CSE 1325 Exam 3 Study Sheet

This study sheet is provided AS IS, in the hope that the student will find it of value in preparing for the indicated exam. As always, it is the student's responsibility to prepare for this exam, including verification of the accuracy of information on this sheet, and to use their best judgment in defining and executing their exam preparation strategy. *Any grade appeals that rely on this sheet will be rejected.*

Most concepts and much of the coding is the same as with Java in the previous exams or C from CSE1310 and CSE1320. The last exam will likely focus on the unique features of C++ relative to Java and C.

Changes since Exam 2: Deleted some non-C++ vocab words and changed "generic" to "template" and "collection" to "container". Added Pointers, Custom C++ Types (replaces Custom Java Types), Method Parameters ("pass by..."), Namespaces (replaces Packages and JavaDoc), Operator Overloading, Casting, Streams (replaces Input / Output), and Standard Template Library (replaces Java Class Library). Removed Concurrency (Threads), Lambdas, and Templates (Generics) which are no longer covered for C++.

Vocabulary

- **Object-Oriented Programming (OOP)** – A style of programming focused on the use of classes and class hierarchies

The "PIE" Conceptual Model of OOP

- **Polymorphism** – The provision of a single interface to multiple subclasses, enabling the same method call to invoke different subclass methods to generate different results
- **Inheritance** – Reuse and extension of fields and method implementations from another class
- **Encapsulation** – Bundling data and code into a restricted container
- **Abstraction** – Specifying a general interface while hiding implementation details (sometimes listed as a 4th fundamental concept of OOP, though I believe it's common to most paradigms)

Types and Instances of Types

- **Primitive type** – A data type that can typically be handled directly by the underlying hardware
- **Enumerated type** – A data type that includes a fixed set of constant values called enumerators
- **Class** – A template encapsulating data and code that manipulates it. Don't confuse *class*, which is a template for making objects, with *template class*, which is a type of class (called a *generic* in Java) used to create other classes based on generic types (see "Templates" below).
- **Instance** – An encapsulated bundle of data and code (e.g., an instance of a program is a process; an instance of a class is an object)
- **Object** – An instance of a class containing a set of encapsulated data and associated methods
- **Variable** – A block of memory associated with a symbolic name that contains a primitive data value or the address of an object instance
- **Operator** – A short string representing a mathematical, logical, or machine control action

Class Members Et. Al.

- **Field** – A class member variable (also called an "attribute" or "class variable")
- **Constructor** – A special class member that creates and initializes an object from the class
- **Destructor** – A special class member that cleans up when an object is deleted
- **Method** – A function that manipulates data in a class (also called a "class function")
- **Friend** – A class or a function (NOT a method!) that is granted access to its friend class' private class members
- **Operator Overloading** – Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (e.g., a class)
- **Getter** – A method that returns the value of a private field
- **Setter** – A method that changes the value of a private field

Inheritance

- **Multiple Inheritance** – A subclass inheriting class members from two or more superclasses
- **Superclass** – The class from which members are inherited (called "base class" in C++)
- **Subclass** – The class inheriting members (called "derived class" in C++)
- **Abstract Class** – A class that cannot be instantiated (called "pure virtual class" in C++)
- **Abstract Method** – A method declared with no implementation (called "pure virtual method" in C++)
- **Override** – A subclass replacing its superclass' implementation of a virtual method

Scope

- **Namespace** – A named scope
- **Declaration** – A statement that introduces a name with an associated type into a scope
- **Definition** – A declaration that also fully specifies the entity declared
- **Shadowing** – A variable declared in a narrower scope than that of a variable of the same name declared in a broader scope

Memory Spaces

- **Stack** – Scratch memory for a thread of execution (in C++, e.g., `Foo f;`)
- **Heap** – Memory shared by all threads of execution for dynamic allocation (`Foo* f = new Foo;`)
- **Global** – Memory for static fields and non-scoped variables
- **Code** – Read-only memory for machine instructions

