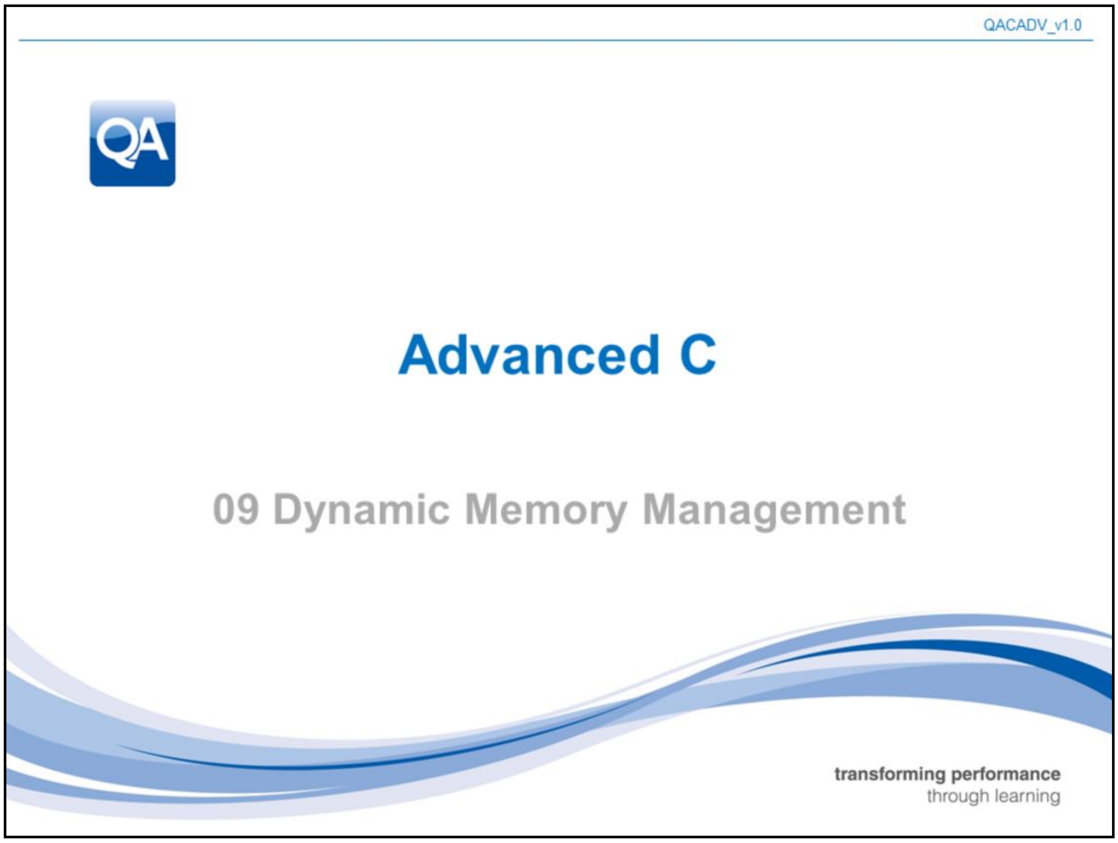
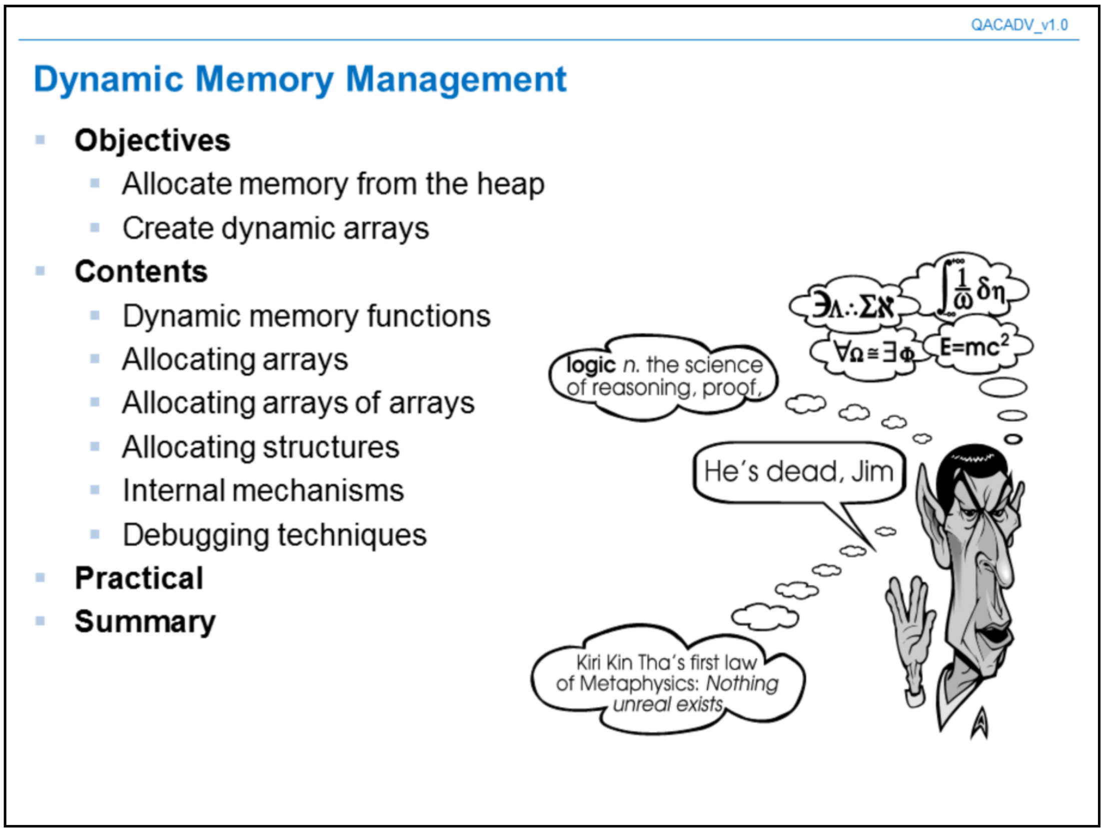


QACADV_v1.0

Advanced C

09 Dynamic Memory Management

transforming performance
through learning



QACADV_v1.0

Dynamic Memory Management

- **Objectives**
  - Allocate memory from the heap
  - Create dynamic arrays
- **Contents**
  - Dynamic memory functions
  - Allocating arrays
  - Allocating arrays of arrays
  - Allocating structures
  - Internal mechanisms
  - Debugging techniques
- **Practical**
- **Summary**

logic n. the science of reasoning, proof,
He's dead, Jim
Kiri Kin Tha's first law of Metaphysics: Nothing unreal exists
E=mc²

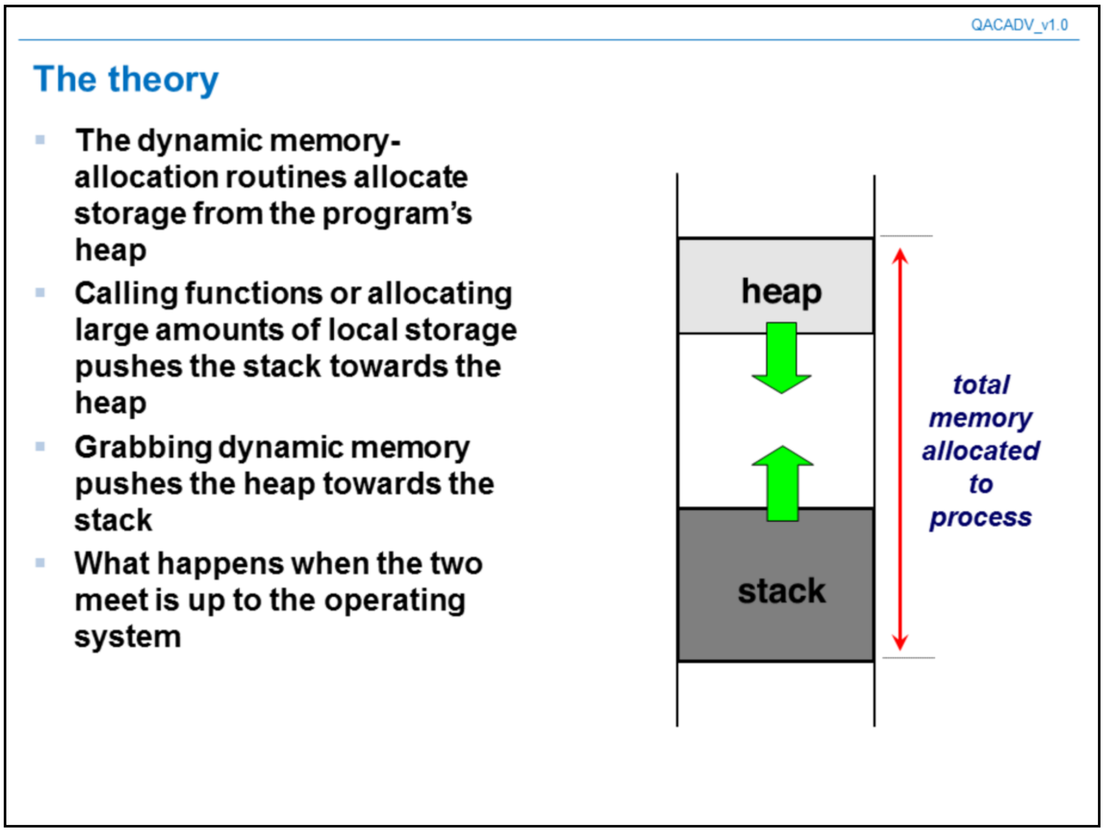
In this chapter, we shall study the topic of dynamic memory allocation. Many programs cannot determine their exact memory requirements until the program is running. In these circumstances, the program can use one of three standard functions to allocate a block of dynamic memory of the required size. These functions are `malloc`, `calloc` and `realloc`. We shall discuss each of these functions in some detail during the chapter.

When the allocated memory is no longer required, the `free` function can be used to relinquish the memory.

One of the most difficult tasks when using dynamic memory is to detect errors when things go wrong. Dynamic memory relies heavily on pointers, and there are a number of specific problems that can crop up:

- What if there is insufficient memory available?
- What if the programmer writes beyond the end of a memory block?
- What if a block of memory is never released?
- What if the programmer tries to release a block twice?

We shall spend a large part of the chapter presenting a specific mechanism to help you to debug your programs. The technique involves replacing `malloc`, `calloc`, `realloc` and `free` with equivalent debug versions that perform additional sanity checking on each allocation and deallocation. The strategy is entirely portable, and you are welcome to use it in your own programs.



QACADV_v1.0

The theory

- The dynamic memory-allocation routines allocate storage from the program's heap
- Calling functions or allocating large amounts of local storage pushes the stack towards the heap
- Grabbing dynamic memory pushes the heap towards the stack
- What happens when the two meet is up to the operating system

heap
stack
total memory allocated to process

When a program is loaded into memory by an operating system, it is allocated an initial amount of memory. Taking a simplified view, this memory is divided into the process' *stack* and *heap*.

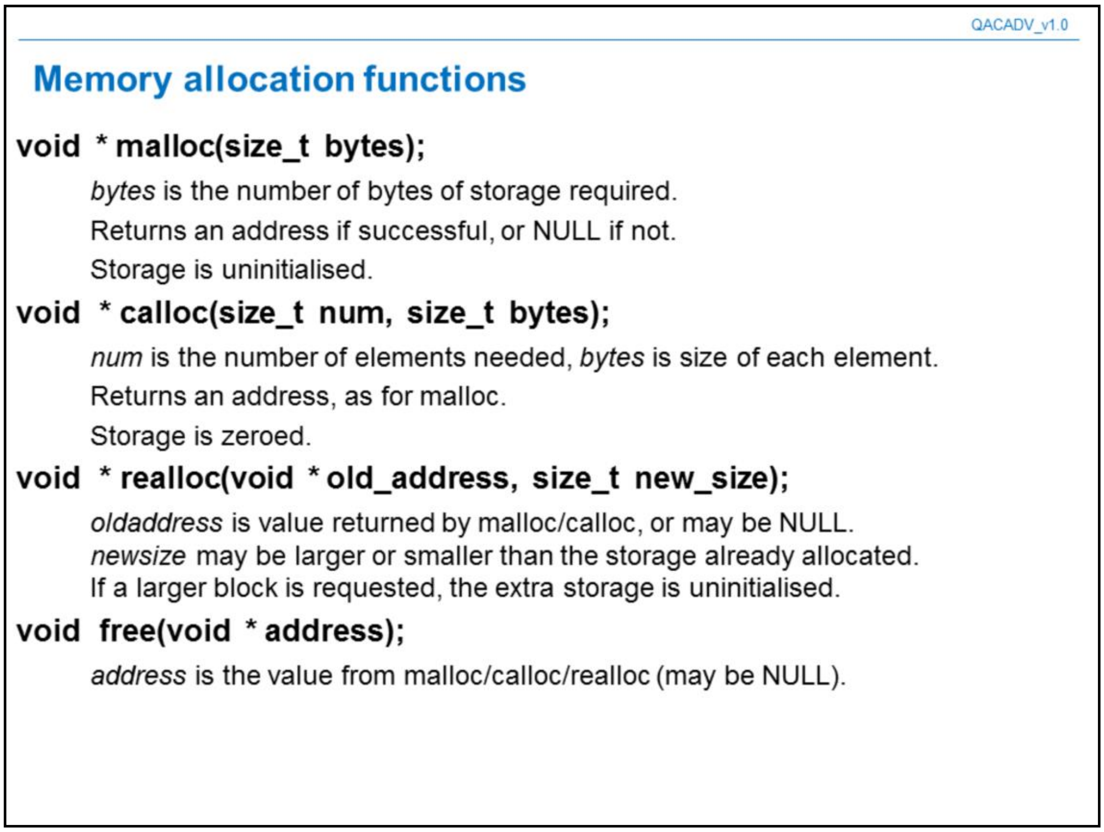
When a function is called, parameters, registers and return addresses are pushed onto the stack. The effect of this is to grow the stack and thus to use more of the memory allocated to the process. Declaring large local variables such as big arrays helps to push the stack boundary further.

