Module 2

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: Most programming languages require the programmer to declare the data type of every data object, and most database systems require the user to specify the type of each data field.

The available data types vary from one programming language to another, and from one database application to another, but the following usually exist in one form or another:

integer**:** In more common parlance, *whole number*; a number that has no fractional part.
floating-point**:** A number with a decimal point. For example, 3 is an integer, but 3.5 is a floating-point number.
character**(text):** Readable text

Purpose that types serve in a programming language:

Types provide implicit context for many operations, so that the programmer does not have to specify that context explicitly.
In C, for instance, the expression a+b will use integer addition if a and b are of integer type, it will use floating point addition if a and b are of double type.

Type Systems:

A type system consist of (1) a mechanism to define types and associate them with certain language constructs, and (2) a set of rules for type equivalence, type compatibility, and type inference.

- Type equivalence rules determine (a) when the types of two values are the same.
- Type compatibility rules determine (a) when a value of a given type can be used in a given context.
- Subroutines are considered to have types in some languages, but not in others. Subroutines need to have types if they are first- or second-class values.
- Type information allows the language to limit the set of acceptable values to those that provide a particular subroutine interface.

In a statically scoped language the compiler can always identify the subroutine to which a name refers.

Type checking:-

Type checking is the process of ensuring that a program obeys the languages type compatibility rules.
A violation of the rules is known as a type clash.
A language is said to be strongly typed if it prohibits, in a way that the language implementation can enforce, the application of any operation to any object that is not intended to support that operation.
A language is said to be statically typed if it is strongly typed and type checking can be performed at compile time.
Ex: Ada is strongly typed and for the most part statically typed.

A Pascal implementation can also do most of its type checking at compile time, though the language is not quite strongly typed: untagged variant records are its only loophole.
Polymorphism allows a single body of code to work with objects of multiple types. It may or may not imply the need for run-time type checking.

- Because the types of objects can be thought of as implied (unspecified) parameters, dynamic typing is said to support *implicit parametric polymorphism*.

ML and its descendants employ a sophisticated system of *type inference* to support implicit parametric polymorphism in conjunction with static typing.
Compiler determines whether there exists a consistent assignment of types to expressions that guarantees, that no operation will ever be applied to a value of an inappropriate type at run time.
This job can be formalized as the problem of *unification.*
In object-oriented languages, *subtype polymorphism* allows a variable *X* of type *T* to refer to an object of any type derived from *T*.
Explicit parametric polymorphism (generics), allows the programmer to define classes with type parameters.
Generics are particularly useful for *container* (*collection*) classes: "list of T" (List<T>), "stack of T" (Stack<T>), and so on, where T is left unspecified.

The Meaning of "Type"

There are at least three ways to think about types, which we may call the *denotational*, *constructive*, and *abstraction-based* points of view.
From the denotational point of view, a type is simply a set of values.
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* types), or a *composite* type created by applying a type *constructor* (record, array, set, etc.) to one or more simpler types.
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.
For most programmers, types usually reflect a mixture of these viewpoints.

In denotational semantics i.e one of the leading ways to formalize the meaning of programs, a set of values is known as a *domain.*
Here *everything* has a type—even statements with side effects.
Each function maps a *store*—a mapping from names to values that represents the current contents of memory—to another store.
This represents the contents of memory after the assignment.
When a programmer defines an enumerated type (e.g., enum hue {red, green, blue} in C), he or she certainly thinks of this type as a set of values.
One usually thinks in terms of the way the type is built from simpler types, or in terms of its meaning or purpose.

<u>Classification of Types:-</u>

Most languages provide built in types similar to those supported in hardware by most processors: integers, characters, Boolean, and real (floating point) numbers.
Booleans are typically implemented as single byte quantities with 1 representing true and 0 representing false.
Characters have traditionally been implemented as one byte quantities as well, typically using the ASCII encoding.
More recent languages use a two byte representation designed to accommodate the Unicode character set.
Numeric Types:-

C and Fortran distinguish between different lengths of integers and real numbers.
Differences in precision across language implementations lead to a lack of portability: programs that run correctly on one system may produce run-time errors or erroneous results on another.
A few languages, including C,C++,C# and Modula-2,provide both signed and unsigned integers.
Fortran,C99 and Common Lisp provide a built in complex type, usually implemented as a pair of floating point numbers that represent the real and imaginary Cartesian coordinates.
Ada supports fixed point types, which are represented internally by integers.
Integers, Booleans, characters are examples of discrete types.
Discrete, rational, real, and complex types together constitute the *scalar* types.
Scalar types are also sometimes called *simple* types.
Enumeration Types
Enumerations were introduced by Wirth in the design of Pascal.
They facilitate the creation of readable programs, and allow the compiler to catch certain kinds of programming errors.
An enumeration type consists of a set of named elements.
In Pascal one can write:

Type weekday=(sun, mon, tue), ordered, so comparisons are generally valid(mon<tue).

There is usually a mechanism to determine the predecessor or successor of an enumeration value(in Pascal, tomorrow :=succ (today).
Values of an enumeration type are typically represented by small integers, usually a consecutive range of small integers starting at zero.
In many languages these ordinal values are semantically significant.
Subrange Types
Like enumerations, subranges were first introduced in Pascal.
A subrange is a type whose values compose a contiguous subset of the values of some discrete base type.
In Pascal subranges look like this:

```
Type test_score=0..100;
     Workday= mon..fri;
```

In Ada one would write

```
Type test_score is new integer range 0..100;
Subtype workday is weekday range mon..fri;
```

The range... portion of the definition in Ada is called a type constraint.

test_score is a derived type, incompatible with integers.
The workday can be more or less freely intermixed.
Composite Types:-
Nonscalar types are usually called composite, or constructed types.
They are generally created by applying a type constructor to one or more simpler types.
Common composite types include records, variant records, arrays, sets, pointers, lists, and files.

- ➢ Records- A record consists of collection of fields, each of which belongs to a simpler type.
- ➢ Variant records-It differs from normal records in that only one of a variant records field is valid at any given time.
- ➢ Arrays-Are the most commonly used composite types.
- ➢ An array can be thought of as a function that maps members of an index type to members of a component type.
- ➢ Sets- A set type is the mathematical powerset of its base type, which must often be discrete.
- ➢ Pointers-A pointer value is a reference to an object of the pointers base type. They are most often used to implement recursive data types
- ➢ Lists-Contain a sequence of elements, but there is no notion of mapping or indexing.
- ➢ A list is defined recursively as either an empty list or a pair consisting of a head element and a reference to a sublist.
- ➢ Files-Are intended to represent data on mass storage devices, outside the memory in which other program objects reside.

