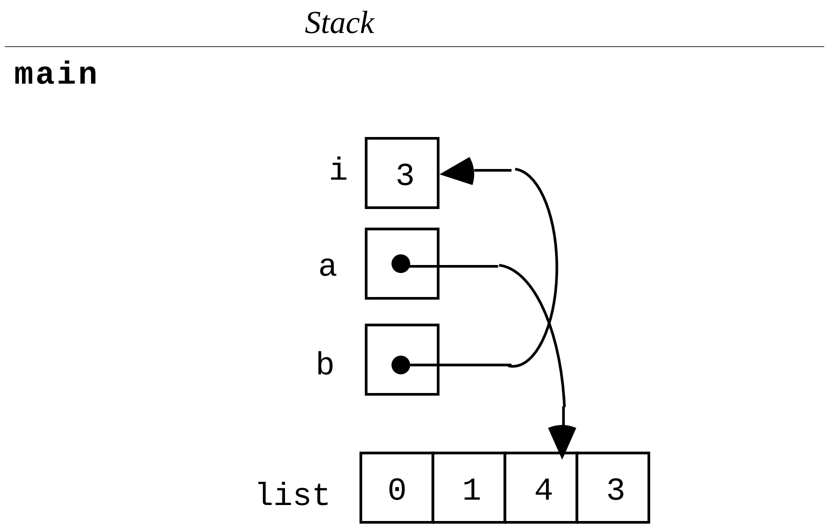
CS106X
Spring 99-00

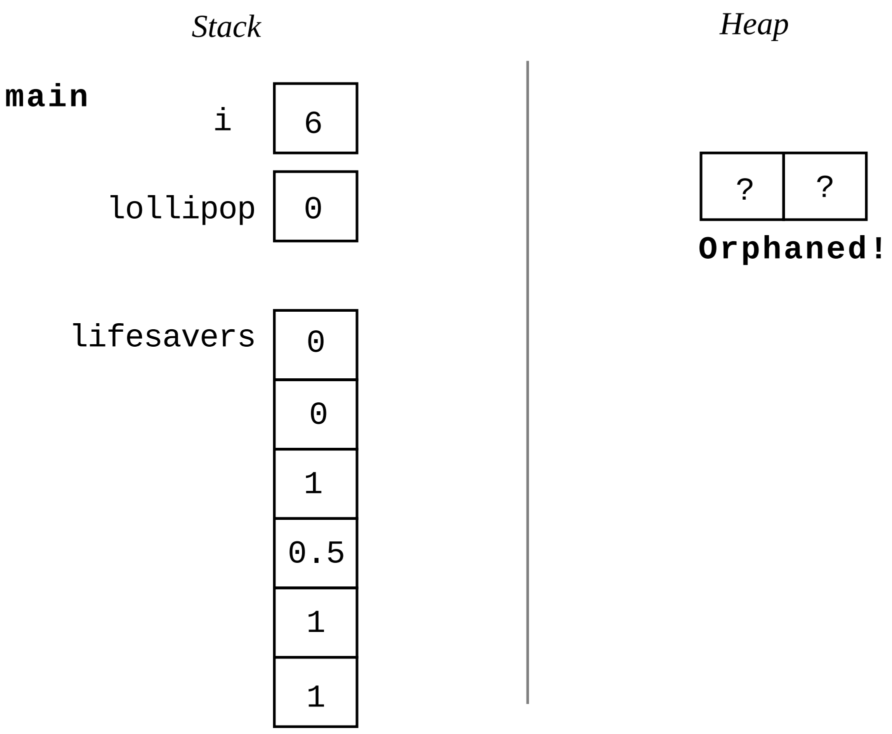
Handout #23
April 24, 2000

Assignment #3 solutions

Problem 1
a)