The dynamic memory routines covered in the chapter allocate unused storage in the heap. As more dynamic memory is allocated, so the heap requirements for the program grow.

Clearly, if too much "function calling" is done (via recursion especially), or too much memory is allocated, the stack will be forced into the heap. What happens then is entirely dependent upon the operating system, normally the process will receive some type of stack overflow signal (EXCEPTION_FLT_STACK_CHECK on Windows or SIGSTKFLT on Linux).

In Microsoft Visual C++ the default heap and stack sizes are both 1MB, but may be altered using the /STACK and /HEAP linker switches. On Unix the functions `getrlimit()` and `setrlimit()` can get and set various process limits (e.g. `RLIMIT_STACK`), however these are usually within further limits imposed by kernel parameters.

Note that some compiler vendors (including Microsoft and GNU) provide a function called `alloca()` which allows memory to be allocated on the stack dynamically.



QACADV_v1.0

Memory allocation functions

void * malloc(size_t bytes);

bytes is the number of bytes of storage required.

Returns an address if successful, or NULL if not.

Storage is uninitialised.

void * calloc(size_t num, size_t bytes);

num is the number of elements needed, *bytes* is size of each element.

Returns an address, as for malloc.

Storage is zeroed.

void * realloc(void * old_address, size_t new_size);

oldaddress is value returned by malloc/calloc, or may be NULL.
newsize may be larger or smaller than the storage already allocated.
If a larger block is requested, the extra storage is uninitialised.

void free(void * address);

address is the value from malloc/calloc/realloc (may be NULL).

`malloc` is invoked with a single parameter: the number of bytes of memory required. An address (a `void*`) is returned, which is the address at which the allocated memory starts. The memory is not initialised. Thus, using `malloc` to allocate an array of 10 `int`s:

```
int * p = malloc(10 * sizeof(*p));
```

`calloc` is invoked with two parameters: the number of elements required, and the size of each element. Internally, `calloc` multiplies these values together and allocates a single block of the required size. The memory is cleared to the value of zero. Thus, using `calloc` to allocate an array of 10 `int`s:

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

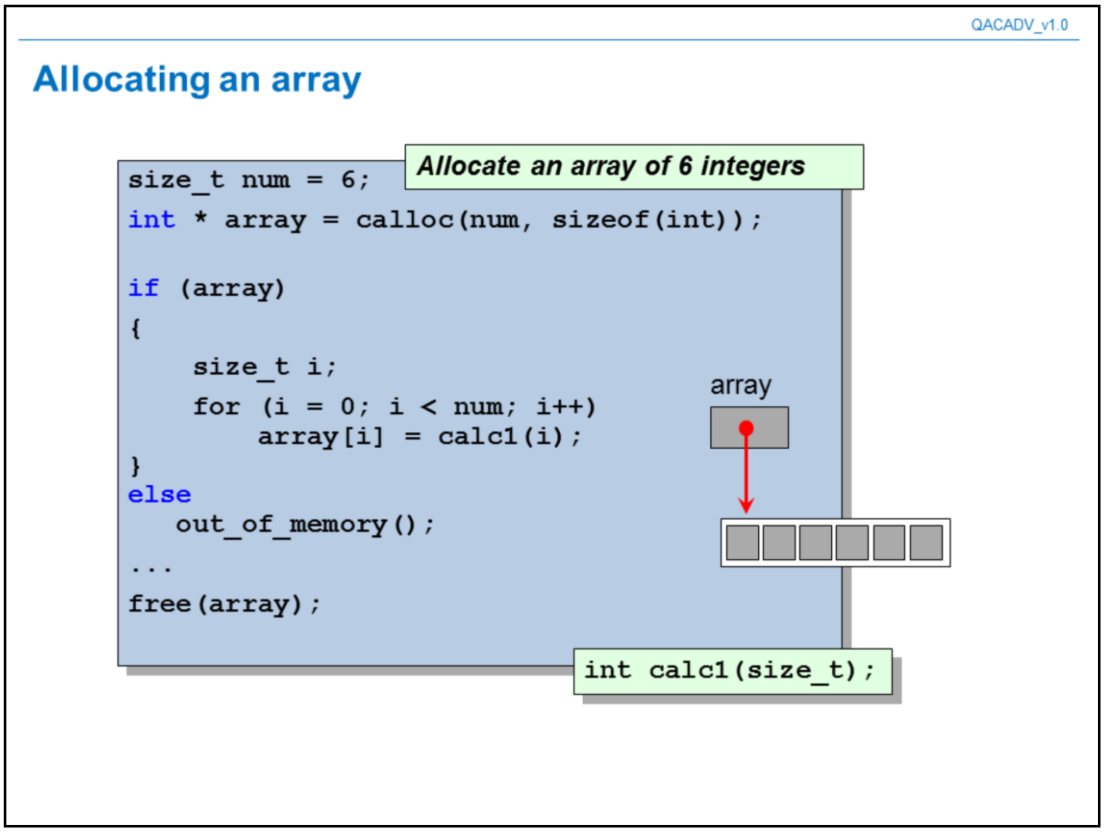
`realloc` is used to change the size of a previously allocated block of memory. The address previously returned from `malloc` or `calloc` is supplied as the first parameter. The second parameter is the new size. Thus, using `malloc` first to allocate an array of 5 integers, then using `realloc` to extend it to an array of 10 integers:

```
int * p = malloc(5 * sizeof(*p));
int * q = realloc(p, 10 * sizeof(*p));
```

`free` deallocates the storage at a specified address. The address *must* have been previously returned by `malloc/calloc/realloc`. To release the storage allocated to `q` above:

```
free(q);
```

On most implementations `stdlib.h` must be `#include`'ed for these functions, and some also require `malloc.h`.



Allocating an array

```
size_t num = 6;
int * array = calloc(num, sizeof(int));

if (array)
{
    size_t i;
    for (i = 0; i < num; i++)
        array[i] = calc1(i);
}
else
    out_of_memory();
...
free(array);
```

The program shown above allocates storage for an array of num integers. The value of num would most probably be supplied at run time:

```
    printf("How many integers to allocate ? ");
    scanf("%d", &num);
```

No other code would need to be changed beyond this. To allocate an array of 500 long doubles, for instance, only two lines would need to be altered:

```
    long double * array = malloc(num * sizeof(*array));
    if (!array) ...
```

The following code might look a little dubious but works fine:

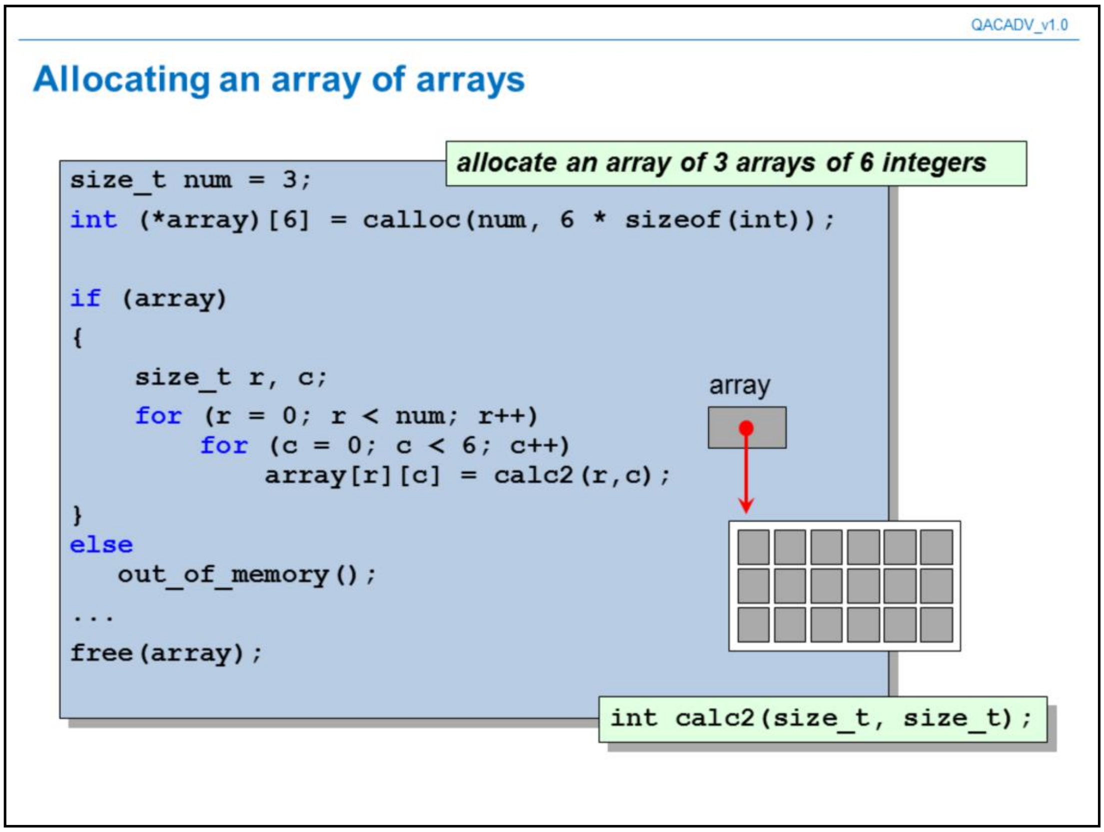
```
    array[i] = i;
```

The [ ] notation is converted by the compiler into pointer and offset notation as follows, which is clearly valid:

```
    *(array + i) = i;
```

The function **malloc** returns a void pointer. In C this can be implicitly cast to a non-void pointer (in this case a long double pointer). In C++ the implicit cast will not compile: the type checking is much stricter in C++. So if portability to C++ is an issue you should write

```
    long double * array;
    array = (long double*)malloc(num * sizeof(*array));
```



QACADV_v1.0

Allocating an array of arrays

allocate an array of 3 arrays of 6 integers

```
size_t num = 3;
int (*array)[6] = calloc(num, 6 * sizeof(int));

if (array)
{
     size_t r, c;
     for (r = 0; r < num; r++)
          for (c = 0; c < 6; c++)
               array[r][c] = calc2(r,c);
}
else
    out_of_memory();
...
free(array);
```

```
int calc2(size_t, size_t);
```

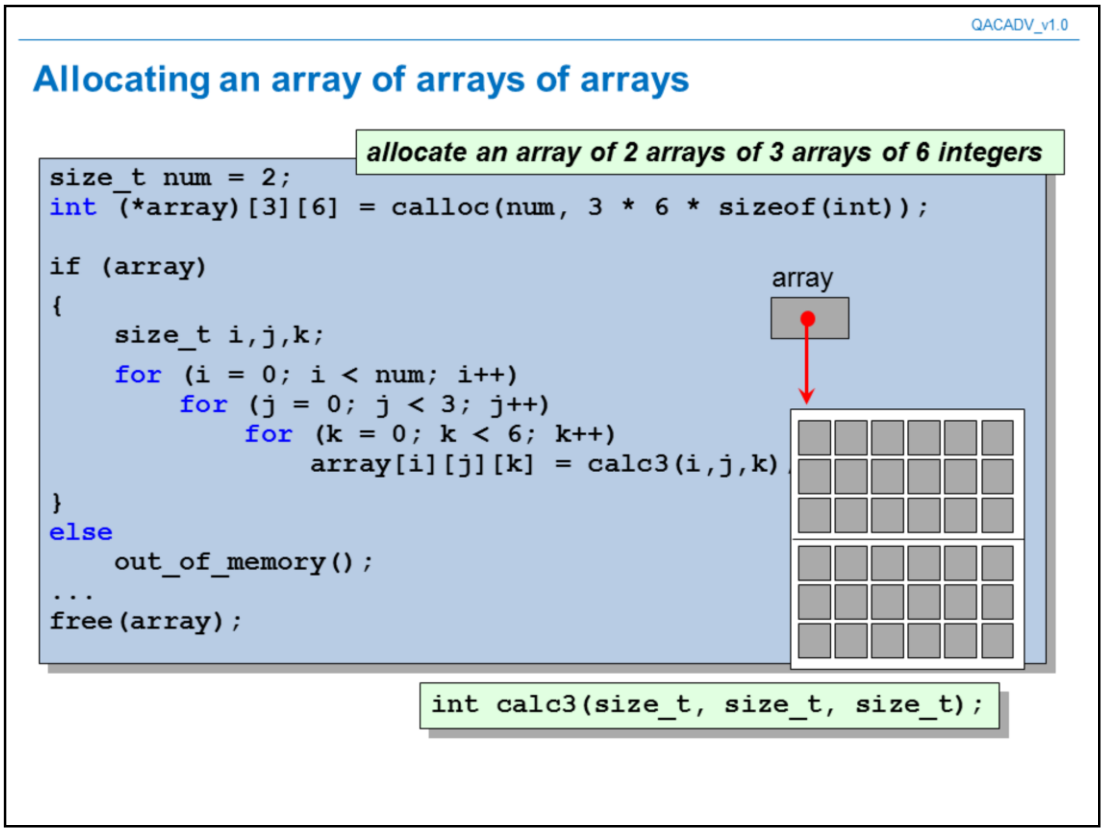
Allocating an array of arrays is an extension of this notation. Remember, the compiler always allocates arrays in contiguous storage. The example above allocates an array of 3 arrays of 6 integers. The return address is cast to a pointer to an array of 6 integers. Notice the second parameter to `calloc` is `6 * sizeof(int)` - each of the elements in the new arrays is an array of 6 ints.

Here, two sets of `[]` brackets are used to initialise the array. As seen previously, this notation may be replaced by "pointer to array notation", which is quite suitable in the example, since the variable "array" is indeed a pointer.

To allocate an array of arrays of `float`s, for example, the trivial changes examined on the previous page would be required.

