MODULE - II

Syllabus:-

Data Types:- Type Systems, Type checking, Records and Variants, Arrays, Strings, Sets, Pointers and Recursive Types, Lists, Files and Input/Output, Equality Testing and Assignment.

DATA TYPES

A data type, in programming, is a classification that specifies which type of value a variable has and what type of mathematical relational or logical operations can be applied to it without causing an error.

Eg: A string is a data type that is used to classify text & an integer is a data type used to classify whole numbers.

Two purposes of data type

i) Types provide implicit context for many operations, so that the programmer does not have to specify that context explicitly.

Eg: The expression 'a+b' will perform integer addition if a & b are of integer type and will perform floating point addition if a & b are of floating point type.

ii) Types limit the set of operations that may be performed in a semantically valid program.

Eg: prevents programmer from adding a character and a record. or prevents user from passing a file as a parameter to a subroutine that expects an integer.

1. Type Systems

A type system is

1) a mechanism to define types and associates certain language constructs with them,

2) a set of rules for type equivalence, type compatibility and type inference.

The constructs includes named constants, variables, record fields, parameters, subroutines, literal constants (eg: 17, 3.14, "foo") etc... and more complicated expressions containing these.

Type equivalence rules, determines when the types of two values are the same.

Type compatibility rules determine when a value of a given type can be used in a given context.

Type inference rules define the type of an expression based on the types of its constituent parts or the surrounding context.

Subroutines also have types in some languages but not in all languages. They need to have type if

- they can be passed as parameters
- they are returned by functions
- stored in variables.

1.1. Type Cheking

It is the process of ensuring that a program obey's the language's type compatibility rules. The violation of the rule is called as **type clash**.

strongly typed language

A language is said to be strongly typed if it prohibits the application of any operation to any object that is not intended to support that operation. Variables are necessarily bound to a particular data type.

statically typed language.

A language is said to be ~~strongly typed~~ statically typed if it is strongly typed and type checking can be performed at compile time. Here variables are declared before using it/them

dynamically typed

Dynamically typed programming languages are those in which variables must be necessarily be defined before they are used. this implies that dynamic typed languages do not require the explicit declaration of the variables before they are used.

weakly typed languages

weak typed languages are those in which variables are not of a specific data type. this does not means that variables do not have types; it does mean that variables are not "bound" to a specific data type.

Examples.

① statically typed

```
/* C code */
static int num, sum ; // Explicit declaration
num = 5;    // now use the variables
sum =10;
sum = sum + num;
```

③ Dynamic typing

```
/* Python code */
num = 10  // directly using the variable
```

④ strong typing

```
/* Python code */
foo = "x"
foo = foo + 2        // Error while compiling.
                     Type Error : Cannot concatenate
                     'str' and 'int' objects.
```

⑤ weak typing

```
/* PHP code */
<? php
$foo = "x";
$foo = $foo+2;   // not an error
echo $foo;
? >
```

Statically typed ⟶ Java, C, C++

dynamically typed ⟶ Python, PHP

strongly typed ⟶ Python

weakly typed ⟶ PHP, C

1.2. Polymorphism

It allows a single body of code to work with the objects of multiple types.

Module V & VI

1.3. Type

The type is the sum of its behaviour under all conditions. Types can be broadly interpreted in three ways;

- denotational
- constructive
- abstraction based.

In denotational point of view, a type is simply a set of values. A value has given type if it belongs to the set. An object has a given type if its value is guaranteed to be in the set.

From the constructive point of view, a type is either one of a small collection of built-in types (integer, character, Boolean, real etc... also called primitive or predefined type.), or composite type (record, array, set...).

From abstraction-based point of view, a type is an interface consisting of a set of operations with well defined and mutually consistent semantics.

Set of values in denotational view – **domain**
Eg: 'integer' domain

(Our focus is on constructive point of view.)

1.4. Classification of Types

i) Booleans

Booleans (or logicals) are as represented as a single bit quantities with 1 representing true and 0 representing false.

Eg: C, until C99, had no boolean, but later it supported, requires <stdbool.h> header file.

ii) Characters

Characters are implemented as one-byte quantities usually using ASCII encoding. But recent languages (Java and C#) use a two-byte representation to accomodate the **Unicode** character set.

Unicode - It is an international encoding standard for use with different languages and scripts, by which each letter, digit or symbol is assigned a unique numeric value that applies across different platforms and programs. It is developed by Unicode Consortium.