Algorithms

- **Generic Programming** – Writing algorithms in terms of types that are specified as parameters during instantiation or invocation
- **Template** – A C++ construct representing a class, method, or function in terms of generic types
- **Standard Template Library** – A library of well-implemented algorithms focused on organizing code and data as C++ templates
- **Container** – An object that stores and manages other objects (called a Collection in Java)
- **Algorithm** – A procedure for solving a specific problem, expressed as an ordered set of actions
- **Iterator** – A pointer-like standard library abstraction for objects referring to elements of a container

Error Handling

- **Exception** – An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code
- **Assertion** – An expression that, if false, indicates a program error (in C++, via the `assert` function from <cassert>)
- **Invariant** – Code for which specified assertions are guaranteed to be true (often, a class in which fields cannot change after instantiation)
- **Data Validation** – Ensuring that a program operates on clean, correct and useful data
- **Validation Rules** – Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data

Version Control

- **Version Control** – The task of keeping a system consisting of many versions well organized
- **Branch** – A second distinct development path within the same organization and often within the same version control system
- **Fork** – A second distinct and independent development path undertaken (often by a different organization) to create a unique product
- **Baseline** – A reference point in a version control system, usually indicating completion and approval of a product release and sometimes used to support a fork

Process

- **Unified Modeling Language (UML)** – The standard visual modeling language used to describe, specify, design, and document the structure and behavior of object-oriented systems
- **Class Hierarchy** – Defines the inheritance relationships between a set of classes
- **Class Library** – A collection of classes designed to be used together efficiently
- **Principle of Least Astonishment** – A user interface component should behave as the users expect it to behave

Object-Oriented Programming

Encapsulation is simply bundling data (fields) and code (methods) into a container (class) with restricted scope (protected and private). Know how to write a class declaration (a .h file) from a UML class diagram. Constructors may chain to other constructors using the class name (`Foo() : Foo(0) {}`).

Inheritance is reuse and extension of fields (fields) and method implementations from another class. Know how to extend a subclass from one or more superclasses, as in `class Dog : public Animal;`. Protected and public members of the superclass inherit (that is, they are visible from the subclass); private members, constructors, and friends do NOT inherit, although superclass constructors may be invoked by name (`Dog(int x) : Animal(x) { }`).

Multiple inheritance, e.g., `class TA : public Student, public Faculty;` is perfectly fine. Be able to recognize, explain, and resolve the diamond problem, and its solutions – scope resolution operator (`Dog::bark()`) and virtual inheritance (`class Dog: virtual public Animal;`).

Use keyword override on a method declaration when overriding a superclass method, so the compiler will report an error if the superclass declares no matching method signature.

Polymorphism is the provision of a single interface (*virtual* superclass methods) to multiple subclasses, enabling the same method call (via pointers or references only) to invoke different subclass methods to generate different results.

When a (virtual) method is called on a variable of a superclass that holds an object of a subclass, the subclass object's method is invoked. In C++, **polymorphism only works when the superclass method is declared as virtual and the variable is a pointer or reference**, due to memory layout issues. Thus, a vector of superclass type intended to hold subclass objects *must be a vector of references or pointers in C++ only* (this is NOT true in most languages!).

```
std::vector<Superclass*> v;
v.push_back(new Subclass);
for(auto* d : v) d->foo();
```

Above is an example of polymorphism, assuming `Superclass::foo()` is virtual and `Subclass::foo()` overrides it.

General C++ Knowledge

C++ syntax: (from CSE 1320) assignments, operators, relationals, naming rules, the 4 most common primitives (bool, char, int, double) and common classes (string, vector, map, set), instancing (invoking the constructor), for and for each, while, if / else if / else, the ? (ternary) operator `(x = (a > b) ? a : b;)`, switch statements (NO expressions!)

#include: Difference between #include <iostream > (search the system libraries only) and #include "shape.h" (search the local directory first, then the system libraries)

Namespaces: Purpose and how to use them. Know the scope resolution (membership) operator `::`

Four types of scope: Global, class, local, and statement (for) and be able to explain them.

