DATA TYPES

- It 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:

- String is a datatype used to classify text

- integer is a datatype used to classify whole numbers

Two Purposes of datatype:

- Types provide implicit context for many operations, so that the programmer does not have to specify that context explicitly.
- Eg: a + b will use integer addition if a and b are of integer type
- Types limit the set of operations that may be performed in a semantically valid program
- Eg: prevent the programmer from adding a character and a record

Type Systems

- Consist of

1. a mechanism to define types and associate certain language constructs with them

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

- **Type equivalence** rules determine 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 (sometimes) 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
- returned by functions
- stored in variables

Type Checking

- Type checking is the process of ensuring that a program obeys the language's type compatibility rules.

- A violation of the rules is known as a **type clash**

Strongly typed

- 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 datatype
- Ada

Weakly Typed

- Weak typed languages are those in which variables are not of a specific datatype.
- This doesnot means that variables donot have types, it means that variables are not bound to a specific data type
- PHP

Statically typed

- A language is said to be statically typed if it is strongly typed and type checking can be performed at compile time
- Variables are declared before they are used
- Java,C,C++

- **Dynamic Typing**
- Checks typing correctness at runtime
- Most scripting Languages
- Eg: Lisp & SmallTalk

Statically typed

Static int,num, sum;

Num=5;

Sum=2;

Sum=sum+num;

Dynamic Typing

num=2;

//Python,PHP directly using the variable

Strong Typing

Foo=‘x’;

Foo=foo+2; //Error cannot concatenate str and int

Python

Weakly Typed

```
<?Php
$foo=“x”;
$foo=foo+2;  // Not an error
Echo $foo;
    ?>
```

Polymorphism

- allows a single body of code to work with objects of multiple types

Type - Type is the sum of its behaviour under all conditions. Types can be categorised in 3 ways:

- *Denotational*
- *Constructive*
- *Abstraction-based*

- From the denotational point of view,

- A type is simply a set of values.
- A value has a 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 or a *composite* type

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

Classification Of Types

- Boolean
- Characters
- Numeric Types
- Decimal
- Enumeration
- Subrange

Ordinal types

- Integers, Booleans, and characters are all examples of discrete types (also called

  ordinal types)
- the domains to which they correspond are countable (they have a one-to-one correspondence with some subset of the integers),
- Have a well defined notion of predecessor and successor for each element other than the first and the last.

User Defined types

- Enumeration
- Subrange

Enumeration

- An enumeration type consists of a set of named elements.

- In Pascal, one can write:

type weekday = (sun, mon, tue, wed, thu, fri, sat);

- The values of an enumeration type are ordered, so comparisons are generally valid
- (mon < tue),

- There is usually a mechanism to determine the predecessor or successor of an enumeration value

- The ordered nature of enumerations facilitates the writing of enumeration-controlled loops:

for today := mon to fri do begin ...

- It also allows enumerations to be used to index arrays:
- var daily_attendance : array [weekday] of integer;

- An alternative to enumerations

- const sun = 0; mon = 1; tue = 2; wed = 3; thu = 4; fri = 5; sat = 6;

- In C,
- enum weekday {sun, mon, tue, wed, thu, fri, sat};

is **equivalent to**

```
typedef int weekday;
const weekday sun = 0, mon = 1, tue = 2,
wed = 3, thu = 4, fri = 5, sat = 6;
```

SubrangeTypes

- First introduced in Pascal, and are found in many subsequent languages

- A subrange is a type whose values compose a contiguous subset of the values of some discrete *base* type

- In Pascal,

  type test_score = 0..100;

  workday = mon..fri;

- In Ada

```
type test_score is new integer range 0..100;
subtype workday is weekday range mon..fri;
```

Type Checking

- In statically typed languages, every definition of an object must specify the object's type

- 3 concepts:

- Type Equivalence
- Type Compatibility
- Type Inference

- Type checking can be static and dynamic

Type Equivalence

- Two types are considered same if they have
1. Name Equivalence
2. Structural Equivalence