U+0030
⋮
U+0039 } Digits 0 to 9

U+0041
⋮
U+005A } A - Z

U+0061
⋮
U+007A } a - z

iii) Numeric Types

a) Integer

It a numeric value without a decimal. Integers are whole numbers and can be positive or negative. Integer type can have

a) short integer - stored using 16-bits i.e it can store upto 2^{16} or 65,536 unique values.

b) long integer - if larger number to be stored, use long integer, which uses 32 bit or more.

b) Decimal

A number with a decimal is referred to as a **decimal** a **float** or a **double**. The term float comes from 'floating point', we can control where the decimal point is located, (like 0.4f, 0.2f etc...). The term double refers to use double the amount of storage relative to float.

NOTE :

ORDINAL TYPES

Integers, Booleans and characters are all examples of discrete types or ordinal types — the domain to which they belong are countable and have a well defined predecessor and successor for each element other than the first and the last.

In ordinal type, the range of possible values can be easily associated with a set of positive integers.

User-defined ordinal types are - Enumeration & subrange. Here the user can determine the order of values.

(v) Enumeration Type

Enumeration type is the one in which all of the possible values, which are named constants are enumerated (means mentioned one-by-one) in the defenition. Enumeration types provide a way of defining and grouping collections of named constants which are called enumeration constants.

Example

In C, enum can be declared as;

```
enum   week { sun, mon, tue, wed, thu, fri, sat };
             0   1   2   3   4   5   6
```

- `enum` → keyword
- `week` → user defined data type
- `{ sun, mon, tue, wed, thu, fri, sat }` → value allocated for 'week'

```
enum week { sun, mon, tue, wed, thu, fri, sat };
enum week today;    // variable 'today' is of week type.
```

Eg:

```
#include<stdio.h>
#include<conio.h>
enum ABC{x,y,z};
void main()
 {
  int a;
  a=x+y+z;  // 0+1+2
  printf("sum : %d ", a);
  getch();
 }
```

Output

sum : 3

Eg

```
#include <stdio.h>
#include <conio.h>
enum week{sun, mon, tue,
          wed, thu, fri, sat};
void main()
 {
  enum week today;
  today = tue;
  printf("%d day", today+1);
  getch()
 }
```

Output

3 day.

By default, the value of the first enumeration symbol is 0. It is possible to give user defined values to enumeration constants.

enum week { sun=0, mon = 2, tue, wed, thu, fri, sat };
8 4 5 6 7

V. Subrange Types

Subrange types allows a variable to assume values that lie within a certain range.

Example. Syntax:

type subrange_identifier = lower-limit ... upper-limit ;

eg: const p=18;
Q=90;
type Number = 1...100;
Value = P...Q;

Subrange types can be created from a subset of an already defined enumerated type. From the example of enumeration;

type Day = tue... fri;

NOTE :

SCALAR DATA TYPE

All the above discussed data types are scalar data types because they all can hold a single data item. They are basic data types or simple data types. They are also **discrete types**, as they preserve a one-to-one order with a set of integer

vi) Composite Types

Nonscalar types are usually called composite or cu types. They are created by applying type constructor to one o more simpler types. Different composite types are;

1) Records (structures) :-

It consists of a collection of fields, each of which belong to a simpler type. Records are similar to mathematical tuples

2) Variant records (unions) :-

they differ from normal records is that only one of a variant record field is valid at any given time.

3) Arrays :-

Array is the collection of similar objects / types. It is the most commonly used composite type. Array of characters are referred to as strings.

4) Sets :-

A variable of a set type contains a collection of distinct elements of the base type. – unordered

5) Pointers :-

A pointer is a reference to an object of the pointer's base type.

6) List :-

List, like array, contains sequence of elements of different type. To search an element in the list, it should be recursively or iteratively examined, starting at the head element. A list may contain reference to sublist also. List is ordered.

7) Files :-

They represent data on mass-storage devices outside the ~~program~~ memory in which other program objects reside.

1.5. Orthogonality

Orthogonality means that, features can be used in any combination, the combination all makes sense, and the meaning of a given feature is consistent.

Or, orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. Algol 68 was the first language to use the principle of orthogonality.

~~Aggregate~~ Orthogonality makes even complex designs compact. In a purely orthogonal design, each action changes just one thing without affecting others.

Aggregates :-

It means multiple values. Examples for aggregate type includes arrays (homogeneous collection of objects), structures and classes, enumeration types, integer types etc..

2. Type Checking

In statically typed languages, it is important that every definition of an object (constant, variable, subroutine etc.) must specify the object's type.

In type checking, three important concepts include :

- type equivalence,
- type compatibility &
- type inference.

Type checking means, in statically typed languages, whenever a typed object is used, we must check that it occurs in the right type context.

Type checking can be static or dynamic;

Static type checking is that the type checking is done at compile time. Eg: C

In dynamic type checking, the type checking is done at run time. Eg: Python.

Type Equivalence

There are two standard ways to determine whether two types are considered the same;

- Name Equivalence
- Structural Equivalence.