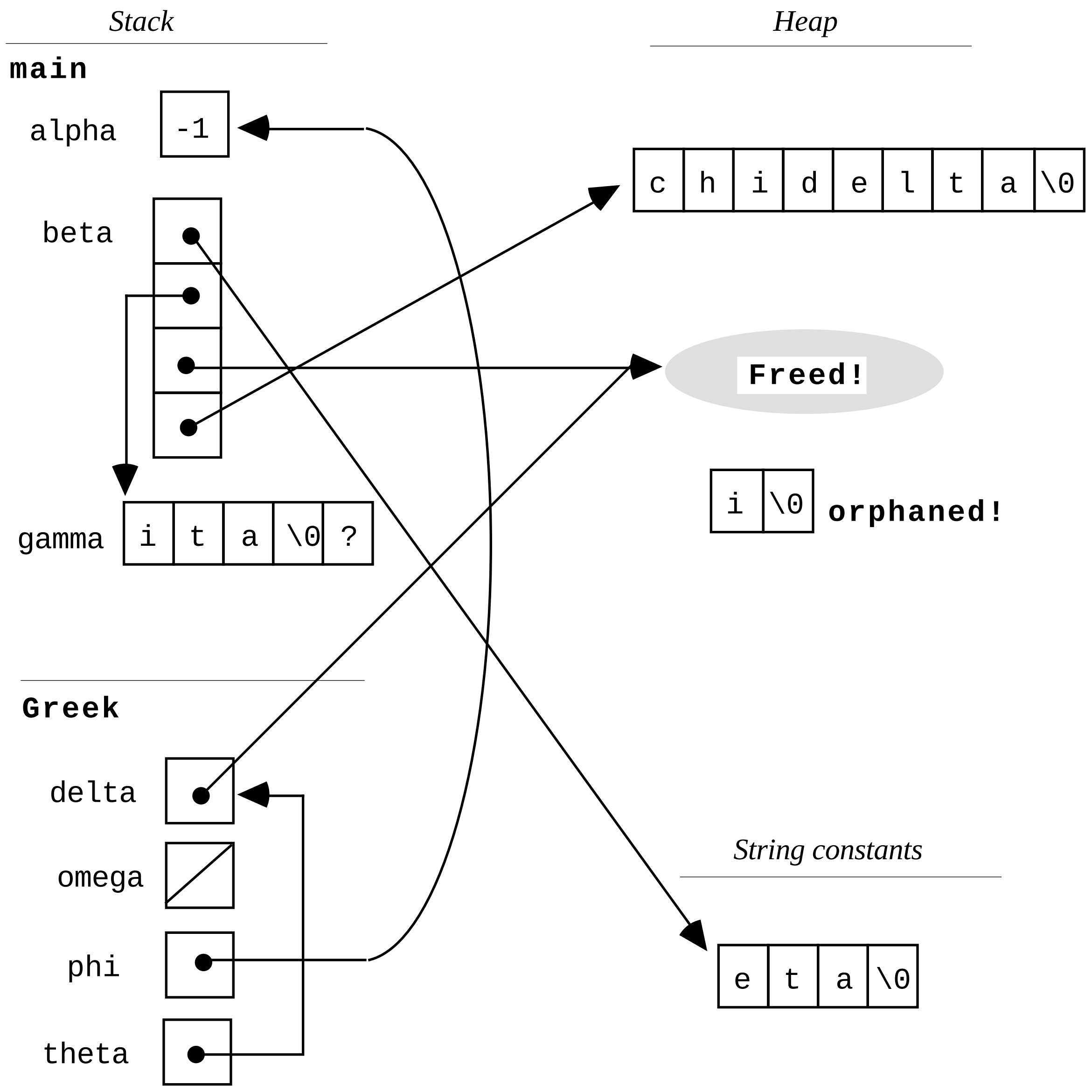


b)



Note the local variables and parameters of JollyRancher are deallocated after the function exits. The array of 2 doubles is orphaned in the JollyRancher function, which dynamically allocated that array and left the values uninitialized. The only pointer to that memory was the parameter `melon`, which was a local copy of a pointer variable, which was later assigned to point elsewhere. When that function exits, the stack memory (i.e. local variables and parameters) is automatically deallocated, but the memory in the heap remains with no way of accessing it.

c)



My, what a mess! If you can trace through this correctly, you're doing well!

Problem 2

a) Remember that `string` is just a synonym for `char *`, thus the declaration:

```
string result;
```

allocates space in the stack for just one pointer variable. It does not allocate storage for any characters pointed to by the pointer. In fact, the code never sets `result` to point to a valid memory location at all, and thus the pointer is uninitialized and contains a garbage value. When the loop assigns to locations such as `result[3]`, it follows that garbage pointer and writes into the third location past it. What a mess!

