Dynamic Memory

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

We won't insult your intelligence by wasting more than a few lines, explaining what a pointer is, and what it does.
However, towards the end of this piece, we explain how pointers to two-dimensional arrays work, since even experienced programmers have difficulty making this work properly.
Suffice to say, a pointer is a piece of memory, which contains the address of the variable, in which you are actually interested.

The declaration
char *p;
says, "The pointer p will contain the address of a char, when we say so. At present it contains garbage"

The declaration
char x;
p = &x;
says, "Pointer p now contains the address of the char variable x"

Enough said. Now let's talk applications.

It is not always possible, at compile time, to know how big to make all of our data structures. When we send an SQL query to the database, it may return twenty million rows, or it may return one.
The mechanism by which we persuade the operating system to give us memory on the fly, is called dynamically allocated memory. This memory is outside of the memory allocated to the process, in an area known as the 'heap', and our doorway into it, is a pointer to the first byte, returned by a function called malloc().

When we see code containing calls to malloc(), it may be difficult to see what it all means, because of the way it has been written, so it may be advantageous to assemble this code, piece by piece.

The basic function, takes one argument, the number of bytes of memory required, and returns a pointer to the first byte of this, like this:

```
char *pointer;
int size = 1000000;
        pointer = malloc(size);
```

Originally, malloc() used to return a pointer to char, since this pointed to one byte, as well as anything could, but this was too simple. These days, malloc() returns a pointer to 'void', which is exactly the same as a pointer to char, but the compiler won't let you use it, without a cast to your favourite data type.
Therefore, if we need a character array, in the midst of our computation, we would need to rewrite the call, to say:

```
        pointer = (char *)malloc(size);
```

If malloc() fails, it returns a NULL pointer, which we are duty bound to check, so we code it as:

```
if((pointer = (char *)malloc(size)) == NULL){
    printf("Memory allocation failed\n");
}
```

Now, it's starting to look ugly, and can be made downright hideous, by allocating an array of structures:

```
struct this{
   int one;
   int two;
   int three;
};
struct this *pointer;
int size = 1000000;

if((pointer = (struct this *)malloc(size * sizeof(struct this))) == NULL){
    printf("Memory allocation failed\n");
}
```

Occasionally, it isn't possible to know in advance, exactly how much memlry we need. We may be collecting data from several different sources, to place in one array, and only know how much each source will provide, when we access it.
There is another function, which permits us to alter the amount of memory which we previously allocated with malloc(), called realloc().
The realloc() function takes a pointer to a dynamically allocated block of memory, and a new size value, and returns a new pointer, to the extended memory:

```
char *pointer;

int newsize = 2000000;

temp = (char *)realloc(pointer, size);
```

or, to be pedantic,

```
if((temp = (char *)realloc(pointer, size)) == NULL){
    printf("Memory reallocation failed\n");
}
```

If we need to use our original pointer, for cosmetic, or aesthetic reasons, to point to the new memory, we simply reassign it:

```
pointer = temp;
```

Very brave programmers, who have faith in the order in which operations are performed, can save the cost of a pointer, by recycling the original pointer:

```
if((pointer = (char *)realloc(pointer, size)) == NULL){
    printf("Memory reallocation failed\n");
}
```

Don't do this because, down this road lies madness, and a few core dumps.

All of that was quite easy, really but, occasionally, we need an array of pointers to things which, themselves, are of variable size. For instance, we may be rifling the bank's database, looking for the loan payment records of all of its hapless customers. We don't know, in advance, how many customers there will be, or how many payments they made.

We start with the declaration of the two dimensional pointer:

char **pointer;

Some programmers declare this kind of pointer as 'char *pointer[]', since this looks like a pointer to an array, but it may be more intuitive to think of this as a pointer to a pointer.

What follows here, can be extremely difficult to understand, especially if we're not too au fait with pointers, and how they work, so the following explanation will be a little pedestrian. Experienced programmers are invited to skip it.

Let's say that we want to make 26 arrays, into each of which we can put 255 letters, such that array[0] has 255 A's, and array[25] has 255 Z's.

