CSE 1325 Final Exam Study Sheet (May 8-10, 2018)

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. Information that has been added since Exam #2, or that may be emphasized on this exam, may be in red.**

Definitions (understand, don't memorize):

- **Object-Oriented Programming (OOP) –** A style of programming focused on the use of classes and class hierarchies. The "PIE" model of OOP is based on 3 fundamental concepts:
  - **Encapsulation** – Bundling data and code into a restricted container
  - **Inheritance** – Reuse and extension of fields and method implementations from another class
  - **Polymorphism** – The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results

- **Class** – A template encapsulating data and code that manipulates it
- **Class Hierarchy** – Defines the inheritance relationships between a set of classes
- **Class Library** – A collection of classes designed to be used together efficiently.
- **Base Class** – The class from which members are inherited
- **Derived Class** – The class inheriting members
- **Override** – A derived class replacing its base class' implementation of a method
- **Multiple Inheritance** – A derived class inheriting class members from two or more base classes
- **Primitive type** – A data type that can typically be handled directly by the underlying hardware
- **Method** – A function that manipulates data in a class
- **Getter -** A method that returns the value of a private variable
- **Setter -** A method that changes the value of a private variable
- **Instance** – An encapsulated bundle of data and code
- **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 an object or a primitive data value
- **Operator** – A short string representing a mathematical, logical, or machine control action
- **Operator Overloading** – Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (e.g., a class)
- **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
- **Friend** – A function that is granted access to its friend class' private members
- **Regular Expression** – A string that defines a search pattern for other strings
- **Reference Counter** – A managed memory technique that tracks the number of references to allocated memory, so that the memory can be freed when the count reaches zero
- **Garbage Collector** – In a managed memory system, the program that periodically runs to free unreferenced memory
- **Unified Modeling Language (UML) -** The standard visual modelling language used to describe, specify, design, and document the structure and behavior of software systems, particularly OO
- **Algorithm** – A procedure for solving a specific problem, expressed in terms of an ordered set of actions to execute

- **Software Design Pattern -** A general reusable algorithm solving a common problem
- **Anti-Pattern** – A common error that occurs in software systems and processes
- **Exception** – An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code
- **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
- **Namespace** – A named scope
- **Invariant** – Code for which specified assertions are guaranteed to be true
- **Assertion -** An expression that, if false, indicates a program error
- **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 -** The task of keeping a software system consisting of many versions and configurations 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, using indicating completion and approval of a product release and sometimes used to support a fork
- **Waterfall Process** – Separating software development into long, discrete phases such as Requirements, Design, Implementation, Verification, and Validation
- **Agile Process** – Prioritizing individuals and interactions, frequent delivery of working software, customer collaboration, and flexible response to change
- **Principle of Least Astonishment** – A user interface component should behave as the users expect it to behave
- **Second System Effect** – The common mistake of attempting to include far too many features in (typically) version 2.0, causing catastrophic schedule slips
- **Architecture** – The set of implementation elements and associated integration mechanisms necessary to meet the system requirements.
- **Architect** – The engineering role that specifies a system's architecture.
- **Generalization** – The process of extracting shared characteristics from two or more classes, and combining them into a generalized base class.
- **Specialization** – The process of identifying and specifying derived classes from a base class.
- **State** – The cumulative value of all relevant stored information to which a system or subsystem has access
- **State Diagram** – A model that documents the states, permissible transitions, and activities of a system.
- **Guard** – A Boolean expression that enables a state transition when true & disables it when false
- **Event** – A Type of occurrence that potentially affects the state of the system
- **Stack** – Scratch memory for a thread
- **Heap** – Memory shared by all threads for dynamic allocation
- **Template** – A C++ construct representing a function or class in terms of generic types
- **Generic Programming** – Writing algorithms in terms of types that are specified as parameters during instantiation or invocation
- **Standard Template Library** – A library of well-implemented algorithms focused on organizing code and data as C++ templates

