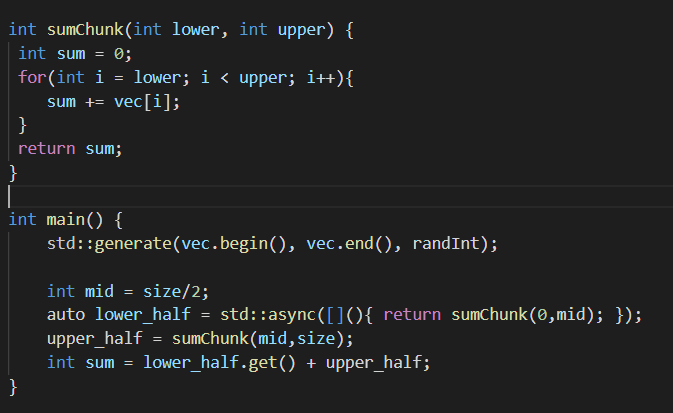
IMPORTANT: The comments sometimes contain useful information about the questions, read them to get more info. CLOSING THE DOCUMENT NOW.

1. 1. 1. int, it will take up 4 bytes. int number = 10; (it could only take 1 byte to represent)
      2. char, it will take up 1 byte. char c = ‘C’;
      3. Both are correct depending on the explanation.  
         Float-It will take up 4 bytes of memory. float f = 255.1234567f;. This is because the representation takes up 32 bits in memory which is 4 bytes, and therefore can be stored in a float. A double would take up unnecessary space.  
         Double- double f = 255.1234567;. Even if it could be barely represented as a float, it could be used in calculations with more precise numbers, in which case keeping this as a double would have a margin of error.
      4. It will take up 13 bytes of memory, as 9 bytes for the pointer (8 characters and an additional byte for the null character ‘\0’, 9 bytes total), and 4 bytes for the integer value.
   2. 1. Helper function:  
         freeNodes(struct A\* node) {  
          if(node == NULL) return;  
          freeNodes(node->right);   
          free(node);  
         }  
         21. lex = NULL; [why do we need to set lex to NULL]  
         26. freeNodes(lex);  
         ALTERNATIVE SOLUTION:  
         21. lex = newL(NULL);  
         26. freeNodes(lex);
      2. 12. \*l == NULL;   
         13. \*l = li;  
         ALTERNATIVE SOLUTION:  
         12. (\*l)->val == NULL  
         13. \*l = li;
      3. Adds values 1, 2 and 3 to the doubly linked list. The newL function creates the node for each value, and the addL function adds the newly created node to the list. The entire linked list is then printed on line 25.  
         Diagram consists of: nodes that point to each other (both in front of them and behind them, as it is a doubly linked list. The end of the list does not point to the front of the list).
2. 1. 1. For this question, Lito has said in the revision lecture that global variables and variables at the top of main should be on the heap (this is not the case in reality but in the exam it will be marked as such).

