Stanley Wei
CS32 Notes
Winter 2023
Smallberg

Table of Contents

Classes

Principles

- Class design
    - Not all related functions need to be member functions
    - Interface vs implementation
        - First think about interface (what a class should do)
        - Implementation - consider the design constraints (e.g. are there values for member variables that should be invalid?) for a class while implementing
            - Class-invariant - all instances of a class may share the same constraints
    - Exceptions thrown to indicate a failed constructor
        - Done since constructors cannot return values
            - For normal functions, return 1 (instead of 0) or false (instead of true)
            - Alternatively, can print an error message and exit() the function
- For non-member functions that take classes as parameters, it may be more memory-efficient to pass by reference
    - Classes are by default passed by value, no matter the (potentially very large) size of the class
- *Default arguments*: all parameters after a parameter with a default value should be given default values
- When an object is run, it runs its **constructor**; when an object is to be destroyed (e.g. when going out of scope), it runs its **destructor**
    - When a function goes out of scope, the destructor is run for all objects created in the function, from bottom to top

Class Construction

- Syntaxes for constructing an object with arguments: Class obj(*parameters*) or Class obj = *single parameter*;
- **Copy constructor** - constructor for a class that takes another [reference to an] instance of the class as a parameter (***ClassName::ClassName(const& ClassName other_instance) {}***)

    - Is run when initializing a new instance of a class with the value of another class (String newString = oldString; or String newString(oldString)) or when passing a class to/from a function [by value] (passing a class as a function parameter without passing by reference, or returning an instance of a class from a function)
    - Will be auto-generated by the compiler if not provided
        - Can alternately be implemented via initializing an object and filling its member variables via the assignment operator
        - Auto-generated copy constructor usually insufficient in the case that the class contains dynamically-allocated variables, since those are default copied by reference (not by value)

Class Initialization

- Members can be initialized by a constructor, but outside of the curly braces of the constructor, via ***initializer lists***
    - Syntax: Class::Class(*parameters*) **: member1(*value*), member2(*value*), ..., memberN(*value*)** { *constructor body* }
        - The values can utilize the input parameters
        - If the member variable is an object, the values are replaced by the parameters for the object constructor
- When running a constructor, the program will construct the data members of the class (step 2) before executing the body of the constructor (step 3)
    - Step 2: If the initializer list provides a value for a data member, the initializer list's value will be used; if not, the program will run the default constructors of the data members (leaving built-in types uninitialized)
        - If the intended value of a data member is known when implementing the constructor, setting it in the initializer list (rather than overwriting whatever was created by the default constructor) would be more efficient
- Attempting to run the default constructor of a class without a default constructor will result in a compilation error
    - Any member variable of a class, where that variable is a class without a default constructor, must be explicitly initialized in the initializer list

- **Assignment operator** (***ClassName& ClassName::operator=(const& ClassName other_instance)***) - function run when assigning an already-initialized object the values of another instance of the class with the "=" operator (i.e. given ClassName instance;, the assignment operator is run when the command instance = other_instance; is given)
    - Copy constructor is run when creating a new instance of the class and assigning it the value of a given object; assignment operator is run assigning an already-existing instance of the class, the value of a given object
        - Object = Other_Object = (Object.operator=(Other_Object))
    - By convention, returns value held by the item on left-hand side of the equals sign
    - Case to consider - setting a variable equal to itself
- Copy constructor will be auto-generated by the compiler if not provided
    - Auto-generated assignment operator usually insufficient if the class contains dynamically-allocated variables (similar to copy constructor)

C++ Classes

- Members of a class can see the private members of any other object of that class (e.g. a member function of a class, passed in an object of that class, can look at said object's private member variables despite them being private)
- Class member functions must be explicitly declared as const to be used on const instances of the class, even if the function explicitly does not modify the object
    - The compiler does not examine the implementation of the function to determine const vs not const
- A compiler does not need to know anything about a class to declare [in another class' declaration] a data member that is a pointer to an object of that class
    - Only need to tell the program that the keyword (name of the class) indicates a class, e.g. with a class prototype
    - The compiler does not need to know how large the class is (i.e. see the class' declaration)
- To declare a data member that is an actual instance of an object, the compiler will actually need to see the class' declaration to allocate enough size to hold the data member
    - *Circular dependencies* are not allowed (e.g. object A has a member object B (an instance, not a pointer) has a member object A etc.)

