C: Part 8
LANGUAGE OF THE 'C'

In part eight of Steve Goodwin's 'C' tutorial we take a look at memory allocation

The data we've worked with so far has all been static – i.e. of a predetermined, known size. We declare an array, structure, or array of structures within the program and use it. That's fine for the projects we've done so far but if we were writing a "real" program it is highly unlikely we could do this. A word processor couldn't know the size of every file it would have to work with; no more than a paint package could know the size of every picture it needs to manipulate. So how do we allocate memory dynamically at run-time?

Set adrift on memory bliss

The answer, quite naturally enough, is by using a function to allocate memory dynamically at run-time! That function is called *malloc*.

Listing 1, line 2 includes the appropriate header for memory allocation. This file, as we've seen several times before, includes the prototypes for several functions that are implemented inside glibc. We've used it before for the *atof* and *atoi* functions and random numbers. Now we're using it for memory management.

The *malloc* in line 8 stands for *M*emory *ALLOC*ation and requests 4Kb of memory (1,024 integer elements, each 4 bytes in size). Because the memory allocation routines don't know what type of data you want, it cannot return a pointer with the correct type. Although there could be routines called 'malloc_int' and 'malloc_short', this would not help if we created a custom type called 'Person' as we'd have to write special allocation code for 'malloc_Person' and recompile it into 'glibc' every time we wrote a new program. Instead, *malloc* uses a 'void pointer', which allows the *pData* variable to point to data of an undetermined type. Review last month's article on type casting to remind yourself about this, if necessary.

Because memory is finite, it is possible for this function to fail. We must therefore check that the pointer is valid (line 9), and gracefully handle any allocation that does not happen. The 'C' standard specifies that *malloc* must return a NULL pointer – which is numerically equal to zero – should the memory allocation fail.

```
ptr = malloc(1000000000); /* Probably can't
allocate this! */
   /* ptr is now NULL */
*ptr = 10;   /* Trying to store the number 10
at memory location 0 */
   /* This causes a segmentation fault, or core
dump */
```

The complexity of processing failed allocations is likely to grow proportionally with the size of the project. Still, you must keep thinking 'what should happen if this fails' and handle it. It is not polite for the user to be thrown out of the program (by a segmentation fault) because you didn't code it properly! If you absolutely need some memory (for an 'I've run out of memory' message, say) then allocate that memory when the program starts. That way, if the program can start successfully it can run successfully – in all situations and without problems. When loading a file, for instance, some text editors create a working

Listing 1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[])
5 {
6 int *pData;
7
8  pData = (int *)malloc (1024*sizeof(int));
9  if (pData)
10    {
11    *(pData+0) = 1; /* Write an int to the first available memory location */
12    printf("%d", *(pData+512));;/ *  Output a byte from somewhere in the middle */
13    *(pData+1023) = 1024; /* Write into the last */
14    free(pData);
15    pData = NULL;
16    }
17 return 0;
18 }
```

buffer ahead of time. If there is not enough memory for this buffer then the file is opened in read only mode. This is not ideal, perhaps, but significantly better than letting the user edit a text for an hour, only to be told 'I've run out of memory' and find they are unable to save!

Lines 11-13 read and write data from the newly created memory block. Like arrays, referencing elements outside the permitted bounds can cause segmentation faults. The data is not initialised to any specific value either, so line 12 could print junk.

Pointers, being expressions likes any other, can reference data with simple pointer arithmetic. Like so:

```
int *pData2;

   pData2 = pData+511;  /* assign a new pointer
to the 512 nd element */
   *(pData2 + 10) = 1;  /* equivalent to
*(pData + 521) = 1; */
```

Referencing the elements can be done with either pointers or square brackets:

```
pData[10] = 123; /* equivalent to *(pData + 10)
= 123; */
```

although the latter is not recommended, as it implies you're referencing an array – which you're not!

Finally, in the same box labelled "what goes up, must come down" is the memory *de*-allocation routine, *free*, at line 14. It is good practice to do this explicitly whenever you've finished using the memory, as this allows someone else to use it and can, in turn, also improve performance. However, some programmers assume that all un-*free*d memory is de-allocated automatically when you exit the program! We aren't amongst them – and we would hope you're not either!

Line 15 is a stylistic point. Because *free* does not (and cannot) clear the memory pointer, it remains non-NULL. However, the data to which it points does not remain valid, so if we re-used our validity check (line 9) it would succeed – but be in error. Therefore, in the same way as we reset file pointers previously, we set the pointer to NULL – just to be safe.

Now, before we move onto something completely different, let's briefly visit two different allocation functions, *realloc* and *calloc*.

I can see clearly now

The other main function for memory allocation is *calloc*. Rewriting the above example using *calloc* would require just one change to Listing 1:

```
8  pData = (int *)calloc(1024, sizeof(int));
```

The only difference (syntax aside) is that *calloc* will reset each newly allocated memory address from *pData* to zero for you automatically. Which is nice!

You can de-allocate the memory with the same *free* function above and, as with *malloc*, no types are passed in – only numbers.

From a personal point of view I usually use *malloc*, as opposed to *calloc*. This is because we know that clearing the memory to zeros takes time and since we will be manually filling the memory with useful data, pertinent to our program, we don't need to waste processor cycles.

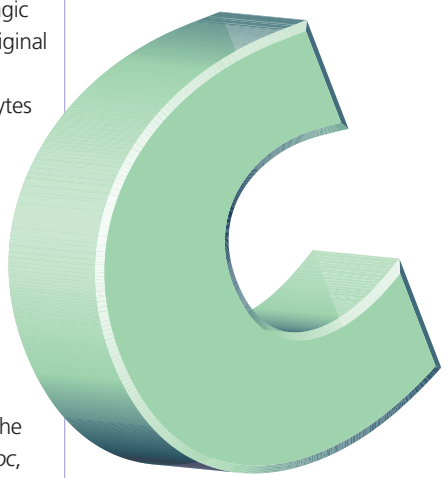
Before we move on, there is one other allocation function you should be aware of, *realloc*.

Let's go round again

The only *re-alloc*ation routine we have at our disposal is *realloc*. This is used to resize a memory block you've already allocated.

```
pNewPtr = realloc(pOldPtr, iNewMemorySize);
```

So, should you request 5Kb of space then find you need 10Kb, you can use *realloc* to make a little magic work and produce a larger block! pOldPtr is the original pointer you got from *malloc* or *calloc*, and *iNewMemorySize* should be the total number of bytes you want in memory. It could be larger or smaller than the size of the original block, but is an absolute value (i.e. not relative). (As a bonus, passing a NULL pointer for pOldPtr will cause *realloc* to function exactly like *malloc*.) The data from pOldPtr will automatically get copied into pNewPtr if the function succeeds.



There are two caveats however. One is that there might not be enough memory for the larger size block, and pNewPtr will be NULL. The other is that the pointer to the new memory block, returned by *realloc*, might differ from the old pointer you passed into it! This is the important point. If you've been holding references to your data as pointers to this pOldPtr block they will all be invalid and, as they're probably scattered throughout your code, very hard to track down and reassign with the new memory locations.

```
struct sMY_DATA *g_pData;

int GetMyDataEntry(int idx)
{
   return *(g_pData + idx).iEntry; /* Same as
g_pData[idx].iEntry */
}
```

The code above solves that problem by providing a single point of access to your allocated data – the global *g_pData* variable – and is a good thing. However, be careful of the code such as:

```
g_pData = realloc(g_pData, iNewSize + 1024); /*
Bad coding!!! */
```

Should the allocation fail g_pData will become NULL and you will lose your original pointer to the data. A core dump will ensue next time the pointer is de-referenced and you'll be unable to free the memory.

Finally, if you're looking for a function to remind you how much memory has been allocated at the memory location *g_pData* then keep looking! There is no safe, portable way of finding this information out. If you need the information, then store it along with your pointer.

Book of days

Because of the similarities with dynamic memory and static strings (both point to anonymous blocks of memory) there are manipulation functions with almost identical names. They are almost identical in operation, too! However, whereas strings know their size (because of the NUL terminator), the memory functions have to be told. Their descriptions (i.e. prototypes) also live in string.h.

The void pointer comes to the fore here, too. Since writing code to handle a memory copy of integers and a memory copy of floats is doubling up code, 'C' uses generic memory handling functions, as shown in Table 1, that take void pointers and describe the amount of data in bytes (even if the memory refers to floats). This is because the routines know nothing of the type (naturally – they're void pointers!) and a byte is the lowest common denominator.

One word describes the differences between a *memcpy* and a *memmove* – overlap. One sentence describes it – that is, if the range of memory locations in the source overlap any part of the memory locations indicated in the destination you will have to use *memmove* to prevent memory corruption. This is to allow the C library implementers to use more optimised code within the *memcpy* function.

There are also two simple functions to query memory contents shown in Table 2.

At first glance it might look like two structures could be checked for equality with the *memcmp* function, but this is not necessarily a good idea. The reason for which, we shall now explain.

Table 1 – Memory handling functions

Function example	Description
memcpy(pFrom, pTo, 100);	Copy 100 bytes of memory from pFrom to pTo. Naturally, both memory ranges should point to our own valid memory.
memmove(pFrom, pTo, 100);	Move 100 bytes of memory from pFrom to pTo. See note below.
memset(pFrom, 'A', 100);	Fill 100 bytes of memory with the ASCII character 'A'. If you wanted to write 25 floats instead, you could not use this function since it deals in bytes, and would write the data manually with a *for* loop. This function is often used with '\0' or 0 in place of 'A' to clear a portion of memory.

Hole in my shoe

Take a structure such as:

```
struct sPOSITION {
   int iXGridPos;
   int iYGridPos;
   char   iFloor;
   };
```

It could, for example, be used to store the location of a ghost in a 3D version of Pacman! We could write a simple AI routine that moved the ghosts around the maze by changing the iXGridPos, iYGridPos and iFloor elements (we're not sure how ghosts would climb stairs, but bear with us!). Then, when the player's position (also stored in an sPOSITION structure) equalled the ghost's position we could kill the player. The code would probably look like this:

```
if (memcmp(&Ghost.Position, &Player.Position,
sizeof(struct sPOSITION)==0)
   KillPlayer();
```

However, this would probably never work and the reason for this is padding.

The lunatics have overtaken the asylum

If you count the number of bytes in the above structure you should get nine: two integers at four bytes a piece and one single byte character. However, calling *sizeof* on this structure will yield a different answer – 12! This is because the compiler has automatically *padded* the structure to 12 bytes to fit in with the memory model of the host machine. This, on an Intel family IA32 system, requires that all structure sizes must be in multiples of four bytes, padding the *char* above to four bytes. It also requires that all 32 bit values (like *int*s) should start on 32 bit (i.e. four byte) boundaries. This is called structure alignment.

So? Well, the padding means there are three bytes unaccounted for in the structure and the *memcmp* function will be trying to compare them for equality. Since we haven't (and can't easily) set them up they will be uninitialised (set to junk) and will prevent our player from every getting killed (since junk is never the same twice!). Instead of a *bitwise* test, we need an *element* test, and the way to do that is to manually compare each element:

```
if (Ghost.Position.iXGridPos ==
Player.Position.iXGridPos &&
   Ghost.Position.iYGridPos ==
```

```
Player.Position.iYGridPos &&
   Ghost.Position.iFloor ==
Player.Position.iFloor)
   KillPlayer();
```

For the completists out there, we will just say there are two ways around doing this! The first is to *memcmp* the first nine bytes only! This is very ugly, doesn't port well and breaks if the *iFloor* variable becomes a *short* or if it becomes the first element in the structure.

The other way is to *memset* the whole structure to 0 at the start of the game. From any point thereafter the 'other three bytes' will be 0 and compare exactly, enabling you to use the normal *memcmp* routine! It is almost an acceptable solution but will be problematic if you forget to *memset* any sPOSITION structure you consequently try to *memcmp*!

If you're wondering why we've mentioned structure padding in a section on memory (and not structures) your answer is thus: there wasn't enough rope for you to hang yourselves in the structures article. You can't compare structures with '==', like you can integers, so to stop people trying to open the backdoor with the *memcmp*, we decided to lock it first!

Wrapped around your finger

One trick used by a lot of programmers is to write a "wrapper" for the memory allocation routines. This means instead of calling *malloc*, your program will call a wrapper (a function that sits around some code to monitor its usage, for example) like *MyMalloc* (*free* would be wrapped with *MyFree*). In this way, you can easily keep track of how much memory you are using, how often they have been called, how much has been *free*d, and how many leaks (memory you have allocated, but not *free*d) you have. The code might start off like this:

```
int g_MemAlloced=0;

void *MyMalloc(int iSize)
{
void *ptr = malloc(iSize);

   if (ptr)    g_MemAlloced += iSize;
   return ptr;
}
```

We can then find out how much memory we're using with:

```
printf("Total allocated = %d K\n",
g_MemAlloced/1024);
```

But that's just a start. We will also want to know how much we're *free*ing, and perhaps why the memory is being used. Writing the *MyFree* function then becomes problematic since we can't find out from a pointer how much memory has been allocated there. What we need is to attach our own structure to each block allocated through *malloc* to hold information such as size and reason for allocation.

The most common way of doing this is to create a structure (for the auxiliary data) and use MyMalloc to allocate enough memory for the user's data and your structure, in one block:

```
ptr = malloc(iSize + sizeof(MEMORY_BLOCK));
```

and then give the user the memory pointer N bytes after *ptr*. This can be done in two ways. Either create a byte pointer and increment it N times (where N is the size of the block), or use a MEMORY_BLOCK pointer and increment it once. Both are equivalent.