Visibility levels: (C++ does NOT support package-private)

- **Private** – Accessible in this class and its friend functions and friend classes only
- **Protected** – Accessible in this class, its subclasses, and its friend functions and friend classes only
- **Public** – Accessible anywhere in scope

Difference between:

- **Declarations** – The interface, which may appear many times, e.g., `int s();`, generally in the .h file
- **Definitions** – The implementation, which must appear once, e.g., `int s() {return v.size();)`, generally in the .cpp file. Constructors, methods, and *static* fields **must** be defined in the .cpp file!

Know how to use a **header guard** `#ifndef __X_H / #define __X_H / #endif`

Pointers

- Declare and access pointed-to value with an asterisk, e.g., `int x = 7; int* i = &x; std::cout << *i;` We call `*i` a **dereference**. Dereferencing a non-initialized pointer gives a segfault.
- Construct an object on the heap with `new`, access members with ->, and free with `delete`, e.g., `Foo* foo = new foo; foo->update(); delete foo;`
- Delete arrays using `delete[]`, e.g., `int* ia = new int[1000]; delete[] ia;` Deleting a `nullptr` is harmless, but deleting a deleted memory region gives a runtime error.

Custom C++ Types

4 (maybe 5) C++ custom type mechanisms:

- **Enum** – All members public, NO constructors, methods, or fields, just a list of int constants.
  ```
  enum Color {red, green, blue};
  std::string c_to_s[3] = {"Red", "Green", "Blue"};
  Color c = Color::green; std::cout << c << ' ' << c_to_s[c];
  ```
  prints "1 Green".
- **Enum class** – All members public, NO constructors, methods, or fields, but type checking is enforced. NOT interchangeable with ints! `enum class Animal {dog, cat}; Animal a = Animal::dog; std::map<Animal, std::string> a_to_s = {{Animal::dog, "Dog"}, {Animal::Cat,"Cat"}}; std::cout << a_to_s[a];` prints "Dog".
- **Struct** – All members public by default, identical to `class` except default visibility, *traditionally* has no methods but they *are* permitted exactly like a class. But avoid `struct` in CSE1325. Use `class` instead.
- **Class** – All members private by default, fully definable type just as in Java.
- **typedef** - Aliases a type, for example, `typedef double Altitude;`

Class Members

```
class Foo {
    public: Foo(int x, int y) : _x{new int{x}}, _y{new int{y}} {}
            Foo() : Foo{0, 0} { }
            Foo(Foo& f) : Foo{*f._x, *f._y} {}
            virtual ~Foo() {delete _x; delete _y;}
            void operator=(Foo& f) {if(this == &f) return;
                _x = new int{*f._x}; _y = new int{*f._y};}
            int p() {return *_x * *_y;}
            Foo operator+(const Foo& rhs) {Foo f{*_x + *rhs._x, *_y + *rhs._y}; return f;}
            friend Foo operator*(const Foo& lhs, const Foo& rhs) {
                Foo f{*lhs._x * *rhs._x, *lhs._y * *rhs._y}; return f;
            }
    private:
            int* _x;
            int* _y;
};
int product(int x, int y) {Foo f{x, y}; return f.p();}
```

- The class is `Foo`
- The fields are `int* _x` and `int* _y;`
- The 3 constructors are `Foo(int x, int y) : _x{new int{x}}, _y{new int{y}} {}` and `Foo() : Foo{0,0} { }` and `Foo(Foo& f) : Foo{*f._x, *f._y} {}`
- Constructors `Foo()` and `Foo(Foo& f)` chain to constructor `Foo(int x, int y)`
- The initialization lists are `: _x{new int{x}}, _y{new int{y}}` and `: Foo{0,0}` and `: Foo{*f._x, *f._y}`
- The copy constructor is `Foo(Foo& f) : Foo{*f._x, *f._y} {}`
- The copy assignment operator ("operator overloading") is `void operator=(Foo& f) {if(this == &f) return; _x = new int{*f._x}; _y = new int{*f._y};}`
- The addition operator method ("operator overloading") is `Foo operator+(const Foo& rhs) {Foo f{*_x + *rhs._x, *_y + *rhs._y}; return f;}`
- The multiplication operator function ("operator overloading") is `friend Foo operator*(const Foo& lhs, const Foo& rhs)`
- The destructor is `virtual ~Foo() {delete _x; delete _y;}`
- The method is `int p() {return *_x * *_y;}`
- The friend function (*not* a class member) and more "operator overloading" is `friend Foo operator*(const Foo& lhs, const Foo& rhs)`
- The non-friend function (*not* a class member) is `int product(int x, int y)`
- An instance is `Foo f{x, y};`

