APS105: Dynamic Memory Allocation

Daniel Zingaro

March 29, 2010

1 Why Dynamic Memory Allocation?

Up to this point in the course, C has automatically managed our programs' memory for us. For example:

- (1) When a function is called, memory for the function's local variables and parameters is allocated. When the function returns, that memory is automatically released. We can call a function as many times as we like and not worry about how that memory gets acquired and freed.
- (2) When we declare an array with a specific number of elements, C automatically allocates a chunk of memory in which to store these elements. We are assured that when we start writing into the array's elements, we will be writing into valid memory that has been properly allocated to our program. This is also true for structures: C handles the details of allocating memory for our structure variables.

Both of these facts greatly simplify writing programs. It would be a chore to have to manually allocate space for each variable when we call a new function, or to have to explicitly state how much memory is required for an array of a certain size. However, both of these C features also restrict what we can do with the memory our program uses. For example:

- (Drawback of (1)) What if we want to allocate some memory in a function, but we do not want to lose that memory when the function terminates? That is, we want to be able to put some value into a local variable, and make that variable available long after the function returns. For example, what if we wanted to write a function that put a random phrase into a newly allocated string, and returned a pointer to that string? We cannot let the standard behavior of local variables (losing their memory when the function terminates) take place; we must take greater control of memory management.
- (Drawback of (2)) What if we do not know how big to make an array? Arrays have the property that once we choose their size, we're stuck with that choice until the function terminates. We might assume, for example, that 50 elements will be enough for our purposes. But then, the user might type in way more stuff than we expect, so the array will not be able to hold it all. If we let C manage the array's memory, we must stop accepting data, since we have nowhere to put it (we cannot extend the array). But if we take control of memory management, we can change the size of the array as required. We can increase its size to accommodate more data, and even take away array elements that are no longer being used.

2 Malloc

To take control of memory management, there are two important functions.

- `malloc`: we tell this function how many bytes of memory we require, and it finds a chunk of memory that can satisfy this request. It then returns a pointer to the beginning of this chunk of memory. We can use this chunk of memory for whatever we

like. It is guaranteed to be contiguous, so we can use it to store an array, a structure, a string, a single variable, and so on.

- `free`: releases the memory we obtained using `malloc`. We use `free` when we're finished using a piece of memory.

Let's begin by studying the following program, which does not use `malloc` at all.

```
#include <stdio.h>

int main(void) {
  int howMany;
  printf ("How many grades? ");
  scanf ("%d", &howMany);
  int grades[howMany];
  printf ("Enter all grades.\n");
  for (int i = 0; i < howMany; i++)
    scanf ("%d", &grades[i]);
  printf ("\n***You entered***\n");
  for (int i = 0; i < howMany; i++)
    printf ("%d\n", grades[i]);
  return 0;
}
```

This program begins by asking the user how many marks they wish to enter. It then declares an array to hold the specified number of marks, and then asks for and stores each grade in the array.

We could use `malloc` to accomplish the very same thing, as shown in the next program.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int howMany;
  printf ("How many grades? ");
  scanf ("%d", &howMany);
  int *grades;
   grades = malloc (howMany * sizeof(int));
  printf ("Enter all grades.\n");
  for (int i = 0; i < howMany; i++)
    scanf ("%d", &grades[i]);
  printf ("\n***You entered***\n");
  for (int i = 0; i < howMany; i++)
    printf ("%d\n", grades[i]);
  return 0;
}
```

- We have added the `stdlib.h` include file at the top of the program. This is required if we want to call `malloc`. (This is similar to the `stdio.h` requirement for when we want to use `printf`.)
- We have replaced the array with a pointer `grades`. We then give `grades` the return value from `malloc`, which is a pointer to a new block of memory large enough to store the array's contents. We want to be able to store `howMany` integers, and the size of each

of these integers is `sizeof (int)`. Therefore, we multiply these two values in order to find the total number of bytes required.

What we are doing here is simulating exactly what C would have done for an array. We have asked for the appropriate number of bytes, so that we can use that block of memory just like an array.

Today, we typically do not use `malloc` to allocate memory for an array (as we did here). However, in the early days of C, there was a restriction that goes like this: "the number of elements of an array must be determined by a constant value". What that meant was that we had to decide on an array's size before we compiled the program. We could choose a value like 5, or 10, or 20, but no matter what, the constant value itself had to be inside the `.c` file and could not be the value of a variable at runtime. This was quite problematic. If we wrote a program to manage the marks of students, we would have to guess before compiling the program how many students we would be able to handle. 30? Well, what if someone wanted to use the program for a huge class. 1000? Now we will waste tons of memory for small classes! The resolution, of course, was to use `malloc`, allocating memory once we have determined exactly how much to allocate.

But, as said, today we don't have to worry about doing that, so our `malloc` example above, in the context of storing grades, is more educational than practical.

3 Free

Once we use `malloc`, C no longer manages that piece of memory for us. Specifically, we are required to free (release, give up) the memory once we are done with it. To do so, we use the `free` function. We can add a call of `free` to our prior example, as follows.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int howMany;
  printf ("How many grades? ");
  scanf ("%d", &howMany);
  int *grades;
   grades = malloc (howMany * sizeof(int));
  printf ("Enter all grades.\n");
  for (int i = 0; i < howMany; i++)
    scanf ("%d", &grades[i]);
  printf ("\n***You entered***\n");
  for (int i = 0; i < howMany; i++)
    printf ("%d\n", grades[i]);
  free (grades);
  return 0;
}
```

