Red = Couldn't find any info / don't know what it means

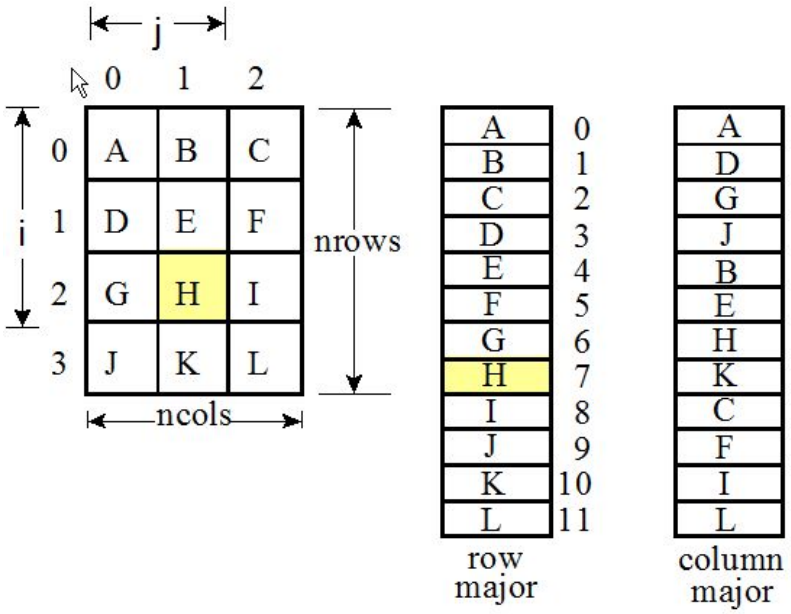
Green = Skip

Learning Objectives:

- **Understand binding and typing =**
  - Defined as an association between an entity and an attribute.
    - Such as a variable and its type or value, or between an operation and a symbol.
  - Simply defined as, assigning one thing to another.
- **Understand variable scope**
  - Defined as the range of code over which a variable is visible. (*Examples*)
- **Be able to identify both static and dynamic scope**
  - Static scoping is when a variable is referenced to the uppermost declaration in a program.
    - In this scoping a variable always refers to its top level environment. (*Source*)
  - Dynamic scoping is when a variable is referenced to the most recent declaration in the program.
    - a global identifier refers to the identifier associated with the most recent environment.
    - dynamic scoping the compiler first searches the current block and then successively all the calling functions. (*Source*)
  - *Example* (Chegg)
- **Understand basic data types and their implementation**
  - Records
    - A possibly heterogeneous aggregate of data elements in which the individual elements are identified by names. (*Sebesta Chapter 6 Slides*) (*Examples*
  - Arrays
    - Collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. (*Examples and Source*)
  - Unions
    - User defined data type
    - Only one spot in memory that can be referred to with multiple data types
      - Depending on how it is created
      - Allocates memory space for the largest data type
    - All members share the same memory location (*Examples and Source*)
  - Primitives
    - Most basic form of a data type, not defined in terms of another data type.
- **Know implementation of arrays**
  - FORTRAN arrays:
    - Key differences
      - Array index ranges may start and end at an arbitrary integer, not

limited to 0 to n-1 as in C. So you can start your array at position - 50 and end it at 5 if you wanted to.

- Fortran uses *parenthesis* to refer to a particular index of an array
    - Example:
      x(3) refers to the third item in the x array
    - Reason for this: square brackets [ ] weren't an option on the keyboard yet.
- Most languages store arrays in row-major order, Fortran stores them in column major order:



    - ^ note the difference between the position of H. Follow the path to H starting from A in both orders.
    - DIMENSION y (4, 3)
        - ^ This is a two dimensional array, with 12 elements, named y:

	column 1	column 2	column 3
row 1	y(1,1)	y(1,2)	y(1,3)
row 2	y(2,1)	y(2,2)	y(2,3)
row 3	y(3,1)	y(3,2)	y(3,3)
row 4	y(4,1)	y(4,2)	y(4,3)

Array element:	y(1,1)	y(2,1)	y(3,1)	y(4,1)	y(1,2)	y(2,2)	...y(3,3)	y(4,3)
Position:	1	2	3	4	5	6...	11	12

        -
        - ^Note the numbering: First element is 1 (not 0).
        - ^Note how MEMORY is laid out. With column major order, the elements still have the same layout ["array name(row number, column number)"], however, memory stores them going down the columns.

