Calloc, Malloc, Pointers and Structs

Ian Anderson

10th April 2005

The aim of this worksheet is to introduce the functions `calloc()` and `malloc()` and the concepts of pointers and structs. The keyword here being introduce, it is expected that you will read up on this in your own time.

Please remember whilst it is very important that you understand the concepts in this worksheet please try and keep your code logical and legible!

1 Structs

In C a structure is a collection of related items, like a record. Note: Structs do not need to contain items of the same type.

1.1 Declaring Structs

A struct can be declared in either of the following ways (the end result for both is equivalent):

```
struct point {
    int x;
    int y
};

struct point point_a, point_b;
```

Alternatively:

```
struct {
    int x
    int y
} point_a, point_b;
```

1.2 Using Structs

Using structs is very easy, below is some sample code to illustrate this:

```
int main() {
    struct point myPoint;
    myPoint.x = 5;
```

```
    myPoint.y = 10;
}
```

1.3 Pointers to Structs

The concept of pointers to structs is discussed in the next section.

1.4 Exercises

- Write an address book style program that stores contact details using structs.
- Make sure you create a couple of functions in your code because they will be needed for the next section.

2 Pointers

Remember when you declare a variable you are actually doing 2 things. Firstly you are creating a bit of space in memory to put the value that the variable will store and secondly you are linking the address of that space to the variable. In short, variables have addresses and values. You will have already come across the concept of addresses and values when using the `scanf()` and `printf()` functions. For example:

```
1.  int myInt;
2.  printf("Please enter an integer:  ");
3.  scanf("%d", &myInt); /* Note the & meaning address of myInt */
4.  /* Now we want the value so we don't use the & */
5.  printf("You entered the number %d\n", myInt);
```

The `scanf()` function needs to know where abouts (the memory address) to put the value entered by the user. The `printf()` function just requires the value to print to the console.

Normal variables have both a value and a address however sometimes we may simply want to use the address of something and have no need to store a value. We can do this by declaring a pointer:

```
1.  int i; /* Normal integer variable called i */
2.  int *z; /* Integer pointer called z */
```

A simple example:

```
1.  int myFirstInt = 7; /* Normal integer variable */
2.  int mySecondInt; /* Normal integer variable */
3.  int *myPointer; /* Pointer */
4.  myPointer = &myFirstInt; /* Set myPointer to the address of myFirstInt */
5.  /*Set the value of mySecondInt to the value that myPointer is pointing to*/
6.  mySecondInt = *myPointer;
7.  printf("mySecondInt = %d", mySecondInt);
```

```
8.
9.  OUTPUT:
10.  mySecondInt = 7
```

At first this may not appear particularly useful. So here is another example:

```
1.   #include <stdio.h>
2.
3.   void myFunction(int *myVar);/*Prototype*/
4.
5.   void main()
6.   {
7.       int anInt = 4;
8.       printf("Before - the value of anInt is %d \n", anInt);
9.       myFunction(anInt);
10.      printf("After - the value of anInt is %d \n", anInt);
11.  }
12.
13.  void myFunction(int *myParam)/*Note the * indicating myParam is a pointer*/
14.  {
15.      *myVar += 2;
16.  }
17.
18.  OUTPUT:
19.  Before - the value of anInt is 4
20.  After - the value of anInt is 6
```

The last code segment removed the need for any additional local variables thus meaning the program will take up less memory when running. Pointers are typically used when working with large arrays.

2.1 Pointers to Pointers

Remember you can have pointers to pointers:

```
int myInt; //Int
int *myIntPtr; //Pointer to int
int **myIntPtr2Ptr; //Pointer to pointer to int
```

2.2 Pointers to Functions

This is probably one of the most confusing concepts so don't worry if it doesn't make sense the first (or even second) time you read this.

Function pointers point to the addresses of functions.

2.2.1 Define a function pointer

The following line of code creates a function pointer called `myFuncPtr` that takes an int, float and char and returns an int:

```
int (*myFuncPtr)(int, float, char) = NULL;
```

2.2.2 Assign address to function pointer

We now need to assign an address for our function pointer.

```
int myFunction(int myInt, float myFloat, char myChar){
    ...
    return anInt;
}
//Now we assign the address to our function pointer
//in one of the following ways:
myFuncPtr= myFunction;
myFuncPtr = &myFunction;
```

2.2.3 Calling a function using a function pointer

Now we can call the function by using the function pointer in one of the following ways:

```
int returnedVal = myFuncPtr(7, 2.2, 'a');
int returnedVal = (*myFuncPtr) (7, 2.2, 'a');
```

2.2.4 Passing function pointers between functions

We can also pass function pointers between functions:

```
void anotherFunction(int (*anotherFuncPtr)(int, float, char))
{
    //Call the function we have just been passed
    int returnedVal = (*pt2Func)(7, 2.2, 'a'); // call using function pointer
    ...
}

void demo()
{
    //Call anotherFunction() passing the address of myFunction
    anotherFunction(&myFunction);
}
```

