KTU NOTES

Module II

31. What is the difference between strict and loose name equivalence.
(3)

Name equivalence is based on the lexical occurrence of type definitions.
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.

32. What do you mean by holes in records? Describe with example.
(3)

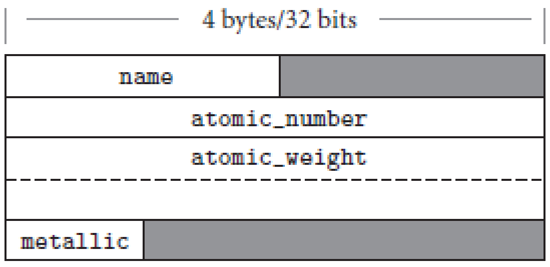


Fig: Likely layout in memory for objects of type element on a 32-bit machine. Alignment restrictions lead to the shaded "holes."
To access a nonaligned field, it will have to issue a multi-instruction sequence that retrieves the pieces of the field from memory and then reassembles them in a register.

33. What do you mean by bit vector representation of sets? (3)

There are 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.

34. a. Explain about Type checking. (6)

b.Differentiate between structural equivalence and named equivalence with examples.
(4)

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.

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.
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:
<u>Name vs structural equivalence</u>

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.

b. Describe the meaning of Type. Why C is not a strongly typed language? (3)

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.
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. C is not strongly typed.

35. a. Write notes on Records. (6)

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."

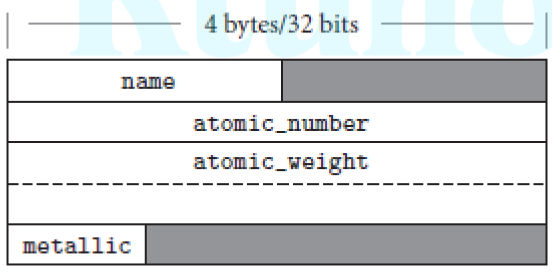


Fig: Likely layout in memory for objects of type element on a 32-bit machine.
Alignment restrictions lead to the shaded "holes."
To access a nonaligned field, it will have to issue a multi-instruction sequence that retrieves the pieces of the field from memory and then reassembles them in a register.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

b. Describe about Variant Records. (4)

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.

36. Differentiate between enumeration and subrange data types.
(3)

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;
```

37. a. What is a type clash?

Also Differentiate between type coercion, type conversion and type casts. (4)

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.
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.
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.

b. What is an array? What are its features ? Write the syntax and discuss its operations. (4)

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.

<u>Syntax and operations</u>: 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.
<u>Declarations</u>: 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*.

b. What are Type systems? (4)

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.

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.

39. a. What are Numeric and Composite Data types?
(5)

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.

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.

b. Describe about Type compatibility and Type Inference? (5)

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.
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.

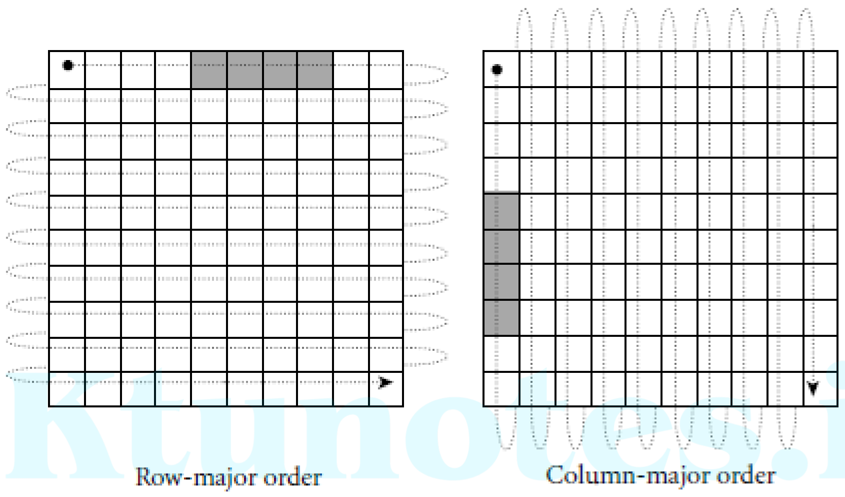
The most sophisticated form of type inference occurs in certain functional languages like ML, Miranda, and Haskell.
Programmers have the option of declaring the types of objects in these languages, in which case the compiler behaves much like that of a more traditional statically typed language

41. a. Explain the difference between row-major and column-major layout for contiguously allocated arrays. Why does a programmer need to know which layout the compiler uses? Why do most language designers consider row major layout to be better? (4)

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]

b. Differentiate between Scalar and Discrete types. (3)

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.
42. Discuss about the Set types of Pascal. (3)

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 kth position in the bit vector indicates that the kth element of the base type is a member of the set.

44. What are Data types? List the different types.
(3)

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
It also includes Numeric types, Composite types etc..

45. What are Non converting Type Casts? (3)

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.

46. a. What are Universal Reference Types? (3)

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.

b. Discuss about the representation and memory layout of Records. (4)

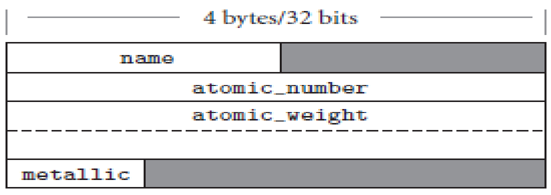


Fig: Likely layout in memory for objects of type element on a 32-bit machine.
Alignment restrictions lead to the shaded "holes."
To access a nonaligned field, it will have to issue a multi-instruction sequence that retrieves the pieces of the field from memory and then reassembles them in a register.

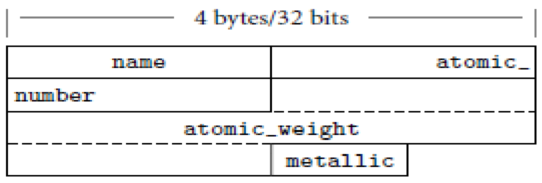
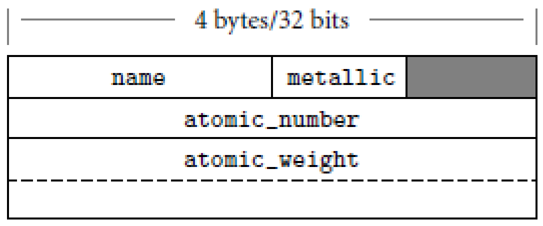


Fig: Likely memory layout for packed element records.
The atomic_number and atomic_weight fields are nonaligned, and can only be read or written via multi-instruction sequences.
Holes in records waste space.
Some compilers, sort a record's fields according to the size of their alignment constraints.



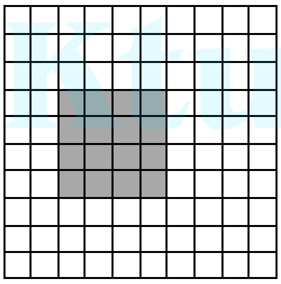
All byte-aligned fields might come first, followed by any half-word aligned fields, word-aligned fields, and double-word–aligned fields.
Fig above shows, Rearranging record fields to minimize holes.
By sorting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

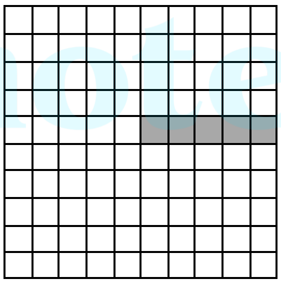
47. Discuss about Slices and Array operations. (5)

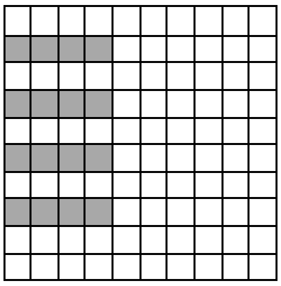
A slice or section is a rectangular portion of an array.
A slice is simply a contiguous range of elements in a one- dimensional array.
Fortran 90 has a very rich set of array operations: built in operations that take entire arrays as arguments.

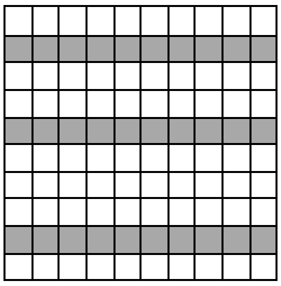
Slices of the same shape can be intermixed in array operations, even if the arrays from which they were sliced have very different shapes.
Any of the built in arithmetic operators will take arrays as operands; the result is an array,of the same shape as the operands.
Ada allows one-dimensional arrays 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.

In the slice, [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.]


`matrix(3:6, 4:7)`


`matrix(6:, 5)`


`matrix(:4, 2:8:2)`


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

48. a. What are Dope Vectors? What are its purposes?
(3)

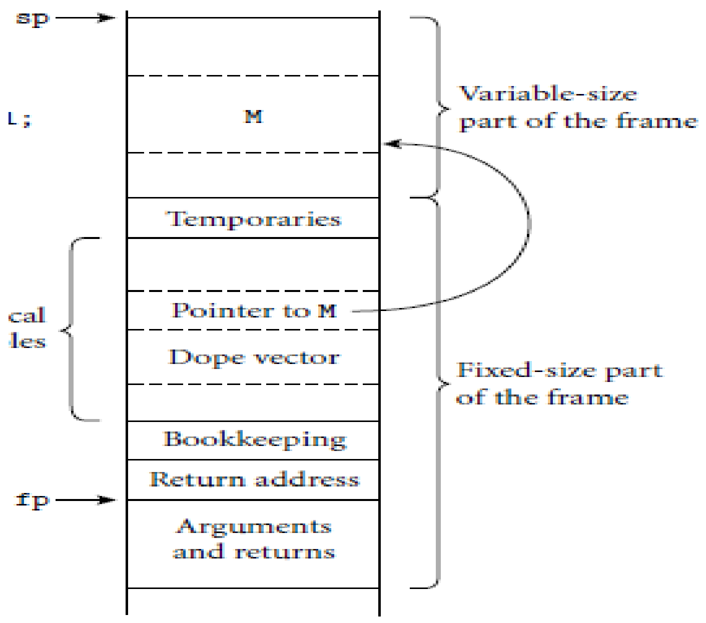
During compilation, the symbol table maintains dimension and bounds information for every array in the program.
For every record, it maintains the offset of every field.

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.

b. Discuss about Heap allocation in arrays. (4)

Discuss about Stack allocation in arrays.
(3)

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. Fig: Elaboration-time allocation of arrays in Ada or C99.



Local variables

<u>Heap</u> <u>Allocation</u>:

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.

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];
    ………
}
```