Here is an alternative. The two dimensions of the array are specified in only one place, and the type contained in the array is also specified in only one place.

```
#define ASIZE(x) (sizeof(x) / sizeof((x)[0]))
typedef int row[6];
size_t num = 3;
row * array = (row*)calloc(num, sizeof(*array));
if (array) {
     size_t r,c;
     for (r = 0; r < num; ++r)
          for (c = 0; c < ASIZE(array[0]); ++c)
               array[r][c] = calc2(r,c);
}
```



QACADV_v1.0

Allocating an array of arrays of arrays

allocate an array of 2 arrays of 3 arrays of 6 integers

```
size_t num = 2;
int (*array)[3][6] = calloc(num, 3 * 6 * sizeof(int));

if (array)
{
    size_t i,j,k;
    for (i = 0; i < num; i++)
        for (j = 0; j < 3; j++)
            for (k = 0; k < 6; k++)
                array[i][j][k] = calc3(i,j,k)
}
else
    out_of_memory();
...
free(array);
```

```
int calc3(size_t, size_t, size_t);
```

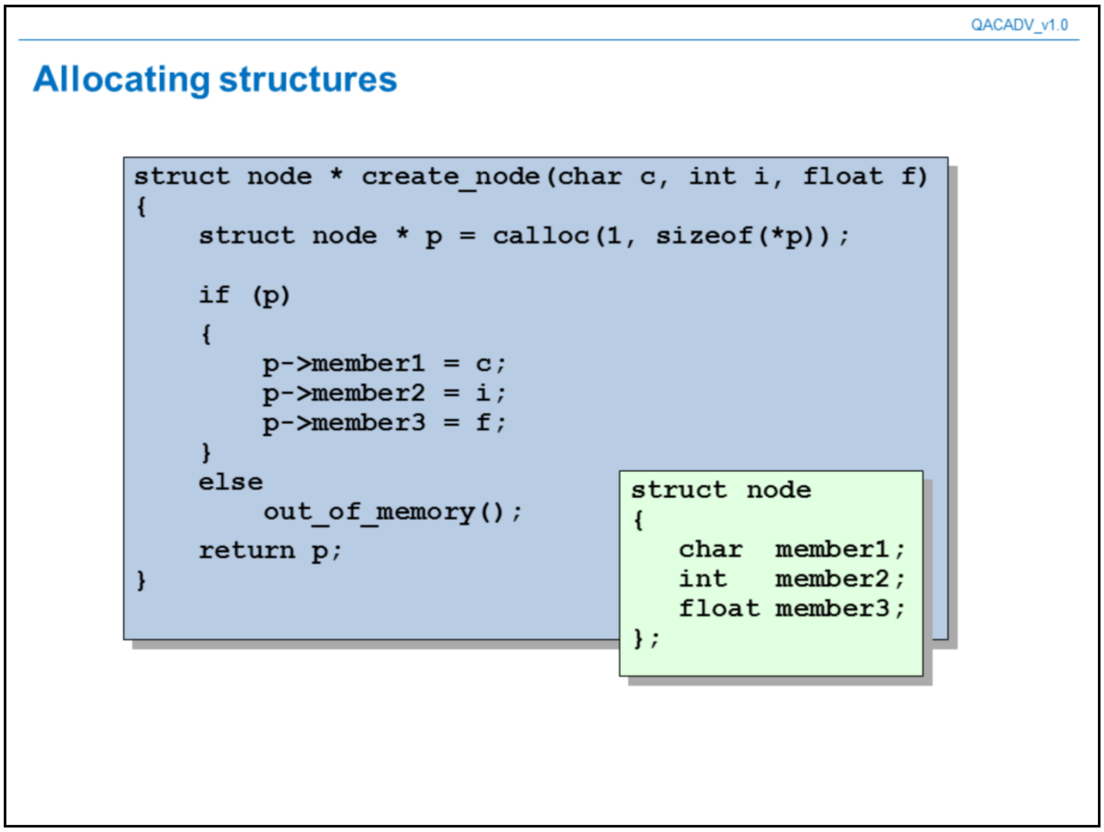
The example shown above allocates an array of 2 arrays of 3 arrays of 6 integers. If the memory had been allocated as a fixed-size variable, the declaration would have read as follows:

```
    int  array[2][3][6];
```

The two elements of `array` are 3 * 6 integers large. The code could easily be extended to allocate an array of arrays of arrays of arrays, and so on.

Here is an alternative to the above. Notice that the three dimensions of the array are specified in only one place. Notice also that the type contained in the array is also specified in only one place.

```
#define ASIZE(x)  (sizeof(x) / sizeof((x)[0]))
typedef int row[6];
typedef row matrix[3];
size_t num = 2;
matrix * array = (matrix*)calloc(num, sizeof(*array));
if (array) {
    size_t i,j,k;
    for (i = 0; i < num; ++i)
        for (j = 0; j < ASIZE(array[0]); ++j)
            for (k = 0; k < ASIZE(array[0][0]); k++)
                array[i][j][k] = calc3(i,j,k);
}
```



QACADV_v1.0

Allocating structures

```
struct node * create_node(char c, int i, float f)
{
    struct node * p = calloc(1, sizeof(*p));

    if (p)
    {
        p->member1 = c;
        p->member2 = i;
        p->member3 = f;
    }
    else
        out_of_memory();
    return p;
}
```

```
struct node
{
   char  member1;
   int   member2;
   float member3;
};
```

Since the compiler allocates structures from a single piece of memory, structures are also suitable objects that can be created dynamically.

In the example shown above, parameters are passed into the function in order to initialise the new structure. In a sense, the structure is already initialised, since `calloc` clears the allocated memory to zeros. *Remember, this would not be the case if* `malloc` *were used.* The block of memory pointed to by `p` would be random and the data members will have nonsensical values.

The issue of random values appearing in pointer members within dynamically-allocated structures is a problem. When trying to construct a linked list, it will look as though a newly allocated node points somewhere. Linking such a node into a chain can only cause disaster.

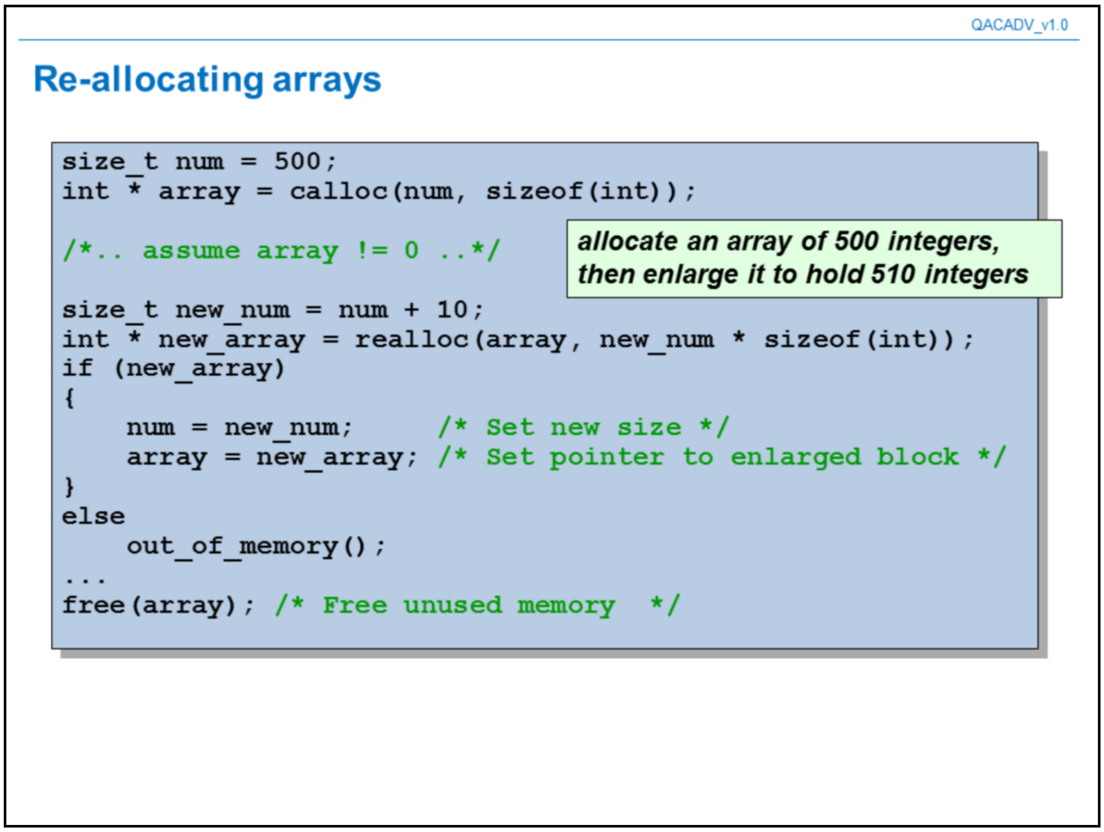
To allocate an array of 10 `Node` structures, the above code would read:

```
p = calloc(10, sizeof(*p));
if (!p) ...
```

in much the same way as the examples previously examined.

The return type of `calloc` is `void*`. In the slide this is implicitly converted to a `struct node *` in the initialisation of p. This implicit conversion will generate a compiler error when compiled in C++. If C++ compatibility is an issue you should make the explicitly:

```
p = (struct node *)calloc(10, sizeof(*p));
```



QACADV_v1.0

Re-allocating arrays

```
size_t num = 500;
int * array = calloc(num, sizeof(int));

/*.. assume array != 0 ..*/

size_t new_num = num + 10;
int * new_array = realloc(array, new_num * sizeof(int));
if (new_array)
{
    num = new_num;      /* Set new size */
    array = new_array; /* Set pointer to enlarged block */
}
else
    out_of_memory();
...
free(array); /* Free unused memory  */
```

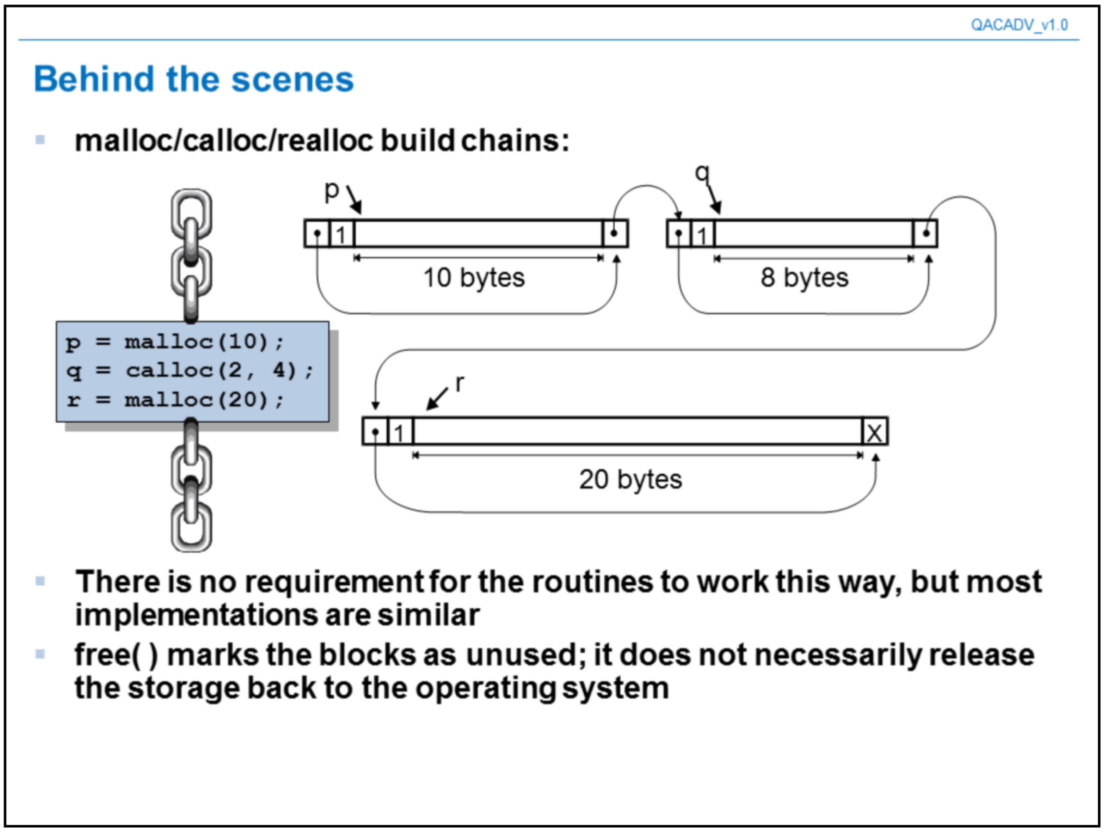
allocate an array of 500 integers, then enlarge it to hold 510 integers

The realloc function allows the size of a dynamically-allocated object to be changed. The allocated object may be made smaller or larger.

As discussed before, the "original" pointer (the one handed back from calloc/malloc) is passed into realloc as the first parameter. The second parameter is the new size the object should have, providing there is enough memory. Note that realloc is, unfortunately, more like malloc than calloc; rather than having the comparative safety of passing two parameters, it is left to the programmer to multiply by the size of the base type:

```
    (size + 10) * sizeof(int)
```

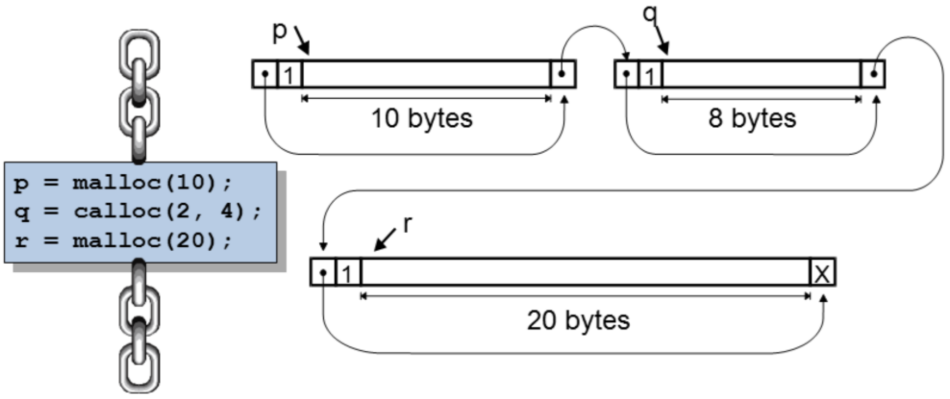
Just as with the malloc and calloc functions, it is possible for realloc to fail, and so the NULL return value must be checked for. For this reason, the address returned by realloc is assigned to a different pointer variable new_array until we know for sure that the re-allocation has been successful. This is because if realloc fails, the original address is still valid and may still be used. If realloc succeeds, the value in new_array is assigned back to array.



