CS106X
Spring 99-00

Handout #16
April 14, 2000

Assignment #3: Where is pointer?

Due: at interactive grading next week

With most of the concepts you've been learning in this class, you can start using them in your programs right away, even before you completely understand them. In the course of writing the program, you encounter some bugs and as you fix them, you come to more thoroughly understand the details of the material. This "learn as you go" strategy works fine for simple C, parameters, graphics, basic arrays, and strings, but not so for pointers. The complexity and subtlety of working with memory in C does not mix well with a trial-and-error approach. You really need to grasp pointers thoroughly before attempting to write a program using them. Thus this assignment is a problem set designed to solidify your understanding of pointers and memory.

There are a half-dozen problems and I'd like you to work through all of them. However, you do not need to write up formal solutions to hand in, since our plan is to save you that effort required to neatly write up your answers that often takes more time that just solving the problems in the first place. This assignment will be graded on a strictly credit/no-credit basis and instead of handing in written solutions in class, you will bring your work from doing the problems to the interactive grading session where you and your section leader can go over it together. Your efforts will not be graded on correctness, but it will be noted if you haven't worked through the problems at all. The IG session will be a great time to clear up any lingering confusion you may have about pointers or dynamic memory and make sure you are well-prepared for putting pointers to use in the following assignments.

You are encouraged to freely collaborate with others on these exercises. The most important result is that everyone understand pointers thoroughly before we move on!

1) Pointer Pictures

For the following code segments, you are asked to draw the state of the computer's memory during code execution. Trace through, starting from **`main`** and stop at the point marked. Be sure to distinguish between stack memory and heap memory—local variables are allocated in the stack, all dynamically allocated memory should be in the heap. Label all the parts of the drawing: include the names of variables, their current values, draw pointers as arrows, indicate uninitialized/garbage values with a question mark, and indicate whenever memory is orphaned. Note that this code is purposely written to be hideous and confusing, don't you dare ever write code like this!

a)

```
main()
{
   int i, *a, *b, list[4];

   a = &i;
   for (i = 0; i < 4; i++)
      list[i] = i;
   b = a;
   a = &list[2];
   *a = *b;
   *b = a[1];

 <—  Draw the state of memory at this point
}
```

b)

```
main()
{
   int i;
   double lollipop, lifesavers[6];

   lollipop = 2;
   for (i = 0; i < 6; i++)
      lifesavers[i] = i/2;
   JollyRancher(lollipop, lifesavers, &lollipop);

  <— Draw the state of memory at this point
}


static void JollyRancher(double apple, double cherry[], double *melon)
{
   int i;

   *melon = *cherry;
   melon = GetBlock(apple*sizeof(double));
   apple++;
   melon = &cherry[3];
   for (i = 0; i < apple; i++)
      melon [i] = melon[i]/2;
}
```

c)

```
main()
{
   int alpha;
   string beta[4];
   char gamma[5];

   beta[0] = "eta";
   beta[1] = gamma;
   strcpy(gamma, beta[0]);
   alpha = -1;
   beta[2] = GetBlock(5);
   Greek(beta[2], gamma, &alpha, &beta[2]);
}

static void Greek(string delta, char *omega, int *phi, string *theta)
{
   strcpy(delta, "chi");
   theta[1] = Concat(delta, "delta");
   omega = SubString(delta, 2, 2);
   **(theta + *phi) = *omega;
   FreeBlock(*theta);
   theta = &delta;
   omega = NULL;

  <— Draw the state of memory at this point
}
```

2) Memory Mistakes Hall-of-Fame

The following fragments exhibit problems that often show up when students are learning how to work with pointers and memory in C. Your job is to look carefully at each fragment, determine the memory error and show how to correct it. These are excellent problems for improving your skills of "debugging by inspection".

a) The **`ConvertToUpperCase`** function is given a string and returns a new string which is a copy of the original string with all lowercase letters converted to uppercase. There are two mistakes lurking in here, find them and explain how you would fix the function to operate correctly.

```
static string ConvertToUpperCase(string str)
{
    string result;
    int i;

    for (i = 0; i < StringLength(str); i++)
         result[i] = toupper(str[i]);
    return result;
}
```

b) **`GetRandomArray`** is supposed to prompt the user for a count and then create an array of random numbers of size count and return the size of the array created. There are two errors here to identify and explain how you would re-write the code so that it performs correctly.

