STEP FOUR STUDY NOTES

This document contains some brief notes on topics covered in *Step Four* 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 Four examines a number of problems associated with pointers. These are by no means the only pointer problems you may encounter, but they are among the more common ones. The best way to understand these problems is to work your way through the samples supplied in the code archive. These examples contain explanatory comments. You may find the example of deep and shallow copying takes some effort to understand. The kind of bugs illustrated here are, in real-world applications, frustratingly difficult to find and fix. So it's worth taking the time and effort to understand them now so that you can avoid them in future.

Common Pointer Problems

Key Points

- In a shallow copy the **pointers** are copied while the **data** is not
- In a deep copy, the **data** is copied while the **pointers** are not
- Shallow copies are fast but may be unsafe
- Deep copies are slower but safer
- Be careful when using `free` that you free all memory – but don't do it twice!
- Don't try to use memory that has been freed
- Make sure you don't try to use an 'out of scope' pointer
- Don't try to dereference a NULL pointer
- Not all errors show up immediately – but may cause problems later on
- Some compilers/platforms handle errors in different ways
- Some errors may *seem* to go away when using a debugger

Deep and Shallow Copies

There are two ways of making a copy of data in a list. The first is to create pointers to existing data. That is called a **shallow copy**. There is one bit of data but two pointers to it.

The second way is to make a new copy of the data itself. That is called a **deep copy**.

Shallow copies are typically fairly fast because no *real* copying (of data) is done. You are just creating a new reference – a *pointer* whose value is the *address* of some existing bit of data.

Deep copies are slower because you have to make a copy of some data into a new memory location.

If the data items are small and there aren't many of them, the speed difference between deep and shallow copies would be negligible. But in a real-world program, the data could be enormous – for example, you might need to make copies of huge lists containing vast numbers of structs each of which contains large amounts of data.

Problems with Shallow Copies

When two lists contain pointers to the same data items, any operation done to the items in one list (that is, the data referenced by the *pointers* in one list) can have unintended side-effects on the other list. For example, if I changed the value of an element in list 2, the value of the element in list 1 would also change. Worse still, if I deleted an element from *list 2*, then *list 1* might continue to try to use that same element (possibly causing your program to crash). That's because the *pointers* in *both lists* point to the same data items.

DeepShallowCopy

To understand the problems associated with shallow copies (and how easy it is to make a mistake when copying lists!) work through the sample program, *DeepShallowCopy*. This contains numerous comments to explain how the code works and where problems can arise.

Other Common Problems

Incorrect Casts

Once you cast a pointer, it is up to you to make sure that it points to the type to which it was cast. Here I cast the generic pointer `p` to the `MYSTRUCT` struct type:

```
p = (MYSTRUCT*)calloc(COUNT, 24);
```

Without casting `p` the compiler only knows that it is a pointer to *something*. It could point to anything. The cast tells it that it will point specifically to a `MYSTRUCT` struct. From now on, it's *the programmer's* responsibility to make sure that *that* is really what it points to. If you now point `p` to something other than a `MYSTRUCT` struct, the behaviour of the program will be uncertain. In fact, it will probably crash.

When you cast one type to another type you are not actually *changing* one type to another type. A cast is an instruction to the compiler to let you *treat* one type as another type. It is up to you to make sure that you actually use the type in your cast. If you don't do so, the compiler won't help you – so take care!

FREEING MEMORY THAT HAS ALREADY BEEN FREED

You allocate some memory:

CommonProblems

```
char *b;
b = (char*)malloc(10);
```

Then later on, when it's no longer needed, you free it:

```
free(b);
```

But at some stage yet later on (having forgotten you freed it previously) you do so again:

```
free(b);
```

Freeing the same memory twice can cause unpredictable behaviour.
Don't do it!

With some compilers, you may not even see any problems when a pointer is freed twice. That doesn't mean that there *is* no problem! The simple fact is that the way errors are handled and the problems which they may or may not produce sometimes varies according to the platform and the compiler and runtime you are using - and also the compiler settings you may have set. Also (and this is quite common) *the behaviour of errors may change when you are running your program in the debugger*. This can be particularly frustrating. For example, sometimes when you debug your program there may seem to be no errors (because the debugger itself changes the behaviour of the program), then when you run it, the program crashes! Generally, assume that if an error may even *potentially* occur at some time, one day it *will* occur. So make a habit of trying to find and fix potential bugs even if they don't seem to be causing any problems.

Memory Leaks

Memory leaks occur when you allocate memory but don't free it when it is no longer in use. This was mentioned earlier in the course. Bear in mind, however, that not all memory leaks are obvious. Look at the code below. I create and allocate some strings by allocating 10 bytes for the char pointer b inside a loop.

Then at the end I free b. That looks ok. But it isn't. I've actually lost 90 bytes of memory. That's because each time I set the b pointer to point to a new chunk of memory inside the loop I lose track of the chunk of memory that I allocated on the previous turn through the loop.

```
char *b;
for (int i = 0; i < 10; i++) {
    b = (char*)malloc(10);
}
free(b);
```

Remember that even though b is a local variable, malloc() allocates memory globally – on the heap – so that memory remains allocated even after you exit the function containing the local variable, b.

Using memory that has been freed

CommonProblems

```
char *b, *c;

b = (char*)malloc(100);
c = b;
free(b);
strcpy(c, "hello");
printf("c is %s\n", c);
```

Here I allocate 100 bytes for my char pointer b. I then assign b to c. Later on, I free b. Now I use my char pointer c. I copy the string "hello" into the memory to which it points. But that's a mistake. Because that memory is no longer allocated. I freed it when I called free with the b pointer. The results of this are, at best, unpredictable and, at worst, disastrous.

POINTERS OUT OF SCOPE

Be careful that you do not try to use a pointer outside its scope. Consider this:

CommonProblems

```
char* problem4() {
    char b[20];
    printf("enter your name ... ");
    gets(b);
    return b;
}

int main() {
    printf("you typed %s\n", problem4());
    return 0;
}
```

I've declared an array of chars which is initialized when the user enters a name. Then I return the array. And in the `main()` function I attempt to print the returned name.

But this doesn't work. That's because the variable `b` is local to the `problem4()` function. It is an array, which is an address, and the return value of the function is that address. But, as `b` is a local variable, its data is stored on the *stack*. The stack, remember, is an area of memory that stores *local* variables but once the function exits the stack is, in effect, cleaned up. The data is not retained for later use. So the address is returned correctly but the data at that address no longer exists!

Dereferencing a null pointer

A pointer that is set to NULL doesn’t point to any usable memory so if you try to use a NULL pointer you will have problems. Look at this:

CommonProblems

```
void problem5() {
   char *b;
   b = (char*)malloc(10);
   b = NULL;
   b[0] = 1;
}
```

The final line of code tries to assign some data to a NULL pointer. On a desktop computer, this error should normally cause an immediate and obvious problem. However, if are programming microcontrollers, without virtual memory management, the error might not be at all obvious. This error is sometimes called ‘dereferencing a null pointer’. This is a serious bug and should be avoided at all costs.