Name Equivalence

- Name Equivalence - Two types are equal if and only if they have same type name

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

typedef strct{
     int data[100];
     int count;
          }set;
```

- Stack x,y;
- Set r,s;

- Using name equivalance
- x and y are of same type and r and s are of same type
- But type of x and y will not be equivallent to r and s
- i.e
- X=y and r=s are valid but x=r is invalid

Structural Equivalence

- Two types are equal if and only if they have same structurei.e name and types of each componenet must be same and must be listed in same orderin type defenition
- Here 2 types stack and set can be considered equallent i.e
- x=r is valid
- Ada follows name equivalence and ML follows structural equivalence

Type Compatibility

- Type compatible means a value's type must be compatible with that of the context in which it appears
- The type compatibility is categorized three types by the compiler:
  - ✔ **Assignment compatibility**
  - ✔ **Expression compatibility**
  - ✔ **Parameter compatibility**

Type Compatibility

- **Assignment compatibility**

  the type of the RHS must be compatible with that of the LHS

  if the type of variable assigned to another variable is different it results into loss of value of assigned variable if the size of assigned variable is large than the size of variable to which it is assigned

- For example

  float n1=12.5;

  int n2=n1; //might give a warning “possible loss of data”.

- assigning of float value to int type will result in loss of decimal value of n1.

Type Compatibility

- **Expression compatibility**
- **types of the operands of + must both compatible with some common types that supports addition**
- Consider following example

```
int n=3/2;
cout<<n;
```

- Here the result will be 1. The actual result of 3/2 is 1.5 but **because of incompatibility there will be loss of decimal value**

Type Compatibility contd..

- **Parameter compatibility**
- **In a subroutine call ,the types of any arguments passed into the subroutine must be compatible with the types of the formal parameters and the types of any formal parameters passed back to the caller must be compatible with the types of the corresponding arguments.**
- incompatibility in type of actual parameter and formal parameters, loss of data occurs
- Here output will be n=8 due to incompatibility in actual and formal parameter type

```
void show(int n)
{
cout<<"n="<<n;
}
void main()
{
show(8.2);
}
```

Type inference

- the automatic detection and deduction of the data type of an expression in a programming language
- automatic deduction usually done at compile time
- involves analyzing a program and then inferring the different types of some or all expressions in that program
- programmer does not need to explicitly input and define data types every time variables are used in the program

Type inference contd..

- **What determines the type of the overall expression?**
  - The result of arithmetic operator has the same type as the operands
  - The result of comparison is Boolean
  - The result of a function call has the type declared in the function header
  - The result of an assignment has the same type as the LHS
- Operations on subrange & composite types do not preserve the types of the operands. Type inference found in ML.

Type inference contd..

- Subranges

- Subranges improve program readability and error checking
- They define an order, so they can be used in enumeration-controlled loops
- They were first introduced in Pascal
- In pascal the result of any arithmetic operation on a subrange has the subrange's base type. Ada calls subtypes with range constraints
- type Atype is new integer range 0..20;

- E.g.
```
type Atype = 0..20;
type Btype = 10..20;
var a:Atype;b:Btype;
What is the type of a+b?
```
- It is either Atype or **Btype.** here it is integer

- Composite Type
  - Nonscalar types are usually called composite, or constructed types
  - generally created by applying a type constructor to one or more simpler types
  - eg. records (structures), variant records (unions), arrays, sets, pointers, lists, and files
  - Composite literals are sometimes known as aggregates (in Ada)

- Composite Type

- Character String provides a simple example.
  - In Pascal the literal string "abc" has the type array[1..3] of char;
  - ✔ In Ada "abc" is a three character array,"defg" is a four character array, and the result is a seven character array formed by concatenating the two
  - ✔ The seven char result may be assigned into an array of type array(1..7)of character
  - ✔ For all three ,size of the array is known ,but **the bounds and index types are not; they must be inferred from context.**

Type inference contd..

- ML Type System

- Type inference occurs in certain functional languages like ML
- **Programmers have the option of declaring the types of objects** in these languages
- Programmers may also choose not to declare certain types,in which case the compiler will infer them,based on the known types of the literal constants and the syntactic structure of the program
- ML type inference is the invention of the language creator,Robin Milner

Type Conversion

- conversion of a value into another of a different data type
- implicitly or explicitly made
- Implicit conversion/ type coercion, is automatically done
- **Explicit conversion/ casting**, is performed by code instructions

conversion example

- The C code below illustrates implicit and explicit coercion

```
Line 1  double x, y;
Line 2  x = 3;                  // implicit coercion (coercion)
Line 3  y = (double) 5; // explicit coercion (casting)
```

- int constant 3 is automatically converted to double before assignment (implicit coercion).
- An explicit coercion is performed by involving the destination type with parenthesis

Non*converting* Type Cast

- **A change of type that does not alter the underlying bits is called a nonconverting type cast or type pun**

- In Ada , using a built in generic subroutine called unchecked conversion