Name Equivalence is the most straight forward: two types are equal if and only if, they have the same ~~name~~ type name.

For example, consider the C code,

```
typedef struct {
        int data [100];
        int count;
        } stack;

typedef struct {
        int data [100];
        int count;
        } set;

stack x, y;
set r, s;
```

If the variables are defined in same declaration or in declarations that use the same type name
⇒ name type equivalence

if name equivalence is used in a language then x and y would be of the same type and r and s would be of the same type, but the type of x or y would not be equivalent to the type of r and s. This means that the statement such as

```
x = y;
r = s;
```

would be valid, but statement such as

```
x = r
```

would not be valid

Using **structural equivalence** : two types are equal if and only if they have the same "structure" i.e. the names and types of each component of the two types must be the same and must be listed in the same order in the type definition.

From the above example, using structural equivalence,

the two types Stack and Set would be considered equivalent, which means that

$$x = r;$$

is valid.

Structural Equivalence is used in Algol-68, Modula-3, C and ML.

Name Equivalence is used in recent languages like Java, C#, Pascal, Ada etc...

Type Conversion and Casts

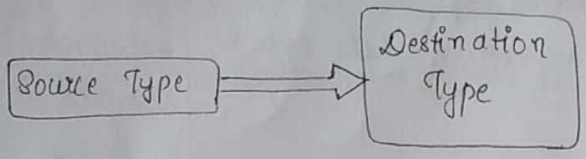
Type casting and type conversion occur when there is a need to convert one data type to another. The main difference between type conversion and type casting is that type conversion is automatically done by the compiler whereas type casting is to be explicitly done by the programmer.

Type conversion can be
- Implicit type conversion (coercion)
- Explicit type conversion (casting)

Type Conversion - Definition

It is the **automatic conversion** of one data type to another whenever required, done explicitly by the compiler. It can be also called as **coercion**. But there are two conditions to be satisfied before type conversion
- Source and destination type must be compatible
- Destination type must be larger than source type.

This type of conversion is called **widening** conversion, i.e. a smaller type is converted to larger type, widening of type occurs. For example numeric types 'int' & 'float' are compatible with each other while numeric to char and boolean, or char to boolean is not compatible.



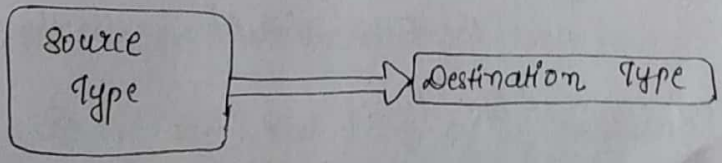
Eg: int a=3;
float b;
b = a; // Value in b = 3.000, int is converted to float.

Type Casting :- Defenition.

It can be defined as, casting of one data type to another data type, by the programmer, at the time of program design. Automatic type conversion is not possible all the time. Conditions to be followed are

- Source and destination type must ~~be~~ not be compatible.
- Destination type is smaller than the source type.

Here the programmer has to cast explicitly the larger data type to smaller data type using the casting operator '()'. It is also called as **narrowing** conversion.



Eg: int a;
byte b;
b = (byte) a;

Basis for comparison	Type casting	Type Conversion
Meaning	One data type is assigned to another by the user, using cast operator	Conversion of one data type to another automatically by the compiler.
Applied	can be applied to two incompatible data types.	can be only implemented when two data type are compatible.
Operator	Casting operator '()' is required.	No operator required
Size of data types	Destination type can be smaller than source type	Destination type must be larger than source type.
Implemented	Done during program designing	Done explicitly while compiling
Conversion type	Narrowing conversion	Widening conversion.

Type compatibility

Compatibility is the capacity for two systems to work together without having to be altered to do so. It is same as in the case of type compatibility. The data types should be able to work together without any errors. For example, in an assignment statement, the LHS and the RHS should be in the same type category or of compatible types.

Type Inference.

Type inference is the automatic detection of the data types of specific expressions in a programming language, usually done at compile time.

It involves analysing a program and then inferring the different types of some or all expressions in that program. It also tries to determine the argument types and the return value type.

Eg: `Foo(a,b) = x+y;`

the compiler knows that + operator takes two integers and also returns an Integer, so the compiler or interpreter can infer that FOO also has type Integer.

Three context in which type inference occurs:-

i) Subranges.

Lets take a subrange example in Pascal

```
type  Atype = 0...20;
      Btype = 10...20;

var   a: Atype;
      b: Btype;
```

What is the type of a & b?

It is neither Atype nor Btype, it is said that the result of any arithematic operation on a subrange has the subrange's base type, in this case it is integer.

So if the variable of a subrange type is used in any expression/operation the compiler will keep track of the largest and smallest values possible values of the expression/

operation and infer the type as its base type.

ii) Composite Types.