A **default constructor** (with no parameters, like `Foo()` above) is provided by default, but **only** if no non-default constructor is defined. Any number of constructors may be defined ("overloading"), as long as the types of each set of parameters is unique (called the “parametric signature” of the constructor). "Overloading" and "parametric signature" also apply to methods and functions, but they include its return type as well.

Class Member Options

- **Non-static field** – A unique value for every object, e.g., `std::string name;` in .h
- **Static field** – The same value (and memory location) shared among all objects of this class, e.g., `static int next_id;` in .h file and `int Dog::next_id = 1;` in .cpp file (to allocate memory)
- **Non-static method** – Called only via the object (`dog.roll_over()`), can access both static and non-static fields
- **Static method** – Called via object (`dog.bark()`) or class (`Dog::bark()`), can access only static fields
- **Non-const field** – Value can be changed by any code in scope
- **Const field** – Value cannot be changed once constructed by init list, e.g., `const int age;` in .h file
- **Non-const method** – May change any non-const field
- **Const method** – May not change any field, e.g., `void bark() const;` in .h file and `void Dog::bark() const {` in .cpp file
- **Non-virtual method** – Cannot be overridden in subclass
- **Virtual method** – May be overridden in subclass, e.g., `virtual void foo();` in .h file
- **Overridden method** – Matches superclass' virtual method's name, parameter types and order, and return type, e.g., `void roll_over() override;` in .h file
- **Overloaded method** – Matches method name in *same* class, but with different parameter types or order - also consider default parameter values as an alternative, e.g., `void bar(int p1, int p2=0)` in class Foo can be called via `foo.bar(42,1024)` OR `foo.bar(7)`

Method Parameters

- Pass by **value** – Copied to stack, so changes will not be reflected in the calling program. Best for primitive or small parameters that need not change, e.g., `int x`
- Pass by **reference** – Address is passed, so changes WILL be reflected in the calling program. Use ONLY when calling program values must change, such as sort in place, e.g., `Foo& foo`
- Pass by **const reference** – Address is passed, but compiler will prohibit changes to the parameter. Best for large parameters that need not change, e.g., `const Foo& foo`.
- Pass by **pointer** – Address is passed, so changes WILL be reflected in the calling program, PLUS pointer math but risk of segfaults, e.g., `Foo* foo`.

Operator Overloading

Know how to define an operator such as + or << (given the method OR function signature such as `std::ostream& operator<<(std::ostream& ost, const Foo& f);` - DON'T memorize).

```
std::ostream& operator<<(std::ostream& ost, const Complex& c) {
    return ost << "( " << c.r << " , " << c.i << " ) ";
}
std::istream& operator>>(std::istream& ist, Complex& c) {
    char x; return ist >> x >> c.r >> x >> c.i >> x; // x captures the ( , ) delimiters
}
```

Rule of Three

If you need to define a copy constructor, copy assignment operator, OR destructor for a class, you almost certainly need to define all three. (This is less commonly called the Rule of Four, because if you need a destructor, you probably need a custom constructor as well - but that's not on the exam.)

