float M(10) ; int N ;
    N = initval ;
    if (N > finalval) {...}
        return (20 * X + M(N)) ; }
```

This definition defines the following components needed for an activation of the subprogram at runtime :

1. FN's signature line provides information for storage for parameters (the data object X & Y) & storage for the function result, a data object of type float.
2. Declarations provide for storage of local variables (array M & variable N)
3. Storage for constant initval.
4. Storage for the executable code generated from the statements in the subprogram body.

The definition of the subprogram allows these storage areas to be organized & the executable code determined during translation. The result from translation is the template used to construct each particular activation at the time the subprogram is called during execution. Fig shows the subprogram definition as translated into a run-time template

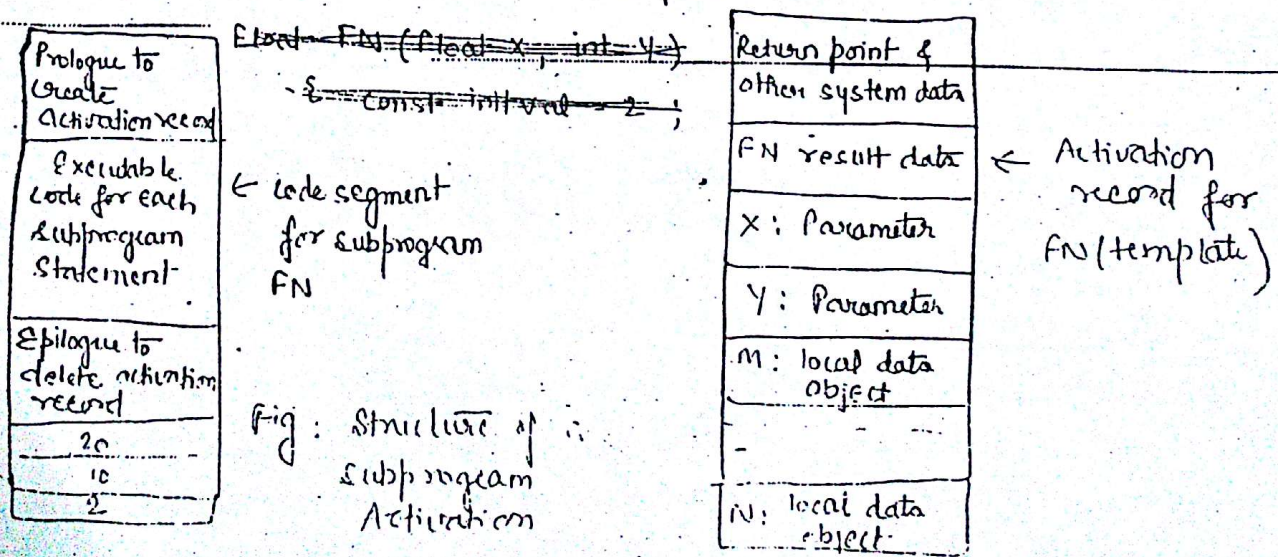


Fig : Structure of Subprogram Activation

To construct a particular activation of the subprogram from its template, the entire template might be copied into a new area of memory. However, rather than making a complete copy, it is far better to split the template into two parts:

- A static part, called the code segment, consisting of the constants & executable code. This part would be invariant during execution of a subprogram, & thus a single copy may be shared by all activations.
- A dynamic part, called an activation record, consisting of parameters, functions results, and local data, plus other implementation defined housekeeping data such as temporary storage areas, return points, & linkages for referencing nonlocal variables. This part has the same structure for each activation, but it contains different data values. Thus, each activation necessarily has its own copy of the activation record part.

The resulting structure during execution is shown in fig. for each subprogram, a single code segment exists in storage throughout program execution. Activation records are dynamically created & destroyed during execution each time the subprogram is called & each time it terminates with a return.

