Final Exam Study Guide

* C++ Inheritance
  + Inheritance: hierarchical relationships between types seen in object-oriented programming (everything in Java inherits from Object)
    - Super-class (base class, parent class)
    - Sub-class (derived class, child class)
      * **class subClassName : public superClassName {** is C++ equivalent of **public class subClassName extends superClassName** in Java
      * Inherits: data members defined in parent class, member functions of the parent class, same initial data layout as the parent class
        + However, CANNOT access any member fields/functions that are PRIVATE 🡪 must call super-class’s methods to access them
      * Can add: new data members, member functions, constructors, destructors, etc.
    - Derived classes are constructed from the top down (super-class constructor called first to initialize its data members, and sub-class constructor called last to initialize its data members): space is allocated on stack or heap for ALL data members (base and derived class)
  + Single inheritance problems: if a sub-class (and its super-class) are NOT primitive types, but must also be comparable, serializable, etc. 🡪 to solve this, C++ has multiple inheritance (similar to Java interfaces)

**class subClassName : public superClass1, public superClass2, public superClass3**

* + - **Comparable**: C++ class with a compareTo method (like Java’s comparable) 🡪 if a comparable is the parameter for a method, than anything that inherits from comparable can be passed through
    - **Serializable**: enables a type to be written from a disk
    - Java interfaces allow **type-substitution** for multiple classes that implement the same interface: if parameter called for is comparable, and a class *sphere* implements the comparable interface, then a *sphere* (or any type that implements the comparable interface) can be substituted for a *comparable* as the argument for the method
  + Other C++ Inheritance Issues
    - Multiple inheritance can create semantic ambiguities

**class A {**

**void foo (bar = 7);**

**}**

**class B {**

**void foo (bar = 1);**

**}**

**class C: public A, public B {**

Which foo() is used? Which bar? When bar is set, which bar is set? Do both foos use the same bar?

* + - * Replicated multiple inheritance: a class inherits via two different paths (default)
        + Ex: there will be an **A 🡪 bar** and a **B 🡪 bar** which are different
      * Shared multiple inheritance: a parent class is specified as virtual and therefore the duplicate fields are merged into one field
      * Non-replicated (non-repeated): a language does not allow shared or replicated fields, such as one that allows no common ancestors
      * Mix-in: faking multiple inheritance (such as Java’s use of interfaces)
* Static v. Dynamic Dispatch
  + Static dispatch: the decision on which member function to invoke is made using the compile-time type of the object
  + Dynamic dispatch: the decision on which member function to invoke is made using the run-time type of the object
    - For a super-class shape and sub-class square, which both have a getLength() method

**shape \*ptr = new square();**

**ptr->getLength();** //Will have to generate a **virtual method table**

and follow the pointers described below in order to discover which getLength() method to use

* + - Incurs run-time overhead: program must maintain extra information, compiler must generate extra code in order to determine which method to use
      * C++ uses the **virtual** keyword to indicate that, **at run-time**, the compiler must check if the sub-class implements the same method, and if it does, then the sub-class’s method should be used instead of the parent class’s method (note: every method in Java is automatically marked virtual – programmer does not have to specify)
        + **virtual** must be used in the forward declarations of the methods (in the .h file)

**class A {**

**virtual void foo()**

**}**

**class B : public A {**

**virtual void foo()**

**}**

* + - * Virtual Method Tables (VMT): an array of subroutine pointers that is constructed by compiler whenever a derived class is constructed
        + During construction of an object:

Base constructor called and table is made with the base constructor’s method addresses 🡪 derived constructor called and the table is updated to hold the new address’s of any overlapping methods (methods with the virtual keyword)

* + - * + Then, each object contains pointer to VMT
        + When a method is called: virtual method follows pointer to object 🡪 object follows pointer to VMT 🡪 correct method pointer identified and followed to the method (either base or derived class)
* Abstract classes
  + Methods are pure virtual methods (pure virtual functions): must use the virtual keyword, and have ONLY a forward declaration

