Lab 07 - `malloc` and Friends

Out: November 5-6, 2012

1 Introduction

1.1 C and Memory

Roughly speaking, data in C can be stored in 4 different regions of memory: the **bss**[1], the **data** section, the **stack**, and the **heap**.

1.1.1 BSS/Data

The **bss** and **data** sections are used to store various pieces of static data, such as global variables, fixed string and array constants, and local variables declared with the `static` keyword. Uninitialized variables are stored in the bss and initialized to 0, while statically initialized variables are stored in the data section. The size of these regions is determined at static link time and cannot be changed while the program is running.

1.1.2 Stack

The **stack** is the region of program memory used to store data local to a function, including non-static local variables, arrays, and program arguments. Each time a function is called, the program allocates space on the stack known as a *stack frame* to store these variables. When the function returns, the stack frame is popped off the stack, freeing the space for future use.

1.1.3 Heap

The **heap**[2] is a region of memory that can be used to store large blocks of data. The program can request memory in this region using a special set of functions, described below, through a process known as *dynamic allocation*. Blocks allocated on the heap must be explicitly freed when they are no longer needed.

In this lab, you will use the heap and dynamic memory allocation to implement a hash set.

1.2 Dynamic Memory Allocation in C

Up to this point in the course, your C programs have only used local and global variables (including arrays and structures) to store program information. Non-static local variables are stored within a given function's stack frame (or, in some cases, in registers). This is nice because the space used

[1] The term "bss", an acronym for "Block Started by Symbol", originates from a pseudo-operation in UA-SAP, an assembler written in the 1950s for the IBM 704.

[2] This is *not* the same as the min-heap data structure used in a priority queue. You should think of the heap as an unordered pool of memory, in contrast to the ordered stack. The origin of this usage of "heap" is unknown.

to store the variables is automatically created when the function is called, and discarded when the function returns. However, stack variables have several limitations:

- Memory used for stack variables is only valid over the lifetime of a function call. This means that pointers to local variables or arrays should not be returned from a function, because the variable referred to will be popped off the stack when the function returns, rendering the pointer invalid.
- The stack has a fixed maximum size, so storing too much data on the stack can cause a *stack overflow*. This can corrupt other parts of the program's memory or produce a segmentation fault. (The heap also has a maximum size, but it is much larger.)
- Prior to the C99 standard, the size of a local variable was required to be known at compile time so the compiler could know how much space to allocate on the stack. In particular, this meant that arrays had to have a fixed size. If you did not know in advance how much memory your program would require, this could present an obstacle. With C99, array size may be determined at runtime, but arrays on the stack still cannot be resized after they are created.

The solution to the above problems lies in the `malloc()`, `calloc()`, `realloc()`, and `free()` functions, defined in `<stdlib.h>`. This group of functions provides a way in which arbitrary-size blocks of memory may be requested by the program at runtime. Dynamically allocated memory is stored in the heap, not the stack, so it will persist until it is explicitly deallocated.

1.2.1 Requesting a New Block of Memory

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
```

The `malloc()` and `calloc()` functions are used to request new blocks of memory on the heap. `malloc()` allocates a contiguous block of memory whose length in bytes is specified by the `size` parameter, and returns a pointer to the start of the block. `calloc()` takes two arguments, `nmemb` and `size`, and allocates space for an array of `nmemb` elements, each of which has size `size`. The function fills the requested memory with 0s before returning it. This is sometimes convenient, but is also slower. Both functions will return a null pointer on failure.

Often, `malloc()` and `calloc()` are used in conjunction with the `sizeof` operator, which yields the number of bytes required for a given data type. For instance, to allocate an array of `n` integers on the heap, you could write:

```
int *array = (int *) malloc(n * sizeof(int));
```

or:

```
int *array = (int *) calloc(n, sizeof(int));
```

Note that in either case, the returned pointer is of type `void *`, so it has to be cast to the desired type.

1.2.2 Resizing an Existing Block

