Pointers and Dynamic Memory

William E. Skeith III

In this lecture, we'll introduce a special type of variable called a *pointer* and explore a few fundamental applications, including dynamic memory allocation and linked lists.

Section 1

Introduction

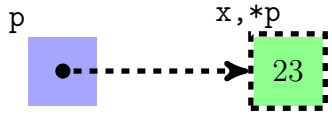
1.1 What are pointers?

Pointers are simply *variables that store memory addresses*. That's it. Lecture over! I'm of course kidding about ending the lecture, but it is true that there's not much more to say about pointers in general. There's nothing scary or intimidating about them. To convince you of this, and to keep things concrete and intuitive, let's find a way to continue our post-it note analogy for pointers before going further. You could think of them perhaps as ultra-precise GPS coordinates of some other post-it note on your desk, but perhaps the following is better: think of pointers as being "tethered" to another post it note by a piece of string. Something like this maybe (the pointer is the blue note on the left):

In a departure from previous lectures, let's introduce a few programming details early on. In particular, let's see how to declare a pointer and to establish a situation like the above picture. Here is the syntax in C/C++ for pointer basics.

```
int x = 23;  /* declare an integer */
int* p = &x; /* set p to point to x */
```

Details of the syntax and other formalities will be covered as we go. For the moment, just know that the two lines above would produce the following picture:



Note that we have labeled each post-it note with the corresponding C++ variable name(s), and as usual the value is written in the center of each note.[1] **Important:** notice that there are **two** names for the green post-it note! You could reference it as usual using the variable name `x`, or via the pointer with the expression `*p`, and there is essentially no difference.[2]

The impatient reader might now be wondering "what's the point?" The only thing that the above code seems to have accomplished was to produce a strange synonym (`*p`) for an existing variable (`x`), which admittedly does not seem very useful. So what's the big deal? Well, it turns out that in some cases a name of the form `*p` will be the *only* name for that variable. For example, there are some situations where you must allocate memory *as your program is running*, and in cases like these, you might not even know how many variables your program will require ahead of time.[3] Reflecting on this a moment, one realizes that *necessarily some variables must be anonymous* in the sense that you did not explicitly declare and name them in your program.[4] Such variables are often kept track of only by a pointer, which *may itself have been dynamically allocated*! We will see such examples when we discuss dynamic memory and linked lists in Sections 2 and 3, but to facilitate those discussions, it will be nice to understand a few more technical details of pointers in C/C++ first. If you need a bit more convincing that pointers are useful, perhaps just browse the later sections for a few minutes and then come back here.

1.2 Technical details

Let's take a closer look at the bit of code we saw to setup a pointer.

```
int x = 23;  /* declare an integer */
int* p = &x; /* set p to point to x */
```

The first thing to notice is the `*` after the `int`. In C/C++, for every datatype `T`, there is a corresponding pointer datatype called `T*`, which you should think of as "pointer to something of type `T`". Let's see what's in there:

```
cout << p << "\n"; /* prints something like 0x7ffaecf0442e */
```

It's just a long, unsigned integer (like `size_t`), which makes good sense for storing a memory address. (Note that pointers are usually printed in hexadecimal.) The only other potentially unfamiliar thing in the snippet is the `&x`. The ampersand is called the "address-of" operator,

[1] The actual value of a pointer in C++ will be an integer, but the particular value has no independent meaning. The meaning of the integer is the memory location it represents, and thus we visualize the *value* of a pointer by an arrow showing *where it points*.

[2] The low level assembly code for accessing `x` might differ a little from that of accessing `*p`. However, in a simple case like the above (where `&p` is never referenced and the pointer never refers to anything but the stack variable `x`), `p` itself may never be stored (this is the case for `gcc -O2`), and the references to `*p` vs `x` will be indistinguishable in the assembly.

[3] Perhaps consider the problem of how to print, in reverse order, an arbitrarily long list of integers given on standard input

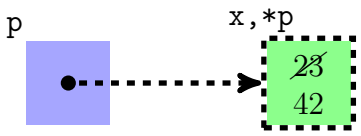
[4] After all, any program's source code has a well-defined, constant size when it is compiled!

and it gives the memory address of whatever it is applied to,[5] in this case `x`. It should be no surprise that if `x` was an `int`, then `&x` is of type `int*`, so the assignment makes sense (the types are the same on both sides of the `=`).

