Names, Scopes, and Bindings:-

- Name: **name** is a string of characters used to identify some entity
  - Allow us to refer to variables, constants, functions, types, operations etc
- Binding: An association of a name with an object
- Scope: The part of the program in which the binding is active

Variables

- A variable in an imperative language, or an object-oriented language, is a six-tuple:

  <name, address, value, type, lifetime, scope>

- A **name** is a string of characters used to identify some entity.
- Declaration of type, usage with a value, lifetime, scope of names are a major consideration in programming languages

- Names are identifiers(alpha numeric tokens)
- Names refer to abstraction
- programmer associates name with complicated program fragment/elem
  - Control abstraction
    - Allows the programmer to hide sequence of complicated code in a name.
  - Data Abstraction
    - Allows the programmer to hide data representation behind a set of operations.

- The address of a variable is sometimes called its ***l*-value** because that is what is required when a variable appears in the **left** side of an assignment statement

- **Type**

- Determines the **range of values of variables and the set of operations** that are defined for values of that type
- For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations (+,-,*,/,%)

- **Value**

- The value of a variable is the **contents of the memory cell** or cells associated with the variable.
- A variable's value is sometimes called its r-value because that is what is required when a variable appears in the right side of an assignment statement.

Binding , Binding time,& Referencing Environment

- Binding is association of 2 things-an attribute with an entity.
- **Binding time** is the time at which a binding takes place.
- Referencing Environment-complete set of bindings at a given point in a program.

- ***Binding***
  - the operation of associating two things, like a name and the entity it represents.

- The compiler performs a process called binding when an object is assigned to an object variable.

Binding time

- Binding time is the moment when the binding is performed (compilation, execution, etc).
- The early binding (static binding) refers to compile time binding
- late binding (dynamic binding) refers to runtime binding.

The Concept of Binding

- The ***l*-value** of a variable is its **address**.
  The ***r*-value** of a variable is its **value**.
- A **binding** is an association, such as between an attribute and ar entity,or between an operation and a symbol.
- A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.

Possible Binding Time

- Binding Time is the time at which a binding is created

1. Language design time
2. Language implementation time
3. program writing time
4. compile time
5. link time
6. load time
7. *Runtime*

1. **Language design time** (bind operator symbols to operations. * to mul)
   - program structure, possible types , control flow constructs are chosen
2. **Language implementation time**
   - Coupling of I/O to OS, arithmetic overflow, stack size, type equality ,handling of run time exceptions
   - Ex)A data type such as **int** in C is bound to a **range** of possible values
3. **program writing time**
   - Programmers choose algorithms, data structures and names
4. **compile time**
   - bind a variable to a **particular data type** at compile time

5. link time
   - Library of standard subroutines joined together by a linker.
6. load time
   - Refers to the point at which the **OS loads the program into memory** so that it can run. virtual address are chosen at link time and physical addresses change at run time.
   - bind a variable to a **memory cell** (ex. C **static** variables)
7. Runtime
   - refers to the entire span from the beginning to the end of execution..virtual functions, values to variables, many more.
   - bind a **nonstatic** local variable to a memory cell

Binding Time Examples

Language feature	*Binding time*
Syntax, e.g. `if (a>0) b:=a;` in C or `if a>0 then b:=a end if` in Ada	Language design
Keywords, e.g. `class` in C++ and Java	Language design
Reserved words, e.g. `main` in C and `writeln` in Pascal	Language design
Meaning of operators, e.g. + (add)	Language design
Primitive types, e.g. `float` and `struct` in C	Language design
Internal representation of literals, e.g. `3.1` and `"foo bar"`	Language implementation
The specific type of a variable in a C or Pascal declaration	Compile time
Storage allocation method for a variable	Language design, language implementation, and/or compile time
Linking calls to static library routines, e.g. `printf` in C	Linker
Merging multiple object codes into one executable	Linker
Loading executable in memory and adjusting absolute addresses	Loader (OS)
Nonstatic allocation of space for variable	Run time

