**CS 138 Final Review**

**Background:**

* **OS & Shells**
* Unix command line:
  + A Unix bash command:
    - $ cmd –opt1 –opt2 arg1 –opt3 arg2 arg3
  + Alias: e.g. rm → rm –iv
  + $ which cmd / $ type cmd
    - Is cmd an alias, built-in, an executable, etc.
  + $ man cmd
    - Prints information about command
  + $ ls
    - Lists contents of directory
    - -l gives long listing (shows file permissions)
  + $ mkdir dir-name
    - Creates new directory
  + $ cp src-file tgt-file / $ cp src-file tgt-dir
    - Copies file in current or different directory
  + $ mv src-file tgt-file / $ mv src-file tgt-dir
    - Renames file in current directory or moves it to different directory
  + $ rm file-or-dirlist
    - Delete files
  + $ cat file
    - Prints file in one continuous stream
  + $ less file
    - Prints file one page at a time
* File permissions
  + Access levels: user, group, other
  + Permission: read, write, execute
  + - --- --- --- → file/directory, user permissions, group permissions, other permissions
    - E.g. - rwx r-x ---
  + $ chgrp grp-name file-or-dir
    - Change group name associated with file(s)
  + $ chmod mode-list file-or-dir
    - Change permissions of file(s)
    - mode-list = level operator permission
    - E.g. $ chmod u-r g+w o-x file

**Basics:**

* **Strings** – #include<string>
  + + → concatenation
    - String literals are char\* and cannot be concatenated (cast to string)
  + ==, <=, >= → lexicographical comparison
  + s.length() → length
  + s.substr(i, n) → n chars startin at i
  + s.c\_str() → returns char\* of string
  + s.find(str, i) → 1st index of str in s starting at i
  + s.clear() → empty string
* **Input/Output**
* #include<iostream>
  + cin >> s → read token
    - while (cin >> nextToken) { … }
  + getline (cin, s); → read line of input
  + cout/cerr << s << endl; → output
* #include<fstream>
  + ifstream input;

input.open (“inputFile.txt”);

while (input >> s) { … }

input.close();

* + ofstream output;

output.open (“outputFile.txt”);

output << s << endl;

output.close();

* **Memory**
* Run-time stack
  + Local variables & parameters
  + Direct instantiation:
  + someStruct s;

s.field; s.method();

* + Instance destroyed at the end of defining scope
* Heap/freestore
  + Access by pointers only
  + Allocate with new, deallocate with delete or delete[] (contiguous)
  + Dynamic instantiation:
  + someStruct \*s = new someStruct;

s->field; s->method();

delete s;

* **Arrays/Vectors**
* Both contiguous memory
* Array
  + string s[N];
    - Static (on stack); array bound must be a constant
  + string \*s = new string[n]; delete [] s;
    - Dynamic (on heap); array bound can be a variable
* Vector – #include<vector>
  + vector<int> v; vector<int> v (10); → both dynamic (on heap)
  + Size ≤ capacity
  + s.at(n) vs. s[n] → checked access vs. unchecked access

**Concepts:**

* **Abstract data types**
* Contains data that can only be accessed via a set of operations
* Abstract specifications are the same regardless of implementation
* Interface – enforces limited access to internal details
* Adapter – an interface that provides exactly what the client needs and no more
  + Prevents client from depending on features they shouldn’t use
  + Protects them from changes in underlying implementation
* Information hiding – separating information from implementation
* **Reference parameters**
* Call by value – C/Java
  + Copy of parameter is created on the call stack
  + Changes made to parameter values do not propagate back to caller
* Call by reference – C++
  + The reference param is an alias for the variable in calling environment
  + Changes made do propagate back to caller
  + Const reference param is not allowed to be changed
  + void function1 (Object x);
    - Copy of x created on call stack, changes to x do not propagate back
  + void function2 (const Object x);
    - Copy of x created on call stack, x may not be changed
  + void function3 (Object & x);
    - x refers to variable in calling environment, changes propagate back
  + void function4 (const Object & x);
    - x refers to variable in calling environment, may not be changed
* **Recursion**
* Scope – extent to which an identifier is visible
* E.g. factorial
  + Base case(s) – if (n <= 1) return 1;
  + Reduction operator – factorial(n-1);
  + Composition operator – return n \* factorial(n-1);

