Chapter 12: Dynamic Allocation

WHAT IS DYNAMIC ALLOCATION ?

Dynamic allocation is very intimidating to a person the first time he comes across it, but that need not be. Simply relax and read this chapter carefully and you will have a good grounding in a very valuable programming resource. All of the variables in every program up to this point have been static variables as far as we are concerned. (Actually, some of them have been automatic and were dynamically allocated for you by the system, but it was transparent to you.) In this chapter, we will study some dynamically allocated variables. They are variables that do not exist when the program is loaded, but are created dynamically as they are needed while the program is running. It is possible, using these techniques, to create as many variables as needed, use them, and deallocate their memory space for reuse by other variables. As usual, the best teacher is an example, so examine the program named dynlist.c.

We begin by defining a named structure, **animal**, with a few fields pertaining to dogs. We do not define any variables of this type, only three pointers. If you search through the remainder of the program, you will find no variables defined, so we have nothing to store data in. All we have to work with are three pointers, each of which point to the defined structure. In order to do anything, we need some variables, so we will create some dynamically.

DYNAMIC VARIABLE CREATION

The statement in line 14, which assigns something to the pointer **pet1** will create a dynamic structure containing three variables. The heart of the statement is the **malloc()** function buried in the middle of the statement. This is a memory allocate function that needs the other things to completely define it. The **malloc()** function, by default, will allocate a piece of memory on a heap that is "n" characters in length and will be of type character. The "n" must be specified as the only argument to the function. We will discuss "n" shortly, but first we need to define a heap.

WHAT IS A HEAP ?

Every compiler has a set of limitations on it as to how big the executable file can be, how many variables can be used, how long the source file can be, etc. One limitation placed on users by most early MS-DOS C compilers is a limit of 64K for the executable code if you happen to be in the small memory model. This is because the earliest IBM-PC used a microprocessor with a 64K segment size, and it required special calls to use data outside of a single segment. In order to keep the program small and efficient, these calls are not used in some MS-DOS implementations of C, and the memory space is limited but still adequate for most programs. Note that most later PC's use 32 bit addressing which permits much larger programs and 32 bit compilers alleviate the need for the various memory models.

A heap is an area outside of this 64K boundary which can be accessed by the program to store data and variables. The data and variables are put on the heap by the system as calls to **malloc()** are made. The system keeps track of where the data is stored. Data and variables can be deallocated as desired, leading to holes in the heap. The system knows where the holes are and will use them for additional data storage as more **malloc()** calls are made. The structure of the heap is therefore a very dynamic entity, changing constantly.

BACK TO THE MALLOC FUNCTION

Hopefully the above description of the heap and the overall plan for dynamic allocation helped you to understand what we are doing with the **malloc()** function. It simply asks the system for a block of memory of the size specified, and returns the block with the pointer pointing to the first element of the block. The only argument in the parentheses is the size of the block desired and in our present case, we desire a block that will hold one of the structures we defined at the beginning of the program. The **sizeof** operator is new, new to us at least. It returns the size in bytes of the argument within its parentheses. It therefore, returns the size of the structure named **animal**, in bytes, and that number is sent to the system with the **malloc()** call. At the completion of that call, we have a block on the heap allocated to us, with the pointer named **pet1** pointing to the block of data.

WHAT IS A CAST ?

We still have a funny looking construct at the beginning of the **malloc()** function call, which is called a cast. The **malloc()** function returns a block with the pointer pointing to it being a pointer to type **void** by default. You really cannot use a pointer to **void**, so it must be changed to some other type. You can define the pointer type with the construct given on the example line. In this case we want the pointer to point to a structure of type **animal**, so we tell the compiler with this strange looking construct. Even if you omit the cast, most compilers will return a pointer correctly, give you a warning, and go on to produce a working program. It is better programming practice to provide the compiler with the cast to prevent getting the warning message.

The data space of the computer is depicted graphically by figure 12-1 following execution of line 14. The graphical notation defines the pointer as pointing to the structure. As far as the program is concerned, the pointer is actually pointing to all three members taken as a group rather than to the first element.