Orthogonality : Orthogonality is important in the design of expressions, statements, and control-flow constructs.
A highly orthogonal language tends to be easier to understand, to use, and to reason about in a formal way.
To characterize a statement that is executed for its side effect(s), and that has no useful values, some languages provide an “empty” type.
In a language (e.g., Pascal) without an empty type, the latter of these two calls would need to use a dummy variable:

```
var dummy : symbol_table_index;
...
dummy := insert_in_symbol_table(bar); _
```

One particularly useful aspect of type orthogonality is the ability to specify literal values of arbitrary composite types.
Composite literals are sometimes known as *aggregates*.
We can write the following assignments:

```
p := ("Jane Doe ", 37);
q := (age => 36, name => "John Doe ");
A := (1, 0, 3, 0, 3, 0, 3, 0, 0, 0);
B := (1 => 1, 3 | 5 | 7 => 3, others => 0);
```

Here the aggregates assigned into p and A are *positional*; the aggregates assigned into q and B name their elements explicitly.
The aggregate for B uses a shorthand notation to assign the same value (3) into array elements 3, 5, and 7, and to assign a 0 into all unnamed fields.
ML provides a very general facility for composite expressions, based on the use of *constructors.*
Type checking: In most statically typed languages, every definition of an object must specify the objects type.
Type compatibility is the one of most concern to programmers.
It determines when an object of a certain type can be used in a certain context.
Objects and contexts are often compatible even when their types are different.
Type *conversion* (also called *casting*), changes a value of one type into a value of another.
Type *coercion*, which performs a conversion automatically in certain contexts.
Nonconverting type casts, are sometimes used in systems programming to interpret the bits of a value of one type as if they represented a value of some other type.
Type equivalence:
In a language in which the user can define new types, there are two principal ways of defining type equivalence.
Structural equivalence is based on the content of type definitions.
Name equivalence is based on the lexical occurrence of type definitions.
Structural equivalence is used in Algol-68,Modula-3,C.
The exact definition of structural equivalence varies from one language to another.
Structural equivalence in Pascal:

```
Type R2=record
   a,b : integer
end;
```

should probably be considered the same as

```
type R3 = record
     a : integer;
     b : integer
end;
```

But what about

```
Type R4 = record
  b : integer;
  a : integer
end;
```

The reversal of the order of the fields change the type.
Consider the following arrays,again in a Pascal like notation:

```
type str = array [1.....10] of char;
type str = array [0......9] of char;
```

Here the length of the array is the same in both cases, but the index values are different.
Some (Fortran, Ada) consider them compatible.
Variants of Name Equivalence

Simplest of type declarations:
TYPE new_type = old_type; (* Modula-2 *)
Here new_type is said to be an *alias* for old_type.
We treat them as two names for the same type, or as names for two different types that happen to have the same internal structure.
A language in which aliased types are considered distinct is said to have *strict name equivalence*.
A language in which aliased types are considered equivalent is said to have *loose name equivalence*.
Ada achieves the best of both worlds by allowing the programmer to indicate whether an alias represents a *derived* type or a *subtype*.
A subtype is compatible with its base (parent) type; a derived type is incompatible.
Consider the following example:
Name vs structural equivalence

1. type cell = . . . – – whatever
2. type alink = pointer to cell
3. type blink = alink
4. p, q : pointer to cell
5. r : alink
6. s : blink
7. t : pointer to cell
8. u : alink

Under strict name equivalence, p and q have the same type, because they both use the *anonymous* (unnamed) type definition on the right-hand side of line 4, and r and u have the same type, because they both use the definition at line 2.
Under loose name equivalence, r, s, and u all have the same type, as do p and q. Under structural equivalence, all six of the variables shown have the same type, namely pointer to whatever cell is.
Type Conversion and Casts: In a language with static typing, there are many contexts in which values of a specific type are expected.
In the statement
a := *expression*
we expect the right-hand side to have the same type as a.
In the expression
a + b
The overloaded + symbol designates either integer or floating-point addition.
We expect either that a and b will both be integers, or that they will both be reals.
In a call to a subroutine,
foo(arg1, arg2, . . . , argN)

- We expect the types of the arguments to match those of the formal parameters.
- If the programmer wishes to use a value of one type in a context that expects another, he or she will need to specify an explicit *type conversion* (type *cast*).

There are three principal cases:
1. The types would be considered structurally equivalent, but the language uses name equivalence.
2. The types have different sets of values, but the intersecting values are represented in the same way .
3.The types have different low-level representations, but we can define some sort of correspondence among their values.
NonconvertingType Casts: In systems programs, one needs to change the type of a value *without* changing the underlying implementation.
To interpret the bits of a value of one type as if they were another type.

- A change of type that does not alter the underlying bits is called a *nonconverting type cast*, or sometimes a *type pun*.
- *Cast* is the term used for conversions in languages like C.

Type compatibility: Most languages do not require equivalence of types in every context.
A value's type must be compatible with that of the context in which it appears.
In an assignment statement, the type of the right hand side must be compatible with that of the left-hand side.
In a subroutine call, the types of any arguments passed into the subroutine must be compatible with the types of the corresponding formal parameters.

- The definition of type compatibility varies greatly from language to language.
- An Ada type S is compatible with an expected type T if and only if (1) S and T are equivalent, (2) one is a subtype of the other, or (3) both are arrays, with the same numbers and types of elements in each dimension.

Coercion: Whenever a language allows a value of one type to be used in a context that expects another, the language implementation must perform an automatic, implicit conversion to the expected type.
This conversion is called a type coercion.

- A coercion may require run-time code to perform a dynamic semantic check, or to convert between low-level representations.
- Ada coercions need the former, never the latter:

```
d : weekday;
k : workday;
type calendar_column is new weekday;
c : calendar_column;
...
k := d; -- run-time check required
d := k; -- no check required; every workday is a weekday
c := d; -- static semantic error;
      -- weekdays and calendar_columns are not compatible
```

To perform this third assignment in Ada we would have to use an explicit conversion:
c := calendar_column(d);
Fortran 90 allows arrays and records to be intermixed if their types have the same *shape*.

Two arrays have the same shape if they have the same number of dimensions, each dimension has the same size.
Field *names* do not matter, nor do the actual high and low bounds of array dimensions.

- C allow arrays and *pointers* to be intermixed in many cases.
- C++ provides an extremely rich, *programmer-extensible* set of coercion rules.
  The programmer can define coercion operations to convert values of the new type to and from existing types.

Overloading and Coercion: An overloaded name can refer to more than one object; the ambiguity must be resolved by context.
Consider the addition of numeric quantities.
In the expression a + b, + may refer to either the integer or the floating-point addition operation.
In a language without coercion, a and b must either both be integer or both be real.

- Ada formalizes the notion of "constant type" for numeric quantities: an integer constant is said to have type universal_integer.
- A floating-point constant is said to have type universal_real.

Universal Reference Types: To facilitate the writing of general-purpose *container* (*collection*) objects that hold references to other objects, several languages provide a *universal* reference type.

- In C and C++, this type is called void *.
- In Clu it is called any; inModula-2, address; inModula-3, refany; in Java, Object; in C#, object.
- Arbitrary l-values can be assigned into an object of universal reference type, with no concern about type safety.
- Here we need to include in the representation of each object a *tag* that indicates its type.
- This approach is common in object-oriented languages, which generally need it for dynamic method binding.

Type Inference: Type checking ensures that the components of an expression have appropriate types.
The result of an arithmetic operator usually has the same type as the operands.
The result of a function call has the type declared in the function's header.
Subranges: For simple arithmetic operators, the principal type system subtlety arises when one or more operands have subrange types.
Given the following Pascal definitions, for example,

```
type  Atype = 0..20;
Btype = 10..20;
var  a : Atype;
     b : Btype;
```

The type of a + b is neither Atype nor Btype, since the possible values range from10 to 40.
This is a new anonymous subrange type with 10 and 40 as bounds.
Composite Types: Most built-in operators in most languages take operands of built-in types. Some operators, can be applied to values of composite types, including aggregates.

- Pascal and Modula, support union (+), intersection (*), and difference (-) on sets of discrete values.
- Set operands are said to have compatible types if their elements have the same base type T.

The MLType System: The most sophisticated form of type inference occurs in certain functional languages like ML, Miranda, and Haskell.
Programmers have the option of declaring the types of objects in these languages, in which case the compiler behaves much like that of a more traditional statically typed language.
Records(Structures) and Variants(Unions): Record types allow related data of heterogeneous types to be stored and manipulated together.
Some languages like Algol 68, C,C++,Common Lisp use the term structure instead of record.
Fortran 90 simply calls its records "types".
Structures in C++ are defined as a special form of class.
C# uses a reference model for variables of class types, and a value model for variables of struct types.
Syntax and Operations:-
In c a simple record might be defined as follows.

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

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

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

We can say,

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

In Fortran 90 and Common Lisp, only the second alternative is permitted: record fields can have record types, but the declarations cannot be lexically nested.

