

30. Double pointers in dynamic data structures

Herman Kamper

There is one piece of code in the linked-list material that many students find intimidating, and honestly it is probably the single most confusing thing in the whole course. It is the function that adds a node to the front of a linked list, and in particular its first parameter, which has *two* stars in it. In this lecture we take that parameter apart slowly. The idea behind it is not complicated, but it is easy to get lost in the notation, so we build up to it in small steps: first a reminder of how a function can change a variable that belongs to someone else, then the same trick applied to a pointer, and finally the linked list.

If it does not click the first time, that is normal. This is one of those topics that you only really understand once you have drawn the pictures for yourself, and you will probably need to do that more than once.

The code that causes the confusion

Here is the node definition from the lecture on linked lists, together with a function that adds a new node to the front of the list:

```
struct listNode
{
    int data;
    struct listNode *nextPtr;
};
typedef struct listNode ListNode;
typedef ListNode * ListNodePtr;

int main()
{
    ...
    addNodeAtFront(&startPtr, 5);
    ...
}

void addNodeAtFront(ListNodePtr * startPtrPtr, int data)
{
    newPtr = malloc(sizeof(ListNode));
    ...
    newPtr->data = data;
    newPtr->nextPtr = *startPtrPtr;
    *startPtrPtr = newPtr;
    ...
}
```

(The dots hide the parts that are not important for this lecture, such as declaring `newPtr` and checking that `malloc` worked.)

The first part is something we have seen before. A `listNode` holds some data and a pointer to the next node. Because we are lazy and do not want to type `struct listNode` every time, we create a shorthand with `typedef`, and that shorthand is `ListNode`. Then comes the second `typedef`:

```
typedef ListNode * ListNodePtr;
```

This says that `ListNodePtr` is a shorthand for `ListNode *`. Anywhere that you see `ListNodePtr`, it really means “pointer to a `ListNode`”. This is a convenience: in `main` we can write

```
ListNodePtr startPtr = NULL;
```

instead of `ListNode *startPtr = NULL;`. The two are exactly the same thing. Programmers like this style for dynamic data structures, and you will see it in many textbooks.

So far so good. The part that causes the trouble is the function `addNodeAtFront`, whose first

parameter is

```
ListNodePtr * startPtrPtr
```

Since `ListNodePtr` means `ListNode *`, we could just as well have written the parameter as

```
ListNode ** startPtrPtr
```

which is the same declaration with the shorthand expanded. Now the double star is out in the open. How should we read it? Read the declaration from right to left: `startPtrPtr` is a box, and the stars tell us what goes inside that box. One star means that the box holds an address. Two stars mean that the box holds an address of another address, and the thing at the end of that second address is a `ListNode`. In short, `startPtrPtr` is a pointer to a pointer to a `ListNode`.

That is what the declaration says. The more interesting question is why on earth we would want such a thing.

Why would a function need the address of a pointer?

To see why, consider what `addNodeAtFront` has to do. In `main` we have the start pointer `startPtr`, which is our one way into the linked list: it points at the first node. When we add a node at the front, the first node of the list changes. The new node becomes the first node, so afterwards `startPtr` must point at the *new* node and no longer at the old first node.

So the function must change the value stored in `startPtr`, which is a variable in `main`. And that is exactly the situation that we have run into before in the course: we want a function to change a variable that belongs to the function that called it. We solved that problem with call by reference. To see how the solution carries over, let us recap call by reference first, with no linked lists involved, and then do the same thing for a pointer.

Recap: changing a value

Here is a small program that does nothing useful, but it is easy to follow. In `main` there is an integer `b` holding 5, and we want a function that changes it to 7.

Suppose that we write the function like this:

```
void changeValueOfB(int b, int value)
{
    b = value;
}
```

and call it as `changeValueOfB(b, 7)`. Will this work? No. This is call by value: the 5 in `b` is *copied* into a new box that belongs to the function. That box is also called `b`, but it is a completely different box from the one in `main`. A second new box, `value`, is created and holds 7. The line `b = value;` then copies the 7 into the function's own `b`. When the function ends, its boxes disappear and the `b` in `main` still holds 5. Nothing happened to our original variable.