QACADV_v1.0

Behind the scenes

- **malloc/calloc/realloc build chains:**



```
p = malloc(10);
q = calloc(2, 4);
r = malloc(20);
```

- **There is no requirement for the routines to work this way, but most implementations are similar**
- **free( ) marks the blocks as unused; it does not necessarily release the storage back to the operating system**

The dynamic memory allocation routines build chains within the heap. For example, when allocating 10 bytes of memory, there is an overhead of a few extra bytes of control information located just before the actual block of memory that the user will access. The control information may not be accessed unless the pointer is indexed negatively (e.g. `p[-2]`).

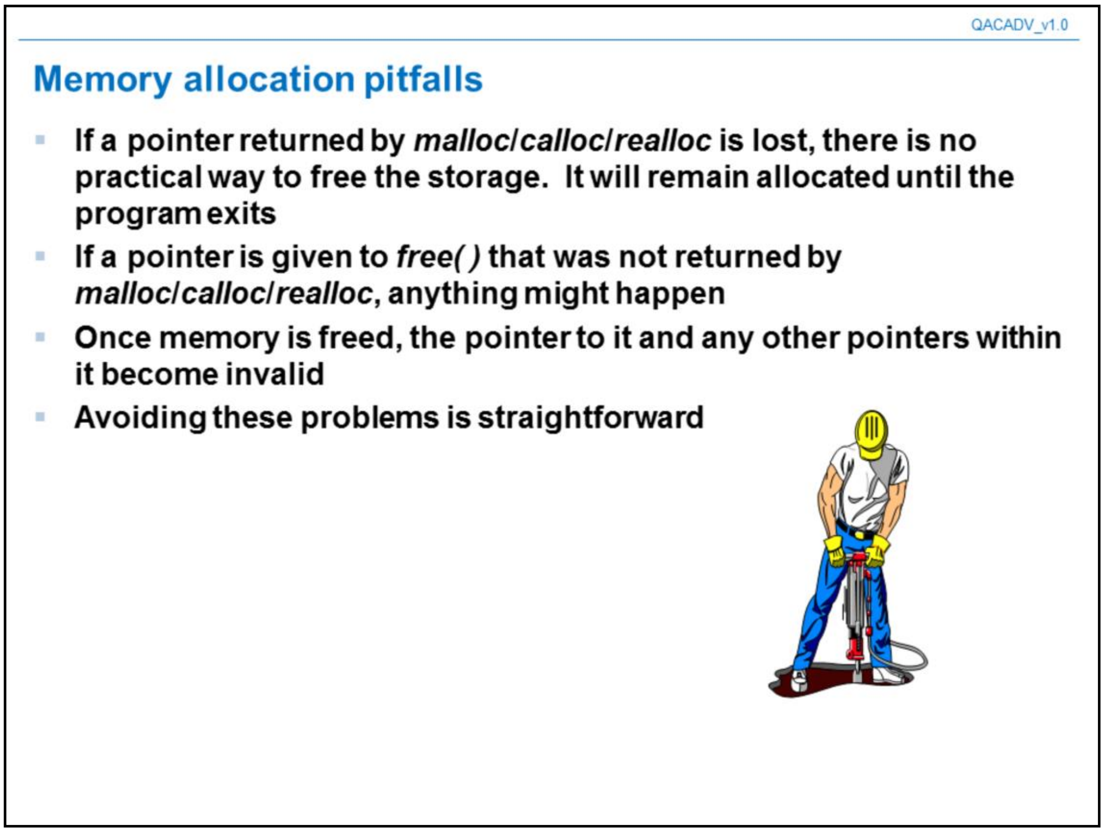
The first piece of information in the control block is the address of the end of the current chunk of memory. With this information, the memory allocation routines can easily establish how much memory was allocated. The second piece of control information is a flag to indicate whether or not the block of memory is currently in use. The `free` routine merely unsets this flag. The next `malloc/calloc/realloc` will scan the chain for any allocated block which is not in use. If this block is too small the routine will continue on down the chain until a suitable chunk is found.

The following scenario is possible:

```
free(r);
s = malloc(8);
t = realloc(s, 15);
```

`free()` would mark the block "pointed to" by `r` as unused. When `malloc` is called, it could find the free block of 20 bytes and reallocate it to `s`. `realloc` would then be able to extend the memory allocated to `s` into 15 bytes (since a total of 20 bytes is available) before assigning to `t`.

There is no requirement within the ISO standard for these routines to work in this way. The above mechanism is popular because it is discussed in Kernighan and Ritchies' book (pp 185-189 of the 2nd edition).



QACADV_v1.0

Memory allocation pitfalls

- **If a pointer returned by *malloc/calloc/realloc* is lost, there is no practical way to free the storage. It will remain allocated until the program exits**
- **If a pointer is given to *free( )* that was not returned by *malloc/calloc/realloc*, anything might happen**
- **Once memory is freed, the pointer to it and any other pointers within it become invalid**
- **Avoiding these problems is straightforward**

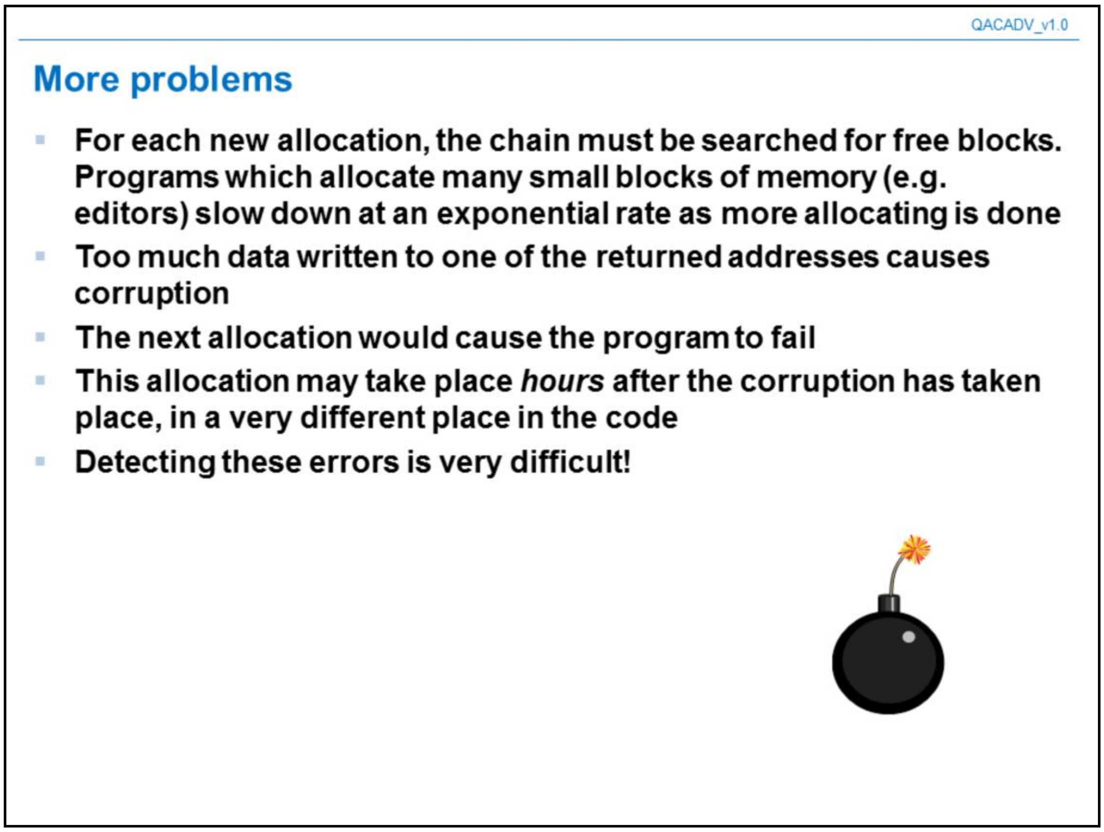
Some care is required when using dynamic memory. A pointer to a piece of dynamic memory must be carefully preserved. The only way to release the storage is to call `free` with the pointer. The following code fragments are not guaranteed to work:

```
int * p;
int   i;
p = malloc(200 * sizeof(*p)); // Allocate 200 ints
for (i = 0; i < 200; i++)     // Initialise
   *p++  =  i;
free(p); // p no longer points to start of storage
```

or:

```
int * p;
int * q;
p = malloc(300 * sizeof(*p)); // Allocate 300 ints
q  = p + 34;            // q points to element [34]
free(p);                // Free storage
*q = 99;                // Write to where q points!
```

In practice these problems are easy to avoid. Assign the address of the dynamically-allocated storage to one pointer. Do not try to duplicate it to another pointer variable. Avoid altering the address, for example, use `[]` notation rather than modifying the pointer itself (remember that the compiler converts `[]` notation into pointer and offset notation internally - the pointer remains fixed and the offset varies).



QACADV_v1.0

More problems

- For each new allocation, the chain must be searched for free blocks. Programs which allocate many small blocks of memory (e.g. editors) slow down at an exponential rate as more allocating is done
- Too much data written to one of the returned addresses causes corruption
- The next allocation would cause the program to fail
- This allocation may take place *hours* after the corruption has taken place, in a very different place in the code
- Detecting these errors is very difficult!

As we have seen, `malloc`, `calloc` and `realloc` build chains. A flag within the control information block keeps track of whether the currently allocated block is in use. This makes life easy for the `free` routine. It is a little more difficult for the allocation routines, which have to walk the chain in search of an unused block, and then see whether the block is large enough to satisfy the request.

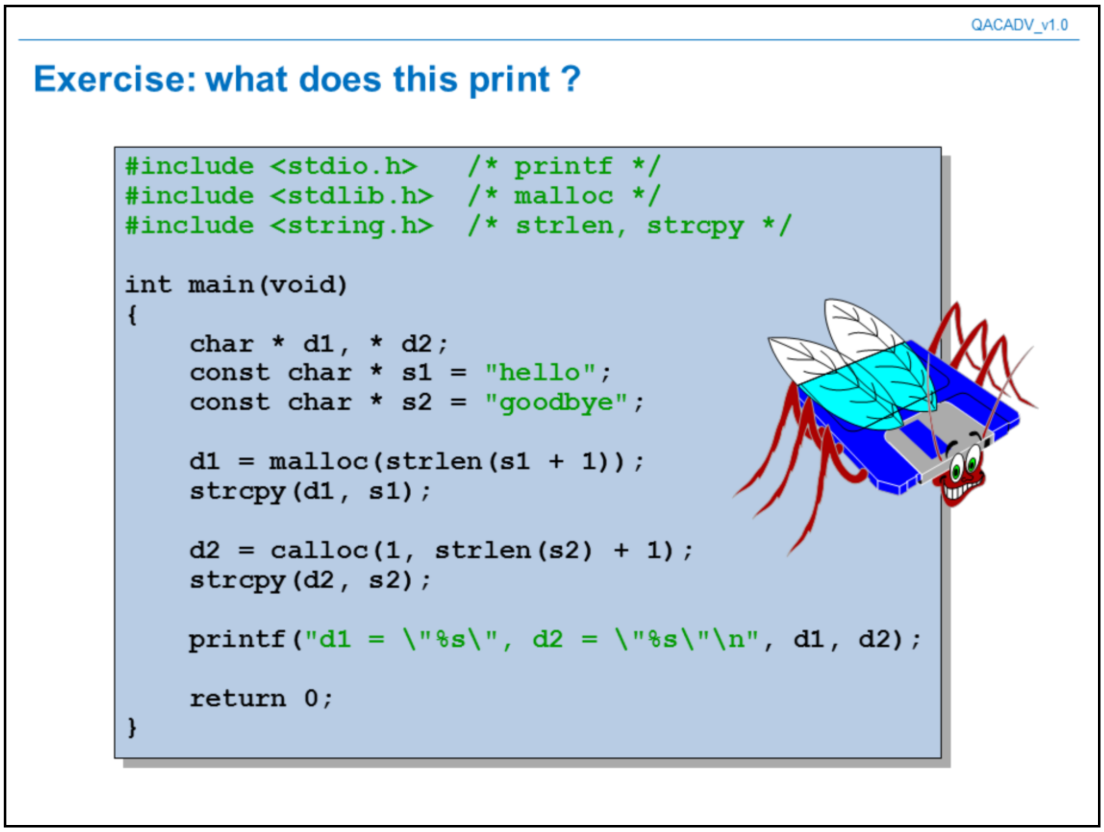
For applications which allocate all of their dynamic memory on startup, this can be bad news. The application allocates more and more memory, becoming progressively slower with each allocation. Some vendors provide different versions of `malloc`, `calloc`, `realloc` and `free`, the idea being that the programmer is left to choose the version that gives the best performance.

A more serious problem exists as shown in the following example:

```
const char * greeting = "hi there";
char * copy;
copy = malloc(strlen(greeting));
strcpy(copy, greeting);
```

The memory allocated for the copy of the string is one byte too short. The length of `greeting` is 9 bytes, but `strlen` returns 8. The call to `strcpy` copies all 9 bytes of `greeting` into the 8 bytes allocated. This corrupts the pointer which is stored for chain traversal.