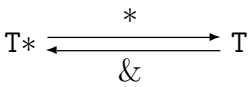
One last thing that is worth explicitly discussing is the `*`, or "dereference" operator, as used in the expression `*p`. We mentioned above (and illustrated in our pictures) that upon setting `p = &x`, the expressions `x` and `*p` became synonyms, in the sense that they were both labels for the same piece of memory. Let us check that this is really the case:

```
*p = 42;
cout << x << "\n"; /* prints 42 */
```

Indeed, anything that happens to one, seems to happen to both. But looking again at the picture, this should have been obvious!



You can perhaps think of dereference (`*`) and address-of (`&`) as inverses for one another. If you identify a datatype with the set of all variables of that type, then we have the following



such that $* \circ \& = \mathbf{1}_{\mathtt{T}}$ and $\& \circ * = \mathbf{1}_{\mathtt{T*}}$. Put another way:

```
cout << &(*p) << "\n"; /* prints value of p */
cout << *(&x) << "\n"; /* prints value of x */
```

Pointer pitfalls. There are a few common mistakes that you should be aware of. The first is that when trying to declare multiple pointers in a single statement, you must have a `*` *for each pointer*. For example, the following declares two `int*` pointers (`p` and `q`) and one plain old integer (`n`).

```
int* p, *q, n;
```

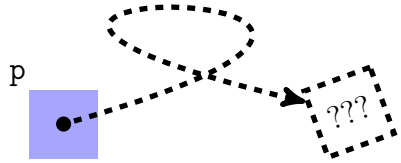
Note also that the space is not important – `int* p;` has the same effect as `int *p;` which is the same as `int * p;` as well as `int*p;`. Perhaps think of it like this: `p` is of type `int*` if and only if `*p` is of type `int`, so both `int* p;` and `int *p;` carry the very same meaning.

Another common mistake is trying to dereference a meaningless pointer. (This will typically result your program crashing with a *segmentation fault*.) Just as every integer must have *some* value, so must every pointer. If we just declare one like this,

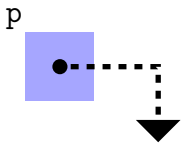
[5] Note that you can't apply address-of to everything, as many expressions do not "live" anywhere in main memory. E.g., trying to take `&(x*10)` will be rejected by the compiler. If this is confusing, consider the fact that the value of such expressions may exist only for a moment in a CPU register.

```
int* p;
```

it still points *somewhere*, but we have no idea where. The following whimsical illustration is one possible visualization of such a situation.



If we mistakenly follow that string, for all we know it might lead us into the middle of the ocean! (Segmentation fault...) To avoid such mishaps, there is a designated value which signifies a meaningless pointer which should **never** be followed: the symbol `NULL` is defined to be this value.[6] If you don't immediately know where your pointer should go, it is good practice to set it to `NULL`. Similarly, if a pointer ever becomes meaningless, say by deallocating the memory it pointed to (more on this in Section 2), you should again set the pointer to `NULL` (if it isn't immediately going out of scope). In pictures, it is common to illustrate such null pointers as arrows pointing downward like so:

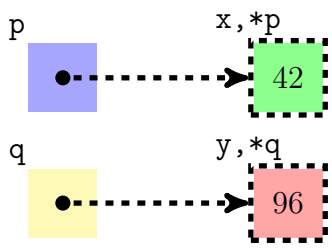


It should also be mentioned that simply setting meaningless pointers to `NULL` is certainly no guarantee that you won't accidentally dereference a meaningless pointer. It will, however, at least give you a fighting chance of not doing so, and if it does happen, it will make debugging much easier.

Lastly, some people get confused by what it means to assign one pointer to another. If you think carefully, you should be able to guess, but let's draw a picture anyway just in case. Let's say we have a situation like this:

```
int x = 42, y = 96;
int* p = &x;
int* q = &y;
```

As we've seen, you can visualize the situation as follows:

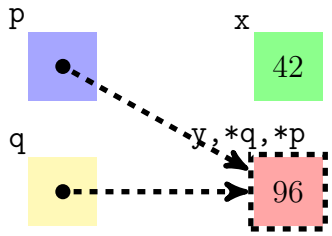


[6] In C, the symbol `NULL` has simply been `#define`'d to be 0, cast to `void*`. See `stddef.h`.