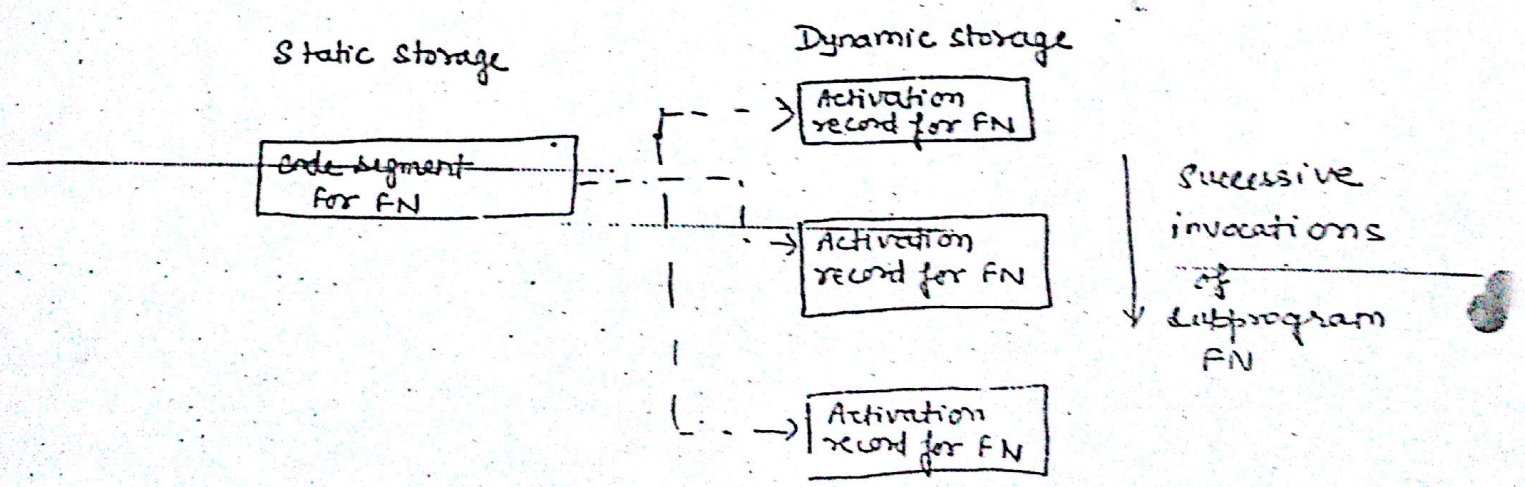


fig :- Shared code & separate Activation records.

It is important to remember that new types in C are generated by the struct feature, whereas typedef does not create a new type.

Benefits of typedef

① It allows the modification to be made only to the single type definition, rather than to many separate instances of declarations of individual variables.

② If a data object is transmitted as an argument to a subprogram, the subprogram definition must usually include a description of the argument. Here again, rather than repeating the entire structure description, we need use only the type name.

Type definition with Parameters :- For example, suppose we wish to define a type Section as a record, as in the Ada definition

```
type Section is
        record
            Room : integer ;
            Instructor : integer ;
            Classsize : integer range 0..100 ;
            ClassRoll : array (1..100) of Student_ID ;
        end record ;
```

This definition of Section defines a section of almost 100 students; to get larger sections requires a new type definition.

A parameterized type definition for Section allows the type definition to have a MaxSize parameter that determines the maximum class size:

```
type Section(Maxsize : integer) is
        record
            room : integer ;
            Instructor : integer ;
            Class size : integer range 0.. Maxsize ;
            ClassRoll : array (1..maxsize) of Student_ID ;
        end record ;
```

With this type definition, the maximum class size may be declared as part of the declaration of each individual variable:

```
X : Section (100)   -- gives maximum size 100
Y : Section (25)    -- gives    "     "  25
```

Generic Subprograms :- The specification of a subprogram ordinarily lists the number, order & datatypes of the arguments.