- Our pointer variable, above, as declared above char **q, a pointer to a pointer.
- Pointers come out of the box uninitialised, until you put a memory address into them. In other words, p is a space for the address of a pointer.
- At the moment, it contains garbage, pointing to garbage. We must first make it contain a real address, of a real pointer, even if that points to nothing yet.
- Therefore, we need malloc to attach real memory to q, to make it a real pointer to char **, like this:

```
if((q = (char **)malloc(n)) == NULL){
    printf("Error: cannot allocate memory\n");
    return(-1);
}
```

- Now that we have a real pointer, we need to make our pointers to 26 arrays i.e we ask malloc for sizeof(char *), for 26 pointers

```
for(i = 0; i < 26; i++){
    if((q[i] = (char *)malloc(sizeof(char *))) == NULL){
        printf("Error: cannot allocate memory\n");
        return(-1);
    }
}
```

- For the moment, we can do nothing with each q[i], since it contains no address, so this is our next step: to get each of the 26 to point to 255 bytes of char:

```
for(i = 0; i < 26; i++){
    if((q[i] = (char *)malloc(255 * sizeof(char *))) == NULL){
        printf("Error: cannot allocate memory\n");
        return(-1);
    }
}
```

- Now the job is done. We have real memory, arranged as char q[26][255], so let's give it some data..

```
for(i = 0, n = 0x41; i < 26; i++, n++){
    memset(q[i], n, 255);
    q[i][254] = 0x00;
}
```

All of the code above is deceptively simple. The trouble is, if you perform the memory allocation in the wrong order, or leave out a step, you may appear to have allocated the correct amount of memory, but the rows and columns may not be what was intended. As long as you don't put much data into it, everything will work. One day (about a week after your code is in production), a long string will need storing, somewhere near the end of the array. The runtime system will say 'Received SIGSEGV. Core dumped'.

Now what?

Our program has been completed, and runs very nicely, for the first eight hours.
Then, other users of our machine begin to complain that everything is running slow.
Finally, after twelve hours, the machine is so slow, that it needs a reboot.
Examination reveals that it has run out of memory, and further examination reveals that it is our code, which has gobbled it all up.
The term 'memory leak' features heavily in the email they send, terminating your contract...
The missing ingredient, is a useful little system call, named 'free()'. You pass it a pointer to a piece of memory previously allocated with malloc(), and it returns that memory to the operating system.
For a simple pointer, char *p it's a simple call:

```
free(p);
```

However, we need to be more careful when freeing a two-dimensional array, such as described above. The freeing needs to be done in the reverse order, to that in which the memory was allocated.

```
for(i = 0; i < 26; i++){
    free(q[i]);
}
```

Finally, we can simply say

```
free(q);
```

Linked Lists

When we are collecting data, the obvious, and simplest way of doing so, is to declare a structure, then declare a pointer to its type, and malloc an instance. As we acquire more data, we simply realloc our array of structures, and tack the data on to the end.

For getting rows of data out of a database cursor, this is great, and you shouldn't consider any other approach.
However, what happens if you want to remove the 154th data element from the array? Or, perhaps, insert the 154th element?

What if, you are storing data from several sources, like the roads on a map, which you need to attach to specific elements of your array, like the road junctions?

Not so simple.

Despite the mental picture conjured up by the word ‘list’, a linked list can be one dimensional, two dimensional or multi-dimensional. Apart from the street map mentioned above, another well-known application is an electronic circuit diagram, where there are components, connected by wires which, together, form a two-dimensional figure. Add to that, airline routes, railway systems, and the dynamically changing positions of pieces on a chessboard, and you get an idea of the usefulness of linked lists.

The Unix filesystem uses a linked list to map the blocks allocated to all of the files on a disk. As files are added, deleted, increase or decrease in size, the linked list is appropriately manipulated to reflect the current position.

Okay, so what, exactly, is a linked list?

One of my lecturers described linked lists as ‘a hundred blind men, holding hands in the dark’.
To stretch the analogy a little further, we can add that two of the men have a little red light attached to their heads, so you can see them.

Basically, a linked list is a series of data structures, with a special data structure at the head, and another special data structure at the tail of the list.

Let’s begin with a definition of the data structure.

```
struct queue {
   struct queue *fwd;
   struct queue *rev;
   char data[1024];
};
```

Ignoring the embedded data array, notice that there are two pointers, each to a type ‘struct queue’ within the data structure. One is a forward pointer (*fwd), and the other, a reverse pointer (*rev).
It is these pointers, which link the linked list. Since we are using a forward and a reverse pointer, this will be a doubly linked list, but for some applications, we can omit either pointer, and just create a singly linked list.
We’ll only consider the doubly linked list, as the amount of extra effort to do so is minimal.