**virtual void bar() = 0;**

* + - If an abstract class contains ONLY pure virtual methods, then it is the equivalent of a Java interface
      * Therefore, an abstract class CANNOT be instantiated 🡪 sub-classes must implement the methods (and must implement ALL of the methods if it is to be instantiated)
* Covariant Arrays: an array of super-class objects can hold any objects that inherit that super-class
  + Type-safety: extent to which a programming language prevents type errors (erroneous behavior caused by discrepancies between different data types)
* C++ 11
  + Move constructors: method of preventing need for deep copy
    - Ex: assign a global vector X from a local vector Y, where Y is going out of scope 🡪 instead of performing deep copy, just make vector X point to vector Y’s array, and don’t **delete vector Y’s array** when it goes out of scope
  + Uniform initialization: curly brackets {} used for both constructor calls and initialization lists because:
    - Programmers want variable definition that takes in an object
    - Compiler wants variable definition that takes in a function as a parameter
  + Range-based for loops: similar to for each loops in Java

**for (int &x : myArray) {**

**x \*= 2;**

**}**

* + New (optional) function/method syntax: trailing return type that solves issues when adding C++ qualifiers (such as const)
  + **New NULL**: there is now a nullptr, whereas before NULL was actually just an int 0, which is difficult to check against a pointer
  + **Smart pointers**: pointers that include **memory management** algorithms
    - auto\_ptr: caller must de-allocate the pointer
    - unique\_ptr: forces only one pointer to point to the given object (the “old” pointer must point to nullptr)
    - shared\_ptr: performs reference counting, will be de-allocated by the compiler when nothing points to it anymore
    - weak\_ptr: similar to shared\_ptr but it does not modify the reference count, used to handle circular references and reference counting (like in a loop)
  + Other improvements
    - Better handing of multitasking and threads
    - New type of long long int that is guaranteed to be at least 64 bits
    - Static assertions for debugging
    - Better STL hash tables

Heaps

* Priority Queue: data structure that enables some members to have a higher priority than others
  + Uses: OS choose which process to run first, management of limited resources (ex: have limited bandwidth but want to give the best possible performance – prioritize voice calls over downloading an image because you can wait five seconds for the image, but you don’t want to wait five seconds to hear what the other person says)
  + Operations:
    - insert() : insert with a priority number, where lower number indicates higher priority
      * Worst-case Θ(log n) but on average half of elements will move up one level, a quarter will move up two levels, etc. so it averages out to constant time
    - findMin() : finds element with highest priority (lowest priority number)
      * If two elements have the same priority, then it doesn’t matter which one is found
      * Worst-case Θ(1)
    - deleteMin() : finds and returns element with highest priority (lowest priority number), and then removes that element from the queue
      * Worst-case (and average case) Θ(log n)

All of the data structures that we know thus far, and their time complexities:

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **insert()** | **findMin()** | **delete()** |
| Unsorted Array | Θ(1) amortized | Θ(n) | Θ(n) |
| Unsorted Linked List | Θ(1) | Θ(n) | Θ(n) |
| Sorted Array | Θ(n) | Θ(1) | Θ(1) |
| Sorted Linked List | Θ(n) | Θ(1) | Θ(1) |
| Binary Search Tree | Θ(n) or Θ(1) | Θ(n) | Θ(n) |
| AVL/Red-Black Tree | Θ(log n) | Θ(log n) | Θ(log n) |
| Hash Table | Ideally constant | Θ(n) | Θ(n) |