A generic subprogram is one with a single name but several different definitions, distinguished by a different signature. A generic subprogram name is said to be overloaded
For eg, In writing a set of subprograms for a university class registration program, two routines might be needed : one to enter a section in the table of class sections & another to enter a student in a class roll for a section. Both subprograms might be defined using the name Enter :

```
procedure Enter (student : in integer ; sect : inout Section) is
  begin - Statements here to enter student in a section roll list
  end ;

Procedure Enter (S: in section ; Tab : inout Classlist) is
  begin - Statements here to enter section in a classlist
  end ;
```

The name Enter is overloaded & has become the name of a generic Enter subprogram. When a call to Enter is made (Enter (A, B)), the translator must resolve the types of the arguments A & B with the appropriate types for the two Enter procedures. In this case, if A is of type integer, then the call is a call on the first definition of the Enter subprogram ; if A is of type section, then it is a call on the second
This resolution is made during translation.

Once the overloading is resolved, translation of the subprogram call proceeds as for any other subprogram call.

Type Definition :- In a type definition, a type name is given, together with a declaration that describes the structure of a class of data objects. The typename then becomes the name for that class of data objects. When a particular data object of that structure is needed, we need give only the type name, rather than repeating the complete description of the data structure.

For eg :- In Pascal, for eg, if several records, A, B & C are needed, each of which has the same structure (eg, a record describing a rational number) then the program may contain the type definition

```
type Rational = record
                  numerator : integer ;
                  denominator : integer
                end
```

followed by the declaration

```
var A, B, C : Rational ;
```

so that the definition of the structure of a data object of type Rational is given only once rather than being repeated three times for each of A, B & C.

In C, the situation is similar, New types can only be defined by the struct feature. Therefore, type Rational may be given as

```
struct RationalType
{ int numerator ;
  int denominator ; } ;

struct RationalType A, B, C ;
```

It can be done as follows:

typedef definition name

It can be written as

```
typedef struct RationalType
{ int numerator ;
  int denominator ; } Rational ;
```

followed by the declaration

```
Rational A, B, C ;
```

Storage Representations :- The storage representation for a data structure includes

① Storage for the components of the structure, &

② An optional descriptor that stores some or all of the attributes of the structure. The basic representations are shown in fig:

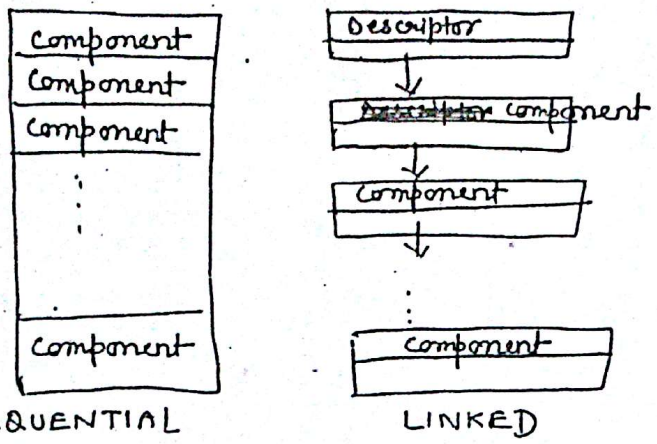


Fig: Storage representations for linear data structures

① Sequential representation → In which the data structure is stored in a single contiguous block of storage that includes both descriptor & components.

② Linked representation → In which the data structure is stored in several noncontiguous blocks of storage, with the blocks linked together through pointers.

Sequential representations are used for fixed-size structures & sometimes for homogeneous variable-size structures such as character strings or stacks. Linked representations are commonly used for variable-size structures such as lists.

Implementation of Operations on Data Structures :-

Component selection is of primary importance in the implementation of most data structures. Efficiency of both random-selection & sequential-selection operations is needed. For eg, for vector A, selection of component A[I] must be efficient. It is also desirable to step through the array sequentially, selecting A[1], A[2], ... more efficiently than simply as a sequence of random selections. These two basic types of component selection are implemented differently for sequential & linked storage representations.

Sequential representation : Random selection of a component often involves a base-address-plus-offset calculation using an accessing formula. The relative location of the selected component within the sequential block is called its offset. The starting location of the entire block is the base address. The offset is then added to the base address to get the actual location in memory of the selected component.