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

   count = GetRandomArray(randomNumbers);
   printf("The last number is %d\n", randomNumbers[count-1]);
}

static int GetRandomArray(int array[])
{
   int i, count;

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

c) `CensorVowels` takes an incoming string and creates a new string which is a copy of the original string with all vowels replaced by dashes. This function is written using explicit pointer manipulation in the style of the ANSI **`string.h`** interface. There is one very subtle pointer error here that you should find and explain how to fix.

```
static char *CensorVowels(char *str)
{
    char *start, *cur;

    start = GetBlock(strlen(str + 1));
    for (cur = start; *str != '\0'; str++) {
      if (IsVowel(*str))
         *cur++ = '-';
      else
         *cur++ = *str;
    }
    *cur = '\0';
    return start;
}
```

3) The stack vs. the heap

This function was intended to create an array of one-character strings, one for each letter in the alphabet. Draw the state of memory at the designated point during function execution. Is the function doing its job? Going on from there, the function frees up the alphabet strings. Is this necessary? Is this correct?

```
static void AlphabetSoup(void)
{
   int i;
   char str[2];
   string words[26];

   for (i = 0; i < 26; i++) {
      str[0] = 'A' + i;
      str[1] = '\0';
      words[i] = str;
   }
```

<— Draw the state of memory at this point

```
   for (i = 0; i < 26; i++) {
      FreeBlock(words[i]);
   }
}
```

4) Freeing Memory

Memory that is dynamically allocated (via **`malloc`**, **`GetBlock`**, or indirectly by **`strlib.h`** functions such as **`Concat`** that allocate memory for strings they return) is persistent in that it remains allocated until you explicitly free it. On one hand this is desirable—it allows you to allocate storage that lives beyond the current stack frame, but as a result, the heap can become clogged with unused variables if the memory isn't freed when it is no longer needed. Most of the time this is fine—the space wasted by not freeing things is often not that large and the problems introduced by not-quite-correct attempts to free the memory is a cure worse than the disease. However, it is instructive to think about how to correctly deallocate memory. Unfortunately keeping track of what to free can be tricky. And as important as it is to know what to free, it's also critical to know what not to free. Add code at the end of the following code fragment to free each piece of heap memory once and exactly once.

```
string where, *array;
char why[12];
int **ip;

where = "Palo Alto";
array = GetBlock(8 * sizeof(string));
strcpy(why,where);
ip = GetBlock(sizeof(int *));
*ip = GetBlock(sizeof(int));
array[0] = "Menlo Park";
array[1] = GetLine();
array[2] = where;
array[3] = why;
array[4] = array[1];
array[5] = ConvertToLowerCase(why);
```

5) Strings as pointers

a) As we discussed, the strlib functions that return strings (**`Concat`**, **`SubString`**, etc.) allocate memory for the strings they return. If the programmer isn't aware of this, it's quite easy to orphan that memory. Often we can ignore this little inefficiency, but to be correct we should free those strings when we're done with them. Consider the **`RemoveOccurrences`** function show below. Given a string, it will return a new string which consists of the original string with all occurrences of the character removed:

```
/*
 * Function: RemoveOccurrences
 * Usage: result = RemoveOccurrences(str, charToRemove);
 * ----------------------------------------------------
 * This function returns a copy of the input string with all
 * occurrences of the specified character removed. The original
 * string is not modified, a new copy is returned.
 */
static string RemoveOccurrences(string str, char ch)
{
   string result;
   int i;

   result = "";
   for (i = 0; i < StringLength(str); i++) {
      if (IthChar(str, i) != ch)
         result = Concat(result, CharToString(IthChar(str, i)));
   }
   return result;
}
```

How does this function leak memory in the heap? Rewrite this function, adding appropriate calls to **`free/FreeBlock`** so that it no longer orphans memory.

b) Write an implementation of the above **`RemoveOccurrences`** function that prototypes the argument as a pointer to a character and uses only pointer manipulation to work with the strings. Do not use array operations (e.g. subscript notation) or call any **`strlib.h`** or **`string.h`** string functions other than **`strlen`**.

```
char *RemoveOccurrences(char *str, char ch)
```

c) Which of the two above versions would you rather be in charge of maintaining for the future? Why?

6) Pointers and structures

Another tracing draw-a-memory-diagram problem like #1, this one involving structs along with pointers. By the way, this problem was taken from an old CS106X midterm exam.

```
typedef struct {
   int ID;
   int age;
} Student;

typedef struct {
   string clubName;
   Student heads[2];
   Student *members;
   Student *chairs[2];
   int numMembers;
} ClubInfo;


main()
{
   ClubInfo info;

   info.chairs[0] = GetBlock(sizeof(Student));
   info.chairs[1] = &(info.heads[1]);
   info.members = GetBlock(3 * sizeof(Student));
   info.clubName = Concat("Big", "Party");
   info.numMembers = 57;
   info.heads[0].ID = 42;
   info.heads[0].age = 21;
   UpsetClub(info, info.heads);
```

<— Draw the state of memory at this point

```
}

static void UpsetClub(ClubInfo info, Student bozos[])
{
   bozos[1].ID = 81;
   info.heads[1].age = 18;
   info.members[1].ID = 23;
   info.chairs[0] = bozos;
   info.chairs[0]->age++;
   info.members = GetBlock(sizeof(Student));
   info.numMembers++;
}
```

If I have seen farther than others,
it is because I was standing on the shoulders of giants.
— Isaac Newton

If I have not seen as far as others,
it is because giants were standing on my shoulders,
— Hal Abelson