CTEC2901 Data Structures and Algorithms 2013-14

C Preliminaries (Part One)

David Smallwood and Amelia Platt

Contents

1 Topics

We assume a basic working knowledge of C as contained in the Level 4 module CTEC1401 Programming in C. The classical data structures will be written in C which will permit a detailed presentation of the underlying implementation issues. *Static* implementations will utilise fixed storage using an array. *Dynamic* implementations will utilise dynamically allocated memory so that structures can grow (and shrink) as required. Dynamic implementations require the manipulation of small blocks of memory (using structs) linked together using memory addresses (pointers).

We will take an "abstract data type" view of data structures in which the behaviour of a data structure is defined by its operations. These operations are implemented using C functions which will be packaged into separate libraries and compiled independently from applications that use them. It is therefore necessary to understand fully the mechanisms in C for passing data to functions (by value and by reference) and for compiling modules separately.

In this tutorial we will summarise four important topics that will feature prominently in the implementation of data structures in C. Please note that this summary does not provide a comprehensive treatment of each topic (a working knowledge is assumed) but rather it highlights specific points of interest that will recur during the course of the taught programme.

1.1 Pointers

A pointer is a variable whose value is a memory address. In C pointers are used to process arrays; to pass data by reference to a function; to build dynamic data structures; and to point to functions when providing callbacks. Callbacks are used, for example, to pass (the address of) a function to be applied to each element when iterating over a data structure, or when an ordering relation needs to be passed to a sort function in order to parameterise its behaviour.)

The C programmer must declare pointer variables and manage the allocation and deallocation of memory explicitly. This provides the programmer with very fine control of the system resources but it must be undertaken with great care. Modern programming languages such as Java automate a great deal of the memory management (e.g. through garbage collection) and relieve the programmer from having to deal with this low-level activity. In C, however, the programmer must take full responsibility.

Although pointers are fairly low-level it is extremely useful to study them. They provide a deeper insight into the way in which data structures are implemented on standard architecture machines. Pointers underlie the implementation of objects in modern object oriented languages and an understanding of pointers helps in understanding their language abstractions and how to use them effectively. Furthermore, most serious applications written in C itself will make extensive use of pointers, so a proper understanding of them is essential.

You should understand fully how to declare pointer variables, e.g.:

```
int *ar;
char *s;
```

and how to use them. Pointer values can be obtained either by using the *address-of* operator (&) applied to an lvalue (e.g. a variable) or by acquiring a fresh memory address via a function call such as `malloc`.

Pointers are used to pass variables *by reference*, for example:

```
void foo(int * x);
```

Pointer variables may contain pointers to other variables. This happens fairly frequently in C and is demonstrated by the passing of command line arguments to a main function:

```
int main(int argc, char **argv);
```

In this definition, `argv` is a pointer to a pointer to char – or, in other words, an array of strings. As another example, suppose that a function needs to pass a pointer by reference:

```
void foo(int **v);
```

Here we see that it is an *address of an address* of a value that is passed to the function. As a final example, we show how to pass a function as a parameter to another function. The syntax is rather unusual.

```
void sort(int *ar, int size, int (*rel)(int,int));
```

In this definition of a sort function, there are three parameters. Firstly there is the array, `ar`, of int values to be sorted which is passed by reference. Secondly, there is a size parameter stating the number of elements in the array. Finally there is an ordering relation, `rel`, which can be passed any conformant function. For example:

```
int smaller(int x, int y)
{
  return x<=y ? x : y;
}

int main()
{
  int *n, size;
  ...
  sort(n, size, smaller);
  ...
}
```

Although the actual parameter consists of the function's name (`smaller`) it is actually the function's memory address that is passed.

1.2 Arrays

An array in C is a block of contiguous memory in which a predetermined number of *similar* data values can be stored. The block of memory has a specific size and we refer to the "locations" in the array by using an index. For example, the declaration

```
int ar[100];
```

declares an array, called ar, that can store 100 int values indexed from zero (ar[0]) to 99 (ar[99]). The compiler will reserve sufficient memory to store 100 int values (on a 4-byte int machine this will be 400 bytes) and the user can access each of these int values using the array subscript notation ar[0], ar[1], etc. It is an error to access the array outside its defined bounds: therefore the expressions a[-1] and a[100] are illegal accesses to the array despite the fact that C will not prevent you from writing them. Moreover, C will attempt to deliver the value referred to by an illegal subscript – this is not to be "helpful" (because, arguably, it is not) but rather a consequence of the way in which array subscripts are implemented.