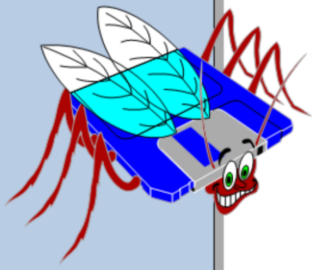
The application will not fail immediately. It is only when the next chunk of memory is allocated dynamically that problems occur. The routines walk the chain, find the now corrupted pointer, follow it and wander off. *If the program does not allocate any more dynamic memory the problem will go undetected.*



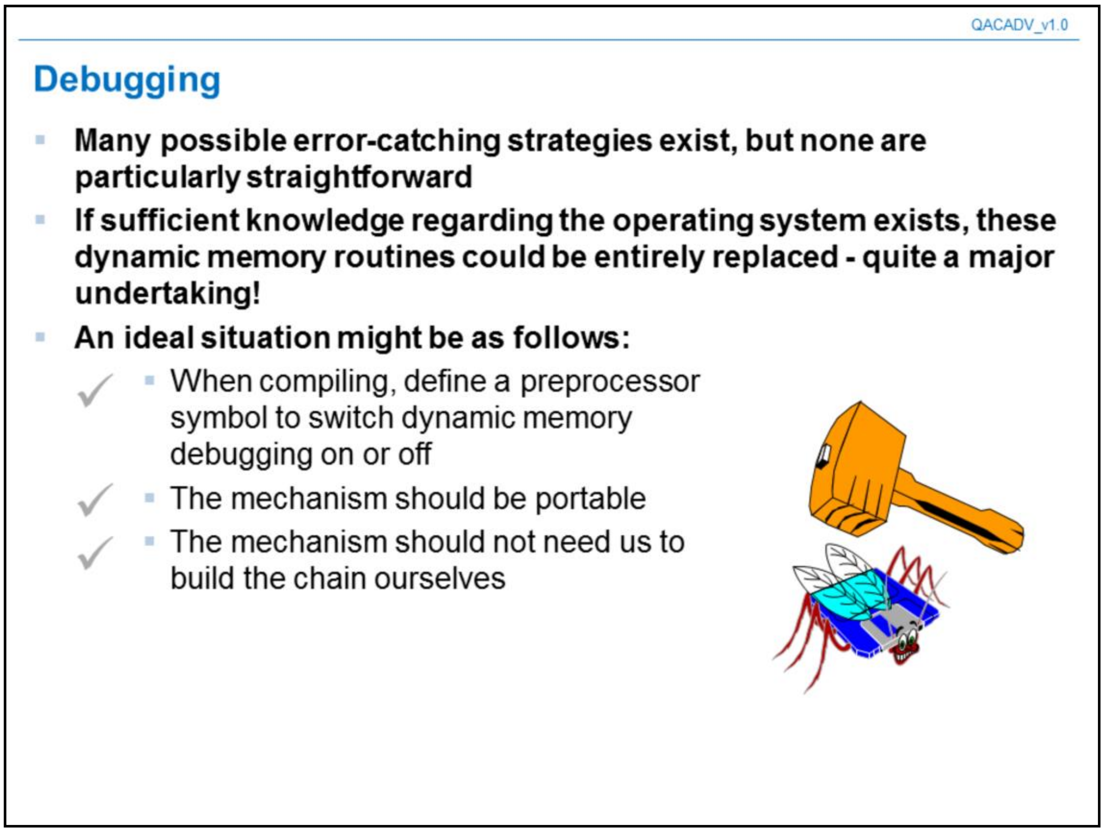
QACADV_v1.0

Exercise: what does this print ?

```c
#include <stdio.h>   /* printf */
#include <stdlib.h>  /* malloc */
#include <string.h>  /* strlen, strcpy */

int main(void)
{
    char * d1, * d2;
    const char * s1 = "hello";
    const char * s2 = "goodbye";

    d1 = malloc(strlen(s1 + 1));
    strcpy(d1, s1);

    d2 = calloc(1, strlen(s2) + 1);
    strcpy(d2, s2);

    printf("d1 = \"%s\", d2 = \"%s\"\n", d1, d2);

    return 0;
}
```



Consider the program shown above and how much memory should be requested to hold both of the strings.



QACADV_v1.0

Debugging

- **Many possible error-catching strategies exist, but none are particularly straightforward**
- **If sufficient knowledge regarding the operating system exists, these dynamic memory routines could be entirely replaced - quite a major undertaking!**
- **An ideal situation might be as follows:**
  - ✓ When compiling, define a preprocessor symbol to switch dynamic memory debugging on or off
  - ✓ The mechanism should be portable
  - ✓ The mechanism should not need us to build the chain ourselves

A large number of very different strategies exist for finding the problems outlined.

The least helpful is to avoid mistakes in the first place (the "avoid debugging, get it right first time" approach). More realistically however, checking the validity of the pointer information in the chain would involve much work. Replacing `malloc`, `calloc`, `realloc` and `free` entirely would be one approach, but would involve a *very* large amount of work. It would probably be non-portable too, requiring re-coding for each operating system encountered.

With Microsoft Visual C++, a special error checking Debug heap is used (see MSDN "Debug heap").

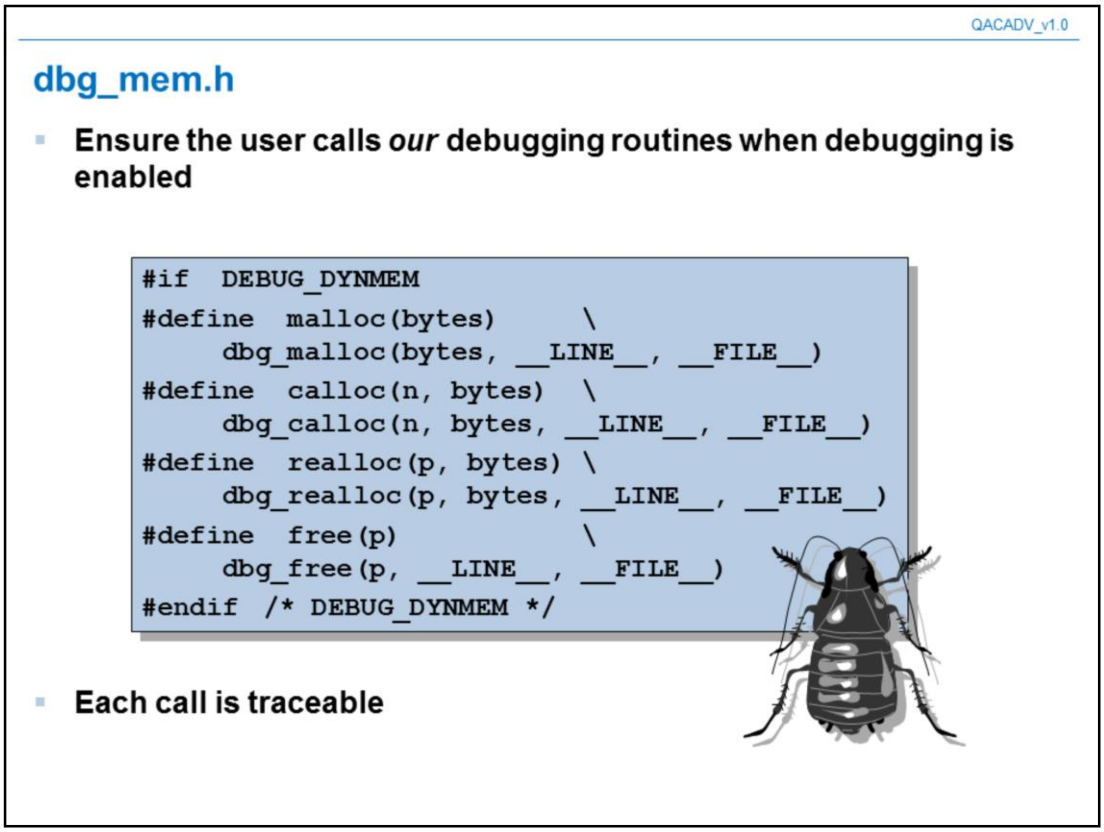
The GNU libc (2.x) enables simple error checking driven by the environment variable `MALLOC_CHECK_` (see `man malloc`).

The native Sun C compiler has the functions `watchmalloc(), cfree(), memalign()`, and `valloc()` that are worth investigation (see the 'man' pages). SUN also has an environment variable to aid heap debugging: MALLOC_DEBUG. In debug, Sun sets free'ed memory to `0xdeadbeef`.

None these aids are portable. There are also "third-party" products available, like Purify and ElectricFence.

Ideally, all that would be needed to switch on debugging in an application would be a simple symbol definition:

```
            #define   DEBUG_DYNMEM   1
```



QACADV_v1.0

dbg_mem.h

- Ensure the user calls *our* debugging routines when debugging is enabled

```
#if  DEBUG_DYNMEM
#define  malloc(bytes)      \
     dbg_malloc(bytes, __LINE__, __FILE__)
#define  calloc(n, bytes)   \
     dbg_calloc(n, bytes, __LINE__, __FILE__)
#define  realloc(p, bytes)  \
     dbg_realloc(p, bytes, __LINE__, __FILE__)
#define  free(p)            \
     dbg_free(p, __LINE__, __FILE__)
#endif  /* DEBUG_DYNMEM */
```

- Each call is traceable

The header file shown above, together with the `#define` of `DEBUG_DYNMEM` would ensure that the calls to the dynamic memory routines in the following code would actually result in the debug versions of the functions being invoked:

```
int       i;
double   *p, *q;
p = malloc(20 * sizeof(*p));
for (i = 0; i <= 20; i++)    /* Bug! Should be i < 20 */
   p[i] = i;
q = realloc(p, 40 * sizeof(*p));
```

The allocation calls would be modified by the preprocessor to become the following:

```
p = dbg_malloc (   20 * sizeof(*p), 75, "strange.c");
q = dbg_realloc(p, 40 * sizeof(*p), 78, "strange.c");
```

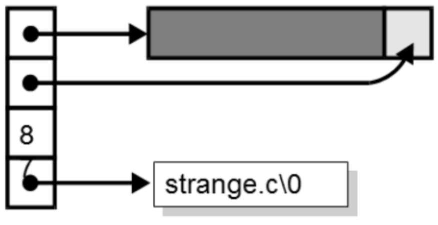
`dbg_realloc` could find that the chain has been corrupted (due to the fact that the `for` loop has one too many iterations - the test should have read `i < 20`). `dbg_realloc` could then produce the line number and filename where the memory was allocated and which had since become corrupted (that is, line 75 and filename "strange.c").

In this example, the problem would be particularly easy to spot from the debugging message, since there are only two lines of code between the two calls. In reality, even with this information the problem might still be difficult to find.

Sanity checking

- **The following structure could be used to maintain debugging information:**

```
typedef struct mem_control
{
   void *          user_address;
    char *          end;
   int             line_number;
   const char *  file_name;
} MCB;
```



The `MCB` structure shown above holds the additional debugging information passed to `dbg_malloc`, `dbg_calloc` and `dbg_realloc`. The members of the structure are as follows:

`user_address`: the address of the memory block requested by the user via `malloc`, `calloc` or `realloc`. Will be one byte larger than the user requested.

`end`: the address of the check byte(s) beyond the end of the storage block. The check byte(s) will be checked during allocation and deallocation to ensure that the heap hasn't been corrupted.

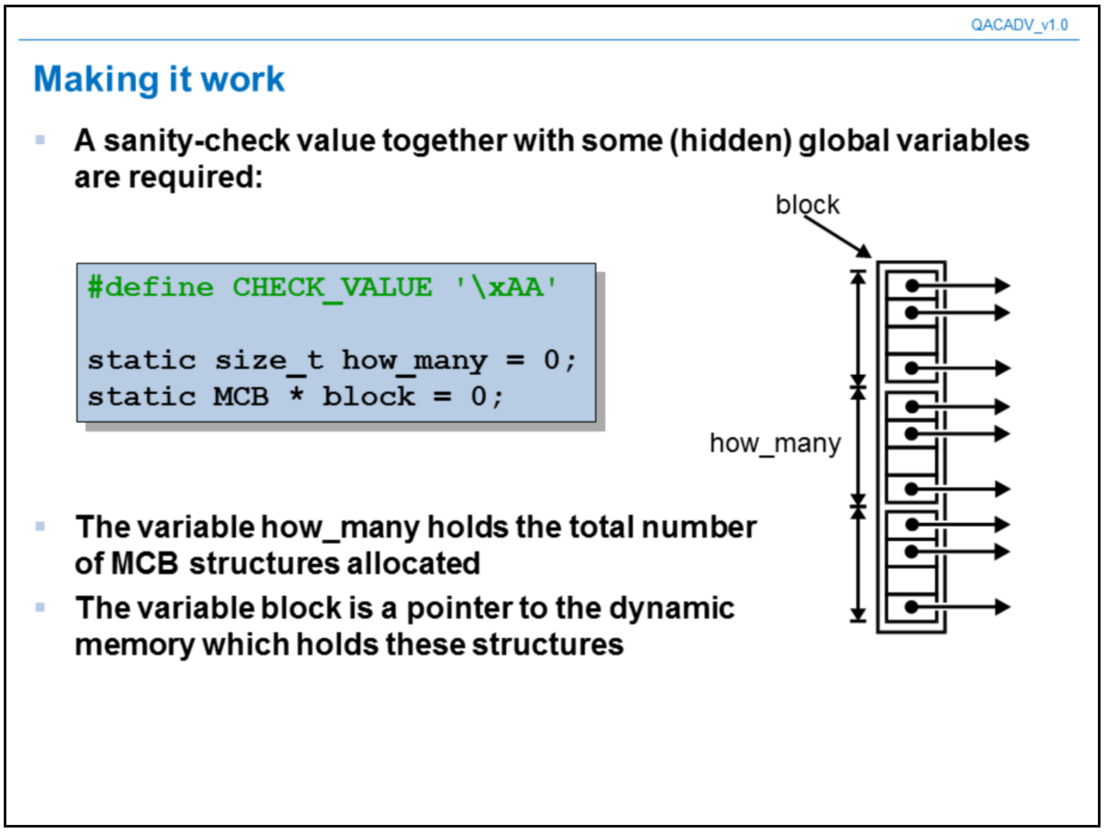
`line_number`: initialised directly from the `__LINE__` preprocessor symbol, and indicates the line in the program at which the allocation took place.

`file_name`: this string is initialised from the `__FILE__` preprocessor symbol and indicates the file in which the allocation took place.

If the user calls `malloc` or `calloc` for 10 bytes of data, the debugging routines request 11 bytes. A 1-byte *check value* is added after the block the user requested. On the next `malloc`, `calloc` or `realloc`, this check byte can be examined to see if it is still intact. If not, the routine can report the line number and filename of the code which allocated the memory.