* + There is no data structure in the table above that fulfills the necessary time complexity requirements of a priority queue 🡪 instead, a binary heap must be used
    - Note: binary heap is never used for anything other than a priority queue, and a priority queue is never implemented with anything other than a binary heap
* Binary Heap: a binary tree with a different structure property and a different ordering property
  + Heap structure property: binary tree is completely filled with the exception of the bottom row, which is filled left to right (**almost complete binary tree**)
    - For h = 0 : one node
    - For h = 1 : left child or two children
    - For h ≥ 2 :
      * New node inserted into the left sub-tree so that the **left** sub-tree is **almost complete** with height *h-1* and the **right** sub-tree is **complete** with height *h-2*
      * New node inserted into the right sub-tree so that the **left** sub-tree is **complete** with height *h-1* and the **right** sub-tree is **almost complete** with height *h-1*
  + Heap ordering property: for every non-root node X, the key (priority) in the parent of X is less than or equal to the key in X, and thus the tree is partially ordered
    - Min-Heap: know that parent is less than child 🡪 to make a Max-Heap, you would just swap the ordering property
  + Heap representation using an array
    - Each spot in the array *i* represents a node
      * Left child is at *2\*i*
      * Right child is at *(2\*i)+1*
      * Parent is *floor(i/2)*
    - Advantages:
      * Arrays are better with cache
      * Better than pointers
        + Far faster than using dynamic memory (each node would have to have a left, right, and parent pointer)

Pointers use dynamic memory while arrays use static memory 🡪 **dynamic memory requires allocating memory** (buddy blocks or fixed-size-blocks) which adds overhead to new/malloc()

* + - * + \*2, /2, and + operations are faster than pointer dereferencing
        + More compact in memory than pointer use (think about pages of memory)
    - Disadvantage is that you have to double array size when it is full, however this is rare and therefore is amortized out 🡪 people only use arrays to implement heaps
  + Primary operations
    - findMin() is constant time: just look at the root node
    - insert()
      * Node is inserted in the proper spot to preserve the heap structure property
      * Percolate the node up: if the inserted node is less than its parent, then swap the inserted node and the parent 🡪 continue until the node is in its proper spot
      * Expected running time is
        + With the assumptions represented above (half inserts move up one level, a quarter move up two levels, etc.) 🡪 turns out to be Θ(1) amortized, with worst-case Θ(log n)
    - deleteMin()
      * The minimum node is the root node: return that node, and then place the last inserted node (right-most leaf) in its spot to preserve the structure property
      * Percolate the node down: swap the new root node with the lesser of its two children (to minimize percolations) until the node is in its proper spot
        + If the heap was a max-heap, then you would swap new node with the greater of two children in order to minimize percolations
  + Other possible (but less common) operations 🡪 running time for all is Θ(n) because there is a find() involved
    - **findMax()** : find the maximum element
      * Linear running time because the max element could be literally anything (the only known element is the min element at the root)
    - **decreaseKey (processId, amount)** : raise the priority and percolate up
    - **increaseKey (processId, amount)** : lower the priority and percolate down
    - **remove (processId)** : remove a process
      * Use decreaseKey(processId, -infinity) to move the process to the root node, and then delete the root node with deleteMin()
    - **expandHeap()** : when the heap fills up (the array fills up, if you aren’t using a vector) copy the heap into a new space
  + HeapSort: *n* elements are inserted and then *n* elements are removed, has a running time of Θ(n log n)
    - NOT a stable sort: initial order is **not** maintained if two elements are considered equal
    - MergeSort is a stable sort and therefore is favored over HeapSort
* Perfect binary tree (complete binary tree) : all leaf nodes are the same depth, all internal nodes have two children
  + Has height *h*, *2h+1-1* nodes, *2h-1* internal nodes, and *2h* leaf nodes
  + Useful only as a comparison tool because it can only have a specified number of nodes in order to maintain the property
* Full binary tree: each node has exactly two or zero children
* File Compression: used if disk space is limited, for a file transfer, or to fit a file into memory more easily
  + File: named collection of information (C++ program, application executable, word document, email, web page, picture, audio, video, etc.)
  + Lossless Data Compression: X = X’
    - No data is lost and compression ratios of 4:1 are attainable
    - Lossless compression of text
      * American Standard Code for Information Interchange (ASCII) uses a fixed 8 bits per character
      * Huffman Encoding: uses min-heaps and the frequencies with which characters appear to build prefix codes
  + Lossy Data Compression: X != X’
    - Information is lost irreversibly: acceptable for files consumed by our senses (picture, audio, video), data is lost but not noticeably
    - Compression ratios of 10:1 are attainable
* Huffman Encoding: more frequent characters have shorter prefix codes
  + Any character’s beginning bit string is NOT the same as any other character’s beginning bit string
  + Huffman Encoding
    - Determine the frequencies of the characters in the source file and store the characters in a min-heap with priority = frequency
    - Build the Huffman encoding tree from the min-heap in order to determine the prefix codes for each character
      * A left child in the tree means a 0, a right child in the tree means a 1
  + \*\*\*\*\*Huffman Encoding Cost
    - is how many bits it takes to encode each character of a given string, where *pi*is the frequency with which a symbol occurs, and *ri* is the length of the path from root to node
      * , where *n* is the number of characters in the file
    - Compression ratio: how many bits the Huffman encoding compressed the file by when compared to ASCII encoding
      * + *li*is the length of the prefix code, *fi* is the number of times a character occurs, and *n* is the total number of chars in the original file

