Question 1. (8 points) (debugging) One of your colleagues is trying to use `gdb` to diagnose the reason for a segfault. Unfortunately, whenever he tries to look at his program, `gdb` won't display the source code, and it doesn't display any variable names – just hexadecimal memory addresses and values.

What is likely to be the problem here? What needs to be done to get `gdb` to display source code and symbolic variable names properly?

The debugging information was not produced when the program was compiled. Recompile everything with the `-g` option for `gcc` or `g++` and things should be ok.

Question 2. (8 points) (svn) Another colleague is a bit confused about the difference between the subversion `add` and `commit` commands, since both of them seem to have something to do with putting information in the repository. What is the difference between `svn add` and `svn commit`?

`svn add` indicates to `svn` that new files in the local working directory should be added to the repository the next time updates are made to the repository, but it does not actually update the repository. `svn commit` copies local changes to the repository, including adding any new files that were designated earlier by the `svn add` command.

Question 3. (10 points) (testing) When we talked about testing, we distinguished between two kinds of tests: “white box” and “black box”.

(a) What is the main difference between these two kinds of tests?

Black box tests do not use any knowledge of the internal implementation, but only its external specification. White box tests are developed with knowledge of the implementation, and can take advantage of that knowledge to test for specific things that are not necessarily visible from outside.

(b) Applying these ideas to the memory manager (`getmem`/`freemem`) assignment, describe two “black box” tests that you could perform to test `getmem`, and two “white box” tests you could perform to test `freemem`.

Two **black box** tests for `getmem`:

Here are a few possibilities:

- **Allocate a single block of memory. Test that the result is a non-null pointer and that it is possible to write and read data from the entire area requested.**
- **Allocate several blocks of memory. Verify that the resulting pointers are all different and their addresses are such that the allocated blocks don’t overlap.**
- **Allocate an indefinite number of blocks in an attempt to exhaust all available heap storage. Check that `getmem` eventually returns `NULL`, indicating that allocation failed, and that it doesn’t crash first.**

Two **white box** tests for `freemem`:

A few possibilities:

- **Allocate several blocks of storage, then free a couple of them in an order different from that in which they were allocated. Verify that they are placed properly on the free list.**
- **Allocate several small blocks of storage, then free all of them. Verify that they are merged back into a small number of blocks or a single block, depending on how big the underlying blocks are.**
- **Allocate just one block of storage and then free it. Verify that there is a single block on the free list of the expected size.**
- **Call `freemem(NULL)` and verify that nothing happens to the free list.**

Question 4. (14 points) (trees & strings) As you may recall, a *binary search tree* (BST) is a binary tree where the values in the tree nodes are organized so that all the values in the left subtree of a node are less than the value in the node, and all the values in the right subtree are greater. For this problem, assume that we have a binary search tree whose values are C strings (or more precisely, the values in the nodes are pointers to null-terminated C strings). The tree nodes are defined by the following struct:

```
struct tnode {           /* node for a BST of strings */
  char *val;             /* string value stored in this node */
  struct tnode * left;  /* left subtree or NULL if empty    */
  struct tnode * right; /* right subtree or NULL if empty  */
};
```

Complete the definition of the following function so it returns true (1) if string `s` is contained in the binary search tree `t` and returns false (0) if `s` is not contained in tree `t`. For full credit your solution should only search subtrees that might contain the string value. You should assume that any necessary standard library headers are already `#include`d and you do not need to write any `#include`s.

```
/* return true (1) if s is contained in the bst */
/* with root t, otherwise return false (0).     */

int contains(struct tnode * t, char * s) {

  /* return false if the tree is empty */
  if (t == NULL) return 0;

  /* Compare s to contents of node t.  Return true if */
  /* found, otherwise return search of proper subtree.*/
  int cmp = strcmp(s, t->val);
  if (cmp < 0) {
    return contains(t->left, s);
  } else if (cmp > 0) {
    return contains(t->right, s);
  } else /* cmp == 0 */
    return 1;
}
```

Many solutions evaluated `strcmp` several times for the same two strings instead of storing the int result and testing that a couple of times. While this isn't too expensive, and we didn't deduct points for it, it is redundant and better avoided.

Question 5. (12 points) (make) Suppose we have the following collection of C header and implementation files.

```
--------------------
thing.h
--------------------
#ifndef THING_H
#define THING_H

#include "gadget.h"
...
#endif

--------------------
gadget.h
--------------------
#ifndef GADGET_H
#define GADGET _H
...
#endif
```

```
--------------------
thing.c
--------------------
#include "thing.h"
...

--------------------
gadget.c
--------------------
#include "gadget.h"
...

--------------------
widget.c
--------------------
#include "gadget.h"
#include "thing.h"

int main() { ... }
```

These source files are to be used to build an executable program file named `widget`, whose `main` function is in the source file `widget.c` and which uses all of the functions defined in all of the above source files.