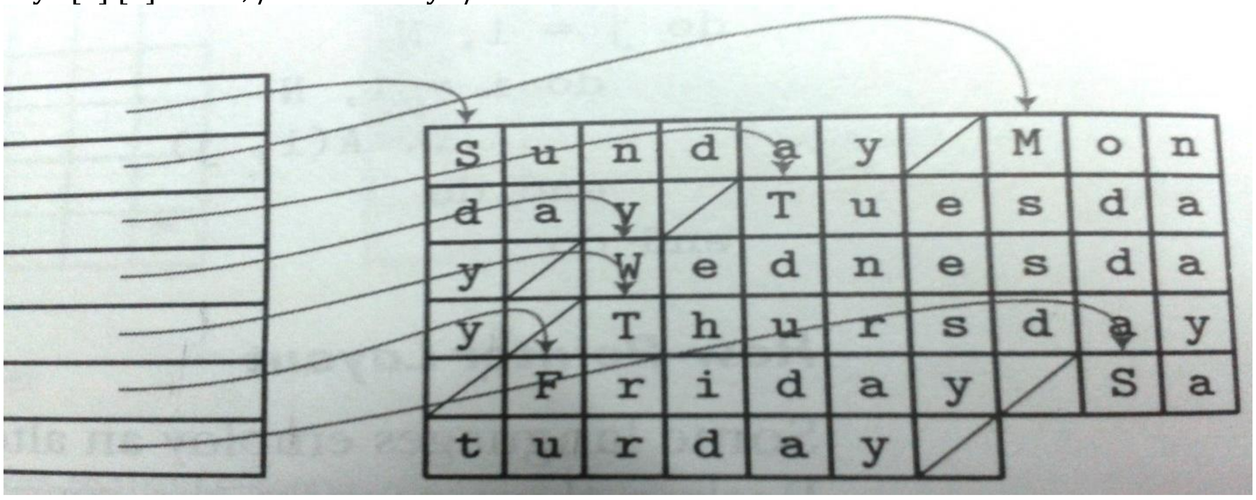
50. a. Discuss about the Row Pointer Layout in C. (5)

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
<u>Row-Pointer Layout in C:</u>

```
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.
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 */
```



b. Describe about the address calculation in the case of 3-D Arrays. (4)

Suppose a compiler is given the following declaration for a three-dimensional array:
A : array [$L1 \,.\,. U1$] of array [$L2 \,.\,. U2$] of array [$L3 \,.\,. U3$] of elem type;
define constants for the sizes of the three dimensions:
$S3$ = size of elem type
$S2 = (U3 - L3 + 1) \times S3$
$S1 = (U2 - L2 + 1) \times S2$

Here the size of a row ($S2$) is the size of an individual element ($S3$) times the number of elements in a row (assuming row-major layout).
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
$+ (i - L1) \times S1$
$+ (j - L2) \times S2$
$+ (k - L3) \times S3$
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.

51. a. Explain about String representation in programming languages.
(5)

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.