            - So for example: y(1,2) would be the second item placed in memory under *row major* order, but in *column major* order, it is the fifth element stored in memory.
        - REAL, DIMENSION(0:9) :: C
            - ^ array dimensioned with ten storage slots, 0 thru 9
        - INTEGER, DIMENSION(10) :: A
            - ^ array dimensioned with ten storage slots, 1 thru 10
    - <u>C Arrays:</u>
        - Usual subscript symbol is [ ] (this is true for most languages)
        - Arrays are stored in row-major order.
        - C-based languages subscript begins at 0, and the upper bound is provided so that enough storage is set aside.
        - int a[4] = {0, 1, 2, 3};
            - Array of 4 elements.
        - int a[4] = {0, 1, 2, 3};
          int *p = a;
            - ^note: *p == a[0] == *(a)
              *(p+1) == a[1] == *(a + 1)
            - ^ The array access operator [ ] is really only a shorthand for pointer arithmetic + dereference.
        - int a[3][4];
            - Array a has 3 rows, 4 columns
    - <u>Ada Arrays</u>
        - Usual subscript symbol is { }
- **Be able to program and/or trace simple FORTRAN, Python, C**
    - For some examples in various languages, check out:
        - FORTRAN: (*Link*)
        - Python: (*Link*)
        - C: (*Link*)

Chapter 5

- **Variables=(name,address,value,type,lifetime,scope)**
    - Name: not all variables have them
    - Address: the memory address with which it is associated
        - Can vary during execution
        - If two variables can be used used to access the same memory location, they are called aliases
            - Aliases are created with pointers, reference variables, unions
    - Type: determines the range of values of variables and the set of operations that are defined for values of that type

        - String “Midterm2” , Type: String
    - Lifetime: The time during which it is bound to a particular memory cell
    - Scope: Defined as the range of code over which a variable is visible.
        - Examples are given in the other mention of scope earlier in the document.
    - (*Sebesta Slides, Chapter 5 from Blackboard*)
- **Aliases, binding and binding times**
    - Aliases: A concept in some languages, most notably C and C++, refers to where two different expressions or symbols refer to the same object. (*Source*)
    - Binding: Defined as an association between an entity and an attribute.
        - Better explained earlier in the document.
    - Binding times: is the time at which a binding takes place.
        - During runtime:
            - On entry to a subprogram or block
                - Binding of formal to actual parameters
                - Binding of formal parameters to storage locations
            - At arbitrary points during execution
                - binding of variables to values
                - binding of names to storage location in Scheme.
                    - e.g. (define size 2)
        - During compile time:
            - Bindings chosen by the programmer
                - Variable names
                - Variable types
                - Program statement structure
            - Chosen by the translator
                - Relative location of data objects.
            - Chosen by the linker
                - Relative location of different object modules.
    - (*Source for binding time*)
- **Names: case-sensitivity, length, style (_ vs. camel), reserved or not**
    - Case sensitivity:
        - Disadvantage: readability
            - Names in the C-based languages are case sensitive
            - Names in others are not
    - Length:
        - If too short, they cannot be connotative
            - C99:
                - no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
            - C# and Java:
                - no limit, and all are significant
            - C++:
                - no limit, but implementers often impose one