Answer the questions on the next page using the above information. You may remove this page from the exam if it makes it easier to use. (Suggestion: sketch your answer to part (a) below before you make a clean copy of it on the next page.)

$$\begin{tikzcd}
\texttt{foo.c} \\
\texttt{foo.o} \arrow[u] \\
\texttt{foo} \arrow[u]
\end{tikzcd}$$

Question 5. (cont.) (a) Recall that we can specify the dependencies between files in a program using a graph, where there is an arrow drawn from each file name to the file(s) it depends on. For example, the drawing to the left shows how we would diagram an executable program named `foo` that depends on (is built from) `foo.o`, which in turn is depends on `foo.c`.

In the space below, draw a graph (diagram) showing the dependencies between the executable program `widget` and all of the source (`.c`), header (`.h`), and compiled (`.o`) files involved in building it from the files on the previous page.

$$\begin{tikzcd}
\texttt{thing.c} & \texttt{thing.h} & \texttt{widget.c} & \texttt{gadget.h} & \texttt{gadget.c} \\
& \texttt{thing.o} \arrow[ul] \arrow[u] \arrow[urr] & \texttt{widget.o} \arrow[ul] \arrow[u] \arrow[ur] & \texttt{gadget.o} \arrow[u] \arrow[ur] & \\
& & \texttt{widget} \arrow[ul] \arrow[u] \arrow[ur] & &
\end{tikzcd}$$

(b) Write the contents of a `Makefile` whose default target builds the program `widget`, and which only recompiles individual files as needed. Your `Makefile` should reflect the dependency graph you drew in part (a).

```
widget: widget.o thing.o gadget.o
     gcc –Wall –g –o widget widget.o thing.o gadget.o

widget.o: widget.c gadget.h thing.h
     gcc –Wall –g –c widget.c

thing.o: thing.c thing.h gadget.h
     gcc –Wall –g –c thing.c

gadget.o: gadget.c gadget.h
     gcc –Wall –g –c gadget.c
```

Question 6. (12 points) (memory management) While working on the memory manager assignment, several groups ran into problems with free lists that somehow became circular when they shouldn't have been. We'd like to implement a function that could be used to help detect these problems.

For this problem, assume that the following `struct` defines the layout of header part of each free list node.

```
struct free_node {
   int size;              /* number of bytes in this node, */
                          /*   including this header       */
   struct free_node *next; /* next node on the free list,   */
                          /*   or NULL if no more nodes.   */
};
```

In a properly formed free list, the successive nodes should occupy increasing memory addresses. If some node on the list has a `next` pointer that that is not `NULL` and contains a lower memory address, then something is wrong with the list. At a minimum the nodes are out of order, but it also may be that the list is circular.

Complete the definition of function `looks_ok` on the next page so that it returns true (1) if the successive nodes on list `p` are stored at increasing addresses, and returns false (0) if some node has a successor with an address that is less than the address of the node itself.

You should assume that any necessary standard library headers are already `#included` and you do not need to write any `#includes`.

(write your code on the next page)

Question 6. (cont.) Free list node definition repeated for reference.

```
struct free_node {
   int size;                 /* number of bytes in this node, */
                             /*   including this header       */
   struct free_node *next; /* next node on the free list,   */
                             /*   or NULL if no more nodes.   */
};
```

Write your code for this function below.

```
/* return true (1) if the node addresses on list p are */
/* strictly increasing, otherwise return false (0)     */
int looks_ok(struct free_node *p) {

   /* An empty free list is ok */
   if (p == NULL) return 1;

   /* Free list contains at least 1 node.  Check node  */
   /* p against its successor as long as there is one. */
   while (p->next != NULL) {
      if ((int)p >= (int)(p->next)){
         /* successor has smaller address – not ok */
         return 0;
      }
      p = p->next;
   }
   /* p->next == NULL here, so end of list reached */
   /* without detecting an error. */
   return 1;
}
```

Notes: A more robust implementation would cast the pointers to type `size_t` or some other appropriate unsigned integer type to avoid problems with very large addresses that would appear negative if treated as ordinary integers. We didn't expect to see this since it is something we glossed over in the memory manager assignment.

The question turned out to be a bit ambiguous since it was not clear if a block that pointed to itself should be reported as an error. In grading we allowed either strictly greater or >= without deducting anything. (In a real implementation you'd want to check that the addresses were strictly increasing to make it more useful for detecting bugs.)