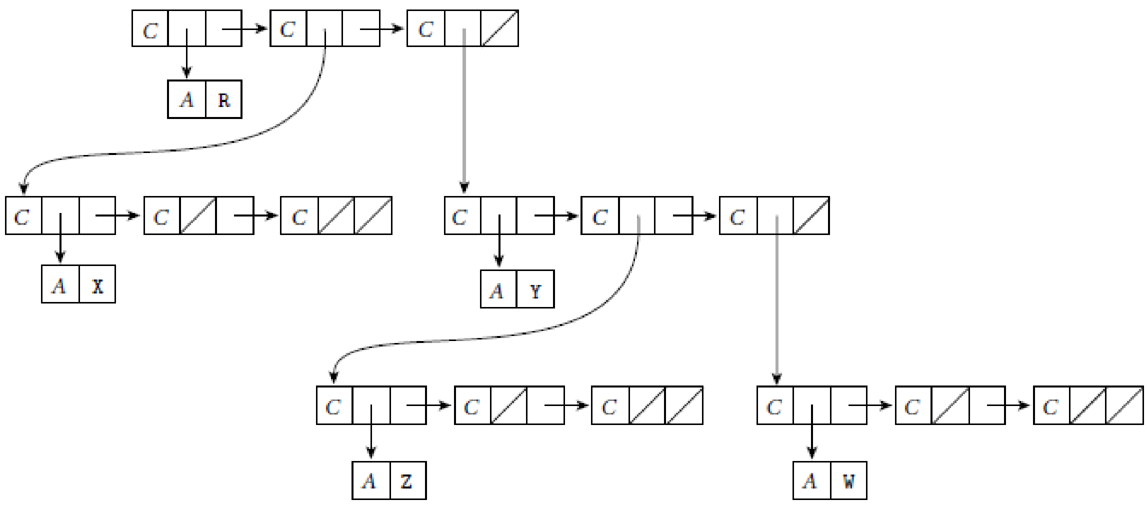
b. Explain about Pointers and Recursive types. Discuss about the syntax and operations. (5)

Discuss the implementation of tree with pointers or, describe the value model in the case of Ada, Pascal and C. (5)

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.

<u>Reference Model</u>: 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;
};
```

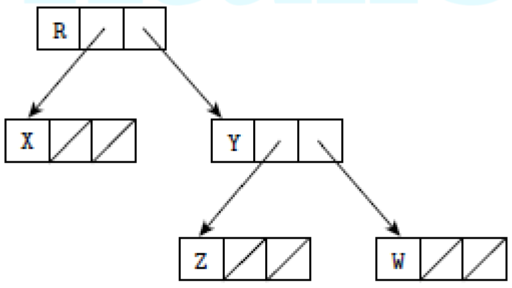


Fig: Typical implementation of a tree in a language with explicit pointers.
As in a diagonal slash through a box indicates a null pointer.

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_ptr^.val := 'X';
In Ada dot-based syntax can be used to access either a field of the record foo or a field of the recordpointed to by foo, depending on the type of foo.

b. Discuss about Pointers and Arrays in C. (4)

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.

53. a. Explain about Dangling references. (5)

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;
 }
 …..
```

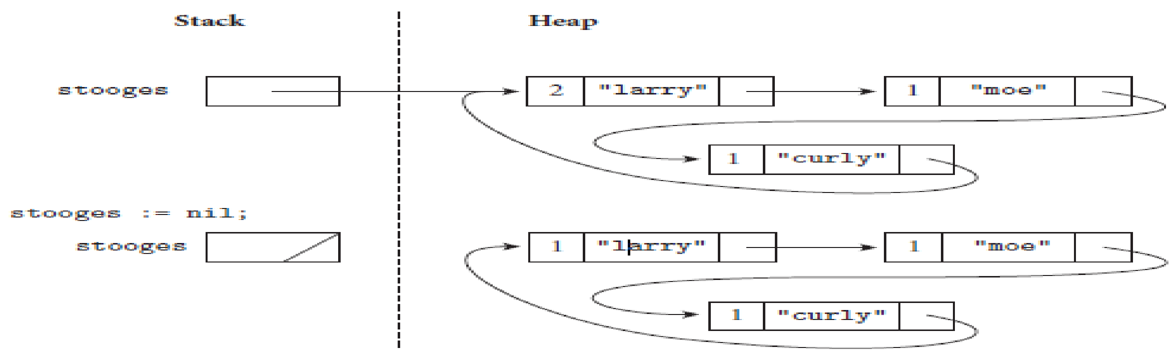
```
cout<<*p;  //prints 3
foo( );
 ….
cout<< *p;  //Undefined behavior: n is no longer live
```

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.

b. Describe about Garbage collection and Reference counts.
(5)

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.



54. What is Tracing collection? **(3)**

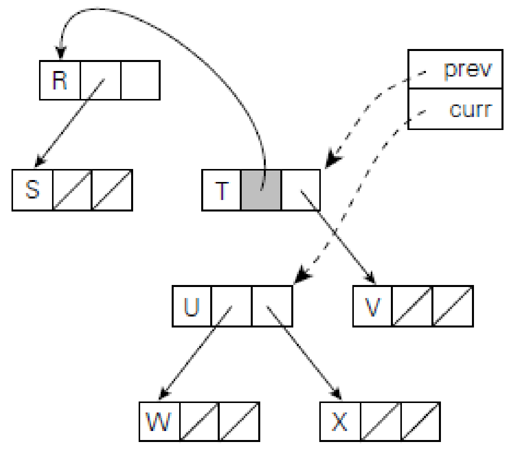
- A better definition of a “useful” object is one that can be reached by following a chain of valid pointers starting from something that has a name.
- The blocks in the bottom half of are useless, even though their reference counts are nonzero.
- Tracing collectors work by recursively exploring the heap, starting from external pointers, to determine what is useful.

55. a. Describe about Mark and Sweep mechanism.
(3)

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.

b. What is Pointer reversal? (4)

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.

(c)


c. What do you mean by Stop and Copy mechanism?
(3)

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

56. What is Conservative Collection and generational garbage collection?
(3)

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*.

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.
57. a. Discuss about the features of Lists? Explain with examples.
(5)

Discuss about the operations on Lists.
(4)

A list is defined recursively as either the empty list or a pair consisting of an object and another list.

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.