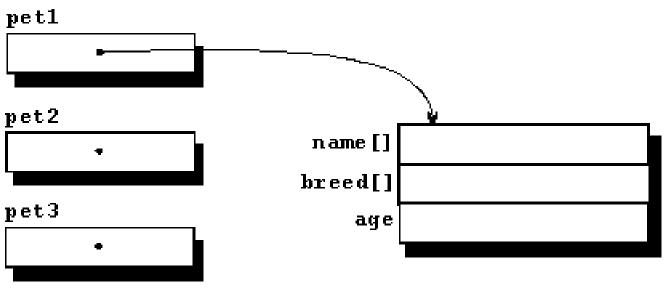

Figure 12-1

USING THE DYNAMICALLY ALLOCATED STRUCTURE

If you remember our studies of structures and pointers, you will recall that if we have a structure with a pointer pointing to it, we can access any of the variables within the structure. In lines 15 through 17 of the program, we assign some silly data to the structure for illustration. It should come as no surprise to you that these assignment statements look just like assignments to statically defined variables. Figure 12-2 illustrates the state of the data space at this point in the program execution.

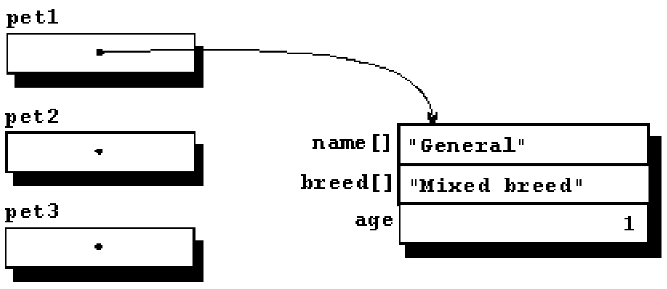

Figure 12-2

In line 19, we assign the value of **pet1** to **pet2** also via a pointer assignment statement which we introduced in chapter 8. This creates no new data, we simply have two pointers to the same object. Since **pet2** is pointing to the structure we created above, **pet1** can be reused to get another dynamically allocated structure which is just what we do next. Keep in mind that **pet2** could have just as easily been used for the new allocation. The new structure is filled with silly data for illustration in lines 22 through 24.

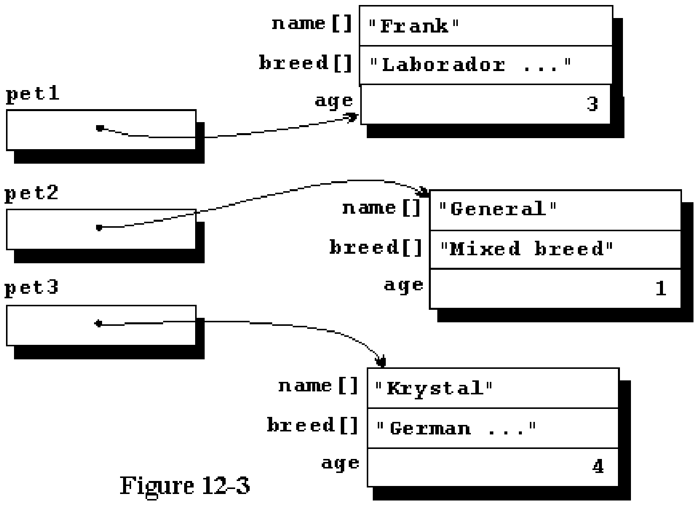

Figure 12-3

Finally, we allocate another block on the heap using the pointer **pet3**, and fill its block with illustrative data. Figure 12-3 illustrates the condition of the data space after execution of line 29 of the program.

Printing the data out should pose no problem to you since there is nothing new in the three print statements.

Even though it is not illustrated in this tutorial, you can dynamically allocate and use simple variables such as a single **char** type variable. This should be discouraged however since it is very inefficient.

GETTING RID OF THE DYNAMICALLY ALLOCATED DATA

Another new function is used to get rid of the data and free up the space on the heap for reuse, the function **free()**. To use it, you simply call it with the pointer to the dynamically allocated block of data as the only argument, and the block is deallocated.