> **Copy constructor**, e.g., `Foo::Foo{Foo& y};)`, is invoked directly when a new class is constructed from an existing class (`Foo x{y};`), when an object is passed by value to a function or method (`v.push_back(foo);`), or when an object is returned by value from a function (`return foo;`).
>
> **Copy assignment operator** (overloaded operator=) is called when an existing object is overwritten / assigned new values (`x = y;`). As with constructors and destructors, **C++ provides a default copy constructor and copy assignment operator** for each class (which copy the binary contents of corresponding variables – including just the addresses for pointers). Be able to write a copy constructor and (given the function signature) a copy assignment operator. Also know how to disable a copy constructor and copy assignment operator – like this (in the public member area):
> `Foo(const Foo&) = delete; Foo& operator=(const Foo&) = delete;`
>
> **Destructor** is declared exactly like a constructor, except with a leading tilde (~) and no parameters, e.g., the `~Foo()`. It typically frees up resources such as heap, locks, sockets, etc. **If a superclass has *any* virtual methods, each subclass should declare a virtual destructor**, even if its empty, to ensure that all destructors are invoked when an object is deleted.

Casting

While C-style casting is available (`((T*) p)`), use `static_cast<T*>(pv);` (note that despite the suspicious similarity to templates, *casts are not templates* – they are operators). Static_cast does additional checking.

Given a class hierarchy from a superclass *with at least one virtual member*, if you want to cast pointers between subtypes, use dynamic_cast to verify the object matches the pointer type. For example, given a superclass named `Super` and two subclasses `Sub1` and `Sub2`, then `Super* b = new Sub2; Sub2* d = dynamic_cast<Sub2*>(b);`. If b is indeed pointing to an object of type Sub2 (and it is), the pointer is cast. If it were not, d would be set to nullptr.

If dynamic_casting to a reference instead of a pointer, and the cast is invalid, `std::bad_cast` is thrown instead (since a reference cannot be null).

Namespaces

Just a named region of code. Can be declared in multiple locations, in which case they aggregate.

`namespace rice {const int spam = 7;}  int main() {std::cout << rice::spam;}`

Can be imported into the local namespace, e.g.,

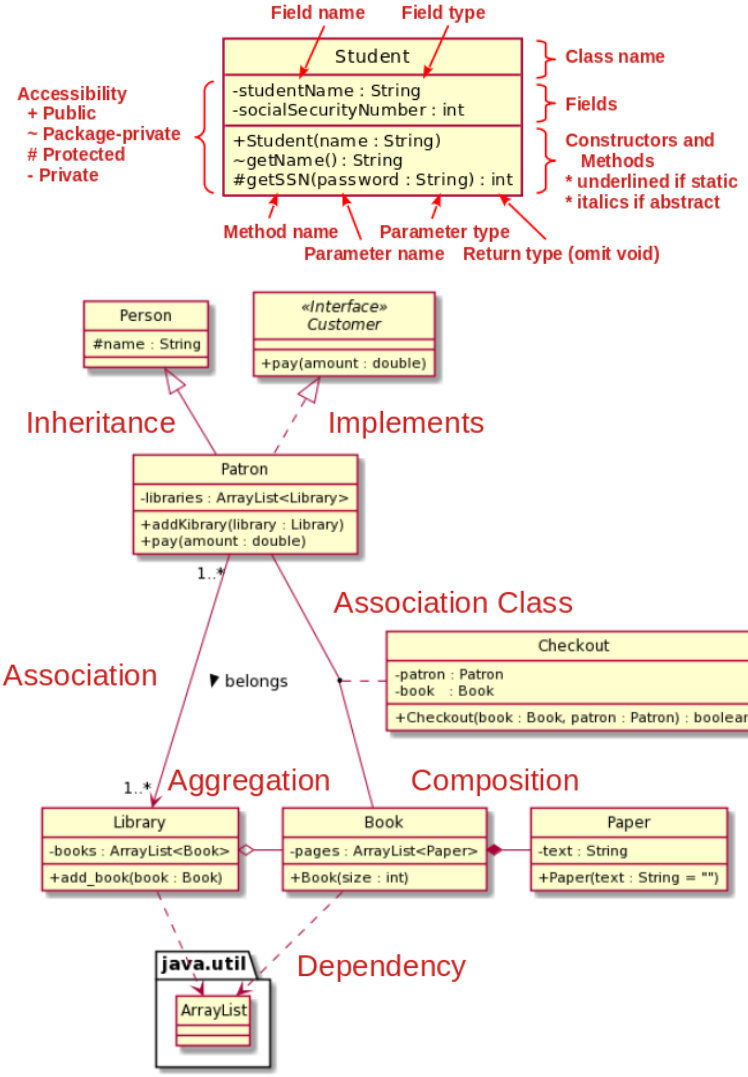
`using namespace std; int main() { cout << "No std::!" << endl; }`