Returning function pointers:

```
int (*getFuncPtr())(int, char)
{
    return &aFunction; //return pointer to function
}

void demo()
{
    //declare a function pointer
    int (*myFuncPtr)(int, char) = NULL;
    myFuncPtr=*getFuncPtr(); //get function pointer
    int returnedVal = (*myFuncPtr)(7, 'a'); //call function using the pointer
}
```

2.3 Pointers and Arrays

As you'd expect pointers and arrays go hand in hand in C. Remember arrays are arranged in consecutive memory locations. The following code segment creates an array and a pointer to the first element:

```
int myArray[100];
int myInt;
int *myPtr;
...
//Point to first element of the array
myPtr = &myArray[0];// Or simply:  myPtr = myArray;
myInt = *myPtr;//myInt = the values that myPtr points to
```

So to navigate through the array we can do things like:

```
myPtr++;
```

2.4 Pointers to Structs

Creating a pointer to a struct is just the same as for normal pointers:

```
struct point *myPoint;
```

Then to use it you use one of the following:

```
printf("MyPoint.x = %d", (*myPoint).x);
```

Or:

```
printf("MyPoint.x = %d", myPoint->x);
```

2.5 Exercises

- Modify the address book program you created in section one to use pointers to structs.
- Modify the address book program to use at least one function pointer.

3 Calloc and Malloc

The functions `calloc()`, `malloc()`, `free()` and `realloc()` are all used when working with dynamic memory, As their names suggest the functions `calloc()`, `malloc()` and `realloc()` are all used when allocating memory and the `free()` function is used to release this memory.

3.1 Malloc

Malloc is used to allocate a continuous portion of memory (memory-allocation, malloc).

3.1.1 Synopsis

```
#include <stdlib.h>
void *malloc(size_t size);
```

3.1.2 Man page

`malloc()` allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared. For `calloc()` and `malloc()`, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or `NULL` if the request fails.

3.1.3 Sample Usage

The following line attempts to grab 10 bytes and assign the starting address to the pointer myPtr:

```
int *myPtr;
myPtr = malloc(10);
```

It is perhaps more useful however to use the `sizeof()` function:

```
#include <malloc.h>
void main()
{
    struct person {
        char name[20];
        int age;
    };
    struct person *me;
    me = (struct person*)malloc(sizeof(struct person));
    ...
}
```

3.2 Calloc

The `calloc()` function like `malloc()` is used to allocate a continuous space in memory but unlike `malloc()` this memory is cleared to 0. Calling `calloc()` is therefore slightly more computationally expensive then calling `malloc()`.

3.2.1 Synopsis

```
#include <stdlib.h>
void *calloc(size_t n_elem, size_t e_size);
```

3.2.2 Man page

`calloc()` allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. For `calloc()` and `malloc()`, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or `NULL` if the request fails.

3.2.3 Sample Usage

The following code grabs a block of memory for 10 integer elements and initialises the values to 0.

```
int *myPointer;
myPointer = (int*)calloc(10, sizeof(int));
```

3.3 Realloc

The `realloc()` function is used to change the size of a previously allocated block of memory. If the new requested size is smaller than the old size then

3.3.1 Synopsis

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

3.3.2 Man page

`realloc()` changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old an new sizes; newly allocated memory will be uninitialized. If `ptr` is `NULL`, the call is equivalent to `malloc(size)`; if size is equal to zero, the call is equivalent to `free(ptr)`. Unless `ptr` is `NULL`, it must have been returned by an earlier call to `malloc()`, `calloc()` or `realloc()`. `realloc()` returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from `ptr`, or `NULL` if the request fails or if size was equal to 0. If `realloc()` fails the original block is left untouched - it is not freed or moved.

3.3.3 Sample Usage

Assume we have already executed the following `calloc()`:

```
int *myPointer;
myPointer = (int*) calloc(10, sizeof(int));
```

We can then use the following `realloc()`:

```
myPointer = (int*)realloc(myPointer, 20);
```

3.4 Free

The `free()` function as its name suggests is used to release memory that has previously been allocated using either `malloc()`, `calloc()` or `realloc()`.

3.4.1 Synopsis

```
#include <stdlib.h>
void free(void *ptr);
```

3.4.2 Man page

`free()` frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()`, `calloc()` or `realloc()`. If `ptr` is `NULL`, no operation is performed. `free()` returns no value.

3.4.3 Sample Usage

Assume we have already executed the following `calloc()`:

```
int *myPointer;
myPointer = (int*) calloc(10, sizeof(int));
```

We can then use the following `free()` command to release the memory:

```
free(myPointer);
```

3.5 Exercises

- Modify the address book program so that you now use the `malloc()`, `calloc()` and `free()` functions where appropriate.