Type inference becomes an issue when an operation performed on a composite type yields a result of a different type other than the operands.

Character strings is an exp example. In Ada "abc" and is a three character array, "defg" is a four character array and by concatenating the two, the result is a seven character array. In the above three case, the size of the three arrays are known but the bounds and the index type are not known. they must be inferred from the context.

iii) ML Type System

ML (Meta Language) is a general purpose functional programming language which has its roots in LISP (can be called as 'LISP with types').

It uses the polymorphic Hindley-Milner type system — which automatically assigns the type of the most expressions without requiring explicit type annotations, and ensures type safety. A well-typed ML program does not cause runtime type errors.

It provides features like pattern matching for function arguments, garbage collection, imperative programming, call-by-value etc.. which makes it well suited to operate on other formal languages, such as in compiler writing, automated theorem proving and formal verification etc...

It was created by Robin Milner.

3. Records (Structures) & Variants (Unions)

Record types allow related data of heterogeneous type to be stored and manipulated together. Some languages like Algol 68, C, C++ and common LISP use the term structure (declar with the keyword struct) instead of record.

Fortran 90 – calls its record "types".

C++ – "structures" (a special form of class whose members are globally visible by default.).

Java – "classes" (there is no notion of struct).

In C, simple record can be defined as follows;

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```

In Pascal, the above example can be written as,

```
type two_chars = packed array [1...2] of char;
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean;
end;
```

Each component in a record is called as **fields**

Inorder to access a field/member, (.) dot operator is used along with the structure variables.

Eg: element copper; // declaring structure variable

```
copper.name[0] = 'c';
copper.name[1] = 'u';
```

Pascal is similar to that of C.

In Fortran, 90,

Copper % name and copper % atomic_weight etc... is used.

Nested Records / structures:

Most languages allow record definitions to be nested. In C,

```
struct oxe {
        char name[30];
        struct element {
                char name[2];
                int atomic_number;
                double atomic_weight;
                Bool metallic;
                } element_yielded;
        };
```

OR, it can be written also as,

```
struct oxe {
        char name[30];
        struct element element_yielded;
};
```

Struct tags and typedef in C & C++

In C&C++, struct tag is not exactly type names. The name of the type is the two-word phrase struct element (from above eg:). So inorder to declare a structure type variable we need to give struct element element_yielded. (above eg:). So inorder to avoid using struct keyword everytime when we create a structure variable, typedef can be used.

Eg:
```
typedef struct
{
  char name[20];
  int atomic_number;
}element_yielded;
```

OR

```
typedef struct element element_yielded;
```

Now elemente_yielded becomes a struct type, using which we can create more structure variables as;

```
element_yielded element1;
element_yielded element2;
```

ML differs from most languages that the order of record field is insignificant.

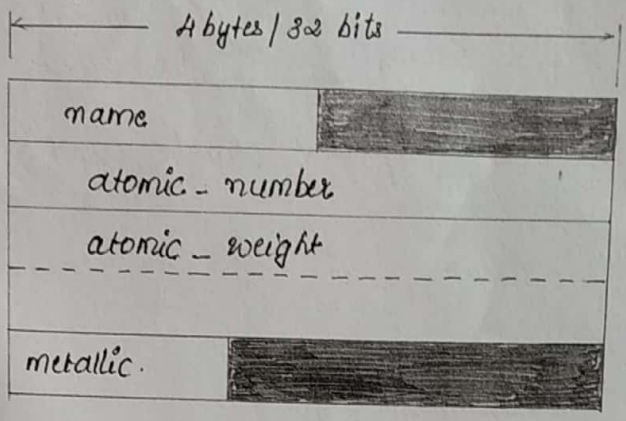
i.e { name = "Cu", atomic_number = 29, metallic = true } is same as { atomic_number = 29, name = "Cu", metallic = true }.

they will test true for only equality. ML tuples are defined as abbreviations for records whose field names are small integers like,

the values ('Cu', 29), {1='Cu', 2=29} and {2=29, 1='Cu'} test true for equality.

Memory Layout

The fields of a record are usually stored in adjacent locations in memory.



As the name field is only two characters long, it occupies two bytes in memory. Since atomic number is an integer and must be word aligned, there is a two-byte **hole** between the end of name and the beginning of atomic_number. Similarly in the case of Boolean variable (metallic).