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]
```

58. What are Files? What are its features?
(3)
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.

59. What are the features of Equality testing and assignment?
(5)

Discuss about the implementation of Equality testing functions. (4)

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.

Chapter 7

Expressions and Assignment Statements

Chapter 7 Topics

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Chapter 7

Expressions and Assignment Statements

Introduction

- Expressions are the fundamental means of specifying computations in a programming language.
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation.
- Essence of imperative languages is dominant role of assignment statements.

Arithmetic Expressions

- Their evaluation was one of the motivations for the development of the first programming languages.
- Most of the characteristics of arithmetic expressions in programming languages were inherited from conventions that had evolved in math.
- Arithmetic expressions consist of operators, operands, parentheses, and function calls.
- The operators can be **unary**, or **binary**. C-based languages include a **ternary** operator, which has three operands (conditional expression).
- The purpose of an arithmetic expression is to specify an arithmetic computation.
- An implementation of such a computation must cause two actions:
    - o Fetching the operands from memory
    - o Executing the arithmetic operations on those operands.
- Design issues for arithmetic expressions:
  1. What are the operator **precedence** rules?
  2. What are the **operator associativity** rules?
  3. What is the **order of operand evaluation**?
  4. Are there restrictions on operand evaluation **side effects**?
  5. Does the language allow user-defined **operator overloading**?
  6. What **mode mixing** is allowed in expressions?

Operator Evaluation Order

1. Precedence

- The operator precedence rules for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated (“adjacent” means they are separated by at most one operand).
- Typical precedence levels:
  1. parentheses
  2. unary operators
  3. ** (if the language supports it)
  4. *, /

5. +, -

- Many languages also include unary versions of addition and subtraction.
- Unary addition **(+)** is called the **identity operator** because it usually has no associated operation and thus has no effect on its operand.
- In Java, unary plus actually does have an effect when its operand is short or byte. An implicit conversion of short and byte operands to int type takes place.
- Unary minus operator (-) Ex:

```
A + (- B) * C       // is legal
A + - B * C         // is illegal
```

2. Associativity

- The operator associativity rules for expression evaluation define the order in which adjacent operators with the **same precedence** level are evaluated. An operator can be either left or right associative.
- Typical associativity rules:
    - o Left to right, except **, which is right to left
    - o Sometimes unary operators associate right to left (e.g., FORTRAN)
- Ex: (Java)

```
a – b + c           // left to right
```

- Ex: (Fortran)

```
A ** B ** C         // right to left

(A ** B) ** C       // In Ada it must be parenthesized
```

Language	Associativity Rule
FORTRAN	Left: * / + -
	Right: **
C-BASED LANGUAGES	Left: * / % binary + binary -
	Right: ++ -- unary – unary +
ADA	Left: all except **
	Non-associative: **

- **APL** is different; **all** operators have **equal** precedence and all operators associate **right to left**.
- Ex:

```
A X B + C           // A = 3, B = 4, C = 5 ➔ 27
```

- Precedence and associativity rules can be overridden with parentheses.

3. **Parentheses**
    - Programmers can alter the precedence and associativity rules by placing parentheses in expressions.
    - A parenthesized part of an expression has precedence over its adjacent un-parenthesized parts.
    - Ex:

```
(A + B) * C
```

4. **Conditional Expressions**
    - Sometimes **if-then-else** statements are used to perform a conditional expression assignment.

```
if (count == 0)
    average = 0;
else
    average = sum / count;
```

    - In the C-based languages, this can be specified more conveniently in an assignment statement using a conditional expressions. Note that ? is used in conditional expression as a ternary operator (3 operands).

expression_1 ? expression_2 : expression_3

    - Ex:

average = (count == 0) ? 0 : sum / count;

Operand evaluation order

- The process:
  1. Variables: just fetch the value from memory.
  2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction.
  3. Parenthesized expressions: evaluate all operands and operators first.

- **Side Effects**
    - A **side effect** of a function, called a **functional side effect**, occurs when the function changes either one of its parameters or a global variable.
    - Ex:

```
a + fun(a)
```

- If fun does not have the side effect of changing a, then the order of evaluation of the two operands, `a` and `fun(a)`, has no effect on the value of the expression.
- However, if `fun` changes `a`, there is an effect.

- Ex:

  Consider the following situation: fun returns the value of its argument divided by 2 and changes its parameter to have the value 20, and:

```
a = 10;
b = a + fun(a);
```

- If the value of a is returned first (in the expression evaluation process), its value is 10 and the value of the expression is 15.
- But if the second is evaluated first, then the value of the first operand is 20 and the value of the expression is 25.
- The following shows a **C** program which illustrate the same problem.

```
int a = 5;
int fun1() {
   a = 17;
   return 3;
}
void fun2() {
   a = a + fun1();          // C language a = 20; Java a = 8
}
void main() {
   fun2();
}
```

- The value computed for a in `fun2` depends on the order of evaluation of the operands in the expression a + `fun1()`. The value of a will be either 8 or 20.
- Two possible solutions:
  1. Write the language definition to disallow functional side effects
     - No two-way parameters in functions.
     - No non-local references in functions.
     - **Advantage**: it works!
     - **Disadvantage**: Programmers want the flexibility of two-way parameters (what about C?) and non-local references.
  2. Write the language definition to demand that operand evaluation order be fixed
     - **Disadvantage**: limits some compiler optimizations

Java guarantees that operands are evaluated in **left-to-right order**, eliminating this problem. **// C language a = 20; Java a = 8**

Overloaded Operators

- The use of an operator for **more than one purpose** is operator overloading.
- Some are common (e.g., + for int and float).
- Java uses + for addition and for **string catenation**.
- Some are potential trouble (e.g., **&** in C and C++)

```
x = &y  // as binary operator bitwise logical
        // AND, as unary it is the address of y
```

  - Causes the address of y to be placed in x.
  - Some loss of readability to use the same symbol for two completely unrelated operations.
  - The simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler “difficult to diagnose”.
  - Can be avoided by introduction of new symbols (e.g., Pascal’s **div for integer division and / for floating point division**)

Type Conversions

- A **narrowing conversion** is one that converts an object to a type that cannot include all of the values of the original type e.g., **double to float**.
- A **widening conversion** is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., **int** to **float**.

Coercion in Expressions

- A **mixed-mode expression** is one that has operands of different types.
- A **coercion** is an implicit type conversion.
- The disadvantage of coercions:
    - They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- Language are not in agreement on the issue of coercions in arithmetic expressions.
- Those against a broad range of coercions are concerned with the reliability problems that can result from such coercions, because they eliminate the benefits of type checking.
- Those who would rather include a wide range of coercions are more concerned with the loss in flexibility that results from restrictions.
- The issue is whether programmers should be concerned with this category of errors or whether the compiler should detect them.
- Java method Ex:

```
void mymethod() {
   int a, b, c;
   float d;
   …
   a = b * d;
   …
}
```

- Assume that the second operand was supposed to be c instead of d.
- Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. Simply, b will be coerced to **`float`**.

Explicit Type Conversions

- Often called casts in C-based languages.
- Ex: Ada:

  FLOAT(INDEX)--INDEX is INTEGER type

  Java:

  (int)speed /*speed is float type*/

Errors in Expressions

- Caused by:
  - Inherent limitations of arithmetic e.g. division by zero
  - Limitations of computer arithmetic e.g. overflow or underflow
- Floating-point overflow and underflow, and division by zero are examples of **run-time errors**, which are sometimes called exceptions.

Relational and Boolean Expressions

- A relational operator: an operator that compares the values of its tow operands.
- Relational Expressions: two operands and one relational operator.
- The value of a relational expression is **Boolean**, unless it is not a type included in the language.
  - Use relational operators and operands of various types.
  - Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #)
- The syntax of the relational operators available in some common languages is as follows:

Operation	***Ada***	***C-Based Languages***	***Fortran 95***
Equal	=	==	.EQ. or ==
Not Equal	/=	!=	.NE. or <>
Greater than	>	>	.GT. or >
Less than	<	<	.LT. or <
Greater than or equal	>=	>=	.GE. or >=
Less than or equal	<=	<=	.LE. or >=

Boolean Expressions

- Operands are Boolean and the result is **Boolean**.

FORTRAN 77	**FORTRAN 90**	**C**	**Ada**
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not

- Versions of **C** prior to C99 have no Boolean type; it uses int type with **0 for false and nonzero for true**.
- One odd characteristic of C's expressions:
  a < b < c is a legal expression, but the result is not what you might expect.
- The left most operator is evaluated first because the relational operators of C, are left associative, producing **either 0 or 1**.
- Then this result is compared with var c. There is **never** a comparison between b and c.

Short Circuit Evaluation

- A **short-circuit evaluation** of an expression is one in which the result is determined **without** evaluating all of the operands and/or operators.
- Ex:

```
(13 * a) * (b/13 – 1) // is independent of the value
                      (b/13 – 1) if a = 0, because 0*x = 0.
```

- So when a = 0, there is no need to evaluate `(b/13 – 1)` or perform the second multiplication.
- However, this shortcut is not easily detected during execution, so it is never taken.
- The value of the Boolean expression:

```
(a >= 0) && (b < 10) //  is independent of the second
                         expression if a < 0, because(F && x)
                         is False for all the values of x.
```

- So when a < 0, there is no need to evaluate b, the constant 10, the second relational expression, or the && operation.
- Unlike the case of arithmetic expressions, this shortcut can be easily discovered during execution.
- Short-circuit evaluation exposes the potential problem of side effects in expressions

```
(a > b) || (b++ / 3) //  b is changed only when a <= b.
```

- If the programmer assumed b would change every time this expression is evaluated during execution, the program will fail.
- C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (**&&** and **||**), but also provide bitwise Boolean operators that are not short circuit (**&** and **|**)

Assignment Statements

Simple Assignments

- The C-based languages use == as the equality relational operator to avoid confusion with their assignment operator.
- The operator symbol for assignment:
  1. = FORTRAN, BASIC, PL/I, C, C++, Java
  2. := ALGOL, Pascal, Ada

Conditional Targets

- Ex:

```
flag ? count 1 : count2 = 0; ⇔     if (flag)
                                      count1 = 0;
                                    else
                                      count2 = 0;
```

Compound Assignment Operators

- A compound assignment operator is a shorthand method of specifying a commonly needed form of assignment.
- The form of assignment that can be abbreviated with this technique has the destination var also appearing as the first operand in the expression on the right side, as in

```
a = a + b
```

- The syntax of assignment operators that is the catenation of the desired binary operator to the = operator.

```
sum += value; ⇔    sum = sum + value;
```

Unary Assignment Operators

- C-based languages include two special unary operators that are actually abbreviated assignments.
- They combine increment and decrement operations with assignments.
- The operators ++ and -- can be used either in expression or to form stand-alone single-operator assignment statements. They can appear as prefix operators:

```
sum = ++ count;    ⇔    count = count + 1; sum = count;
```

- If the same operator is used as a postfix operator:

```
sum = count ++;    ⇔    sum = count; count = count + 1;
```

Assignment as an Expression

- This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand.
- Ex:

```
while ((ch = getchar())!=EOF)
   {…}                         // why ( ) around assignment?
```

- The assignment statement must be parenthesized because the precedence of the assignment operator is lower than that of the relational operators.
- Disadvantage: Another kind of expression side effect which leads to expressions that are difficult to read and understand. For example

   a = b + (c = d / b++) – 1

   denotes the instructions
   Assign b to temp
   Assign b + 1 to b
   Assign d / temp to c
   Assign b + c to temp
   Assign temp – 1 to a

- There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors.

```
if (x = y) …
```

instead of

```
if (x == y) …
```

Mixed-Mode Assignment

- In FORTRAN, C, and C++, any numeric value can be assigned to any numeric scalar variable; whatever conversion is necessary is done.
- In Pascal, integers can be assigned to reals, but reals cannot be assigned to integers (the programmer must specify whether the conversion from real to integer is truncated or rounded.)
- In **Java**, only **widening** assignment coercions are done.
- In **Ada**, there is no assignment coercion.
- In all languages that allow mixed-mode assignment, the coercion takes place **only after** the right side expression has been evaluated. For example, consider the following code:

  ```
  int a, b;
  float c;
  …
  c = a / b;
  ```

  Because c is float, the values of a and b could be coerced to float before the division, which could produce a different value for c than if the coercion were delayed (for example, if a were 2 and b were 3).

Chapter 6

Data Type

Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference Types

Chapter 6

Data Type

Introduction

- A data type defines a collection of **data objects** and a set of **predefined operations** on those objects.
- Computer programs produce results by manipulating data.
- ALGOL 68 provided a few basic types and a few flexible structure-defining operators that allow a programmer to design a data structure for each need.
- A **descriptor** is the collection of the attributes of a variable.
- In an implementation a descriptor is a collection of memory cells that store variable attributes.
- If the attributes are static, descriptor are required only at compile time.
- They are built by the compiler, usually as a part of the symbol table, and are used during compilation.
- For dynamic attributes, part or all of the descriptor must be maintained during execution.
- Descriptors are used for type checking and by allocation and deallocation operations.

Primitive Data Types

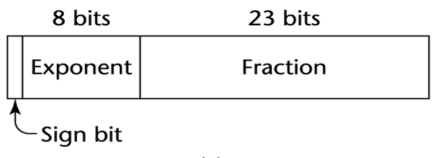
- Those not defined in terms of other data types are called primitive data types.
- The primitive data types of a language, along with one or more type constructors provide structured types.

Numeric Types

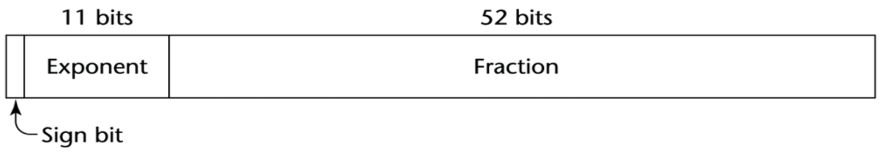
1. **Integer**
   - Almost always an exact reflection of the hardware, so the mapping is trivial.
   - There may be as many as eight different integer types in a language.
   - Java has four: **byte**, **short**, **int**, and **long**.
   - Integer types are supported by the hardware.

2. **Floating-point**
   - Model real numbers, but only as **approximations** for most real values.
   - On most computers, floating-point numbers are stored in binary, which exacerbates the problem.
   - Another problem is the loss of accuracy through arithmetic operations.
   - Languages for scientific use support at least two floating-point types; sometimes more (e.g. **float**, and **double**.)

- The collection of values that can be represented by a floating-point type is defined in terms of precision and range.
- **Precision**: is the accuracy of the fractional part of a value, measured as the number of bits. Figure below shows single and double precision.
- **Range**: is the range of fractions and exponents.



(a)



(b)

3. **Decimal**

- Most larger computers that are designed to support business applications have hardware support for **decimal** data types.
- Decimal types store a fixed number of decimal digits, with the decimal point at a fixed position in the value.
- These are the primary data types for business data processing and are therefore essential to **COBOL**.
- **Advantage**: accuracy of decimal values.
- **Disadvantages**: limited range since no exponents are allowed, and its representation wastes memory.

Boolean Types

- Introduced by ALGOL 60.
- They are used to represent switched and flags in programs.
- The use of Booleans enhances readability.
- One popular exception is C89, in which **numeric** expressions are used as conditionals. In such expressions, all operands with **nonzero** values are considered **true**, and **zero** is considered **false**.

Character Types

- Char types are stored as numeric codings (ASCII / Unicode).
- Traditionally, the most commonly used coding was the **8-bit** code ASCII (American Standard Code for Information Interchange).
- A **16-bit** character set named Unicode has been developed as an alternative.
- **Java** was the first widely used language to use the Unicode character set. Since then, it has found its way into **JavaScript and C#.**

Character String Types

- A character string type is one in which values are sequences of characters.
- **Important Design Issues**:
  1. Is it a primitive type or just a special kind of array?
  2. Is the length of objects static or dynamic?
- C and C++ use **char arrays** to store char strings and provide a collection of string operations through a standard library whose header is `string.h`.
- How is the length of the char string decided?
- The null char which is represented with 0.
- Ex:

```
char *str = “apples”; // char ptr points at the str apples0
```

- In this example, str is a char pointer set to point at the string of characters, apples0, where 0 is the null char.

String Typical Operations:

- Assignment
- Comparison (=, >, etc.)
- Catenation
- Substring reference
- Pattern matching
- Some of the most commonly used library functions for character strings in C and C++ are
  - strcpy: copy strings
  - strcat: catenates on given string onto another
  - strcmp: lexicographically compares (the order of their codes) two strings
  - strlen: returns the number of characters, not counting the null
- In Java, strings are supported as a **primitive** type by **String** class

String Length Options

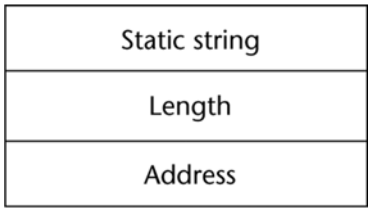
- **Static Length String**: The length can be static and set when the string is created. This is the choice for the **immutable** objects of **Java's String class** as well as similar classes in the C++ standard class library and the .NET class library available to C#.
- **Limited Dynamic Length Strings**: allow strings to have varying length up to a **declared and fixed maximum** set by the variable's definition, as exemplified by the strings in **C.**
- **Dynamic Length Strings**: Allows strings various length with no maximum. Requires the overhead of dynamic storage allocation and deallocation but provides flexibility. Ex: **Perl and JavaScript**.

Evaluation

- Aid to writability.
- As a primitive type with static length, they are inexpensive to provide--why not have them?
- Dynamic length is nice, but is it worth the expense?

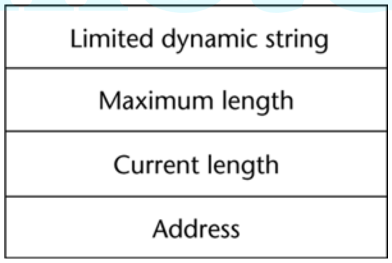
Implementation of Character String Types

- **Static length** - compile-time descriptor has three fields:
    1. Name of the type
    2. Type's length
    3. Address of first char



Compiler-time descriptor for static strings

- **Limited dynamic length Strings** - may need a run-time descriptor for length to store both the fixed maximum length and the current length (but not in C and C++ because the end of a string is marked with the **null** character).



Run-time descriptor for limited dynamic strings

- **Dynamic length Strings**–
    - Need run-time descriptor because **only current** length needs to be stored.
    - Allocation/deallocation is the biggest implementation problem. Storage to which it is bound must grow and shrink dynamically.
    - There are **two** approaches to supporting allocation and deallocation:
    1. Strings can be stored in a **linked list** "Complexity of string operations, pointer chasing"
    2. Store strings in adjacent cells. "What about when a string grows?" Find **a new area** of memory and the old part is moved to this area. Allocation and deallocation is **slower** but using adjacent cells results in faster string operations and requires less storage.

User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of **primitive** ordinal types in Java
  - integer
  - char
  - boolean
- In some languages, users can define two kinds of ordinal types: **enumeration** and **subrange**.

Enumeration Types

- All possible values, which are named constants, are provided in the definition
- C++ example

```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
myColor++;                              // would assign green to myColor
```

  - The enumeration constants are typically implicitly assigned the integer values, 0, 1, …, but can be explicitly assigned any integer literal.
- Java does not include an enumeration type, presumably because they can be represented as data classes. For example,

```
class colors {
    public final int red = 0;
    public final int blue = 1;
}
```

Subrange Types

- An ordered contiguous subsequence of an ordinal type
- Example: 12..18 is a subrange of integer type
- Ada's design

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2 := Day1;
```

Array Types

- An array is an aggregate of **homogeneous** data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
- A reference to an array element in a program often includes one or more non-constant subscripts.
- Such references require a run-time calculation to determine the memory location being referenced.

Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- What is the maximum number of subscripts?
- Can array objects be initialized?

Arrays and Indexes

- Indexing is a mapping from indices to elements.
- The mapping can be shown as:

  map(array_name, index_value_list) $\rightarrow$ an element

- Ex: Ada

  ```
  Sum := Sum + B(I);
  ```

  Because **( )** are used for both subprogram parameters and array subscripts in **Ada**, this results in **reduced readability**.

- C-based languages use [ ] to delimit array indices.
- Two distinct types are involved in an array type:
  - The element type, and
  - The type of the subscripts.
- The type of the subscript is often a sub-range of integers.
- Ada allows other types as subscripts, such as Boolean, char, and enumeration.
- Among contemporary languages, C, C++, Perl, and Fortran **don't** specify range checking of subscripts, but **Java, and C# do**.

Subscript Bindings and Array Categories

- The binding of subscript type to an array variable is usually **static**, but the subscript value ranges are sometimes **dynamically bound**.
- In C-based languages, the lower bound of all index ranges is fixed at **0**; **Fortran** 95, it defaults to **1**.