```
-- assume 'float' has been declared to match IEEE single-precision
function cast_float_to_int is
    new unchecked_conversion(float, integer);
function cast_int_to_float is
    new unchecked_conversion(integer, float);
...
f := cast_int_to_float(n);
n := cast_float_to_int(f);
```

- C++ inherits casting mechanisms of C

Coercion contd..

- Fortran has lots of coercion, all based on operand type

- coercion rules are a relaxation of type checking
  - Recent thought is that this is probably a bad idea
  - Languages such as Modula-2 and Ada do not permit coercions

Records

- A record is heterogeneous collection of data for storage and manipulation
- Languages like Algol 68,C,c++ and Lisp call it structure

- Known as types in Fortran
- Structure in C++ are specialized form of class.
- Java doesn't support structure
- C# use reference model for class types and value model for struct type.

Syntax

- Record in C

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

- Record In Pascal

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

Accessing Record Fields

- Each component of the record is called a field
- To access a field, many languages use dot notation

Accessing Record Fields

- In C

```
struct element copper;
copper.name[0] = 'C';
copper.name[1] = 'u';
copper.atomic_number=92;
```

Pascal uses same notation as that of C

Accessing Elements In Records

- Fortran 90
  - copper%name
  - copper%atomic_weight
- Cobol and Algol 68 (reverse the order of the field)
  - name of copper
  - atomic_weight of copper
- ML
  - #name copper
  - #atomic_weight copper

Accessing Elements In Records

- Common Lisp
  - (element-name copper)
  - (element-atomic_weight copper)

Nested Records

- Many languages allow nested records
- In C,

```
struct ore {
    char name[30];
    struct {
        char name[2];
        int atomic_number;
        double atomic_weight;
        _Bool metallic;
        } element_yielded;
    };
```

Alternative Nesting

```
struct ore {
    char name[30];
    struct element element_yielded;
    } malachite;
struct element {
char name[2];
int atomic_number;
double atomic_weight;
_Bool metallic;
}
```

Accessing Elements of Nested Records

- Pascal / C
- malachite.element_yielded.atomic_number

- Cobol
- atomic_number of element_yielded of malachite

Accessing Elements of Nested Records

- ML
- #atomic_number #element_yielded malachite

- Common Lisp
- (element-atomic_number (ore
  -element_yielded malachite))

- In ML the order of record fields is insignificant

- ML record value
- {name = "Cu",atomic_number = 29,
  atomic_weight = 63.546, metallic = true}

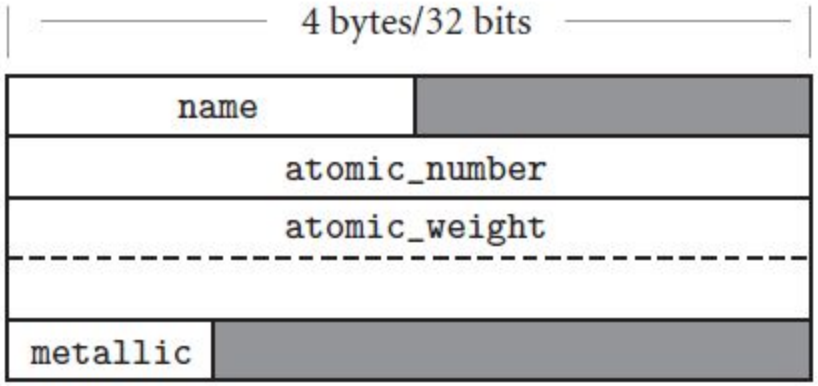
- Is the same as
- {atomic_number = 29, name = "Cu",
  atomic_weight = 63.546, metallic = true}

Memory Layout

- Fields are stored in adjacent locations
- To access a field, the compiler generates a load or store instruction with displacement addressing

Memory Layout of Record

- In a 32 bit machine word aligned
- Each block is 4 bytes



name field is 2 characters ---> 2 BYTES
atomic_number is an integer ---> Word Aligned
Makes a hole of 2 bytes
Boolean variables occupy 1 byte
Makes a hole of 3 bytes
Most compilers will allocate 20 bytes for each member

Value Model

- In language with value model of variables
- nested records are naturally embedded in the parent record

```
type
    T = record
        j : integer;
    end;
    S = record
        i : integer;
        n : T;
    end;
var s1, s2 : S;
...
s1.n.j := 0;
s2 := s1;
s2.n.j := 7;
writeln(s1.n.j);          (* prints 0 *)
```

The assignment of `s1` into `s2` copies the embedded `T`.

Reference Model

- In language with reference model