An array is identified by its base memory address and an array subscript is interpreted as an offset to be added to the base memory address. Note that the addition referred to here is performed using *pointer arithmetic* which is a special kind of arithmetic that respects the underlying element sizes. Returning to our example, suppose that ar is at memory address 1000; then the subscripted element ar[7] dereferences the seventh element in the array calculated as follows: *(ar + 7). In our example, the computed memory address would be 1000+(7*4) = 1028. The multiplication by four is needed because each of the array items occupies four bytes of memory. C's pointer arithmetic does all of this automatically which is why ar[0] refers to the value at address 1000 and ar[-1] refers to the "value" at address 996.

The relationship between C's array subscripting and pointer arithmetic is very useful for writing short, efficient code but does need to be treated with care. At runtime an illegal subscript may not be caught and the program will (probably) be "wrong". (We say, 'probably', because such behaviour may in certain circumstances be planned - e.g. malware. C is very vulnerable to this type of attack.)

When an array is declared within a main program, e.g.

```
int main()
{
  int ar[100];
  ...
```

then it is declared *statically*. This means that the memory is determined at compile-time and may not be changed subsequently. The memory address for the array is determined at link time and may not be changed subsequently. The obvious drawback of a statically allocated array is its rigidity. An underused array wastes space but an array that is not big enough for a particular set of data cannot be extended. Alternatively, it is possible to allocate an array dynamically. To do this we declare a pointer to a block of memory:

```
int *ar;
```

At this stage it is wrong to try and access the array by subscript because no memory has been allocated. Thus, ar[3], for example, is an error. However, suppose we now wish to allocate memory to store an array of k int values (where k is some computed or read-in value) then we write:

```
ar = (int *)malloc(k * sizeof(int));
```

This code will cause the runtime system to allocate memory for an array of k int values dynamically. Once this has been (successfully) executed it is then legal to access the array using the normal subscript notation (ar[0] up to ar[k-1]). Note that it is possible for a dynamic allocation to fail. The standard C function malloc will return a zero memory address to indicate that memory allocation failed. Properly designed C code should always check for this eventuality and take appropriate action. If malloc fails and no check is made then it is possible to reference memory that may not have been allocated which will lead to unpredictable results. A standard C idiom is:

```
if (!(ar = (int *)malloc(k * sizeof(int))))
  /* deal with the error */
else {
  /* allocation was successful */
  ...
}
```

This works because in C the result of an assignment expression is the thing assigned (in this case, a memory address, or zero (NULL)), and the value zero (NULL) is treated as *false* whereas any non-zero value is treated as *true*).

1.3 Structs

Whereas an array is used for storing a collection of same-typed values a struct is used for grouping together related values which may be of different types. For example, data relating to a 'person' may include a 'name' (string) and an 'age' (int):

```
struct person
{
  char name[20];
  int age;
};
```

The "dot" notation is used for selecting fields within a struct value. For example, given the declaration

```
struct person joe;
```

then the notation joe.name and joe.age is used to access each of the fields separately. It is the compiler's job to allocate sufficient memory space to store an instance of a struct: the amount of memory allocated may be slightly larger than the minimum required for reasons of access efficiency but this is a detail that the programmer need not be concerned with. However, it is important not to make assumptions about the memory addresses of fields within a struct. If a memory addresses is required then it should be determined using the *address-of* operator: for example, `&(joe.age)`.

Structures can be nested as required. For example, the following struct definition of a transaction relates a struct person (customer) and a product id:

```
struct transaction
{
  struct person customer;
  int product;
};
```

Suppose we have 100 transactions stored in an array:

```
struct transaction trans[100];
```

We can access the name of the customer involved in the fifth transaction (say) using a combination of array subscript and field selection notation: `trans[4].customer.name`.

Nested structures should present no difficulty because the syntax for accessing the fields is consistent at all levels. Structs and arrays can be mixed freely as required. In the previous example we have an array of structs which themselves contain structs which, in turn, contain arrays (the name field is an array of char). Once again we stress that such nesting is not uncommon and should not cause problems provided the programmer is careful.