Graphs

* G = (V, E), where V is vertices (nodes) and E is edges (connectors)
  + Edges (*v*, *w*) connect two adjacent nodes of the graph *v*, *w*
    - Directed Graphs and digraphs have ordered pairs (*v*, *w*) where movement can only occur in one direction
    - Undirected graphs have unordered pairs (*v*, *w*) where movement can occur in both directions
  + Path: sequence of nodes *w1*, *w2*, …, *wn* such that (*wi*, *wi+1*) is an element of E for *i* between 1 and *n*
    - Path length: number of edges in path
    - Simple path: all vertices are distinct
  + Graphs can have cycles:
    - Digraph: path length ≥ 1, the beginning and ending nodes are the same
    - Undirected graph: path length ≥ 1, the beginning and ending nodes are the same, and all edges (connectors) are distinct
  + A loop means that there is an edge from a node to itself
  + There is a path from every vertex to every other vertex in a strongly connected graph
    - A weakly connected graph is a directed graph whose underlying *undirected* graph is strongly connected
  + A complete graph has an edge between every pair of vertices
* Directed Graphs (Digraphs)
  + Strongly or weakly connected
  + Directed Acyclic Graph (Digraph, DAG): no cycles
  + Digraph representation
    - Adjacency Matrix: a 2D array representation of the graph **array[i][j]** where every array spot represents the pathway from *i* to *j* with the weight of that pathway
      * Advantages
        + Speed: constant look-up time
        + Good for maps where each node has many intersections – minimizes linear time search through linked list
      * Disadvantages
        + Memory: potentially wastes a lot of space
        + Poor for maps where each node has just a few intersections – matrix will be very sparse and mostly empty
    - Adjacency List: a 1D array of linked list pointers, where each spot **array[i]** represents a node, and the list out of that node holds all of the other nodes that it is connected to
      * Advantages
        + Saves memory when compared to the matrix
        + Good for maps with sparse connections, such as a road map with intersections
      * Disadvantages
        + Potentially very slow if many nodes have many connections
* Topological Sort: given a directed **acyclic** graph, construct an ordering of the vertices such that if there is a path from vi to vj, then vi appears before vj in the ordering
  + Indegree: the number of edges (*v*, *w*) going into a node
  + Algorithm: place all nodes of indegree zero in a queue 🡪 place the first eligible node of indegree zero into the topological sort 🡪 identify all of the nodes it points to, and mark those nodes as eligible by placing them in the queue 🡪 repeat until there are no more nodes in the queue
    - Time complexity: Θ(v2)
      * Queue will hold all nodes at some point and must be iterated through 🡪 every time you look at the next node in the queue, must look at all of the nodes that the new node of indegree zero is connected to
* Shortest Path Algorithms: used for map routing, internet routing, puzzle answers
  + Single source algorithms: given a source, find the shortest (weighted) path between that node and every other node in the graph
    - Un-weighted shortest path algorithm: load the start node into the queue 🡪 look at the start node’s adjacent nodes and place them in the queue with the number of edges traversed to arrive there 🡪 look at the nodes adjacent to the newly added nodes in the queue, and if:
      * They ARE NOT in the queue already 🡪 they haven’t been path’d yet, so add them to the queue with a path length +1
      * They ARE in the queue, move on
      * Time complexity is Θ(v2)
    - Breadth First Search: essentially Dijkstra’s algorithm but for un-weighted graphs, runs in Θ(v2) worst case but could be faster if every node is not connected to every other node
    - Dijkstra’s algorithm: used for weighted shortest path, assume **no negative cost edges** (assumes that adding ANY edge will increase the cost, even through adding a negative edge would decrease the cost), uses a set of known nodes and a set of unknown nodes 🡪 when all nodes are known, then the algorithm is finished
      * Each node must have a distance field, which is infinity to begin with, and a path field, which holds the node immediately prior in the path
      * Set start node’s distance to zero 🡪 find unknown node with the shortest distance *v* 🡪 mark this node as known 🡪 consider the nodes adjacent to *v* and the total distance to those nodes (*v*’s distance and the weight edge (*v*, *w*)) 🡪 if this distance is shorter than the current distance, update the distance and then mark the node as known 🡪 loop until there are no more unknown nodes
      * Time complexity is Θ(v2), which can be optimized to Θ(e log v) but that is NOT discussed here
    - Bellman-Ford Algorithm: will work for negative cost edges, there are