The fix is to pass in the *address* of `b` instead of its value. The parameter changes from `int b` to `int *b`, meaning that it is a box that holds an address, and inside the function we write `*b`

where we previously wrote b:

```
void changeValueOfB(int *b, int value)
{
    *b = value;
}
```

In main we call it with `changeValueOfB(&b, 7);`. The figure below shows what happens. The top half is main, the bottom half is the function, and the dashed line separates their memory.

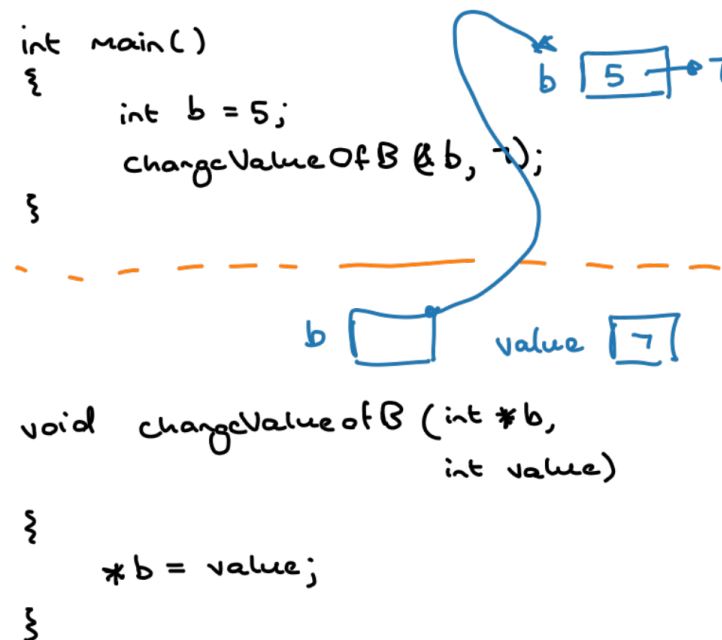


Figure 1: Call by reference. The function's b is a box that holds the address of the b in main (the arrow). The line `*b = value;` goes to the box that the arrow points at and changes the 5 into a 7.

When the function is called, a new box b is created for it, and the address of b in main goes into that box. Eight bytes are used for this, since that is the size of an address on our machines. We can draw this as an arrow from the function's b to the b in main, although you should remember that the arrows do not really exist; there is only an address stored in a box. The 7 is copied into the function's value box as before. Then the line `*b = value;` is executed. The star here means "go to the box that this address belongs to", and the line says to take whatever is in value and put it into that box. The 5 in main becomes a 7. We have updated the original variable.

The key point is that we changed the thing that b points at, and we could do so because the function was handed the address of that thing.

Changing a pointer

Now for the step that gives us the double pointer. What if the variable that we want to change is not an integer but a *pointer*?

Here is a second small program. In main we create an integer a holding 7 and an integer c holding 8. We also create a pointer bPtr and make it point at a:

```
int main()
{
    int a = 7;
    int c = 8;
    int *bPtr;
    bPtr = &a;
    ...
}
```

Right now bPtr holds the address of a. What we would like is a function, called changeWhatBPointsAt, so that after the call bPtr holds the address of c instead. The arrow from bPtr should move from a to c. The thing that we want to change is the value stored in bPtr, and that value is an address.

You might be tempted to write the function like this:

```
void changeWhatBPointsAt(int *bPtr, int *newAddress)
{
    bPtr = newAddress;
}
```

and call it with changeWhatBPointsAt(bPtr, &c). This does not work, and the reason is exactly the same as for changeValueOfB with a plain int. The pointer is passed by value: the address stored in bPtr is copied into a new box that belongs to the function. The function then changes its own copy, and the bPtr in main still points at a. (Notice that the function and the variable even happen to use the same name. Those are two different boxes.)

We have a variable that we want a function to change, so we need to pass the address of that variable. The variable is bPtr, so we pass &bPtr, the address of the pointer. And what is the type of the address of an int *? It is int **, a pointer to a pointer to an int. That is where the double star comes from. The correct version is:

```
void changeWhatBPointsAt(int **bPtr, int *newAddress)
{
    *bPtr = newAddress;
}
```

and the call is changeWhatBPointsAt(&bPtr, &c);. This is the same pattern as in the previous section; the only difference is that the thing we are changing is itself an address. The figure below shows the situation after the function has been called but before it has returned.