Inheritance

- A class can be declared as a **subclass (derived class)** of a **superclass (base class)**
    - **Inheritance**: Instances of a derived class automatically include ("inherit") all of the member variables and functions of the base class
        - Variables/functions do not need to be mentioned or declared in the derived class to be inherited from the base class
        - All derived classes automatically inherit the base class's implementations of inherited member functions, unless explicitly overridden
            - All derived classes can only access public/protected member variables and functions of the base class
    - Derived classes can still declare their own member variables and functions, which will not be inherited by the base class or the other derived classes
    - Instances of [or pointers to instances of] derived classes can be automatically converted to instances of [or pointers to instances of] the base class (e.g. a variable that holds instances of the base class can still hold instances of the derived class)
        - Similarly, an instance of a derived class can still be dynamically allocated using a pointer to the base class, i.e. "BaseClass* ptr = new DerivedClass();"
        - Variables holding instances of the derived class cannot hold instances of only the base class
    - Syntax: *class DerivedClass : public BaseClass {};*
- **Polymorphism** ("many forms"): a derived class, in a sense, has multiple types (the derived class itself, and any base classes)
    - Similarly, the same function can take different forms [implementations] between different classes (i.e. via function overriding)
- In the case that a member function declared in a base class is implemented by both a derived class and the base class, the implementation for the derived class will ***override*** the implementation for the base class (in the context of that derived class only)
    - (i.e. the derived class's implementation when be used, when that function is called by an instance of the derived class specifically)
    - The derived class can still call the base class's implementation of the function

        - Syntax: *[this->]BaseClass::FunctionName();*
    - Note: member variables can also be overridden (and the member variables of the base class then referred back to)
        - Any references to the variable in the base class and its member functions will refer to the base class's variable; any references to the variable in the derived class and its member functions (including virtual functions) will refer to the derived class's variable
- A base class must provide an implementation of all of its member functions (to properly compile), unless the function is explicitly declared as a **pure virtual function**
    - In the case of a pure virtual function, the base class is exempt from providing an implementation for the function; all derived classes, however, must provide their own implementations [for the pure virtual function]
        - Syntax: *ReturnType FunctionName()* ***= 0;*** (in the base class)
- If a class is declared with at least one pure virtual function, it becomes an **abstract class** and can no longer be **instantiated** (have instances of that class created)
    - Derived classes of abstract base classes can still be instantiated, provided they provide a valid implementation for any inherited pure virtual functions
        - A derived class of an abstract base class can itself become abstract if it does not provide an implementation for all inherited pure virtual functions
    - Pointers to an abstract class can still be made
    - Destructors are the only function that must be implemented (cannot be pure virtual) for an abstract class (either by the compiler or by the user)
        - Should be declared as virtual anyway
- In the case of overridden functions, the specific function implementation used will be determined either at compile time (*static binding*) or during runtime (*dynamic binding*)
    - Most languages use dynamic binding by default, and static binding is opt-in; in C++, static binding is the default, and dynamic binding is the opt-in
        - The only exception (for C++) is destructors, which automatically follow dynamic binding (are automatically virtual)
    - The keyword ***virtual*** must be used in the declaration of a member function in a base class for that function to use dynamic binding when called
        - Specifying "virtual" in derived classes is optional, but more readable
        - Attempting to override a non-virtual function will have no effect

        - (Syntax: *virtual* ReturnType FunctionName();)
- In memory, every instance of a derived class is essentially constructed with an embedded instance of the base class inside of it
    - The embedded instance is the first item constructed during the construction of the derived class, even before the construction of non-inherited data members
        - Inherited data members in a derived class can be initialized by calling a constructor for the base class in the initializer list of the constructor of the derived class
            - Otherwise, the default constructor for the base class will be called
        - Conversely, it is the last item destroyed during the destruction of a derived class, after the destruction of non-inherited data members
    - A pointer to an instance of the base class, created in reference to an instance of a derived class, will be a pointer to the embedded instance of the base class in the instance of the derived class
    - When determining what function implementation a class uses, the compiler effectively draws a hash table for each class that maps each member function to the function implementation that should be used
        - Every instance of any class is constructed with a pointer to that class's (class-specific) hash table
            - In the case of an instance of a derived class, that pointer is held by the embedded base class instance in the derived class instance
        - The hash tables of derived classes will point to the base class's implementation of inherited member functions
            - In the case of overridden functions, the overriding derived class's hash table (and only that class's hash table) is changed to map to that derived class's implementation, leaving the base class and the other derived classes' hash tables alone
- **Construction/destruction order:**
    - Construction: Base class[es] -> data members (in the order of declaration - either via default constructor or initializer list) -> constructor body
    - Destruction: Destructor body -> data members (in reverse order of declaration) -> base class[es]

Templates

- Motivation - There are many cases in which we want a certain function, to operate in similar ways across different item types (e.g. min() for ints vs for doubles looks similar, but needs to return different item types)
- **Templates** allow functions and classes to be written with dynamically generated variable types (though the actual operations called are constant)
    - Multiple typename arguments can be used in a template
        - *Note: May result in strange behavior when considering return types*
- **Templated functions:**
    - Can take consts and references to template types as parameters
    - At least one parameter of a template function must use the template type
    - Syntax: template<typename T> + a normal function definition (treating T as any other variable type)
        - *Ex: template<typename T> void foo(const T& a, int b);*
- **Templated classes:**
    - If implementing functions of a template class outside of the class, each individual function implementation must be explicitly redeclared as template
        - *Ex: template <typename T> void foo<int>::bar() {}*
    - Syntax: template <typename T> + class definition
        - Initialization: ClassName<type> obj;
        - *Case: Implementing a function returning an instance of class Bar declared inside templated class Foo:*
            - *Ex: template <typename T> typename Foo<T>::Bar* Foo<T>::Foo2() {}*
- ***Template argument deduction*** - The compiler will match all calls and references to a templated function/class with a matching function/class (with the appropriate types) iff the newly generated function/class compiles
    - The specific item type used by a template is determined dynamically by the compiler, depending on the types of the arguments of the function call
    - If a call to a template does not exactly match the template (is inconsistent, e.g. two types where one is expected), the function will not compile
        - Function will not compile even if an automatic conversion exists
        - Exception: Pointers can be passed in in place of arrays

