The Four-Fold Path
Elimination of Pointer-Sorrow of Womankind

12th October 2004
Sathyaish Chakravarthy

This article applies to the C programming language. It presents a case where a pointer to a pointer to a character array is to be preferred over an ordinary character pointer, by highlighting the shortcomings of using an ordinary pointer in such a case.

Introduction

Often times, we need to pass a character array to a function in order that the called function works on the string, modifies it and returns it to the called function. The most common way of doing it is by using a character pointer. For the sake of demonstration, let us create a similar scenario by writing the code for two functions, one of which will call the other by supplying a character pointer as an argument, and will expect the called function to modify the contents of the character array pointed to by the argument. Have a look at the following snippet.

Code Listing 1

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

#define MAX_LEN 20
int ChangeString(char *);

int main(void)
{

        //Declare a character pointer
        char *Str;
        //Allocate memory to the character pointer
        Str=malloc(sizeof(char)*(MAX_LEN+1));

        //Write some text into the memory, and print it out
        strcpy(Str, "Money comes and goes.");
        printf("Original String: %s\n", Str);

        /*Modify the contents of the string by calling ChangeString() supplying Str
as an argument, and then print the changed Str out*/
        ChangeString(Str);
        printf("Modified String: %s\n", Str);

        return 0;
}

int ChangeString(char *TheString)
{
        /*Do something to change the contents of
        the string, so that TheString now becomes “Morality comes and grows
        Pseudocode:*/
        if (error) return –1;
        TheString=”Morality comes and grows.” //Pseudocode
        return 0;
}
```

What happens in the above-snippet is pretty simple. The *main()* function declares a character pointer called *Str*, to which it allocates some amount of memory by calling the *malloc()* function. At first, it writes the contents, **”Money comes and goes”** into the string. Then it calls the *ChangeString()* function, which is supposed to change the contents of the string to **“Morality comes and grows”**. Finally, the *main()* function prints out the new contents of the string to the VDU by calling the *printf()* function.

The desired output we want from the above function is:

```
Original String: Money comes and goes.
Modified String: Morality comes and grows.
```

You'd have noticed that the implementation of the *ChangeString()* function provided above is not correct. Instead, what has been provided is only pseudo-code. That has been done on purpose because that is the very subject of this article. We are heading for a discussion of the implementation inside the *ChangeString()* function, given an ordinary character pointer argument, and we will do that by considering a handful of paths, that will finally divulge to us, the shortcomings in the design of the argument to the *ChangeString()* function.

An alternative design could of course be letting the *ChangeString* function return a character pointer instead of an integer. However, we don't want to do that since it is important that we return an error code for success or failure.

A simple case: strcpy

The first and the foremost implementation is a case where we use the *strcpy()* function to copy the new contents to the memory pointed to by the character pointer received as an argument.

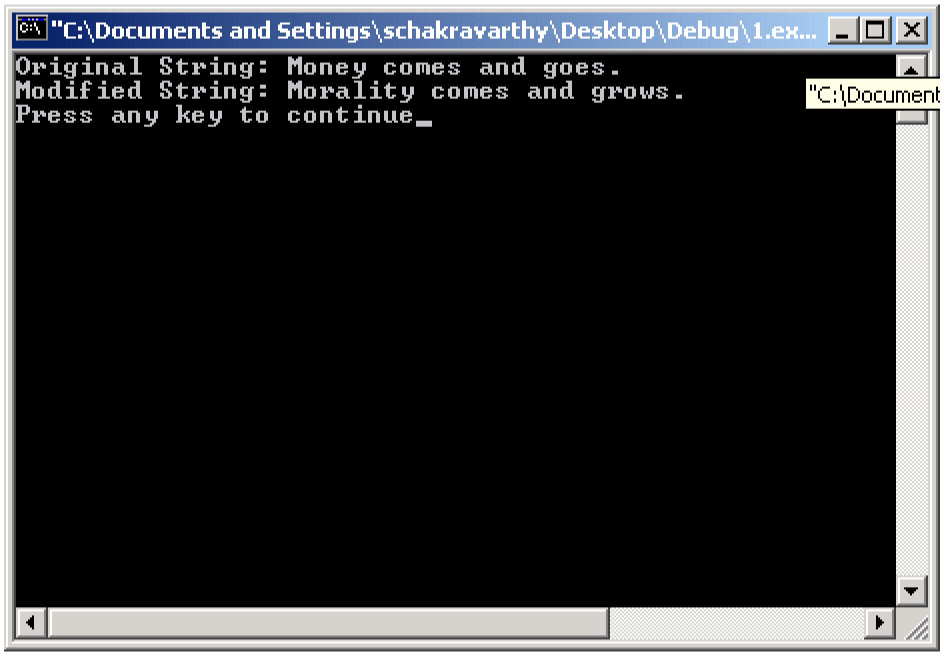
Let us revisit the code again for the *main()* and *ChangeString()* functions. Only this time, we will provide an actual implementation of the *ChangeString()* function, that works.

The code inside the main function will look exactly the same. So this time, let us concentrate on the code inside the *ChangeString()* function only. Look at **Code Listing 2** for the source code.

Code Listing 2

```
int ChangeString(char *TheString)
{
        /*If the original string itself is not valid,
        there's no way in hell we can over-write its contents,
        so we return an error.*/
        if (TheString==NULL) return -1;

        strcpy(TheString, "Morality comes and grows.");
        return 0;
}
```

That is so straightforward. I like it. Let us compile this code to see if it works.
And…it works. I swear! See **Figure 1** below.


"C:\Documents and Settings\schakravarthy\Desktop\Debug\1.ex...
"C:\Document