First, we need to define the special structures for the head and tail.

```
struct queue *head;
struct queue *tail;
```

Since these are currently pointers to nothing, let’s initialise them to some real memory:

```
if((head = (struct queue *)malloc(sizeof(struct queue))) == NULL){
   printf("Can't allocate memory for head\n");
   return(-1);
```

```
}

if((tail = (struct queue *)malloc(sizeof(struct queue))) == NULL){
   printf("Can't allocate memory for tail\n");
   return(-1);
}
```

Now we have two blind men with lights on their heads, so we can see them, but they still can't see each other. Let's fix that.
We take the fwd pointer of the head, and attach it to the tail, and the rev pointer of the tail, and attach it to the head.

```
head->fwd = tail;
tail->rev = head;
```

To identify the head and tail, we need to set the rev pointer of the head to NULL, and to do the same with the fwd pointer of the tail.

```
head->rev = NULL;
tail->fwd = NULL;
```

Now the two blind men have placed their free hand onto a wall, which gives a clue as to how we know we've reached either end, when we're searching the list.
Now, we have to add an element to our list, which we can do at the head or at the tail. This is usually done within a subroutine, imaginatively called 'add_elmnt()' or something, since we don't want to repeat the code a few hundred times in our program.

First, we create an element

```
struct queue *elmnt;

if((elmnt = (struct queue *)malloc(sizeof(struct queue))) == NULL){
   printf("Can't allocate memory for elmnt\n");
   return(-1);
}
```

Then, to add this at the head, we do the following, in the following order. Changing the order may lead to attempts to attach to undefined pointers:

- We first take our rev pointer, and point it to the head, whose address we know.
  elmnt->rev = head;

- Then, we point our fwd pointer to the address pointed to by the head's fwd pointer.
  elmnt->fwd = head->fwd;

- Next, we take the rev pointer of the structure pointed to by the fwd pointer of the head, and point it to ourselves.
  elmnt-fwd->rev = elmnt;

- At this point, we are attached to both head and tail, and can safely detach the head's fwd pointer from the tail, and attach it to ourselves.
  head->fwd =elmnt;

Why did we do the acrobatics in the third step? Why not, instead, just say
tail->rev = elmnt; ?

The answer is, that we only know the position of the tail before we add the first element. However, we always know that the rev pointer of the structure following the head points back to the head.

If we're adding our elements to the end of the list, we follow the same method, except that we only know the address of the tail::

```
elmnt->fwd = tail;
elmnt->rev = tail->rev;
elmnt->rev->fwd = elmnt;
tail->rev = elmnt;
```

Let us now assume that we have a list of a hundred elements, and we want to scan it.
We can't do an indexed scan, since we don't have an array, and we can't make any assumptions about the addresses of the elements, since malloc just grabs memory from wherever it's free.

We need a pointer to struct queue, to traverse the structures, so we define a cursor
struct queue *cursor;

Then, we set up a loop:

```
for(cursor = head-fwd; cursor->fwd-fwd != NULL; cursor = cursor->fwd){
    /* do loopy things */
}
```

The initialisation is obvious: we just need to start at the first element, past the head of the list. Occasionally, the head and tail contain extra elements, such as queue length etc, so it may be necessary to start with 'cursor = head', but we have no such need.
The loop increment is equally obvious, in that the cursor sets its new address to that pointed to by the current element.

The loop termination conditions may not be so obvious. Why not just say 'cursor != tail'? Well, you can. However, it is not a good habit to get into, since some loops may have conditions within them, which cause the cursor to increment by more than one element. Down that road lies 'segmentation error – core dumped'...
Looking for a NULL fwd pointer is a guarantee that you've reached the end of the list, since only the tail has it set to NULL.

How about searching in reverse? Easy.

```
for(cursor = tail->rev; cursor->rev->rev != NULL; cursor = cursor->rev){
    /* do loopy things */
}
```

Now that we can insert elements, and create a long list, then search our list, this just leaves us with the task of deleting an element.