1. A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time).
   - Advantages: efficiency "No allocation & deallocation."
   - Ex:

     Arrays declared in C & C++ function that includes the **`static`** modifier are **static**.

2. A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at elaboration time during execution.
   - Advantages: Space efficiency. A large array in one subprogram can use the same space as a large array in different subprograms.
   - Ex:

     Arrays declared in C & C++ function without the **`static`** modifier are **fixed stack-dynamic arrays**.

3. A **stack-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic "during execution." Once bound they remain fixed during the lifetime of the variable.
   - Advantages: Flexibility. The size of the array is not known until the array is about to be used.
   - Ex:

     Ada arrays can be **stack dynamic**:

```
Get (List_Len);
declare
    List : array (1..List_Len) of Integer;
    begin
  . . .
    end;
```

The user inputs the number of desired elements for array **`List`**. The elements are then dynamically allocated when execution reaches the **`declare`** block. When execution reaches the end of the block, the array is deallocated.

4. A **fixed heap-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, but they are both fixed after storage is allocated.
   - The bindings are done when the user program requests them, rather than at elaboration time and the storage is allocated on the heap, rather than the stack.
   - Ex:

     C & C++ also provide **fixed heap-dynamic arrays**. The function *`malloc`* and `free` are used in C. The operations `new` and `delete` are used in C++.

     In Java all arrays are **fixed heap dynamic arrays**. Once created, they keep the same subscript ranges and storage.

5. A **heap-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, and can change any number of times during the array's lifetime.
   - Advantages: Flexibility. Arrays can grow and shrink during program execution as the need for space changes.
   - Ex:

     C# provides **heap-dynamic arrays** using an array class *`ArrayList`.*

     ```
     ArrayList intList = new ArrayList( );
     ```

     Elements are added to this object with the *Add* method, as in
     `intArray.Add(nextOne);`

     Perl and JavaScript also support heap-dynamic arrays.
     For example, in Perl we could create an array of five numbers with

     ```
     @list = {1, 2, 4, 7, 10);
     ```

     Later, the array could be lengthened with the push function, as in

     ```
     push(@list, 13, 17);
     ```

     Now the aary's value is (1, 2, 4, 7, 10, 13, 17).

Array Initialization

- Usually just a list of values that are put in the array in the order in which the array elements are stored in memory.
- Fortran uses the **DATA** statement, or put the values in **/ ... /** on the declaration.

```
Integer List (3)
Data List /0, 5, 5/  // List is initialized to the values
```

- C and C++ - put the values in braces; let the compiler count them.

```
int stuff [] = {2, 4, 6, 8};
```

- The compiler sets the length of the array.
- What if the programmer mistakenly left a value out of the list?
- Character Strings in C & C++ are implemented as arrays of **`char`**.

```
char name [ ] = “Freddie”; //how many elements in array name?
```

- The array will have 8 elements because the null character is implicitly included by the compiler.
- In Java, the syntax to define and initialize an array of references to String objects.

```
String [ ] names = [“Bob”, “Jake”, “Debbie”];
```

- Ada positions for the values can be specified:

```
List : array (1..5) of Integer := (1, 3, 5, 7, 9);
Bunch : array (1..5) of Integer:= (1 => 3, 3 => 4, others => 0);
     Note: the array value is (3, 0, 4, 0, 0)
```

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:

  address(list[k]) = address (list[lower_bound]) + ((k-lower_bound) * element_size)

- Accessing Multi-dimensioned Arrays
- Two common ways:
  - Row major order (by rows) – used in most languages
  - column major order (by columns) – used in Fortran
- For example, if the matrix had the values

  3 4 7
  6 2 5
  1 3 8

  - it would be stored in row major order as:

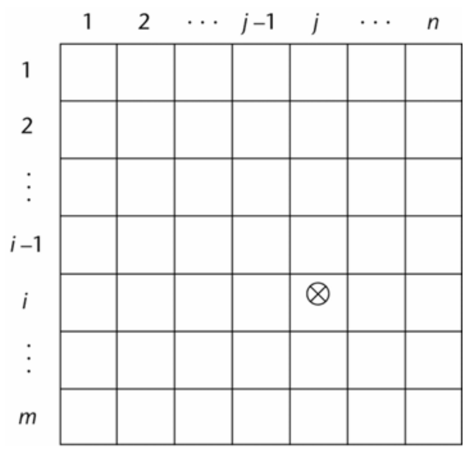
    3, 4, 7, 6, 2, 5, 1, 3, 8

  - If the example matrix above were stored in column major, it would have the following order in memory.

    3, 6, 1, 4, 2, 3, 7, 5, 8

  - In all cases, sequential access to matrix elements will be faster if they are accessed in the order in which they are stored, because that will minimize the **paging**. (Paging is the movement of blocks of information between disk and main memory. The objective of paging is to keep the frequently needed parts of the program in memory and the rest on disk.)
- Locating an Element in a Multi-dimensioned Array (row major)

  Location (a[i,j]) = address of a [1,1] + ((i - 1) * n + (j - 1)) * element_size



Associative Arrays

- An associative array is an unordered collection of data elements that are indexed by an equal number of values called **keys**.
- So each element of an associative array is in fact a pair of entities, a key and a value.
- Associative arrays are supported by the standard class libraries of Java and C++ and Perl.
- Example: In Perl, associative arrays are often called **hashes**. Names begin with %; literals are delimited by parentheses

```
%temps = ("Mon" => 77, "Tue" => 79, “Wed” => 65);
```

  - Subscripting is done using braces and keys

```
$temps{"Wed"} = 83;
```

  - Elements can be removed with delete

```
delete $temps{"Tue"};
```

  - Elements can be emptied by assigning the empty literal

```
@temps = ( );
```

Record Types

- A record is a possibly **heterogeneous** aggregate of data elements in which the individual elements are identified by names.
- In C, C++, and C#, records are supported with the **struct** data type. In C++, structures are a minor variation on classes.
- COBOL uses level numbers to show nested records; others use recursive definition
- Definition of Records in COBOL
    - COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.
  02 EMP-NAME.
    05 FIRST PIC X(20).
    05 MID   PIC X(10).
    05 LAST  PIC X(20).
  02 HOURLY-RATE PIC 99V99.
```

- Definition of Records in Ada
    - Record structures are indicated in an orthogonal way

```
type Emp_Rec_Type is record
     First: String (1..20);
     Mid: String (1..10);
     Last: String (1..20);
     Hourly_Rate: Float;
 end record;
 Emp_Rec: Emp_Rec_Type;
```

References to Records

- Most language use dot notation