|V-1| iterations to ensure that the shortest path

* + - Greedy algorithms: an approximation of the shortest path, perform the best option at every point and DO NOT go back and check
  + Single pair algorithm: given two nodes, find the shortest path between them 🡪
    - There is no direct algorithm: use single source algorithm and stop when the solution enters the set *S* 🡪 running time is the same
  + All pair: not covered
* Traveling Salesperson Problem: find the least expensive Hamiltonian path in a weighted, connected graph
  + Hamiltonian path: a path in a connected graph that visits each vertex exactly once
    - Hamiltonian cycle: a Hamiltonian path that ends where it began
  + An NP-complete problem: there is no known efficient solution
    - This one runs in Θ(n!)
* Minimum Spanning Tree (MST): a sub-graph of a graph G that contains every vertex of G and is a tree, and has the minimum cost
  + Given a graph G = (V, E), find a graph G’ = (V, E’) such that:
    - E’ is a subset of E
    - |E’| = |V| - 1
    - G’ is connected
    - is minimal 🡪 for every edge in E’, sum of costs is minimal
  + Any connected graph has a spanning tree, and any two spanning trees of the same graph have the same number of nodes
  + Applications: wiring a house, Internet connections, power grids, etc.
  + Prim’s MST Algorithm: pick a root node 🡪 choose an edge (*v*, *w*) with MINIMAL WEIGHT that connects a node in the tree *v* and a node NOT in the tree *w* 🡪 add that edge to the tree
    - Use a min-heap to hold the eligible edges: look at edges connecting known and unknown nodes 🡪 once an edge is added, another node becomes known and all of its edges connecting known and unknown nodes are added to the min-heap
    - Time complexity is Θ(v2), but Θ(e log v) optimized
    - Use a set of **known** vertices and a set of **unknown** vertices
  + Kruskal’s MST Algorithm: place all edges in a min-heap 🡪 call deleteMin() to get the starting edge and include that edge in the minimum spanning tree (put it back in the min-heap) so that the edge and the two nodes it connects are a tree 🡪 repeat until there is only one tree in the min-heap (similar to Huffman encoding)
    - Θ(v2), but Θ(e log v) optimized running time
    - Use **min-heap of edges**
    - Easier to code than Prim’s but merging trees can be costly (use colors – color different trees differently)