Memory layout and its impact: The fields of a record are usually stored in adjacent locations in memory.
In its symbol table, the compiler keeps track of the offset of each field within each record type.

- For a local object, the base register is the frame pointer.
- The displacement is the sum of the records offset from the register and the fields offset within the record.
- On a RISC machine, a global record is accessed in a similar way, using a dedicated *globals pointer* register as base.
- A packed array of packed records might devote only 15 bytes to each; only every fourth element would be aligned.
- A compiler will implement a packed record without holes, by simply "pushing the fields together."

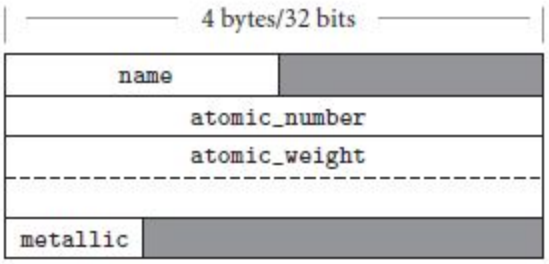


Fig: Likely layout in memory for objects of type element on a 32-bit machine.
Alignment restrictions lead to the shaded "holes."
To access a nonaligned field, it will have to issue a multi-instruction sequence that retrieves the pieces of the field from memory and then reassembles them in a register.

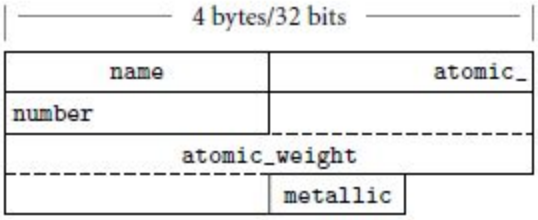


Fig: Likely memory layout for packed element records.
The atomic_number and atomic_weight fields are nonaligned, and can only be read or written via multi-instruction sequences.

- Holes in records waste space.
- Some compilers, sort a record's fields according to the size of their alignment constraints.

- All byte-aligned fields might come first, followed by any half-word aligned fields, word-aligned fields, and double-word–aligned fields.

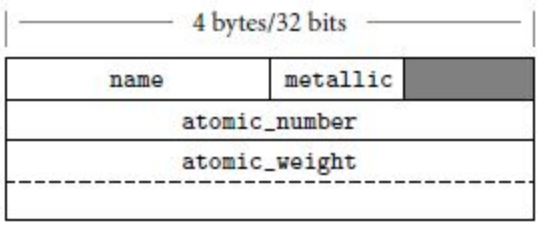


Fig: Rearranging record fields to minimize holes.

By sorting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

With **Statements :** In programs with complicated data structures, manipulating the fields of a deeply nested record can be awkward:

```
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;
```

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): Many languages allowed the programmer to specify that certain variables should be allocated "on top of" one another, sharing the same bytes in memory.

C's syntax was heavily influenced by Algol 68.

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

- The overall size of this union would be that of its largest member (d).
- Exactly which bytes of d would be overlapped by i and b is implementation dependent, and influenced by the relative sizes of types, their alignment constraints etc..
- Unions have been used for two main purposes.
- Unions allow the same set of bytes to be interpreted in different ways at different times. Example is in the case of memory management, where storage may sometimes be treated as unallocated space, sometimes as bookkeeping information etc..
- The second purpose for unions is to represent alternative sets of fields within a record. A record representing an employee, might have several common fields and various other fields such as salaried, hourly or consulting basis.

Array: Arrays are the most common and important composite data types.
Arrays are usually homogeneous.
They can be thought of as a mapping from an index type to a component or element type. Some languages allow nondiscrete index types.
The resulting associative arrays must generally be implemented with hash tables or search trees.
Associative arrays also resemble the dictionary or map types supported by the standard libraries of many object-oriented languages.
Syntax and operations: Most languages refer to an element of an array by appending a subscript delimited by parentheses or square brackets-to the name of the array.
In Fortran and Ada ,one says A(3);in Pascal and C, one says A[3].

- Since parentheses are generally used to delimit the arguments to a subroutine call, square bracket subscript notation has the advantage of distinguishing between the two.

Declarations: One declares an array by appending subscript notation to the syntax that would be used to declare a scalar. In C:
Char upper[26];

In Fortran:
character, dimension (1:26)::upper
character (26) upper //shorthand notation
In C the lower bound of an index range is always zero; the indices of an n-element array are 0.....n-1.
In Fortran the lower bound of the index range is one by default.
Most languages make it easy to declare multidimensional arrays:
mat : array (1..10, 1..10) of real; -- Ada
real, dimension (10,10) :: mat ! Fortran
In Ada,

mat1 : array (1..10, 1..10) of real;
is not the same as
type vector is array (integer range <>) of real;
type matrix is array (integer range <>) of vector (1..10);
mat2 : matrix (1..10);
Variable mat1 is a two-dimensional array; mat2 is an array of one-dimensional arrays.
With the former declaration, we can access individual real numbers as mat1(3, 4); with the latter we must say mat2(3)(4).

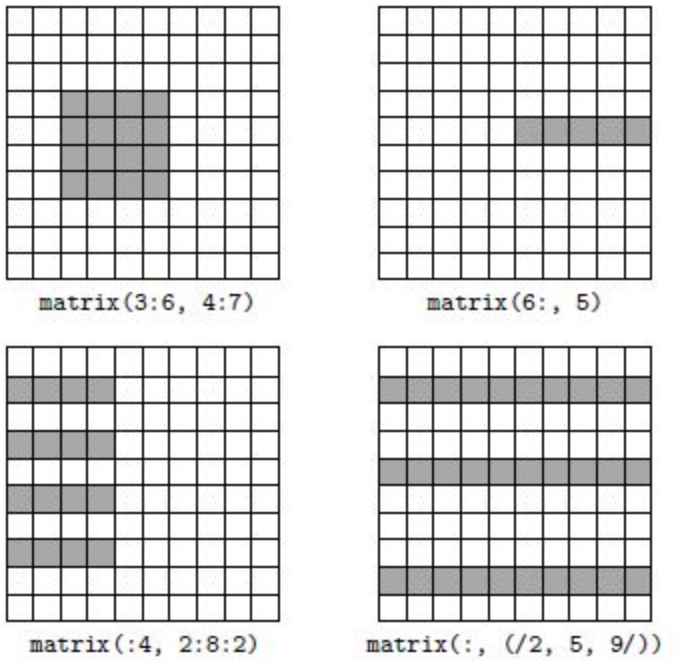
- The two-dimensional array is more elegant, but the array of arrays supports additional operations.
- it allows us to name the rows of mat2 individually and it allows us to take *slices*.

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.
- Fortran 90 has a very rich set of array operations: built in operations that take entire arrays as arguments.

- Slices of the same shape can be intermixed in array operations, even if the arrays from which they were sliced have very different shapes.
- Any of the built in arithmetic operators will take arrays as operands; the result is an array,of the same shape as the operands.
- Ada allows one-dimensional arrays whose elements are discrete to be compared for *lexicographic ordering* : A < B if the first element of A that is not equal to the corresponding element of B is less than that corresponding element.
- Ada also allows the built-in logical operators (or, and, xor) to be applied to Boolean arrays.
- Fortran 90 has a very rich set of *array operations*: built-in operations that take entire arrays as arguments.
- Fortran 90 also provides a huge collection of *intrinsic*, or built-in functions.
- An equally rich set of array operations can be found in Single Assignment C (SAC), a purely functional language for high-performance computing developed by Sven-Bodo Scholz.

Dimensions, Bounds, and Allocation: Storage management is more complex for arrays whose shape is not known until elaboration time.
For these the compiler must arrange not only to allocate space, but also to make shape information available at run time.