            - FORTRAN
                - The earliest programming languages used single-character names (because in math, single-character variable names were common)
                - FORTRAN I broke the tradition by allowing up to 6 characters in its names.
    - Special characters
        - PHP:
            - all variable names must begin with dollar signs
        - Perl:
            - all variable names begin with special characters, which specify the variable's type
        - Ruby:
            - variable names that begin with @ are instance variables; those that begin with @@ are class variables
    - Special words
        - Keywords:
            - Word that is special in certain contexts
        - Reserved word:
            - A special word that cannot be used as a user-defined name
        - Issues with reserved words can occur if there are too many.
    - (*Chapter 5 Sebesta slides from Blackboard*)
- **Addresses can vary (think stack-dynamic and redeclarations); meaning of l-value**
    - Stack-dynamic variables:
        - one that is bound to an address on the stack, which is dynamically (during run-time) allocated for that purpose.
        - It may also be unbound during run-time, and its memory cell deallocated by being popped off the stack. (*Source*)
    - Redeclaration:
        - Redeclaring a variable in another function would give the local variable a new memory location.
    - L-value (referred to as l-value, but I put capital 'L' to not mistake it for 'I')
        - refers to memory location which identifies an object.
        - l-value may appear as either left hand or right hand side of an assignment operator.
        - l-value often represents an identifier. (*Source*)
    - R-value
        - refers to data value that is stored at some address in memory.
        - r-value is an expression that can't have a value assigned to it which means r-value can appear on the right but not on the left hand side of an assignment operator. (Same source as L-value)
- **Type as range of values and applicable operations; abstract vs. physical memory cells**
    - Abstract (virtual) memory cell:

        - the physical cell or collection of cells associated with a variable (*Sebesta Ch. 5*)
    - Physical memory cell:
        - A virtual memory cell mapped to a physical memory address.
            - The process is complicated, it is explained well here: *Link*
- **Binding times: language-design, language-implementation, compile-time, link/load/run-time**
    - Language design time
        - bind operator symbols to operations
    - Language implementation time
        - bind floating point type to a representation
    - Compile time
        - bind a variable to a type in C or Java
    - Load time
        - bind a C or C++ static variable to a memory cell
    - Runtime
        - bind a non static local variable to a memory cell
    - *Source: Sebesta Slides Chapter 5*
- **Static vs. dynamic; physical mapping (segmentation/paging etc.) abstracted away…**
    - Static binding
        - If the binding first occurs before run time and remains unchanged throughout the program's execution.
    - Dynamic binding
        - Binding first occurs during execution or can change during execution of the program. (*Sebesta Chapter 5 Slides*)
        - (*Example in Java and more info*)
    - Physical mapping
        - Describes the process of memory abstraction. As stated above, it's complicated. Read more here: (*Link*). Has info on paging, idk what segmentation is.
- **Understand explicit and implicit static type binding**
    - Explicit: a statement in a program that lists variable names and specifies their types
        - int x;
    - Implicit: means of associating variables with types through default conventions.
        - Implicit is something that very few programs do anymore. An example of implicit binding would be how in Fortran: I, J, K, L, M, N are recognized by the language as INTEGER otherwise REAL. (*Source*)
- **Distinguish declaration from definition**
    - Declaration
        - Provides basic attributes of a symbol
            - Type and name
        - Example: class MyClass;
    - Definition

        - Provides all the details of that symbol
            - If it's a function, what it does
            - If it's a class, what fields and methods it has
            - If it's a variable, where that variable is stored (*Source*)
        - Example: x = 3;
- **Understand an example of dynamic type binding**
    - Dynamic type binding can be achieved in a language like Java with @Override (*Example and Source*)
- **Four categories of storage binding: static, stack-dynamic, explicit and implicit heap-dynamic**
    - How scalar variables are categorized
    - Static
        - Bound to memory cells before execution and remains bound to the same memory cell throughout execution
    - Stack-dynamic
        - Storage bindings are created for variables in the run time stack when their declaration statement are elaborated
            - If scalar, all attributes except address are statically bound
    - Explicit heap-dynamic
        - Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution.
            - Referenced only through pointers or references
    - Implicit heap-dynamic
        - Allocation and deallocation caused by assignment statements and types not determined until assignment.
    - (*Source including more information, as well as table's information*)