**Data Structures:**

* **Stack**
  + Last-in-first-out
  + void initStack(Stack & s);
  + bool isEmpty(const Stack & s);
  + void push(Stack & s, string val);
    - Adds new element at the front
  + void pop(Stack & s);
    - Removes first element
  + string peek(const Stack & s);
  + void nuke(Stack & s);
  + Stack as a linked list:
  + struct Node {

string val;

Node\* next;

};

* + typedef Node\* Stack;
  + Stack as a vector:
    - v.push\_back(e); v.pop\_back(); v.back();
* **Queue**
  + First-in-first-out
  + void initQueue(Queue & q);
  + bool isEmpty(const Queue & q);
  + void enter(Queue & q, string val);
    - Adds new element at the end
  + void leave(Queue & q);
    - Removes first element
  + string first(const Queue & q);
  + void nuke(Queue & q);
  + Queue as a vector: bad
  + Queue as a linked list:
  + struct Node {

string val;

Node\* next;

};

* + - struct Queue {

Node\* first;

Node\* last;

};

* **Doubly linked list**
  + Has \*prev and \*next Node pointers
* **Sorted linked list**
  + bool lookup(const SortedList & first, string val);
    - Cases: list is empty / list is not empty
  + void insert(SortedList & first, string val);
    - Cases: val is first element / val is after first element
  + void remove(SortedList & first, string val);
    - Cases: val is first element / val is after first element
* **Priority queue**
  + void enter(PQ & pq, string val, int priority);
    - Adds new element at the end of its priority
  + void leave(PQ & pq);
    - Removes first element of highest priority
  + string first(const PQ & pq);
    - Returns first element of highest priority
  + Priority queue as a list of lists:
  + struct QNode {

string val;

QNode\* next;

}

* + struct PQNode {

int priority;

QNode\* first;

QNode\* last;

PQNode\* next;

};

* + typedef \*PQNode PQ;
  + Priority queue as a heap (binary tree):
    - Heap property – value of parent >= value of children
    - Largest element is always at root node
* **Binary search tree**
  + Each node has 0 to 2 children & 1 parent
    - Root node has no parent
  + Leaf – node w/ no children; internal node – anyone else
  + Height – longest path from node to a leaf (count down)
  + Depth – path from root to node (count up)
  + BST property – value of all nodes in left/right subtree </> value of node
  + Balanced tree, degenerate tree
  + bool lookup(const BST & root, string key);
    - Recursively call lookup for left & right subtrees
  + void print(const BST & root);
    - In-order traversal – print left subtree, current node, then right subtree
  + void insert(BST & root, string key);
    - Navigate recursively until at leaf, add new element as a child of leaf
  + void delete(BST & root, string key);
    - Cases: node has no children / node has 1 child / node has 2 children
      * If node has 2 children, “replacement node” is the node w/ max value in the left subtree
  + BST:
  + struct Node {

string key;

Node\* left;

Node\* right;

}

* + typedef \*Node BST;
* **Sequence**
  + Container indexed by contiguous non-negative integers
  + void insert(element, index);
  + void append(element);
  + string at(index);
  + void remove(index);
* **Dictionary**
  + Associative container using key-value pairs; e.g. dict[key] = value
  + void add(key, value);
  + void overwrite(key, value);
  + string lookup(key);
  + void remove(key);
* **Hash table**
  + A vector of K buckets; unordered
  + Hash function: key → index
    - Must be deterministic, based on some intrinstic property of the key value
    - Good “spread”; uniformity
    - Cheap to compute
    - Supports variable range (K)
  + Collision – different keys are mapped to the same index
  + Closed hashing + linear probing – go to index given by hash function, then search linearly for empty slot/desired element
    - enum Status {EMPTY, ACTIVE, ZOMBIE};
    - struct Node {

string value;

Status status;

};

* + - typedef vector<Node> HashTable;
  + A bucket can be active, empty, or zombie
  + As N → K, operations time O(1) → O(N)
  + Open hashing + chaining – each bucket holds a pointer to a linked list; add elements to list if collision occurs
    - struct Node {

string value;

Node\* next;

};

* + - typedef vector<Node> HashTable;
* **Time complexities**
* Queue (linked list)

|  |  |  |
| --- | --- | --- |
| Enter | Leave | Search |
| O(1) | O(1) | O(n) |

* Stack (linked list)

|  |  |  |  |
| --- | --- | --- | --- |
| Push | Pop | Peek | Search |
| O(1) | O(1) | O(1) | O(n) |