Much like the values in the header of an enumeration-controlled loop.

[a : b : c in a subscript indicates positions a, a + c, . . . through b.
If a or b is omitted, the corresponding bound is assumed.

If c is omitted, 1 is assumed.
If c is negative, then we select positions in reverse order.
The slashes in the second subscript of the lower-right example delimit an explicit list of positions.]

- A local array whose shape is known at elaboration time may still be allocated in the stack.
- An array whose size may change during execution must generally be allocated in the heap.
- Global lifetime, static shape: allocate space for the array in static global memory
- Local lifetime, static shape: space can be allocated in the subroutine's stack frame at run time
- Local lifetime, shape bound at elaboration time: an extra level of indirection is required to place the space for the array in the stack frame of its subroutine (Ada, C)
- Arbitrary lifetime, shape bound at elaboration time: at elaboration time either space is allocated or a preexistent reference from another array is assigned (Java, C#)

Dope Vectors: During compilation, the symbol table maintains dimension and bounds information for every array in the program.
For every record, it maintains the offset of every field.
When the number and bounds of 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 these values are not statically known, the compiler must generate code to look them up in a dope vector at run time.

Dope vector; Purposes that it serve: A dope vector will contain the lower bound of each dimension and the size of each dimension other than the last.
If the language implementation performs dynamic semantic checks for out of bounds subscripts in array references, then the dope vector may contain upper bounds as well.

- The contents of the dope vector are initialized at elaboration time, or whenever the number or bounds of dimensions change.

The compiler may use dope vectors not only for dynamic shape arrays, but also for dynamic shape records.
The dope vector for a record typically indicates the offset of each field from the beginning of the record.
Stack Allocation: Subroutine parameters are the simplest example of dynamic shape arrays. Early versions of Pascal required the shape of all arrays to be specified statically.
Standard Pascal allows array parameters to have bounds that are symbolic names rather than constants.

- It calls these parameters *conformant arrays*.
- Ada and C99 support not only conformant arrays, but also *local* arrays of dynamic shape.
- 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*.
Changes in size do not in general occur in FIFO order.
Here fully dynamic arrays must be allocated in the heap.

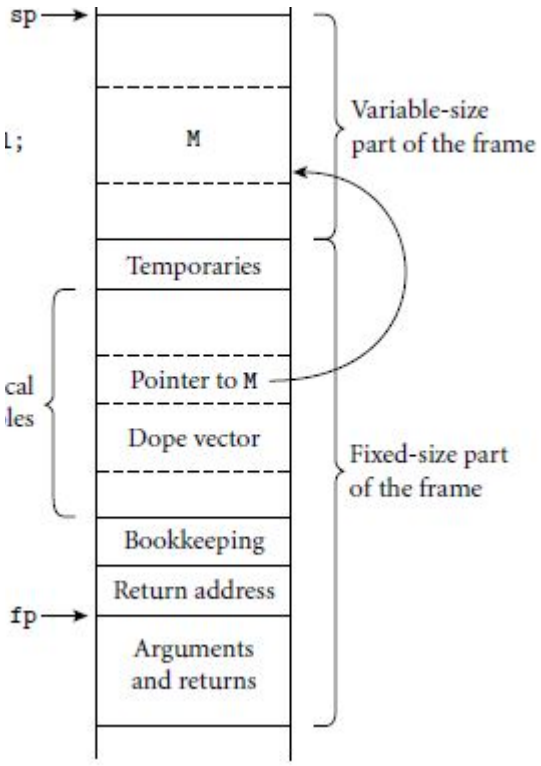


Fig: Elaboration-time allocation of
arrays in Ada or C99.
Here M is a square two dimensional array
whose bounds are determined by a
parameter passed to foo at run time.
The compiler arranges for a pointer to M and a
dope vector to reside at static offsets from
the frame pointer.

Dynamically resizable arrays (other than strings) appear in APL, Common Lisp, and the various scripting languages.
They are also supported by the vector, Vector, and ArrayList classes of the C++, Java, and C# libraries, respectively.
Space for stack-allocated arrays is of course reclaimed automatically by popping the stack.
Allocation in Ada of local arrays whose shape is bound at elaboration time.
//Ada:

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

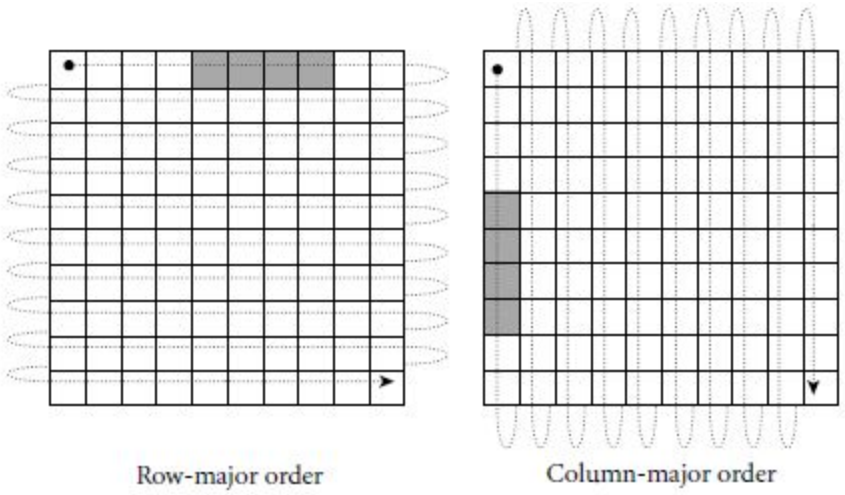
//C99:

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

Memory Layout of Arrays: Arrays in most language implementations are stored in contiguous locations in memory.
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.
For arrays of records, it is common for each subsequent element to be aligned at an address appropriate for any type; small holes between consecutive records may result.
For multidimensional arrays, there are two layouts: row-major order and column-major order

- In row-major order, consecutive locations in memory hold elements that differ by one in the final subscript.
- In column-major order, consecutive locations hold elements that differ by one in the initial subscript.

The difference between row- and column-major layout can be important for programs that use nested loops to access all the elements of a large, multidimensional array.
For a large array, however, lines that are accessed early in the traversal are likely to be evicted to make room for lines accessed later in the traversal.



[In row major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous.
The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating point number, that cache lines are 32 bytes long and that the array begins at a cache line boundary]

Row-Pointer Layout: Allow the rows of an array to lie anywhere in memory, and create an auxiliary array of pointers to the rows.

- Only the contiguous layout is a true multidimensional array.
- This row-pointer memory layout requires more space in most cases but has three potential advantages.
  - ➢ It sometimes allows individual elements of the array to be accessed more quickly, especially on CISC machines with slow multiplication instructions.
  - ➢ This representation is sometimes called a *ragged array*;
  - ➢ It allows a program to construct an array from preexisting rows (possibly scattered throughout memory) without copying.
- C, C++, and C# provide both contiguous and row-pointer organizations for multidimensional arrays

Row-Pointer Layout in C:

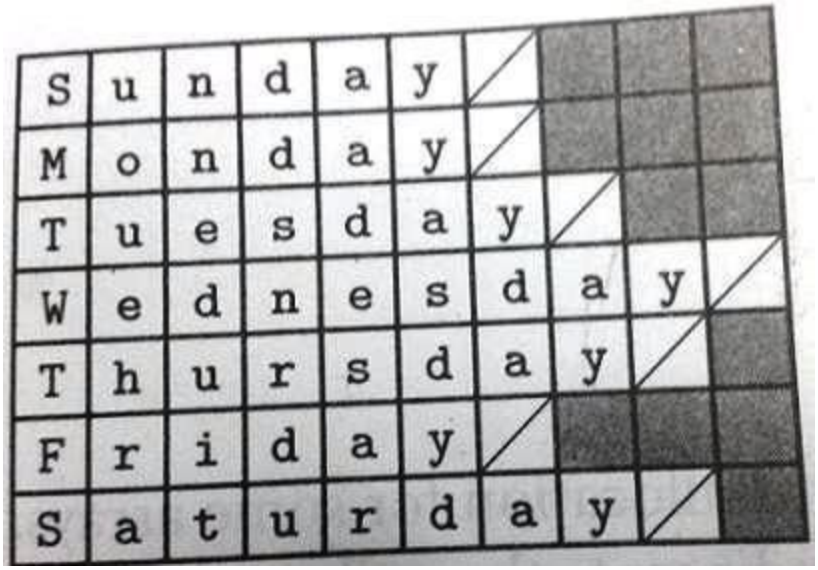
```
char days [ ][10]={
    "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"
};
.....
days [2] [3] = ='s'; /*in Tuesday */
```

The additional space required for the row-pointer organization comes to 21%.