- **Iterator** – A pointer-like instance of a nested class used to access items in the outer class container
- **Concurrency** – Performing 2 or more algorithms (as it were) simultaneously
- **Process** – A self-contained execution environment including its own memory space
- **Thread** – An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space
- **Reentrant** – An algorithm that can be paused while executing, and then safely executed by a different thread
- **Mutex** – A mutual exclusion object that prevents two properly written threads from concurrently accessing a critical resource

General C++ Knowledge

C++ syntax: assignments, operators, relationals, naming rules, the 4 most common primitives (bool, char, int, double) and two common classes (string, vector), instancing (invoking the constructor), for and for each, while, if / else if / else, the ? (ternary) operator, and streams (cout, cin, and getline). Know the four types of string representations: the **string** class, the **basic_string** template, proprietary types e.g., gtkmm's **Glib::ustring**, and old C-style **char[] or char*** arrays, and their implications.

#include: Difference between < > (search the system libraries) and " " (also search the local directory)

Namespaces: Purpose and how to use them. Know the membership operator ::, e.g., std::cout

Four types of scope: Global, class, local, and statement (for)

Visibility levels: private (accessible in this class and its friend functions only), protected (accessible in this class, its derived classes, and its friend functions only), and public (accessible anywhere in scope)

Difference between **declarations** (the interface, which may appear many times) and **definitions** (the implementation, which may appear only once).

Declarations go in the .h, definitions in the .cpp *except for templates*, which go entirely in the .h. The use of #ifndef in the header file to avoid multiple class definitions.

Difference between **normal** and **static** class members (static are shared by all derivatives of the class). Know how to define (similar to a method definition in the .cpp file, with initialization in-line)

Difference in **call by value** (normal), **call by reference** (&), and **const call by reference** (const &)

Pointers: declaring, using to reference heap memory allocated via new and deallocated via delete or delete[] (and when to use each), dereferencing. Dangers of pointers. Why we often use pointers anyway (objects on the heap are persistent until explicitly deleted); access object members via a pointer using -> rather than .

Stack vs Heap vs Static Global vs Code memory spaces – Stack allocates memory for return address and local variables at start of a scope (called the "stack frame") as a LIFO stack. Heap is used for "new" allocations, and is much more complex to manage. Static is used for global variables, and is allocated at program launch. Code contains the executable code, and is usually read-only.

Four types of errors in programs: Compile-time, Link-time, Run-time, and Logic

Why **exceptions** are usually superior to a global error variable or returning an error code. How to create, throw, catch, and handle an exception (you may be asked to write code using exceptions!).

Difference between **cerr** and **cout**, and when and how to use each.

Concepts of **pre-conditions** and **post-conditions** for error detection in a method / function, and the use of the standard **assert** macro (from cassert) to check them. Turn off asserts by defining **NDEBUG**.

4 types of C++ classes: **enum**, **enum class**, **struct**, and **class**. Members of enum, enum class, and struct are public by default, but members of class are private by default. Struct and class are identical except for default visibility of their members.

Enum vs Enum Classes – An enum (enumeration) is just a list of aliases for integers. Enum class members are are NOT interchangeable with ints, and thus enforce type checking at compile time. An enum class can NOT include methods or other data, though.

Difference between **method**, **constructor** (a special type of class member), **delegated constructor** (also called chained constructor), **destructor** (a special type of class member that cleans up on deletion), and **function**. How to use **initialization lists** (sometimes called direct initialization) in a constructor.

A **default constructor** is supplied only if no non-default constructor is defined. Any number of constructors may be defined, as long as the types of each set of parameters / return types is unique.

Namespaces: Purpose and how to use them. Multiple definitions add to the namespace.

Operator Overloading: How to define an operator such as + or << (given the method / function signature, which you DON'T need to memorize). Why friend is often useful with function-based operator overloading (for access to its private variables).

Inheritance: How to derive a class from one or more base classes. Protected and public methods and field inherit. Private members, constructors, and friends to NOT inherit. How to model inheritance in UML (open-headed arrows) and how to code it in C++ (class Derived : public Base1, public Base2). Using the **override** keyword on a method declaration, so that the compiler will report an error if no matching method signature is found in a base class.