```
class T {
    public int j;
}
class S {
    public int i;
    public T n;
}
...
S s1 = new S();
s1.n = new T();                    // fields initialized to 0
S s2 = s1;
s2.n.j = 7;
System.out.println(s1.n.j);        // prints 7
```

- Here the assignment of s1 into s2 has copied only the reference, so s2.n.j is an

  alias for s1.n.j.

Packed Types (Over come mmy wastage)

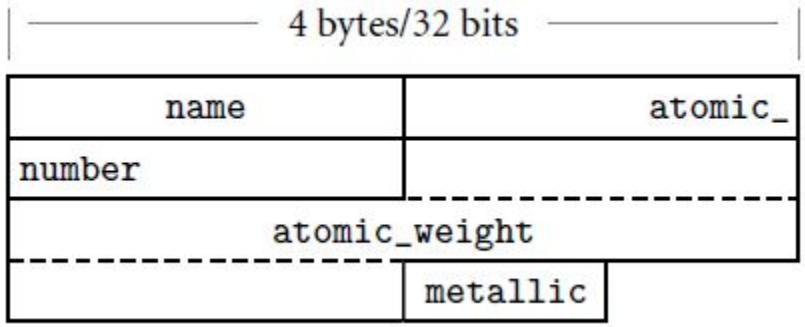
- Pascal allows packed records

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

- Packed records pushes fieldstogether without holes.
- Packed keyword intimates compiler to optimize for space than speed

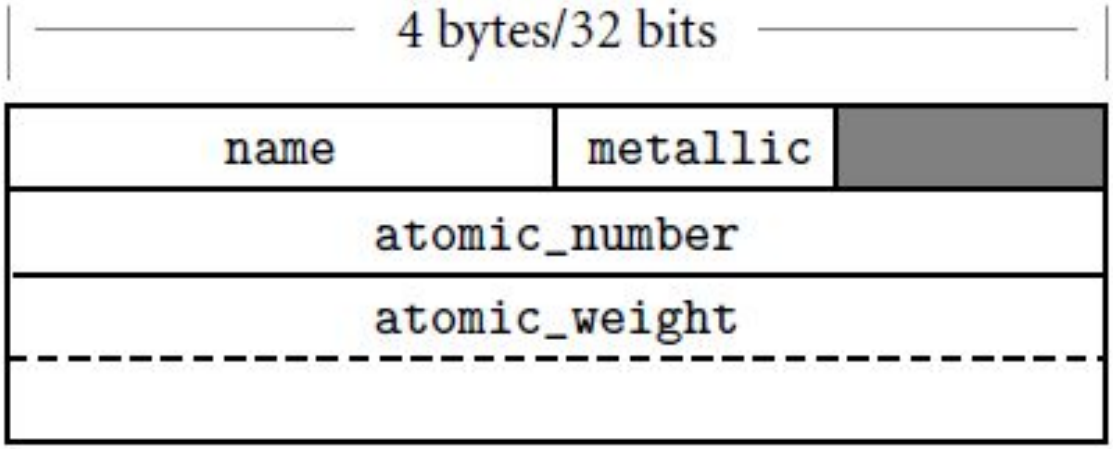
Layout of Packed Records

- Packed records pushes fieldstogether without holes.
- Packed keyword intimates compiler to optimize for space than speed



Holes

- Holes waste space
- Packing eliminates holes but at heavy cost in access time
- Solution is to sort according to size of alignment
- byte-aligned fields might come first, then half-word aligned fields, then word-aligned fields, double-word–aligned fields and so on


4 bytes/32 bits
name
metallic
atomic_number
atomic_weight

With Statements

- manipulating the fields of a deeply nested record becomes complicated in some languages

```
ruby.chemical_composition.elements[1].name := ’Al’;
ruby.chemical_composition.elements[1].atomic_number := 13;
ruby.chemical_composition.elements[1].atomic_weight :=
26.98154;
ruby.chemical_composition.elements[1].metallic := true;
```

With Statements

- Pascal provides a with statement to simplify such constructions