We need to allcoate storage for the result string. This storage must be allocated from the heap—we cannot use the stack since we are going to be returning a pointer to that storage from this function. Adding this line at the beginning of the function:

```
result = (string)GetBlock(strlen(str) + 1); // add one for NULL char
```

The second mistake is that we never null-terminate the new result string, so need to add a final line to write a null character into the last position before we return it.

```
result[i] = '\0';
```

b) One bug is fairly simple to spot, which is the inccorect size requested in the GetBlock call. We need to specify the size in bytes which requires multiplying count by sizeof(int). The second problem is more difficult to detect and it comes from a subtle issue with pointers and parameters. When you pass a parameter to a function, the new function gets a *copy* of the variable you passed in and changes made only affect the copy. When we send a pointer to a function, the function receives a copy of that pointer and it can follow the pointer to that location and make visible changes to that location. But the pointer itself cannot be changed to point somewhere new unless the pointer itself is passed by reference.

Follow the code: When we start out in main, **`randomNumbers`** is an uninitialized pointer. We pass the pointer into **`GetRandomArray`**, so the function makes a copy of that pointer in the parameter **`array`**. (**`array`** and **`randomNumbers`** are *aliases* for the same location.) The function then changes its copy of **`array`** to point to a new block of memory by calling **`GetBlock`**. The variable `array` now points to this new piece of memory, but the original pointer **`randomNumbers`** is not changed by this, the copy is distinct from the original. When **`GetRandomArray`** ends, the parameter `array` goes away and the original pointer in main is unchanged, still pointing off to the same bogus address. After the function call, **`randomNumbers`** does not have the value we expect and in addition, we've orphaned the memory allocated since with no pointer pointing to it, we can never get back to it.

The way to fix that is the same way we write any function that needs call by reference. When we wanted the values to change, we passed pointers to those values. Now we want the value of a pointer itself to change, we pass a pointer to the pointer. Here's the corrected code:

```
main()
{
   int count, *randomNumbers;

   count = GetRandomArray(&randomNumbers);
   ...
}

int GetRandomArray(int **array) // pointer to a pointer to int
{
   int i, count;

   printf("How many random numbers do you need? ");
   count = GetInteger();
   *array = (int *)GetBlock(count * sizeof(int));
   for (i = 0; i < count; i++)
       (*array)[i] = RandomInteger(1, 100);
   return count;
}
```

c) This bug was very subtle, but one I have seen more than once in student programs. Look carefully at the line which allocates the memory for the result string, in particular, pay attention the placement of the parentheses:

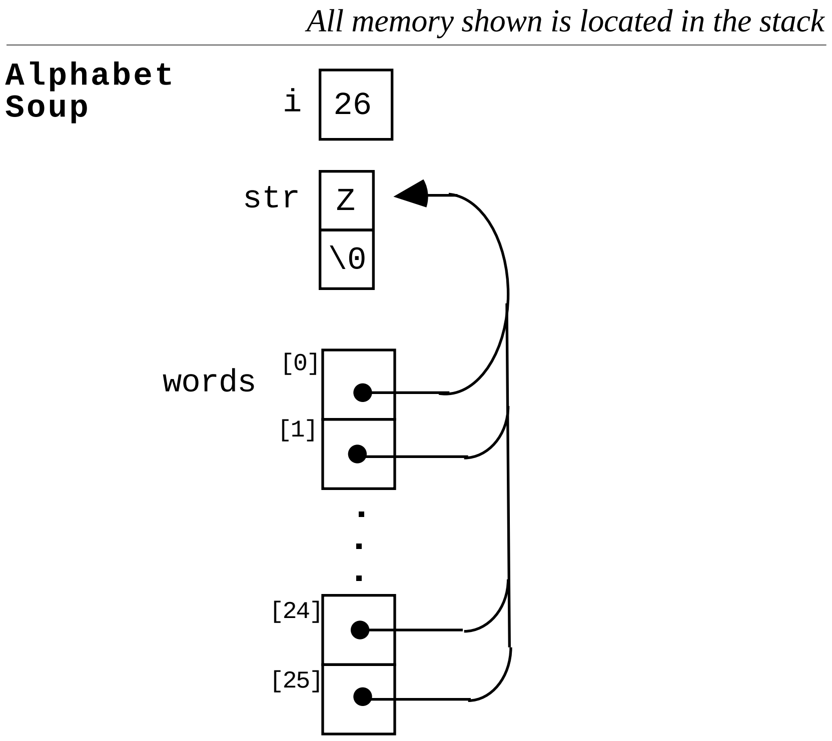
```
start = GetBlock(strlen(str + 1));
```

