Pointers in C

Notation First

To make this explanation clearer, we're going to look some examples of source code, and draw pictures to show what is happening in memory. To keep things consistent, we're going to develop a pictorial notation that will be used throughout this discussion. Let's start off with an example.

```
int main(void) {
    int a = 7, b = 5, c;
    c = a - b;
    printf("%d\n", c);
    return 0;
}
```

Source Code 1

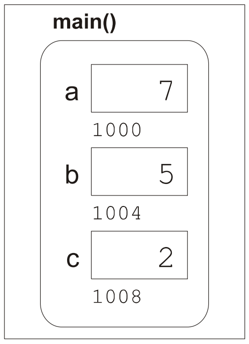

Diagram 1

This is a very simple program. The diagram on the right shows what is happening in memory. The rectangular boxes indicate memory locations. The value inside a box indicates the value in that memory location. The value under a box is the memory address of that box. In this case, we have three boxes – one for each of the three variables in main(). The name of each variable has been identified to the left of each box. Not all boxes will have names: memory locations allocated by malloc() will be nameless. Finally, all of these boxes represent local variables within the function main(). We have represented this with a rounded rectangle.

What's In a Pointer? – Address-of and Dereference Operators

Using the notation above, a pointer is nothing more than a box that contains a value that is *underneath* some other box. There are two basic operators for dealing with pointers in C. One is the address-of operator '&'. If you give it a box, it returns the value *underneath* it. The other operator is the dereference operator '*'. This does exactly the opposite thing. If you give it a box, it takes the value inside that box, then goes and finds the other box that has that value underneath it. Here's an example:

```
1: int main(void) {
2:     int num;
3:     int *pNum;
4:     num = 57;
5:     pNum = &num;
6:     *pNum = 23
7:     printf("%d\n", num);
8:     return 0;
9: }
```

Source Code 2

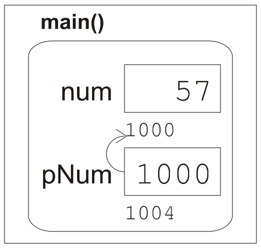

Diagram 2a
After line 5.

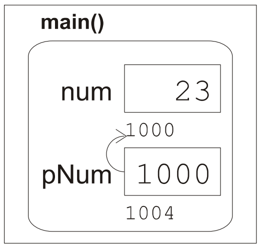

Diagram 2b
At end.

Look first at Diagram 2a. This shows the state of the program memory just after line 5 has been executed. The variable `num` was assigned the value 57 at line 4. At line 5, the address-of operator is used to retrieve the address of the variable `num`, and store this value in `pNum`. Since `num` resides at memory location 1000 (as is shown under the box), the value 1000 is put in `pNum`. `pNum` is now “pointing to” `num`. This has been indicated on the diagram with an arrow. Note how the variable `pNum` was declared in line 3. The asterisk (*) there is *not* the dereference operator. In this context, it is used to tell the C compiler that `pNum` is not an `int`, but is a *pointer to* an `int` (an `int*`). Do not think of the arrow as a pointer. It’s better to think of the box `pNum` as the pointer. A pointer is just like any other variable; the arrow is just there to help you find the box it is pointing to.

Now look at Diagram 2b. This shows the program memory when the program is finished. The dereference operator is used at line 6. `*pNum` tells the compiler to take the value inside `pNum` (which is 1000), and go get the box with that value underneath it. That box happens to be the one labelled “num” in this case. The rest of line 6 tells the compiler to put the value 23 inside that box. If you run this program, the number 23 will be printed. The value of `num` has been changed.

Parameter Problems

Suppose we need to write a function that changes the value of one of its parameters. We can’t do this directly, because in C, parameters are passed by value. Here is a code example that does not work properly. What we intend is for the function changeBad() to change the value of `num` in main(). This does not happen.

```
1: void changeBad(int value) {
2:      value = 23;
3: }
4: int main(void) {
5:      int num = 17;
6:      change(num)
7:      printf(“%d\n”, num);
8:      return 0;
9: }
```

Source Code 3
Code to change a parameter value (incorrect).

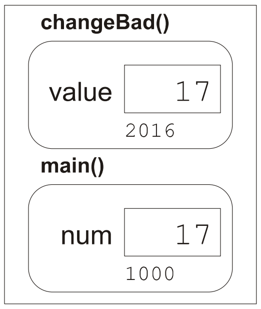

Diagram 3a
After line 6:1.