```
with ruby.chemical_composition.elements[1] do begin
    name := 'Al';
    atomic_number := 13;
    atomic_weight := 26.98154;
    metallic := true
end;
```

Variant Records (Unions)

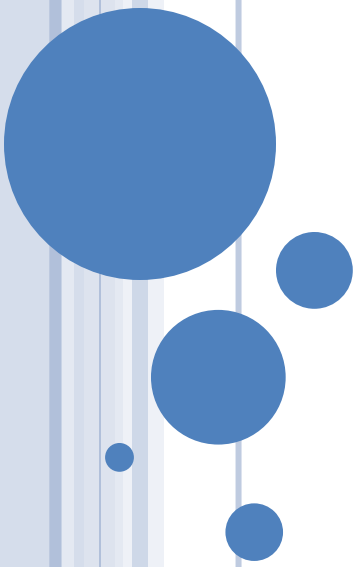
- A variant record is a record in which the fields act as alternatives,
- only one of which is valid at any given time.
- Each of the fields is itself called a variant.
- In C,

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

- Programming languages of 1960 and 1970 were designed in an era of severe memory constraints
- Variables share same bytes in memory

- The overall size is that of its largest member (say d)
- Bytes of d would be overlapped by i and b is implementation dependent

- Purpose
- Useful in system programs
  - Allow the same set of bytes to be interpreted in different ways at different times.
  - Storage used as unallocated space, bookkeeping information or as user allocated data of arbitrary type
- To represent alternative sets of fields within a record



ARRAYS

- Arrays are homogeneous.
- It is a mapping from an index type to a component or element type
- Languages like Fortran require that the index type be integer
- Languages like Ada and Pascal aloows index be of discrete type

- Languages like Fortran 77 requires elements type to be scalar
- Many languages allow any element type
- Some languages like AWK and Perl allow non discrete index types

Syntax

- Element of an array can be accessed by appending a subscript delimited by parentheses or square brackets

- In Fortran or Ada

  A(3)

- In Perl and C

  A[3]

Declarations

- In C

  char upper[26];

- Lower bound of index is 0
- Indices are 0..n-1

- In Fortran:

  Character, dimension (1:26) :: upper

  Character (26) upper - shorthand notation

- Lower bound of index is 1

- Dimension keyword declares an array of <lwb><upb>
- Upper is array name
- Character keyword specifies that the array is a character array

- In some languages array constructor is used
- In Pascal

  var upper : array [’a’..’z’] of char;
- In Ada:

  upper : array (character range ’a’..’z’) of character;

Multidimensional Arrays

- **Ada**
- mat : array (1..10, 1..10) of real;

- **Fortran**
- real, dimension (10,10) :: mat

- **Modula-3**,
- VAR mat : ARRAY [1..10], [1..10] OF REAL;

- is same as
- VAR mat : ARRAY [1..10] OF ARRAY [1..10] OF REAL

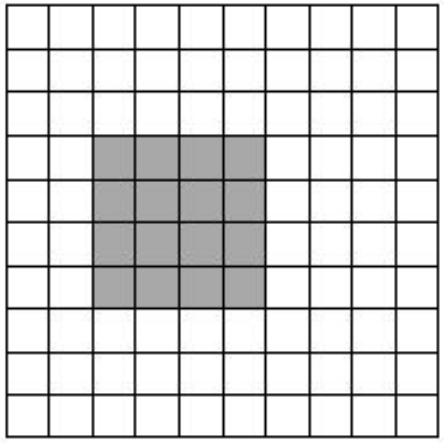
- In **ada**
- mat1 : array (1..10, 1..10) of real;
- In **C**
- double mat[10][10];

Slices and Array Operations

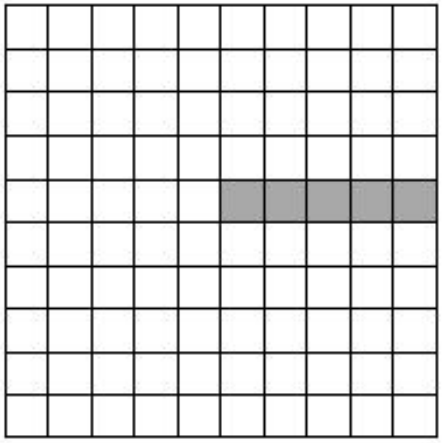
- A slice or section is a rectangular portion of an array.

- Fortran 90, and Single Assignment C, Perl, Python, Ruby and R provide slicing

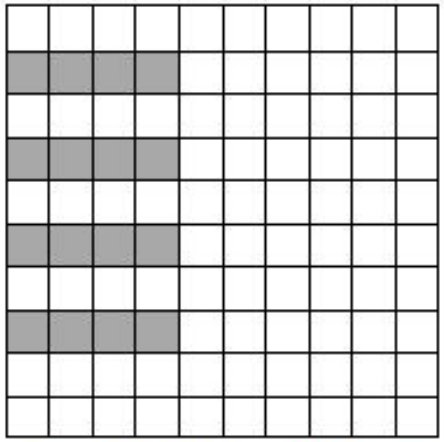
Slicing In Fortran 90



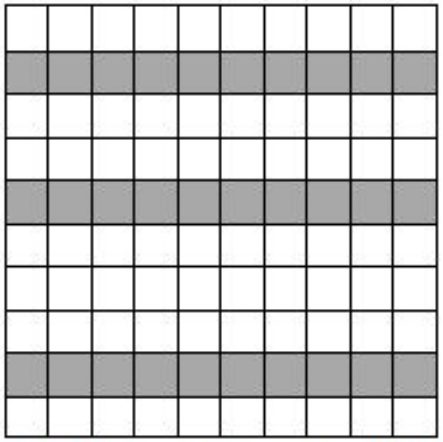
matrix(3:6, 4:7)



matrix(6:, 5)



matrix(:4, 2:8:2)



matrix(:, (/2, 5, 9/))

- In most languages, the only operations permitted on an array are **selection** of an element and **assignment**
- Ada and Fortran 90 allow arrays to be compared for equality

- In **Ada**
- 1D arrays can be compared for lexicographic ordering : A < B
- Allows the built-in logical operators to be applied to Boolean arrays.

- **Fortran**
- Fortran has a very rich set of array operations
- built-in operations that take entire arrays as arguments

- A + B is an array each of whose elements is the sum of the corresponding elements of A and B.
- tan(A), returns an array consisting of the tangents of the elements of A.
- Supports many intrinsic functions for searching and summarization, transposition, and reshaping and subscript permutation

Dimensions, Bounds, and Allocation

- **Static Arrays**

- Static allocation for 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 lifetime.

- Storage management is more complex for arrays whose shape is not known until elaboration time or whose shape may change during execution

- Compiler must arrange to allocate space as well as make shape information available at run time

Dope Vectors

- Descriptors or Dope vectors are used to hold shape information at run time
- On compilation symbol table maintains bounds and dimensions for every array
- If array dimensions are statically known, the compiler can look them up in the symbol table in order to compute the address of elements of the array

- When they are not statically known, the compiler must generate code to look them up in a dope vector at run time
- Dope vector contain the lower bound and the size of each dimension
- If the language performs dynamic semantic checks for out-of-bounds subscripts, then the dope vector may contain upper bounds as well

Stack Allocation

- Subroutine parameters are the simplest example of dynamic shape arrays
- Earlier Pascal required the shape of all arrays to be specified statically
- Standard Pascal relaxes this requirement by allowing array parameters to have bounds
- These parameters are called **conformant**
- **arrays**