* Sorted linked list

|  |  |  |
| --- | --- | --- |
| Insert | Delete | Search |
| O(n) | O(n) | O(n) |

* Sorted array

|  |  |  |
| --- | --- | --- |
| Insert | Delete | Search |
| O(n) | O(log n) | O(log n) |

* Priority queue (list of lists)

|  |  |  |
| --- | --- | --- |
| Enter | Leave | Search |
| O(k) | O(1) | O(k) + O(n/k) |

* + k = # of priorities
* Priority queue (heap)

|  |
| --- |
| Enter/leave |
| O(log n) |

* BST

|  |  |
| --- | --- |
| Lookup/insert/delete | Print |
| O(log n) | O(n) |

* Hash table

|  |
| --- |
| Lookup/insert/delete |
| O(1) amortized |

**Object-Oriented Programming**

* Procedural programming
  + Variables are passed from main() to procedures
  + Each struct defines a type of variable, with variable subparts
* Object-based programming
  + Structs/classes have variable (fields) and procedure (methods) subparts
  + Object – an instance of a class/struct
* Object-oriented programming
  + Inheritance, abstract base classes, polymorphism, generics
* **Class basics**
  + Class declaration:
  + class SomeClass {

public:

SomeClass(); //default constructor

SomeClass(string val); //method overloading

virtual ~SomeClass(); //destructor

string method();

private:

string field;

};

* + Class definitions:
  + SomeClass::SomeClass() : field(“value”) {} //initializer

SomeClass::SomeClass(string val) : field(val) {}

SomeClass::~SomeClass() {

//delete dynamically allocated stuff

}

string SomeClass::method () {

return this->field; //’this’ points to current object

}

* + Classes can be directly or dynamically instantiated (like structs)
  + Access specifiers (not the same as visibility)
    - Public – anyone can access the field/method
    - Protected – only accessed by this class and its children
    - Private – only accessed by this class
  + Instance variable – created for each object instance
    - Instance method – can access instance & static variables of that object
  + Static variable – only one for each class; stored in special memory location (not stack or heap)
    - Static method – cannot access instance variables, only static variables
* **Types of Methods**
  + Copy constructor – creates a copy of existing object (of the same class)
    - Declaration:

SomeClass(const SomeClass & c);

* + - Definition:

SomeClass::SomeClass(const SomeClass & c) : field(c.field) {}

* + - Usage:

SomeClass c(“val”);

SomeClass c1(c); //copy ctor

SomeClass c2 = c; //assignment also calls copy ctor

* + Destructor is called when the object’s scope is exited (stack) or it’s deleted (heap)
  + Const method – cannot change any subparts, cannot call non-const methods
    - Can change object pointed to by a ptr (can’t change ptr itself)
  + Public API – make as many fields private as possible; information hiding
    - Accessor – reports value(s) without changing the object

string getName() const;

string SomeClass::getName() const { … }

* + - Mutator – changes value(s) of the object

void setName(string name);

void SomeClass::setName(string name) { … }

* **Inheritance & polymorphism**
  + Parent/abstract base class – never instantiated; only to provide a shell/template
    - Pure virtual method – has no definition in parent class; must be defined by children

virtual void abstractMethod() = 0;

* + Child/concrete/derived class – overrides members of parent class, adds new members
    - Only need to declare new or overriding parts

class Child : public Parent { //indicate inheritance

public:

Child();

Child(string val);

virtual ~Child();

virtual void abstractMethod(); //override virtual method

};

Child::Child() : Parent() {} //child ctor needs to call parent ctor

Child::Child(string val) : Parent(val) {}

void Child::abstractMethod() const { … } //define overriding method

Child::~Child() { … } //parent’s dtor is called automatically

* + Member variables & methods of parent are inherited by child
    - Constructor & destructor not inherited
  + Template method design – parent is a template and uses specific implementations that its children provide
  + Polymorphism – treating similar objects in a uniform way
    - Any instance of child class can be treated as an instance of its parent class
    - A polymorphic container of a parent class can contain instances of its subclasses

vector<Parent\*> v; v.push\_back(childInstance);