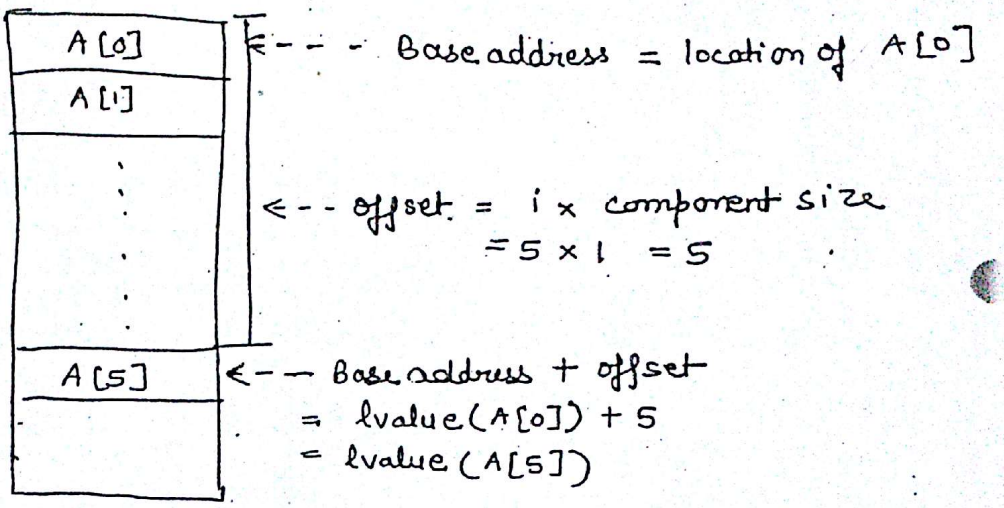


fig : Accessing components in a C char array.

Linked representation :- Random selection of a component from a linked structure involves following a chain of pointers from the first block of storage to the desired component. Selection of a sequence of components proceeds by selecting the first component & then following the link pointer from the current component to the next component for each subsequent selection.

Declaration & Type checking for Data Structures :-

The 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 :

1. Datatype is an Array
2. Number of dimensions is one
3. Number of components is 20
4. Subscripts naming the rows are the integers from 0 to 19.
5. Datatype of each component is float.

Declaration of these attributes allows a sequential storage representation for A and the appropriate accessing formula for selecting any component A[I] of A to be determined at compile time.

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 2 main problems :

1. 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. for eg, a subscripting operation that selects a component from an array may receive a subscript that is out of bounds for the particular array (i.e that produces an invalid l-value for the array component).) This is not a type-checking problem provided that the selection operation fails gracefully by noting the error & raising an exception (e.g. a subscript range error). However, if run-time type checking is disabled for efficiency & the exception is not raised, the result of a selection operation is almost always used immediately as an argument by some other operation. If the selection operation produces an incorrect l-value, the effect is similar to a type-checking error. Run-time checking is often required to determine whether the selected component exists before the formula is used to determine its precise l-value

② Type of a selected component :- A selection sequence may define a complex path through a data structure to the desired component. for example, the C :

A[2][3].link → item

selects the contents of the component named item in the record reached via the pointer contained in the component named link of the record that is the component in Row 2 & Column 3 of Array A. Static type checking only guarantees that if the component does exist, then it is of right type.

Explain Vectors & Arrays.

Vectors and Arrays.

Vector :→ A vector is a data structure composed of a fixed no: of components of the same type organized as a simple linear sequence. A component of a vector is selected by giving its subscript, an integer indicating the position of the component in the sequence. A vector is also termed a one-dimensional array or linear array. A two-dimensional array, or matrix, has its components organized into a rectangular grid of rows & columns. Both a row subscript & a column subscript are needed to select a component of a matrix. Multidimensional arrays of 3 or more dimensions are defined in a similar manner.

Attributes of a vector :

① Number of components, usually indicated implicitly by giving a sequence of subscript ranges, one for each dimension.

② Data type of each component, which is a single data type, because the components are all of the same type.

③ Subscript to be used to select each component, usually given as a range of integers, with the first integer designating the first component, the second designating the second component, & so on. This may be either a range of values as -5..5 or an upper bound with an implied lower bound, as A(10).