but this is a bad practice. More acceptable is to import specific namespace elements, e.g.,

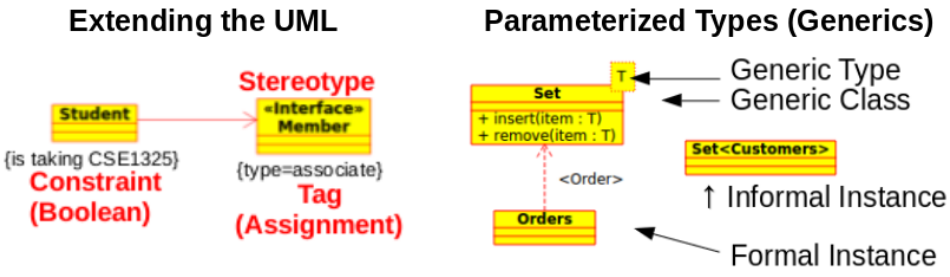
```
using std::this_thread::sleep_for;
using std::chrono::milliseconds;

void pause_and_print(int ms, std::string msg) {
      sleep_for(milliseconds(ms));   // wait for ms millisecond
      std::cout << msg << std::endl; // print the message
}
```

UML Class Diagram

Understand and be able to draw the basic forms of the UML class diagram, the various relationships between, and user-defined extensions below.



In other words, just like Java. :) In C++, an interface is written as a purely abstract class (no .cpp file).



3 types of extensions: Stereotype (such as «`port`»), tags (assignments such as `{language=Java}`), and constraints (Booleans such as `{interfaces}` or `{sloc>100}`).

Streams

Streams generalize I/O as a sequence of characters to or from any source - the console, keyboard, file, a string, a device, whatever.

- The << operator streams data to an output stream, e.g., `std::cout << "The answer is " << x << std::endl;`
- The >> operator streams data from a source, parsing on whitespace, e.g., `std::cin >> x >> y >> s;` given input "3 5 Hello, World!" will set x to 3, y to 5, and s to "Hello,".
- `std::getline` streams data from a source, parsing on newlines, e.g., `std::getline(std::cin, s);` given input of "George F. Rice" will read the entire string into s
- `std::getline` can also parse on a character other than `\n` using a third parameter, e.g., `std::getline(std::cin, s, ',');` parses on commas instead of newlines.

Stream States

The 4 stream states are **good, eof, fail,** and **bad**. Check them using e.g., `std::cin.good()`, `std::cout.fail()`, or using the common idioms `while (std::cin >> w) words.push_back(w);` or `while (std::getline(std::cin, s)) lines.push_back(s);`. Know what they mean (`good()` = success, `eof()` = end of file, `fail()` = recoverable error, `bad()` = unrecoverable error).

I/O Manipulators

The I/O manipulators allow you to format data to a stream, similar in functionality to printf and sprintf. You must `#include <iomanip>`. Know the common I/O manipulators `std::dec`, `std::hex`, `std::showbase`, `std::precision(digits)`, `std::fixed`, `std::setfill(char)`, and (the only NON-sticky manipulator) `std::setw(width)`. Know how to use them to format output streams, e.g., `std::cout << std::hex << std::showbase << std::setw(4) << i << std::endl;`.