We need to take the same amount of care with deleting, as we took with adding an element. For the sake of example, let's say we want to delete any element with an empty data element in the queue structure;

```
for(cursor = head-fwd; cursor->fwd-fwd != NULL; cursor = cursor->fwd){
      if(cursor->data[0] == 0x00){
         cursor->fwd->rev = cusor->rev;
         cursor->rev->fwd = cursor->fwd;
         free(cursor);
      }
}
```

We take the rev pointer of the structure pointed to by our fwd pointer, and point it at the address being pointed to by our rev pointer. Next, we take the fwd pointer of the structure being pointed to by our rev pointer, and point it at the address being pointed to by our fwd pointer.
This has now bypassed our current element, so we can free it. Right? Well, the cursor address is still the same as that of the original element so, yes, we can.

However, what happens when we get back to the top of the loop? It'll try and set cursor to cursor->fwd. This will work – most of the time.

The problem is, that we just freed that piece of memory, which gives the operating system permission to give it to someone else. On an idle system (like the development machine), nothing will happen, and the loop will run to completion but, on a busy system (like production) another process might snatch that piece of memory, leaving our cursor to jump into the weeds, somewhere on the heap, and the testers will call you out in the middle of the night to fix it.

You could decide that you can live with the memory leak, and omit the 'free()' call, in which case, you should firmly close this page, and seek an alternative career.

To do it properly, what you need, is a second cursor.

```
struct queue *sentry;

for(cursor = head-fwd, sentry = cursor; cursor->fwd-fwd != NULL; cursor = cursor-
>fwd){
      if(cursor->data[0] == 0x00){
         sentry = cursor->rev;
         cursor->fwd->rev = cusor->rev;
         cursor->rev->fwd = cursor->fwd;
         free(cursor);
         cursor = sentry;
      }
}
```

Now, let's see what happens.
As soon as we've found the element we wish to delete, we set sentry to the previous element. When we've deleted our element from the list, and freed its memory, we set cursor to the same address as sentry, which is the element before the current one. The loop now advances the cursor, correctly, to the next element.

As we mentioned earlier, linked lists can be multi-dimensional. To create a two-dimensional list, suitable for creating matrices, maps, and other topological representations, we only need to change the basic element.

```
struct elmnt (
   struct elmnt *fwd;
   struct elmnt *rev;
   struct elmnt *up;
   struct elmnt *dn;
   char data[1024];
}
```

Now, instead of just a forward and a reverse pointer, we have an up and a down pointer, as well.
The process of adding an element now also includes setting the two latter. If the element being added is just another linear element, we set the up and dn pointers to NULL but, if it is a branch point, we have to set them to point up to the newly added structure, and back down to the branch point.

Let's say we already have our linear linked list, and we wish to add one element above, and another below the first element after the head.

```
elmnt->dn = head->fwd;
head->fwd->up = elmnt;
head->fwd->dn = NULL;
elmnt->up = NULL;
```

Note that we leave no trailing pointers, but terminate them with a NULL, so we can find the end of the branch.

Next, we add a new element below the first element after the head.

```
elmnt->up = head->fwd;
head->fwd->dn = elmnt;
head->fwd->dn->dn = NULL;
```

Note that we don't set the head->fwd->up pointer to NULL, as we just added an element there.

Traversing such a list will require two cursors, in two nested loops. The main loop traverses the list in a horizontal direction, with hcursor, while the two inner loops traverse the branches vertically up or down, with vcursor.

```
for(hcursor = head-fwd; hcursor->fwd-fwd != NULL; hcursor = hcursor->fwd){
   if(hcursor->up != NULL){
      for(vcursor = hcursor; vcursor->up != NULL; vcursor = vcursor->up){
          /* traverse the upward bound list */
      }
      if(hcursor->dn != NULL){
         for(vcursor = hcursor; vcursor->dn != NULL; vcursor = vcursor->dn){
          /* traverse the downward bound list */
      }

   }
```

}

Three dimensional linked lists work in exactly the same way, with an element defined as

```
struct elmnt (
    struct elmnt *fwd;
    struct elmnt *rev;
    struct elmnt *up;
    struct elmnt *dn;
    struct elmnt *out;
    struct elmnt *in;
    char data[1024];
}
```

where ‘out’ and ‘in’ are the z-axis pointers.
It is left as an exercise for the reader, to design a function to add such an element to a linked list, and then to define a traversal function.