In order to illustrate another aspect of the dynamic allocation and deallocation of data, an additional step is included in the program on your monitor. The pointer **pet1** is assigned the value of **pet3** in line 42. In doing this, the block that **pet1** was pointing to is effectively lost since there is no pointer that is now pointing to that block. It can therefore never again be referred to, changed, or deallocated. That memory, which is a block on the heap, is wasted from this point on. This is not something that you would ever purposely do in a program. It is only done here for illustration.

The first **free()** function call removes the block of data that **pet1** and **pet3** were pointing to, and the second **free()** call removes the block of data that **pet2** was pointing to. We therefore have lost access to all of our data generated earlier. There is still one block of data that is on the heap but there is no pointer to it since we lost the address to it. Figure 12-4 illustrates the data space as it now appears. Trying to free the data pointed to by **pet1** would result in an error because it has already been freed by the use of

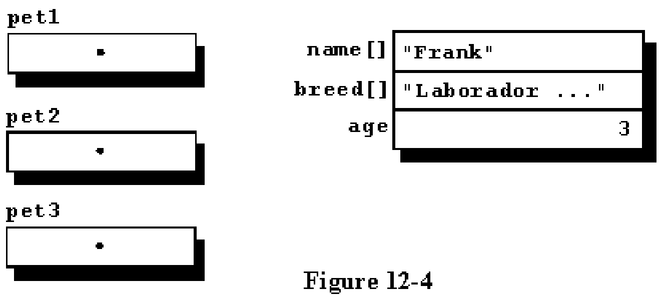

Figure 12-4

pet3. There is no need to worry, however. When we return to DOS, the entire heap will be disposed of with no regard to what we have put on it. The point does need to made that, if you lose a pointer to a block of the heap, it forever removes that block of data storage from our use and we may need that storage later.

Compile and run the program to see if it does what you think it should do based on this discussion.

THAT WAS A LOT OF DISCUSSION

It took six pages to get through the discussion of the last program but it was time well spent. It should be somewhat exciting to you to know that there is nothing else to learn about dynamic allocation, the last six pages covered it all. Of course, there is a lot to learn about the technique of using dynamic allocation, and for that reason, there are two more example programs to study. But the fact remains, there is nothing more to learn about the technique of dynamic allocation than what was given so far in this chapter.

AN ARRAY OF POINTERS

Load and display the file bigdynl.c for another example of dynamic allocation. This program is very similar to the last one since we use the same structure, but this time we define an array of pointers to illustrate the means by which you could build a large database using an array of pointers rather than a single pointer to each element. To keep it simple we define 12 elements in the array and another working pointer named **point**.

The ***pet[12]** is new to you so a few words would be in order. What we have defined is an array of 12 pointers, the first being **pet[0]**, and the last **pet[11]**. Actually, since an array is itself a pointer, the name **pet** by itself is a pointer to a pointer. This is valid in C, and in fact you can go farther if needed but you will get quickly confused. I know of no limit as to how many levels of pointing are possible, so the definition **int ****pt**; is legal as a pointer to a pointer to a pointer to a pointer to an integer type variable, if I counted right. Such usage is discouraged until you gain considerable experience.

Now that we have 12 pointers which can be used like any other pointer, it is a simple matter to write a loop to allocate a data block dynamically for each and to fill the respective fields with any data desirable. In this case, the fields are filled with simple data for illustrative purposes, but we could be reading from a database, from some test equipment, or from any other source of data.

A few fields are randomly picked in lines 24 through 26 to receive other data to illustrate that simple assignments can be used, and the data is printed out to the monitor. The pointer **point** is used in the printout loop only to serve as an illustration, the data could have been easily printed using the **pet[n]** means of definition. Finally, all 12 blocks of data are freed before terminating the program.

Compile and run this program to aid in understanding this technique. As stated earlier, there was nothing new here about dynamic allocation, only about an array of pointers.

A LINKED LIST

We finally come to the granddaddy of all programming techniques as far as being intimidating. Load the program dynlink.c for an example of a dynamically allocated linked list. It sounds terrible, but after a little time spent with it, you will see that it is simply another programming technique made up of simple components that can be a powerful tool.