Example of static and dynamic binding

static

```
//static Example
class Dog{
 private void eat(){ System.out.println("dog is eating...");
}
public static void main(String args[])
{
  Dog d1=new Dog();
  d1.eat();
 } }
```

```
#dynamic example
class Animal{
 void eat(){System.out.println("animal is eating...");}
}
class Dog extends Animal
{   void eat()  {System.out.println("dog is eating...");}
 public static void main(String args[])
    {  Animal a=new Dog();
     a.eat();
      } }
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type. o/p—Dog is eating

Effect of Binding Time

- **Early binding times** (before run time)are associated with **greater efficiency**
  - Syntactic and sematic checking can be done at compile time only once and run time overhead can be avoided
- **late binding times**(at run time) are associated with **greater flexibility.**
  - Interpreters allows programs to be extended at run time
  - Method binding in oops must be late to support dynamic binding.

- Binding lifetime: time between creation and destruction of binding to object
  - Example: A pointer variable is set to the address of an object
  - Example: A formal argument is bound to an actual argument
- Object lifetime: time between creation and destruction of an object.
- Key events in object lifetime
  - Object creation
  - Creation of bindings
  - References to variables, subroutines, types are made using bindings
  - Deactivation and reactivation of temporarily unusable bindings
  - Destruction of bindings
  - Destruction of objects

- Bindings are temporarily invisible when code is executed where the binding (name □□ object) is out of scope
- Memory leak: object never destroyed (binding to object may have been destroyed, rendering access impossible)
- Dangling reference: object destroyed before binding is destroyed
- Garbage collection prevents these allocation/deallocation problems

Scope Rules

- Statically scoped language: the scope of bindings is determined at compile time.
- Dynamically scoped language: the scope of bindings is determined at run time.

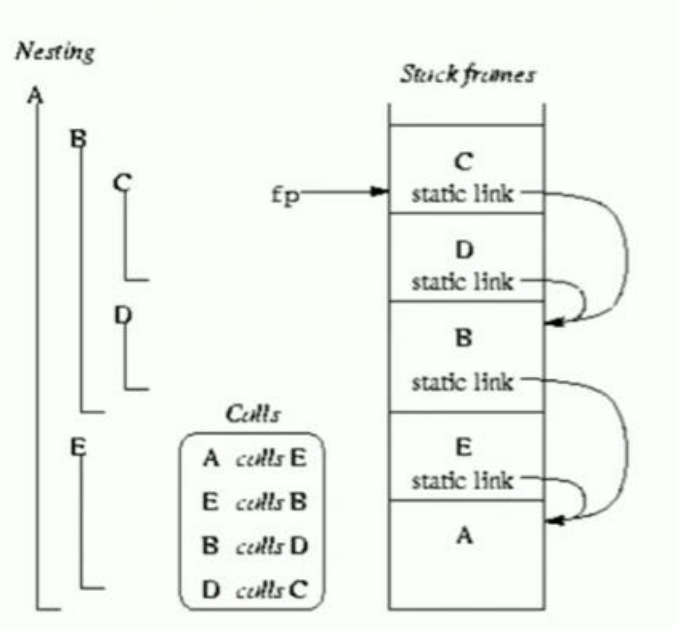
Static Scoping

- C: one declares the variable static;
  - A saved (static, own) variable has a lifetime that encompasses the entire execution of the program.
  - Instead of a logically separate object for every invocation of the subroutine, the compiler creates a single object that retains its value from one invocation of the subroutine to the next.

- Scope rules are designed so that we can only refer to variables that are alive: the variable must have been stored in the frame of a subroutine

Static Scope Implementation with static links

- The simplest way in which to find the frames of surrounding scopes is to maintain a static link in each frame that points to the “parent frame”. Each frame on the stack contains a static link pointing to the frame of the static parent.



Static Chains

- The static links forms a static chain, which is a linked list of static parent frames
- The compiler generates code to make these traversals over frames to reach non-local objects
- Subroutine A is at nesting level 1 and C at nesting level 3
- When C accesses an object of A, 2 static links are traversed to get to A's frame that contains that object

Out of Scope

```
procedure P1;
var X:real;
  procedure P2;
  var X:integer
  begin
    ... (* X of P1 is hidden *)
  end;
begin
  ...
end
```

Static Vs Dynamic Scoping