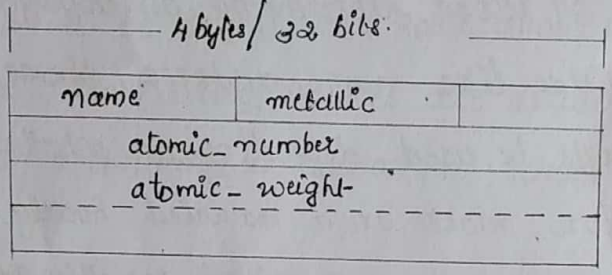
A few languages – like Pascal – allow the programmer to specify that a record type should be **packed**.

```
type element = packed record
        name : two chars;
        atomic_number : integer;
        atomic_weight : integer/real;
        metallic : Boolean;
    end;
```

The keyword **packed** indicates that the compiler should optimize for space instead of speed.

the packing is done by simply "pushing the fields" together

So, actually, holes in records waste space. Packing eliminate holes, but at heavy cost in access time. So, the way adopted by some compilers is to sort the record's fields according to the size of their allignment constraints. All byte alligned fields might come first, followed by half-word alligned fields, word-aligned fields, double-word-aligned fields and so on (if the h/w requires). The rearrangement is shown in figure;



4. Variant Records (Unions)

In 1960's and 1970's, the programming languages were designed with severe memory constraints. The programmer have to allocate variables "on top of" one another, sharing the same bytes in memory. C's syntax looks like;

```
union {
        int i;
        double d;
        _Bool b;
    };
```

Algol-68's syntax also looks the same; the overall size of this union would be the size of its largest member (here 'd'). All the members cannot be implemented simultaneously in the program.

A

Two main purposes of unions are;

i) the first purpose arises in system programs where unions allow the same set of bytes to be interpreted in different ways at different times

Eg: In memory management, it can be treated as unallocated space and sometimes for book keeping information.

ii) the second purpose is to represent alternative sets of fields with in a record

Eg: A record representing an employee might have several fields like name, address, phone, dept, ID etc... which might be used depending on whether the person in question works on a salaried, hourly or consulting basis C unions were not well suited for this purpose, usually variant records of Pascal allows this, which allow the programmer to specify that certain sets of fields should overlap one another in memory.

4. Arrays

Arrays are most common and important ~~data type~~ composite data type which is a fundamental part of almost every high-level language. Arrays groups homogeneous elements. It can be thought of as a mapping from an index type to a component or element type.

Some languages allow only discrete index type and some allows non discrete index types. Some languages require the elements to be scalar but some allow any element type. For convenience, the array indices are assumed to be discrete that admits a continuous allocation scheme.

Declarations

In C, array can be declared as

```
Char upper[26];
```

In Fortran:

```
character, dimension (1:26) :: upper
character (26) upper                    !shorthand notation.
```

In C lower bound of index is always zero, if an array has n elements, the indices will start from 0.....n-1.

In Fortran, the lower bound of index is one.

In Pascal,

```
var upper : array ['a'.... 'z'] of char ;
```

In Ada,

upper : array (character range 'a'..'z') of charact

Multidimensional Arrays :-

```
mat : array (1..10, 1..10) of real;        --Ada
real, dimension (10, 10) :: mat            ! Fortran
double mat[10][10];                        // C
```

Slices and Array Operations

A **slice** or **section** is a rectangular portion of an array. A slice is simply a contiguous range of elements in a one-dimensional array. The elements can themselves be arrays, but we cannot extract a slice along both dimensions as a single operation. Slicing is supported in languages like Fortran 90, Perl, Python, Ruby and R. Ada provides limited support.

In most languages, the only operation permitted on an array are selection of an element and assignment.

Slicing eg:

```
matrix (3:6, 4:7)
matrix (6:, 5)
matrix (2:8:2)        (a:b:c => a, a+c, a+2c ..... b.)
```

Dimensions, Bounds & Allocation

Rank: The number of dimensions is the **rank** of the array

Shape: The tuple of integers giving the size of the array along each dimension is called the **shape** of the array.

For static shaped arrays; storage is managed as;

- Static allocation of arrays whose lifetime is the entire program.
- Stack allocation for arrays whose lifetime is an invocation of a subroutine
- heap allocation for dynamically allocated arrays with more general life time.

- Dynamically typed languages allow run time binding of both the number and bounds of dimensions.

Compiled languages allow bounds to be dynamic, but require number of dimensions to be static.

Local array whose shape is known at elaboration time is allocated in stack. An array whose size may change during execution is generally allocated in heap.

Dope vectors

Dope vector is a data structure that is used by compilers to store some metadata about array like lower bound of each dimension and the size of each dimension. Given lower bound and size, upper bound information is redundant, but sometime, it is included to avoid computing it repeatedly at run time. The contents of the dope vector are initialized at elaboration time or whenever the dimensions change.

Stack Allocation

Early versions of Pascal required the shape (bounds) of the array to be specified statically. But later, it relaxed this requirement by allowing array parameters to have bounds that are symbolic names rather than constants. these parameters are called conformant arrays.