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

Lambdas: Anonymous function object, usually defined where invoked or passed as function argument. In the lambda definition "`[ ] (double x) -> double {return cos(x)+slope(x)};`"

- "[ ]" is the capture clause that defines visibility to surrounding variables. Examples:
  [&total, factor] means make total visible by reference, factor by value (copy);
  [&, factor] means make factor visible by value, all other variables by reference;
  [&total, =] means make total visible by reference, all others by value.
- "(double x)" is the optional parameter list
- "-> double" is the (optional) return type, usually deduced automatically by the compiler.
- The code inside { } is the usual function body.

Copy constructor is invoked 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;).

A **copy constructor** is called during **construction of an object** (Foo x = y;), while **copy assignment** is called when an **existing object** is overwritten / assigned new values (x = y;). As with constructors and destructors, C++ provides default copy constructors and copy assignment operators (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):

```
Shape(const Shape&) = delete;
Shape& operator=(const Shape&) = delete;
```

A **destructor** is declared exactly like a constructor, except with a leading ~ and no parameters, e.g., the String destructor is ~String(). It typically frees up resources such as heap, locks, sockets, etc.

Know the "Rule of Three" and be able to explain why: If you need to define a copy constructor, copy assignment, OR destructor for a class, you almost certainly need to define all three.

Pure virtual methods / functions – Setting a method declaration = 0 allows the base class to compile, but not to be instantiated (example: `virtual double increase(int i) = 0;`). This is superior to throwing a runtime exception in the non-pure virtual base class method, because pure virtual methods allow the compiler to detect invalid instantiation at compile time. Derived classes must implement pure virtual methods if they are to be instantiated.

The Diamond Problem – Be able to define it, and explain its application to and solutions for C++ (Lecture 16 – the solutions are "member notation", e.g., class::method, and "virtual inheritance"). Know that virtual inheritance means that a single instance is maintained of the data structures for the virtualized ancestor class.

Streams and Files – Be able to explain the advantages of stream operators over printf / scanf (more type-safe, less error prone, extensible, and inheritable), and the advantages of text files over binary files (easier to create, read, debug, and share across platforms). Know how to open a file via streams for reading, writing, appending, and random access (reading and writing). Know the 4 states of a stream (good, fail, bad, and eof), and the implications of each.

Stream Operators – Know the basic operators dec, hex, and oct (sets numeric base); showbase (precede octal numbers with "0" and hex numbers with "0x"); defaultfloat, scientific, and fixed (sets numeric output format); setprecision() (sets number of significant digits); and setw() (sets max chars on next output only, e.g., it is the only "non-sticky" stream operator we covered).

String Streams – Know that a string stream (e.g., istringstream) works exactly like other streams such as cin and cout, except interacting with strings in memory rather than the console or files. Be able to use a string stream for simple stream operations, e.g., to convert an object with operator<< defined to a string. Also know the basic char classifications from <cctype> - isspace(c), isalpha(c), isdigit(c), isupper(c), islower(c), and isalnum(c).

Pointers: declaring, using to reference heap memory allocated via new and deallocated via delete or delete[] (and when to use either), dereferencing. Dangers of pointers. Why we often use pointers anyway (objects on the heap are persistent until explicitly deleted). Know that to access object members via a pointer, use -> rather than ., e.g., non-pointer use obj.member, pointer use obj->member.

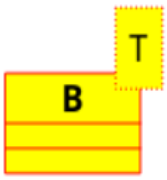
Pointers and Arrays – Memory is allocated from the heap using "new" for single objects or "new[]" for arrays, and deallocated using "delete" for single objects or "delete[]" for arrays, for example:

- `Foo *f = new Foo{}; ... ; delete f; // Single Foo object`
- `Bar *b = new Bar{}[5]; ... ; delete[] b; // Array of 5 Bar objects`

Failing to deallocate heap results in a "**memory leak**", ultimately causing the program to eventually crash when heap memory is exhausted.