```
char *pBytePtr = (char *)ptr;
MEMORY_BLOCK *pMemBlock = (MEMORY_BLOCK *)ptr;

   pBytePtr += sizeof(MEMORY_BLOCK);
   pMemBlock++;
```

Now we can review our memory usage at any time by looking through all the pointers. The difficulty is the innocuous phrase: 'all the pointers'. We need to store each pointer as it is allocated, but where? An array? Another allocated block? Or perhaps the question is how?

Union of the snake

Linked lists are an oft-used data structure and feature on every computer course we've ever known! It is a general-purpose storage method that can grow

Table 2 – Memory querying functions

Function example	Description
iSame = memcmp(ptr1, ptr2, 10);	This compares the first 10 bytes of data stored at each pointer. If they are identical, it returns 0. If the data at ptr1 is less than that at ptr2, –1 is returned. If it is greater the function returns +1. Note the similarity to *strcmp*.
pFound = memchr(ptr, 'A', 128);	Searches 128 bytes of memory, starting at ptr, to look for the 'A' character. If no 'A' is found, a NULL pointer is returned, otherwise pFound points the first location in memory where one occurs. Like memset, this only deals with byte (i.e. character) data. Searching memory for an integer would require a custom loop.

dynamically as your data does, with very little memory overhead. It comprises of two features:

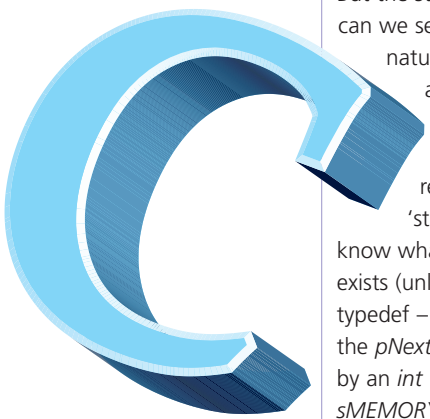
- Each element has a pointer to the next element in the list.
- A single variable that points to the first element in the list.

Declaring a list is easy, and uses the syntax we've already seen:

```
typedef struct sMEMORY_BLOCK {
   char   szReason[64];
   int iSize;

   struct sMEMORY_BLOCK   *pNext;
   } MEMORY_BLOCK;

MEMORY_BLOCK  *g_pFirstBlock;
```



As a structure, MEMORY_BLOCK can point to itself. But the structure hasn't been declared yet – so how can we set up a pointer to itself? It's this recursive nature that can appear a little counter-intuitive at first sight, but the code above should show you how we do it.

By the time the compiler has reached the first brace it knows about a 'struct sMEMORY_BLOCK'. It doesn't yet know what's inside it or how big it is, but it knows it exists (unlike MEMORY_BLOCK – our easy-to-use typedef – because it hasn't seen it yet!). This allows the *pNext* pointer to be declared – the size required by an *int* * is the same as a *void* *, or *struct sMEMORY_BLOCK* * – and incorporated into the structure. From here, we then use typedef to create a synonym for MEMORY_BLOCK to make the code look neater, although this is not essential. (Also see the Mutual Inclusion boxout).

So with this knowledge, let's return to the memory allocation example and see how MyMalloc can add elements to the *head* of the list.

```
pBlock->pNext = g_pFirstBlock;
g_pFirstBlock = pBlock;
```

When calling *MyFree*, it will search the list of allocated blocks for a matching pointer. It does this by iterating through each block with the *pNext* pointer.

```
MEMORY_BLOCK *pBlock = g_pFirstBlock;

   while(pBlock)
      {
      /* Do something with pBlock */
      pBlock = pBlock->pNext;
      }
```

Once *MyFree* has found the pointer it can modify the *pNext* pointers so the previous block to this points to our next block. Also, in the special case where we delete the element at the head of the list, we need to reassign the g_pFirstBlock.

Linked lists, as a data structure, lend themselves well to recursive searching routines and can be extended by adding a previous pointer, or can be upgraded to a tree structure by including two *pChild* pointers instead. It is fairly easy to create code that adds elements to the end of the list, or removes specific elements from the middle. A great amount has been written on these data structures and their implementation, so we won't cover it here. Suffice it to say, however, that these ideas can be utilised in a myriad of software projects, and it's well worth taking the time and effort to master it. The CD has a complete example of the *MyMalloc* and *MyFree* routines using linked lists, filling in the gaps above which have been intentionally left blank, as an exercise for you, the reader!

Mutual inclusion

Although it's not sensible (or possible) to include a structure inside its own structure (as this would create a recursive structure approximately infinity bytes long!), it is possible (and sometimes desirable) to include a pointer inside A to point to B, and vice-versa. Here's how:

```
struct sBETA;

struct sALPHA {
   struct sBETA  *pBeta;
   };

struct sBETA {
   struct sALPHA *pAlpha;
   };
```

As you can see, there's no difference in the creation or handling of the structures compared to the linked list example (we've omitted the typedef's here to prove it can be done!). You just need to add the solo 'struct sBETA' line. This says "there is a structure available called sBETA, but I don't know what it contains yet". The compiler can then use it happily in any structure where the size of sBETA doesn't need to be known (i.e. as a pointer).

GCC will happily forgo the 'struct sBETA' line, and work exactly the same without it. However, this is not guaranteed to happen across all compilers or platforms and so should be included, as above.