Memory Leaks/delete Mistakes

Manually managing heap memory using `new` and `delete` is one of the most error-prone parts of writing a C++ program. Memory leaks are one of the most common programming errors—it’s easy to accidentally let a memory leak slip into your code, because unlike other kinds of errors they rarely (immediately) result in a program crash or incorrect output. And once arrays get introduced into the picture, you have to make sure that when you delete a variable you use the correct kind of `delete` on every variable you’re done with, too!

As you can tell, there are lots of places that you can make small mistakes when dealing with memory management. Below are three small pieces of code that allocate and then free memory—but each one has at least one mistake in it. As practice, you’re going to have to find out what’s wrong with them.

For each snippet of code below explain precisely where the memory isn’t being allocated/freed correctly, what mistake the code’s author made, and what you could do to solve the problem.

a)  
```cpp
int *list = new int[10];
//use 'list'...
for (int i = 0; i < 10; i++) {
    delete &list[i];
}
```

b)  
```cpp
int *list = new int[10];
//use 'list'...
delete list;
```

c)  
```cpp
int *list;
for (int i = 0; i < 5; i++) {
    list = new int[1];
}
//use data...
for (int i = 0; i < 5; i++) {
    delete[] list;
}
```

d)  
```cpp
int **head = new int*[3];
for (int i = 0; i < 3; i++) {
    head[i] = new int[3];
}
//use_data(head);
delete[] head;
for (int i = 0; i < 3; i++) {
    delete[] head[i];
}
```

Working With Command Line Arguments

In assignment #6, your program will have to process command-line arguments from the user. That means that when the user runs your program like this:

> ./minesweeper -r 10 -c 7 -m 12

Your program should automatically start a game of minesweeper on a board with ten rows, seven columns, and twelve mines, all without asking for any input from the user!

To get variables representing the command line arguments into your program, all you have to do is change your `main` function definition to look like this:

```cpp
int main(int argc, char **argv) {
```

The `argc` variable stands for argument count, and it represents the number of arguments passed in to the program. `argv` stands for argument value, and it contains each individual string provided as an argument when running the program. Note that putting a dash at the front of an argument doesn’t make it special—"-r" is a string in `argv` just like the "10" after it is, in the example above.

The first thing to notice when learning about command line arguments is the type of `argv`. Sometimes you’ll see it written as `char *argv[]` instead, but both methods mean the same thing. **What is the meaning of the type of argv—what kind of data does it hold, and how? And, why can its type be written two different ways?**

Once you feel like you have an understanding of how to get command line parameters out of `argv` (writing a simple program to experiment is a good way to get practice with it), for the last part of this exercise come up with a brief design for how you will process command line arguments like the example above for the next assignment. Remember that the options (r, c, and m) can be given in any order, so you can’t just rely on certain values being in certain places!

For **take-home exercises completed in peer-led groups**, each student must participate in the class discussion and write answers to each of the questions on his/her own paper to show for credit.

For **take-home exercises completed on your own**, turn in your work electronically using the TEACH website.