Then we assign one pointer to the other, like this:

```
p = q;
```

We'll now have the following picture, with both pointers anchored to `y`:



There really isn't much more to know about pointers in general besides this. Let's move on and see some applications.

1.3 Exercises

1. If `p` is of type `int*`, what is the datatype of `&p`?
2. Understand why this does not compile:

```
int x = 99;
cout << &(&x) << "\n";
```

3. Why doesn't the following compile, and why should you be thankful that it doesn't? (After all, pointers are all the same thing: they are just unsigned integers.)

```
int x = 99;
char* p = &x;
```

Section 2

Dynamic Memory

Often times you will need to allocate memory as your program is running (think of doing `push_back` in a vector). In this section we'll cover the basics of how to do this in C/C++.

2.1 Quick historical note

The following might help motivate the discussion (and is probably good to know anyway), but it is fairly safe to skip ahead to 2.2 if you aren't in the mood for history. Older C/C++ standards (back in the 90's) would prevent you from doing the following:[7]

[7] If you want to see what it was like back then, give `-std=c90 -pedantic-errors` to GCC.

```c
void array_test(int n)
{
    int A[n]; /* array length not known at compile time... */
    /* now do stuff with A... */
}
```

To understand why, you must have some knowledge about the *call stack*. This will be covered in class, but here's the takeaway:

1. Local variables like `A` are stored on the call stack.
2. The compiler maps your variable names to memory addresses via **fixed offsets** from a stack frame pointer. These offsets must be determined **at compile time**.

Thinking carefully about the above, you can see why something like the following would break that particular scheme of mapping variable names to memory:

```c
int A[n]; /* size of A not known at compile time... */
int x;    /* can't know in advance the offset for x! */
```

The idea is that we couldn't possibly know **at compile time** what the offset should be for `x`, as it would depend on the size of `A`, which is placed before `x` on the stack.[8] Most modern compilers will allow you to declare variable length arrays, but let's see how that is being done behind the scenes.

2.2 Dynamic allocations in C

Let's start by showing how it is done in the C language. The two key functions are named `malloc` and `free`. The `malloc` function is how you request memory for your program. The idea is pretty straightforward: ask and you shall receive! Just tell `malloc` how big of a block you want, and magically[9] you've got it. Naturally you will want to know where this memory is located. Sounds like pointers! Indeed, the return type of `malloc` is a pointer. Here's an example to dynamically allocate an array of 10 integers:

```c
int* A = malloc(sizeof(int) * 10);
```

A few things to notice:

1. You have to specify the size in units of *bytes* when using `malloc`, which is why we had to multiply by `sizeof(int)` to actually get 10 integers worth of memory (note that `sizeof` gives the number of bytes required to store a datatype).

[8] Perhaps if you rearrange the ordering, but it is easy to construct an example that breaks that as well, say by using two arrays.

[9] The details are system-specific, (involving both your `libc` implementation and OS kernel) so we won't go into it here. Also, note that it could potentially fail if the system runs out of memory.

2. It might be puzzling that this same `malloc` function could have been used to allocate a block of `double`s or `char`s since in general, you can't just assign different pointer types to one another. So what really is the return type of `malloc`? The trick is that `malloc` returns a kind of "universal pointer" type, which is `void*`. But don't worry much about that for now (also, it doesn't quite work in C++, as the assignment would require a typecast).

The downside of using dynamic memory is that you also inherit the responsibility of cleaning it up once you are done with it. Fortunately when your program ends, the operating system will free whatever memory it was using. But suppose you have a long-running program (like maybe a web server) that is making lots of allocations over time. In this case, if you don't continually free memory once you're done with it, *no one else will*, and at some point all of the system's memory will be consumed! So here's how to let the system know you're done with a block of dynamically allocated memory:

```
free(A);
```

That's really all there is to it.[10] Now let's see the C++ counterparts.