```
function DotProduct(A, B:array [lower..upper:integer] of real):real;
var i : integer;
    rtn : real;
begin
    rtn := 0;
    for i := lower to upper do rtn := rtn + A[i] * B[i];
    DotProduct := rtn
end;
```

- Here lower and upper are initialized at the time of call
- We divide the stack frame into a fixed size part and a variable-size part

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

Heap Allocation

- Arrays that can change shape at arbitrary times are sometimes said to be fully dynamic

- Fully dynamic arrays must be allocated in the heap

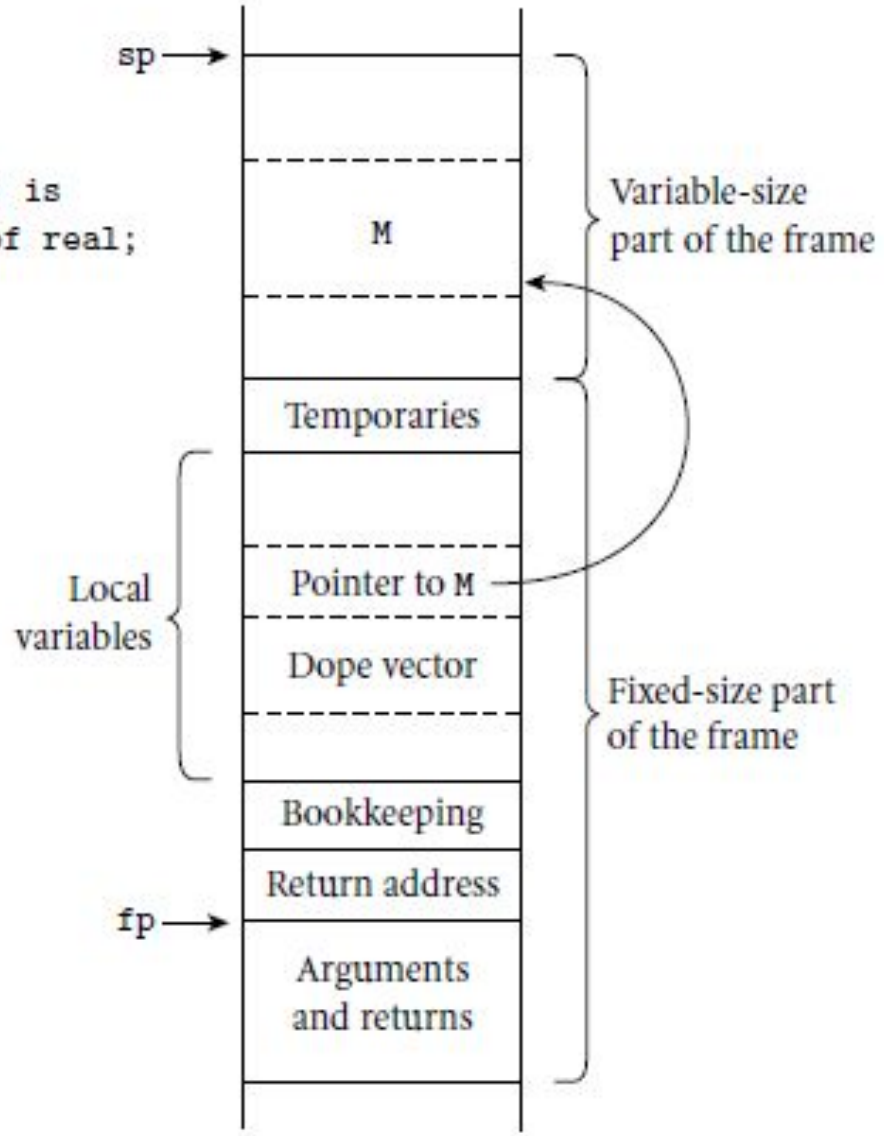
```
String s = "short";
...
s = s + " but sweet";
```

- Dynamically resizable arrays appear in APL, Common Lisp, and the various scripting languages.

- Supported by the vector, Vector, and ArrayList classes of the C++, Java, and C# libraries, respectively