In order to set your mind at ease, consider the linked list you used when you were a child. Your sister gave you your birthday present, and when you opened it, you found a note that said, "Look in the hall closet." You went to the hall closet, and found another note that said, "Look behind the TV set." Behind the TV you found another note that said, "Look under the coffee pot." You continued this search, and finally you found your pair of socks under the dogs feeding dish. What you actually did was to execute a linked list, the starting point being the wrapped present and the ending point being under the dogs feeding dish. The list ended at the dogs feeding dish since there were no more notes.

In the program dynlink.c, we will be doing the same thing your sister forced you to do. However, we will do it much faster and we will leave a little pile of data at each of the intermediate points along the way. We will also have the capability to return to the beginning and traverse the entire list again and again if we so desire.

THE DATA DEFINITIONS

This program starts similarly to the last two with the addition of the definition of a constant to be used later. The structure is nearly the same as that used in the last two programs except for the addition of another field within the structure in line 11, the pointer. This pointer is a pointer to another structure of this same type and will be used to point to the next structure in order. To continue the above analogy, this pointer will point to the next note, which in turn will contain a pointer to the next note after that.

We define three pointers to this structure for use in the program, and one integer to be used as a counter, and we are ready to begin using the defined structure for whatever purpose we desire. In this case, we will once again generate nonsense data for illustrative purposes.

THE FIRST FIELD

Using the **malloc()** function, we request a block of storage on the heap and fill it with data. The additional field in this example, the pointer, is assigned the value of NULL, which is only used to indicate that this is the end of the list. We will leave the pointer named **start** pointing at this structure, so that it will always point to the first structure of the list. We also assign **prior** the value of **start** for reasons we will see soon. Keep in mind that the end points of a linked list will always have to be handled differently than those in the middle of a list. We have a single element of our list now and it is filled with representative data. Figure 12-5 is the graphical representation of the data space following execution of line 24.

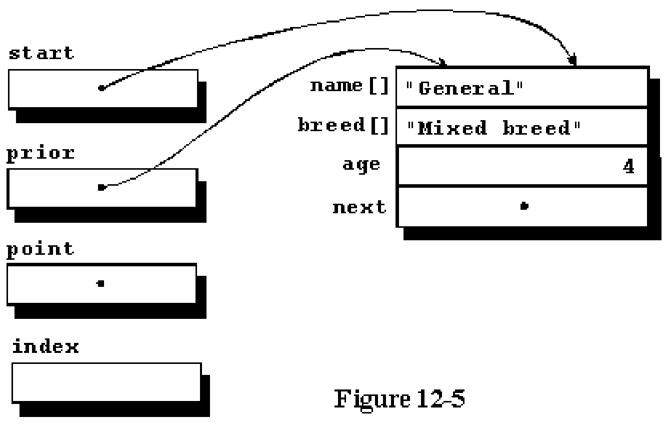

Figure 12-5

FILLING ADDITIONAL STRUCTURES

The next group of assignments and control statements are included within a for loop so we can build our list fast once it is defined. We will go through the loop a number of times equal to the constant RECORDS defined at the beginning of our program. Each time we go through the loop, we allocate memory, fill the first three fields with nonsense, and fill the pointers. The pointer in the last record is given the address of this new record because the **prior** pointer is pointing to the prior record. Thus **prior->next** is given the address of the new record we have just filled. The pointer in the new record is assigned the value NULL, and the pointer **prior** is given the address of this new record because the next time we create a record, this one will be the prior one at that time. That may sound confusing but it really does make sense if you spend some time studying it.

Figure 12-6 illustrates the data space following execution of the loop two times. The list is growing downward by one element each time we execute the statements in the loop. When we have gone through the for loop 6 times, we will have a list of 7 structures including the one we generated prior to the loop. The list will have the following characteristics.