Rather than counting the characters in the string and adding one to that count, it first *adds one to the pointer*, then counts the characters in that string. Which means it will end up under-allocating by 2 bytes every time. The fix is simply to move the close parenthesis over:

```
start = GetBlock(strlen(str) + 1);
```

This is an example of an insidious class of bugs. The code compiles cleanly and "looks right" because it has the right elements, the GetBlock use, the strlen call, the +1, etc. However, it is evilly wrong because it will sometimes work and other times fail (depending on what nearby data gets overwritten) and will be very difficult to track down experimentally. Ouch!

Problem 3



As you can see from the drawing, the function does not do its job well at all. Each time through the loop it puts the ith character of the alphabet in the first slot of `str` and then assigns `words[i]` to point to `str`. There is just one copy of `str`, it is on the stack, and the characters in the buffer are overwritten each time through the loop. So by the end of the loop, every entry in the `words` array points to the same buffer, and the buffer contents is "Z". If we really wanted 26 alphabet strings, we need to allocate space for each string separately!

As for freeing the strings, not only is it unnecessary, it is wrong. First of all, the loop will try to free the same string 26 times. Once you free a pointer, it becomes invalid and trying to do anything with that pointer, such as calling `FreeBlock` on it again, is unwise. More importantly, we shouldn't be freeing the string at all, because we did not allocate the string from the heap. The storage for `str` was allocated on the stack and will be deallocated automatically when the function exits. Never attempt to free a pointer which points to storage on the stack! An easy rule to follow: If you didn't assign the pointer to the result of a `GetBlock` call, don't call `FreeBlock` on the pointer.

Problem 4

```
FreeBlock(array[1]);
FreeBlock(array[5]);
FreeBlock(array);
FreeBlock(*ip);
FreeBlock(ip);
```

You only free memory that came from the heap. Thus, there should basically be a one-to-one correspondence between `GetBlock` and `FreeBlock` calls. (Note the string library functions do a `GetBlock` for you, so also consider those heap allocator functions) Things to avoid: don't free constant strings, don't free memory on the stack, don't free a pointer more than once. Also you need to free things in the correct order. If you free `ip` and then attempt to free what `ip` points to, you will be accessing memory after it has been deallocated, which is an incorrect and dangerous practice.

Problem 5
a)

```
static string RemoveOccurrences(string str, char ch)
{
   string result, oldResult, littleString;
   int i;

   result = CopyString("");
   for (i = 0; i < StringLength(str); i++) {
      if (IthChar(str, i) != ch) { // add if char isn't to be removed
         oldResult = result;
         littleString = CharToString(IthChar(str, i));
         result = Concat(oldResult, littleString);
         FreeBlock(oldResult);
         FreeBlock(littleString);
      }
   }
   return result;
}
```

Note that we make a heap-allocated copy of the empty string to start. As you will remember from class, string constants live in a special section of memory that is distinct from the heap. We do not free things there. Without making a copy of it to start, we will attempt to free that string constant and get unpredictable results. If you didn't allocate it (or ask the string library to allocate it), don't free it!

b) There are some tricky things to watch for with this problem. First, we need to be sure to allocate space in the heap to build the result string into. We allocate space for the full size of the incoming string (including one extra byte for the null character) although we may use a bit less, that's fine.

We will walk the `str` pointer down the incoming string in a loop (note the `str++` used as the increment step) and continue until we reach the null char at the end. In the loop body, we check to see if `str` now points to a character that should be added to the result string. If so, we copy the character to the result string at the current position. The current position is maintained by a pointer which points to the next unassigned spot in the result string. After assigning a character, we increment the current pointer to get ready for the next character to be appended.

After the loop finishes, we need to copy the null character to the current position to mark the string end. And then we need to be sure to return a pointer to the **start** of the result string, which we saved in the variable `start` before entering the loop. If we just returned `current`, we would erroneously be returning the emtpy string, because `current` is pointing to the end of the string (the null character).