2.3 Dynamic allocations in C++

In C++, the approximate analog of `malloc` is the `new` operator, and the approximate analog of `free` is the `delete` operator. Here is a simple example of their usage:

```
int* A = new int[10]; /* dynamically allocate 10 integers */
/* do stuff with A... */
delete[] A; /* free the memory for A */
```

To understand some of the important differences between `malloc` and `new`, we'll have to wait until we have learned more about *classes*. But already you have probably noticed a few things:

1. The `new` operator accepts a datatype, and as such you don't have to calculate the size of the allocation in bytes. Instead, you can just tell `new` how many *variables* you want.
2. Note also that the return type for new is not `void*`, but a pointer to whatever type you are allocating.

There is also a subtle point regarding the brackets (`[]`) after the `delete` operator. You'll appreciate this point more once we have covered *destructors* for classes, but remember this much: if you are deleting an *array*, use `delete[]`, and if you are deleting only one of something, use plain `delete`, like this:

[10] Although in practice it can be tricky to make sure you never forget.

```
int* p = new int; /* dynamically allocate a single integer */
/* do stuff with p... */
delete p; /* Note: no brackets in this case */
```

It is worth mentioning some criticisms that have been made of the C++ method for handling memory allocation versus `malloc` in C. In short, having it built into the language (as an operator, rather than a library function) imposes some frustrating limitations. If you are interested to learn more, perhaps check out this nice article:
`http://www.scs.stanford.edu/~dm/home/papers/c++-new.html`

There isn't much else to say about the basics. The next section will attempt to tackle an interesting problem using what you have learned.

2.4 Exercises

1. Write a C++ function to "grow" an array, while preserving its contents.[11] Use the following prototype:

   ```
   void grow(int*& A, int oldsize, int newsize);
   ```

2. Use your function from the previous exercise to read all of standard input into an array (say of integers). When you run out of space, how do you decide on the size for the new array? Think about the performance cost of incrementing it versus doubling it.

Section 3

Linked Lists

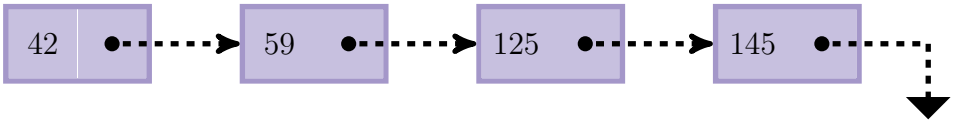
As you have perhaps seen in class, removing from or inserting into the middle of a vector or an array is not efficient. The functions `push_back` and `pop_back` for a vector are efficient, but `erase` for example is not. The issue is that the elements of a vector / array must be right next to each other in main memory. (Remember – the syntax `A[i]` is actually shorthand for `*(A+i)`.) So in order to remove or insert something anywhere but the end, you will have to shift all elements after that location. In this section, we will introduce a data structure called a *linked list* that supports efficient insertion or deletion at any location, meanwhile keeping elements in the order you prescribe.[12]

3.1 List basics

The basic idea behind linked lists is a simple one: each item in the list *explicitly tells you where the next one is*. The picture would be something like this:

[11] There are C library functions called `realloc` and `reallocarray` that do this, but it is good practice to do it yourself.

[12] Contrast with `set` or `map` if you've seen them. They support efficient insertion and removal, but they always keep the elements sorted (so you do not get to choose the order).



That's a nice picture, but you might be wondering how one would realize this in program code. It's actually quite easy! The only ingredients you will need are `struct`'s (or classes) and pointers. We'll simply make a new datatype using a `struct` that has two pieces: one to hold the number (let's say we'll only store integers for now), and the other to store the *location* of the next element of the list. Here's an example in C++.

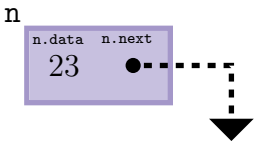
```
struct node {
    int data;
    node* next;
};
```

It might be a little curious that the `node` type contains a reference to itself. But don't worry—there is no infinite nesting since each node only contains a *pointer* to the next node, and does not contain another node in its entirety.