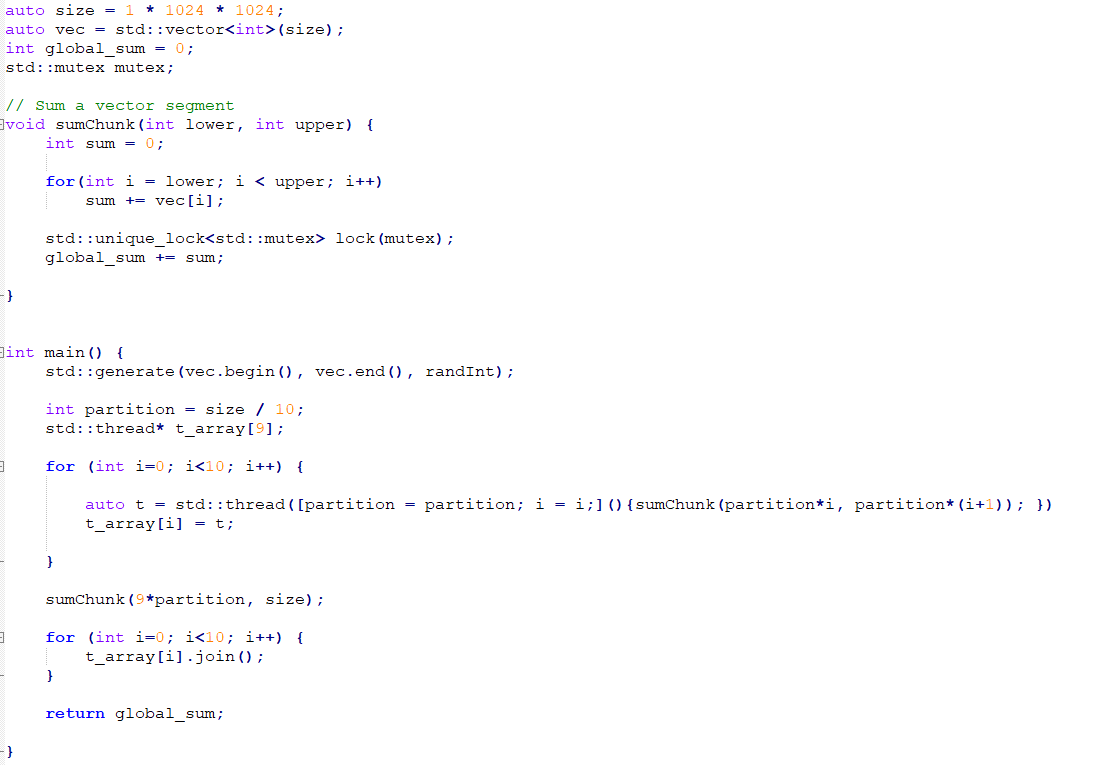
|  | Version 1 | Version 2 |
| --- | --- | --- |
| Stack | i | i |
|  | sum | j |
|  | pt | sum |
|  | n | pt |
|  | arr | n |
|  |  | s |
|  |  |  |
|  | pt | pt |
| Heap | n | n |

* + 1. Version 1 is more cache efficient because you are processing adjacent elements whereas in the second one, you have split your related elements across memory. If you look at the code, you will see that 2 different structs are used in version 2, which means they will have 2 different memory addresses in the heap. Therefore this will be less efficient as they are not neighboring. Version 1 uses the same struct type and so these will be neighboring addresses in the heap and therefore will be faster as you can access them quicker (caching).
  1. 1. Double free error, the same address is freed twice.
     2. Dangling pointer/use after free error. The pointer to the left child still exists in the root node.
     3. It is a memory leak, as the malloc’d address cannot be freed.
  2. struct file\_handle {  
      FILE \* file;   
      file\_handle(const char\* filename, char\* mode) {  
      file = fopen(filename, mode);  
      if (!file) { exit(EXIT\_FAILURE); }  
      }  
      ~file\_handle() { fclose(file); }  
     };  
     int main() {  
      file\_handle f("./file.txt", ‘r’);  
      …  
     }  
     The RAII version is better as we tie the management of a resource to the lifetime of a variable on the stack.

1. 1. Issue 1- The check to see if the key is inside the table is not protected by the lock. It should be done after the lock has been acquired.  
      Issue 2- There is no need to unlock the mutex, as C++ RAII will do this automatically for you once the function returns.
   2. struct hashTable {  
      private:  
       sem\_t count = sem\_create(1,1);  
       std::unordered\_map table;   
      public:  
       bool safe\_insert\_if\_new(char \*key, char \*value) {   
       sem\_wait(count);  
       if (table.find(key) != table.end()) { sem\_signal(count); return false; }  
       table.insert({key, value});  
       sem\_signal(count);  
       return true;  
       }  
      }
   3. Recommendation: mutex. It is about sharing a resource and not the logical order of threads.





Here is my wild guess for d ii:

Here is another version using future, I have also fixed that annoying off by one error:

