

COP 3223: C Programming Spring 2009

Dynamic Storage Structures In C – Part 1

Instructor : Dr. Mark Llewellyn
 markl@cs.ucf.edu
 HEC 236, 407-823-2790
<http://www.cs.ucf.edu/courses/cop3223/spr2009/section1>

School of Electrical Engineering and Computer Science
University of Central Florida



Dynamic Storage Structures In C

- The data structures that we have examined so far in this course, such as arrays, have all had their memory statically allocated. In other words, the memory for the variable was reserved when the program was loaded into memory and once loaded the memory the allocation could not change, i.e., it was static. Thus, if we declared the array to contain 10 integer elements, it was not possible once the program was compiled and loaded into memory to change the number of elements to 20.
- Statically allocated data structures place a fairly severe handicap on the program developer by forcing them to choose the largest possible size that might every be needed for their data structures, even if the majority of the time the program execution does not actually require that much space for the data structure.
- The solution to this problem is to dynamically allocate memory to data structures while the program is in execution.



Dynamic Storage Structures In C

- Dynamic memory allocation allows the size of a data structure to expand and contract as needed during the execution of a program as well as across multiple executions of the same program. Thus, one execution of the program might require 50 elements in a structure while another execution of the same program might require 5000 elements.
- As an example, consider the simple database program from the previous set of notes (Structures In C – Part 3). The array size was set to hold a maximum of 50 students, if we had 51 students or 50,000 students we would need to modify the program, recompile it, and execute the new version to accommodate the larger number of students. If we modified it to hold up to 50, 000 students but only loaded 10 student's information, the remaining 49, 990 locations would be allocated to our program but unused – a terrible waste of memory.



Dynamic Storage Structures In C

- Although it is possible for dynamic memory to be used for any type of data, it is most commonly used for strings, arrays, and structures.
- One of the most useful techniques with dynamic memory allocation is with dynamic allocated structures in which each structure variable contains a pointer to another structure of the same type, allowing them to be linked together to form data structures known as lists, trees, and many other types of data structures.
- Such structures are often called **self-referential structures**, because they contain a member which is a pointer to a structure of the same type. We'll see this shortly.



Dynamic Storage Allocation Functions

- C provides three different mechanisms for allocating memory dynamically. This is done with one of three memory allocation functions which are all declared in the `<stdlib.h>` header file. These functions are:
 - malloc** – allocates a block of memory but does not initialize it.
 - calloc** – allocates a block of memory and clears it.
 - realloc** – resizes a previously allocated block of memory.
- Of the three functions, `malloc` is the most commonly used. It is more efficient than `calloc`, since it does not clear the memory block that it allocates.



Dynamic Storage Allocation Functions

The function prototype for `malloc` is:

```
void *malloc (size_t size);
```

`malloc` allocates a block of `size` bytes and returns a pointer to the first byte in the block. The parameter `size` has type `size_t` which is an unsigned integer type defined in the C library. Unless you are allocating a very large block of memory, you can think of `size` as just an `int`. `malloc` returns a null pointer if a block of `size` is not available.



Dynamic Storage Allocation Functions

The function prototype for `calloc` is:

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

`calloc` is a better solution when allocating memory for an array. `calloc` allocates space for an array with `nmemb` elements, each of which is `size` bytes long; it returns a null pointer if the requested space is not available. After allocating the memory, `calloc` initializes it by setting every bit in the allocation space to 0.



Dynamic Storage Allocation Functions

The function prototype for `realloc` is:

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

Once memory for an array has been allocated, you might find later that it is either too large or too small for current requirements. The `realloc` function can resize the array to better suit current needs. When `realloc` is invoked, `ptr` must point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`. The `size` parameter represents the new size of the block, which can be smaller or larger than the original size.

Although `realloc` does not require that `ptr` point to memory that is used as an array, it typically does.



Dynamic Storage Allocation Functions

- When any of the three memory allocation functions is invoked to request a block of memory, the function has no idea what type of data the programmer is planning to store in the block, so it cannot return a pointer to an ordinary type such as an `int` or `char`. Instead, the function returns a value of type `void *`.
- A `void *` value is a generic pointer – in other words, its just a memory address with no implied storage type associated with the address. It really is just and address in the memory somewhere.



Null Pointers

- When any of the memory allocation functions are invoked, there is always a possibility that it will not be able to locate a block of memory large enough to satisfy the request.
- If this happens, the function will return a **null pointer**. A null pointer is a pointer that points to nothing. It is a special value that is used to indicate that the pointer references no valid address in the memory. The null value can be used to distinguish such a pointer from all other valid pointers.
- After the function returns its pointer value, you must always check to determine if a null pointer has been returned.



Null Pointers

- Null pointers are represented in C by a macro named NULL (i.e., a C defined constant).
- This makes it easy to test for null pointer values, as shown below:

```
//allocate a 10000 byte block of memory
ptr = malloc(10000); //ptr points to the first byte
                        //in the allocated block
if (ptr == NULL) {
    //allocation failed - take appropriate action!
}
else //continue execution with allocated block
```



Null Pointers

- Similar to what we usually do with file pointers, the memory allocation function call and the test for the return of a null pointer are commonly combined into a single statement as shown below:

```
//allocate a 10000 byte block of memory
if ((ptr = malloc(10000)) == NULL) {
    //allocation failed - take appropriate action!
}
else //continue execution with allocated block
```



Dynamically Allocated Strings

- Dynamic storage allocation is often used when working with strings. Strings are stored as arrays of characters and it can be hard to anticipate the length of the array prior to execution.
- By allocating the string dynamically, the decision about the length of the array can be postponed until the program is in execution.
- Using `malloc` to allocate memory for a string is easy because C guarantees that a `char` value requires exactly one byte of storage.



Dynamically Allocated Strings

- Thus to allocate space for a string of n characters, you would write:

```
char *ptr;  
ptr = malloc(n+1);
```

- The argument to `malloc` is $n+1$ to allow for the `'\n'` character.
- The generic pointer that `malloc` returns will be converted to `char *` when the assignment is performed; no cast is required. (In general, you can assign a `void *` value to a variable of any pointer type and vice versa.)
- However, if you prefer to cast for clarity, it is legal and would look like:

```
ptr = (char *) malloc(n+1);
```



Dynamically Allocated Strings

- The example on the next page illustrates a simple application of `malloc`.
- In this program, two different calls to `malloc` create arrays of 29 characters.
- The first pointer returned is used to hold a string of only 4 characters in length. Notice that `puts` properly prints the string since the fifth element in the string will be the `'\n'`. However, the loop printing the string prints all the uninitialized locations in the array as well (since I didn't stop or look for the `'\n'` character).
- The second pointer returned is used to hold a string of 29 characters, so in the end there will be no uninitialized locations in the array, so both `puts` and the loop will properly print the string.



```
5 #include <string.h>
6 #include <stdio.h>
7 #define N 29
8 int main()
9 {
10     char *ptr1, *ptr2; //two pointers to dynamically allocated strings
11     int i; //loop control variable
12
13     if ((ptr1 = malloc(N)) == NULL) {
14         printf("Sorry, couldn't allocated memory for ptr1\n");
15     } //end if stmt
16     else {
17         strcpy(ptr1, "Mark");      puts(ptr1);
18         for (i = 0; i < N; i++) { //will print uninitialized locations in string
19             printf("%c", *ptr1);
20             ptr1++;
21         } //end for stmt
22     } //end else stmt
23     printf("\n\n");
24     if ((ptr2 = malloc(N)) == NULL) {
25         printf("Sorry, couldn't allocated memory for ptr1\n");
26     } //end if stmt
27     else {
28         strcpy(ptr2, "This string has 29 characters");      puts(ptr2);
29         for (i = 0; i < N; i++) { //no uninitialized locations in string
30             printf("%c", *ptr2);
31             ptr2++;
32         } //end for stmt
33     } //end else stmt
34     printf("\n\n");
```



The screenshot shows a Windows command prompt window with the following text and annotations:

- Output from puts**: Points to the first line of output, "Mark".
- Output from loop – note uninitialized positions in string**: Points to the second line of output, "Mark ▶| erver 5.0\bin;C:\Prog".
- Output from puts**: Points to the third line of output, "This string has 29 characters".
- Output from loop – note there are no uninitialized positions in string the entire allocation was used.**: Points to the fourth line of output, "This string has 29 characters".
- Press any key to continue . . .**: The fifth line of output.