It is essential to be able to translate from pictures to code and back. Each part of the picture has a name relative to some program variable. Let's make sure we know what those are. Suppose we declared a node as follows:

```
node n;
n.data = 23;
n.next = NULL;
```

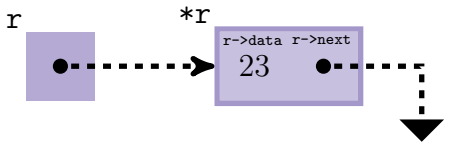
This would result in the following picture:



Note that `n` refers to the *entire node*, which has two sub-variables `data` (an integer) and `next` (a pointer to another node). In the above example, we *statically* allocated the node (so its contents are on the call stack). As we'll see, statically allocating nodes is fairly uncommon. Nodes will almost always be dynamically allocated, like this:

```
node* r = new node;
r->data = 23;   /* r->data is shorthand for (*r).data */
r->next = NULL; /* r->next is shorthand for (*r).next */
```

And now the corresponding picture:



Important: Take note of the names of each box, and make sure you know each one's datatype as well. That's all you need to know to start doing something interesting! Up next we'll figure out some basic list operations.

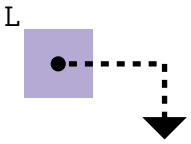
3.2 Operations on lists

One organizational issue comes up immediately: what information do we need to *explicitly* keep track of for a list? There are a few viable options, but we'll draw inspiration from arrays and simply store a *pointer to the first element*. (Recall that a dynamically allocated array `A` is just a pointer to `A[0]`.) Let's call the one pointer we explicitly store `L`. The invariant (what our variables are supposed to mean) is straightforward, but let's sketch it out anyway. The idea is that we want to store an arbitrary sequence of integers, let's call it $\{l_i\}_{i=0}^{k-1}$. We also consider the possibility of the empty sequence, where $k = 0$. Then we can state the invariant (a bit informally) as follows:

1. If `L` is nonzero, it points to a node containing l_0 as its value. The list is empty if and only if `L` is `NULL`. (Recall our convention regarding meaningless pointers.)
2. For a node to be considered part of the list, it must either be pointed to by `L`, or by `n.next` for some node `n` already in the list. This gives rise to a unique integer associated with each node that indicates its location in the sequence. Informally, it is the number of `next`s one must follow to reach the node. We of course require that the i-th node has l_i stored in its `data` field.
3. For a node `n` in the list, `n.next` is `NULL` if and only if `n` is the last node.

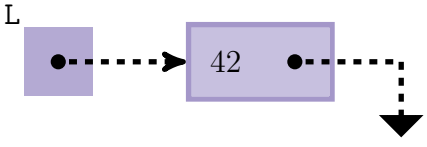
So we can establish the invariant and store an empty list by setting `L` to be a null pointer:

```
node* L = NULL;
```

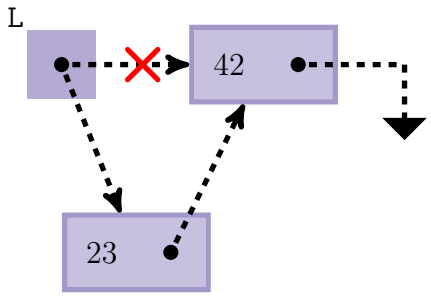


Take a minute right now to see if you can figure out how to add a node, say containing the number 42 (we've actually seen it already, but it's still good practice). When you're ready, see how your answer compares with the following, and see if you missed anything.

```
L = new node;
L->data = 42;
L->next = NULL;
```



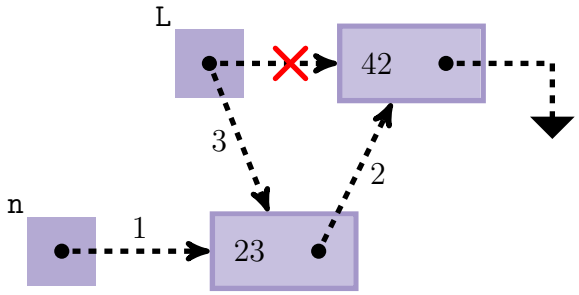
Again, notice that it was critical to set the `next` field to `NULL` to denote the end of the list. Now what if we wanted to put another node with a 23 before the 42? Let's think about doing this in pictures, and then the translation to code should be easy. Here's basically what we want to accomplish:



But there are a few things happening all at once, and we can usually only do one thing at a time. So here's a list of what needs to happen, in a sensible order:

1. Allocate a new node and place 23 in its `data`.
2. Set the `next` field of the new node to the address of the node with the 42.
3. Redirect `L` to this new node.