```
void *realloc(void *ptr, size_t size);
```

To change the size of an existing block of memory, you use the `realloc()` function. `realloc()` resizes the given block of memory to the specified size and returns a pointer to the start of the resized block (which may or may not be the same as the old pointer). The contents of the memory block are preserved up to the minimum of the old and new sizes. If the new size is larger, the value of the newly allocated portion is indeterminate. If memory allocation fails, a null pointer is returned and the block specified by `ptr` is unaffected.

1.2.3 Deallocating a Block

```
void free(void *ptr);
```

The `free()` function will deallocate the entire block beginning at `ptr`, returning it to the heap for further memory allocation. If the given pointer is null, no action will be performed. The behavior of `free()` is undefined if the pointer does not point to the start of a dynamically-allocated block that has not yet been freed. Note that you do not have to specify the size of the block to be freed, as dynamically allocated blocks have metadata that includes this information.

1.2.4 Example

The following example shows the combined use of `malloc()`, `realloc()`, and `free()`:

```
void do_something(int n) {
     //Allocate space for an array of n integers
     int *array = (int *) malloc(n * sizeof(int));

     //...

     //Double the size of the array
     array = (int *) realloc(array, 2 * n * sizeof(int));

     //...

     //Free the array
     free(array);
}
```

1.2.5 Things to Avoid

Dynamic memory management gives you great power. These functions give you the ability to allocate memory in a function that can be used far beyond the lifetime of the function call. They enable you to create large data structures that change size as necessary according to runtime

conditions. But with great power comes great responsibility. Dynamic memory allocation requires careful discipline to avoid bugs and other problems.

1. **Memory Leaks**

   A *memory leak* is an issue that occurs when all pointers to a block of dynamically-allocated memory are lost before the block of memory is freed. Memory allocated on the heap persists beyond function calls and the computer has no way of knowing when a block of memory is not needed any more. In Java, this is handled by garbage collection[3], but there are no such niceties in C. The programmer must explicitly release a block of memory back into the heap using the `free()` command when it is done being used.

   The following code will cause a memory leak:

```
/* Accept an array of dynamically-allocated strings,
 * and replace the string at a given index with a new one
 */
void replace_string(char **array, char *str, int index) {
    array[index] = str;
    //The pointer to the old value of array[index] is lost!
}
```

   Memory leaks can also occur when a data structure or array is freed without first freeing memory pointed to within the structure:

```
struct foo {
    int x;
    char *str;
}

void leakage() {
    struct foo *bar = malloc(sizeof(struct foo));

    bar->x = 5;
    bar->str = (char *) malloc(2 * sizeof(char));
    strncpy(bar->str, "!", 2);

    //...

    free(bar);  //Careful!  bar is freed but bar->str is not
}
```

   Memory leaks may be difficult to detect because they do not directly affect the operation of the program. However, over time the leaked memory will build up. For long-running programs, this can eventually cause the program to run out of memory and fail. In general, each call to `malloc()` or `calloc()` should have a corresponding call to `free()` later on.

[3] A block of allocated memory with no pointers pointing to it cannot be accessed by the program, and is thus called *garbage*. Garbage collectors keep track of allocated memory and periodically clean up garbage during the execution of the program. This process is covered in Chapter 9 of the textbook.

2. **Misuse of `free()`**

   Passing `free()` an invalid pointer, such as a pointer to an already-freed block of memory or a location other than the start of a dynamically allocated block, will cause unpredictable behavior. If you're lucky, you will immediately recieve an error and the program will terminate. In the worst case, you may experience data corruption, segmentation faulting, or other problems. The following example shows incorrect use of `free()`:

   ```
   void abuse_of_freedom() {
       int x = 3;
       free(&x);  //Incorrect!  x is stored on the stack, not the heap
   }
   ```

3. **Dangling Pointers**

   A third type of mistake is attempting to read from or write to a block of memory that has already been freed. This can cause errors similar to those caused by returning a pointer to a local variable. The following is an example of this type of error in action:

   ```
   int dangling_pointage(int n, int x) {
       int *arr = (int *) malloc(a * sizeof(int));