An interesting situation arises in the following case: suppose we wish to define an employee who has a name and an age (just like a person struct defined before) but also a manager who is also an employee:

```
struct employee
{
  char name[20];
  int age;
  struct employee manager;  /* will not compile */
};
```

The recursive use of struct employee is natural (an employee's manager is also an employee) but will not compile. This is because when the compiler meets the definition of the third field in this struct definition it needs to know what is a "struct employee". However, this has not yet been defined (since the compiler is part-way through its very definition) which means that the compiler does not know how much memory to allocate to this field and the entire definition fails. This example may appear to be somewhat contrived but, in fact, such recursive references are used frequently in the dynamic implementation of data structures. The correct way to define the employee is as follows:

```
struct employee
{
  char name[20];
  int age;
  struct employee *manager;
};
```

The use of the *pointer-to-struct-employee* works because the compiler knows precisely how much memory needs to be allocated – sufficient to store a single memory address. Once again, suppose that we have an employee instance called `joe` then to reference Joe's manager's name we would need to write `joe.manager->name`. Note that this is a shorthand for the slightly less elegant `(*joe.manager).name`. The use of the arrow notation is usually preferred. Joe's manager's manager could be accessed using `joe.manager->manager->name` and so on. However, please note that to dereference a pointer it must not be NULL. I.e. if Joe had no manager (NULL) then the expression `joe.manager->manager` will fail at runtime. Such issues need to be borne in mind most carefully when implementing dynamic data structures using such recursive struct definitions. As a final example note that a circular reference could be achieved by writing `joe->manager = &joe`!

1.4 Separate Compilation

Even the simplest C programs use library functions which are spread across many other files. Some of these files are system libraries (e.g. `stdio`, `stdlib`) and some of them are user-defined libraries. The key to reusing code from other (library) files is separate compilation. Each library has an associated relocatable object (.o) file and, in many cases, a header (.h) file. It is the task of the linker (`ld`) to put all of the .o files together, resolving all the memory addresses, into a single executable. Usually the linker is called implicitly via `gcc`.

When writing library files the convention we use is to write a header file surrounded by `#ifndef` / `#define` and `#endif` (in order to avoid multiple imports of the same definitions). For example, suppose we were to define a counter library object:

```
#ifndef COUNTER_H
#define COUNTER_H

typedef struct counter_implementation counter;

counter * new_counter(int limit);
void counter_inc(counter *c);
int counter_atlimit(counter *c);
int counter_value(counter *c);
void counter_reset(counter *c);
void counter_release(counter *c);
```

```
#endif
```

The use of typedef seems to be incomplete because struct counter_implementation has not been defined. However, the compiler will permit the actual definition of struct counter_implementation to appear later (in the .c file). This simple technique allows us to *hide* from the header file (and hence readers of the header file) the detail of the implementation. This *data hiding* is useful because it allows us to change the implementation without affecting any code that imports the header file. This is a good form of *coupling*. When a client program wishes to use a counter it must declare a pointer-to-a-counter (*counter) variable. Any attempt to declare a simple counter variable will not work because the compiler will not know how much memory to allocate to the variable. For example, an attempt to declare

```
counter c;
```

leads to the gcc error:

```
error: storage size of ‘c’ isn’t known
```

Alternatively, the compiler accepts

```
counter *c;
```

because the amount of memory required is known (i.e. sufficient for a memory address).

Our simple counter example provides methods to create and delete counter instances. Furthermore, counters can be incremented, interrogated, and re-set. Counters have a limit. Here is a possible implementation (.c) file:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include "counter.h"

struct counter_implementation
{
    int value;
    int limit;
};

counter * new_counter(int limit)
{
    counter *c;
    if (c = (counter*)malloc(sizeof(counter))) {
        c->limit = limit;
        counter_reset(c);
        return c;
    }
    else {
        printf("Error: new_counter (malloc failed)\n");
        exit(1);
    }
}

void counter_inc(counter *c)
{
    assert(c!=NULL);
    if (!counter_atlimit(c))
        c->value++;
}

int counter_atlimit(counter *c)
{
    assert(c!=NULL);
    return c->value == c->limit;

