CTEC2901 Data Structures and Algorithms 2013-14

C Preliminaries (Part Two)

David Smallwood

Contents

1 Topics

In this section we look at more advanced use of pointers. In particular *pointers to pointers* and *pointers to functions*.

1.1 Pointers to Pointers

Suppose that you have a struct data type, for the purpose of this discussion let's call it `struct info`, and that you create instances of `struct info` dynamically as they are needed. This means that you will be handling pointers to structs because malloc will return a memory address. E.g

```
struct info * x = (struct info *)malloc(sizeof(struct info));
```

This is fairly standard. Now suppose that you wish to write a function that takes a pointer to `struct info` object (src) and returns a fresh copy of that object (dst). The following cannot work:

```
void copy(struct info * dst, struct info * src)
{
  dst = (struct info *)malloc(sizeof(struct info));
  *dst = *src;
}
```

```
int main()
{
  struct info *a, *b;
  ...
  copy(a,b);
```

The reason this fails is because the pointers themselves (a and b) are passed by value so any changes in the copy function are local. What is needed is to pass the pointer a itself by reference. However, you cannot simply write:

```
  copy(&a,b);
```

because this will not compile:

```
warning: passing argument 1 of ‘copy’ from incompatible pointer type
```

The reason this fails to compile is becuase `&a` is a *pointer-to-a-pointer-to-a-struct* and the formal parameter declaration in the function expects a pointer-to-a-struct. We need to redefine the copy function:

```
void copy(struct info ** dst, struct info * src)
{
  *dst = (struct info *)malloc(sizeof(struct info));
  **dst = *src;
}
```

Notice now that we have used two stars `**` in the formal parameter definition. This means that a pointer is being passed by reference (a pointer to a pointer) which, in turn, allows for a new memory address to be returned to the main program's (pointer) variable.

Another way of interpreting "pointers to pointers" arises when fields within structs contain pointers – such as in a linked list. For example, it is common in a linked data structure to have a node pointing to the 'next' node, and so on. The following struct definition is taken from the circular list (`clist`) data structure that will be studied later in the course. A node contains an item (of type `any`) and two pointers to the next/previous nodes in the list.

```
struct node
{
    any item;
    struct node * next;
    struct node * prev;
};
```

The code for one of the functions (a delete method) contains the following line of code:

```
p->prev->next = p->next;
```

See how the left hand side of the assignment refers to the 'next' node of the 'previous' node of node `p`. The *arrow notation* is particularly useful here for specifying such chains.[1]

1.2 Pointers to Functions

There are occasions when a generalised pattern of behaviour can be observed in many functions. When such patterns are observed it can often be useful to design software in such a way that the patterns are reflected in the design. For example, suppose you wish to write two functions: one converts a string to upper case and another converts a string to lower case.

```
void up_case(char *c)
{
  while(*c != '\0') {
      if (*c >= 'a' && *c <= 'z') *c-=32;
      c++;
  }
}
void lo_case(char *c)
{
  while(*c != '\0') {
      if (*c >= 'A' && *c <= 'Z') *c+=32;
      c++;
  }
}
```

It should be clear that these are very similar. In fact, one pattern of computation is the loop that traverses each string and processes each character one at a time. Could it be possible to abstract this behaviour into a common function and then *parameterise* it to determine the required behaviour? Consider this:

```
void up_case(char *c)
{
    if (*c >= 'a' && *c <= 'z') *c-=32;
}
void lo_case(char *c)
{
```

[1] Compare this with the equivalent, but less elegant, $(*(*p).prev).next$.

```
    if (*c >= 'A' && *c <= 'Z') *c+=32;
}
void traverse(char *s, void (*f)(char *c))
{
    while(*s != '\0') {
        f(s);
        s++;
    }
}
```

Here we have have encapsulated the traversal into a single function. The `up_case` and `lo_case` functions now operate on a single character (passed by reference so it can be modified). The significant point to note in the formal parameter list of the `traverse` function is the function-type parameter `f`. The syntax is rather strange but "`void (*f)(char *c)`" declares `f` to be a pointer to a void function that takes a single character passed by reference. The syntax may be odd but the concept is powerful as the following demonstrates:

```
int main()
{
  char title[] = "CTEC2901 Data Structures and Algorithms";
  printf("Original:   \'%s\'\n", title);
  traverse(title,up_case);
  printf("Upper case: \'%s\'\n", title);
  traverse(title,lo_case);
  printf("Lower case: \'%s\'\n", title);
}
```

See how the `traverse` function has been *parameterised* by `lo_case` and `up_case` so that the traversal abstraction is written once and re-used twice.[2]

Let us look at another example. The code below is a simple bubble sort algorithm for an array `a` of `N` int values:

```
void bubblesort( int a[], int N )
{
    int i,j;
    for (i=N-1; i>0; i--)
        for (j=0; j<i; j++)
```

[2]For those readers familiar with functional languages this may remind them of the `map` function which also abstracts the act of traversing a structure from the detail of any particular structure or any particular function. In Haskell, for example:
$map :: (a \to b) \to [a] \to [b]$
$map\ f\ []\ =\ []$
$map\ f\ (x : xs) = f\ x\ :\ map\ f\ xs$

```
            if (!(a[j] <= a[j+1]))
                SWAP(a,j,j+1);
}
```

The mechanics of the algorithm is not the main point of this example but rather the use of the relational operator (<=) in the if-statement condition. The items are swapped if they are not in order as defined by the <= order relation. This will sort the numbers into *ascending* order. But what if we wanted the numbers to be sorted into *descending* order? We would need to copy the entire code into a new function but change the relational operator to >= instead. This code copying can be avoided by abstracting the algorithm and making the relational operator a parameter. It can be achieved like this:

```
void bubblesort( int a[], int N int (*rel)(int,int))
{
    int i,j;
    for (i=N-1; i>0; i--)
        for (j=0; j<i; j++)
            if (!rel(a[j],a[j+1]))
                SWAP(a,j,j+1);
}
```

The algorithm has not been changed but the use of the `rel` parameter allows the specific ordering relation required to be supplied as an actual parameter when the function is called.

2 Exercises

Practical Exercise 1

Write a program that uses the `copy` function described in section 1.1 to create a fresh copy of a pointed-to object. For the purpose of the exercise use the following definition:

```
struct info
{
  int total;
};
```

Your main program will begin something like this:

```
int main()
{
  struct info *a, *b;
```

```
a =
b =
```

You should print out the value of the copied object to ensure that the copying worked as expected.

Practical Exercise 2

This exercise refers to the example code in section 1.2.

1. Write a program using the C code provided: the `traverse` function, the `up_case` and `lo_case` functions, and the main program. Compile and run the program.
2. Add some more functions to your program that manipulate single characters and use them with `traverse`.

Practical Exercise 3

Adapt the `bubblesort` function to work with characters. You will need to write a `SWAP` function to complete this. Then write two order relations for characters – one for $\leq$ and one for $\geq$ – these should be C functions with the signatures:

```
int less_eq(char x, char y);
int greater_eq(char x, char y);
```

and use your bubblesort routine to sort the character string firstly in ascending order and secondly in descending order. (You should be passing your `less_eq` and `greater_eq` functions as parameters to `bubblesort`).