In other cases, row pointers may actually save space.

A Java compiler written in C, for example, would probably use row pointers to store the character-string representations of the 51 Java keywords and word-like literals.

Address Calculations: For the usual contiguous layout of arrays, calculating the address of a particular element is somewhat complicated, but straightforward.

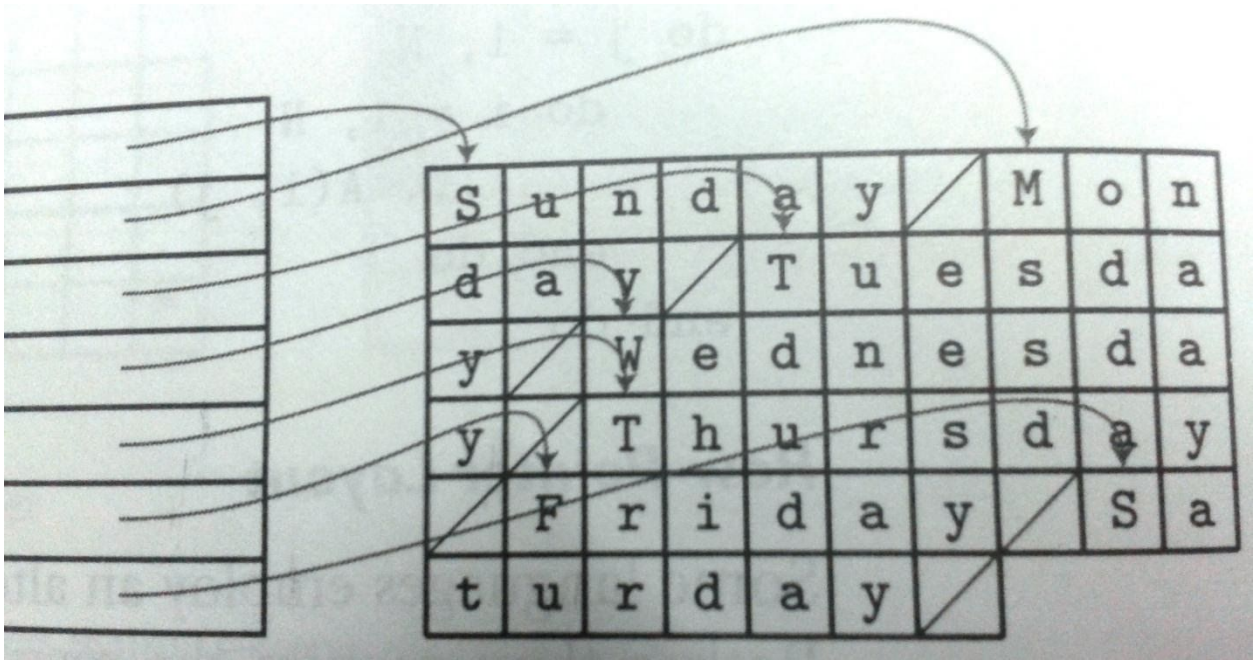


[It is a true two dimensional array.

The slashed boxes are NUL bytes; the shaded areas are holes.]

```
char *days [ ] = {
```

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



[It is a ragged array of pointers to arrays of characters.]

Suppose a compiler is given the following declaration for a three-dimensional array:

A : array [*L*1 . .*U*1] of array [*L*2 . .*U*2] of array [*L*3 . .*U*3] of elem type;

define constants for the sizes of the three dimensions:
*S*3 = size of elem type
*S*2 = (*U*3 − *L*3 + 1) × *S*3
*S*1 = (*U*2 − *L*2 + 1) × *S*2

Here the size of a row (*S*2) is the size of an individual element (*S*3) times the number of elements in a row (assuming row-major layout).
The size of a plane (*S*1) is the size of a row (*S*2) times the number of rows in a plane.
The address of A[i, j, k] is then,
address of A
+ (*i* − *L*1) × *S*1
+ (*j* − *L*2) × *S*2
+ (*k* − *L*3) × *S*3
If A is a local variable of a subroutine, then the address of A can be decomposed into a static offset plus the contents of the frame pointer at run time.

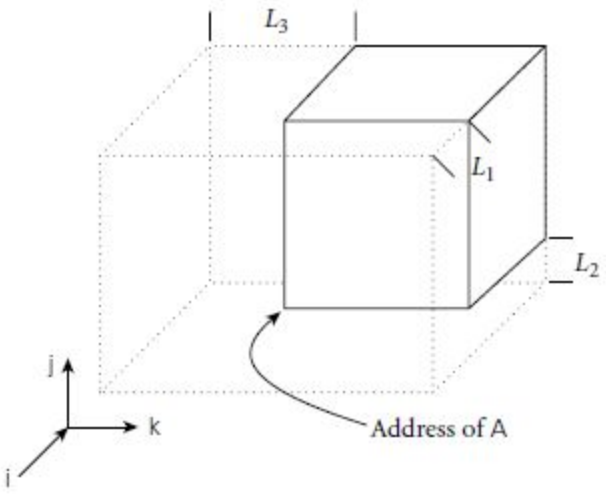


Fig: Virtual location of an array with nonzero lower bounds.

By computing the constant portions of an array index at compile time, we effectively index into an array whose starting address is offset in memory, but whose lower bounds are all zero.

String representations in programming languages: In many languages, a string is simply an array of characters.

In other languages, strings have special status, with operations that are not available for arrays of other sorts.

It is easier to provide special features for strings than for arrays in general because strings are one-dimensional.

Manipulation of variable-length strings is fundamental to a huge number of computer applications.

Powerful string facilities are found in various scripting languages such as Perl, Python and Ruby.

Lisp, Icon, ML, Java, C# allow the length of a string-valued variable to change over its lifetime, requiring that space be allocated by a block or chain of blocks in the heap.

Many languages, including C and its descendants, distinguish between literal characters and literal strings.

Other languages (e.g., Pascal) make no distinction: a character is just a string of length one.

Most languages also provide *escape sequences* that allow nonprinting characters and quote marks to appear inside of strings.

An arbitrary character can be represented by a backslash followed by (a) 1 to 3 octal (base-8) digits, (b) an x and one or more hexadecimal (base-16) digits, (c) a u and exactly four hexadecimal digits, or (d) a U and exactly eight hexadecimal digits.

The variable is to be implemented as a contiguous array of characters in the current stack frame.

- Pascal and Ada support a few string operations, including assignment and comparison for lexicographic ordering.
- Given the declaration char *s, the statement s = "abc" makes s point to the constant "abc" in static storage.

A string variable is a *reference* to a string.

Assigning a new value to such a variable makes it refer to a different object.

Sets: A set is an unordered collection of an arbitrary number of distinct values of a common type.

The type from which elements of a set are drawn is known as the base or universe type.

Introduced by Pascal.