Standard Template Library (STL) is part of the C++ library and provides **(1) Algorithms** like sort, search, copy, **(2) Containers** such as vectors, lists, maps, usually as templates **(3) Iterators** which allow algorithms to generically manipulate containers, and **(4) functors**, which are NOT on the exam (FYI: they are objects with operator() defined which can thus be called like a function – in effect, a function with methods, private / protected class-level variables, inheritance, etc.). Know that you **NEVER** inherit from an STL class; use composition instead as in our FILO class example.

Templates: What they are, why we want them (to write algorithms independent of the types to which the algorithms can apply – called "generic programming"), how to declare and instance. Know that the complete definition goes in the header, ***with no .cpp***. The template type must be "duck typed" - it must have the methods used by the template, or a compile error will result. The C++ compiler will only create one concrete definition for each type, regardless of how many template function calls or class instances for that type are used. Know how they are represented in UML (dashed "T" box to upper right of class as shown to the right). Be able to code simple examples. For example:



- `template <class T> class FILO { /* definition goes here */ };`
- `FILO<string> filo; // Create a FILO that manages strings`

Containers: What they are (an object that holds a collection of other objects, called its elements). The four major types of containers in the STL: (1) **Sequence Containers** (vector, array, etc.) (2) **Container Adapters** (wrap Sequence Containers to create a stack, queue, priority_queue, etc.), (3) **Associative Containers** map keys of any type to values of any type (map, set, and multimap / multiset), and (4) **Unordered Associated Containers** just like the previous ones but that don't sort the keys automatically. Know the most common methods (**begin()** and **end()** to obtain an associated iterator, front()/back() to access the first / last element, **size()** to find the number of elements, **push_back()** / **pop_front()** et. al. to append and delete from both ends, **insert(p, x)** to add element x before an arbitrary element p, and **erase(p)** to remove an arbitrary element p)

Iterators: What they are (nested classes within container classes that enable access to container items, especially via for-each loops), that they behave like a pointer but are usually objects, how to get one (typically **begin()** and **end()** methods on the container), what methods and operators are usually provided (dereference (*), increment (++), copy, destruct, compare for equality). They come in **iterator** (allows item modification) and **const_iterator** (forbids item modification) varieties. Be able to code simple examples. Know about **iterator delegation** and how to write it in C++ (you DON'T need to be able to write an iterator from scratch!)

Maps – An associative container template that stores elements via key-value, in essence, **a vector equivalent using any type as the index instead of just an integer**. Add elements using **`m[key] = value`** or **`m.insert(make_pair(key, value))`**, and access using **`value = m[key];`** or (in a loop)

```
for (auto it = m.begin(); it != m.end(); ++it)
    cout << it->first << " = " << it->second << endl;
```

Regular Expressions (regex) – a string that defines a search pattern for other strings.

- Non-special characters represent themselves, . (period) any character, ^ the start of a line, $ the end of a line, [ ] matches any single character inside (with "-" the range character, e.g., A-Z), and ( ) matches all characters inside as a single element.
- Know the traditional character classes: \s = whitespace (except newline), \w is a word character [A-Za-z0-9_], \d is a digit [0-9], and \n, \r, and \t represent newline, carriage return, and tab, respectively. The capitalized version means the inverse, e.g., \D means "not a digit".
- Know the multipliers: {m,n} means the preceding class must repeat at least m but no more than n times, * means 0 or more {0,}, ? means 0 or 1 {0,1}, and + means one or more {1,}.
- To search with a regex in C++, **`#include <regex>`**, instance a regex object, e.g., **`regex integer{"\d+"};`**, and use **`regex_match(string_to_search, integer)`** to match the entire string_to_search, or **`regex_search`** to search for substrings (return true for a match).