Variable Category	Storage Binding time	Dynamic storage from	Type binding
Static	Before Execution	~~~~~~~~~~~~~~	Static
Stack-dynamic	When declaration is elaborated (run time)	Run-time stack	Static
Explicit heap-dynamic	By explicit instruction (run time)	Heap	Dynamic
Implicit heap-dynamic	By assignment (run time)	Heap	Dynamic

- **History/structural sensitivity of static variables; lifetime**
    - History talks about tracing the variable, declared etc.
    - Lifetime of a static variable is the lifetime of the function (*Source*)
- **Relationship of scope to stack-dynamic lifetime**

    - Lifetime is while the subprogram is active
        - Scope is directly related to the variable's lifetime (*Source*)
- **Advantages and disadvantages of explicit heap-dynamic variables**
    - Advantages:
        - Provides for dynamic storage management
    - Disadvantages:
        - Unreliable (forgetting to delete)
        - Difficult of using pointer and reference variables correctly
        - Inefficient
    - (*Sebesta Slides Chapter 5*)
- **Implicit heap-dynamic management; garbage collection**
    - lifetime is from implicit allocation to implicit deallocation
        - If you forget to deallocate, (which has to do with garbage collection), it is inefficient. (*Source and Sebesta Chapter 5 Slides*)
- **Static type checking, coercion and dynamic type checking (think virtual)**
    - If all type bindings are static nearly all type checking can be static
        - Static type checking is difficult when the language allows a cell to store a value of different types at different times, such as C unions, Fortran Equivalences or Ada variant records. (*Source*)
- **Strong typing examples and nearly strong typing; advantage and disadvantage**
    - Strong typing
        - Type errors are always detected.
        - There is strict enforcement of type rules with no exceptions.
        - All types are known at compile time, i.e. are statically bound.
        - With variables that can store values of more than one type, incorrect type usage can be detected at run time.
    - Advantages:
        - Strong typing catches more errors at compile time than weak typing, resulting in fewer run time exceptions.
        - Detects misuses of variables that result in type errors. (*Source*)
    - Disadvantages:
        - Programmer has less control of the language
            - A function can return one type in one case, and another type in another
            - Arguments can be of any type without having to write overloaded versions of the function.
            - An attribute can be read from an object, no matter what type it is, as long as it has that attribute. (*Source*)
- **Type compatibility and type equivalence (name vs. structural)**
    - Name type compatibility
        - Means that two variables have compatible types if:
            - They are defined in the same declaration or
            - They are defined in declarations that use the same type name.

    - Structural type compatibility
        - Means that two variables have compatible types if their types have identical structures (*Source*)
- **Static scope rules and static ancestry; follow an example of name-resolution using scope; application to blocks**
    - In static scoping, the scope of a variable can be determined at compile time, based on the text of a program.
    - To connect a name reference to a variable, the compiler must find the declaration
        - Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name.
        - Enclosing static scopes to a specific scope are called its static ancestors; the nearest static ancestor is called a static parent. (*Sebesta slides Chapter 5*)
- **Hiding possibilities of scope and evaluation of this**
    -
- **Problems with static scope include forcing programs into unusual configurations**
    -
- **Dynamic scope rules and dynamic ancestry; follow an example of name-resolution given a sequence of calls**
    - Based on calling sequences of program units, not their textual layout.
    - References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution at this point. (*Source*)
- **Referencing environments (know definition) (just another way of capturing scope)**
    - Collection of all names that are visible in the statement
    - In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
    - In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms. (*Sebesta Chapter 5 Slides*)

Chapter 6

- **Primitive data types vs. user-defined data types**
    - Primitive data types
        - Those not defined in terms of other data types
        - Example: int x = 5;
    - User-defined data types
        - Data types which reference a primitive type
        - Example: #define TRUE 1 (*source and examples*)
- **Basic information about these data types:**
    - 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's signed integer sizes: byte, short, int, long (*Sebesta Chapter 6*

Slides)