The truth is that memory that our program uses is automatically freed when the program terminates, so in a practical sense this call of `free` does nothing. But it is very good style to always `free` everything that was allocated with `malloc`.

4 Example of malloc and free

Here is a real example of a function that uses `malloc`, and a `main` function that uses `free` to release memory when it is no longer required. The function `getJoke` uses a **local**

variable array to store the beginning of one of two bad jokes. We want to return this joke from the function, so that the caller can print it. Unfortunately, `choice` is a local variable, whose memory is destroyed when the function returns. If we were to say `return choice;`, our caller would get a pointer to the first character of the joke. But, the characters of the joke are going to be long gone, because the function has already terminated.

Instead, what we do is allocate our own memory using `malloc`. We know that **this** memory will not go away when the function finishes. When we `return newString`, we're giving our caller a pointer to the first character of the joke, and our caller can safely use this to read the entire string starting from that first character.

```
#include <time.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

char *getJoke () {
  char choice[100];
  int i = rand () % 2; /*Two jokes*/
  if (i == 0)
    strcpy (choice, "Why did the chicken...");
  else if (i == 1)
    strcpy (choice, "What did one wall...");
  /*Our joke is now in choice. But choice is a local variable!
  It is destroyed when the function returns.
  To get permanent memory, we use malloc.*/
  char *newString = malloc (strlen (choice) + 1);
  strcpy (newString, choice);
  return newString;
}

int main (void) {
  srand (time(0));
  char *j;
  j = getJoke ();
  printf ("%s\n", j);
  /*Done with the first joke...*/
  free (j);
  j = getJoke (); /*Get another joke*/
  printf ("%s\n", j);
  free (j);
  return 0;
}
```

Two points about the `malloc` call:

- We ask for one more byte than `strlen (choice)`; this extra byte is going to be used to store the `'\0'` character that gets copied by `strcpy`.
- There's no `sizeof` expression in this `malloc` call. The reason is that we're assuming characters to be one byte, so `sizeof (char)` will be 1.

I'd like you to think back to lab 4, where you wrote a `subtract` function that placed characters in the `result` string. We assumed that `result` had enough space in it for the

characters; it was the caller's responsibility to declare an array that could be passed for the `result` parameter. You now know that we could have used `malloc` as an alternative. Rather than having `result` as a parameter and having the caller of the function declare an array with enough space, we could have returned a pointer to the result, allocating enough memory inside the function using `malloc`.

5 Declaring and Allocating

Let's say we wanted to use `malloc` to allocate enough memory to store one integer. We could do it like this:

```
int *p; /*Pointer to int (pointing to random memory)*/
p = malloc (sizeof (int));
*p = 42; /*Store 42 in the memory we got from malloc*/
```

As the code shows, we can store an integer in the memory given to us by `malloc` using `*`.

We can also write the code more compactly, taking advantage of the fact that we can initialize a variable's value at the same time we declare that variable:

```
int *p = malloc (sizeof (int));
*p = 42; /*Store 42 in the memory we got from malloc*/
```

Finally, compare that code to this code:

```
int q;
int *p = &q;
*p = 42; /*Store 42 in the memory referenced by q*/
```

Here, we are not using `malloc`. Instead, we are letting C manage the memory for one integer, making it available through the variable `q`, and making `p` point to the same memory address as `q`.

In both cases — using `malloc` and using `&` — we are assigning a memory address to a pointer variable.

6 Mallocing Arrays

We know how to ask `malloc` for memory for one integer, and we know how to store an integer in the memory `malloc` gives us:

```
int *p;
p = malloc (sizeof (int));
*p = 42;
printf ("We stored %d.\n", *p);
```

What if we want room to store multiple integers, like 5? We could do that like this:

```
int *q;
q = malloc (5 * sizeof (int));
```

To access the memory for these five integers, we can treat `q` like an array!

```
*q = 2; /*2 is in the first array element*/
*(q + 1) = 48; /*48 in the second element*/
q[2] = 100; /*100 in the third*/
```