Pascal supports sets of any discrete type, and provides union, intersection, and difference operations:

```
var A,B,C :set of char;
    D,E : set of weekday;
.....
A := B + C;
A := B * C;
A := B - C;
```

Many ways to implement sets, including arrays, hash tables, and various forms of trees

The most common implementation employs a bit vector whose length (in bits) is the number of distinct values of the base type.

Operations on bit-vector sets can make use of fast logical instructions on most machines.

There are many ways to implement sets, including arrays, hash tables, and various forms of trees.

For discrete base types with a modest number of elements.

A *characteristic array* is a particularly appealing implementation: it 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 in the bit vector indicates that the *k*th element of the base type is a member of the set.

Trade offs between Pointers and the Recursive Types that arise naturally in a language with a reference model of variables: A recursive type is one whose objects may contain one or more references to other objects of the type.

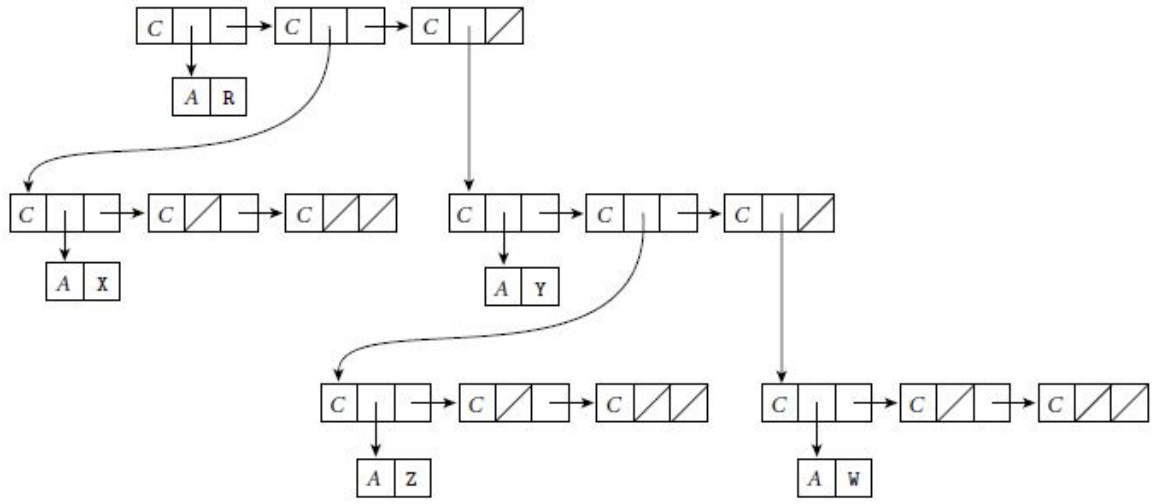
- Recursive types are used to build a wide variety of “linked” data structures, including lists and trees.
- In languages that use a reference model of variables, it is easy for a record of type foo to include a reference to another record of type foo: every *is* a reference anyway.
- Recursive types require the notion of a *pointer*: a variable (or field) whose value is a reference to some object.
- Automatic storage reclamation (*garbage collection*) dramatically simplifies the programmer’s task, but imposes certain run-time costs.

Syntax and Operations:- Operations on pointers include allocation and deallocation of objects in the heap, dereferencing of pointers to access the objects to which they point, and assignment of one pointer into another.

- In C, Pascal, or Ada which employ a value model, the assignment A: = B puts the value of B into A.
- If we want B to refer to an object and we want A: = B to make A refer to the object to which B refers, then A and B must be pointers.

- The assignment A := B in Java places the value of B into A if A and B are of built-in type; it makes A refer to the object to which B refers if A and B are of user-defined type.

Reference Model: In Lisp, which uses a reference model of variables but is not statically typed, tree could be specified textually as (# \ R (# \X ( ) ( )) (# \ Y (# \ Z ( ) ( )) (# \ W ( ) ( )))).



[Implementation of a tree in Lisp, A diagonal slash through a box indicates a null pointer.
The C and A tags serve to distinguish the two kinds of memory blocks: cons cells and blocks containing atoms].

- When writing in a functional style, one often finds a need for *types* that are *mutually recursive*.
- In a compiler, for example, it is likely that symbol table records and syntax tree nodes will need to refer to each other.
- A syntax tree node that represents a subroutine call will need to refer to the symbol table record that represents the subroutine.
- The symbol table record, will need to refer to the syntax tree node at the root of the subtree that represents the subroutine's code.

Value Model: In Pascal tree data types would be declared as follows:

```
type chr_tree_ptr = ^chr_tree;
     chr_tree = record
          left,right : chr_tree_ptr;
          val : char
        end;
```

In C:

```
struct chr_tree
{
  struct chr_tree * left, *right;
  char val;
};
```

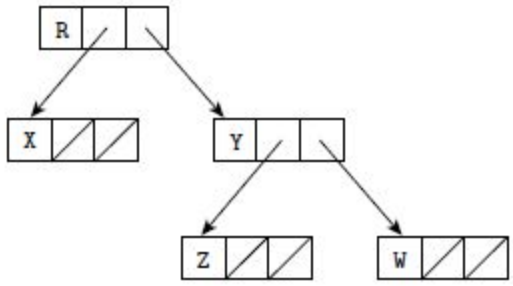


Fig: Typical implementation of a tree in a language with explicit pointers.
As in a diagonal slash through a box indicates a null pointer.

In Ada:
my_ptr := new chr_tree;
In C:
my_ptr = malloc(sizeof(struct chr_tree));

C's malloc is defined as a library function, not a built-in part of the language.
The programmer must specify the size of the allocated object explicitly, and while the return value (of type void*) can be assigned into any pointer, the assignment is not type-safe.
C++, Java, and C# replace malloc with a built-in, type-safe new:
my_ptr = new chr_tree(*arg list*);

- The C++/Java/C# new will automatically call any user-specified *constructor* (initialization) function, passing the specified argument list.
- To access the object referred to by a pointer, most languages use an explicit dereferencing operator.
- In Pascal and Modula this operator takes the form of a postfix "up-arrow":

my_ptrˆ.val := 'X';
In Ada dot-based syntax can be used to access either a field of the record foo or a field of the record*pointed to* by foo, depending on the type of foo.
Pointers and Arrays in C : Pointers and arrays are closely linked in C.
Consider the following declarations:
int n;
int *a; /* pointer to integer */
int b[10]; /* array of 10 integers */
Now all of the following are valid:
1. a = b; /* make a point to the initial element of b */
2. n = a[3];
3. n = *(a+3); /* equivalent to previous line */

- In most contexts, an unsubscripted array name in C is automatically converted to a pointer to the array's first element as shown here in line 1.

- Lines 3 illustrate *pointer arithmetic*: 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.
- C allows pointers to be subtracted from one another or compared for ordering, provided that they refer to elements of the same array.

A declaration must allow the compiler to determine the size of the *elements* of an array or, equivalently, the size of the objects referred to by a pointer.
Neither int a[ ][ ] nor int (*a)[ ] is a valid variable or parameter declaration: neither provides the compiler with the size information it needs to generate code for a + i or a[i].

- The built-in sizeof operator returns the size in bytes of an object or type.
- When given a pointer as argument it returns the size of the pointer itself.

Dangling References: When a heap allocated object is no longer live, a long running program needs to reclaim the objects space.
Stack objects are reclaimed automatically as part of the subroutine calling sequence.
There are two alternatives to reclaime heap objects.
Languages like Pascal, C, and C++ require the programmer to reclaim an object explicitly.

- C++ provides additional functionality: it automatically calls any user-provided *destructor* function for the object.
- A destructor can reclaim space for subsidiary objects, remove the object from indices or tables, print messages etc..