```
1.  n : integer              -- global declaration
2.  procedure first
3.      n := 1
4.  procedure second
5.      n : integer          -- local declaration
6.      first()
7.  n := 2
8.  if read_integer() > 0
9.      second()
10. else
11.     first()
12. write_integer(n)
```

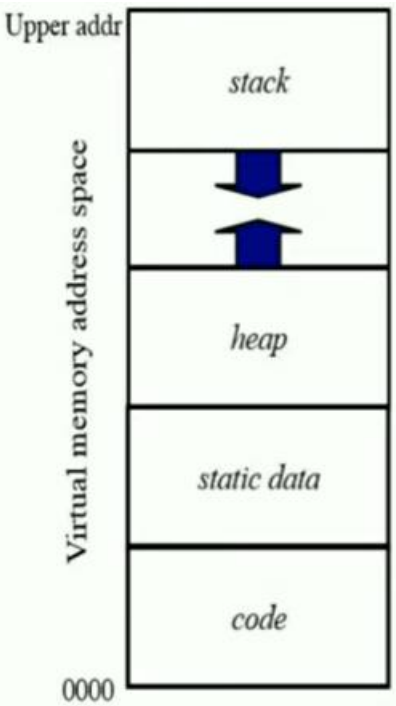
Static versus dynamic scoping.

Program output depends on both scope rules and, in the case of dynamic scoping, a value read at run time.

Consider the program in Figure 3.9. If static scoping is in effect, this program prints a 1. If dynamic scoping is in effect, the output depends on the value read at line 8 at run time: if the input is positive, the program prints a 2; otherwise it prints a 1. Why the difference? At issue is whether the assignment to the variable n at line 3 refers to the global variable declared at line 1 or to the local variable declared at line 5. Static scope rules require that the reference resolve to the closest lexically enclosing declaration, namely the global n. Procedure first changes n to 1, and line 12 prints this value. Dynamic scope rules, on the other hand, require that we choose the most recent, active binding for n at run time.

Storage management

- Objects (program data and code) have to be stored in memory during their lifetime
- Three principal storage allocation mechanisms, used to manage the object's space - static allocation, stack allocation, and heap allocation



- Program code is at the bottom of the memory region (code section)
  - The code section is protected from run-time modification by the OS
- Static data objects are stored in the static region
- Stack grows downward
- Heap grows upward

FIGURE: TYPICAL PROGRAM AND DATA LAYOUT IN MEMORY

STATIC ALLOCATION

- Static Objects have an absolute storage address that is retained throughout the execution of the program
- **Absolute addresses are also called real addresses and machine addresses**
- **A fixed address in memory**
- They are often allocated in protected, read-only memory
- Any attempt to write to them will cause a processor interrupt, allowing the operating system to announce a run-time error
- **Advantage** of statically allocated object is the **fast access**

STACK ALLOCATION

- Every (recursive) subroutine call must have separate instantiations of local variables
- **Stack Objects** are allocated in last-in first-out(LIFO) order, usually in conjunction with subroutine calls and returns
  - Each instance of a subroutine at run time has a frame on the run-time stack (also called activation record)

STACK ALLOCATION

- Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards
- Frame layouts vary between languages and implementations
- A frame pointer (fp) points to the frame of the currently active subroutine at run time (always topmost frame on stack)

STACK ALLOCATION

- Subroutine arguments, local variables, and return values are accessed by constant address offsets from fp
- The stack pointer (sp) points to free space on the stack
- Even in a language without recursion, it can be advantageous to use a stack for local variables

HEAP ALLOCATION

- heap is a region of storage in which sub-blocks can be allocated and de-allocated at arbitrary times.
- Heaps are required for the dynamically allocated pieces of linked data structures, and for objects like fully general character strings, lists, and sets, whose size may change as a result of an assignment statement or other update operation

Referencing Environment

- collection of all names that are visible in the statement
- In a static-scoped language, local variables plus all of the visible variables in all of the enclosing scopes
- In a dynamic-scoped language, local variables plus all visible variables in all active subprograms

Ex, Ada, **static-scoped language**

```
procedure Example is
   A, B : Integer;
   …
   procedure Sub1 is
      X, Y : Integer;
      begin      -- of Sub1
      …                          ← 1
      end        -- of Sub1
   procedure Sub2 is
      X : Integer;
      …
      procedure Sub3 is
         X : Integer;
         begin   -- of Sub3
         …                       ← 2
         end;    -- of Sub3
    begin  -- of Sub2
    …                            ← 3
    end;   { Sub2}
begin
   …                             ← 4
end;       {Example}
```

- The referencing environments of the indicated program points are as follows:

Point	*Referencing Environment*
1	X and Y of Sub1, A & B of Example
2	X of Sub3, (X of Sub2 is hidden), A and B of Example
3	X of Sub2, A and B of Example
4	A and B of Example

Precedence and Associativity of Operators

Precdence Group (Highest to Lowest)	Operators	Associativity
(param) subscript etc.,	( ) [ ] –>.	L → R
Unary operators	- + ! ~ ++ – – (type) * & sizeof	R → L
Multiplicative	* / %	L → R
Additive	+ –	L → R
Bitwise shift	<< >>	L → R
Relational	< <= > >=	L → R
Equality	= = !=	L → R
Bitwise AND	&	L → R
Bitwise exclusive OR	^	L → R
Bitwise OR	|	L → R
Logical AND	&&	L → R
Logical OR	||	L → R
Conditional	?:	R → L
Assignment	= += -= *= /= %= &= ^= |= <<= >>=	R → L
Comma	,	L → R

Reasons for Evaluation Ordering within Expressions

- **Side effects:** e.g. if f(b) above modifies b, the expression value will depend on the operand evaluation order
- **Code improvement**: compilers rearrange expressions to maximize efficiency
  - **Improve memory loads:**

    a:=B[i]; load a from memory

    c:=2*a+3*d; compute 3*d first, because loads are slow

- **Common sub-expression elimination:**
- a:=b+c; d:=c+e+b; rearranged as d:=b+c+e, it can be rewritten into d:=a+e
- **Register allocation**: rearranging operand evaluation can decrease the number of processor registers used for temporary values

Expression Reordering Problems

- may lead to arithmetic overflow or different floating point results
  - Assume b, d, and c are very large positive integers, then if b-c+d is rearranged into (b+d)-c arithmetic overflow occurs
  - Floating point value of b-c+d may differ from b+d-c
  - Most programming languages will not rearrange expressions when parentheses are used, e.g. write (b-c)+d to avoid problems

Short-Circuit Evaluation

- Boolean expressions provide a special and important opportunity for code improvement and increased readability
- Short-circuit evaluation of Boolean expressions means that computations are skipped when logical result of a Boolean operator can be determined from the evaluation of one operand

- Short circuiting is not necessarily as attractive for situations in which a Boolean sub expression can cause a side effect
- **Delayed or Lazy Evaluation:**Short circuiting can be considered as a delayed or lazy evaluation because the operands are passed unevaluated. Internally, the operator evaluates the first operand in any case, the second only when needed.

Unstructured flow

- use of goto statements and statement labels to obtain control flow
- Generally considered bad, but sometimes useful for jumping out of nested loops and for programming errors and exceptions
- Java has no goto statement

Structured flow

- Sequencing: the subsequent execution of a list of statements in that order
- Selection: if-then-else statements and switch or case-statements
- Iteration: for and while loop statements
- Subroutine calls and recursion

All of which promotes structured programming

Iteration

- ***Enumeration-controlled loops*** repeat a collection of statements a number of times, where in each iteration a loop index variable takes the next value of a set of values specified at the beginning of the loop
- ***Logically-controlled loops*** repeat a collection of statements until some Boolean condition changes value in the loop

i. Pre-test loops test condition at the begin of each iteration
ii. Post-test loops test condition at the end of each iteration
iii. Mid-test loops allow structured exits from within loop with exit conditions

Module -2

Data Type

Two principal purposes

1. Types provide implicit context for many operations
2. limit the set of operations that may be performed

Type Systems

- Consists of
  1. a mechanism for defining types and associating them with certain language constructs
     - constructs include named constants, variables, record fields, parameters , and sometimes subroutines ; literal constants (e.g., 17, 3.14, "foo");

Type Systems

- Consists of

2. a set of rules for type equivalence, type compatibility, and type inference
    - Type equivalence rules determine when the types of two values are the same
    - Type compatibility rules determine when a value of a given type can be used in a given context.
    - Type inference rules define the type of an expression based on the types of its constituent parts or (sometimes) the surrounding context

- Type checking - process of ensuring that a program obeys the language's type compatibility rules
- Type clash - violation of the type compatibility rules
- STRONG TYPED means that the language prevents you from applying an operation to data on which it is not appropriate
- *Strongly typed languages* always detect types errors
- STATIC TYPING - type checking performed at compile time
- DYNAMIC TYPING - checks type correctness at run-time

Type equivalence

- two standard ways to determine whether two types are considered the same: **name equivalence** and **structural equivalence**
- **Name equivalence**: ***two types are equal if, and only if, they have the same name***
- **structural equivalence:** *two types are equal if, and only if, they have the same "structure"*, which can be interpreted in different ways

- Name equivalence is based on declarations
- Structural equivalence is based on meaning behind those declarations
- Name equivalence is more fashionable these days

Name equivalence

- if *name equivalence* is used in the language then **x and y would be of the same type** and **r and s would be of the same type**,
- but the type of x or y would not be equivalent to the type of r or s.
- x = y; r = s; would be valid, but statements such as
- x = r; would not be valid
- x , y are name equivalent

Name equivalence Example

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

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

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

Structural Equivalence

- names and types of each component of the two types must be the same and must be listed in the same order in the type definition.
- the names of the components could be different
- **using structural equivalence the two types Stack and Set would be considered equivalent**, which means that a translator would accept statements such as x = r;
- C doesn't support structural equivalence and will give error for above assignment

Example

- They assigned a value of type school into a variable of type student.
- A compiler whose type checking is based on structural equivalence will accept such an assignment
- In the example x and y will be considered to have different types under name equivalence.
- X uses the type declared at line 1
- Y uses the type declared at line 4