Dynamically Allocated Strings

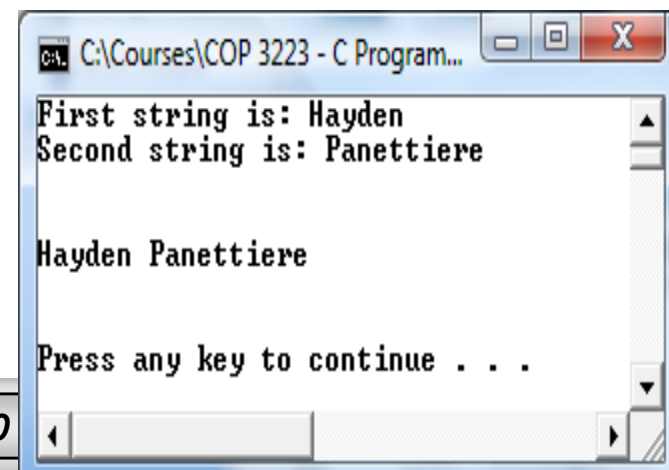
- Recall when we were looking at strings in C that the function `strcat`, which concatenated two strings together was destructive in the sense that the second string was added to the end of the first string and thus the original first string was lost after the call.
- Suppose, we wanted to write our own string concatenation function that was not destructive like the built-in function but maintained the original strings and created a new third string to hold the concatenation of the other two string.
- Our function will need to determine the lengths of the two strings to be concatenated and then call `malloc` to allocate just the right amount of space for the concatenated string.
- The program on the next page implements this function.



```
1 //Dynamic Structures In C - Part 1 - malloc example 2 - string concatenation
2 //April 21, 2009      Written by: Mark Llewellyn
3
4 #include <stdlib.h>
5 #include <string.h>
6 #include <stdio.h>
7 #define MAX 25 //can handle up to 24 character strings
8
9 char *concatenate(const char *stringPtr1, const char *stringPtr2)
10 {
11     char *resultStrPtr; //ptr to the concatenated string - to be returned
12
13     if ((resultStrPtr = malloc(strlen(stringPtr1) + strlen(stringPtr2) + 1)) == NULL) {
14         printf("Sorry, couldn't allocate that much space for the new string\n");
15     } //end if stmt
16     else {
17         strcpy(resultStrPtr, stringPtr1); //copy string 1 into result string
18         strcat(resultStrPtr, stringPtr2); //stick string 2 onto end of result string
19         return resultStrPtr;
20     } //end else stmt
21 } //end concatenate function
22
```



```
22
23 int main()
24 {
25     char *stringPtr1, *stringPtr2; //two dynamically allocated strings
26     char *concatenatedStringPtr; //dynamically allocated result string
27
28     if ((stringPtr1 = malloc(7)) == NULL) {
29         printf("Sorry, couldn't allocate space for string.\n");
30     } //end if stmt
31     else {
32         stringPtr1 = "Hayden ";
33         printf("First string is: %s\n", stringPtr1);
34         if ((stringPtr2 = malloc(12)) == NULL) {
35             printf("Sorry, couldn't allocate space for string.\n");
36         } //end if stmt
37         else {
38             stringPtr2 = "Panettiere";
39             printf("Second string is: %s\n\n\n", stringPtr2);
40             concatenatedStringPtr = concatenate(stringPtr1, stringPtr2);
41             puts(concatenatedStringPtr);
42         } //end else stmt
43     } //end else stmt
44
45     printf("\n\n");
46     system("PAUSE");
47     return 0;
48 } //end main function
49
```



```
C:\Courses\COP 3223 - C Program...
First string is: Hayden
Second string is: Panettiere

Hayden Panettiere

Press any key to continue . . .
```

Dynamically Allocated Arrays

- Dynamically allocated arrays have the same basic advantages as dynamically allocated strings (fairly obvious since strings are stored as arrays).
- The close relationship between pointers and arrays makes a dynamically allocated array just as easy to use as a statically allocated arrays (if you know how to use pointers that is).
- Although `malloc` can be used to allocate dynamic arrays, it is often `calloc` that is used for this purpose, but will start with an example that uses `malloc` just to show how its done.
- Unlike, with arrays of characters (strings) where the size of every element in the array is 1 byte, ordinary arrays contain elements of various sizes depending on the type of elements in the array. As a result the `sizeof` operator is frequently used.



Dynamically Allocated Arrays

- The `sizeof` operator is used to determine the amount space that is required for each element in the dynamically allocated array.
- Suppose that we need to declare an array of `n` integer values, where `n` is read in from the keyboard when the program begins execution. We'd need to do something like this:

```
int *ptr; //a pointer to an integer
. . .
scanf("%d", &n); //read in value of n
//allocate n location each the size of 1 int
ptr = malloc(n * sizeof(int));
```



```
3 |
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     int *ptr; //pointer to an integer - used to point to dynamic array
10    int n; //number of integers in the array - user input from keyboard
11    int i; //loop control variable
12
13    printf("How many integer values do you want to store? \n");
14    scanf("%d", &n); //read in n
15    if ((ptr = malloc(n * sizeof(int)) ) == NULL ) {
16        printf("Sorry, couldn't allocate that much space.\n");
17    } //end if stmt
18    else {
19        for (i = 0; i < n; i++){
20            printf("Enter number %d: ", i);
21            scanf("%d", &ptr[i]); //notice pointer being used as a normal array name
22        } //end for stmt
23        printf("\n\nThe array contains:\n");
24        for (i = 0; i < n; i++){
25            printf("ptr[%d] = %d\n", i, ptr[i]);
26        } //end for stmt
27
28        printf("\n\n");
29        system("PAUSE");
30        return 0;
31    } //end else stmt
32 } //end main function
```



```
C:\Courses\COP 3223 - C Programming\Spring 200...
How many integer values do you want to store?
4
Enter number 0: 3
Enter number 1: 5
Enter number 2: 7
Enter number 3: 9

The array contains:
ptr[0] = 3
ptr[1] = 5
ptr[2] = 7
ptr[3] = 9

Press any key to continue . . . _
```

Two different runs of the program on page 23

```
C:\Courses\COP 3223 - C Programming\Spring 2009\...
How many integer values do you want to store?
8
Enter number 0: 2
Enter number 1: 4
Enter number 2: 6
Enter number 3: 8
Enter number 4: 10
Enter number 5: 12
Enter number 6: 14
Enter number 7: 16

The array contains:
ptr[0] = 2
ptr[1] = 4
ptr[2] = 6
ptr[3] = 8
ptr[4] = 10
ptr[5] = 12
ptr[6] = 14
ptr[7] = 16

Press any key to continue . . .
```



Practice Problems

1. Write your own version of the `strcpy` function similar to the way we wrote the concatenation function. Make your function non-destructive to both input parameters.