In Pascal:

```
dispose(my_ptr);
```

In C:

```
free (my_ptr);
```

A dangling reference is a live pointer that no longer points to a valid object.
Dangling reference to a stack variable in C++:

```
int i=3;
int *p = &i;
....
void foo( )
 {
   int n=5;
   p=&n;
 }
  .....
cout<<*p;  //prints 3
foo( );
 ....
cout<< *p;  //Undefined behavior: n is no longer live
```

In a language with explicit reclamation of heap objects, a dangling reference is created whenever the programmer reclaims an object to which pointers still refer.
Even if the reclamation operation were to change its argument to a null pointer, this would not solve the problem, because *other* pointers might still refer to the same object.

A program that uses a dangling reference may read or write bits in memory that are now part of some other object.

Garbage Collection: Explicit reclamation of heap objects is a serious burden on the programmer and a major source of bugs.

The code required to keep track of object lifetimes makes programs more difficult to design, implement and maintain.

Automatic garbage collection has become popular for imperative languages as well.

It tends to be slower than manual reclamation.

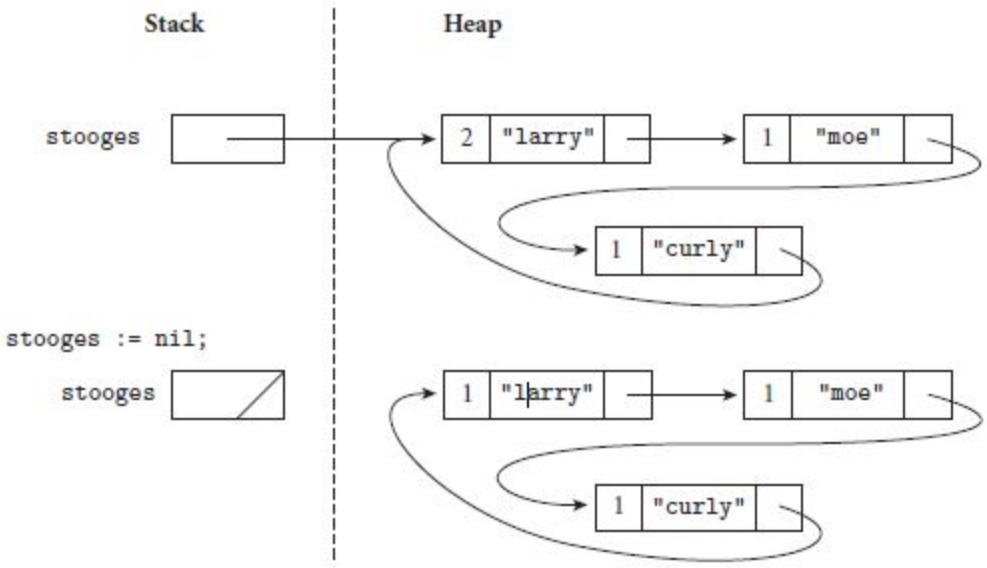
Reference counts: The simplest garbage collection technique simply places a counter in each object that keeps track of the number of pointers that refer to the object.

When the object is created, this reference count is set to 1.

- When one pointer is assigned into another, the run time system decrements the reference count of the object formerly referred to by the assignments left hand side.
- It increments the count of the object referred to by the right hand side.
- When a reference count reaches zero, its object can be reclaimed.
- To prevent the collector from following garbage addresses, each pointer must be initialized to null at elaboration time.

Type descriptors are simply a table that lists the offsets within the type at which pointers can be found, together with the addresses of descriptors for the types of the objects referred to by those pointers.

- For a tagged variant record type, the descriptor is a bit more complicated.
- It must contain a list of values (or ranges) for the tag, together with a table for the corresponding variant.
- For *untagged* variant records, reference counts work only if the language is strongly typed.
- **Tracing Collection:** A better definition of a “useful” object is one that can be reached by following a chain of valid pointers starting from something that has a name.
- The blocks in the bottom half of are useless, even though their reference counts are nonzero.
- Tracing collectors work by recursively exploring the heap, starting from external pointers, to determine what is useful.



[Reference counts and circular lists]

Differences among mark- and –sweep, stop- and-copy, pointer reversal and generational garbage collection: 1). Mark –and-Sweep: - The classic mechanism to identify useless blocks.
It proceeds in there main steps:

a). The collector walks through the heap, tentatively marking every block as useless.

b). Beginning with all pointers outside the heap, the collector recursively explores all linked data structures in the program, marking each newly discovered block as useful.

c). The collector again walks through the heap, moving every block that is still marked useless to the free list.

2). Pointer Reversal

When the collector explores the path to a given block, it reverses the pointers it follows, so that each points back to the previous block instead of forward to the next.

Each reversed pointer must be marked (indicated with a shaded box), to distinguish it from other, forward pointers in the same block.

To return from block X to block U the collector will use the reversed pointer in U to restore its notion of previous block (T).

It will then flip the reversed pointer back to X and update its notion of current block to U.

Fig. shows Heap exploration via pointer reversal.

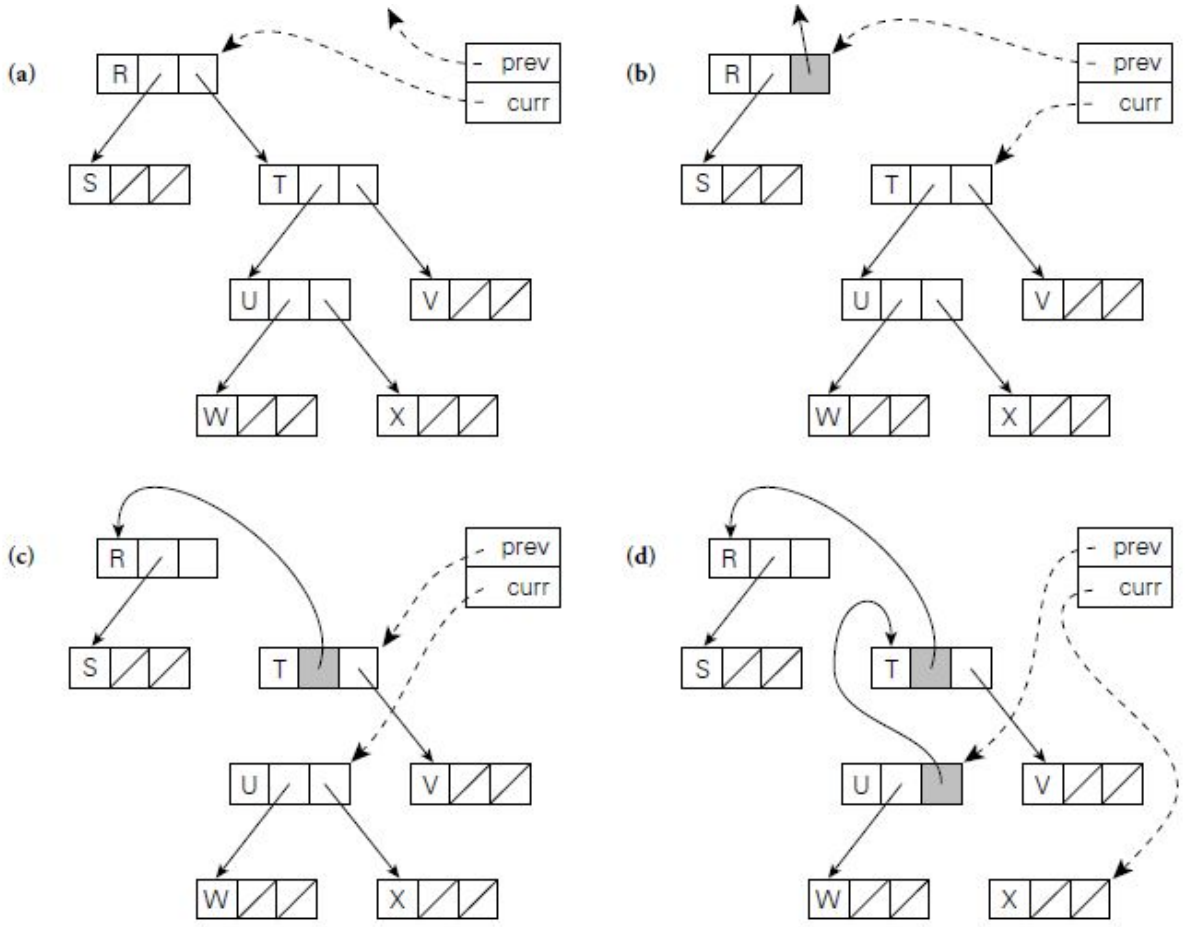
The block currently under examination is indicated by the curr pointer.