A typical declaration for a vector is the Pascal declaration

V: array [-5..5] of real ;

which defines a vector of 11 components, each a real number, where the components are selected by the subscripts, -5, -4, ..., 5. C declarations are simpler:

float a[10] ;

declares a C array of 10 components with subscripts ranging from 0 to 9.

Implementation :- The homogeneity of components & fixed size of a vector makes storage & accessing of individual components straightforward. Homogeneity implies that the size & structure of each component is the same, & fixed size implies the number & position of each component of a vector are invariant throughout its lifetime. A sequential storage representation is appropriate, as shown in fig, where components are stored sequentially. A descriptor may also be included to store some or all of the attributes of the vector, especially if all such information is not known until execution time. The upper & lower bounds of the subscript range, not needed for accessing components, are commonly stored in the descriptor if range checking for computed subscript values is required. The other attributes often are not stored in the descriptor at run time; they are needed only during translation for type checking & for setting up the storage representation.

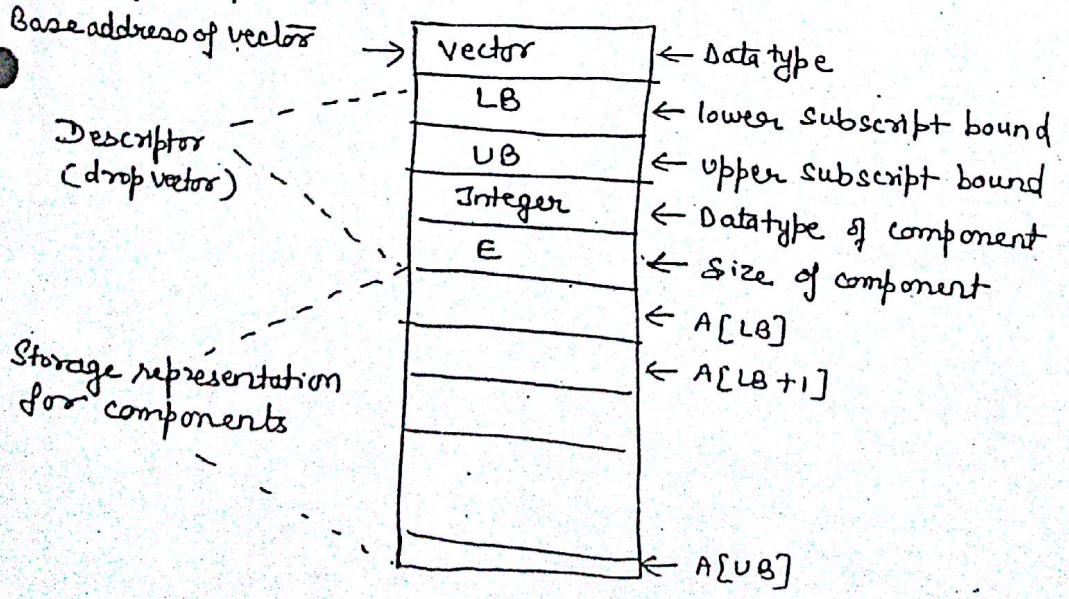


Fig.

Beginning at the initial component, the I^{th} component can be addressed by skipping I-1 components. If E is the size of each component, then we skip $(I-1) \times E$ memory locations. If LB is the lowerbound on the subscript range, then the No: of such components to skip is $I - LB$ or $(I - LB) \times E$ memory locations. If the first element of the vector begins at location α, we get the accessing formula for the l-value of a vector component:

$$lvalue(A[I]) = \alpha + (I - LB) \times E$$

This can be rewritten as

$$lvalue(A[I]) = (\alpha - LB \times E) + (I \times E)$$

Note that once storage for the vector is allocated, $(\alpha - LB \times E)$ is a constant (call it K) and the accessing formula reduces to

$$lvalue(A[I]) = K + I \times E .$$

In C, char arrays, E is 1; because LB is always 0, the equivalent C accessing formula reduces to the extremely efficient

$$lvalue(A[I]) = \alpha + I .$$

Multidimensional Arrays :- A vector is one-dimensional array; a matrix composed of rows & columns of components is a two-dimensional array; a three-dimensional array is composed of planes of rows & columns; similarly, arrays of any no: of dimensions may be constructed from arrays of fewer dimensions.