```
Original String: Money comes and goes.
Modified String: Morality comes and grows.
Press any key to continue_
```

Figure 1: The strcpy() function worked like a charm!

What did we do here? Remember that the *main()* function had declared a character pointer called *Str*. The *Str* was given memory by calling the *malloc()* function. Then it was written into by the *main()* function. Subsequently, it was handed over under the sobriquet *TheString*, to the function *ChangeString()*. That was the pedigree of this call.

Over here, in the *ChangeString()* function, we called the *strcpy()* function giving it the address of the first element of the character array in the memory, and the new content we wanted written onto that memory. The *strcpy()* function copied the contents **"Morality comes and grows"** into the memory. Both the functions, *main()* and *ChangeString()* were pointing to the same memory. As a result, when the function returned, the *main()* function got the modified contents from the same memory and printed them on the screen. This is how it worked. See **Figure 2**.

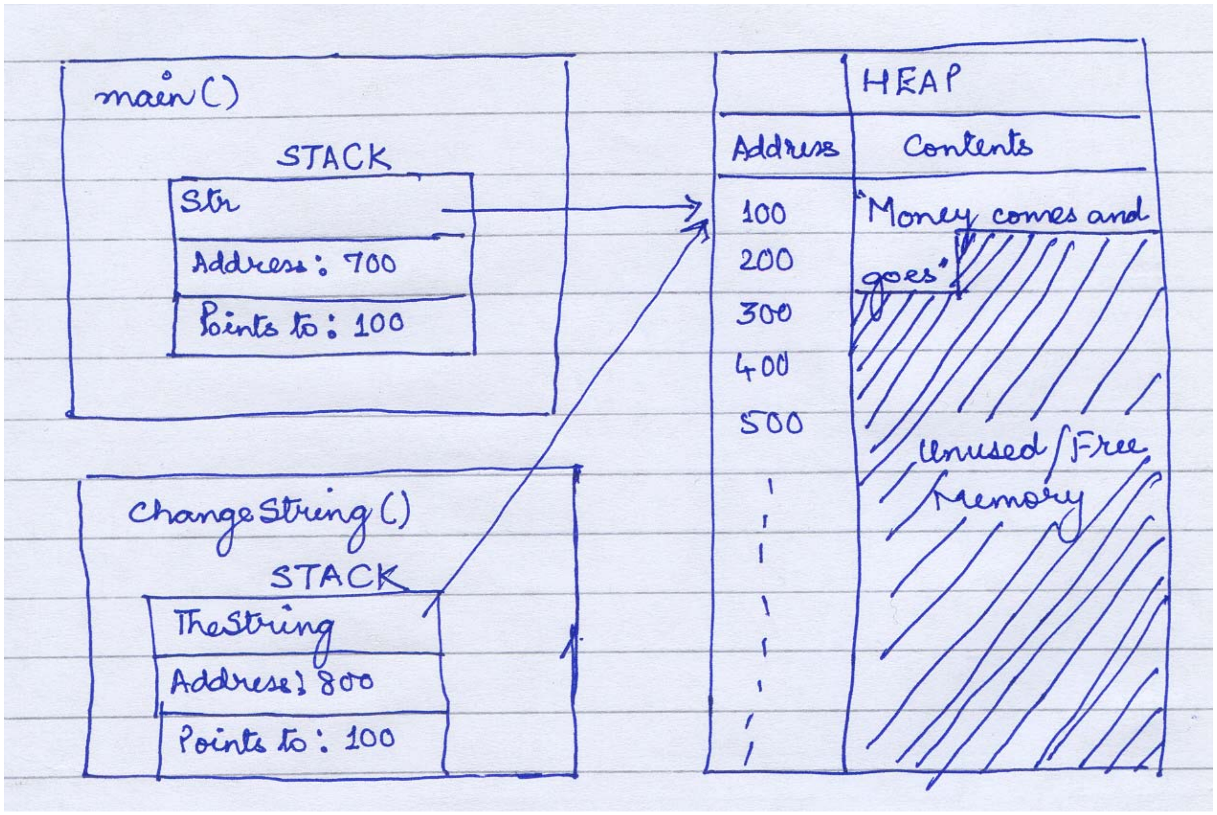


Figure 2: The memory diagram of Code Listing 2

I wish life too was that simple. But the truth is that I did not tell you the truth. I told you a white lie. What is a white lie? Well, a white lie is when you say only something out of the whole thing such that the something looks like a conclusive truth, whereas you are concealing much of the truth, which is in contradiction to your conclusion.

The confession I have to make is that I told you a pack of lies because I had to. I saved you from the nastiness of what lies ahead. This code actually is rife with danger.

A closer investigation of this code will tell us that we declared the character pointer in the *main()* function, and allocated to it only 21 units of memory. The *main()* function behaved well and wrote exactly 21 units of memory into the string, because **"Money comes and goes"** measures exactly 20 units and 1 unit is taken up by the null terminator. However, when the function *strcpy()*, it did not re-allocate new memory to the string. It blindly wrote on the old string without bothering about the memory allocated. The string **"Morality comes and grows"** is 24 units of memory and its null-terminator is 1 unit, thereby adding up to 25 units of memory. Instead of reporting an error about insufficient memory, *strcpy()* so unthinkingly overwrote unowned memory from the heap.

It did? Yes. Let's take a look at the documentation of this function strcpy().

char * strcpy (char * ***dest*****, const char *** ***src*** **);**
dest should have enough memory space allocated to contain *src* string.

The documentation clearly states that the destination parameter must have enough memory to hold the contents to be copied. If they do not, however, the function will not complain. It will continue writing, without a second thought as to what the previous contents of the memory were, thus causing a buffer over-run.

One question you might have in your mind at this point is how did it display the two strings correctly then? What was the Figure 1 all about? Did I forge it?

It worked because we were lucky that the memory it infringed upon did not belong to another variable. But like everyday is not a Sunday, we won't be lucky every time.

Having failed with *strcpy()*, it is time for some retrospection on that code. Why did we fail? What intent of that code was culpable?

The answer I got from asking myself this question was, "Son! You failed because you did not articulate your needs before you expended them."

"What are my needs, O Noble One?" I inquired.

"*strlen(***"Morality comes and grows"***)*+1, my child! Beware of your needs and they shall come to fruition", The Voice proclaimed.

"Eureka", I ejaculated.

"Rejoice!"

With that overwhelming revelation from the Liberated One, I think we are to celebrate a new perspective. Let us articulate the memory we need before we expend it.

Second case: Becoming aware of our needs

But if you have programmed for sometime, you would know that it is not always as easy as said in order to determine at run-time, just how much memory you need. That is because the strings that we have to write in complex programs are not given to us just as the string **"Morality comes and grows"** is given in our example. In real life programs, we may have to perform some arithmetic on the state of other variables in order to arrive at our memory requirements. Either ways, we have to make good guesses about memory requirements. However, in our simple case, we can go by counting the number of characters in the new string we want to write, and that is 25. Given that there is a null-terminator too, we need exactly 26 units of memory.

You might be thinking to yourself, "Tell you what, I know what we're going to do now. We're going to free *TheString* by calling the *free()* function on it, then we're going to give it 26 new units of memory and finally copy the new string into the those 26 units. Just like **Code Listing 3**".

Code Listing 3