```
Emp_Rec.Name
```

- Fully qualified references must include all record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL

FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

Operations on Records

- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides MOVE CORRESPONDING
    - Copies a field of the source record to the corresponding field in the target record

Unions Types

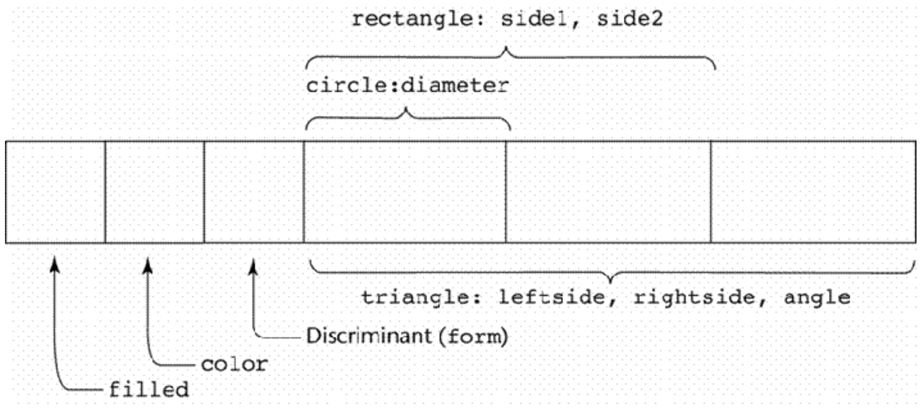
- A union is a type whose variables are allowed to store different type values at different times during execution.

Discriminated vs. Free Unions

- **Fortran, C, and C++** provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*
- Type checking of unions require that each union include a type indicator called a *discriminanted union.*
- Supported by **Ada**

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
```



Ada Union Type Illustrated: A discriminated union of three shape variables

Evaluation of Unions

- Potentially **unsafe** construct
- Java and C# **do not** support unions
- Reflective of growing concerns for safety in programming language

Pointers

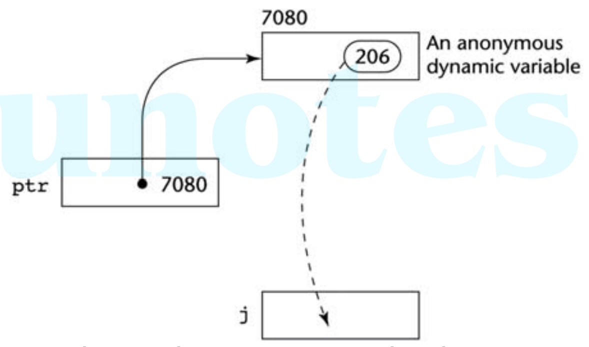
- A pointer type in which the vars have a range of values that consists of **memory addresses** and a special value, nil.
- The value nil is not a valid address and is used to indicate that a pointer cannot currently be used to reference any memory cell.

Pointer Operations

- A pointer type usually includes two fundamental pointer operations, assignment and dereferencing.
- Assignment sets a pointer var's value to some useful address.
- Dereferencing takes a reference through one level of indirection.
- In C++, **dereferencing** is explicitly specified with the (*) as a prefix unary operation.
- If *`ptr`* is a pointer var with the value 7080, and the cell whose address is 7080 has the value 206, then the assignment

```
j = *ptr
```

  sets j to 206.



The assignment operation j = *ptr

Pointer Problems

1. **Dangling pointers** (**dangerous**)
   - A pointer points to a heap-dynamic variable that has been **deallocated**.
   - Dangling pointers are dangerous for the following reasons:
     - The location being pointed to may have been allocated to some new heap-dynamic var. If the new var is not the same type as the old one, type checks of uses of the dangling pointer are invalid.
     - Even if the new one is the same type, its new value will bear no relationship to the old pointer's dereferenced value.
     - If the dangling pointer is used to change the heap-dynamic variable, the value of the heap-dynamic variable will be destroyed.
     - It is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail.
   - The following sequence of operations creates a dangling pointer in many languages:

     a. Pointer p1 is set to point at a new heap-dynamic variable.
     b. Set a second pointer p2 to the value of the first pointer p1.
     c. The heap-dynamic variable pointed to by p1 is explicitly deallocated (setting p1 to nil), but p2 is not changed by the operation. P2 is now a dangling pointer.

2. **Lost Heap-Dynamic Variables** (wasteful)

   - A heap-dynamic variable that is no longer referenced by any program pointer "no longer accessible by the user program."
   - Such variables are often called **garbage** because they are not useful for their original purpose, and also they can't be reallocated for some new use by the program.
   - Creating Lost Heap-Dynamic Variables:

     a. Pointer p1 is set to point to a newly created heap-dynamic variable.
     b. p1 is later set to point to another newly created heap-dynamic variable.
     c. The first heap-dynamic variable is now inaccessible, or lost.

   - The process of losing heap-dynamic variables is called **memory leakage**.

Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can be **automatically** de-allocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada

Pointers in Fortran 95

- Pointers point to heap and non-heap variables
- Implicit dereferencing
- Pointers can only point to variables that have the TARGET attribute
- The TARGET attribute is assigned in the declaration:

```
INTEGER, TARGET :: NODE
```

Pointers in C and C++

- **Extremely flexible** but must be used with care.
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible in C and C++ makes their pointers **more interesting** than those of the other programming languages.
- Unlike the pointers of Ada, which can only point into the heap, C and C++ pointers can point at virtually any variable **anywhere** in memory.
- Explicit dereferencing and address-of operators
- In C and C++, the asterisk (*) denotes the **dereferencing** operation, and the ampersand (&) denotes the operator for producing the **address of** a variable. For example, in the code

```
int *ptr;
int count, init;
…
ptr = &init;
count = *ptr
```

- the two assignment statement are equivalent to the single assignement

```
count = init;
```

- Example: Pointer Arithmetic in C and C++

```
int list[10];
int *ptr;
ptr = list;

*(ptr+5)   is equivalent to list[5]   and  ptr[5]
*(ptr+i)   is equivalent to list[i]   and  ptr[i]
```

- Domain type need not be fixed (**void ***)
- void * can point to **any** type and can be type checked (cannot be de-referenced)

Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters in function definition.
- A C++ reference type variable is a **constant** pointer that is always **implicitly** dereferenced.
- Because a C++ reference type variable is a constant, it **must** be **initialized** with the address of some variable in its definition, and after initialization a reference type variable can **never** be set to reference any other variable.
- Reference type variables are specified in definitions by preceding their names with ampersands (&). for example,

```
int result = 0;
int &ref_result = result;
…
ref_result = 100;
```

  I this code segment, result and ref_result are **aliases**.

- In **Java**, **reference variables** are extended from their C++ form to one that allow them to replace pointers entirely.
- The fundamental difference between C++ pointers and Java references is that C++ pointers refer to **memory addresses**, whereas Java references refer to **class instances**.
- Because Java class instances are implicitly deallocated (there is no explicit deallocation operator), there **cannot** be a dangling reference.
- C# includes both the references of Java and the pointers of C++. However, the use of pointers is strongly discouraged. In fact, any method that uses pointers must include the **unsafe** modifier.
- Pointers can point at any variable regardless of when it was allocated