       // ...

       free(arr);

       return arr[0];
       //There is no guarantee on what the value of arr[0] will be
   }
   ```

   The above code will compile successfully, but may cause unpredictable behavior at runtime. To avoid such errors, you should keep track of all pointers to a given dynamically-allocated block of memory and make sure they are never used after the block has been freed. If there is only one pointer to a given block, setting the pointer to `NULL` directly after calling `free()` can prevent accidental pointer misuse.

2 Assignment

2.1 Specification

In this lab, your task is to use dynamic memory allocation to implement a basic hashset in C. The hashset will be represented by an array of unsorted linked lists, called "buckets". Each element is associated with a fixed "hash code"; this hash code is divided by the array size and the remainder is used as an index in the bucket array for that element. To complete this lab, you must fill in the implementations of the following functions:

- `HashSet *hashset_new()`
- `void hashset_delete()`

- `int hashset_add()`
- `int hashset_remove()`

When created, the hashset is given four function pointers that will used be for various aspects of the hashset:

- A `hashcode()` function to generate the hash code for each element,
- An `equals()` function to test two elements for equality,
- A `copy()` function to create a copy of an element for the hashset to store, and
- A `delete()` function to delete the hashset's private copy of an element.

When adding an element to the array, the program should first check each element in the corresponding bucket to make sure the element is not already present. If it is not, a new list node should be created for the element and added to the bucket. If the ratio of elements to buckets exceeds a certain fixed *load factor*, a new bucket array should be created with twice the size of the old array. Each element should then be transferred from the new array to the old array.

Removal works much like addition: the program should check the appropriate bucket to see if a matching element exists. If an element is found, it should be removed from its bucket and deleted. Note that the bucket array size is never decreased on removal, it only increases.

2.2 Handout

Your handout should contain the following files:

- ***hashset.h*:** A header file containing declarations of functions providing set functionality
- ***hashset.c*:** The C file where you will be implementing the set functions
- ***string_fns.h*, *string_fns.c*:** Files supporting various string operations
- ***tester.c*:** A C file providing a testing interface for the lab
- ***addme*, *addme2*:** Test data to use with the test interface
- ***Makefile*:** A makefile for the lab
- ***lab07.pdf*:** This handout

You will only need to modify *hashset.c* to complete the lab.

2.3 Compiling and Running

We have provided a makefile to allow you to easily compile this lab. To compile your code and the test code, run:

```
make
```

This will create an executable called *tester* that uses the hashset to store a collection of strings. The testing code generates a hashcode for a given string by using the first four bytes of the string as the bytes of a little-endian 32-bit integer. This would not be a good hashcode in "real life", but it is fine for testing your code.

The test program accepts the following commands:

- `a <str>`: Add `<str>` to the set
- `c <str>`: Test if `<str>` is in the set
- `r <str>`: Remove `<str>` from the set
- `x`: Remove all elements from the set
- `s`: Print the set size
- `p`: Print a representation of the set
- `fa <file>`: Add each whitespace-separated character string in file `<file>` to the hashset. For convenience, you are provided with test files *addme* and *addme2* which contain 12 and 46 elements respectively.
- `fr <file>`: Remove each whitespace-separated character string in file `<file>` from the hashset.
- `help`: Print help information
- `quit`: Delete the set and exit the program

To clear all compilation products, run:

```
make clean
```

2.4 Testing

To check for memory leaks and other types of memory errors, you can use the Valgrind utility. Valgrind reports issues such as accessing memory that has been freed or allocating memory without later freeing it. To run your program within Valgrind, enter the following at the command line:

```
valgrind --leak-check=full your_executable
```

where `your_executable` is the file path of the executable to run (for example, `./tester`).

3 Getting Checked Off

Once you've completed the lab and your solution contains no memory leaks, call a TA over and have them inspect your work. If you do not complete the lab during your lab session, you can get credit for it by completing it on your own time and bringing it to TA hours before next week's lab. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.