

Memory Management and Flex Demo

1 Memory Management Issues with Flex

Proper use of the Flex module provided for use in problem set 4 requires careful attention to issues of memory management. We describe them in some detail below.

1.1 How does Flex use dynamic memory?

The Flex data structure consists of two dynamically allocated parts:

1. An instance of type `struct flex`. This is allocated by `newFlex()`. The pointer to it of type `Flex` is returned by `newFlex()`.
2. An array of elements of type `String`. This is allocated the first time that `insertFlex()` is called and is automatically expanded as necessary by `insertFlex()`. The pointer to it of type `String*` is stored in the `slot` field of the `struct flex` instance.

The user who calls `newFlex()` owns the `Flex` instance and is responsible for calling `freeFlex()` when it is no longer needed. The `slot` array is owned by the `Flex` instance until `extractFlex()` is called. At that point, ownership is transferred to the caller, and the `slot` field is set back to `NULL`. A decision was made in the design of `Flex` that the user must assume responsibility for the `slot` array before `freeFlex()` is called. Thus, `freeFlex()` should only be called when the data structure is empty, as is the case right after `extractFlex()` is called. Otherwise, it gives the fatal error, "Attempt to free non-empty flex".

1.2 What does one do with the `String` array returned by `extractFlex()`?

`extractFlex()` returns a value of type `String*` (which is the same as type `char**`). This is a pointer to an array of strings. Each element of this array is a `String`, i.e., a pointer to another array of `char`. Figure 1 gives a picture of such a structure.

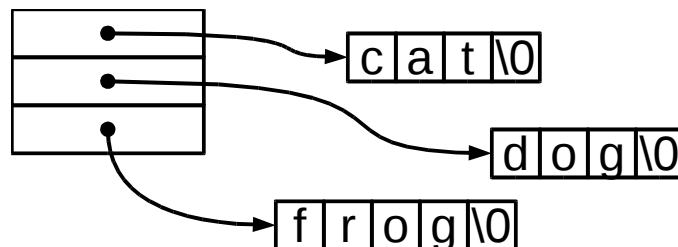


Figure 1: String array structure in memory.

Notice in this example that four blocks of memory are involved: A block of three pointers and three blocks of characters. The block of pointers was allocated by `Flex`. The blocks of characters are the user's responsibility to allocate and to later free.

2 Demonstration of How to Use Flex

A demonstration program, `demo.c` and accompanying `Makefile` have been placed in the PS4 assignment directory on the Zoo. The program is largely self-explanatory, but I want to point out in particular how the character buffers to store the individual strings get allocated and deallocated.

```

1  int fill( Flex cart ){
2      // put some things into the cart
3      char buf[100];
4      String newStr;
5      printf( "Please type a sentence "
6              "(single period or ^D ends session): " );
7      for (;;) {
8          if (scanf( "%99s", buf) == EOF ||
9              (strcmp( buf, "." )==0 && lenFlex( cart )==0) ) {
10             return EOF;
11         }
12         // copy contents of buf to new String
13         newStr = safe_malloc( strlen( buf )+1 );
14         strcpy( newStr, buf );
15         insertFlex( cart, newStr );
16         if ( buf[strlen(buf)-1] == '.' ) break;
17     }
18     return 0;
19 }
```

Figure 2: The `fill()` function.

The function `fill()` in Figure 2 reads words from the user into the buffer `buf` and puts them in the `Flex` structure `cart` (line 15). However, before inserting into the cart, a new storage block, just large enough to store the newly-read string, is allocated in line 13. Line 14 copies the characters from `buf` into the new storage area `newStr`. `strcpy()` also copies the terminating NUL byte, so nothing special needs to be done to preserve it. Line 15 puts a pointer to the new string into the slot array inside of `cart`.

(As an aside, note that the long format in the `printf()` statement that begins on line 5 has been broken into two parts and placed on separate lines. Standard C permits string constants to be broken in this way and treats the multipart string just as if it had been written as a single long string.)

The strings that were allocated in line 13 of `fill()` must eventually be freed when we are done with them. This is done in `process()`, shown in Figure 3. The `String` array that was pointed to by `slot` in `cart` is extracted from the `cart` in line 27 and a pointer to it placed in `wordList`. The data is used in lines 34–35. The remainder of the code is to free the storage that we are done with. Lines 39–40 run down the `String` array, freeing each character block in turn. Finally, line 44 frees the `String` array itself.