Let us step through it with some made-up addresses. In main, bPtr is first set to the address of

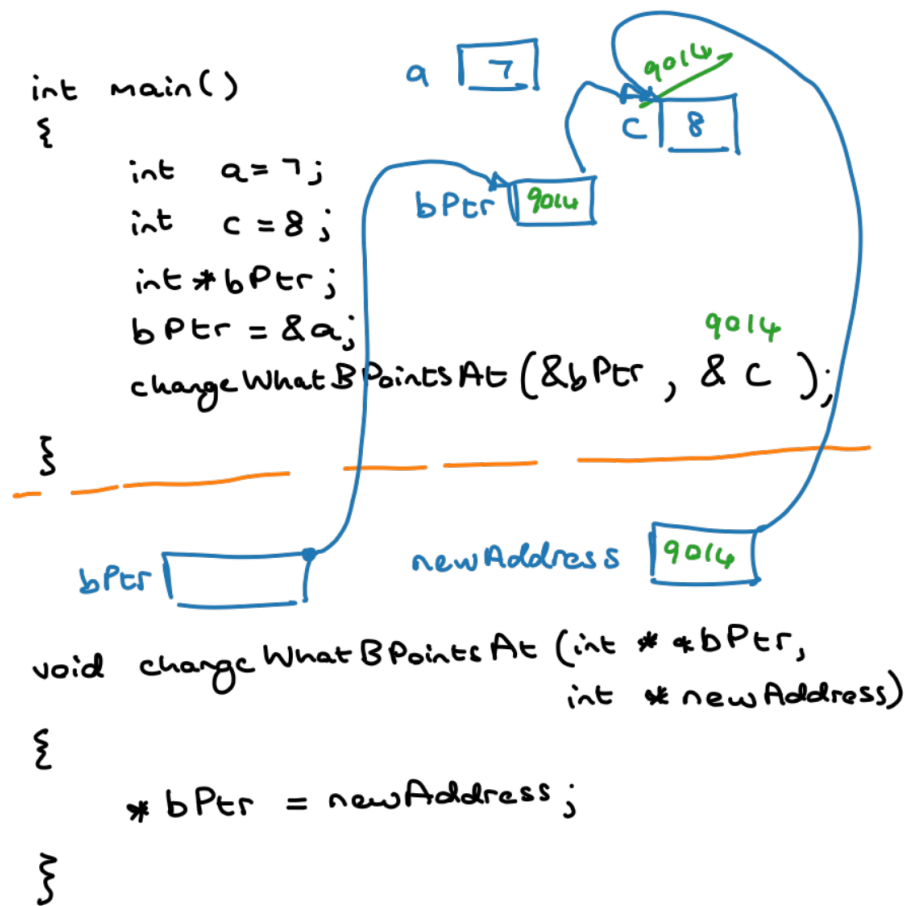


Figure 2: Changing a pointer by reference. In main, bPtr is first set to point at a. We pass &bPtr and &c to the function. The function's bPtr points at the box in main, and its newAddress holds the address of c (made-up address 9014). The line *bPtr = newAddress; writes 9014 into the box in main, so bPtr now points at c.

a. We pass in two things. The first is `&bPtr`, the address of the box `bPtr` in `main`. That address is copied into the function's own box, which is also called `bPtr`, although it is a different box and it has the type `int **`. So the function's `bPtr` now points at the `bPtr` in `main`. The second is `&c`, the address of `c`. Say that `c` sits at address 9014. Then 9014 is copied into the function's box `newAddress`.

Now the line `*bPtr = newAddress;` runs. The star on the left means “go to the box”. Which box? The function's `bPtr` holds the address of the `bPtr` in `main`, so we go to the `bPtr` in `main`. Then the line says to take whatever is on the right-hand side (that is 9014) and put it into that box. So the `bPtr` in `main` now holds 9014, which is the address of `c`. In picture form, the arrow has moved: `bPtr` in `main` points at `c` and no longer at `a`.

Notice what did *not* change. The variable `bPtr` in `main` stayed exactly where it was in memory. We did not change the address of `bPtr` itself, we changed the *contents* of `bPtr`, which is the address of something else.

I can verify this with a few print statements. Inside the function, just before the assignment, `**bPtr` has the value 7, and just after it has the value 8. Start from the function's `bPtr`: one star takes us to the `bPtr` in `main`; the second star follows the address in that pointer. Before the assignment that leads to `a`, and after it, to `c`. After the function has returned, printing `*bPtr` in `main` gives 8.