Eg: In Pascal, the size of the array is part of its type

```
VAR MyArray1 : ARRAY [1...10] OF INTEGER;
    MyArray2 : ARRAY [1..20] OF INTEGER;
```

MyArray1 & MyArray2 are of two different types. So it is not possible to define a procedure / function that works on arrays of arbitrary lengths. Conformant arrays were introduced to work around these issues by allowing a procedure to be defined taking a conformant array as parameter.

```
PROCEDURE MyProc (VAR x : ARRAY [low...high : INTEGER] OF INTEGER;
```

The value of the variables low, high is known when the procedure is invoked. x is a pointer to the array, low is lowerbound and high is upperbound.

Stack can be divided into

- fixed size part
- variable size part. , During elaboration time in Ada

An object whose size is statically known goes to fixed size part and whose size is not known until elaboration time goes to variable size part, a pointer to the object along with a dope-

vector. If the elaboration of array is in a nested block, the compiler delays allocating space until the block is entered, but still it allocates space for the pointer and dope vector.

Heap Allocation

Fully dynamic arrays (change shape at arbitrary times) must be allocated in the heap. If the number of dimensions of a fully dynamic array is statically known, the dope vector and the pointer to the data can be kept in the stack. But if the dimension can change, the dope vector must be placed at the beginning of the heap block.

If there is no garbage collection, the compiler itself have to deallocate the space occupied by the fully dynamic arrays when the control returns from the subroutine which they are declared. Space allocated in stack will be automatically deallocated by popping the stack.

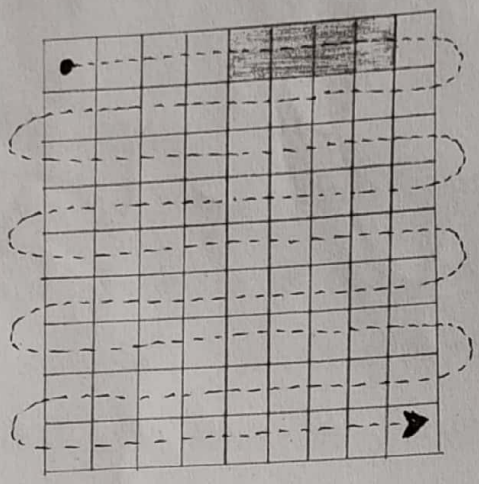
Memory Layout

In 1-D arrays, the elements are stored in continuous locations in the memory, second element is stored immediately after the first, third element immediately after the second and so on

But for multi-dimensional arrays, the order of the elements will be :-

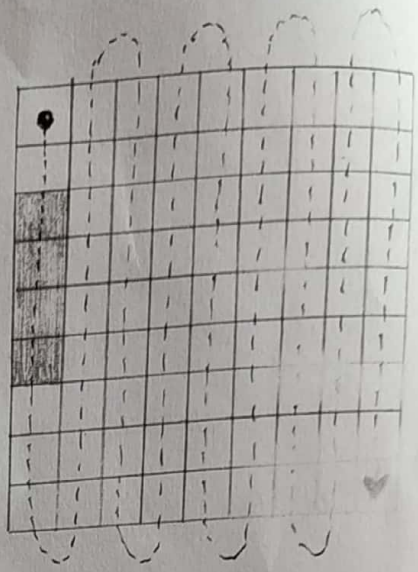
- row major order
- column major order

In row-major order, the elements of a row are contiguous in memory; in column major order, the elements of a column are contiguous.



Row-major order

A[0,1], A[0,2], A[0,3]



column-order major

A[1,1], A[2,1], A[3,1], A[4,1]

In row-major, row if fixed, column varies, In column major, column is fixed and row varies.

Row-major code in C

```
for( i=0; i<N; i++)   // rows
  {
    for j=0; j<N; j++) // columns
      ... A[i][j]
```

Column Major code – Fortran

```
do j = 1, N     ! columns
   do i = 1, N   ! rows
    ..: A[i, j)...
   end do
end do.
```

Address calculations

5. Strings

A string is simply an array of characters. There are some languages where there are pocerful string ~~fascities~~ fascilities - Snobol, Icon and other scripting languages like Perl, Python, Ruby. Literal strings are specified as a sequence of characters enclosed in either single quotes or double quotes. In C, literal character is delimited in single quotes and literal string is delimited in double quotes. character is a string of length one.

Cqq and c++ provides a rich set of escape sequences - (non-printing characters that serves special purpose which will be a combination of a character and a special character (\).).

Eg: \n - newline
\t - tab

Some languages that do not allow arrays to change size dynamically, allows strings to do so because of the following two reasons;

i) Manipulation of variable length strings is fundamental in in a huge number of computer applications

ii) Strings are one-byte elements in a one-dimensional array which makes it easier to implement than other dynamic arrays.