Memory

* Pointers and references
  + **Rectangle \*rectPtr** is a declaration of a new rectangle pointer, while **\*rectPtr** accesses the rectangle (dereferences the pointer) that that pointer is pointing to after it is declared
  + **ptr1 == ptr2** is different from \***ptr1 == \*ptr2** because ptr1, ptr2 are addrs., while \*ptr1, \*ptr2 are values 🡪 the first will be different because pointers will point to a new addr. each time
  + **List &someList = list;** is the declaration of a reference to list, while **&someList** is the address of a list
    - C++ **automatically dereferences** reference types 🡪 must use & to get the addr. of a reference
    - Reference addr. CANNOT be changed
* Pointer and memory allocation issues (seg. faults)
  + Uninitialized pointers
  + Dereferencing NULL pointers
  + Dangling pointer: pointer is pointing to something valid (has memory allocated to it) 🡪 valid object deleted (allocated memory de-allocated) 🡪 try to access what that pointer is pointing to but now there is nothing there (memory no longer allocated)
    - Pointer **still exists** and then can be pointed to a new spot

**void \*realloc (void \*ptr, size\_t size)**

* + - * Alters the size of the memory object pointed to by the pointer and in doing so changes the address that the pointer points to 🡪 if used, must update ALL pointers that point to the same address or some pointers (non-updated pointers) will point to **invalid** memory
      * Θ(n) running time because “old” data must be copied into the new location
  + Memory hole: losing the address of dynamically allocated memory
    - Variable goes out of scope without being deleted
    - Pointer to memory allocation is reassigned
* Static vs. Dynamic allocation
  + Static allocation: the space required is known at compiler-time, typically placed on the stack, occurs in callee’s prologue
    - Ex: an array
  + Dynamic allocation: the space required is not known until run-time, typically placed on the heap
    - Ex: pointers in a linked list 🡪 this is why arrays are the preferred implementation for heaps
* Memory Layout
  + Stack: begins at the end of memory and grows downward towards the heap (although the two will NEVER meet) 🡪 typically need about 250 Mb, might need more with a recursive program because that results in many activation records being pushed onto the stack
    - Managed by the compiler: when **ret** is called (the program goes out of scope), the activation record is popped off the stack
  + Heap: begins right after the binary executable in memory and address space grows up
    - Managed by the programmer: must free the space or else will have memory leaks (some part of the heap is allocated but there is no pointer pointing to it – either variable went out of scope, or variable was reassigned)
      * Memory leaks reduce amount of available memory, will use Disk memory when all RAM is used up, which usually causes program to run so slow it is considered to have failed
      * OS will clean the heap up when the program is exited, but that is not efficient for long-running programs
      * Using **free()**
        + Must free all dynamically allocated memory at some point HOWEVER use caution because, like **realloc()**, if you free one pointer, any other pointers pointing to the same spot will become **dangling pointers** and point to **invalid memory**
    - Heap directory management routine is compiled into program’s code (in the libc library), takes up ~8 million bytes (compared to 40,000 for a program)
      * When **new** or **malloc()** is called, the OS and heap directory management are consulted, which adds extra time to a **new** or **malloc()** call
      * Allocation methods
        + Fixed-size-blocks allocation: a list of available blocks of memory, appropriate number of blocks allocated 🡪 good for machines with small memory, but must be searched linearly so is time expensive
        + Buddy blocks: blocks of memory stored in a tree-like structure in decreasing powers of two (4GB root node, two 2GB children, four 1 GB children, etc.) 🡪 far faster than fixed-size-blocks, BUT must use next highest power of two, and therefore can waste a lot of memory

When new is called, the amount of necessary memory is rounded up to the next highest block size 🡪 that block is allocated from the tree of free memory

When delete is called, the memory allocated is returned to the available memory in the tree

* + Binary executable: the program file itself, begins at address zero (\*not quite – there are unallocated small addresses) and takes about 10,000 addresses (40,000 bytes)
* Memory allocators
  + Local variable declarations: known size and scoped lifetime, popped off the stack (de-allocated) when ret is called
  + **alloca()**: unknown size and scoped lifetime, similar to malloc() but used on the stack, automatically de-allocates the space when **ret** is called
    - Rarely used because it is not that difficult for programmer to de-allocate the space on the heap, it is useful for only one method, and there are size limitations on the stack
  + Global, Static variable declarations: known size and unlimited lifetime, defined outside of program/subroutine and prior to the creation of any objects so that is is visible to all objects and does not take up space in any objects
  + **malloc()** and **new**: unknown size and unlimited lifetime