Console I/O

```
#include <iostream>
```

- Use `std::cout` to output data, but `std::cerr` to output error messages. Bash can redirect these separately, e.g, `./a.out > cout.txt 2> cerr.txt`
- Use `std::cin` to input from the keyboard. Use `std::cin.ignore` to clear the input buffer on bad input, e.g., `int i; std::cin >> i; std::cin.ignore();`

File I/O

```
#include <fstream>
```

Know why text file formats are superior to binary (flexible, portable, easy to document and debug, etc.). Know the major ios_base file open types `read`, `write`, and `app` (append).

- ifstream: `std::ifstream ifs{"in.txt"}; std::string s; ifs >> s;`
- ofstream: `std::ofstream ofs{“out.txt”}; ofs << “Hello”;`
- ios_base: `std::ofstream ofs{“out.txt”, std::app}; // append to out.txt`

After opening a file, check for errors (`if(!ifs)) throw std::istream::failure{"failed"};`).

String I/O (String Streams)

```
#include <sstream>

std::string s = "3.14"; double d;   // Declare a string and a double
std::istringstream iss{s};          // Create input stream from s
iss >> d;                           // Read string s as a double
if (!iss) {std::cerr << "Invalid";} // Detect an error on read above

std::ostringstream oss;             // Declare an output stream
oss << d;                           // Stream the double to the stream
s = oss.str();                      // Convert the stream to a string
```

Error Handling

Know how to instance, throw, catch, and handle exceptions. The following prints "Failed with bad data".

```
int foo(int data) {
    if (data > 10) throw std::runtime_error{“bad data”};  // Throw an exception
}
int main() {
    try {  // Create scope in which exceptions can be caught
        foo(42);
    } catch (std::runtime_error e) {  // Catch the runtime_error
        std::cerr << “Failed with “ << e.what() << std::endl;
    }
}
```

You may also define custom exceptions by inheriting from class std::exception. I will give you the `what()` method signature if needed - do NOT memorize that!

```
class Bad_char : public std::exception {
  public:
    Bad_char(std::string s, char c) {
        msg =  "Bad character " + std::string{c} + " in " + s;
    }
    const char* what() const noexcept override {
        return msg.c_str();
    }
  private:
    std::string msg;
};
```

`return 0;` from `main()` or via `exit(0);` from anywhere for success, a non-zero error code if the program fails (but `exit(n)` doesn't run destructors). Use `std::cout` for data only and `std::cerr` for error messages only.

Understand the concepts of pre-conditions (at the start of the method, usually verifying parameters) and post-conditions (at the end of a method, usually verifying results). Use the assert macro (from <cassert>) to check them: `assert(error_code == 0);` If error_code isn't 0, the program aborts with an “assertion failed: error_code == 0” message on std::cerr.

Standard Template Library

The Standard Template Library (STL) is part of the C++ library and provides

- **Containers** such as vector, linked_list, set, and map, usually as templates
- **Algorithms** like sort, random_shuffle, and find
- **Iterators** which allow algorithms to generically manipulate collections (called containers in C++), and
- **Functors**, which are NOT on the exam (FYI: they are objects with `operator()` defined which can thus be called like a function – in effect, functions with methods, private / protected class-level variables, inheritance, etc.).

Know that you NEVER inherit from an STL class; use composition instead as in our example during lecture.

Algorithms

Functions

Know how to use these C++ functions (the specification shown in bold will be provided on the exam).

- **void sort (RandomAccessIterator first, RandomAccessIterator last)** (from <algorithm>) – To add 10 ints between 0 and 99 inclusive into vector v and then sort them, try `for(int i=0; i<10; ++i) v.push_back(rand()%100); std::sort(v.begin(), v.end());` To sort with a custom type, you must either overload its operator< or provide a function that accepts two references to that type and returns true if the first is less than the second (think lambda).
- **void random_shuffle (RandomAccessIterator first, RandomAccessIterator last)** (from <algorithm>) – To shuffle (randomly order) an unordered container, use iterators. `std::random_shuffle(v.begin(), v.end());`
- **InputIterator find (InputIterator first, InputIterator last, const T& val)** (from <algorithm>) – Returns an iterator to the first element found that `== val`. `auto it = std::find(v.first(), v.last(), val); if(it == v.last()) std::cerr << "Not found!";`
- **int distance(InputIterator first, InputIterator last)** (from <algorithm>) – Returns the number of elements between iterator `first` and iterator `last`. `int index = std::distance(v.begin(), it);` calculates the index of the element found by `std::find`.
- **int count(InputIterator first, InputIterator last, const T& val)** – Returns the number of `val` values between iterators `first` and `last`.