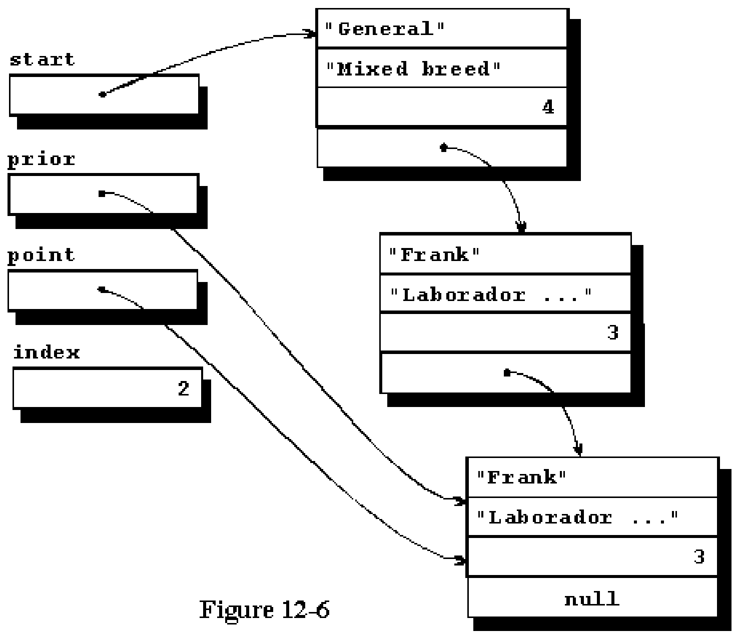

Figure 12-6

1. The pointer named **start** points to the first structure in the list.
2. Each structure contains a pointer to the next structure.
3. The last structure has a pointer containing the value NULL which can be used to detect the end.

It should be clear to you, if you understand the overall data structure, that it is not possible to simply jump into the middle of the list and change a few values. The only way to get to the third structure is by starting at the beginning and working your way down through the list one record at a time. Although this may seem like a large price to pay for the convenience of putting so much data outside of the program area, it is actually a very good way to store some kinds of data.

A word processor would be a good application for this type of data structure because you would never need to have random access to the data. In actual practice, this is the basic type of storage used for the text in a word processor with one line of text per record. Actually, a program with any degree of sophistication would use a doubly linked list. This would be a list with two pointers per record, one pointing down to the next record, and the other pointing up to the record just prior to the one in question. Using this kind of a record structure would allow traversing the data in either direction.

PRINTING THE DATA OUT

To print the data out, a similar method is used as that used to generate the data. The pointers are initialized and are then used to go from record to record, reading and displaying each record one at a time. Printing is terminated when the NULL in the last record is found, so the program doesn't even need to know how many records are in the list. Finally, the entire list is deleted to make room in memory for any additional data that may be needed, in this case, none. Care must be taken to assure that the last record is not deleted before the NULL is checked. Once the data is gone, it is impossible to know if you are finished yet.

MORE ABOUT DYNAMIC ALLOCATION AND LINKED LISTS

It is not difficult, nor is it trivial, to add elements into the middle of a linked list. It is necessary to create the new record, fill it with data, and point its pointer to the record it is desired to precede. If the new record is to be installed between the 3rd and 4th, for example, it is necessary for the new record to point to the 4th record, and the pointer in the 3rd record must point to the new one. Adding a new record to the beginning or end of a list are each special cases. Consider what must be done to add a new record in a doubly linked list.

Entire books are written describing different types of linked lists and how to use them, so no further detail will be given. The amount of detail given should be sufficient for a beginning understanding of C and its capabilities.

TWO MORE FUNCTIONS

Two additional functions must be mentioned, the **calloc()** and the **realloc()** functions. The **calloc()** function allocates a block of memory and clears it to all zeros which may be useful in some circumstances. It is similar to **malloc()** and will be left as an exercise for you to read about and use **calloc()** if you desire. Generally, you allocate a block of memory and immediately fill it with meaningful data so it wastes time to fill it with zeros in the **calloc()**, then fill it with real data. For this reason, the **calloc()** function is rarely used.

The **realloc()** is used to change the size of an allocated block, either bigger or smaller. You should ignore this until you gain a lot of experience with C. It is rarely used, even by experienced programmers.

Programming Exercise:

1. Rewrite the example program struct1.c from chapter 11 to dynamically allocate the two structures.
2. Rewrite the example program struct2.c from chapter 11 to dynamically allocate the 12 structures.

Index Previous Next . . . . *C Tutorial*

The Webwizard