```
free(TheString);
int Len=strlen("Morality comes and grows");
```

```
TheString=malloc(sizeof(char)*(Len+1));
strcpy(TheString, "Morality comes and grows");
return 0;
```

You're *nearly* right. But we won't do that.

Would it be nice if we *freed* the argument, and subsequently *malloc()* failed on it and returned NULL? What would happen to *TheString* then? Wouldn't it be rude to destroy even the original contents of the argument if we could not change the contents? Imagine your father handed you five bucks and asked you to escort your younger brother to get him a crew cut. To save the five bucks for your own beer, you tried the haircut on your brother yourself. You got it all wrong, and to cover it up you had to give his head a clean shave. How would that be?

If we corrupted the original string, we would be doing the clean shave hair cut instead of what was ordered. So, to be sure we can go on with the hair cut, we could try our hands with the crew cut on someone else, whose head has the exact shape of that of your little brother's head. If it works out, we'll do it on the kid brother. If it doesn't, we'll have the beer instead.

Thus is ordained the third path - the path of elimination of sorrow. Well, pointer-sorrow; not the sorry of mankind.

The Third Case: Playing Safe

So, to be nice and polite to the calling function, we will first create new memory equal to the size of the new string. If the new memory is allocated successfully and the new contents are copied into the new string, then we could go ahead and dereference that new memory to the argument received by the *ChangeString()* function. That would guarantee to us that the *ChangeString()* function either:

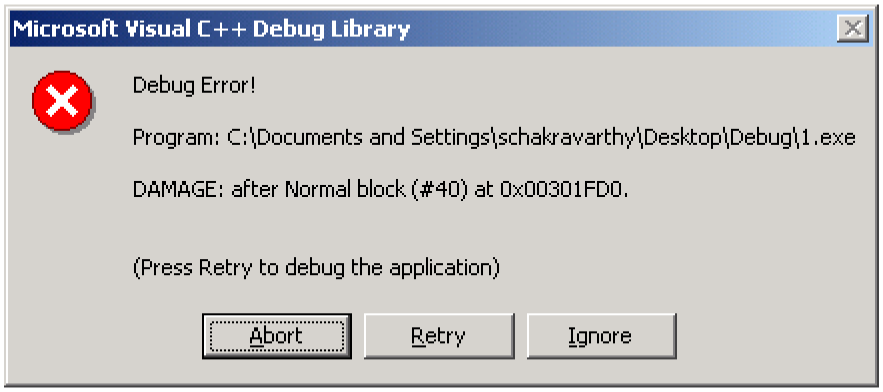
(1) Was not successful at changing the contents of its argument *TheString*. However, it did not destroy the old contents of the argument; or
(2) That the function *ChangeString()* was successful at modifying the contents of its argument.

It would not leave us into a situation wherein the function *ChangeString()* was not able to modify the contents of it's argument *TheString* and it also destroyed the original contents of the argument *TheString* such that the argument now points to an invalid memory location.

Code Listing 4

```
int ChangeString(char *TheString)
{
        int Len=0;
        char *NewString;

        /*If the original string itself is not valid,
        there's no way in hell we can over-write its contents,
        so we return an error.*/
        if (TheString==NULL) return -1;

        /*Next, we do just as the Holy One said, we articulate our needs
        before we expend them*/
        Len=strlen("Morality comes and grows.");

        /*We declare a new character pointer, give it the required amount of
        memory and write the new string to it.*/
        NewString=malloc(sizeof(char)*(Len+1));
        strcpy(NewString, "Morality comes and grows.");

        /*Free the reference of TheString to the old memory block,
        and dereference it to the new block of memory just acquired*/
        free(TheString);
        TheString=NewString;

        return 0;
}
```

But when you compile this code, it does not work. Instead, it gives you a message like the one shown below in **Figure 3**.



Microsoft Visual C++ Debug Library

```
Debug Error!

Program: C:\Documents and Settings\schakravarthy\Desktop\Debug\1.exe

DAMAGE: after Normal block (#40) at 0x00301FD0.