Note that we **cannot** swap steps 2 and 3. Remember, the only name we have for the node with the 42 is `*L`. So if we redirect `L` without saving the old value, we will have lost the node with the 42. Here's a more complete picture with each step labeled by its order. Note that since we are dynamically allocating the new node, we can only refer to it by an address, which is the reason for the pointer `n`.



Now that we have such a clear picture, the rest is cake! All we have to do is just remember the names of the little boxes so we can make the right assignments. But we've seen this already – each box has a name derived from `L` or `n`, the two pointers we've explicitly stored. Here's one way we can make all the assignments we need:

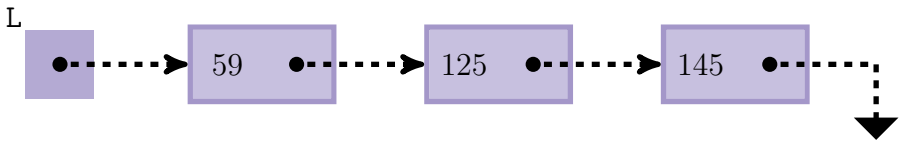
```
node* n = new node; /* step 1 */
n->data = 23;       /* step 1 (continued) */
n->next = L;        /* step 2 */
L = n;              /* step 3 */
```

Note that at this point, there is no need to keep track of the pointer `n`, since the same value is stored in `L`. But most likely, this bit of list manipulation would be in a function like this, so `n` would only be around for a moment during the call:

```
/* insert a node with value x into the list, placed immediately
 * before the node pointed to by "beforethis", lastly updating
 * "beforethis" to point to the new node. */
void insert(node*& beforethis, int x)
{
    node* n = new node;
    n->data = x;
    n->next = beforethis;
    beforethis = n;
}
```

If we were to call `insert(L,23)`, it would have accomplished the same as our bit of code above for placing 23 before 42 in the list. Question: why is the pointer passed by reference? It is actually critical for this function to work as intended, so make sure you understand why. Also notice that this function won't work on just *any* pointer to the node you want to place something before – it will only work when `beforethis` is the `next` of a node in your list, or the first pointer `L`. Again, make sure you understand why. Lastly, make sure you see why `insert(L,23)` would even work if the list `L` was empty.

Here's something that seems like it should be easy: let's just print out the entire contents of a list. First, let's visualize. Here's a generic looking list:



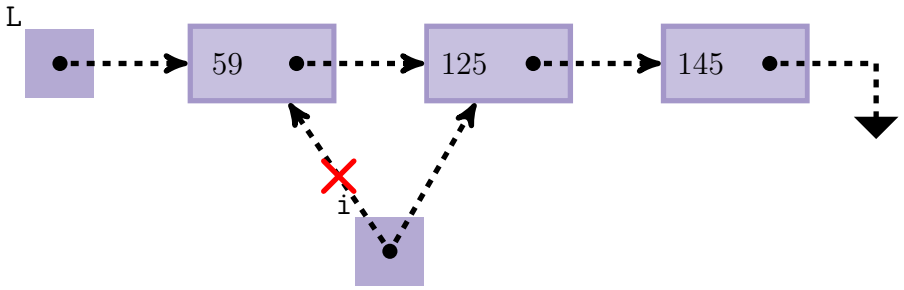
There's really not much choice in how to do this. The sole piece of information we explicitly keep track of is `L`, a pointer to the first element, so we had better start there. Now if we knew the list had only 3 elements, we could maybe just print `L->data` and then `L->next->data`, and then `L->next->next->data`. But we have no idea how long the list will be, so this approach will not work (and it was gross anyway). Let's again draw inspiration from arrays. When you want to examine all elements of an array `A`, you typically use an integer `i`, ranging from 0 to $k-1$, where k is the size of the array. But each integer `i` can be thought of as describing a pointer – in particular, the pointer `A+i`. Here's how you could print an array using pointers directly (`i` will be of type `int*`, and we won't use the bracket operator):

```
void printarray(int* A, int k)
{
    for (int* i = A; i != A+k; i++)
        cout << *i << "\n";
}
```