Stars in declarations and stars in expressions

A very common source of confusion here is that the star means two different things, depending on where it appears.

- In a *declaration* such as `int *newAddress` or `int **bPtr`, the stars say what kind of box this is. `int *newAddress` creates a new box that is going to hold an address of an `int`, and `int **bPtr` creates a box that is going to hold the address of an address of an `int`.
- In an *expression*, such as `*bPtr = newAddress;` or `printf("%d", **bPtr);`, a star means “go to the box at this address”. It is a command, and it is almost the opposite of declaring something.

This is not a problem when you work with plain variables, but it becomes confusing with pointers, because declaring a pointer and following (dereferencing) a pointer look the same while meaning different things. The line `int *newAddress` in the parameter list is a declaration, while the line `*bPtr = newAddress;` in the body is not.

A helpful way to read these is from right to left. For `**bPtr`: `bPtr` is a box; the star next to it means “go to the address in that box”; the next star means “go to the address in *that* box”. Each star is one step of following an address.

Back to the linked list

Now we have everything needed to understand `addNodeAtFront`. Compare the two functions:

```
void changeWhatBPointsAt(int **bPtr, int *newAddress)
{
    *bPtr = newAddress;
}

void addNodeAtFront(ListNodePtr *startPtrPtr, int data)
{
    ...
    newPtr->nextPtr = *startPtrPtr;
    *startPtrPtr = newPtr;
    ...
}
```

They have the same structure. In both cases the first parameter is the address of a pointer variable that belongs to main. In the list function that variable is `startPtr`, and in main we pass its address with `&startPtr`. The parameter `startPtrPtr` has type `ListNodePtr *`, which is `ListNode **`, a pointer to a pointer to a `ListNode`. It is just like `int **bPtr` above, with a `ListNode` in place of the `int`.

Inside the function the two important lines are:

```
newPtr->nextPtr = *startPtrPtr;
*startPtrPtr = newPtr;
```

Since `startPtrPtr` holds the address of `startPtr` in main, `*startPtrPtr` is the start pointer itself. The first line reads it: the new node's `nextPtr` is set to whatever the start pointer holds, which is the old first node (or `NULL` if the list was empty). The second line writes to it: the start pointer in main is changed to point at the new node. The second line is identical in shape to `*bPtr = newAddress;`.

Let us run the function on a list that holds 8 and 12, and add a 5 at the front. After the call, the list is 5, 8, 12. I compiled and ran this version to check it:

```

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

struct listNode
{
    int data;
    struct listNode *nextPtr;
};
typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

void addNodeAtFront(ListNodePtr *startPtrPtr, int data)
{
    ListNodePtr newPtr = malloc(sizeof(ListNode));

    if (newPtr != NULL) {
        newPtr->data = data;
        newPtr->nextPtr = *startPtrPtr;
        *startPtrPtr = newPtr;
    }
}

int main(void)
{
    ListNodePtr startPtr = NULL;

    addNodeAtFront(&startPtr, 12);
    addNodeAtFront(&startPtr, 8);
    addNodeAtFront(&startPtr, 5);

    for (ListNodePtr p = startPtr; p != NULL; p = p->nextPtr) {
        printf("%d ", p->data);
    }
    printf("\n");

    return 0;
}

```

The output is 5 8 12. Each call to `addNodeAtFront` makes `startPtr` point at the node that was just created.

Why use `malloc` here?

You might ask why we have to go through such a complicated route. The answer is `malloc`. All that `malloc` gives us is an *address*: the address of a fresh block of memory for a node. We need

that address to end up inside the start pointer in main, so that the list starts at the new node. And the only way for a function to put a new value into a variable of main is to be told where that variable is. That is the whole reason for the double pointer.

With a linked list we always add a single node at a time, so the list as a whole grows by one node per call. But each time, the place that holds the start of the list has to be updated, and that place lives in main.