(Press Retry to debug the application)
```

Figure 3: The code in Code Listing 4 crashes the program with an error message like this.

We reasoned out a solution, and we were quite sure it would work, but it didn't. Time for some soul-searching yet again. I mean code searching.

But before we open our diagnostic kit, it would be nice if we learnt the mechanics of memory management in C. What I am refering to is the inner working of the *malloc()* and *free()* functions in C. Once we know what exactly happens inside these functions, and how these functions affect the state of the pointers in our programs, it will give us useful insight into spotting the error in our approach.

Let me begin by quoting some of the prominent authorities on software programming, particularly the C language. The famous book on C, The C Programming Language, by **Brian Kernighan and Denis Ritchie**, the creators of the C language says that *malloc()* maintains a linked list of memory chunks that are available for use by our program. Every time *malloc()* is called with some size, it walks through the linked list to find the "first fit" of memory that fits the size demanded, rather than the "best fit". It starts with lower addresses and walks up to the higher addresses.

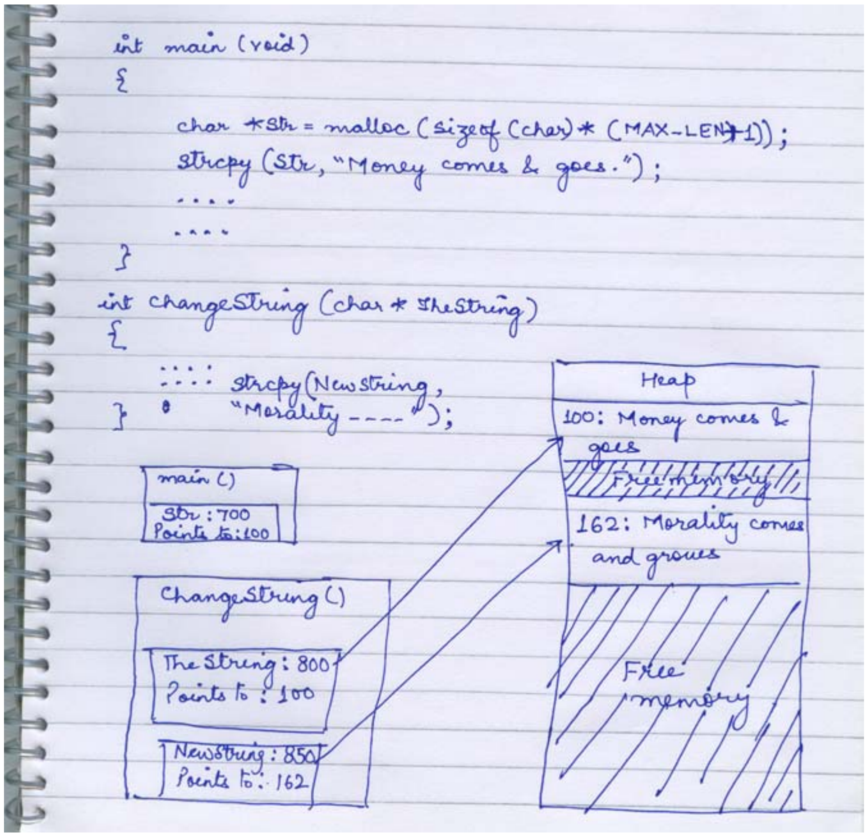
Here's what Joel Spolsky, a software development guru, who writes a weblog called Joel On Software has to say about malloc.

Joel says,

"Do you know how **malloc** works? The nature of **malloc** is that it has a long linked list of available blocks of memory called the *free chain*. When you call **malloc**, it walks the linked list looking for a block of memory that is big enough for your request. Then it cuts that block into two blocks -- one the size you asked for, the other with the extra bytes, and gives you the block you asked for, and puts the leftover block (if any) back into the linked list. When you call **free**, it adds the block you freed onto the free chain. Eventually, the free chain gets chopped up into little pieces and you ask for a big piece and there are no big pieces available the size you want. So **malloc** calls a timeout and starts rummaging around the free chain, sorting things out, and merging adjacent small free blocks into larger blocks. This takes 3 1/2 days. The end result of all this mess is that the performance characteristic of **malloc** is that it's never very fast (it always walks the free chain), and sometimes, unpredictably, it's shockingly slow while it cleans up. (This is, incidentally, the same performance characteristic of garbage collected systems, surprise surprise, so all the claims people make about how garbage collection imposes a performance penalty are not entirely true, since typical **malloc** implementations had the same kind of performance penalty, albeit milder.)"

With that, we are ready to do a post-mortem on the implementation in **Code Listing 4**. Let us go back to have a look at what the code in **Code Listing 4** did and why it did not work.

Look at **Figure 4** and see the status of the program memory.