Several people wrote recursive solutions. While this is not wrong, in most C implementations it takes space proportional to the length of the list, while a simple loop takes constant space.

Question 7. (8 points) One of the ways we could use the `looks_ok` function from the previous question to test our memory manager code would be to use it in `assert` macros to check the condition of the free list at strategic points. Describe where you would place these `assert` checks in the memory manager code to try to detect a free-list bug as soon as possible, without adding too many extraneous checks that would simply consume time without being likely to find anything useful.

For starters, place `assert`s right before each `return` statement in `getmem` and `freemem` to verify that the list is ok at the end of each execution of these functions. It might also be useful to place `assert`s at the end of any other functions that splice the list to insert or remove nodes, particularly if other processing occurs before `getmem` and `freemem` return. A good example of this would be at the end of the routine that gets a large block from `malloc` and puts it on the freelist.

Question 8. (8 points) C++ allows member functions of a class to be declared `virtual`, meaning that dynamic dispatch is used to select the actual function to be executed depending on the runtime type of an object. But it also provides non-virtual functions where the code to execute for a function call is selected statically at compile time. By contrast, in Java and some other languages all function calls are virtual.

Give one technical reason why it can be useful to have non-virtual functions in an object-oriented language and why this option is available in languages like C++.

There are at least two good technical reasons:

- **A class designer might want to ensure that functions in the class only call other, known functions in the same class, and that this can't change even if someone extends the class later with a new subclass.**
- **If no functions in a class are virtual, then there is no need to set up a vtable for the class or, more significantly, include vtable pointers in instances of the class (objects). That saves space and is particularly useful for small objects like rational or complex numbers where a function vtable pointer could add a noticeable amount to the size of each object, particularly if we have large arrays of them. Even if only some functions are non-virtual, there is a slight reduction in the overhead needed to call those functions, since the call does not need to be done indirectly through the vtable.**

Question 9. (12 points) The ~~dreaded~~ traditional C++ “what does this print” question.

What output is produced when this program is executed? It does compile and execute with no warnings or errors using g++.

```
#include <iostream>
using namespace std;

class A {
public:
  virtual void y()   { z(); cout << "A::y" << endl; }
          void z()   {      cout << "A::z" << endl; }
};

class B : public A {
public:
  void y() { z(); cout << "B::y" << endl; }
  void z() {      cout << "B::z" << endl; }
};

class C: public B {

};

int main() {
  B* b = new B();
  b->y();
  b->z();

  cout << "----" << endl;

  A* a = new B();
  a->y();
  a->z();
  return 0;
}
```

Output:

```
     B::z
     B::y
     B::z
     ----
     B::z
     B::y
     A::z
```

Question 10. (8 points) (concurrency) Problems with electronic voting machines have been much in the news. Because of your expertise in C code and concurrency, you have been asked to look at the secret source code for the Democracy 'R Us ®©™ voting machines to see if you can figure out why they don't always work.

The voting machines are set up on a network, and the individual machines communicate with a central server that adds up the votes. There is a separate process (or thread) on the server for each of the voting machines, and when a vote is cast, the appropriate server process adds one vote to the selected candidate's total by calling this function:

```
/* record a vote for the specified candidate.     */
/* vote_b and vote_g are shared, global variables */
void tally_vote(int candidate) {
   switch (candidate) {
      case BUSH:
               vote_b = vote_b + 1;
               break;
      case GORE:
               vote_g = vote_g + 1;
               break;
   }
}
```

(a) What could cause the vote totals to be incorrect if this function is executed by more than one process at the same time? How might the results be incorrect? (Hint: to compute `x=x+1`, we have to read the old value of `x`, add 1, and store the new value back in `x`, and each of these operations requires executing a separate machine instruction.)

Trouble can occur if one process accesses a variable while some other process is updating it. For instance, suppose two processes are tallying votes for `GORE` and both of them fetch `vote_g` simultaneously, then both increment it, then both store the updated value. If the interleaving is just wrong they could both fetch the same old value, add 1 to it, then store that value back, leaving the total increased by only 1 instead of 2.

(b) How could you fix the code so that the problem(s) you described in part (a) would not occur in the future?

Ensure that the individual `vote_x = vote_x + 1` statements cannot be interrupted by another process either reading or writing the same variable. If the "atomic" statement described in class were available, we could add `atomic { vote_x = vote_x + 1; }` around each assignment. Without `atomic`, we would have to use some sort of lock protocol to ensure exclusive access to each variable while it is being updated.