```
1.  type student = record
2.          name, address : string
3.          age : integer
4.  type school = record
5.          name, address : string
6.          age : integer
7.  x : student;
8.  y : school;
9.  . . .
10. x := y;          -- is this an error?
```

Type Compatibility

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

Type Compatibility

- **Assignment compatibility**
  the type of the RHS must be compatible with that of the LHS

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

- For example

  float n1=12.5;

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

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

Type Compatibility

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

  int n=3/2;

  cout<<n;

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

Type Compatibility contd..

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

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

Type inference

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

Type inference contd..

- **What determines the type of the overall expression?**
  - The result of arithmetic operator has the same type as the operands
  - The result of comparison is Boolean
  - The result of a function call has the type declared in the function header
  - The result of an assignment has the same type as the LHS
- Operations on subrange & composite types do not preserve the types of the operands. Type inference found in ML.
- Composite Type
  - Nonscalar types are usually called composite, or constructed types
  - generally created by applying a type constructor to one or more simpler types
  - eg. records (structures), variant records (unions), arrays, sets, pointers, lists, and files
  - Composite literals are sometimes known as aggregates (in Ada)

Type Conversion

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

conversion example

- The C code below illustrates implicit and explicit coercion

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

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

Non*converting* Type Cast

- **A change of type that does not alter the underlying bits is called a nonconverting type cast or type pun**
- In Ada , using a built in generic subroutine called unchecked_conversion

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

- C++ inherits casting mechanisms of C

Records(structures) & Variants(unions)

- Record types allows **related data of heterogeneous types to be stored and manipulated together**

❑ **Each of the record components is known as fields**

❑ To refer to given field of a record, most language uses "dot" notation.

Records (Structures)& Variants (Unions)

- Unions (variant records)
  - certain variables allocated "on top of one another " by sharing the same bytes in memory
  - overlay space. only one member can contain value at any given time
  - allows to store different data types in the same memory location
  - Overall **size of this union** would be the that of its **largest member**

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

Records (Structures) and Variants (Unions)

2 main purposes of union

1. **unions allows the same set of bytes to be interpreted in different ways at different times**
2. To **represent alternative sets of fields within a record**

Eg)A record representing an employee,might have common fields(name,address,phone,department,ID number) and various other fields depending on whether the person works on a salaried , hourly or consulting basis.

Arrays

- Two **distinct types** are involved in an array type:
  - The **element type**, and
  - The **type of the subscripts**.

Arrays: Initialization

- a list of values put in the array in the order in which the array elements are stored in memory

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

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

✔Ada –

mat : array(1..10,1..10)of real

Array layouts: why we care

- Layout makes a big difference for access speed
- in high performance computing is simply to set up your code to go in row major order