    - If any functions/operators used inside the template do not compile in the context of the provided variable type, the function will not compile
- Notes on templates:
    - Calls to an ordinary [non-template] function will automatically override the template, if applicable
    - Good practice entails fully implementing all member functions of any templated classes in the header file
    - When initializing objects of unknown type - can use default constructors to initialize temporary objects (e.g. double temp = double();)
        - Works even for built-in types

Data Structures

Overview

- Using the appropriate data structure can drastically affect the runtime and performance of various operations
- Choice of data structure usually depends on constraints of the system + types of operations expected to be performed
    - Ex: Linked lists require more memory and are slower to traverse or access by index than vectors, but are cheaper to insert into and sort [swap elements]
- Data structures:
    - *Vectors* (<vector>) store their elements in contiguous chunks of memory, similar to a dynamically allocated array
        - Have O(1) indexing and allow for direct access + sequential traversal, since the memory addresses of all elements are known
        - Are cheap to insert at the end, slow to modify in the middle
        - Have worst-case O(N) deletion [deleting the beginning of a vector], since all subsequent elements would need to be moved forward
    - *Linked lists* (<list>) store their elements in sequentially-ordered nodes, where each node contains the value of the element and a pointer to the next node in the list
        - Do not allow for direct access, since nodes are not stored in contiguous memory, but are cheap to insert into the middle of/swap elements
    - *Binary search trees* (<map>, <set>/<multiset>) store data in the form of a binary tree of nodes storing the value of the element and pointers to left and right children
        - Allow for O(log N) searching when the tree is properly balanced
        - Are not suitable for sorting once the tree is in place
        - STL requires operator< defined
    - *Hash tables* (<unordered_set>, <unordered_map>) map keys to indices in an array
        - Allow for average case O(1) insertion when properly hashed
        - Cannot be iterated/traversed in sequential order [without using an auxiliary data structure]
- Special data structures:

    - *Stacks* (<stack>) and *queues* (<queue>) allow for insertion/deletion of elements from one end/opposite ends only, respectively
    - *[Max] Heaps* (<priority_queue>) are binary trees organized such that the element of largest value is always kept at the root [top] of the tree
- Misc notes:
    - Many problems can be modeled with **graphs** (e.g. topological sort)

Data Structure	Indexing	Searching [Sorted]	Direct Access	Sequential Iteration	Insertion	Deletion
Vector	O(1)	O(log N)	Yes	Yes	O(N) [wc]	O(N) [wc]
Linked List	O(N)	O(1) [head/tail] O(N) [middle]	No	Yes	O(1) + O(N)	O(1) + O(N)
Binary Search Tree	O(log N)	O(log N)	No	Yes	O(log N)	O(1) + O(log N)
Hash Table	O(1)	N/A	Yes*	No	O(1)	O(N) [wc]
Stack/Q	O(1)	N/A	N/A	N/A	O(1)	O(1)

Linked Lists

- *Motivation*: Keeping elements ordered in an array while inserting new elements can be difficult and costly, depending on the size of the array
    - Inserting an item near the beginning of the array may require reshuffling nearly the entire array to preserve the order
    - Insertion into the middle of an array can be costly because variables must be stored in consecutive order in memory - move one, move all

- **Linked lists** store variables in memory in the form of **nodes**, where each node contains a value (of an element) and a pointer to the next node in the list
    - Does not require linked list elements to be stored in consecutive (or even adjacent) memory addresses, only that they point to the right locations
        - Allows for insertion into or deletion at any index in a linked list at a relatively low cost - only need to modify the [pointer held by the] element before the index, rather than the remainder of the array
    - Beginning of a linked list is indicated by a pointer (the ***head***) to the first node
        - End of the list is marked by a variable holding a nullptr (the ***tail***)
- *Special linked lists:*
    - In ***doubly-linked lists***, each node also contains a pointer to the previous node in the list - requires more memory, but makes iterating backwards much faster
        - Overall structure includes a pointer to the last node (a *tail*)
    - In ***circularly-linked lists***, the node at the end of the list points back to the beginning (rather than pointing to nullptr)
        - In circular doubly-linked lists, the head node's previous node pointer points to the end of the list
        - Circularly-linked lists may also have a *dummy node* - a node with no value that is placed at the head of the linked list
            - Guarantees the presence of at least one node in the list when performing linked list operations
- Linked lists can be iterated through via a for loop, continuously following pointers between nodes until a nullptr is encountered
    - May require checking to make sure all pointers are properly initialized