```
int main (void)
{
    char *str = malloc (sizeof (char) * (MAX-LEN+1));
    strcpy (str, "Money comes & goes.");
    ....
    ....
}
int ChangeString (char * TheString)
{
    .... strcpy(NewString,
    ....        "Morality ----");
}
```

Figure 4: The status of the program memory when the ChangeString() function is called.

This is how the memory looked like when the *ChangeString()* function was called. The memory addresses are, of course, faked in the above diagram. Each function has a local stack that carries the local variables that are declared inside the function body.

In the *main()* function's local stack, there was a local variable called *Str*, which was of the type of a character pointer. When we called *malloc()* on this pointer, it allocated 21 units of memory from the heap. The heap is a global area of memory outside any function, but within a process. All pointers are allocated memory from the heap. The memory that was allocated to the *Str* pointer was written onto with the *strcpy()* function inside the *main()* function.

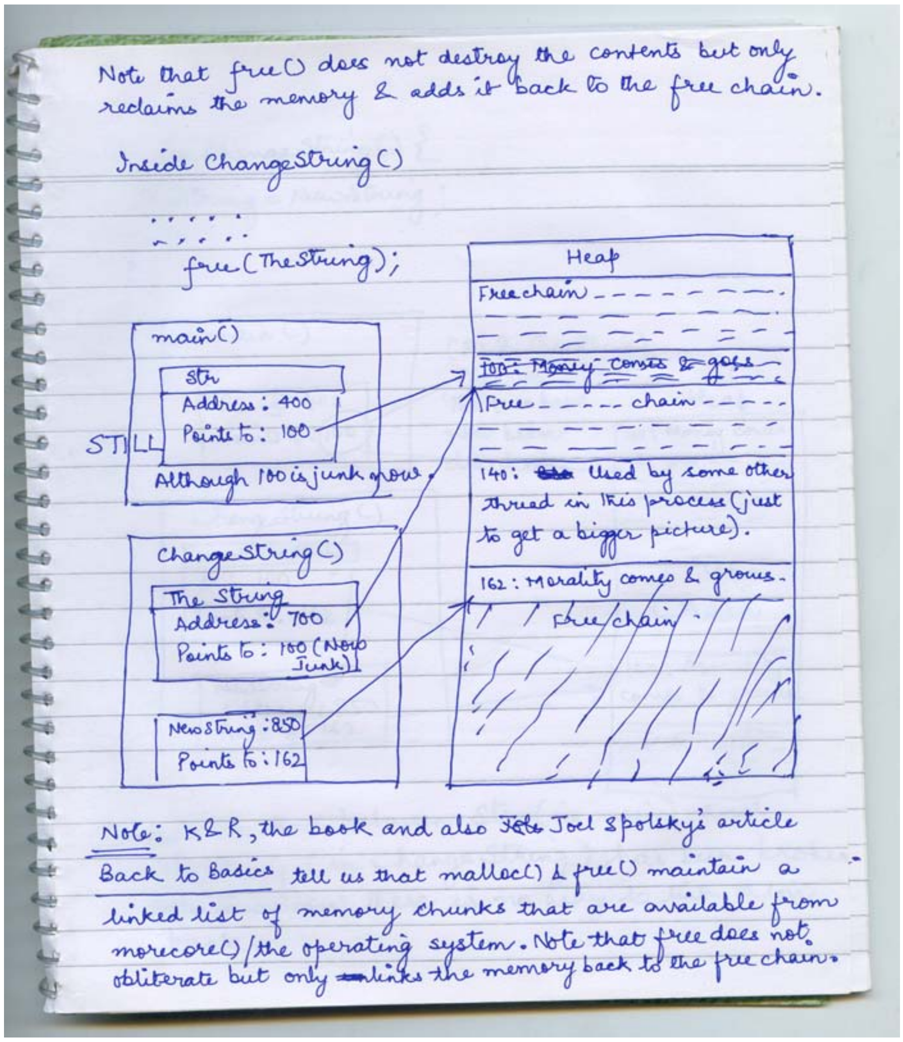
Next, we called the *ChangeString()* function with the *Str* pointer passed as an argument. It is important to note here that although the pointers *Str* and *TheString* point to the same memory, they are different variables in themselves and reside in the memory independent of each other. The *Str* variable is local in scope to the *main()* function whereas the pointer *TheString* is local to the *ChangeString()* function, and goes out of scope once the control is out of the *ChangeString()* function. *TheString* resides on the local stack of the *ChangeString()* function. Consequently, *Str* has a different memory address than *TheString*.

In our diagram above, *Str* has the address 700, whereas the variable *TheString* has the address 800. They both point to the same area in the heap, which is at the base address 100. This is where the string **"Money comes and goes"** starts.

Hell breaks loose when we call *free()* on the argument *TheString*. As noted above, free reclaims the used memory and adds it up to the linked list called the *free chain*. It however does not destroy any content written on that memory.

Figure 5 below shows what the memory of our program looks like immediately after the call to *free()* on *TheString* inside the *ChangeString()* function.

You can see that both, the pointer *Str* that is local to the *main()* function and the pointer *TheString*, which is local to the *ChangeString()* function point to a common address 100. But because *free()* has been called on the variable *TheString*, the memory pointed to by both these pointers at location 100 in the heap is no longer valid because it is now a part of the free chain. When another *malloc()* is called, this memory may be used to fulfil that memory request.


Note that free() does not destroy the contents but only reclaims the memory & adds it back to the free chain.

Inside ChangeString()