- Floating-point
  - Model real numbers, but only as approximations
  - Languages for scientific use support at least two floating-point types (e.g., float and double; sometimes more
  - Usually exactly like the hardware, but not always
  - IEEE Floating-Point (convert floating point to decimal representation)
    - Standard 754 (*Sebesta Chapter 6 Slides*)
- Decimal
  - For business applications (money)
    - Essential to COBOL
    - C# offers a decimal data type
  - Store a fixed number of decimal digits, in coded form (BCD)
  - Advantage
    - accuracy
  - Disadvantages
    - limited range, wastes memory (*Chapter 6 Sebesta Slides*)
- Boolean
  - Simplest of all
  - Range of values
    - two elements, one for "true" and one for "false"
  - Could be implemented as bits, but often as bytes
  - Advantage
    - Readability (*Chapter 6 Sebesta Slides*)
- Character
  - Stored as numeric codings
  - Most commonly used coding: ASCII
  - An alternative, 16-bit coding: Unicode (UCS-2)
    - Includes characters from most natural languages
    - Originally used in Java
    - C# and JavaScript also support Unicode
  - 32-bit Unicode (UCS-4)
    - Supported by Fortran, starting with 2003 (*Chapter 6 Sebesta Slides*)
- Character strings: array or built-in or object, null-termination and associated errors, regular expressions for pattern matching
  - C & C++: "Limited Dynamic Length", use char arrays as strings (not primitive). Also use the null termination character to indicate end of string
  - Java: "Static", use the built-in String class. Sebesta says strings in Java are primitives, despite being instantiated from a class
- Character strings: static or dynamic length and associated bookkeeping, three styles of storage management for dynamic
  - Static length: length is stored at compile time
  - Limited dynamic length: might need a descriptor for runtime to keep track

of length (though not in C and C++ bc they have the null character)
        - Dynamic length: does need a descriptor to keep track of length
            - Allocation/deallocation is costly
- **What is an enumeration type? Know the three design issues on p. 259**
    - An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it.
        - A good example would be the days of the week (*Source*)
    - The design issues for enumeration types are as follows:
        - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
        - Are enumeration values coerced to integer?
        - Are any other types coerced to an enumeration type?
    - All of these design issues are related to type checking. If an enumeration variable is coerced to a numeric type, then there is little control over its range of legal operations or its range of values. If an int type value is coerced to an enumeration type, then an enumeration type variable could be assigned any integer value, whether it represented an enumeration constant or not. (*Copied directly from the book. Sebesta Chapter 6.4.1*)
- **What are subrange types? How do they differ from derived types in terms of type checking?**
    - Defines a subset of the values of a particular type.
        - Subranges differ from derived types in terms of type checking since a subrange type can be used to detect values that are unreasonable since it is out of a given boundary. (*Source*)
- **Know the points made in section 6.4.3**
    - Advantages of Enumeration
        - Readability
            - Named values are easily recognized, whereas coded values are not
        - Reliability
            - No arithmetic operations are legal on enumeration types
                - For example, prevents adding days of the week
            - No enumeration variable can be assigned a value outside its defined range
                - If the colors enumeration type has 10 enumeration constants and uses 0..9 as its internal values, no number greater than 9 can be assigned to a colors type variable.
        - C does **not** provide the reliability functions of enumerated types
    - In C++
        - Numeric values can be assigned to enumeration type variables only if they are cast to the type of the assigned variable.
        - Numeric values assigned to enumeration type variables are checked to