3 Valgrind

One of the most difficult aspects of C programming is doing memory management correctly. There are three kinds of management errors: Failing to allocate (enough) storage before it is used, freeing storage before you are done with it, and failing to free storage after it is no longer needed. The first two kinds of errors lead to faulty programs: segmentation faults or simply wrong (and oft times

```

20 void process( Flex cart )
21 {
22     // sort cart
23     sortFlex( cart );
24
25     // get its length and contents
26     int len = lenFlex( cart );
27     String* wordList = extractFlex( cart );
28     // ExtractFlex passes ownership of wordList
29     // and the strings to which its elements point.
30     // We are responsible for freeing them when we're
31     // done processing them.
32
33     // print out our word list
34     for (int k=0; k<len; k++) {
35         printf( "word[%2i]=\"%s\"\n", k, wordList[k] );
36     }
37
38     // free each of the Strings on our word list
39     for (int k=0; k<len; k++) {
40         free( wordList[k] );
41     }
42
43     // free the word list itself
44     free( wordList );
45 }

```

Figure 3: The process () function.

bizarre) behavior of the program. The third kind of error, memory leaks, do not make the program produce incorrect answers, but they consume unnecessary resources. In a large or long-running program, memory leaks can cause a program to eventually consume all of available memory, at which point it (and perhaps other applications running at the same time) fail due to insufficient memory.

My goal in this course is for all programs submitted to have no memory management errors and to have freed all dynamic memory at the time of exit. Eliminating all memory leaks is not easy to do at first, so I will be tolerant of memory leaks on PS4.

There is a tool on the Zoo called `valgrind` that is very helpful in eliminating memory leaks. To use it, just type the command `valgrind` followed by the name of your program and any arguments that it takes.

Figure 4 shows the result of running the demo program under `valgrind`. Note that the input and output for the demo program appears in the middle of the output. If there had been any memory management errors of the first two kinds, they would have been noted there. At the end, `valgrind` reports separately on memory that was never freed and is no longer accessible, and memory that was never freed but still appears to be in use. It also gives some statistics on the use of `malloc` and `free`.

Figure 5 shows what happens when `valgrind` is run on a program with a memory leak. In this case, I commented out line 44 of `process()` which is supposed to free the `String` array. Notice the report of definitely lost memory. Rerunning with the `--leak-check=full` flag as instructed gives additional output that will help one identify where in the code the lost memory was

```

> valgrind demo
==26543== Memcheck, a memory error detector.
==26543== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==26543== Using LibVEX rev 1732, a library for dynamic binary translation.
==26543== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==26543== Using valgrind-3.2.3, a dynamic binary instrumentation framework.
==26543== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==26543== For more details, rerun with: -v
==26543==
Please type a sentence (single period or ^D ends session): Here is a sentence.
word[ 0]="Here"
word[ 1]="a"
word[ 2]="is"
word[ 3]="sentence."
Please type a sentence (single period or ^D ends session): .
Goodbye!
==26543==
==26543== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 1)
==26543== malloc/free: in use at exit: 0 bytes in 0 blocks.
==26543== malloc/free: 6 allocs, 6 frees, 116 bytes allocated.
==26543== For counts of detected errors, rerun with: -v
==26543== All heap blocks were freed -- no leaks are possible.

```

Figure 4: Good output from valgrind.

```

> valgrind demo
==26576== Memcheck, a memory error detector.
==26576== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==26576== Using LibVEX rev 1732, a library for dynamic binary translation.
==26576== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==26576== Using valgrind-3.2.3, a dynamic binary instrumentation framework.
==26576== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==26576== For more details, rerun with: -v
==26576==
Please type a sentence (single period or ^D ends session): Here is a sentence.
word[ 0]="Here"
word[ 1]="a"
word[ 2]="is"
word[ 3]="sentence."
Please type a sentence (single period or ^D ends session): .
Goodbye!
==26576==
==26576== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 1)
==26576== malloc/free: in use at exit: 80 bytes in 1 blocks.
==26576== malloc/free: 6 allocs, 5 frees, 116 bytes allocated.
==26576== For counts of detected errors, rerun with: -v
==26576== searching for pointers to 1 not-freed blocks.
==26576== checked 69,576 bytes.
==26576==
==26576== LEAK SUMMARY:
==26576==    definitely lost: 80 bytes in 1 blocks.
==26576==    possibly lost: 0 bytes in 0 blocks.
==26576==    still reachable: 0 bytes in 0 blocks.
==26576==    suppressed: 0 bytes in 0 blocks.
==26576== Rerun with --leak-check=full to see details of leaked memory.

```

Figure 5: Output of valgrind showing lost memory.

originally allocated.

Copyright 2008 by Michael J. Fischer