```
. . . . .
. . . . .
free(TheString);
```

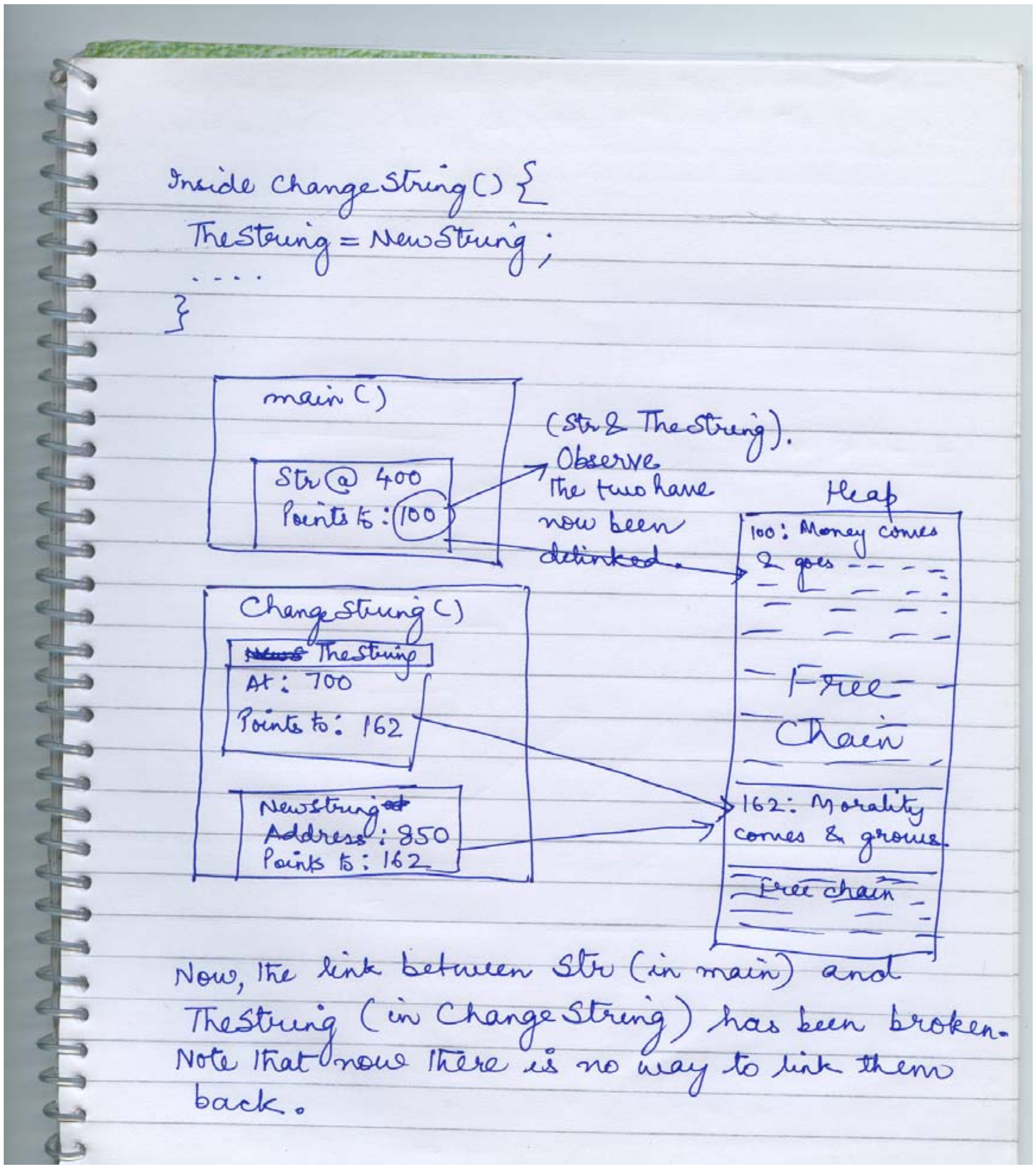
Note: K&R, the book and also ~~Jole~~ Joel Spolsky's article Back to Basics tell us that malloc() & free() maintain a linked list of memory chunks that are available from morecore()/the operating system. Note that free does not obliterate but only ~~un~~links the memory back to the free chain.

main()
Str
Address: 400
Points to: 100
STILL
Although 100 is junk now.
ChangeString()
The String
Address: 700
Points to: 100 (Now Junk)
NewString: 850
Points to: 162
Heap
Freechain
~~100: Money~~ comes & goes
Free chain
140: ~~the~~ Used by some other thread in this process (just to get a bigger picture).
162: Morality comes & grows.
Free chain

Figure 5: The program memory immediately after a call to free() on TheString arugment inside the ChangeString() function.

However, so far both the pointers *Str* and *TheString* still point to the same memory address, albeit an invalid one. The final blow comes from calling dereferencing *TheString* to the memory pointed to by *NewString* inside the *ChangeString()* function. At this point, all ties in between the two pointers *Str* and *TheString* that were to point to the same location are broken. As you would see from **Figure 6** below, after the assignment, both *TheString* and *NewString* now point to the location 162, whereas the pointer *Str* that is local to the *main()* function still points to the old, albeit invalid, location 100.

Figure 6 shows the memory status of the program after this pointer assignment.



```
Inside ChangeString() {
 TheString = NewString;
 ....
}
```

Now, the link between Str (in main) and TheString (in ChangeString) has been broken. Note that now there is no way to link them back.

Figure 6: The status of the program memory immediately after the pointer assignment of TheString to the contents of NewString.

The Fourth Case: Repeating Old Mistakes

In the last approach, we saw that we were nearly there but the call to *free()* inside the *ChangeString()* function spoilt the show because it unlinked the two pointers *Str* and *TheString*.

If you're still not convinced that the pointers *Str* and *TheString* are not the same even though they were meant to point to the same location, you may try this little snippet of code to see it for yourself.

Code Listing 5

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

int ChangeString(char* TheString);
int main(void)
{
	char* Str=malloc(sizeof(char)*10);
	strcpy(Str, "Hello!");
	printf("Address of Str: %d\nStr points to: %d\nString representation at
that address: %s\n\n", &Str, Str, Str);
	ChangeString(Str);
	printf("%s\n", Str);
	return 0;
}

int ChangeString(char* TheString)
{
	printf("Address of TheString: %d\nTheString points to: %d\nString
representation at that address: %s\n\n", &TheString, TheString, TheString);
	/*We're not changing the string here. This code is just to prove that the two
pointers Str and TheString are indeed different.*/
	return 0;

}
```

Another approach that some programmers take it similar to what we discussed in the previous approach with the only difference that after a call to free() on the argument TheString, instead of the assignment of the address of NewString to TheString, new memory is allocated to TheString by calling malloc() on it and then the new string is copied into TheString argument.