```
-- Ada:
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
    ...
end foo;
```

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```



Memory Layout

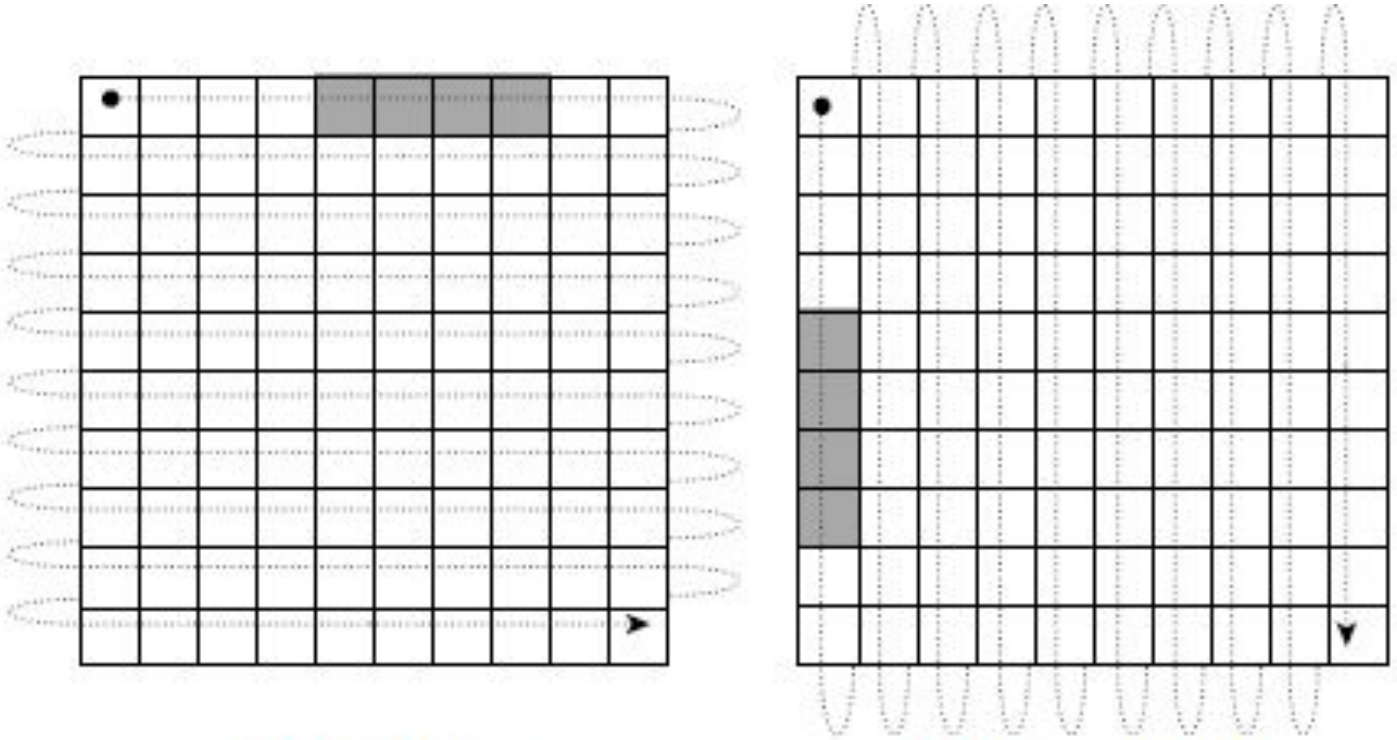
- Arrays are stored in contiguous locations

- In a one-dimensional array, the second element of the array is stored immediately after the first; the third is stored immediately after the second, and so forth

- In multidimensional arrays,the first element of the array is put in the array's first memory location.

- But where to put second element?
  - Row major order
  - Column major order

- In row-major order,
  - consecutive locations in memory hold elements that differ by one in the final subscript
  - A[2, 4] is followed by A[2, 5]
- In column-major order
  - consecutive locations in memory hold elements that differ by one in the initial subscript
  - A[2, 4] is followed by A[3, 4].



Row-major order

Column-major order

Row-Pointer Layout

- Some languages employ an alternative to contiguous allocation
- Rows of an array can lie anywhere in memory, and create an auxiliary array of pointers to the rows is created
- Requires more space

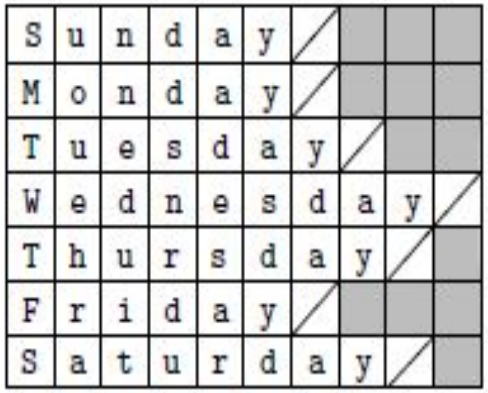
- 3 Advantages:

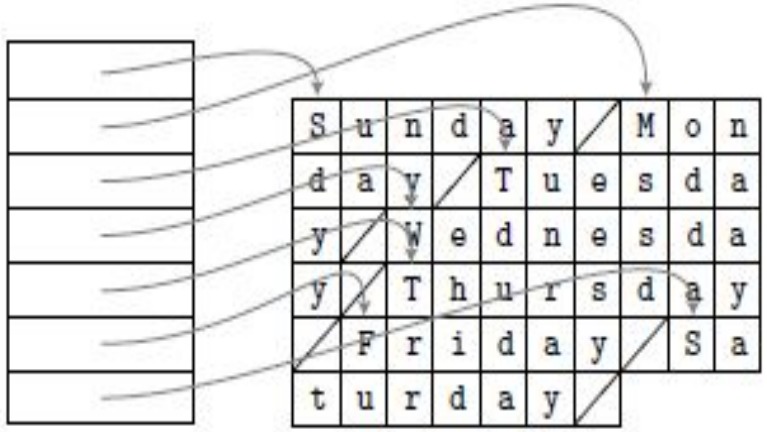
- Sometimes allows individual elements of the array to be accessed more quickly.
- It allows the rows to have different lengths, without devoting space to holes at the ends of the rows. Such arrays are called a **jagged array**
- It allows a program to construct an array from preexisting rows without copying