Void* - type that points to raw memory. Convert **`void* pv;`** to type T using **`static_cast<T*>(pv);`** (note that despite the suspicious similarity to templates, casts are **not** templates – they are operators). Also available is a C-like cast such as (T*)pv or a dynamic_cast (for polymorphic classes only)

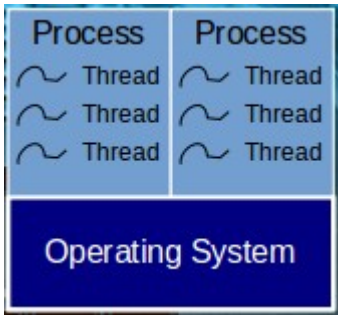
Embedded Programming

Know what it is (software that runs on special hardware) and the two types (hard real-time, where the correct answer after the deadline is wrong, and soft real-time, where some tolerance for missed deadlines is acceptable). Know that C++ embedded programming usually omits memory allocation after the program has started, exceptions, and many of the system libraries; but embraces templates, state machines, and preallocated data structures such as globals, stacks, and pools.

Know the bitset class well enough to code a simple example given the cplusplus.com documentation. Know the bit manipulation operators: & | ^ << >> and ~.

Know the UML state machine notation and the State Design pattern and their C++ implementation. Know hierarchical state diagrams. Know the difference between Mealy and Moore state machines, and that the UML supports both simultaneously. Know guards and events and how they work in practice.

Concurrency (Threads)



Concurrency enables us to make better use of the multiple processing units (cores) on a modern computer. Each program is an operating system **process** with its own private memory, which can each host many **threads** of execution sharing that private memory.

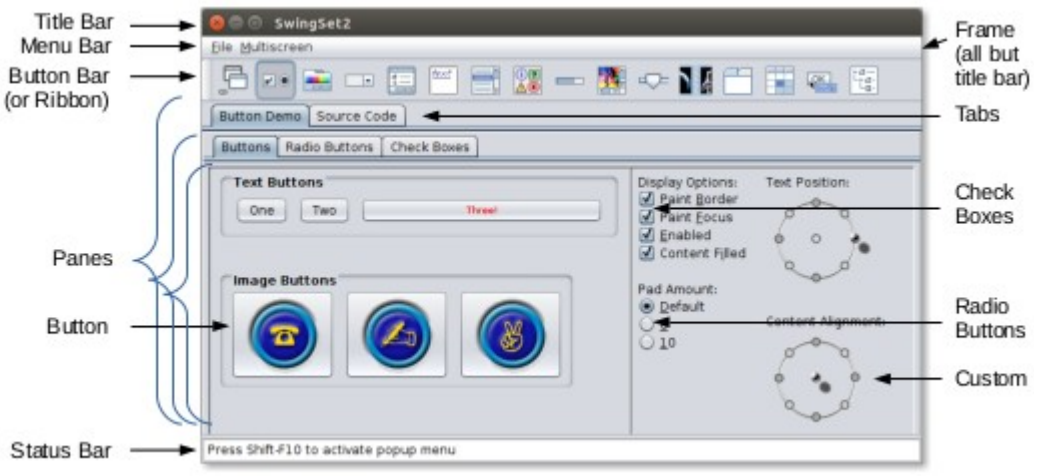
Know how to create and join threads, get their ID, and sleep the thread. Know the idiom of creating an array of threads to manage a "thread pool". Know that the order in which threads execute is unpredictable. Know that **g++** requires the **-pthread flag** to compile and link threaded source code.

Here's an example thread implementation:

```
void thread1_body(string msg) {
  this_thread::sleep_for(chrono::milliseconds(2000));  // wait 2 seconds
  cout << "I'm a thread!" << endl << msg << endl;  // print a message
}
int main() {
  thread thread1(thread1_body, "Hello, thread!"); // Thread1 is now running
  cout << thread1.get_id() << endl; // Print unique thread ID
  thread1.join(); // wait for thread1 to exit
}
```

GUI Concepts

Know the common widgets by name (gtkmm calls the Button Bar a "Tool Bar"):



In a Command Line Interface (CLI), the program is in control and queries for information.
In a Graphical User Interface (GUI), the user is in control and the program reacts to their actions.
Common C++ GUI toolkits include Qt, GTK+ (and gtkmm), wxWidgets, and FLTK.
The general pattern for a GUI main() is (1) create a (not yet visible) window, (2) create and add widgets, (3) set widget properties, (4) set widget callbacks, (5) display the window, and (6) transfer control to the GUI main loop.
Be able to explain the trade offs between scaling / translation in the constructor and in draw.
The "dirty bit" records if existing the program would lose data or not, so the user can be warned.

gtkmm Knowledge

Be able to fill in the blanks in a basic Nim-like program with the correct gtkmm-related method calls, data structures, and supporting concepts given the relevant pages from gnome.org.