6. Sets

A 'set' in programming language is an unordered collection of an arbitrary number of distinct values of a common type. Set was introduced by Pascal and is found in many more recent languages. The type from which elements of a set are drawn is known as **base** or **universe** type.

Pascal supports different set operations like union, intersection and difference operations.

Eg:
```
var A,B,C : set of char;
    D,E : set of weekday;
....
A := B+C;   (* union)
A := B*C;   (* intersection)
A := B-C;   (* difference)
```

Set can be implemented using many ways including arrays, hash tables and various form of trees. Icon, Python, C++, Java, C#, etc supports sets.

For descrete base types like integers, set can be implemented using a characteristic array, which employs a bit vector whose length (in bits) is the number of distinct values of the base type. A 'one' in the k^{th} position ~~indicates that~~ in the bit vector indicates that the k^{th} element of the base type, is a member of the set, a 'zero' indicates that, it is not a member.

Advantage of using bit-vector set is that it can-

make use of fast logical instructions on most machines. It is suited for small base types like characters that would occupy 128 bits - 16 bytes (language that uses ASCII).

Disadvantage is that it is not suited for large base types because when it is represented as a bit vector, it will consume 500 megabytes on a 32-bit machine. Therefore a language that permits set with very large base type should employ an alternative implementation like hash table.

7. Pointers & Recursive Types

A **recursive** type is one whose objects may contain one or more references to other objects of the same type. They are used to build a variety of linked data structures like lists and trees. Recursive types can be records, functions or variables.

The reference to an object/record is implemented by using **pointers**. A pointer is a variable whose value will be a reference to/address of some object. Pointers were introduced in PL/I.

Some languages (Pascal, Ada 83, Modula-3) permits the pointers to point objects only in heap. In languages like C, C++, Ada 95, Algol..) it is possible to create a pointer to non-heap objects by using an "address of" operator.

Dangling references & Garbage Collection

Automatic storage reclamation is called as garbage collection. Whenever an object remains used / after usage, the unused storage must be reclaimed to make room for other things otherwise the program may run out of space and crash. In some languages it is done automatically - Java, C#, scripting languages. But in some like C, C++, Pascal... the space should be reclaimed explicitely

Whenever an object is reclaimed / destructed, the incoming reference to it should also be deleted or deallocated, otherwise it will still point to the memory location of the deallocated memory and will lead to dangling pointer or **dangling references**.

In C, the objects can be reclaimed by

```
free (my-ptr);
```

C++,

```
delete (my-ptr);
```

Pascal,

```
dispose (my-ptr);
```

In C++, additionally, destructor function is provided which automatically reclaims the space at the end of the object's life time.

Dangling Reference

It is a live pointer that no longer points to a valid object.

Eg:

```
void function()
  {
   int *ptr = (int *) malloc(size);
   ....
   ....
   free (ptr);   // ptr is now a dangling pointer.
  }
```

In above example, we first allocated a memory and stored its address int ptr. After executing few statements we deallocated the memory Now, ptr is still pointing to the memory address so it becomes dangling pointer. To solve this problem assign NULL to the pointer after the deallocation of memory that it was pointing.

```
   :
  free (ptr);
   ptr = NULL      // Now ptr is not a dangling pointer
}
```

Garbage Collection

Automic reclamation or garbage collection is very much important in many programming languages. Garbage is an object that cannot be reached through a reference.

Some of the automatic garbage collection techniques are;

i) Reference Counts

* One of the simplest method.
* ~~A counts~~ A counter is placed for every object when created. Whenever a reference (or pointer) is made to -

that object, reference count is increamented by 1. Whenever object is dereferred, the count is decreamented. Whenever count reaches 0, it is added to a list of unreferenced objects and periodically (or on needed) the items are destroyed from the list.

ii) Tracing Garbage Collection

Sometimes, eventhough the reference count of certain objects are non zero, they are useless. So tracing garbage collection works by keeping track of references into the heap (global root) and then tracing through the heap to find unreachable objects eligible for collection. Different tracing collection includes Mark-and-sweep, Pointer Reversal, Stop & Copy, Generational collection.

a) Mark-and-sweep.

3 steps are

1. the collector walks through the heap, tentatively marking every block as "useless."
2. Beginning with all pointers outside the heap, the collector recursively explores all linked data structures, marking each newly discovered block as "useful".
3. the collector again walks through the heap, moving every block that is still marked "useless" to the free list.