You'd have guessed by now that this approach is also sure to yield the wrong results because it does not solve the problem we had in the third approach. You are right. It does not solve that problem. Instead, it helps worsen it. That is because after the two pointers Str and TheString point to an invalid location because of the call free() on TheString, they are still pointing to the same invalid location, but when a call for new memory is made by calling malloc() on TheString, the new memory is allocated from the free chain, which returns any other address that may or may not be the same as the old address to which both the pointers pointed.

In the last example, even though the free had been called, and the free chain had reclaimed the memory at the address 100 in the heap, the contents at that address were not destroyed. More importantly, the pointers *Str* and *TheString* still pointed to the address 100, which of course was a part of the free chain now. But there's no saying till when the contents of that address will remain intact. Probably until *malloc()* walks the free chain the next time to find the first fit for it's next adventurous assignment.

The misfortune is called upon with the call to malloc()where by TheString gets a new memory address to point to, and there is no way Str, the local pointer in the main() function, can point to or even know about this new memory address allocated to TheString.

Code Listing 6 below demonstrates this fourth approach.

Code Listing 6

```
int ChangeString(char *TheString)
{
        int Len=0;
        char *NewString;

        /*If the original stirng itself is not valid,
        there's no way in hell we can over-write its contents,
        so we return an error.*/
        if (TheString==NULL) return -1;
        /*Next, we do just as the Holy One said, we articulate our needs
        before we expend them*/
        Len=strlen("Morality comes and grows.");

        /*We declare a new character pointer, give it the required amount of
        memory and write the new string to it.*/
        NewString=malloc(sizeof(char)*(Len+1));
        strcpy(NewString, "Morality comes and grows.");

        /*Free the reference of TheString to the old memory block*/
        free(TheString);

        /*Allocate new memory to TheString and copy the contents of
        the new string into the new memory of TheString*/
        TheString=malloc(sizeof(char)*(Len+1));
        if(TheString==NULL) return -1;
        strcpy(TheString, NewString);

        //Free the temporary variable NewString
        free(NewString);

        return 0;
}
```

In order to prove for yourself that *malloc* assigns a new memory to *TheString*, of which *Str* is oblivious, you can run the source code provided in **Code Listing 7** below.

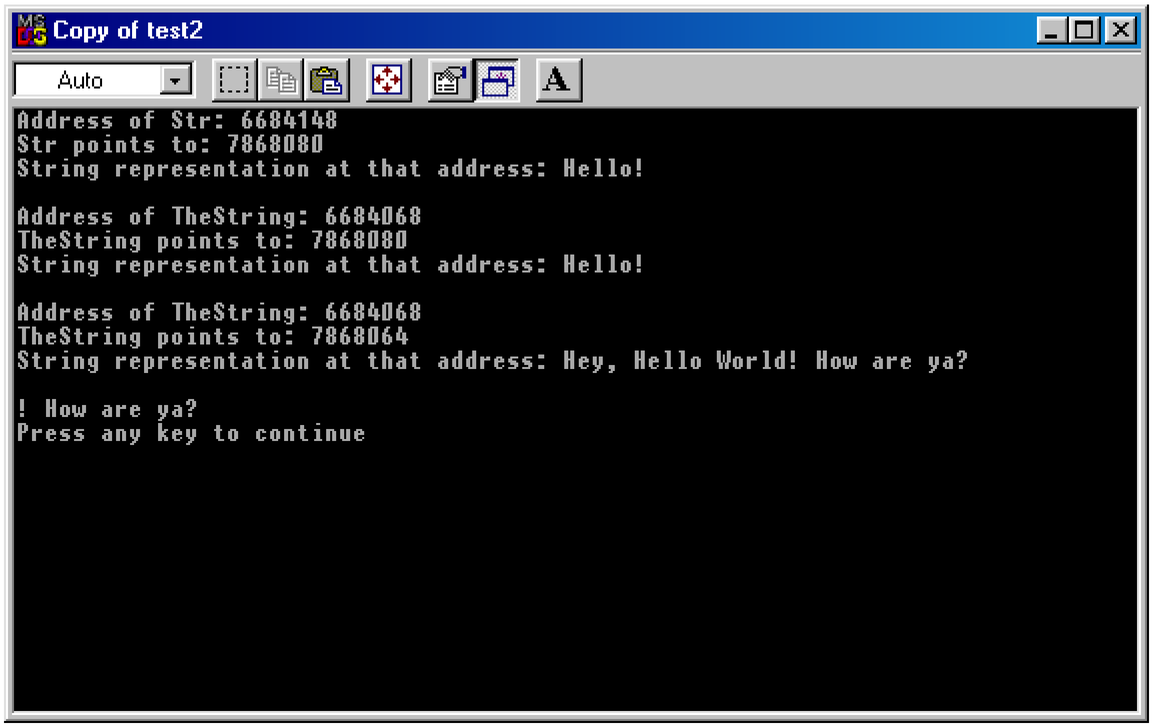
Code Listing 7

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

int ChangeString(char* TheString);
int main(void)
{
        char* Str=malloc(sizeof(char)*10);
        strcpy(Str, "Hello!");
        printf("Address of Str: %d\nStr points to: %d\nString representation at
that address: %s\n\n", &Str, Str, Str);
```

```
        ChangeString(Str);
        printf("%s\n", Str);
        return 0;
}

int ChangeString(char* TheString)
{
        printf("Address of TheString: %d\nTheString points to: %d\nString
representation at that address: %s\n\n", &TheString, TheString, TheString);
        free(TheString);
        TheString=malloc(sizeof(char)*25);
        strcpy(TheString, "Hey, Hello World! How are ya?");
        printf("Address of TheString: %d\nTheString points to: %d\nString
representation at that address: %s\n\n", &TheString, TheString, TheString);
        return 0;

}
```

When you run the above code listing, you might note an interesting fact about the memory allocation done by *malloc*() and the collection of freed memory into the free chain by *free*().

To appreciate what's going on, take a look at **Figure 7** below, which shows a snapshot of the output window on my computer when I ran the code in **Code Listing 7**.