* **Static & dynamic typing**
  + Each object has a vptr to the vtable for its class (in static data segment)
  + Static type – what it’s declared to be; determines the type of API access
  + Dynamic type – what kind of object it’s actually pointing to
    - Always a descendent of its static type

Parent \*c = new Child; //limited to the public API of Parent

c->parentMethod(); //legal

c->childMethod(); //illegal

* + Static dispatch – if method is not declared as virtual in parent, the parent (static type) method will always be called
  + Dynamic dispatch – if method is declared as virtual in parent, the dynamic type method will be called (if it exists)
    - Parent \*p = new Child;

p->method(); //Parent::method() or Child::method() will be

//called depending on if Parent::method() is

//declared as virtual

* + - Thus dtors of parent classes should be declared as virtual
* **Generics & templates**
  + Generics – the parameters’ type is provided as a special parameter
    - template <typename T>

class SomeClass { … } //declaration

* + - template <typename T>

SomeClass<T>::method(T val) { … } //definition

* + - SomeClass<string> \*c = new SomeClass<string>;
* **Object-oriented design**
  + Adapter/wrapper design – don’t inherit STL classes, instead instantiate them in an adapter class – “has-a” over “is-a”

**Standard Template Library**

* **Iterators**
  + Go through a collection of data one at a time, e.g. with a for loop
    - Pointer to first element of the collection
      * c.begin() for STL containers
    - Pointer to just beyond the last element (stopping criterion)
      * c.end() for STL containers
    - ++/-- advances to the next/previous element
    - \* dereferences and returns that element

vector<string> v;

for (vector<string>::iterator vi = v.begin(); v1 != v.end(); vi++) {

cout << \*vi << endl;

}

* + Can also be const, forward, reverse, bidirectional
  + Random access iterator allows vi[n] in ACT – not supported by list, map & set
  + STL supports:
    - v.insert(i1, i2, i3);
      * Insert into v at position i1, a range of external elements from i2 to i3
    - v.erase(i1, i2);
      * Delete from v elements from i1 to i2
* **Algorithms**
  + #include<algorithm>
    - E.g. sort, find, next\_permutation etc.
* **Sequence containers**
  + Elements are organized in the order they are added
  + Vector
    - Supports pointer arithmetic & random access
    - Implemented w/ dynamically allocated array
    - Inserting after size == capacity causes a reallocation (object copy of every element)
      * Thus external refs are not preserved

|  |  |  |
| --- | --- | --- |
| Append/delete last | O(1)/ACT | push\_back()/pop\_back() |
| Prepend/delete first | O(N) | - |
| Random access | O(1) | [], at() |
| Random insert/delete | O(N) | insert()/erase() |

* + Deque aka. Double-ended queue
    - Supports O(1) insert/delete at front & back, doesn’t support pointer arithmetic
    - Implemented w/ a circular buffer of pointers, w/ each pointing to a fixed-sized array (chunk of K elements each)
      * Circular buffer – use first & last ptrs and mod arithmetics to keep track
    - Inserting after circular buffer of ptrs is full causes reallocation (copy of N/K ptrs)
      * Thus external refs are preserved

|  |  |  |
| --- | --- | --- |
| Append/delete last | O(1)/ACT | push\_back()/pop\_back() |
| Prepend/delete first | O(1)/ACT | push\_front()/pop\_front() |
| Random access | O(1) | [], at() |
| Random insert/delete | O(N) | insert()/erase() |

* + List
    - Supports sequential access only
    - Implemented w/ doubly linked list

|  |  |  |
| --- | --- | --- |
| Append/delete last | O(1) | push\_back()/pop\_back() |
| Prepend/delete first | O(1) | push\_front()/pop\_front() |
| Random access | O(N) | - |
| Random insert/delete | O(N) | insert()/erase() |

* **Container adapters**
  + Stack – implemented w/ vector, deque, or list
  + Queue – implemented w/ deque or list
  + Priority queue – implemented w/ vector or deque
* **Associative containers**
  + Elements are organized based on a key value
  + Implemented w/ red-black trees (O(log n) operations)
  + Map
    - map<keyType, valueType> m;
    - Key must support < (strict weak ordering)
    - Equality is defined as if(!(a < b) && !(b < a))
  + Multimap – key is not unique for each item, i.e. one key → multiple elements
  + Set
    - set<Type> s;
    - Does not support duplicate elements
  + Multiset – supports multiple elements of the same key