Specification & syntax :- A multidimensional array differs from a vector in its attributes only in that a subscript range for each dimension is required, as in Pascal declaration

B: array [1..10, -5..5] of real ;

Selection of a component requires that one subscript be given for each d:

Implementation of Multidimensional arrays :

→ A matrix is conveniently implemented by considering it as a vector of vectors.

→ The matrix is considered as a vector in which each element is a subvector representing one row of the original matrix. This representation is known as row-major order

→ In general, an array of any No: of dimensions is organized in row-major order when the array is first divided into a vector of subvectors for each element in the range of the first subscript, then each of these subvectors is subdivided into subsubvectors for each element in the range of the second subscript & so on.

The storage representation for a multidimensional follows directly from that for a vector. Here upper & lower bound for the subscript range of each dimension are needed.

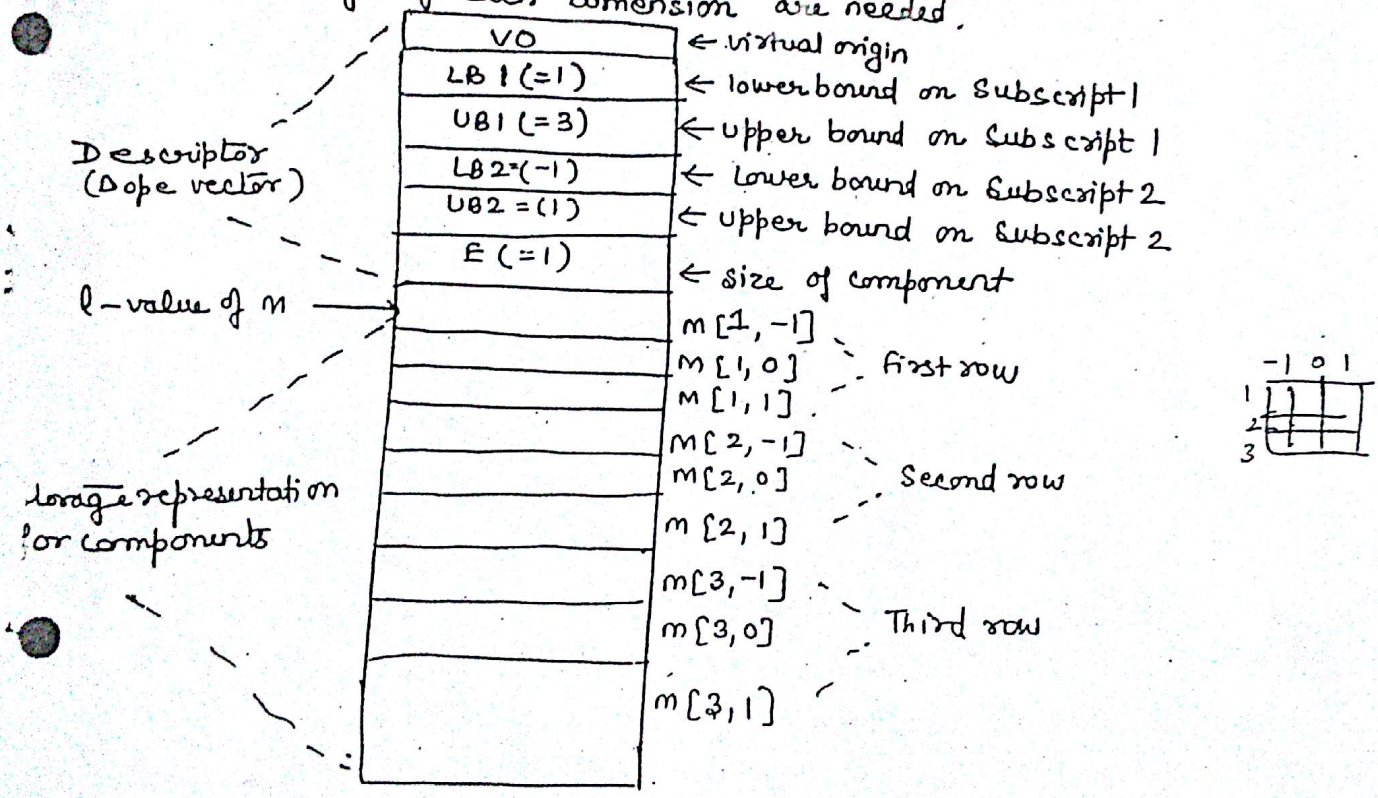


Fig :- Storage representation for 2-dimensional arrays

The subscripting operation, using an accessing formula to compute the offset of a component from the base address of the array, is similar to that for vectors. To find the l-value of A[I, J], first determine the No: of rows to skip over,

$(I - LB_1)$, multiply by the length of a row to get the location of the start of the I^{th} row, & then find the location of the j^{th} component in that row, as for a vector. Thus, if A is a matrix with M rows & N columns & A is stored in row major order, the location of element A[I, J] is given by

$$\text{value}(A[I,J]) = \alpha + (I - LB_1) \times S + [J - LB_2] \times E$$

where

α = base address

S = length of a row $= (UB_2 - LB_2 + 1) \times E$

LB_1 = lower bound on first subscript

LB_2, UB_2 = lower & upper bounds on the second subscript.

UNIT-2

Structured Data types :- A data structure is a data object that contains other data objects as its elements or components. A component may be elementary or it may be another data structure (eg. a component of an array may be a number or it may be a record, character string or another array).

Specification of Data structure types :- The major attributes for specifying data structures include the following:

1) **Number of components** :- A data structure may be of **fixed size** if the No: of components is invariant during its lifetime or of **variable size** if the No: of components changes dynamically. Variable-size datastructures Types usually define operations that insert & delete components from structures. Arrays & records are common example of fixed-size data structures; stacks, lists, sets, tables & files are examples of variable-size types.