Every gtkmm program must include <gtkmm.h>. To compile and link a gtkmm program, you must add a backtick expression to the g++ command line (`/usr/bin/pkg-config gtkmm-3.0 --cflags –libs` - you should recognize this, but do NOT need to memorize it!).

Be familiar with the following widgets. If asked to write code using them you will find a copy of the relevant docs from the reference manual (https://developer.gnome.org/gtkmm/stable/) for them at the end of the exam:

Widgets:

Window and **Dialog**, and how to initialize and use. The pre-defined Dialog derivative classes.

HBox, VBox, ButtonBox, Toolbar, and Grid, and how to use them to control widget layouts.

Menubar and Toolbar, how to add menu items / buttons, and how to set up callbacks.

Button, Entry, ComboBox, and other widgets that we used in class.

DrawingArea (for drawing shapes) with methods move_to, draw_to, etc. for creating an image. Also know that you must override its **on_draw** method to implement your drawing. The OS calls on_draw, not you, though you can call **queue_draw** to request a screen update.

Be familiar with how to derive from and extend Gtk::Window to create your main window.

Callbacks are registered with widgets via the signal_connect **method**, passing the callback function.

Scrum Concepts and Practices

Scrum can manage any project that delivers a product, not just software.

Scrum focuses on **4 primary artifacts**: feature backlog, sprint (or task) backlog, burn down charts, and the task board. Understand what these are, particularly the first 3 (as we use them with projects in class)

At a minimum, Scrum requires **3 primary roles**: Product Owner (or Customer), Scrum Master, and Team Member. More roles exist on much larger projects, including (of course) Architect.

General Software Development Knowledge

Basic command line concepts such as command line flags, redirection (< and >) and pipes (|)

Basic debugger concepts such as breakpoints, watchpoints, step into, step over and how they are implemented in ddd

Basic version control concepts such as repositories (“init”), staging (“add” files), commits, and checkouts, and how they are implemented using the git command line

The basics of a simple Makefile, including rule names, dependencies, and (tab-indented) commands that will bring a rule current

Basic Scrum process flow such as creating and maintaining a feature backlog for the project, sprint backlog (task list) for each sprint, and burn down charts for both. Explain the basic concepts and roles (Customer, Scrum Master, and Developer).

Library classes, functions, and macros:

Be able to use these common C++ library members *without referencing the documentation.*
Be able to employ other library members when given documentation from cplusplus.com.

cin (from iostream) – read from STDIN *to* next whitespace, use **`cin.ignore()`** to skip next character or **`cin.ignore('\n')`** to skip *through* the next newline

getline(cin, *string_var*) (from string) – stores a line of text in *string_var*, discarding the newline

cout (from iostream) – write to STDOUT

cerr (from iostream) – write to STDERR

string (from string) **– `string s;`** to instance, **`s.append(t)`** or **`s += t`** to append t, **`s[x]`** to access char number x, **`s.size()`** to return number of chars in the vector, **`for(char c: s)`** to iterate

vector (from vector) **– `vector<type> v;`** to instance, **`v.push_back(t)`** to append t, **`v[x]`** to access element number x (just like an array), **`v.size()`** to return number of elements in the vector, **`for(auto e: v)`** to iterate through v's elements

assert (from cassert) **– `assert(Boolean)`** takes no action if true, and aborts with a message if false

UML Diagrams

Understand the use and presentation of UML extension mechanisms:

1. **Stereotype** – guillemets-enclosed **specialization**, e.g., «implements»
2. **Tag** – curly-brace-enclosed **addition**, e.g., {tag = value}
3. **Constraint** – curly-brace-enclosed **condition, restriction**, or **assertion**, e.g., {var < 16}

Understand and be able to draw the basic forms of the following UML diagrams:

1. **Top Level Diagram –** NOT UML, but often serves as a graphical index to other UML diagrams
2. **Class diagram** – Representing **class** names, **fields**, **methods**, **extensions**, and the various **relationships** between classes.
3. **Use Case diagram** – Representing **scenarios** as "action" classes (with verb-centric names) within the **system boundary** (and **actors** outside the system boundaries), relationship between actor and use case scenario, include and extend dependency, and how to read the diagram out loud ("The *actor* uses *system name* to *scenario*"). Each use case may be further documented using (1) text (2) Activity Diagram (3) Sequence Diagram (4) State Diagram (not yet covered)
4. **Activity diagram –** Displays a **sequence of activities** at the algorithm level, similar to a classic "flow chart" or "data flow diagram". Supports a single **launch point** ("dot"), arrows for **flow direction**, **branches** (decisions) with **guards** (e.g., [valid_order]), multiple execution paths (**fork** and **join**), **hierarchical** activity bubbles to an arbitrary depth, **interrupts** within **regions**, object definition as data payloads, **swim lanes** for aggregating a subset of activities (similar to tags, but assigned visually), and multiple **termination points** ("targets").
5. **Sequence diagram** (not new, but covered on this exam) – Representing a specific thread of execution through the system, with object lifelines indicating period of existence, occurrences indicating method execution within a timeline, and activation indicating initiation of processing between object timelines. Also represent termination of an object timeline (X) and return of activation (dashed arrow). While the Activity Diagram depicts an entire algorithm, the Sequence Diagram depicts a since thread through that algorithm.
6. **State Diagram –** Documents the **states**, permissible **transitions**, and **activities** of a system. Though a superset of Activity Diagrams, State Diagrams usually operate via a clock. They offer a single launch point, arrows for **state transition** direction, branches (decisions) with **guards**, multiple execution paths (**fork** and **join**), Harel-type **hierarchical** state bubbles to an arbitrary depth, and both **Mealy** and **Moore** semantics.

7. **Deployment Diagram –** Models the physical deployment of **artifacts** onto **nodes.** An **artifact** is a deployable product of the software development process. The **node** is either **physical** (a piece of hardware) or **logical** (an execution environment such as a virtual machine). Deployment diagrams come in 4 basic types:
    1. **Artifact Implementation** – Models how the individual results of the builds (the components such as EXE, images, etc.) are assembled into artifacts.
    2. **Spec-Level Deployment** – The artifacts built per the Artifact Implementation diagram are then allocated to logical and physical node classes in the Spec-Level Deployment diagram.
    3. **Instance-Level Deployment** – This is very much like the Spec-Level Deployment diagram, except that the artifacts and nodes have *specific names*; they are instances, not classes.
    4. **Network Architecture** – This diagram models the communications networks of the host system at the instance-level, that is, with specific device names. Often the objects themselves are visually stereotyped, such that the servers, routers, and such take on their real-world appearance, or the firewalls look like walls rather than objects.

Software Design Patterns

Patterns are discovered, not invented, and validated by the community at large (meritocracy). You should be able to match the definition to the pattern, and *recognize* the pattern in UML or code, but you will NOT be asked to write a C++ class or draw the UML for a pattern on the exam. Patterns come in 4 general types:

- (C)reational – Address creating objects from classes via mechanisms more flexible than "new"
- (S)tructural – Address class and object composition, an alternative to inheritance
- (B)ehavioral – Address communication between objects
- (A)rchitectural – Address design of loosely coupled systems

Singleton (C) – Restricts a class to instancing a single object, typically to coordinate actions across a system. The C++ implementation makes the constructor private, and provides a public get_instance method in its place that returns the one private static instance of the Singleton class.

Decorator (S) - Dynamically adds new functionality to an object without altering its structure. This is different from inheritance, which statically adds functionality. Decorators can be nested as deeply as desired.

Model-View-Controller (MVC) (A) – Separates the business logic (the Model) from input (the Controller) and output (the View). This is particularly useful for providing multiple I/O options, e.g., a command line interface, a graphical user interface, and an automated test interface.