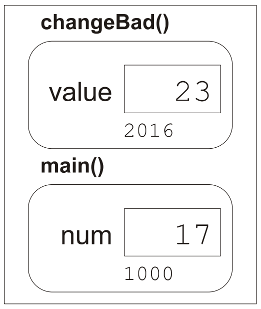

Diagram 3b
After line 6:2.

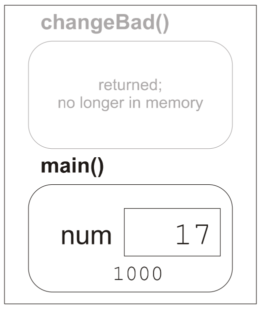

Diagram 3c
At end.

In Diagram 3a, we are at line 2 – just inside the call to changeBad() at from 6. Notice that we have two boxes in the picture: `num` in main(), and `value` in changeBad(). These are two different boxes, at two different memory locations. This illustrates the fact that parameters are passed by value. In Diagram 3b, changeBad() modifies the value of `value`. But `value` is a parameter of changeBad(); so, when we return from changeBad() in Diagram 3c, the value of `num` is left unchanged.

Source Code 3 does not do what we want it to. Source Code 4 corrects the problem:

```
1: void change(int *pValue) {
2:     *pValue = 23;
3: }
4: int main(void) {
5:     int num = 17;
6:     change(&num)
7:     printf(“%d\n”, num);
8:     return 0;
9: }
```

Source Code 4
Code to change a parameter value (correct).

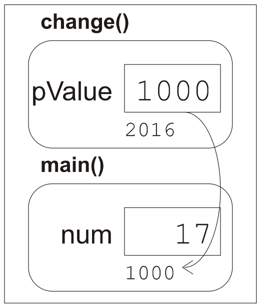

Diagram 4a
After line 6:1.

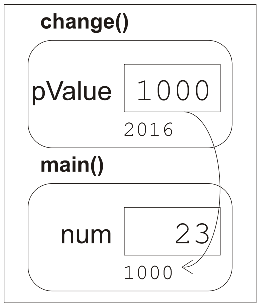

Diagram 4b
After line 6:2.

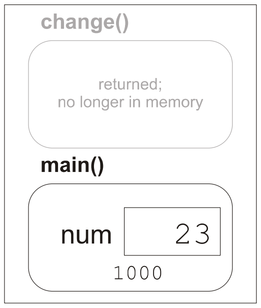

Diagram 4c
At end.

Here, the parameter to change() is not an `int`, but an `int*` (an `int` pointer). In Diagram 4a, `pValue` has the value 1000, which is the *address of* `num`. In Diagram 4b, we have just executed line 2. Line 2 *dereferences* `pValue`, and assigns 23 to the result. Dereferencing `pValue` (taking `*pValue`) returns the “box” at address 1000, which is the box labelled “num”. Therefore, when we assign a value to this box, we are changing the value of `num`. In Diagram 4c, change() has exited, and the value of `num` has been changed.

Double Trouble

We’ve seen some examples of what pointers are used for. Now we’re going to look at some examples of where you might want to use double pointers. Source Code 5 is an example of a data structure you might use for a singly-linked list. Suppose we want to write a function to add a new item to the head (beginning) of the list. Here, we’re going to run into the same problem we saw in the previous example. Source Code 6 is an example of a first attempt at writing a function to insert a new item at the head of the list.

```
typedef struct node_struct {
    int data;
    struct node_struct *next;
} node_t;
```

Source Code 5
Linked-list data structure.

```
 1: void ListInsertBad(node_t *head, int newData) {
 2:     node_t *newHead = malloc(sizeof(node_t));
 3:     newHead->data = newData;
 4:     newHead->next = head;
 5:     head = newHead;
 6: }
 7: int main(void) {
 8:     node_t *myList = NULL;
        /* Assume we have some code here that puts some items in the list,
         * so that it now contains the entries {1, 2, 3}. */
20:    ListInsertBad(myList, 5);
21:    printf("%d\n", myList->data);
22:    return 0;
23: }
```

Source Code 6
Code to insert an item in a linked-list (incorrect).