One issue we have to tackle is *where* the memory comes from to contain each `MCB` structure. One structure will be needed per `malloc` or `calloc` call. The solution is to store the `MCB` structure in dynamic memory too. Note that `realloc` will not require a new `MCB` structure - the existing `MCB` can be reused (make the pointers point somewhere new and copy the check bytes across).

It is tempting to place check information *before* as well as *after* the data. This is not a good idea. Some machines require `doubles` to start on 2-word boundaries. Dynamic allocation routines have to ensure that *every* address they return is 2-word aligned. Putting in a 2-byte check could push the address onto a 6-byte boundary which would give rise to very strange errors.

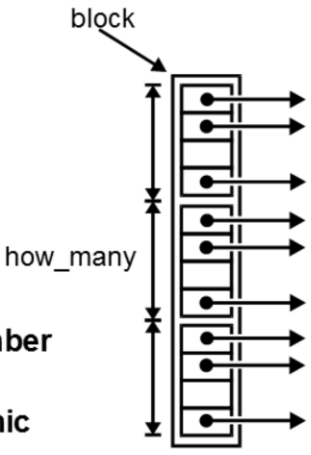


QACADV_v1.0

Making it work

- A sanity-check value together with some (hidden) global variables are required:

```
#define CHECK_VALUE '\xAA'

static size_t how_many = 0;
static MCB * block = 0;
```



- The variable how_many holds the total number of MCB structures allocated
- The variable block is a pointer to the dynamic memory which holds these structures

A sanity-check value is required which can be copied into the check byte following the region of memory the user has requested. The value chosen, 170, will fit into one byte and has the following bit pattern:

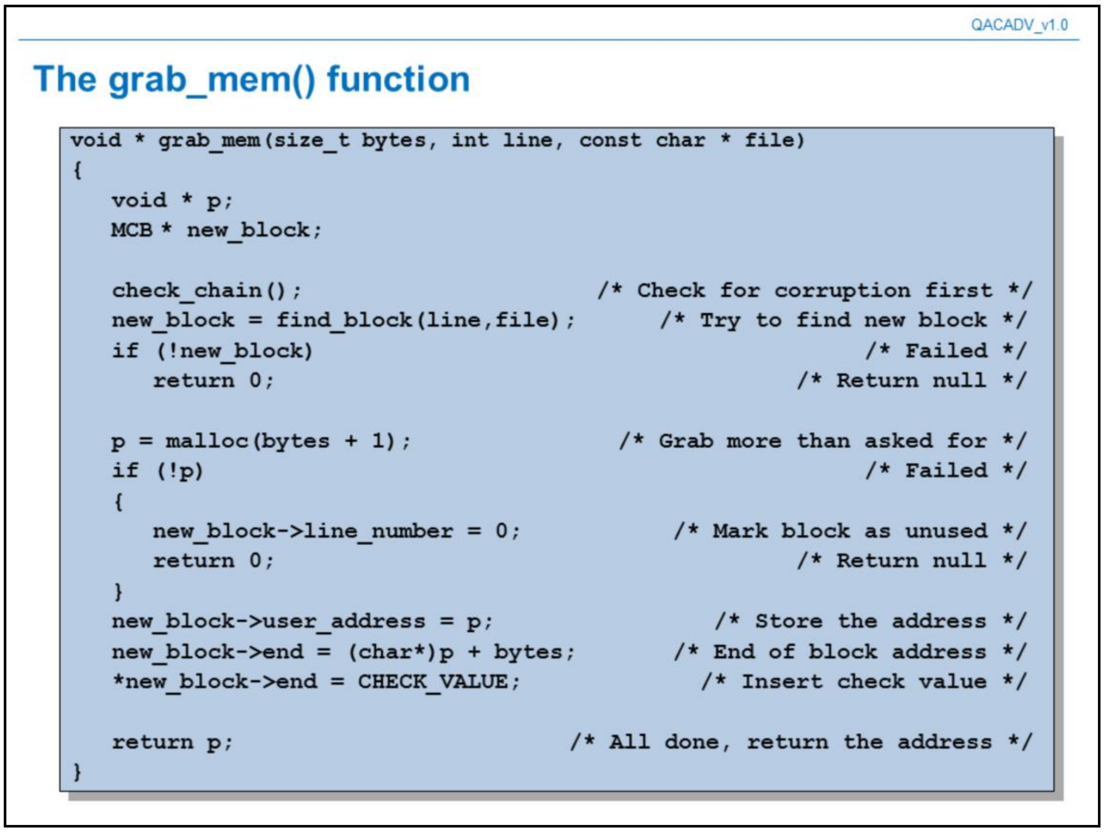
10101010

This value is not really "better" than any other, but it does offer a very distinctive bit pattern that is easily recognisable. It is part of the standard ISO Latin 1 character set (Feminine ordinal indicator - if you must know), but we would have to be unlucky to have a user write this particular character at the end of their string, for example.

More robust checking would use a number of check bytes (Microsoft use four). There is no problem with implementing this; the `end` member would hold the address of the first of these check bytes. We would have to assign each byte by hand and check it by hand as well. One possibility is to construct a strange string and use `strncpy` to copy it to the end of the memory allocated. The `strncmp` routine could then be used to check that it had not been corrupted.

In the code shown above, the variable `how_many` holds the number of `MCB`s allocated thus far. Since it is a global variable, its initial value will be 0. Although not strictly necessary, this variable makes life much easier. To do without it would require a special terminating value on the end of the array pointed to by `block`.

The variable `block` holds the address of the dynamic memory containing the `MCB` structures. Being a global variable (and a pointer) it is initialised with `NULL` automatically. This address may be passed to `realloc` with the value of `how_many` multiplied by the `sizeof` an `MCB`.



QACADV_v1.0

The grab_mem() function

```
void * grab_mem(size_t bytes, int line, const char * file)
{
   void * p;
   MCB * new_block;

   check_chain();                       /* Check for corruption first */
   new_block = find_block(line,file);        /* Try to find new block */
   if (!new_block)                                        /* Failed */
      return 0;                                      /* Return null */

   p = malloc(bytes + 1);                /* Grab more than asked for */
   if (!p)                                                /* Failed */
   {
      new_block->line_number = 0;             /* Mark block as unused */
      return 0;                                      /* Return null */
   }
   new_block->user_address = p;                  /* Store the address */
   new_block->end = (char*)p + bytes;       /* End of block address */
   *new_block->end = CHECK_VALUE;             /* Insert check value */

   return p;                       /* All done, return the address */
}
```

How grabMem works:

First, the `check_chain` function is called to check that dynamic memory has not been corrupted thus far. This is necessary because the `MCB` structures must be allocated from the chain. If the chain is already corrupt, these debugging routines will crash the application.

The `find_block` function returns the address of a suitable `MCB` structure. This is initialised with the `line` and `name` variables passed as parameters. Then the number of bytes the user requested is grabbed from dynamic memory.

If there is no memory available, this allocation will fail, but the `MCB` we were about to use may still be useful later on if the user tries to allocate a (smaller?) lump of memory. The presence of this free `MCB` is indicated by storing the value of 0 in the `line_number` member.

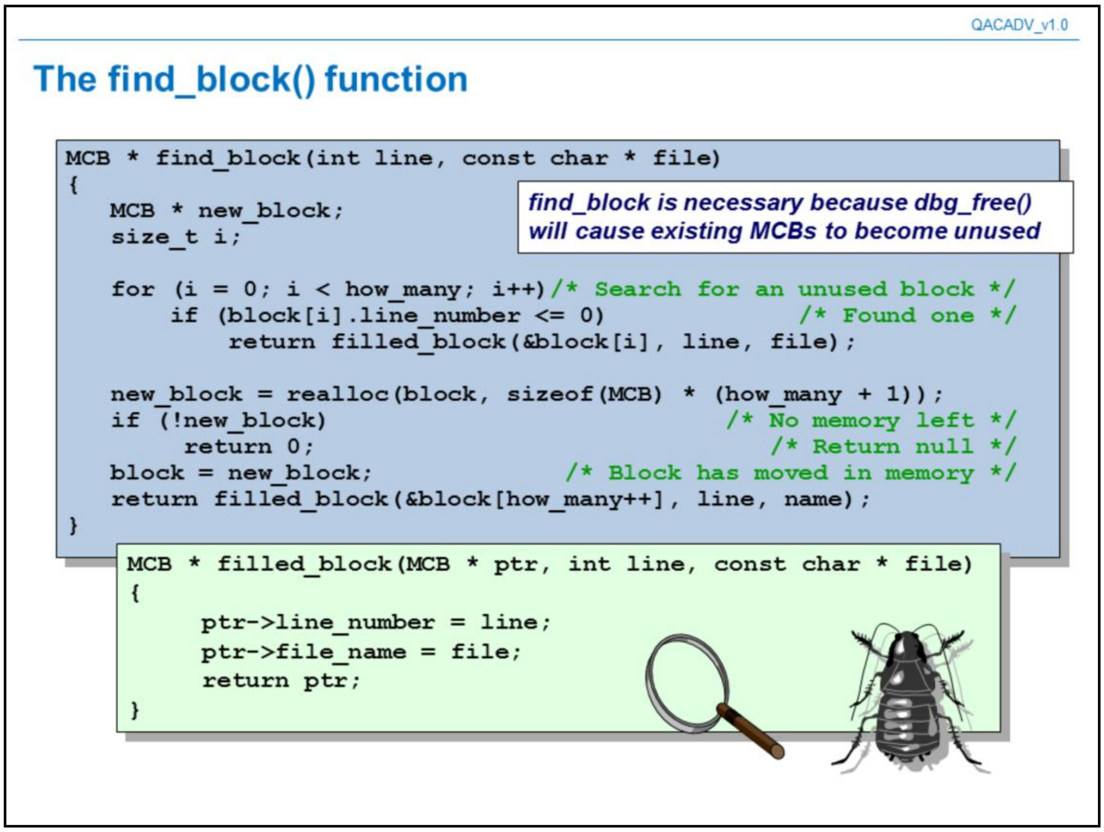
A byte is allocated for the check value, and the address of the end of the block of memory is calculated. This is not as straightforward as it might have been:

```
((char*)p + bytes)
```

Since `p` is a `void` pointer it is cast to a `char*`, so the addition of `bytes` is not scaled and can be assigned to `new_block`**->** `end`.

Once the pointer has been set up, the byte it points to can be assigned the check value. If we had many check-byte values, they could all be copied here. Finally, the address of the block of memory we were supposed to allocate in the first place is returned.

Note: `malloc` *must not be* `#define`d *as* `dbg_malloc` *in this implementation file, otherwise the routines will be infinitely recursive.*



The find_block() function

```
MCB * find_block(int line, const char * file)
{
    MCB * new_block;
    size_t i;

    for (i = 0; i < how_many; i++)/* Search for an unused block */
        if (block[i].line_number <= 0)          /* Found one */
            return filled_block(&block[i], line, file);

    new_block = realloc(block, sizeof(MCB) * (how_many + 1));
    if (!new_block)                          /* No memory left */
        return 0;                               /* Return null */
    block = new_block;               /* Block has moved in memory */
    return filled_block(&block[how_many++], line, name);
}
```

find_block is necessary because dbg_free() will cause existing MCBs to become unused

```
MCB * filled_block(MCB * ptr, int line, const char * file)
{
     ptr->line_number = line;
     ptr->file_name = file;
     return ptr;
}
```

How findBlock works:

The purpose of this routine is to return the address of a valid MCB for use by the grabMem function. Its first attempt is to see if a previously allocated MCB can be "recycled". This will be the case if dbg_free has been called at some point previously.

Unused MCBs are marked by having an invalid value in the line_number member. This will either be 0 (as seen from grab_mem) or negative (as will be seen in dbg_free). If there are no free MCBs, realloc is called to grab enough storage for however many MCBs there currently are, plus one more:

```
sizeof(MCB) * (how_many + 1)
```

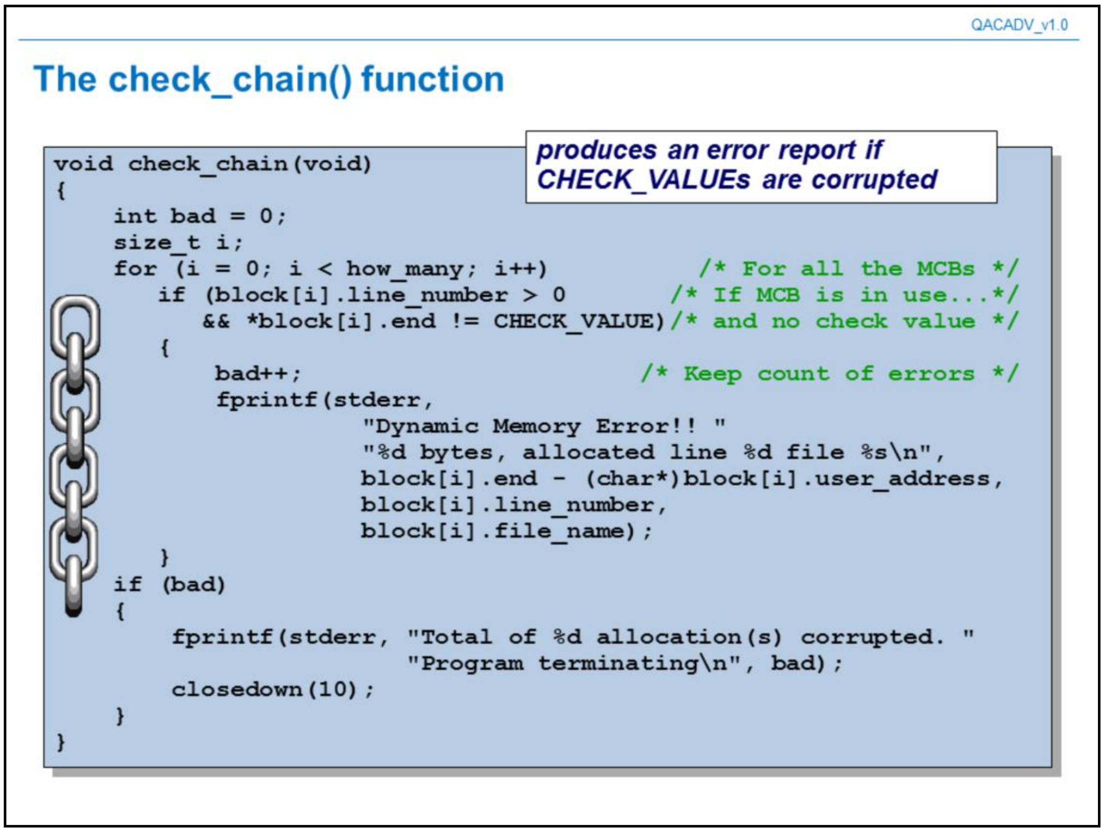
Note that the address returned from realloc *is assigned to* new_block *and not to* block. This is because if realloc failed and the assignment had been:

```
block = realloc(block, (how_many ...));
if (!block)
```

all the debugging information pointed to by block would have been lost. In the safer method shown in the slide, if there is no more heap available we can still debug the dynamic memory that has already been allocated.