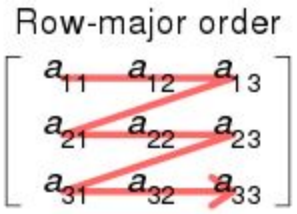
determine whether they are in the range of the internal values of the enumeration type.
    - Type checking can be inefficient if…
        - The user uses a wide range of explicitly assigned values
        - enum colors {red = 1, blue = 1000, green = 100000}
            - In this example, a value assigned to a variable of colors type will only be checked to determine whether it is in the range of 1..100000. (*Copied from Chapter 6 of Sebesta textbook*)
- **Know the design issues for array types on p. 263**
    - The primary design issues specific to arrays are the following:
        - What types are legal for subscripts?
        - Are subscripting expressions in element references range checked?
        - When are subscript ranges bound?
        - When does array allocation take place?
        - Are ragged or rectangular multidimensional arrays allowed, or both?
        - Can arrays be initialized when they have their storage allocated?
        - What kinds of slices are allowed, if any? (*Copied from Sebesta textbook*)
- **Why does Fortran choose to use the same syntax for array indexing as for function calls?**
    - When Fortran was created, square brackets were not on the keyboard([ ]) (*from lecture*)
- **What is involved in subscript range checking? Must it be done? Do all languages do it?**
    - Subscript range checking involves the compiler generating language to check the correctness of every subscript expression.
        - they do not generate such code when it can be determined at compile time that a subscript expression cannot have an out-of-range value
    - This must not be done, and not all languages do it.
        - Java does it
        - In C, subscript ranges are not checked because the cost of such checking
        - was (and still is) not believed to be worth the benefit of detecting such errors. (*Copied and paraphrased from Chapter 14 of Sebesta textbook*)
- **How is array initialization handled in different languages?**
    - Some language allow initialization at the time of storage allocation
        - C, C++, Java, C# example
            - int list [] = {4, 5, 7, 83}
        - Character strings in C and C++
            - char name [] = ″ freddie″ ;
        - Arrays of strings in C and C++
            - char *names [] = {″ Bob″ , ″ Jake″ , ″ Joe″ };
        - Java initialization of String objects
            - String[] names = {″ Bob″ , ″ Jake″ , ″ Joe″ }; (*Sebesta Chapter 5 Slides*)

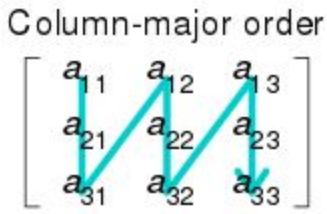
- **What is a static array?**
    - Defined by Sebesta textbook:
        - A static array is one in which the subscript ranges are statically bound and storage allocation is static (done before run time)
    - Simpler definition:
        - A static array is an array whose size cannot be altered. (*Source*)
        - Static arrays store their values on the stack, and their size must be known at compile time. (*Source with examples*)
    - Advantage: efficiency
- **What is a fixed stack-dynamic array? What is an advantage of using them?**
    - Defined by Sebesta textbook:
        - A fixed stack-dynamic array is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution.
    - Simpler definition:
        - You know the size of your array at compile time, but allow it to be allocated automatically on the stack (the size is fixed at compile time but the storage is allocated when you enter its scope, and released when you leave it) (*Source*)
    - The advantage of fixed stack-dynamic arrays over static arrays is space efficiency
        - A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time. The same is true if the two arrays are in different blocks that are not active at the same time.
    - The disadvantage is the required allocation and deallocation time. (*Sebesta Chapter 6*)
- **What is a stack-dynamic array?**
    - 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.
    - You don't know the size until runtime. (*Source*)
- **What is a fixed heap-dynamic array?**
    - Defined by Sebesta textbook:
        - Similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated.
        - The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the storage is allocated from the heap, rather than the stack.
    - The advantage of fixed heap-dynamic arrays is flexibility—the array's size always fits the problem.
    - The disadvantage is allocation time from the heap, which is longer than allocation
    - time from the stack. (*Copied from Sebesta Chapter 6*)
- **What is a heap-dynamic array?**
    - Defined by Sebesta textbook:
        - One in which the binding of subscript ranges and storage allocation is

dynamic and can change any number of times during the array's lifetime.
    - The advantage of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for space changes.
    - The disadvantage is that allocation and deallocation take longer
    - and may happen many times during execution of the program. (*Copied from Sebesta Chapter 6)*
- **(In all of these, what are the binding times of indices and storage?)**
    - Binding time for indices and storage can be determined by where it is trying to bind to in memory.
    -

ARRAY TYPE	MEMORY LOCATION	BINDING OF SUBSCRIPTS	CHARACTERISTICS	EXAMPLE
static	data segment (low memory)	when program is loaded	size is unchangeable during process lifetime	static float cost[10];
fixed stack-dynamic	stack frame	size fixed at load time		
stack-dynamic			function declares array as a local variable	
fixed heap-dynamic	heap			
heap-dynamic				