- *Notable special cases for linked list logic:*
    - Modifying the beginning of a linked list
    - Modifying the end of a linked list
    - Iterating through an empty linked list
        - Can add a dummy node with no value to every linked list to ensure the presence of at least one value
    - Iterating through a one-element linked list
- <***list***> - C++ standard library implementation of a linked list
    - C++ standard does not specify a list needs to be doubly-linked, but it is typically implemented as such for performance
        - Also does not specify the presence of dummy nodes, etc. - only provides the interface for the list

Stacks and Queues

- **Stack** - a collection of data that only allows for the modification (inserting to/deleting from) the end (**top**) of the stack (are *LIFO:* last in, first out)
    - Expressly forbids modifying any part of the stack besides the top (active end)
    - Is more akin to a limited list of [high-level] operations, than to a data structure
        - Provides another level of abstraction + certain guarantees/forbiddances
            - Not strictly meant to boost efficiency or versatility
        - Can be implemented with either linked lists or arrays
        - Can be used with any data type
- ***Allowed stack operations:***
    - Creating a[n empty] stack
    - Pushing [adding] to the top of a stack
    - Popping [removing] from the top of a stack
    - Looking at the top of the stack + checking if a stack is empty
    - Optional (non-universal) operations:
        - Looking at any item in the stack (not just the top)
        - Checking the size of a stack
- **Queue** - a collection of data that only allows for the insertion of elements at one end, and the deletion of elements from the other (are *FIFO*: first in, first out)
    - Has two active ends - the head and the tail, or the front and the back
- ***Allowed queue operations:***
    - Creating a[n empty] queue
    - Enqueuing (adding) an item [to the back]
    - Dequeuing (removing) an item [from the front]
    - Looking at the front item in the queue + checking if a queue is empty
    - Optional (non-universal) operations:
        - Looking at the back item in the queue
        - Looking at any item in the queue (not just the front/back)
        - Checking the size of a queue
- *Consideration:* if implementing a queue with an array, dequeuing items will result in the start of the queue slowly becoming offset over time

- Can be fixed by offsetting [in the opposite direction] the elements of the queue upon dequeuing an element
- (More optimally) Can be mitigated by using a ***"circular array"*** - a queue implemented in a way that the last element in the array effectively "wraps around" to the first element
    - i.e. the element after the last element is the first element (as opposed to the last element needing to be the last element)
    - Eliminates the need to repeatedly move the elements of the array
    - Also known as a ring buffer/circular buffer

Vectors

- **Vectors** - C++ type representing a dynamically allocated array
    - Does not automatically initialize every element in the [allocated capacity for the] vector (i.e. leaves elements uninitialized unless told otherwise)
        - Is true for both built-in and custom types
        - Vector stores current size and current allocated capacity separately
    - Is implemented with an array - accessing any value is equally efficient

Trees

- A ***tree*** is a collection of **nodes**, connected by **edges**, branching out from a **root node**
    - A **path** between two nodes is a sequence of edges connecting the two nodes
    - There exists [by the definition of a tree] a *unique* path between any node in the tree, and the root node
        - The *depth* of a node is defined as the length of the path between the node and the root node
            - The height of a tree is the depth of the deepest node
        - Forbids cycles, e.g.
    - A tree is typically described in terms of parent/child relationships
    - A node without a child is called a **leaf node** (as opposed to non-leaf/**interior nodes**)
- A **binary tree** is a tree that is either empty or has two binary trees as children
    - All trees can be represented by a binary tree (e.g. each node has two pointers - one to children [current depth + 1], one to siblings [current depth])
        - Can represent other trees, without each node requiring a dynamic container to represent its children
    - Traversal types (recursive):
        - *Preorder:* Print root, left subtree, right subtree
        - *Inorder:* Print left subtree, root, right subtree
            - Prints elements of a binary search tree in sorted order
        - *Postorder:* Print left subtree, right subtree, root
- A **binary search tree (BST)** is a binary tree in which, for each node with 2 children, all nodes in the left subtree have value <= the value of the current node and all nodes in the right subtree value >= the value of the current node
    - Can be searched and modified (iteration/deletion/accessing) very quickly, if properly balanced
        - Search algorithm mimics the structure of binary search [O(logN)]
    - Balanced BSTs are cheaper to insert into than a sorted vector, and easier to search than a linked list - useful in the case that both operations are occurring regularly
        - In instances when the structure is first built, and then searched [without continuing insertions], building a sorted vector may be more efficient than a binary search tree

- STL <***set***>, <***multiset***>, <***map***> implemented using binary search trees
    - Require a less-than operator (operator<) defined for the type of element contained
- Insertion, deletion operations: Wikipedia
- *(Note) Example BST balancing algorithms:*
    - AVL tree (algorithm to balance a tree) - each subtree can only differ in depth by 1
    - 2-3 tree - some nodes have two values and three children (where the children are split between the values)
        - Makes tree depth smaller, but accessing/inserting/etc more expensive
        - 2-3-4 trees (represented by a binary tree - red-black tree)

