STEP THREE STUDY NOTES

This document contains some brief notes on topics covered in *Step Three* of the course '*Advanced C Programming: Pointers*'. You will find a similar set of study notes for each step of the course. These notes aim to *summarise* key concepts described in the lectures. You may find it useful to read the study notes *after* you have watched the lectures in each step of the course.

Some notes refer to specific sample programs supplied in the code archive of the course. If you need to explore the topics in more detail, please refer to the sample program. In these notes, relevant program names are shown like this:

ASampleProgram

NOTE:

Step Three explains some challenging concepts and techniques. You may find that it takes a while to understand exactly how linked lists work. If you have never, or only rarely, used singly and doubly linked lists, stacks or queues before, you may need to watch some lessons more than once. The best way to understand how lists work is to use the sample programs. Step through the code in a debugger and watch how the pointers are updated when new elements are added or removed from a list. There is no getting away from the fact that this step contains some quite difficult material. These study notes don't describe everything covered in the video lessons. Instead they focus on *key concepts* to help you to understand the most important ideas from this part of the course.

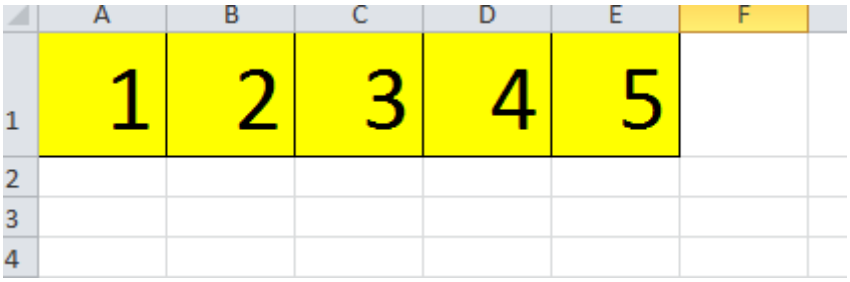
Lists and Data Structures

Key Points

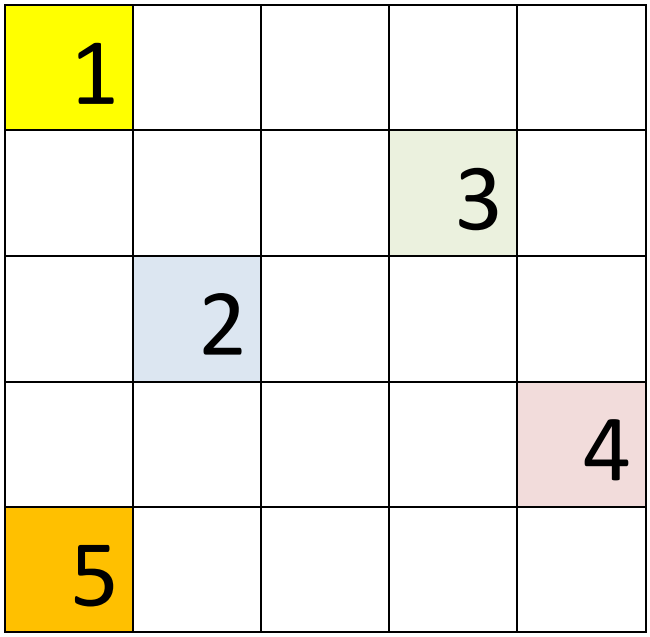
- An array (and its elements) occupies a single block of memory
- Elements in a linked list are likely to be at non-adjacent memory locations
- You need to allocate memory for each element in a linked list
- Arrays have fixed lengths. Once declared, an array's size cannot be changed
- Linked lists may expand and contract as elements are added and removed
- A queue is a list in which the first thing added is the first thing that is removed
- A stack is a list in which the last thing added is the first thing that is removed
- Pointers may be set to point to functions as well as to data

ELEMENTS IN A LINKED-LIST ARE NON-CONTIGUOUS

Elements in an array are stored next to one another (contiguously) in memory. An array of 5 elements is like a row of 5 spreadsheet cells. Each cell represents a chunk of memory sufficient to hold a single element of a certain type. Enough memory to hold exactly 5 elements is allocated when I declare the array. But once I have an array of this size, the size is fixed. I cannot now add a 6th or 7th element.