}

int counter_value(counter *c)
{
    assert(c!=NULL);
    return c->value;
}

void counter_reset(counter *c)
{
    assert(c!=NULL);
    c->value = 0;
}

void counter_release(counter *c)
{
    assert(c!=NULL);
    free(c);
}
```

2 Exercises

Practical Exercise 1

You should organise all of your work for this module within a dedicated directory called `ctec2901`.

1. The first step is to create the `ctec2901` directory and then move into it.

   ```
   cd
   mkdir ctec2901
   cd ctec2901
   ```

2. Now you should create two new directories for storing your practical work. The first one is called `exercises` and is where you can collect the demos, test programs, and exercise solutions. The second one is called `portfolio` and is where you should put your *portfolio* solutions. These will be required for formal assessment.

   ```
   cd
   cd ctec2901
   mkdir exercises
   mkdir portfolio
   ```

   How you organise your exercises within the `exercises` directory is completely up to you. You may wish to create further subdirectories or you may wish to place all of your exercises in the same directory. If you do this you may wish to invent a consistent exercise naming convention so that you can easily find your programs.

   However, we would strongly recommend that you do not subdivide the `portfolio` directory. There will be fewer program files stored here and it will be useful for you to have your portfolio solutions in one place so that it is easy to access them quickly when it is time for assessment.

Practical Exercise 2

Given the following code:

```
#include <stdio.h>
int main()
{
  int array[10] = {3,6,9,12,15,18,21,24,27,30};
  int *pa, temp = 0;
  pa = &array[0];
  printf("\n\n);
```

```
  *pa= temp;
  printf("temp = %10d \t\t *pa = %10d\t\t pa =%10p\n\n", temp, *pa, pa);
}
```

1. (Pen and paper exercise) Predict what the output will be.
2. Check your answer by compiling and running the program.

Practical Exercise 3

Given the declarations

```
struct info
{
  int total;
  char *str;
};

struct info s, *p = &s;
```

1. Using these declarations write a program that initialises the structure `s` with suitable values. You should employ the standard field selection operator when referring to the fields for initialisation.
2. Print out the values contained in the struct. However, when accessing the fields for printing, you should use the pointer `p` and *not* the variable `s`.
3. Referring once again to the structure defined in the previous exercise, consider the following statement sequence:

   ```
   char *q = p->str;
   ++p->total;
   ++*q++;
   (*(++q))--;

   printf("total = %i, str = %s\n", p->total, p->str);
   ```

   Incorporate this code *after* the initialisation code from the previous exercise. By looking at a C operator table that gives operator precedences and associativity predict the final output. Then run the program to see if you were right. The example shows how complicated expressions are resolved by the precedence and associativity rules and also demonstrates that such expressions can be very hard to read.[1]

[1] Certain industrial safety and reliability standards would ban some of the notation you have seen here – expressions such as $++*q++$ without parentheses, or even multiple declarations on a single line would be banned. One such standard is set by the Motor Industry Software Reliability Association (MISRA). However, the latter, which is primarily aimed at the construction of embedded systems software, would also disallow the use of unreliable functions such as *malloc*.

Practical Exercise 4

Write a program that prompts the user to enter an int value representing a maximum size for an array. Then an array of int values should be created dynamically using malloc containing sufficient space to store this maximum number of ints. Using a loop populate the array interactively by asking the user to enter each of the values. Finally output the mean and the standard deviation[2] of the numbers.

Practical Exercise 5

Write a program that uses the *Sieve of Eratosthenes* to calculate the prime numbers up to (but not including) a given limit N.[3] The sieve works like this:

1. Read in N.
2. Allocate memory for an array of N ints.
3. Populate the array entirely with ones.
4. Loop $i = 2$; $i < N$; i++: if the array at index i equals one, then loop through all the remaining indexes j that are *multiples* of i and set each corresponding array element to zero.
5. Print out the prime numbers (those whose elements are set to one).

index :	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19...
init :	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1...
$i = 2$:	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1...
$i = 3$:	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1...

etc.

Practical Exercise 6

Consider the counter program listing in section 1.4.

1. Write a program that uses the counter and tests each of its functions. Then compile and link the program with the counter library and check it all works as expected.
2. Modify the counter.c file so that an extra field is added to the counter implementation. Specifically, add a field called `direction` which is initialised to one. Then change the behaviour of the `counter_inc` function

[2]Use any standard deviation formula you like.

[3]This example appears in "Algorithms in C" (Parts 1-4), 3rd edition, by R. Sedgewick, Addison-Wesley, 1998, pp 83-84.

so that when the `limit` is reached the `direction` is set to -1. Similarly, when zero is reached `direction` is toggled back to 1 again. This way the increment function will allow all of the values in the range $0\,..\,\mathit{limit}$ to be traversed in ascending, then descending order, repeatedly. [4]

[4]Note how your main test program does not have to be re-compiled – the header file `counter.h` has not been changed. The use of the `typedef` in the header file *decoupled* the type `counter` from its implementation. This technique permits incremental development of libraries without affecting client software.