Hash Tables

- **Hash table** - data structure that stores its contents in an array, then uses a hash function to associate any given input (not necessarily numerical) to an index in the array
- Each index in the array represents either a single item or a certain subset of the overall set of items to be stored, and the element at said index is a smaller container ["**bucket**"] holding said subset (such that every item is ultimately stored at some index)
    - *Collisions* occur when a single index is meant to represent multiple items - changes from storing a single item to storing a container
    - Allows for holding a large range of items and fast indexing without needing to allocate space for an entire large array where many indices would be left unfilled
        - Is more efficient than a binary search tree for most purposes (i.e. not extremely large databases)
        - Containers usually implemented with a vector/array
    - Each smaller container should be small enough to not need sorting
- ***Load factor*** - the [average] size of each bucket (num items / num buckets)
    - Typically kept around 0.7-1.0: should take care to make sure load factor is not too low, i.e. there are not many empty indices, but also not too high (reduce collisions)
- A ***hash function*** maps a key (be it an int, a string, etc.) to an integer index
    - Mapping strings to ints can be difficult - typically involve some arithmetic operations involving char encodings
        - Fowler-No11-vo
        - Parentheses can be overloaded as an operator
    - Should produce uniformly distributed values, be cheap to compute
        - Needs to be deterministic
- Big O for insertion/deletion/searching in a fixed-size [i.e. bucket #] hash table is O(N), albeit with a low constant of proportionality (relative to O(logN) binary search trees)
    - Increasing the # of buckets decreases the constant of proportionality (at the cost of larger memory expenditure)
- Hash tables can specify a fixed maximum load factor at which point additional buckets are added dynamically for better time complexity - results in average case O(1) time complexity for insertion/searching
    - Does require **rehashing** all elements into new set of buckets [O(N)], but time

between rehashes also grows linearly as a function of N, hence average constant time complexity

    - *Incremental rehashing* - rehashing a certain number of elements every insertion [progressively], rather than rehashing all elements at once
        - Results in bounded constant time insertion - no single insertion exceeds a certain amount of time
            - Rehashing, rather than being a single O(N) operation, becomes an O(1) operation performed N times
- STL maps (<***map***>) associate keys and values (pairs <key, value>)
    - Iterator visits keys in increasing order
    - Sorts keys in increasing order [key type requires less than operators defined]
    - Implemented in binary search trees
        - Insertion/deletion/lookup runs in O(log N)
- STL unordered sets, multisets, maps, multimaps (unordered_set/map/etc.)
    - Implemented in hash tables
        - Cannot be traversed in order, but insertion/deletion/lookup run O(1)

Heaps

- *Motivation:* FIFO (i.e. a queue) may not always be desired behavior - we may want to order elements by priority instead
- ***Complete binary tree*** - a binary tree completely filled at every level, except for at the lowest level
    - Every node not in the lowest level/second-lowest has either no or two children; every node in the lowest level has no children
        - Every subtree in a complete binary tree is also a complete binary tree
- A [max] **heap** (data structure) is a complete binary tree in which the value at every node is greater than all values contained in its subtrees
    - May need rebalancing for optimal performance (similar to binary search trees)
    - Can be implemented using arrays
        - ***Heapsort*** - sorting algorithm that converts an array to a heap to sort
- ***Heap operations:***
    - *Insertion* - place the new element at the lowest level, then swap up with parents (as needed)
    - *Removal* - remove the top element, move the right-most element on the lowest level to the root [top], and then swap down until it is correctly placed
- A *priority queue* is a queue where elements are ordered by priority
    - The element with the highest priority is the element returned, regardless of when it was inserted into the queue
    - Can be easily implemented with a heap

STL Types

- **Standard Template Library (STL)** - C++ standard libraries for containers, algorithms operating on sequences of items
    - Includes <*stack*>, <*queue*>, <*vector*>, <*list*>, <*map*>
    - In terms of algorithms, only includes algorithms that would be efficient (e.g. push_back may be provided, but not push_front)
- <***vector***> interface:
    - *Initialization*: vector<type> v;
        - vector<type> v(*initialSize*);
        - vector<type> v(*initialSize*, *initialValue*);
            - e.g.: vector<char> v(3 'a'); -> { 'a', 'a', 'a' }
        - vector <type> v(*ptrToBeginningOfArray*, *ptrAtEndOfRangeInArray*);
            - e.g.: int a[5] = { 10, 20, 30, 40, 50 }; vector<int> v(a, a+5);
            - Also works with pointers to lists
        - Copy constructor also defined; all constructors copy by value
    - *Insertion* [to the end]: v.push_back(value)
    - *Deletion* [from the end]: v.pop_back()
    - *Accessing elements*: v.at(), v[i]
        - Values - v.front(), v.back() (for non-empty vectors)
        - Pointers - v.begin(), v.end()
            - v.end() points one after the last entry in the vector
        - .at() will throw out-of-range exceptions for invalid indices; [i] will simply result in undefined behavior
        - Adding elements beyond the vector's capacity with [i] is undefined behavior
    - *Size*: v.size()
- <***list***> interface
    - *Initialization*: list<*type*> li;
        - list<*type*> li(*initialSize*)
        - list<type> li(*initialSize*, *initialValue)*
    - *Insertion*: li.push_front(*value*), li.push_back(*value*)
    - *Deletion*: li.pop_front(), li.pop_back()
    - *Accessing elements*:

        - No [i] operator is defined for lists
        - Values: li.front(), li.back()
        - References (pointers): li.begin(), li.end()
            - If the list is empty, li.begin() == li.end()
- <***map***> interface
    - *Initialization*: map<*key type*, *value type*> m;
        - Requires operator< defined for key type; also requires either operator= for a third function [in the template brackets] for testing equality
    - *Modifying/accessing*: m[*key*], m.erase(*key*), m.find(*key*)
    - *Properties*: m.size(), m.empty()
    - *Iterating*: m.begin(), m.end()
    - Interface is identical for <***multimap***>, except [] is not defined and .find() returns an iterator (rather than a singular value)
- <***set***> interface
    - Same constructors as vector, set.begin(), set.end()
    - set.insert(), set.erase(), set.find() set.swap(), set.clear(), set.empty(), set.size()
- <*vector/list/set/map*>::***Iterator*** class - built-in iteration wrapper
    - list<*type*>::iterator p (initialized with a pointer, e.g. li.begin())
    - *Moving p*: p++;, p−;
        - +=, -=, etc. only defined for vectors (not lists, sets, maps)
    - *Accessing the value at the iterator*: *p
    - *Inserting at p*: li.insert(p, value);
        - li.insert(p, # of values, value);
        - Returns a pointer to the newly inserted item (if relevant)
    - *Deletion at p*: li.erase(p);
        - li.erase(begin, end); erases [begin, end)
        - Returns a pointer to the item following the [now deleted] item
            - Returns li.end() if at the end
        - Turns p into a dangling pointer
- <***stack***>/<***queue***> interfaces:
    - Both libraries contain the standard operations (+ a few more):
        - *Creating*: stack<data type> s, queue<data type> q
        - *Pushing/Enqueuing*: stack.push(item), queue.push(item)

        - *Popping/Dequeuing*: stack.pop(), queue.pop()
        - *Looking at the top/front*: stack.top(), queue.front()
            - *Looking at the back*: queue.back()
            - Neither the <stack> library nor the <queue> library permits looking at other items in the stack/queue
        - *Checking if empty*: stack.empty(), queue.empty()
        - *Checking size*: stack.size(), queue.size()
    - The <***stack***> library does not check for a stack being empty when performing operations [for performance reasons] - relies on the user to not conduct forbidden operations on an empty stack
    - Priority queue similar to regular queue - require operator< defined
- **Algorithms** (<***algorithm***>) - library containing various array/vector/list functions
    - *Finding items in a sequence*: find(*startPtr*, *endPtr*, *valToFind*);
    - *Counting occurrences in a sequence*: count(*startPtr*, *endPtr*, *valToCount*)
    - *Reversing a sequence*: reverse(*startPtr*, *endPtr*)
    - *Sorting a sequence*: sort(*startPtr*, *endPtr*)

- *Note:* Functions can be passed as function parameters, notably to <***algorithm***> functions
    - Calling a function - Function name + parentheses for arguments
    - Pointers to functions - pointers can be generated to functions
        - Specified for specific functions (e.g. “pointer to a function that takes as parameters a double and an int and returns an int”)
        - p = functionName; [no parentheses] -> p(*parameters*), (*p)(*parameters*)
        - Can generate pointers to functions, arrays to pointers to functions, pointers to functions as function parameters, etc.
    - Function pointers can also be passed via templates
    - Essentially passes a pointer to the code for the function (can then be called)
    - Can be used for <algorithm> functions like find (for valToFind)
        - std::sort - Can also add a function *bool Foo(const Type& bar1*, *const Type& bar2)* as a third parameter (uses operator< otherwise)

Algorithms

Recursion

- *Motivation:* when writing solutions for large problems, it may be more efficient to break the problem down into smaller, more easily-solvable parts, which can then be combined to form a solution for the larger problem
- ***Recursion*** is the process of solving a larger problem by breaking it down into (and solving) smaller versions of itself
    - Smaller problems can themselves be broken down - this process can continue until the problem is small enough to solve directly
    - Resembles mathematical induction in design - recursion solves a problem P(n), by proceeding from P(1)
- In a programming context, a **recursive function** is a function that operates in accordance with the idea of recursion
    - A recursive function has two cases, depending on whether the problem has been sufficiently reduced in size:
        - **Recursive case** - The problem is too large, so the function will make *recursive calls* to smaller versions of itself, combine the outputs of those calls, and return
        - **Base case** - The problem has been sufficiently reduced, so the task is computed directly and the output subsequently returned
    - Each recursive call by a recursive function must strictly be for a "smaller" problem (i.e. a recursive call must always move the problem toward the base case)
        - If a recursive function fails to reach the base case, it will become continue calling itself until the program runs out of memory
- Recursion can be used to simplify certain programs, or as an alternative to iterative loops