The previous block is indicated by the prev pointer.

As the garbage collector moves from one block to the next, it changes the pointer it follows to refer back to the previous block.

When it returns to a block it restores the pointer.

3). Stop and Copy: - In a language with variable size heap blocks, the garbage collector can reduce external fragmentation by performing storage compaction.



Many garbage collector employ a technique known as stop and copy that achieves compaction.
Specifically they divide the heap into two regions of equal size.
All allocation happens in the first half.
When this half is full, the collector begins its exploration of reachable data structures.
Each reachable blocks is copied into the second half of the heap, is overwritten with a useful flag and a pointer to the new location.
When the collector finishes its exploration, all useful objects have been moved into the second half of the heap, and nothing in the first half is needed anymore.
4). Generational collection: - The heap is divided into multiple regions.
When space runs low the collector first examines the youngest region, which it assumes is likely to have the highest proportion of garbage.
Only if it is unable to reclaim sufficient space in this region does the collector examine the next older region.
To avoid leaking storage in long running systems, the collector must be prepared, if necessary, to examine the entire heap.
Any object that survives some small number of collections in its current region is promoted to the next older region.

At each pointer assignment, the compiler generates code to check whether the new value is an old to new pointer.
if so it adds the pointer to a hidden list accessible to the collector.
This instrumentation on assignments is known as a *write barrier*.
5). Conservative Collection: When space runs low, the collector tentatively marks all blocks in the heap as useless.
It then scans all word aligned quantities in the stack and in global storage.
If any of these words appears to contain the address of something in the heap, the collector marks the block that contains that address as useful.
The collector then scans all word-aligned quantities in the block, and marks as useful any other blocks whose addresses are found therein.
Finally the collector reclaims any blocks that are still marked useless.
There is only a very small probability that some word in memory that is not a pointer will happen to contain a bit pattern that looks like one of those addresses. The algorithm is completely safe so long as the programmer never "hides" a pointer.
Lists: A list is defined recursively as either the empty list or a pair consisting of an object and another list.
Lists are ideally suited to programming in functional and logic languages, which do most of their work via recursion and higher order functions.
In Lisp, a program is a list, and can extended itself at run time by constructing a list and executing it.
Lists can also be used in imperative programs.
Clu provides a built-in type constructor for lists, and a list class is easy to write in most object-oriented languages.
Lists in ML and Lisp: Lists in ML are homogeneous: every element of the list must have the same type.
Lisp lists, are heterogeneous: any object may be placed in a list, so long as it is never used in an inconsistent fashion.

- An ML list is usually a chain of blocks, each of which contains an element and a pointer to the next block.
- A Lisp list is a chain of cons cells, each of which contains two pointers, one to the element and one to the next cons cell.
- An ML list is enclosed in square brackets, with elements separated by commas:[a, b, c, d]
  A Lisp list is enclosed in parentheses, with elements separated by white space: (a b c d).
- Lisp systems provide a more general, *dotted* list notation that captures both proper and improper lists.
- A dotted list is either an atom (possibly null) or a pair consisting of two dotted lists separated by a period and enclosed in parentheses.
- The list (a . (b . (c . d))) is improper; its final cons cell contains a pointer to d in the second position, where a pointer to a list is normally required.
- Programs are lists in Lisp, Lisp must distinguish between lists that are to be evaluated and lists that are to be left "as is," as structures.

In Lisp:
(cons ‘a ‘(b)) => (a b)
(car ‘(a b)) => a
(car nil) => ??
(cdr ‘(a b c)) => (b c)
(cdr ‘(a)) => nil
(cdr nil) =>??
(append ‘(a b) ‘(c d)) => (a b c d)
Here we have used => to mean “evaluates to”.
The car and cdr of the empty list (nil) are defined to be nil in Common Lisp.

- Miranda, Haskell, Python, and F# provide lists that resemble those of ML, but with an important additional mechanism, known as *list comprehensions*.
- These are adapted from traditional mathematical set notation.
- A common form comprises an expression, an enumerator and one or more filters.
- In Haskell, the following denotes a list of the squares of all odd numbers less than 100:

```
[i*i | i <- [1..100], i `mod` 2 == 1]
```

In Python we would write

```
[i*i for i in range(1, 100) if i % 2 == 1]
```

Files and Input/Output : We can distinguish between *interactive* I/O and I/Owith files.
Input/output facilities allow a program to communicate with the outside world.
Interactive I/O generally implies communication with human users or physical devices, which work in parallel with the running program.

- Files may be further categorized into those that are temporary and those that are persistent.
- Temporary files exist for the duration of a single program run; their purpose is to store information that is too large to fit in the memory available to the program.
- Persistent 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.
- Some languages provide built in file data types and special syntactic constructs for I/O. The principal advantage of language integration is the ability to employ non-subroutine call syntax, and to perform operations that may not otherwise be available to library routines.
- A purely library-based approach to I/O, may keep a substantial amount of “clutter” out of the language definition.

EqualityTesting and Assignment : Consider for example the problem of comparing two character strings.
Should the expression s = t determine whether s and t
are aliases for one another?
occupy storage that is bit-wise identical over its full length?
contain the same sequence of characters?
etc..
The second of these tests is probably too low-level to be of interest in most programs.

It suggests the possibility that a comparison might fail because of garbage in currently unused portions of the space reserved for a string.
In many cases the definition of equality boils down to the distinction between l-values and r-values.
In the presence of references, should expressions be considered equal only if they refer to the same object or also if the objects to which they refer are in some sense equal?
The first option that refer to the same object is known as a *shallow* comparison.
The second that refer to equal objects is called a *deep* comparison.
Under a reference model of variables, a shallow assignment a := b will make a refer to the object to which b refers.
Scheme, has three general-purpose equality-testing functions:
(eq? a b) ; do a and b refer to the same object?
(eqv? a b) ; are a and b known to be semantically equivalent?
(equal? a b) ; do a and b have the same recursive structure?

Both eq? and eqv? perform a shallow comparison.
The simpler eq? behaves as one would expect for Booleans, symbols (names), and pairs but can have implementation-defined behavior on numbers, characters, and strings:
(eq? #t #t) =⇒ #t (true)
(eq? 'foo 'foo) =⇒ #t
(eq? '(a b) '(a b)) =⇒ #f (false); created by separate cons-es
(eq? 2 2) =⇒ *implementation dependent*
(eq? "foo" "foo") =⇒ *implementation dependent*
Numeric, character, and string tests will always work the same way; if (eq? 2 2) returns true, then (eq? 37 37) will return true also.

- The exact rules that govern the situations in which eqv? is guaranteed to return true or false are quite involved.
- The equal? predicate may lead to an infinite loop if the programmer has used the imperative features of Scheme to create a circular list.
- Deep assignments are relatively rare.
- They are used primarily in distributed computing, and in particular for parameter passing in remote procedure call (RPC) systems.
- Languages with sophisticated data abstraction mechanisms usually allow the programmer to define the comparison and assignment operators for each new data type which is not allowed.