CS 253 Fall 2015: Things You Should Know
December 10, 2015

Disclaimer: this study guide is provided in the spirit of assisting students to organize and recall what has been covered. It is not a contract and material not appearing here may still appear on the exam.

1. All object oriented programming languages have to make a decision whether or not to permit multiple inheritance. Since C++ allows for multiple inheritance, it is subject to the diamond problem. You can now to explain that there are ways to avoid the unfortunate consequence of embedding two copies of a parent’s parent in a class.

2. As so often happens in technical enterprises, common terms such as "header" become seriously overworked. That disclaimer aside, in this class we refer to "header classes" as being objects capable of storing variable amounts of data within. You are now comfortable with all of the ins and outs of this basic concept including such details as split memory use where the header might live on the stack and the contents might live on the heap. You can write your own examples of such classes, and are now aware that indeed the standard template Library has been doing this for you in its container classes, e.g. vector.

3. Over and over in the course of teaching C++ you experience interactions between pointer variables, passed by value versus passed by reference distinctions, deep copies versus shallow copies, and indeed ultimately mysterious runtime crashes that can easily result from even simple misconception of code. Exactly such a situation is intentionally created for you in lecture 19 and, should you ever need to explain just how quickly issues of memory usage, constructors, destructors, and double destruction can cause massive failure in a compiled C++ program, you can explain the issue using this example.

4. In this semester, you have learned how to write both templated functions and templated and classes in C++. Along the way, you have developed an appreciation for how an extremely simple text substitution procedure coupled with recursive compilation gives rise to an extraordinary wealth of possibilities and complexity. Returning to simplicity, you absolutely are capable of understanding and writing straightforward examples of templates and functions and template classes.

5. In lecture 20, we had fun (or at least tried to) with the apparent contradictions that arise when you seek to clearly state that a templated function represents executable code. What the two sides of this debate teach us about how the C++ compiler deals with templates is far more important than ultimately trying to resolve the overly simplistic question "Is a template executable code?". You can now convey to others some of the most essential pros and cons in this debate, including when a compiler may or may not discover errors.

6. Some questions associated with templated functions have easy answers. For example, “What came (comes) first, the explicitly defined function or the template instantiated function?”

7. Many good C++ references warn about ambiguity in the context of templates. These warnings should be taken seriously, and should you wish to reinforce the message to a colleague, you can easily construct an example where ambiguity arises. In other words, you can jot down quickly an example where the precise choice of function template is either arbitrary, or at least so close to arbitrary, that the entire situation should be avoided.

8. One of the most common uses for templated classes is to define containers with useful functionality where the type of elements stored in the container may be supplied by the user when instantiating the template. This is a fantastic capability, but with it comes a deep reliance on the defined behavior of the elements. You are now expert enough in this usage of template classes to spot the difficulties, and state clearly what exactly must be done to avoid these problems.

9. We circled back and discussed in greater detail primitive arrays rather late in the semester. This timing is enabled a fairly detailed comparison of how to manage 1 billion random integers using primitive arrays versus Standard Template Library (STL) vectors. Further, this timing dovetailed nicely with the basic concept of STL iterators. There are a number of lessons that can be taken from this comparison including syntax of working with each variant as well as runtime consequences of different coding choices. You are fortunately now well-versed in how both approaches operate as well as in the relative pros and cons of the two approaches.

10. Here is a detail you might be asked about in a job interview. Which manner of random access into an STL vector performs bounds checking?

11. Primitive arrays are wonderful things, they allow one to access the 10th element of a 9 element array. Do you agree?

12. Iterators play an absolutely crucial role in terms of utility and the STL. For starters, you now could with no recourse to external reference construct the most basic for loop required to move through an STL container from beginning to end.

13. In lecture, the question arose "Is an iterator an object or a pointer?". The response in answer was vague in so much as both interpretations have merits and may even in some circumstances be quite literally true. More generally, this is one of
those yes/no questions that has greater value when you begin to appreciate the nuance of the question and the associated underlying issues. Fortunately, your understanding of iterators is sufficiently mature to allow you to appreciate both sides to this question.

14. It took us nearly to the end of the semester before we finally approached the STL as informed consumers and users of the various powerful container types provided. Those types explicitly presented in detail included vector, deque, forward_list, list, set and multiset. You are fortunately now familiar with the code examples employed to illustrate each and therefore you are comfortable with the syntax associated with these STL components along with their common and specialized capabilities.

15. Here are five things to consider when evaluating the utility of an STL container: 1) cost of insertion/ removal, 2) mechanisms for insertion and removal, 3) cost of random access, 4) cost of searching for a specific element, 5) ordering. Through what you have learned you can now discuss each of these five operations for vectors, deques, forward_lists, lists, sets and multisets.

16. All programming languages become immensely more powerful when they provide for the passing of functions as arguments. One of the nicest examples in C++ involves sorting STL containers. This is something that you have seen illustrated and you can reproduce to suit your own needs.

17. In advance of the final exam, you are being provided roughly 200 lines of C++ that compares overlapping word counts between documents, for example novels. As you master this code in preparation for the exam, of course much of what you will see is direct review. However, you will also see novel idioms and constructs. This is not an accident. Indeed, mastery of anything involves reaching a stage where one can continue to grow and learn on one’s own. Therefore, by the time you take the final exam, you will have reviewed every statement in the code provided and assured yourself that you understand what that particular line of code is doing. Whether you do this individually or in study groups is a matter of personal choice. However, keep in mind that often small sets of people working together make progress more quickly, particularly when piecing together extensions to what each has seen before.