- **How are declare blocks in Ada used with stack-dynamic arrays on p. 266**
    -
- **What keywords or function calls are used in C/C++ for allocation and deallocation of fixed heap-dynamic arrays?**
    - Allocation: malloc()
    - Deallocation: free() (*Source*)
- **How are arrays handled in Java?**
    - (*Link*)
- **Be familiar with the various initialization methods in section 6.5.5**
    - C, C++, Java, and C# allow initialization of their arrays.
        - int list [] = {4, 5, 7, 83};
            - The array list is created and initialized with the values 4, 5, 7, and 83.
    - character strings in C and C++ are implemented as arrays of char.
        - char name [] = "freddie";
            - The array name will have eight elements, because all strings are terminated with a null character (zero), which is implicitly supplied by the system for string constants.
        - char *names [] = {"Bob", "Jake", "Darcie"};
            - The literals are taken to be pointers to characters, so the array is an array of pointers to characters.
            - For example, names[0] is a pointer to the letter 'B' in the literal character array that contains the characters 'B', 'o', 'b', and the null character.
    - In Java, similar syntax is used to define and initialize an array of references to String objects.
        - String[] names = ["Bob", "Jake", "Darcie"]; (*Copied from Chapter 6 Sebesta*)
- **What kinds of operations are typical for arrays?**
    - Array assignment
        - Sets an element of an array equal to something

    - Array catenation
        - Add onto the array
    - Comparison for equality and inequality
    - Array Slices
        - A slice of an array is some substructure of that array (*Chapter 6 Sebesta*)
- **What is a slice of an array (give a specific example)?**
    - A slice of an array is some substructure of that array. For example, if A is a matrix, then the first row of A is one possible slice, as are the last row and the first column.
    - A slice is not a new data type. Rather, it is a mechanism for referencing part of an array as a unit. If arrays cannot be manipulated as units in a language, that language has no use for slices. (*Chapter 6 Sebesta*)
- **Given a two-dimensional array, show how the address of an element is computed from the indices and element type**
    - Location (a[I,j]) = address of a [row_lb,col_lb] + (((I - row_lb) * n) + (j - col_lb)) * element_size (*Chapter 6 Sebesta Slides*)
- **What are row-major and column-major order? Be able to show the difference in an example**
    - Row-major order:
        - The elements of the array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the first subscript, and so forth.
        - Used in most languages
        - 

Ordering, not addressing in memory
    - Column-major order:
        - The consecutive elements of a column reside next to each other
        - Used in Fortran
        - 

Ordering, not addressing in memory

For example, the array

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

could be stored in two possible ways:

Address	Row-major order	Column-major order
0	a_{11}	a_{11}
1	a_{12}	a_{21}
2	a_{13}	a_{12}
3	a_{21}	a_{22}
4	a_{22}	a_{13}
5	a_{23}	a_{23}

(*Source*)

- **What is an associative array?**
  - An unordered collection of data elements that are indexed by an equal number of values called keys
    - User-defined keys must be stored (*Sebesta Slides Chapter 6*)
  - Another definition: Associative arrays are arrays that use named keys that you assign to them. (*Source with examples*)

Two's Complement (No clue where to put this but it's going to be on the test apparently):

- Signed binary integers -- this means that the leading bit represents whether the number as a whole is a negative number or not. For example:
  - 1001 -- if declared as a *signed* bit, then this number represents a *negative* number because the leading bit is 1; so the two's complement must be found to extract its value.
    - *How to find a two's complement*:
      - Flip the bits, and add one. So for the above signed bit of 1001 the two's complement would be found as follows:
        - Flip the bits: 0110
        - Add one: 0110 + 0 001
        - Result: 0111
        - Convert 7

        - Add neg sign -7 ← this is your value
    - 1001 -- if declared as a unsigned bit, then this number represents a positive number and you can just find its binary value like normal (so it would just be a 9)