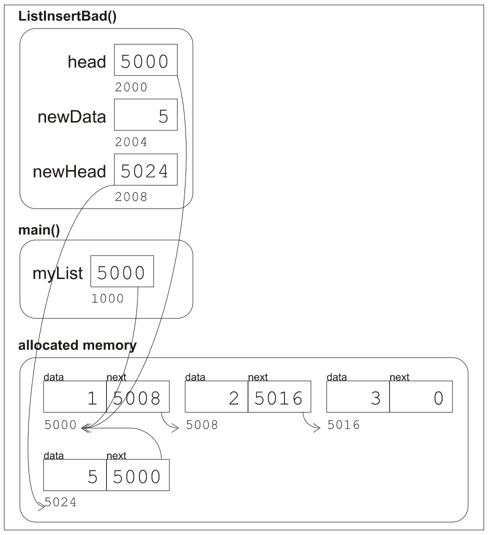

Diagram 6a
After line 20:4.

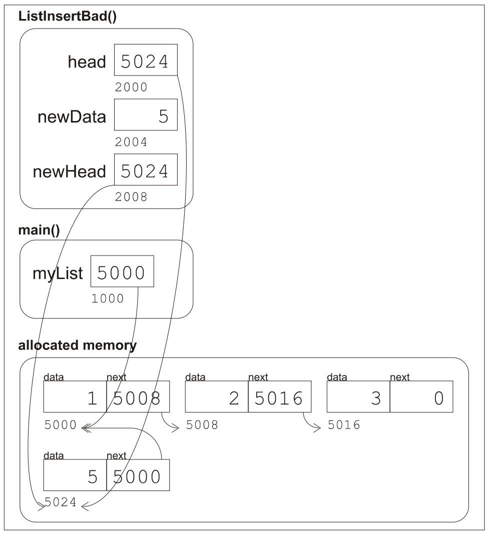

Diagram 6b
After line 20:5.

Figure 6a shows the state of the program after executing line 4 in ListInsertBad(). At this point, we have allocated a new node, and connected the rest of the list to it. Then, in Figure 6b, we have just executed line 5. Now `newHead` points to the new head of the list. So far, this is what we want. We have inserted a new item at the beginning of the list, and we have our variable `newHead` pointing to it. But what happens when we return from ListInsertBad()? This is shown in Figure 6c.

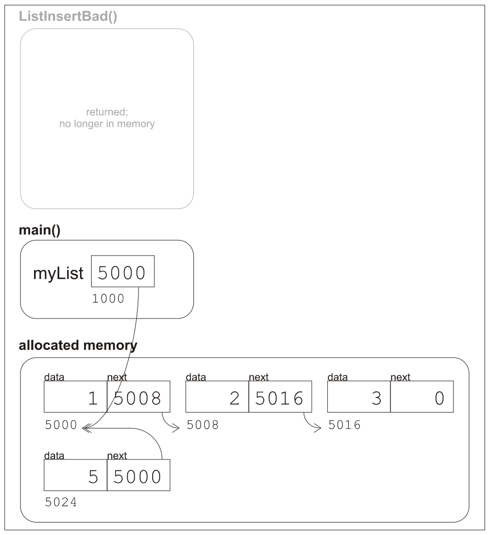

Diagram 6c
At end.

In main(), `myList` still points to the original head of the list – not the new one. The new item we added is still taking up memory, but we don't have any pointers that point to it, so we have no way of accessing it. When you run this program, the number 1 will be printed, not the number 5. ListInsertBad() fails to work for *exactly* the same reason that changeBad() fails. Remember that parameters are passed in *by value*. On line 5, when we set `head` to equal `newHead`, we are only changing the value of the `head` in ListInsertBad(). This is totally separate from the value of `myList` in main(). Look again at Source Code 3. `myList` in main() and `head` in ListInsertBad() (from Source Code 6) are analogous to `num` in main() and `value` in changeBad() (from Source Code 3), respectively.

Make sure that you understand why ListInsertBad() does not work and how it is similar to changeBad() before reading any further.

The problem we encountered in Source Code 3 was that, since parameters are passed by value, `num` was *copied* when we passed it into changeBad(), so we could not change `num` in main() by changing `value` in changeBad(). We fixed this problem in Source Code 4 by changing the parameter to an `int*` instead of just an `int`. Let's see if the same approach can be used to fix ListInsertBad(): instead of passing in a `node_t*` parameter, we will pass in a *pointer to* a `node_t*` parameter – a `node_t**`. The result is shown here:

```
 1: void ListInsert(node_t **pHead, int newData) {
 2:     node_t *newHead = malloc(sizeof(node_t));
 3:     newHead->data = newData;
 4:     newHead->next = *pHead;
 5:     *pHead = newHead;
 6: }
 7: int main(void) {
 8:     node_t *myList = NULL;
        /*  Assume we have some code here that puts some items in the list,
         *  so that it now contains the entries {1, 2, 3}. */
20:    ListInsert(&myList, 5);
21:    printf(“%d\n”, myList->data);
22:    return 0;
23: }
```

Source Code 7
Code to insert an item in a linked-list (correct).

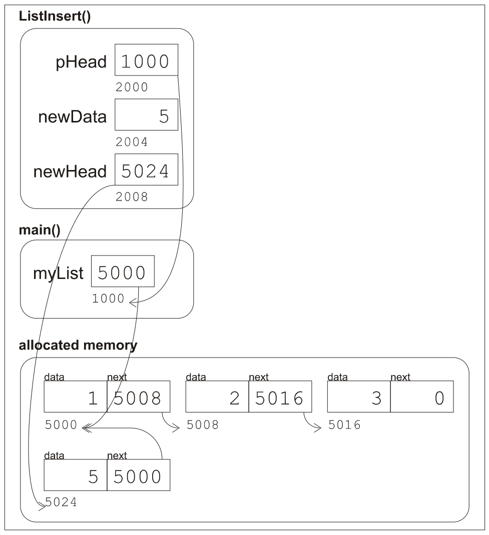

Diagram 7a
After line 20:4.

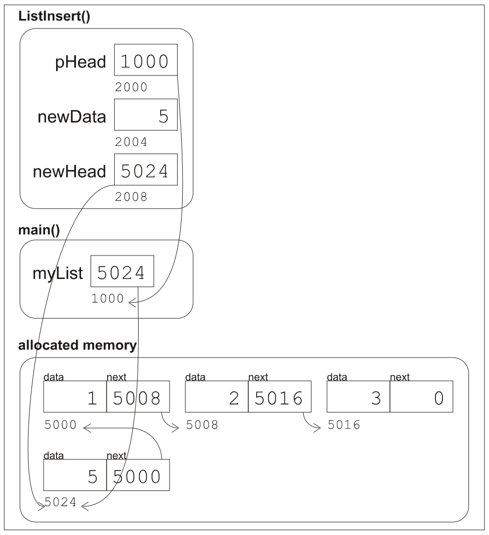

Diagram 7b
After line 20:5.

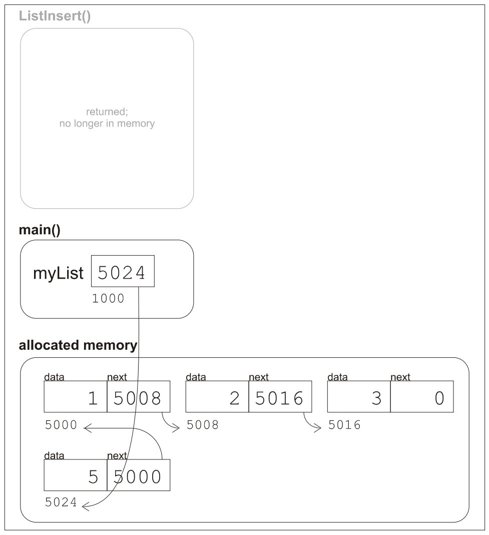

Diagram 7c
At end.

Figure 7a again shows the state of the memory after line 4. Once again, we have allocated a new node and linked it to the former head of the list. This time, the parameter to ListInsert() is a `node_t**` – a *pointer to* a `node_t*`. `pHead` does not point to the head of the list; it points to the variable `myList` in main(). Now on line 5 we *dereference* `pHead` and assign the value of `newHead` to the result. Dereferencing `pHead` gives us the box at memory location 1000. When we assign the 5024 (the value in `newHead`) to this box, we are actually changing the value of the variable `myList` in main(). You can see the result in Figure 7b. When ListInsert() returns, `myList` will have been updated to point to the new head of the list, which is the behaviour we originally intended.

Freedom at Last

There's one last thing which needs to be mentioned. We have been talking about pointers and memory addresses, and have shown some examples using `malloc()`. The memory that is allocated by `malloc()` stays allocated until you free it. You do this by calling `free()` on the pointer that `malloc()` had returned. This releases the memory allowing it to be re-claimed by the system. It is important always to free memory that you have allocated once you are done with it. If you don't, then your program will allocate more and more memory the longer it runs. This is called a "memory leak" – the free memory in your machine will slowly leak away as your program runs. Eventually you will use up all the memory in your machine. When this happens, your program will be terminated by the operating system with an unpleasant error message. Always check your programs to make sure that you have matched each call to `malloc()` with a call to `free()` somewhere else.