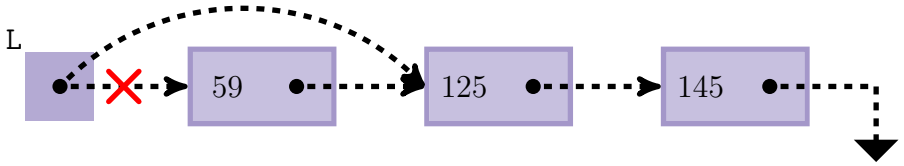
We can of course do very much the same thing with our lists. We'll start with a pointer to the first node, and just move it along the list, printing as we go. We have analogs for all of the ingredients used above: `L` gives a pointer to the first element (≈ `A`), the null pointer tells us when we're out of elements (≈ `A+k`), and if you think about it, the equivalent of `i++` would be `i = i->next` (this just moves the pointer `i` to the next element!). Compare this with the above:

```
for (node* i = L; i != NULL; i = i->next)
    cout << (*i).data << "\n";
```

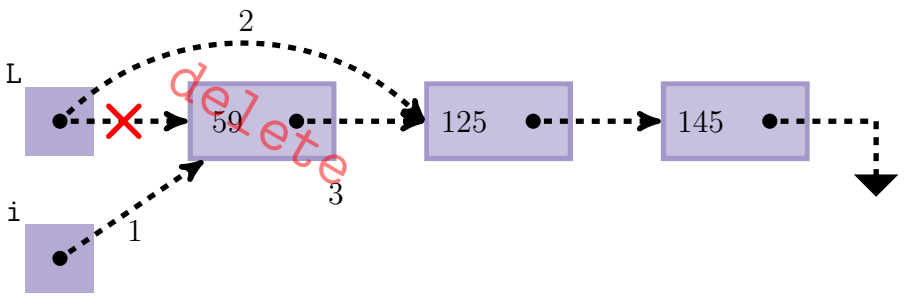
Note that we used `(*i).data` instead of `i->data` just to emphasize the similarities. And in case it helps, here's a picture of "incrementing" the pointer via `i = i->next`.



Now let's figure out how to remove elements. We'll start by removing the very first element. Here's a first attempt in pictures:



This would indeed effectively remove the first node, and could be accomplished in code with little more than an assignment statement. But there is a problem. Can you see it? The issue is that the node with the 59 will still be floating around in memory, and now there's nothing we can do about it (the only name we had for that node was `*L`). One obvious fix is to just save the location of that node first so we can delete it later. Here's an updated picture of our plan, including a sequence for the steps:



And here's the code that would accomplish the above:

```
/* remove node pointed to by L */
void remove(node*& L)
{
    if (L == NULL) return; /* empty list; nothing to do */
    node* i = L; /* step 1 */
    L = L->next; /* step 2 */
    delete i;    /* step 3 */
}
```

Once again, note the reference parameter, and note that this function should be called on "internal pointers" of the list (that is, pointers which are either the `next` of something in your list, or the pointer you are using to store the location of the first node).

3.3 Exercises

1. Write C++ functions for all of the following tasks and make sure you test them out. Also make sure they don't fail on the empty list! (Sometimes a special case is required...)
   (a) Count the total number of elements in a list.
   (b) Append an item to the end of a list. The `insert` function above could be helpful, but perhaps try also to do it from scratch.
   (c) Deallocate an entire list (make the list empty, being sure to delete all the nodes).
   (d) Find the first occurrence of a value and remove it. *Hint: either use two pointers (to keep track of the node before), or you'll have to look "ahead" to* `next->data`. Extra challenge: do it *without a special case for the first node* (maybe using a `node**` to go through the list).
   (e) Remove all nodes containing a particular value. (*Note: you could of course just call the above over and over, but don't! Doing so can waste time, as you will repeatedly read through the beginning of the list.*)
   (f) Insert an element before the first occurrence of a given value. The same hint given for (1d) applies here. The same extra challenge also applies!
2. What is the total size (in bytes) of a linked list of n characters? Compare this to the total size of an array of n characters.
3. This is a bit contrived, but write a function that would detect whether or not a linked list contains a cycle (that is, one of the pointers points backwards to a prior node in the list). If the total number of nodes is n, approximately how many steps does your procedure take? And how much additional memory does it require?
4. For an extra challenge, see if you can do the previous exercise using $\approx n$ steps, and a *constant* amount of additional memory (independent of n).