- C, C++, and C# provide both contiguous and row-pointer organizations for multidimensional arrays

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```





Address Calculations

- A : array [$L1$ . . $U1$] of array [$L2$ . . $U2$] of array [$L3$ . . $U3$] of elem type;

- S1: Size of plane (No of elements in plane)
- S2: number of rows
- S3: size of individual elements

- constants for the sizes of the three dimensions:
- $S3$ = size of elem type
- $S2 = (U3 - L3 + 1) \times S3$
- $S1 = (U2 - L2 + 1) \times S2$

- The size of a row ($S2$) is the size of an individual element ($S3$) times the number of elements in a row

- The size of a plane ($S1$) is the size of a row ($S2$) times the number of rows in a plane

- The address of A[i, j, k] is then
- address of A

$$+ (i - L1) \times S1$$
$$+ (j - L2) \times S2$$
$$+ (k - L3) \times S3$$

This computation involves 3 multiplications and 6 additions/subtractions.

- The entire expression can be computed at run time.
- In most cases, much of the computation can be done at compile time
- If bounds of array are known at compile time then S1,S2 and S3 are compile time constants

- Subtractions of lower bound can be distributed out of parenthesis

- $(i \times S1) + (j \times S2) + (k \times S3) + \text{address of A}$
  $-[(L1 \times S1) + (L2 \times S2) + (L3 \times S3)]$

- Instruction sequence to load A[i, j, k] into a register

- 1. r4 := r1 × *S*1
- 2. r5 := r2 × *S*2
- 3. r6 := &A – *L*1 × *S*1 – *L*2 × *S*2 – *L*3 × S3 – – one or two instructions
- 4. r6 := r6 + r4
- 5. r6 := r6 + r5
- 6. r7 := *r6[r3]