If the call to realloc is successful, we have one more MCB and so how_many may be safely incremented. The address of the newly allocated MCB is finally returned.

As before, we must be very careful that realloc is not #defined to be dbg_realloc in the implementation file.



QACADV_v1.0

The check_chain() function

produces an error report if CHECK_VALUEs are corrupted

```
void check_chain(void)
{
    int bad = 0;
    size_t i;
    for (i = 0; i < how_many; i++)             /* For all the MCBs */
       if (block[i].line_number > 0           /* If MCB is in use...*/
          && *block[i].end != CHECK_VALUE)/* and no check value */
       {
          bad++;                            /* Keep count of errors */
          fprintf(stderr,
                    "Dynamic Memory Error!! "
                    "%d bytes, allocated line %d file %s\n",
                    block[i].end - (char*)block[i].user_address,
                    block[i].line_number,
                    block[i].file_name);
       }
    if (bad)
    {
        fprintf(stderr, "Total of %d allocation(s) corrupted. "
                        "Program terminating\n", bad);
        closedown(10);
    }
}
```

All the `check_chain` function has to do is check that the `end` member of each `MCB` still points to the character with the `CHECK_VALUE` value that was initially copied there. If not, the values must have been corrupted, so an error message is printed and the program is killed (via the `exit` function).

How checkChain works:

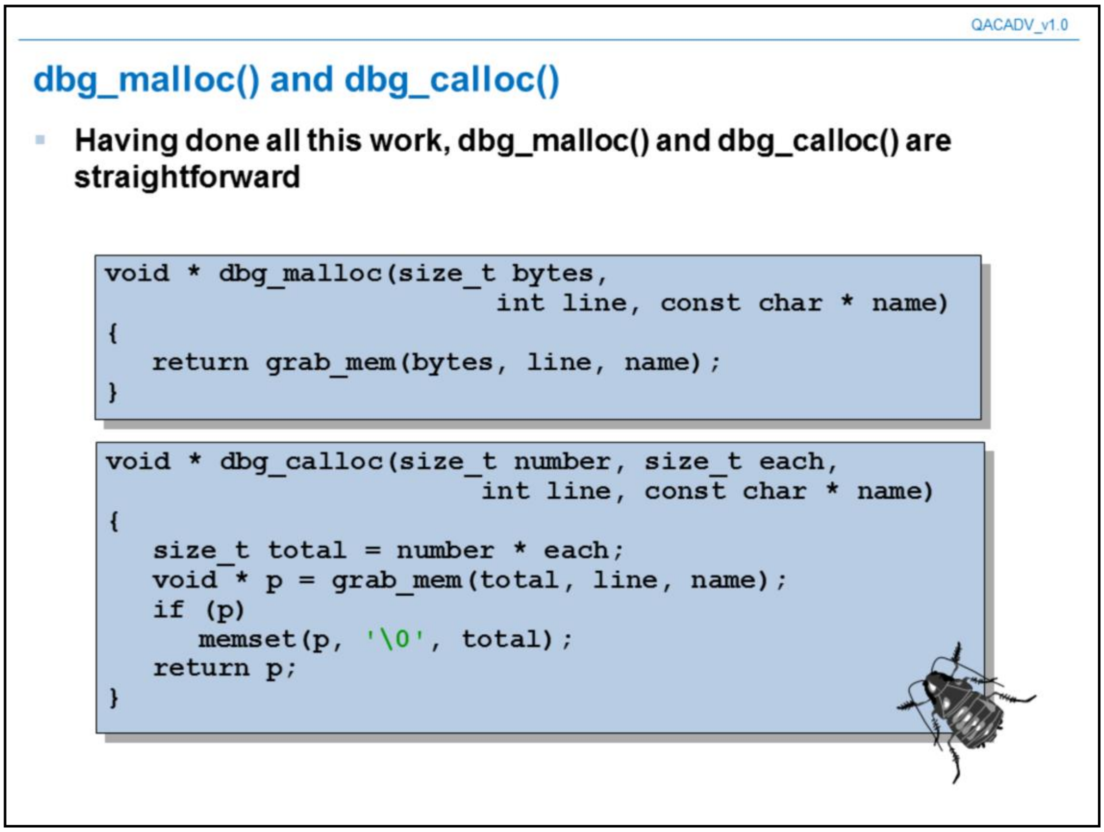
Each of the `how_many` `MCB`s are examined. If an `MCB` contains an invalid line number, it is skipped over (these will be `MCB`s marked by `dbg_free` for recycling or by `grab_mem` as unused). For each `MCB` in use, the `end` pointer is dereferenced and the value at that address is checked against the `CHECK_VALUE`. If any other value is detected, a "bad `MCB`"counter is incremented.

An error message is printed for each corrupt block that is detected. The error message involves a rather intricate calculation:

```
block[i].end - (char*)block[i].user_address
```

This is an example of pointer subtraction, as seen before. However, the two pointers have different types (`end` is a `char*`, while `user_address` is a `void*`). Two different types of pointer may not be subtracted, so the pointers must be reconciled with each other. Both pointers are cast to `char*` to make the types the same. This also avoids the compiler scaling the result.

The line number and filename are printed directly from the structure.



QACADV_v1.0

dbg_malloc() and dbg_calloc()

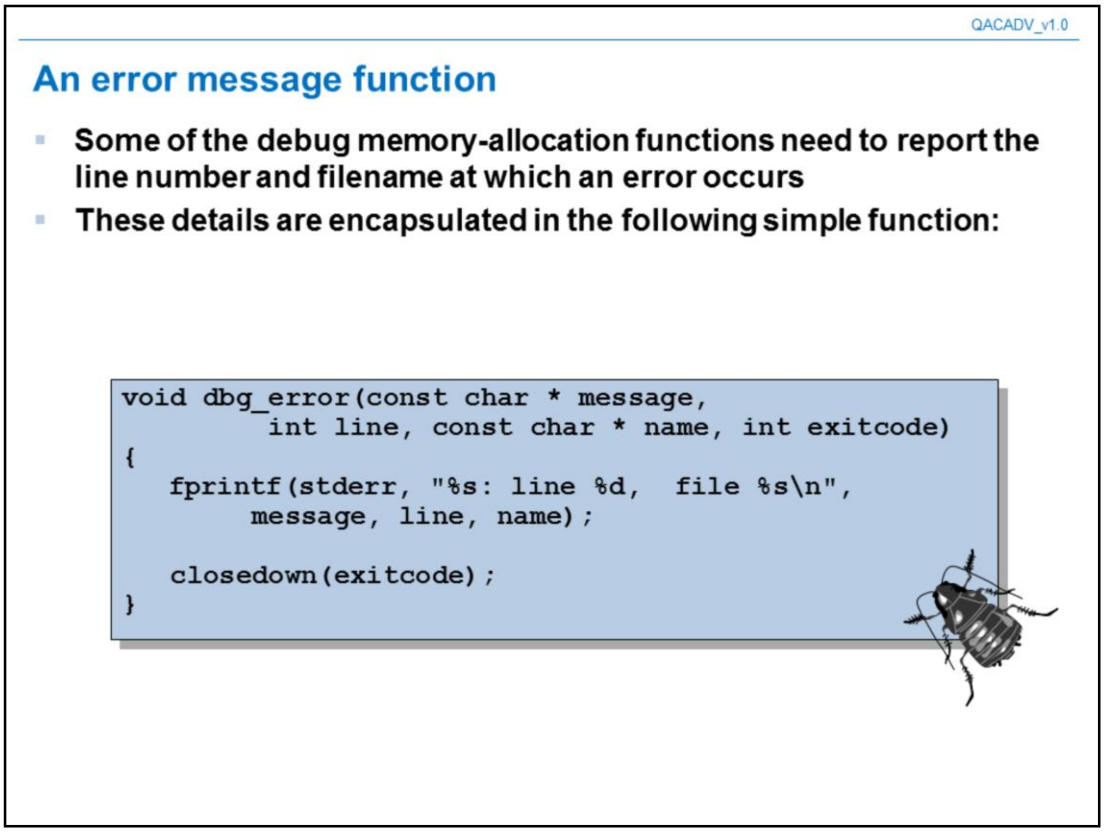
- Having done all this work, dbg_malloc() and dbg_calloc() are straightforward

```
void * dbg_malloc(size_t bytes,
                         int line, const char * name)
{
   return grab_mem(bytes, line, name);
}
```

```
void * dbg_calloc(size_t number, size_t each,
                       int line, const char * name)
{
   size_t total = number * each;
   void * p = grab_mem(total, line, name);
   if (p)
      memset(p, '\0', total);
   return p;
}
```

With so much work already done, `dbg_malloc` and `dbg_calloc` are very straightforward indeed. As far as `dbg_malloc` is concerned, `grab_mem` does everything necessary.

Beyond this, there is another step involved with `dbg_calloc`: that of initialising the memory to zero before its address is returned.



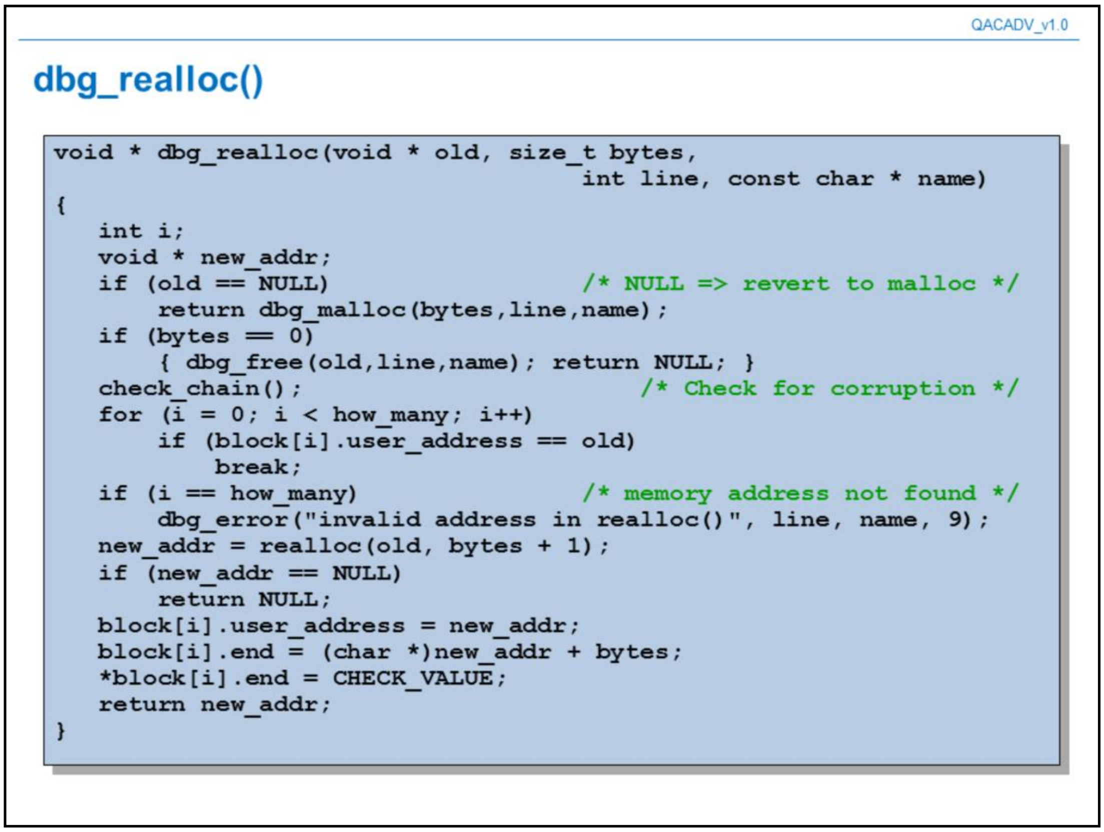
QACADV_v1.0

An error message function

- Some of the debug memory-allocation functions need to report the line number and filename at which an error occurs
- These details are encapsulated in the following simple function:

```
void dbg_error(const char * message,
           int line, const char * name, int exitcode)
{
   fprintf(stderr, "%s: line %d,  file %s\n",
         message, line, name);

   closedown(exitcode);
}
```

The simple error message function shown above will be used by dbg_realloc and dbg_free on the following pages. The function simply prints an error message, together with the line number and filename at which a corrupted chunk of memory was originally allocated. The function could then call exit to terminate the program immediately. Our example performs more sophisticated error processing through our own closedown() routine.