Algorithm Analysis

- Changing the approach (algorithm) used for a task, can significantly affect the time/number of computations needed to complete the task
    - Relative algorithm performance (in the case of algorithms on sequences of elements) can also vary depending on the number of elements in the sequence
        - e.g. the fastest algorithm on a 100-element sequence may be the slowest algorithm when given a 10000-element sequence
- Algorithm runtime is usually classified based on the relationship between the number of elements and runtime ("growth pattern") using **big O notation**
    - *Big O ex.:* if the number of elements is multiplied 10x, what the corresponding increase is in terms of runtime (10x, $(10x)^2$ = 100x, etc.)
        - Big O may be assumed to be worst case if not specified
    - Big O behavior only classified based on the greatest polynomial, without coefficients (e.g. $3n^2 + 2n$ becomes a big O of $O(n^2)$)
    - *Notation:*
        - "Constant order" (O(1)): runtime is constant (10x elements -> 1x runtime)
        - Linear time/O(n) ("Order N"): 10x elements -> 10x runtime
        - Quadratic time/$O(n^2)$ ("Order N^2"): 10x -> 100x
        - Logarithmic time/O log N ("Order log N"): 10x -> log (10)x
            - Any form of log (i.e. any base) expressed as log N
        - Exponential time/$O(k^n)$ [constant k]: 10x -> (e.g. k^{10})
        - "Order N^S^2" (multiple dimensions)
- Definition: A function f(N) is O(g(N)) if there exists a value N_0 and [constant] k such that for all N >= N_0, |f(N)| <= k*g(N)
    - i.e. an algorithm is measured based on its performance as N -> infinity (classified for large sizes of N)
- Given if statements, the costliest branch is usually taken

Sorting

- **Selection sort** - continuously swapping the largest element in the unsorted portion of the array, with the last element in the unsorted portion of the array
    - Number of comparisons constant for best/average/worst cases
- **Bubble sort** - iterating through a list, swapping an element with the element behind it if the first element is larger than the second
    - Compare 1 and 2 (swap if needed) -> compare 2 and 3 (swap if needed) -> … -> (end of array reached) if >0 swaps made, repeat
- **Insertion sort** - Dividing an array into sorted and unsorted subarrays, and then continuously inserting elements from the unsorted subarray into the sorted array [in sorted order] until the entire array is sorted
    - First element sorted -> "include" second element -> sort first two elements -> "include" third element -> sort first three elements…
- **Merge sort** - splitting an array into two subarrays, sort each of those subarrays (directly or by dividing again), merge the [now sorted] arrays
    - *O(NlogN) Proof: T(n) [time to sort n items] = 2 T(n/2) [sorting subarrays] + O(n) [merging]*
- **Quicksort** - Partitioning an array into two subarrays with elements greater (subarray 1) or less (subarray 2) than a certain item [a pivot, e.g. the first item], and then sorting those subarrays (e.g. by dividing again)
    - Can use quicksort to partition a large array down to small arrays, then use insertion sort (O(n)) on the small arrays for efficiency
    - Slowed for mostly/already sorted data [when picking 1st item as pivot]
- **Heapsort**- converting an existing array into a heap, then using that heap to construct a sorted array
    - Convert to heap -> move root element to end -> convert all elements but end to heap -> move root element to 2nd to end -> etc.
        - Conversion works bottom-up - slowly builds up lower levels of a heap, until the root is reached
- Definitions:
    - A sorting algorithm is said to be *stable* if, given an unsorted array containing two elements a1 and a2 of equal value where a1 comes before a2 in the array, running

the sorting algorithm on the array will produce an output array where a1 is always before a2

- A sorting algorithm is said to be *in-place* if it operates without needing to allocate additional arrays in memory

Performance Charts

Algorithm	Best Case	Average Case	Worst Case
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Bubble Sort	O(N) *(Already sorted)*	$O(N^2)$	$O(N^2)$ *(Sorted backwards)*
Insertion Sort	O(N) *(Already sorted)*	$O(N^2)$ (beats bubble/insert)	$O(N^2)$ *(Sorted backwards)*
Mergesort	O(N log N)	O(N log N)	O(N log N)
Quicksort	O(N log N)	O(N log N) (beats merge/heap)	$O(N^2)$ *(Bad pivots)*
Heapsort	O(N log N)	O(N log N)	O(N log N)

Characteristics

Algorithm	Stable?	Linked Lists?	In-Place?
Selection Sort	No	Yes	Yes
Bubble Sort	Yes	Yes	Yes
Insertion Sort	Yes	Yes	Yes
Mergesort	Yes	Yes	No
Quicksort	Generally no	Yes	Depends
Heapsort	Generally no	No*	Yes