**void \*malloc (size\_t *size*)**

* + - * Returns an un-typed pointer to an unused location in memory of *size* bytes (if there is NOT enough space in memory, then returns NULL)

**char \*s = (char \*) malloc (sizeof (\*s) \* n)**

* + - * Casts the un-typed pointer to be a char pointer, returns a pointer to enough space in memory for n \* s\_bytes
* Computer Architecture: Disk (1Tb) 🡪 Memory (4Gb) 🡪 Cache 🡪 CPU Registers in order from slowest (largest) to fastest (smallest)
  + Processor cycle: the time it takes to execute an instruction (mov eax, ebx)
  + Memory access time (latency): time it takes to access memory
    - Page: ~1Kb of memory (contiguous bytes) that is loaded up into cache when memory is accessed
    - Caches: level 1 (64 Kb), level 2 (½ Mb), level 3 (6 Mb), content at each level is a subset of the level below
      * Cache hit: address requested is in cache
      * Cache miss: address requested is not in cache
      * Cache page size: the number of contiguous bytes moved into cache at one time
      * Temporal locality: if an item is referenced, it will tend to be referenced again soon
        + Moving a page up from memory to cache is slow, but then afterwards it is very fast
      * Spatial locality: items with addresses close to the addr. just referenced will tend to be referenced soon
        + Ex: arrays are typically on the same page (unless they are very large arrays) and therefore once the page is moved into cache, access is very quick
    - Want to work with temporal/spatial locality in order to improve program speeds
      * While dynamic memory access is slower, it is slower by a **constant** and therefore there is no effect on Big-Theta analysis 🡪 Big-Theta analysis break-down
      * getpagesize(): returns the byte size of page that is moved into cache
      * Ex: 2D arrays are stored in row major order and therefore nested for loops should be structured in row major order 🡪 a whole row will be stored on the same page (first access slow and every access after that in the same row will be quick)
        + If array is looped through in column major order, a whole new page will have to be loaded for EACH access
    - Capacity and latency trends show that we can store MORE but we cannot store it FASTER: caches and cache algorithms are increasingly important
      * Computer architects, compiler writers, operating system makers, programmers must all recognize importance of working with memory instead of against
  + Memory cycle time: time it takes to write to memory

String functions

* C-style string: just a char \* where each byte is the ASCII representation of that character, and the last byte is a binary zero
  + **string** class handles pointer access
* C-string functions
  + **int strlen (char \*s)** : returns number of chars in s
  + **char \*strcpy(char \*s1, const char \*s2)** : copies s2 to s1
    - Make sure destination (s1) has enough space (managed by **string** class)
  + **char \*strcat(char \*s1, const char \*s2)** : appends s2 to s1
    - Make sure destination (s1) has enough space (managed by **string** class)

Extra Notes

* int-- versus --int : the first means use int and then decrement it, while the second means decrement int and then use it
* Tail Recursion
  + In a typical recursive function, the recursive call must be completed in order to obtain the product 🡪 each time the recursive call occurs another activation record is pushed onto the stack
  + In tail recursion, the function has a second parameter that is the ongoing product, so that when the base case is called, the product can be returned BEFORE the completion of the recursive calls 🡪 the compiler will actually optimize the program into a for or a while loop to avoid all of the inefficient activation records
    - For some functional programming languages, all recursive functions must be written with tail recursion
* Segmentation Fault: try to access some spot in memory (a **segment** of memory) that cannot be accessed (NULL pointer)
* Float vs. Double
  + Float uses less memory
  + Double has higher precision (15 digits)
* C
* Objective C
  + No references, multiple inheritance, no namespaces
  + Has self, which is the equivalent of this in java
* Makefile
  + Macro: $(OFILES)
  + Target: what the makefile should make
  + Suffix Rule: what file to make form what other file (.o from .cpp)