QACADV_v1.0

dbg_realloc()

```
void * dbg_realloc(void * old, size_t bytes,
                                     int line, const char * name)
{
   int i;
   void * new_addr;
   if (old == NULL)                 /* NULL => revert to malloc */
       return dbg_malloc(bytes,line,name);
   if (bytes == 0)
       { dbg_free(old,line,name); return NULL; }
   check_chain();                       /* Check for corruption */
   for (i = 0; i < how_many; i++)
       if (block[i].user_address == old)
           break;
   if (i == how_many)               /* memory address not found */
       dbg_error("invalid address in realloc()", line, name, 9);
   new_addr = realloc(old, bytes + 1);
   if (new_addr == NULL)
       return NULL;
   block[i].user_address = new_addr;
   block[i].end = (char *)new_addr + bytes;
   *block[i].end = CHECK_VALUE;
   return new_addr;
}
```

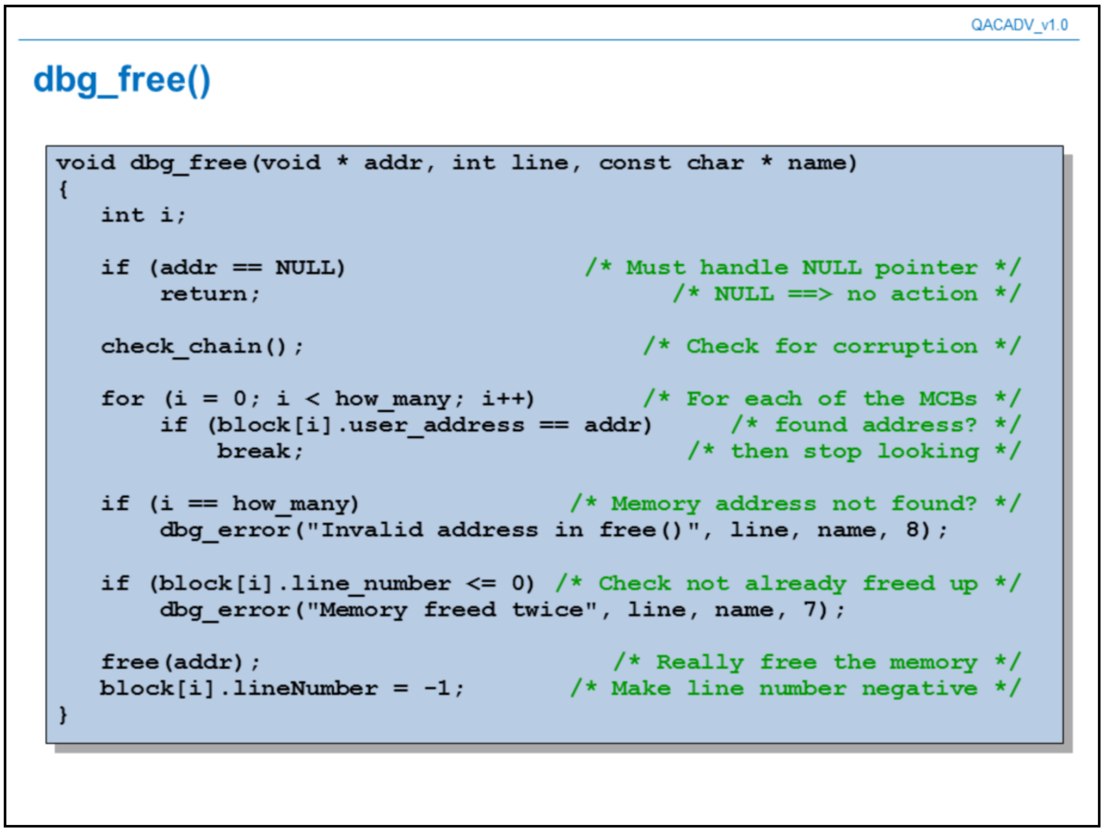
How dbg_realloc works:

`dbg_realloc` is called to re-size an existing block of memory whose address is passed in as the parameter old. If this is a NULL pointer, `dbg_realloc` emulates the pre-defined `realloc` function and simply reverts to allocating a completely fresh block of memory (in the `malloc` fashion). Again, if size is zero, `dbg_realloc` emulates `realloc` by freeing the memory and returning NULL.

Otherwise, the existing chain is checked for consistency before proceeding any further. All being well, `dbg_realloc` scans through the list of existing MCBs trying to find the MCB block associated with the original memory block (as indicated by the old pointer). If no suitable MCB block is found, this constitutes an error condition (effectively, the programmer is trying to re-allocate a block of memory that was not allocated dynamically in the first place).

If all is still in order, `dbg_realloc` calls the pre-defined `realloc` function to carry out the memory re-allocation. The issues regarding the correct usage of `realloc` have already been noted on a previous page. Note that `realloc` will *move* the memory block if necessary to satisfy a request for a larger block of memory.

The new addresses of the start and end of the memory block are stored in the MCB and the CHECK_VALUE is reinstated at the end of the block. The address of the new block is then returned back to the calling function.



QACADV_v1.0

dbg_free()

```
void dbg_free(void * addr, int line, const char * name)
{
   int i;

   if (addr == NULL)                    /* Must handle NULL pointer */
       return;                                /* NULL ==> no action */

   check_chain();                           /* Check for corruption */

   for (i = 0; i < how_many; i++)           /* For each of the MCBs */
       if (block[i].user_address == addr)       /* found address? */
           break;                              /* then stop looking */

   if (i == how_many)                 /* Memory address not found? */
       dbg_error("Invalid address in free()", line, name, 8);

   if (block[i].line_number <= 0)  /* Check not already freed up */
       dbg_error("Memory freed twice", line, name, 7);

   free(addr);                            /* Really free the memory */
   block[i].lineNumber = -1;         /* Make line number negative */
}
```

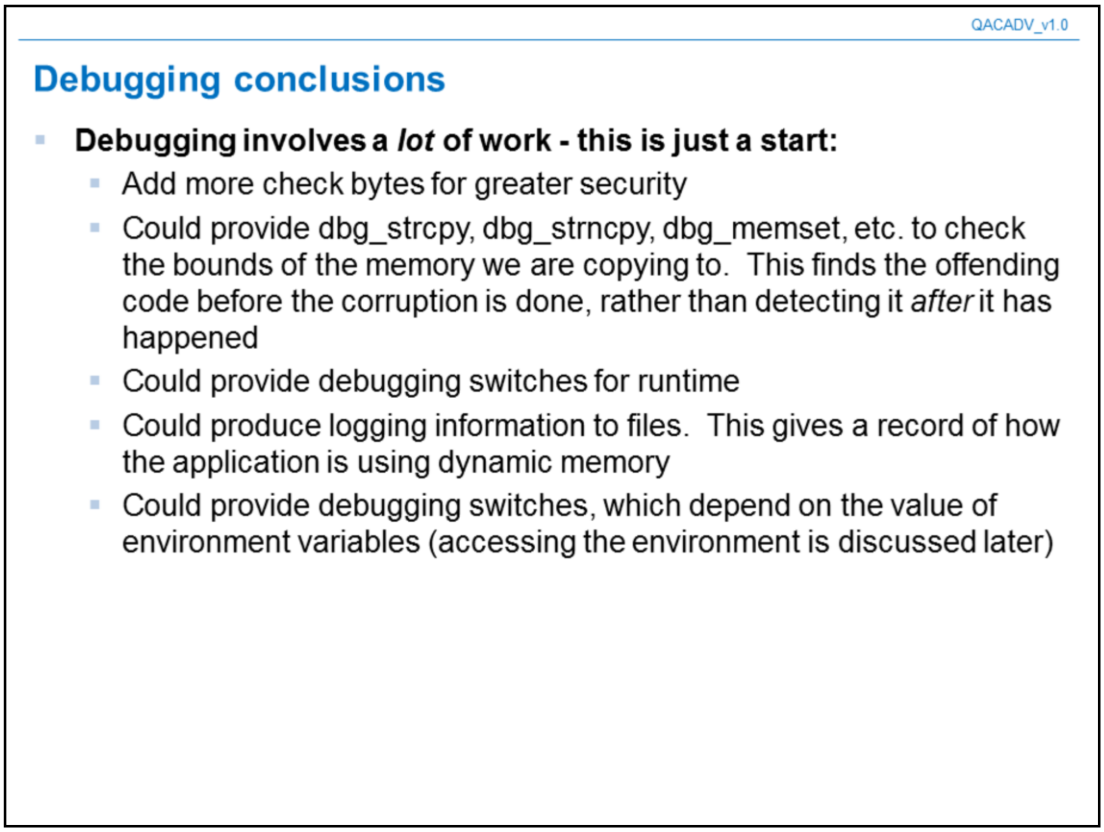
How dbg_free works:

The `dbg_free` routine is quite straightforward. The address to be freed is searched for within each of the `MCB`s. The address must be stored in one of these `MCB`s, since it would have been placed there by either `dbg_malloc` or `dbg_calloc` (which the user must have called at some point previously). If the address is not found in any of the `MCB`s, this constitutes an error, since the implication is that we are freeing a piece of memory which was not allocated dynamically in the first place. Decidedly hazardous to the health!

Another possibility is that we are freeing the same piece of memory twice. This would be less fatal than the first error, depending on how the real dynamic memory allocation routines work. Nonetheless it is an error and as such it is reported.

Barring these error conditions, the line number stored within the corresponding `MCB` is made negative to indicate a recyclable `MCB`. This `MCB` will be re-used by the `find_block` routine.

This approach (flagging the line number as zero) has some advantages. We remember the address of the freed memory. If any accesses are made to it we could potentially trap these problems.



QACADV_v1.0

Debugging conclusions

- **Debugging involves a *lot* of work - this is just a start:**
  - Add more check bytes for greater security
  - Could provide dbg_strcpy, dbg_strncpy, dbg_memset, etc. to check the bounds of the memory we are copying to. This finds the offending code before the corruption is done, rather than detecting it *after* it has happened
  - Could provide debugging switches for runtime
  - Could produce logging information to files. This gives a record of how the application is using dynamic memory
  - Could provide debugging switches, which depend on the value of environment variables (accessing the environment is discussed later)

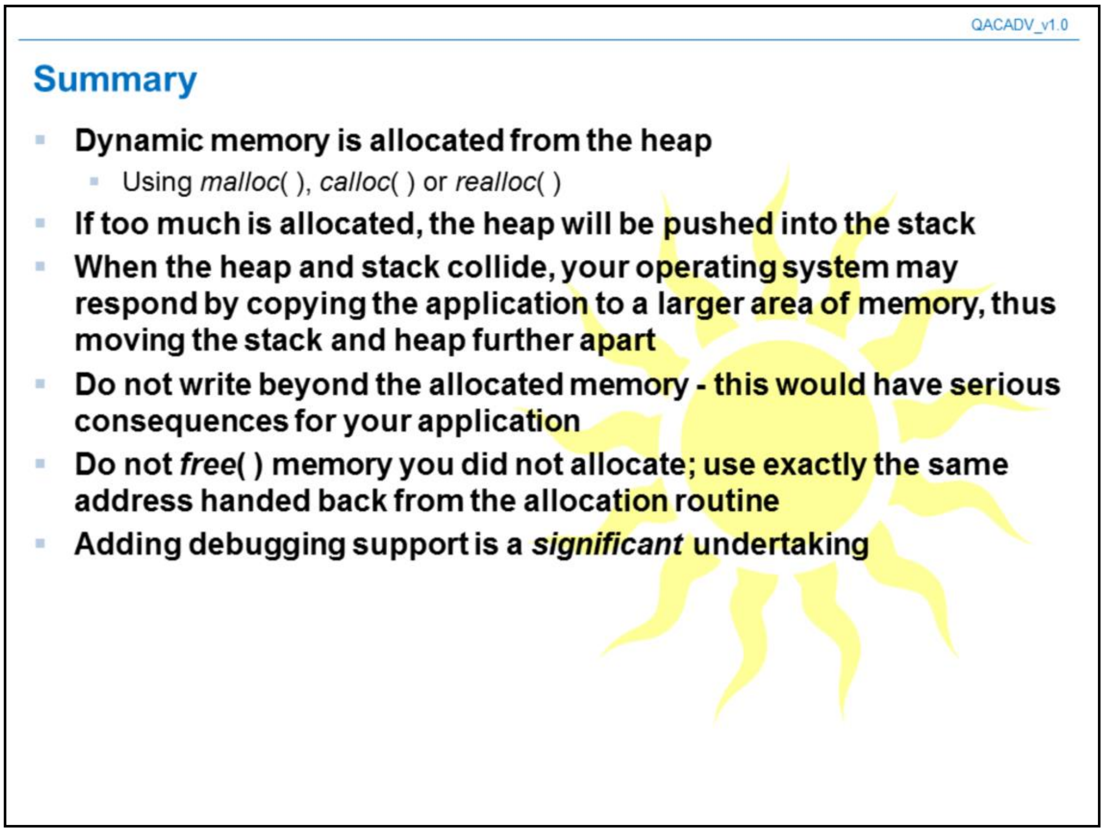
As we've seen, debugging dynamic memory problems is hard work.

This code is only a starting point, however. The most obvious improvement is to add more check bytes. This would be a simple change.

A more profitable pursuit would be to overload `strcpy`, `strncpy`, etc. with their `dbg` equivalents. These routines could walk down the `MCB`s to see if the address being copied to is one it knows about. By subtracting the end and start addresses the amount of memory available may be calculated. The routines could check how many bytes of memory are *going* to be copied across. If too many bytes would be copied across, the program can be stopped before the corruption of memory takes place, rather than picking up the pieces *after* it has happened.

Another possibility would be to switch on debugging half way through the execution of the program. Obviously this would not be as robust, since there would be no way of checking access to memory allocated before debugging was switched on.

Logging information to a file would be a useful option if an application writer wanted to measure how efficiently dynamic memory was being used. The name of the log file could be gained from the program's environment. Accessing the environment may be achieved using `getenv`, as we shall see later in the course.



QACADV_v1.0

Summary

- Dynamic memory is allocated from the heap
  - Using *malloc*( ), *calloc*( ) or *realloc*( )
- If too much is allocated, the heap will be pushed into the stack
- When the heap and stack collide, your operating system may respond by copying the application to a larger area of memory, thus moving the stack and heap further apart
- Do not write beyond the allocated memory - this would have serious consequences for your application
- Do not *free*( ) memory you did not allocate; use exactly the same address handed back from the allocation routine
- Adding debugging support is a *significant* undertaking

At the end of the day, there are only three routines which allocate dynamic memory. These are `malloc`, `calloc` and `realloc`. Of these, the first two are so closely related as to be almost the same, whereas `realloc` alters the amount of memory at the end of a pointer.

The `free` routine exists for releasing the storage. This may not necessarily decrease your heap size. Since these allocation routines chain the blocks of storage together, the consequences of corrupting this chain are invariably fatal.

Much of this chapter has been devoted to illustrate just how difficult it is to debug an application that is doing something wrong with its dynamic memory.