Program Design

- Splitting a program up into multiple files can help speed up compilation time
    - Header field (*.h) typically provide declarations for classes and function prototypes
        - More standard C++ files (*.cpp) contain the implementations for the classes and functions
    - When given multiple files (source files), the compiler will take note of what each file needs (i.e. declarations/prototypes) and what each file defines (i.e. implementations) during translation
        - A linker (responsible for combining the different files into a single executable) will make sure everything is properly defined prior to linking
            - Will also check for and prevent multiple main functions, multiple implementations for a single function, etc.
                - Throws an error even if the implementations are identical
        - The <include> keyword essentially transplants the text of the imported file into the caller file
            - Will also import function implementations - will throw an error if a file implementing a function is imported by multiple files (resulting in multiple implementations of a single function)
                - Function implementations are typically omitted from header files, and .cpp files not <include>'d, for this reason
            - Importing a file that uses a certain namespace will cause said namespace to be used in all caller files (thus, namespace definitions are also typically omitted from header files)
            - Is necessary when creating an object as a variable, using the class' member functions
    - Good practice states that when defining a class that uses other class in a header file, the other classes should be imported in said header file (rather than requiring separate importation later)
        - **Include guards** can be used to ensure the same class is not imported multiple times
            - #ifndef **symbol name** #define **symbol name** #endif

        - Circular dependencies are not allowed (e.g. object A has a member object B (an instance, not a pointer) has a member object A etc.)
            - Can circumvent by using having members be pointers to objects, rather than actual objects
- Compilation process
    - Compiler takes source files (.cpp), outputs:
        - Machine language translation of each file
        - Storage for global objects
        - A list of global names defined and needing definition
    - Linker brings together object files + compiler outputs, produces executable
        - Requires: nothing be defined more than once, every need for a definition must be satisfied, and there must be exactly one main routine
- Limited computer resources - memory, processor time, network connections, open files/threads
    - Amount of entities accessing a database - having multiple entities modifying a database concurrently can result in conflicts, errors, overwrites, etc.
        - Having multiple pointers pointing at (and potentially modifying) the same memory address may result in unintended behavior
    - Watch for memory leaks
        - *delete [] *array name** is the operator for deleting arrays (array objects)
            - Separate operator for arrays needed due to arrays having more bookkeeping information needed than single objects
- Building classes
    - Buildings strings/arrays
        - Default string being an empty string (‘\0’) or literal null pointer - one results in a simpler implementation, the other is more memory-efficient at scale
        - Static (creating objects/arrays of objects) vs dynamic allocation (creating pointers to objects/arrays of pointers to objects with keyword *new*) [of memory] - static allocation is simpler, dynamic allocation results in less wasted memory
            - The C++ string library allocates length dynamically, starting with a relatively small array size and copying the string to a slightly-larger array every time more length is needed

                - Statically allocated arrays only need the compiler auto-generated destructor to be destroyed; dynamically allocated arrays/objects need to be specified with keyword *delete* to be destroyed
                - Statically-allocated arrays must have size known at compile time; dynamically-allocated arrays only need to have size known when initialized, but cannot have their size modified
                - Syntax:
                    - Static allocation: Object myObject(*params*);
                        - Object arr[*size*] = {};
                    - Dynamic allocation: Object* myObj = new Object(*params*);
                        - Object* arr = new Object[*size*]; creates an array of objects based on variable size
                            - Object** arr = new Object*[*size*]; creates an array of pointers to objects
        - Properties of an object (e.g. string length) being stored in memory (as a data member - can be accessed repeatedly without needing to be recomputed) or computed on an as-needed basis (results in a smaller memory footprint for the object [1 less data member])
        - Merging similar/overloaded functions together or splitting functionality off into helper functions can simplify a class (and reduce memory usage)
- Shallow copies vs deep copies
    - A **shallow copy** (of an object) simply copies all values contained in the other object; any pointers in the new object will simply point to the same address as pointers in the original object
    - A **deep copy** copies all values contained in the other object, but any pointers to variables/objects, will have the objects themselves copied (such that the new object's pointers will point to a different location than the original)
- When copying a class (e.g. when passing by value), if a specific copy constructor is not specified for that class, the compiler will auto-generate a copy constructor
    - Will copy members of built-in types by value and copy members of class types via their copy constructor
        - Will copy arrays element-by-element, based on the type of the array as per the above protocol

- Aliasing - having two or more ways of referring to an object

Misc

Memory

- Variables and arrays must be stored in contiguous regions in memory - cannot be broken up or fragmented
    - Can result in memory becoming fragmented over time as variables are declared and deleted - a costly solution is to run an algorithm to optimize memory allocation every so often
- Memory allocation - at the beginning of a program, the OS will allocate a certain amount of memory for the program to use, which the program can then allocate to its own individual components as needed
    - If memory runs out, the program must request more memory from the OS

Arithmetic Operator Notation

- Prefix notation: the operator is stated before the operands (e.g. f(x, y, z))
- Infix notation: the operator is stated between the operands (e.g. x + y)
- Postfix notation: the operator is stated after the operands (e.g. “x y +”)