Containers (Collections)

Containers are objects that hold a collection (the Java term) of other objects, called its *elements*.

Four general categories of containers are generally recognized in modern C++.

- **Sequence Containers** (vector, array, etc.)
- **Container Adapters** that wrap Sequence Containers to create a stack, queue, priority_queue, etc.
- **Associative Containers** (map / multimap) that map (sorted) keys of any type to values of any other type and (set / multiset) that just store sorted keys (multi- allows duplicate keys)
- **Unordered Associated Containers** that are just like the above but don't sort the keys automatically.

Know the most commonly used containers and their common operations:

- **string** (from <string>) – `std::string s;` to instance, `s[1]` to index a char, `s1 + s2` to concatenate, `for(char c: s) std::cout << c << ' ';` to iterate
- **vector** (from <vector>) – `std::vector<T> v` to instance for type T, `v[1]` to index, `v.push_back(val);` to append, `for(auto e : v) std::cout << e << ' ';` to iterate
- **set** (from <set>) – Sorted by value, with duplicates removed! `std::set<double> s; s.insert(3.14);` to instance and append, iterate like a vector
- **map** (from <map>) – Sorted by key! `std::map<key_type, value_type> m;` to instance, `m[key] = value;` to add a pair, `m[key]` to index, `for(auto& [key, value]) std::cout << key << '=' << value << '\n';` to iterate

Comparative Table of Container Methods

	std::vector	**std::string**	**std::set**	**std::map**
Is empty?	**v.empty()**	**s.empty()**	**s.empty()**	**m.empty()**
Clear	**v.clear()**	**s.clear()**	**s.clear()**	**m.clear()**
How many?	**v.size()**	**s.size() or s.length()**	**s.size()**	**m.size()**
Random access	**val = v[index]**	**c = s[index]**	--	**val = m[key]**
only if index exists	**val = v.at(index)**	**c = s.at(index)**	--	**val = m.at(key)**
Value or key exists?	**--**	**--**	**s.count(val)***	**m.count(key)**
Overwrite value	**v[index] = val**	**s[index] = c**	**s.insert(val)**	**m[key] = val**
Erase value	**v.erase(it)**	**s.erase(index, len)**	**s.erase(val)**	**m.erase(key)**
Insert value	**s.insert(val, it)**	**s.insert(index, str)**	**s.insert(val)**	**--** (omitted)
Iterate	**for(auto& val : v)**	**for(auto& c : s)**	**for(auto& val : s)**	**for(auto& [key,val] : m)**
Get iterator to first	**it = v.begin()**	**it = s.begin()**	**it = s.begin()**	**it = m.begin()**
Get iterator to last+1	**it = v.end()**	**it = s.end()**	**it = s.end()**	**it = m.end()**

Iterators

Iterators are **nested classes within container classes** that enable access to container elements, especially via for-each loops.

Iterators come in `iterator` (allows item modification) and `const_iterator` (forbids item modification) varieties. You would obtain a const_iterator much as you would an iterator:

```
std::vector<int> v{1,2,3,4,5};
std::vector<int>::const_iterator p = v.begin();
*p = 10; // ERROR: p is a const_iterator and cannot change v
```

Iterators are coded to behave like a pointer, but are usually objects. We typically obtain one via the container's `begin()` and `end()` methods, in which case they point to the first and one past the last element in the container, respectively.

Common methods and operators for iterators include dereference (*), increment (++), copy, destruct, and compare for equality. Here's a simple example of iterator use.

```
int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    std::vector<int>::iterator it = v.begin();
    do {
        std::cout << *it << std::endl;
    } while(++it != v.end());
}
```