- Two layout strategies for arrays :
  - Contiguous elements
  - Row pointers

Layout of Arrays

1.a) Row-major layout

Each row of array is in a contiguous chunk of memory

- row major - used by most of programming languages

1.b) Column-major layout

Each column of array is in a contiguous chunk of memory

- column major - only in Fortran

2. Row-pointer layout

An array of pointers to rows lying anywhere in memory

Array layouts

1.Contiguous elements

a)**column major**: consecutive memory location hold elements that differ by 1 in initial subscript

- A[2,4] is followed by A[3,4]
  - only in Fortran

b)**row major**: consecutive memory location hold elements that differ by 1 in final subscript

- so A[2,4] is followed by A[2,5] in memory
  - used by everybody else.

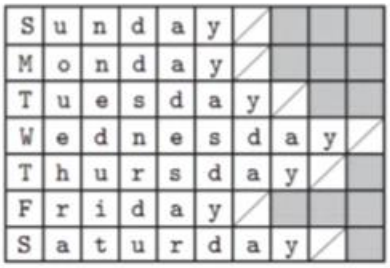
Array Layout

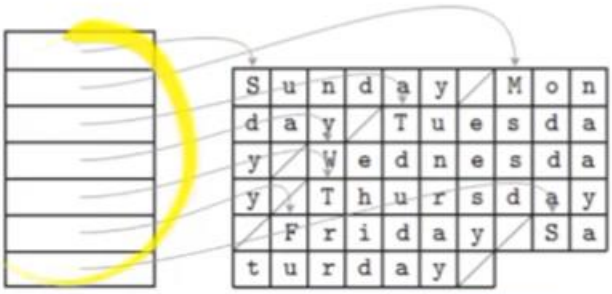
2. **Row pointers**
   - an option in C
   - allows rows to be put anywhere
   - for big arrays on machines with segmentation problems
   - nice **for matrices whose rows are of different lengths**
     - e.g. an array of strings
   - **requires extra space for the pointers**

Arrays

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

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





- A slice is some substructure of an array
- nothing more than a referencing mechanism

Array Categories

- ✔ static array
- ✔ fixed stack-dynamic array
- ✔ *Stack-dynamic*
- ✔ *Fixed heap-dynamic*
- ✔ Heap-dynamic

- *Static array*:
  - **subscript ranges are statically bound and storage allocation is static** (before run-time)
  - Advantage: efficiency (no dynamic allocation)

Ex: Arrays declared in C & C++ function that includes the static modifier are static

- *Fixed stack-dynamic*:
  - **subscript ranges are statically** bound
  - **allocation is done** during execution
  - Advantages: Space efficiency

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

Fixed stack-dynamic:

- **subscript ranges are dynamically bound**
- **storage allocation is dynamic** "during execution."
- Once bound they remain fixed during the lifetime of the variable
- Advantages: Flexibility
- **size of the array is not known until the array is about to be used**

Array Categories

- *Fixed heap-dynamic*:
  - similar to fixed stack-dynamic
  - **storage binding is dynamic but fixed after allocation**
  - The bindings are done when the user program requests them
  - 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**
  - **new and delete are used in C++**

- Heap Dynamic
  - subscript range and storage bindings are dynamic and not fixed
  - In Java, all arrays are objects (heap-dynamic)

Arrays-Dimensions, Bounds, and Allocation

- The **shape of an array** consists of the **number of dimensions and the bounds of each dimension in the array**
- The time at which the shape of an array is bound has an impact on how the array is stored in memory
  - *global lifetime, static shape*
    - shape of an array is known at compile time
    - array can exist throughout the execution of the program
    - compiler can allocate space for the array in static global memory

Arrays

- local lifetime, static shape
  - shape of the array is known at compile time
  - array should not exist throughout the execution of the program
  - space can be allocated in the subroutine's stack frame at run time
- *local lifetime, shape bound at run/elaboration time* - variable-size part of local stack frame
- *arbitrary lifetime, dynamic shape (bound at runtime)* - allocate from heap or reference to existing array
- *arbitrary lifetime, dynamic shape* - also known as dynamic arrays, must allocate (and potentially reallocate) in heap

Dope vector

- **A dope vector contains the dimension, bounds, and size information for an array**
- Dynamic arrays require that the dope vector be held in memory during run-time Contiguous elements
- DOPE vector will contain the lower bound of each dimension and the size of each dimension .

- The most common **array operations - assignment, concatenation, comparison for equality and inequality, and slices**