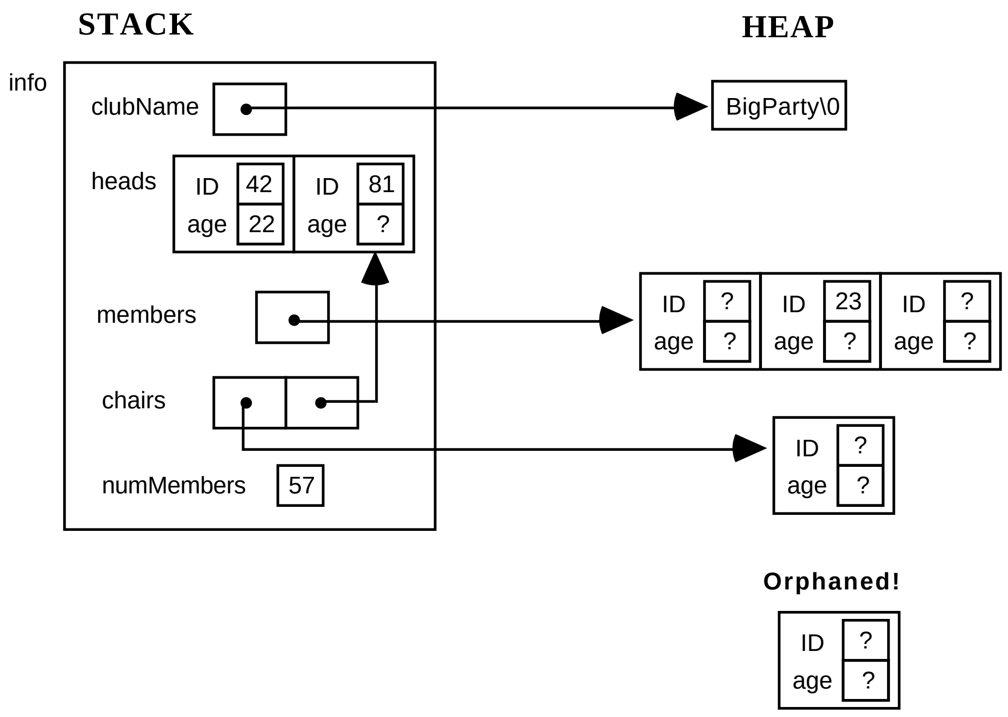
```
static char *RemoveOccurrences(char *str, char ch)
{
   char *current, *start;

   start  = (char *)GetBlock(strlen(str) + 1);
   current = start;
   for (; *str != '\0'; str++) {
      if (*str != ch)
         *current++ = *str;
   }
   *current = '\0';
   return start;
}
```

c) The very first version (the original part (a) complete with memory leaks) is probably the easiest to read and understand, so if you had no qualms about its performance, that would probably be the best choice. The fixed version that plugs those leaks ends up being a bit messier, but still not so bad. The pointer version, on the other hand, is an example of the kind of code you should run screaming from. It's important that it is completely correct and efficient about its use of memory, but trying to make changes or fix bugs in this sort of code is unpleasant and error-prone. At the very least, accessing the strings by array subscripting would be a big benefit to its readability.

Problem 6

There are a few things to notice and make sure you understand. First off, take a look at the three various Students declarations within the Club. `heads` is a fixed array of Student structures, `members` is a pointer to an array of Student structures, and `chairs` is a fixed array of pointers to Student structures.

Note that when we make the call to `UpsetClub`, the ClubInfo structure was passed *by value,* not *by reference.* Everything in the structure got copied into a new, distinct structure. So assignments to the struct fields themselves (clubName, heads, chairs, members, etc.) will only change in the structure copy, not the original, and when the function exits, those changes are lost. However, some fields in the structure are pointers, and thus we have copies of those pointers, giving us aliases to the same locations. When we follow those pointers and write to those locations, changes *will be seen.* For example, we will see the effects of writing to the members[1] ID field or the age field of info.chairs[0]. The orphaned Student struct was created when we assigned the member field using a GetBlock call. When the function exited, we lost the only pointer we had to that location.



"Stocks have reached what looks like a permanently high plateau." — Irving Fisher, Professor of Economics, Yale University, mid-1929.