Avoiding the double pointer

There is another way, which avoids double pointers: the function can *return* the new start of the list, and the caller can store it:

```
ListNodePtr addNodeAtFront(ListNodePtr startPtr, int data)
{
    ListNodePtr newPtr = malloc(sizeof(ListNode));

    if (newPtr == NULL) {
        return startPtr;
    }
    newPtr->data = data;
    newPtr->nextPtr = startPtr;
    return newPtr;
}
```

and in main:

```
startPtr = addNodeAtFront(startPtr, 5);
```

This works (I tested it too), and it is a perfectly good approach. But for operations that might change the start pointer *and* need to report something else, such as whether the operation succeeded, a return value is already taken, and then passing the address of the pointer is the typical solution. This is how it is usually done in C, and it is the pattern that you will see in textbooks. The same approach is used for the stack and queue operations from the previous lecture, where push and enqueue take a pointer to the top, head or tail pointer.

Is the double pointer needed everywhere?

No. It is only needed when a function might *change* the pointer to the start of the list. A function that only reads the list, for instance one that prints it or searches for a value, can take a plain `ListNodePtr`. A function that only ever changes the `nextPtr` of a node that it has already reached, such as one that inserts after a given node, does not need it either. The problems start whenever the first node can change: adding at the front, deleting the first node, or inserting into a sorted list where the new value might belong before everything else.

If you are still confused

Here are a few things that might help.

- Draw boxes. Make a box for every variable, in main and in the function, and write the contents of each box. When a function is called, draw its new boxes and fill in the values that were copied. Then step through the code line by line. A star in an expression means “follow the address in this box”; draw an arrow for that.
- Remember that parameters are always new boxes. When a function is called, its parameters are *new* variables, with their own memory, even if they have the same name as variables in the caller. Only the *values* are copied, and an address is a value like any other.
- Think of it as passing an address by reference. The function `addNodeAtFront` does not do anything magical: it takes the address of a box (the start pointer) and uses it to change what is inside that box, exactly like `changeValueOfB` does for an integer. The only difference is that the thing inside the box happens to be an address.
- If all else fails, memorise the pattern. When a function needs to change a pointer of its caller, the parameter type gets one more star, you pass the address with `&`, and inside the function you write `*` before the parameter name. This is not a nice thing to recommend, but the pattern is the same every time, and the understanding often follows later.

Even people who write C for a living find this confusing. That is partly why the typedef for the pointer type exists: with `ListNodePtr` as a type on its own, a pointer can be thought of as just another variable, and the extra star is a bit less intimidating.

Summary

C passes arguments by value, so a function can only change a variable of its caller if it is given the variable's *address*. For an integer, the parameter has type `int *`, and the function writes `*b = value;`. For a pointer, the same idea means passing the address of a pointer, so the parameter has type `int **`, and the function writes `*bPtr = newAddress;`. The extra star is just one more level of “address of”.

The function that adds a node to the front of a linked list needs this, because it has to change the start pointer in main to point at a newly malloced node. Its first parameter is `ListNodePtr *startPtrPtr`, which means the same as `ListNode **startPtrPtr`. Inside the function, `*startPtrPtr` is the caller's start pointer: first it is read, to link the new node to the old first node, and then it is written to, to make the new node the first node.

A star in a declaration says what kind of box something is; a star in an expression means “go to the box at this address”. A function that only reads a list does not need a double pointer, and a function that returns the new start pointer can also avoid it.

Exercises

1. Without running it, say what the following program prints, and draw the boxes to check.

```
void change(int *p, int *q)
{
    p = q;
}

int main(void)
{
    int x = 1;
    int y = 2;
    int *ptr = &x;

    change(ptr, &y);
    printf("%d\n", *ptr);
    return 0;
}
```

2. Change the function in the previous exercise so that the program prints 2, and change the call in main to match.
3. In the declaration `ListNodePtr *startPtrPtr`, what does `ListNodePtr` stand for? Rewrite the declaration without using `ListNodePtr`.
4. In main you have `ListNodePtr startPtr = NULL;`. What is the type of `&startPtr`?
5. What happens if `addNodeAtFront` is written with a plain `ListNodePtr startPtr` parameter, so that the last line is `startPtr = newPtr;` in the example in this lecture? What does the list look like in main after the call? (There is also a memory problem.)
6. Write a function `deleteFirstNode` that removes the first node of a list and frees it. Does it need a double pointer? Why?
7. Write a function `printList` that prints all the values of a list. Does it need a double pointer? Why not?
8. Rewrite `addNodeAtFront` so that it returns the new start pointer rather than using a double pointer. Show how it is called from main. Then explain why the version with the double pointer can be nicer if you also want the function to return an `int` that says whether `malloc` succeeded.