```
Address of Str: 6684148
Str points to: 7868080
String representation at that address: Hello!

Address of TheString: 6684068
TheString points to: 7868080
String representation at that address: Hello!

Address of TheString: 6684068
TheString points to: 7868064
String representation at that address: Hey, Hello World! How are ya?

! How are ya?
Press any key to continue
```

Figure 7: The output of the code in Code Listing 7

The results in the above figure prove that a call to the *free()* function does not destroy or overwrite the contents of the memory pointed to by a pointer, but instead simply reclaims that target memory into the free chain.

Observe that the original contents the memory pointed to by *TheString* are **"Hello!?"** and then the memory is freed and re-allocated to *TheString* and the new contents written onto it are **"Hey, Hello World! How are ya?"**. Note that the old address that both *Str* and *TheString* point to is the same, i.e. 7868080, in my case. But when the new memory is granted to *TheString*, the address that *TheString* points to is now changed to 7868064, which is exactly 16 bytes before it's old address, which is also the address of the memory pointed to by *Str*. But the length of the new string is more than 16 bytes, and is 29 bytes. Therefore, that means that the address 7868080 is now a part of the memory granted to *TheString* from the free chain. However, *Str* still points to this memory and that is why after a return from the *ChangeString*() call, when we print the contents pointed to by *Str*, it prints, **"! How are ya?"** which is exactly 16 bytes less than the length of the new contents of the string pointed to by *TheString*.

Nirvana: Pointer to the pointer

We've seen four paths and yet we're back to where we started.

The four-fold approach I have shown you above may be enough to convince you that an ordinary pointer is not suitable in such a case. As I have made it clear at the very outset of this writing, another alternative approach is to make the function *ChangeString*() return a character pointer instead and be done with it. However, in cases where an integer value needs to be returned to signal success or to return specific error information, in such cases, it is not suitable to take a pointer to a character array as an argument, where such memory pointed to needs to be changed or re-assigned.

Fortunately for us, the last mistake we made was very insightful. We saw that the pointers *Str* in the *main()* function and the pointer *TheString* in the *ChangeString()* function were independent. And that the new assignment on *TheString* inside the *ChangeString()* or a call to malloc() on TheString broke the link between the two. Also, the variable *TheString* was local to *ChangeString()*, no matter what we did with it. If we could somehow maintain a link between the two pointers *Str* and *TheString*, we'd obtain to Pointer-Actualization.

The remedy in such a case is to conjure an algorithm that lets you hold on to the memory pointed to by the calling function; in this case the memory pointed to by *main()* function's *Str* variable. This can be done if we pass a copy of the actual address of the pointer *Str*. In such a case, the called function could retain the address where *Str* actually resides and so it would never loose track of the memory pointed to by *Str*.

Code Listing 8 below shows the code to achieve the desired affect. The lines of code that are different from the previous snippet have been typed in bold face.

Code Listing 8

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

#define MAX_LEN 20
int ChangeString(char **TheString);

int main(void)
{
        //Declare a character pointer
        char *Str;

        //Allocate memory to the character pointer
        Str=malloc(sizeof(char)*(MAX_LEN+1));

        //Write some text into the memory, and print it out
        strcpy(Str, "Money comes and goes.");
        printf("Original String: %s\n", Str);

        /*Here we pass the address of Str and not a copy of Str*/
        ChangeString(&Str);
        Printf("Modified String: %s\n", Str);

        Return 0;
}


int ChangeString(char **TheString)
{
        int Len=0;
        char *NewString;

        /*If the original stirng itself is not valid,
        there's no way in hell we can over-write its contents,
        so we return an error.*/
        if (*TheString==NULL) return -1;

        /*Next, we do just as the Holy One said, we articulate our needs
        before we expend them*/
        Len=strlen("Morality comes and grows.");

        /*We declare a new character pointer, give it the required amount of
        memory and write the new string to it.*/
        NewString=malloc(sizeof(char)*(Len+1));
        Strcpy(NewString, "Morality comes and grows.");

        /*Free the reference of TheString to the old memory block*/
        free(*TheString);

        /*Allocate new memory to TheString and copy the contents of
        the new string into the new memory of TheString*/
        *TheString=malloc(sizeof(char)*(Len+1));
        if(*TheString==NULL) return -1;
        strcpy(*TheString, NewString);

        //Free the temporary variable NewString
        free(NewString);

        return 0;
}
```

Note that the parameter declaration of the function *ChangeString*() has now been changed to a double-pointer. Therefore, the variable *TheString* now is not a pointer to a character array. Instead it is intended to hold the address of such a pointer. Notice that when calling *ChangeString*() from the *main*() function, we pass the address of the variable *Str* by prefixing the ampersand (&) sign in front of it.

In affect, going by our memory addresses in **Figure 6**, *Str* resides at address 400, *TheString* at 700, and *NewString* at 850. However, what they point to has been changed. *Str* still points to memory location 100. *TheString* now points to the address at which *Str* resides, which is 400. *NewString* points to address 162.

When the call to *free*() on **TheString* is made, the memory pointed to by the pointer whose address is stored in *TheString* is freed. *TheString* points to 400, which is the address at which *Str* resides. As a result, the memory pointed to by *Str* is freed. When *malloc*() is called on **TheString*, the pointer that resides at address *TheString* (or 400) is allocated new memory from the heap. Thus, the same pointer *Str* is again allocated new memory and retains it.