Pointers and Arrays in C

Pointers and arrays are closely linked in C.

```
int n;
int *a;          /* pointer to integer */
int b[10];       /* array of 10 integers */
```

Now all the following are valid,

```
a = b
n = a[3];        /* make a point to the initial element of b */
n = *(a+3);
n = b[3];        /* Pointer arithematic */
n = *(b+3);
```

'*' is the dereference operator. Above examples shows pointer arithematic, is valid only within the bounds of a single array. Given a pointer to an element of an array, the addition of an integer k produces a pointer to the element k positions later in the array. Similarly, pointer addition, pointer subtraction, comparison etc. is possible. All arithematic operations on pointers 'scale' the result based on the size of the referenced objects.

Advantages & Disadvantages of using Pointers.

Advantages

i) Pointers provide direct access to memory

ii) Allows to return more than one value to the functions.

iii) Reduces storage space & complexity of the program.

iv) Reduces the execution time of the program.

v) Provides an alternate way to access array elements.

vi) Allows to perform dynamic memory allocation & deallocation

vii) Helps to build complex data structures like linked list, stack, queues, trees, graph etc..

viii) Allows to resize the dynamically allocated memory block.

ix) Address of objects can be extracted using pointers

Disadvantages

i) Uninitialized pointers may cause segmentation fault.

ii) Dynamically allocated block needs to be freed explicitly. Otherwise it would lead to memory leak.

iii) Pointers are slower than normal variables.

iv) If pointers are updated with incorrect values, it might lead to memory corruption.

v) Pointer bugs are difficult to debug. Its programmer's responsibility to use pointers effectively and correctly.

8. Lists

A list or sequence is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. A list can be empty. Items in the list can be either a list or an atom (an object of built-in type (integer, real, character, string etc)..). ML, LisP, Python, other functional and logic languages uses lists.

* Homogeneous List → All elements of the same type (ML)
  Heterogeneous List → All elements can be of different types (Lisp List).

* List in [ ] square brackets – ML, comma seperated elements
  ( ) parentheses – Lisp, seperated by white space.

The following operations can be implemented on list data structure.

* A constructor for creating an empty list.
* An operation for testing whether list is empty or not.
* An operation for prepending an entity to a list
* An operation for appending an entity to a list.
* Determining head of the list. (first element)
* Referring tail of the list (rest of the elements).

- List can expand and shrink.
- Stored dynamically in the memory.
- Easier to implement than sets.
- Duplicate elements allowed
- More time to add new entry to the list.

Example:-

In Python,

```
my_list = []          # empty list
my_list = [1, 2, 3]   # list of integers
my_list = [1, "Hello", 3.4]   # list with mixed types
my_list = ["mouse, [8, 4, 6], ['a']]   # nested list
```

In Lisp,

cons - cons cells or cons is used to create a pair of values in LISP.
car - 'car' function is used to access first value
cdr - 'cdr' function is used to access second value.

Statement	output
(write (cons 1 2))	(1. 2)
(write (cons 1 nil))	(1)
(write (cons 1 (cons 2 nil)))	(1 2)
(write (car (cons 'a (cons 'b (cons 'c nil)))))	A

9. Files and Input/Output

For a program, inorder to communicate with the outside world, Input/Output (I/O) fasilities are needed. I/O communication takes place in two ways

- interactive I/O
- I/O with files.

Interactive I/O implies communication with human users or physical devices which work in parallel with the running program.

Files are offline storage implemented by the operating system. Files can be categorized into temporary files and persistent files. Temporary files exist for the duration of a single program run and their purpose is to store information that is too large to fit in the memory available to the program.

Peristant files allow a program to read data that existed before the program began running and to write data that will continue to exist after the program has ended.

I/O is one of the most difficult aspects of a language to design. Some language provides built-in file data types and special syntactic constructs for I/O. Other languages assign I/O entirely to library packages and need to be exported whenever required.

10. Equality Testing & Assignment

Assignment operator (=) is a binary operator which operates on two operands that assigns the value of right side expression's or variable's value to the left side variable.

Eg: $x = (a+b);$

$y = x;$

Equality (==) is a comparison operator, it is also a binary operator that operates on two operands, compares the value of left and right hand side expressions, return 1 if they are equal otherwise it will return 0.

For simple primitive data types like integers, floating-point number or characters, equality testing and assignment operations are relatively simple. But for abstract data types

Equality/ Comparison can be
- shallow comparison
- deep comparison.

Shallow comparison :-

There are r-values and l-values; in the presence of references, an expression is considered to be equal only if they refer to the same object.

Deep comparison.

Deep comparison means if the objects to which they refer are same.

Shallow & Deep assignment.

In imperative programming languages (the pgmg paradigm that uses statements that change a program's state), a shallow assignment $a := b$ will make a refer to the object to which b refers, and make a refer to the copy. i.e. it will copy the value of b into a. Deep assignments are relatively rare. They are used primarily in distributed computing and for parameter passing in remote procedure call (RPC) systems.