- The **C-based languages do not provide any array operations,**

Arrays: Address calculations

- **Example**: Suppose we have a 3d array in Ada:

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

S3 = size of elem-type
S2 = (U3-L3+1)* S3 (* size of a row *)
S1 = (U2-L2+1) * S2 (* size of a plane *)

- Then calculating A[i , j , k] each time is:

(Address of A) + (i-L1)*S1 + (j-L2)*S2 + (k-L3)*S3

Arrays: Address calculations

- Given an array [1..8, 1..5, 1..7] of integers. Calculate address of element A[5,3,6], by using rows and columns methods, if BA=900?

- Solution:-

A[i ,j,k]=Address of A+(i-L1)*S1+(j-L2)*S2 +(k-L3)*S3

Let *i=5, j=3, k=6,L1=L2=L3=1, U1=8,U2=5,U3=7*

S3 = 4 (size of element type)
*S2=(U3-L3+1)*S3=(7-1+1)*4=28*

*S1=(U2-L2+1)*S2=(5-1+1)*28=140*

Location(A[5,3,6])= 900 +(5-1)*S1+(3-1)*S2+(6-1)*S3

=900+4*140+2*28+5*4 =1536

Arrays: Address calculations

- Given an array [1..8, 1..5, 1..7] of integers. Calculate address of element A[5,3,6] , by using rows and columns methods, if BA=900?
  Solution:- The dimensions of A are :
  D1= U1-L1+1=8-1+1=8 , D2= U2-L2+1=5-1+1=5, D3 = U3-L3+1=7-1+1=7,

  i=5, j=3, k=6

Rows - wise :

Location (A[i,j,k]) = BA + D1D2(k-1) + D2(i-1) + (j-1)

Location(A[5,3,6])= 900 + 8x5(6-1) + 5(5-1) + (3-1)
= 900 + 40 x 5 +5 x 4 + 2
= 900 + 200 +20 +2
= 1122

Columns - wise :

Location (A[i,j,k]) = BA + D1D2(k-1) + D1(j-1) + (i-1)

Location (A[5,3,6]) = 900 + 8x5(6-1) + 8(3-1) + (5-1)
= 900 + 40 x 5 +8 x 2 + 4
= 900 + 200 +16 +4
= 1120

- Sets are unordered, so you cannot be sure in which order the items will appear
- cannot access items in a set by referring to an index the items has no index
- Sets are unordered collections that can't have duplicate elements
- Using set() is great for creating a unique collection of items from existing data
- remove an element from a set - remove() method

Pointers And Recursive Types

- to **create a new pointer value, call a built-in function that allocates a new object in the heap** and returns a pointer to it

- Pointers serve two purposes:
  - efficient access to elaborated objects (as in C)
  - dynamic creation of linked data structures

Pointers Syntax & operations

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

- Variables in an imperative language may use
  - either a value model
  - or a reference model
  - or some combination of the two

- **Variables of built-in Java types** (integers, floating-point numbers, characters, and Booleans) employ **a value model**
- **variables of user-defined types** (strings, arrays, and other objects in the object-oriented sense of the word) employ **a reference model**

Pointers Syntax & operations- Java

- 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

Pointers And Recursive Types in C

- C pointers and arrays

int *a == int a[]

int **a == int *a[]

  - declaration allocates an array if it specifies a size for the first dimension
  - otherwise it allocates a pointer

```
int **a, int *a[];   \\pointer to pointer to int
int *a[n]; \\n-element array of row pointers
int a[n][m]; \\2-d array
```

- Given this initialization:

```
int n;
int *a;  //pointer to integer
int b[10];//array of 10 integers
a = b;//make a point to the initial element of b
```

- Note that the following are equivalent:

```
n = a[3];
n = *(a+3); //pointer arithmetic: addition of an integer k produces a pointer to the element k positions and prefix * is a pointer dereference operator.
```

- Compiler has to be able to tell the size of the things to which it point
  - following aren't valid:

```
int a[][]   bad
int (*a)[]   bad
```

  - C declaration rule:

```
int *a[n], n-element array of pointers to integer
int (*a)[n], pointer to n-element array of integers
```

Pointers And Recursive Types

- Dangling ref - **no object for a binding** (e.g., a pointer refers to an object that has already been deleted)
- Problems with dangling pointers are due to
  - explicit deallocation of heap objects
    - only in languages that *have* explicit deallocation
  - implicit deallocation of elaborated objects