2) **Type of Each Component** - A data structure is homogeneous if all its components are of the same type. It is heterogeneous if its components are of different types. Arrays, sets & files are usually homogeneous, whereas records & lists are usually heterogeneous.

3) **Names to be used for selecting components**: A data structure type needs a selection mechanism for identifying individual components of the data structure. For an array, the name of an individual component may be an integer subscript; for a record, the name is usually a programmer-defined identifier. Some Data structure types such as stacks & files allow access to only a particular component (eg the top or current component) at any time, but operations are provided to change the component that is currently accessible.

4) **Maximum Number of Components** : For a variable-size data structure such as a stack, a maximum size for the structure in terms of number of components may be specified.

5) **Organization of the Components** : The most common organization is a simple linear sequence of components. Vectors (one-dimentional arrays), records, stacks, lists & files are data structures with this organization. Array, record, & list types, however, are usually extended to multidimensional forms: multidimensional arrays, records whose components are records, & lists whose components are lists.

Operations on Data structures

① **Component selection operations** : Processing of data structures often proceeds by retrieving each component of the structure. Two types of selection operations are there

- Random selection : In which an arbitrary component of the data structure is accessed.
- Sequential selection : In which components are selected in a pre-determined order.

For eg :- In processing a vector, the subscripting operation selects a component at random (eg. V[4]), & subscripting combined with a **for** or **while** loop is used to select a sequence of components - as in

```
for I:=1 to 10 do ... V[I] ... ;
```

② **Whole data structure operations** : operations may take entire data structures as arguments & produce new data structures as results. (Eg : addition of two arrays, assignment of one record to another, or a union operation on sets).

③ **Insertion/deletion of components** :- Operations that change the No: of components in a data structure have a major impact on storage representations & storage management for data structures.

④ **Creation/destruction of Data structures** :- Operations that create & destroy data structures also have a major impact on storage management for data structures.

Implementation of Data structure Types :-

Implementation consideration for data structure types include two issues :

→ Efficient selection of components from a data structure &

→ Efficient overall storage management for the language implementation.