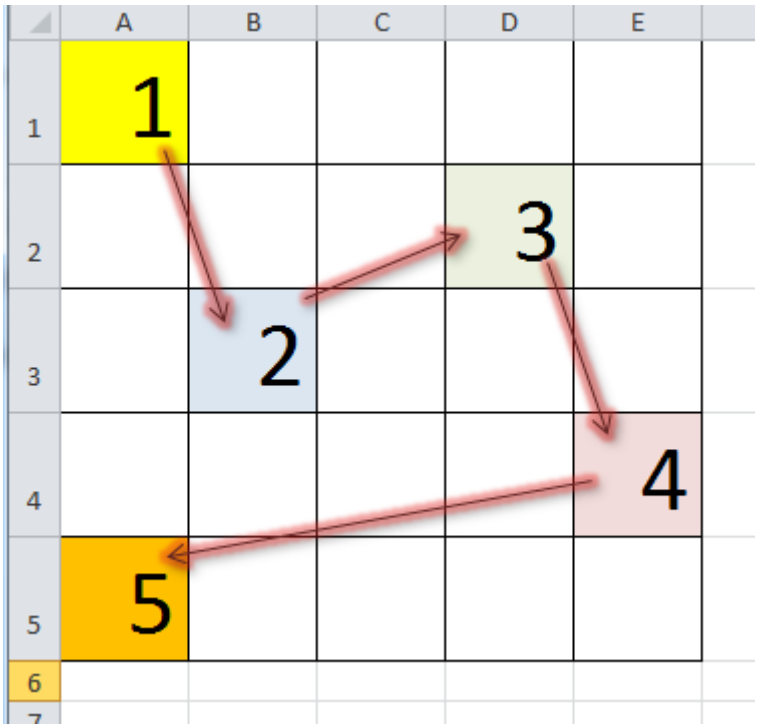


With a linked list I don't allocate a fixed block of memory in advance. Instead, I allocate memory on an element by element basis. If I create a list that contains two elements, I allocate just enough memory to hold two elements. If I now want to add a 3rd element to the list, I allocate another chunk of memory for that new element and then add it to my list. Since I'm adding, and maybe deleting, elements at various different points of my program, their locations in memory are likely to be all over the place.



The elements 'next to one another' in a linked list don't, in fact, occupy a single sequential block of memory, like the elements in an array. So I can't just iterate through memory by counting along a certain number of bytes, from one element to the next, as I did when using pointer arithmetic, or by using array indexing with square brackets.

How then does element 1 in my list know where element 2 can be found? The answer is that it keeps a *pointer* to element 2. The pointer stores element 2's address. So if I know where element 1 is, I can use the pointer in element 1 to find element 2. From element 2, I can use a pointer to find element 3 and so on.



COMMON TYPES OF LISTS

Singly-linked list: A **singly-linked list** is one in which each element contains a single pointer to the next element in the list.

Doubly-linked list: A **doubly-linked list** is one in which each element contains two pointers – one to each adjacent element (the one 'before' it and the one 'after' it in the list).

Specific types of list such as **queues** and **stacks** can be implemented as linked lists.

THE HEADER STRUCT

It is common practice when manipulating lists to keep track both of the first element and the last element in that list. You could do that by defining two pointer variables – a *headpointer* (one that points to the *first* element) and a *tailpointer* (which points to the *last* element). These two pointers could be updated whenever changes are made to the list. However, it is a bit neater to put these two pointer variables into a special **header structure**. For now, just keep it in mind that **the header struct is treated specially; it is *not* an element of the list itself.**

You can see an example of the header struct in the *Queue* sample program. Here, each **element** in the list is defined by a struct containing two pointers, `next` and `prev`, plus some `data`.

Queue

```
// This struct defines a list item
typedef struct listitem {
    struct listitem *next;      // pointer to next item
    struct listitem *prev;      // pointer to previous item
    int data;                   // some data
} LISTITEM;
```

But the list **header** struct only contains two pointers – one to each 'end' of the list. The header struct stands '*outside*' the list itself. Its sole function is to keep track of (to 'point to') the first and last elements in the list:

```
// This struct defines the list header
typedef struct {
    struct listitem *first;     // pointer to first item
    struct listitem *last;      // pointer to last item
} LISTHDR;
```

As new elements are added or deleted, the header struct's pointers are updated. *When the list it **empty**, the two pointers in the header struct point to the header struct itself* (`&head`).

```
head.first = (LISTITEM*)&head;
head.last = (LISTITEM*)&head;
```

Let me emphasise again: **the header struct is *not* a part of the list**. It is used to keep a 'record' of the list state. You could, if you wished, keep more information in such a header. For example, you could add data fields to store the current number of elements in the list, the number of new elements that have been added or deleted during program execution, and so on. In my example programs, I generally keep things simple. The header simply keeps track of the two terminating (the 'first' and 'last') elements in a list.

QUEUES

A queue is a list in which the first thing you add is the first thing that you remove. A queue is sometimes called a First-In-First-Out, FIFO, structure.

STACKS

A stack is a list in which the last thing you add is the first thing that you remove. A stack is sometimes called a Last-In-First-Out, LIFO, structure.

FUNCTION POINTERS

Pointers may be set to point to functions. Function pointers are typically used when handling asynchronous events. It is quite possible that you may never need to use function pointers but even so it is worth knowing about them. Here is how I can declare a pointer to a function that takes an `int` argument and returns an `int`.

```
int (*test)(int);
```

It is common practice to *typedef* function pointers in advance. Once you have the correct `typedef`, you can use the type in function pointer declarations.

```
typedef int(*PFI)(int);
```

So this declares a type called `PFI` which is a pointer to a function that takes one integer argument and returns an integer value. Now it's easy to declare a function pointer variable, like this:

```
PFI test;
```

Function pointers are sometimes put into arrays so that a single piece of code can call different functions using function pointers. Refer to the sample program *FunctionPointerArray* for a working example of this.

FunctionPointerArray

```
typedef int(*PFI)(int);

int identity(int a) {
      return a;
}

int square(int a) {
      return a*a;
}

int cube(int a) {
      return a*a*a;
}

int fourth(int a) {
      return a*a*a*a;
}

PFI power[] = { identity, square, cube, fourth };
```