Façade (S) – Implements a simplified interface to a complex class or package of classes. A good example is Turtle Graphics, easy enough for children to use, as a simplified interface to FLTK, which is not (!).

Observer (B) – Notifies dependent objects when the observed object is modified. The dependent object registers the callback method with the observed object, which is called when the event of interest occurs to the observed object. The dependent object can also unregister when needed.

Factory (C) – Creates new objects without exposing the creation logic to the client. Thus, instead of "Foo foo{};" or "Foo* foo = new Foo{};", you would use "Foo* foo = Foo::create();" instead. Often, create()'s parameters will describe the object needed, and create() will return the object of the type that best fits that description.

Adapter (S) – Implements a bridge between two classes with incompatible interfaces. Typically inherit from one or the other class, or (if our language supports multiple inheritance, as C++ does) both, so that the compiler can verify the adapter's implementation of one or both interface at compile time.

Strategy (B) – Enables an algorithm behavior to be modified at runtime. This is accomplished via polymorphism with each strategy encapsulated in a derived class.

State Design (B) – Supports full encapsulation of unlimited states within a scalable context. Similar to the Strategy pattern in that it allows the behavior of the system to change depending on the currently active state. Note that event handling is NOT addressed by the State Design pattern, and must be implemented independently if required.

Anti-Patterns

Anti-patterns are errors committed during software engineering projects that are common enough to have been given well-known names. Know the six examples covered in class and how to avoid them.

- **Comment Inversion / Documentation Inversion** – Writing obvious and thus unhelpful comments and documents. Consider what the reader actually wants to know instead.
- **Error Hiding** – Catching and ignoring / obscuring an error. Only catch exceptions to which you can helpfully respond.
- **The "God Class"** - Too much functionality generalizes to the root of the class hierarchy. Use UML to identify a helpful hierarchy, or else use a single class instead.
- **The Big Ball of Mud** – A legacy system that "happened", lacking any discernable architecture. Avoid the temptation to rewrite from scratch; the code contains a lot of subtle aggregated domain knowledge that would be lost. Instead, (1) write regression tests (2) define a target architecture (3) migrate small sections in baby steps to the new architecture, using the regression tests to verify no loss of functionality (4) continue until it's "maintainable enough"
- **Not Invented Here (NIH) Syndrome** – Reimplementing an existing solution because "we can write it better". Instead, adapt the existing solution to the need (think patterns like façade and facade), addressing any real issues such as licensing, support, etc.
- **Cargo Cult Programming** – Inclusion of code, language features, data structures, or patterns without understanding them thoroughly. Instead, thoroughly understand everything that you include in your designs and implementations, learning as you go

Intellectual Property

Know the three primary types of intellectual property in the USA of primary interest to computer science and engineering professionals:

- **Trademark –** Assigned on first use of a brand (if unique) but may be registered, and is retained until abandoned. Intended to avoid confusion of brands in the marketplace.
- **Patent** – Assigned for an invention (including new computer hardware and software technologies) only after application and review, and is usually valid for 14 years. Gives the inventor the right to control the use of their invention.
- **Copyright –** Assigned when creating a work of authorship (software source code as well as literary, artistic, dramatic, etc.), but may be registered, and is retained essentially for a lifetime. Gives the author exclusive control over copying, performing, and modifying their work.

Also know the primary types of software licenses, generally from least to most restrictive:

- **Public Domain** - all ownership is disclaimed (SQLite)
- **Permissive** (MIT, BSD, Apache) permits use, copying, distribution, and (usually with attribution) derivatives, even proprietary derivatives (BSD Unix, Apache)
- **Protective** (GPL 2, 3, Lesser GPL, EPL) permits use, copying, distribution, and derivatives with share-alike rules (Linux, Gimp). GPL 3 also includes important patent clauses.
- **Shareware** permits (sometimes limited) use, copying, and (usually) distribution (Irfanview, early WinZip releases)
- **Proprietary** permits (often restricted) use (Windows, Photoshop)
- **Trade Secret** typically restricts use to the copyright holder

Also Study and Know the Quick Reviews for Each Lecture!