C Programming/Memory management

In C, you have already considered creating variables for use in the program. You have created some arrays for use, but you may have already noticed some limitations:

- the size of the array must be known beforehand
- the size of the array cannot be changed in the duration of your program

Dynamic memory allocation in C is a way of circumventing these problems.

Contents

The `malloc` function

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

The standard C function `malloc` is the means of implementing dynamic memory allocation. It is defined in stdlib.h or malloc.h, depending on what operating system you may be using. Malloc.h contains only the definitions for the memory allocation functions and not the rest of the other functions defined in stdlib.h. Usually you will not need to be so specific in your program, and if both are supported, you should use <stdlib.h>, since that is ANSI C, and what we will use here.

The corresponding call to release allocated memory back to the operating system is `free`.

When dynamically allocated memory is no longer needed, `free` should be called to release it back to the memory pool. Overwriting a pointer that points to dynamically allocated memory can result in that data becoming inaccessible. If this happens frequently, eventually the operating system will no longer be able to allocate more memory for the process. Once the process exits, the operating system is able to free all dynamically allocated memory associated with the process.

Let's look at how dynamic memory allocation can be used for arrays.

Normally when we wish to create an array we use a declaration such as

```
int array[10];
```

Recall `array` can be considered a pointer which we use as an array. We specify the length of this array is 10 `int`s. After `array[0]`, nine other integers have space to be stored consecutively.

Sometimes it is not known at the time the program is written how much memory will be needed for some data; for example, when it depends upon user input. In this case we would want to dynamically allocate required memory after the program has started executing. To do this we only need to declare a pointer, and invoke `malloc` when we wish to make space for the elements in our array, *or*, we can tell `malloc` to make space when we first initialize the array. Either way is acceptable and useful.

We also need to know how much an int takes up in memory in order to make room for it; fortunately this is not difficult, we can use C's builtin `sizeof` operator. For example, if `sizeof(int)` yields 4, then one `int` takes up 4 bytes. Naturally, `2*sizeof(int)` is how much memory we need for 2 `int`s, and so on.

So how do we `malloc` an array of ten `int`s like before? If we wish to declare and make room in one hit, we can simply say

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

We only need to declare the pointer; `malloc` gives us some space to store the 10 `int`s, and returns the pointer to the first element, which is assigned to that pointer.

Important note! `malloc` does *not* initialize the array; this means that the array may contain random or unexpected values! Like creating arrays without dynamic allocation, the programmer must initialize the array with sensible values before using it. Make sure you do so, too. (*See later the function `memset` for a simple method.)*

It is not necessary to immediately call `malloc` after declaring a pointer for the allocated memory. Often a number of statements exist between the declaration and the call to `malloc`, as follows:

```
int *array = NULL;
printf("Hello World!!!");
/* more statements */
array = malloc(10*sizeof(int)); /* delayed allocation */
/* use the array */
```

A more practical example of dynamic memory allocation would be the following:

> Given an array of 10 integers, remove all duplicate elements from the array, and create a new array without duplicate elements (a set).

A simple algorithm to remove duplicate elements:

```
    int arrl = 10; // Length of the initial array
    int arr[10] = {1, 2, 2, 3, 4, 4, 5, 6, 5, 7}; // A sample array, containing several duplicate elements

    for (int x = 0; x < arrl; x++)
    {
        for (int y = x + 1; y < arrl; y++)
        {
            if (arr[x] == arr[y])
            {
                for (int s = y; s < arrl; s++)
                {
                    if (!(s + 1 == arrl))
                        arr[s] = arr[s + 1];
                }

                arrl--;
                y--;
            }
        }
    }
```

Because the length of our new array depends on the input, it must be dynamically allocated:

```
int *newArray = malloc(arrl*sizeof(int));
```

The above array will currently contain unexpected values, so we must use `memcpy` to set our dynamically allocated memory block to the new values:

```
memcpy(newArray, arr, arrl*sizeof(int));
```

Some security researchers recommend always using calloc(x,y) rather than malloc(x*y), for 2 reasons:

- Many implementations of calloc() carefully check the x and y arguments and return NULL if "x*y" could overflow. Using malloc(x*y), the multiplication "x*y" can overflow to 0 or some other too-small number, usually leading to buffer overflow.[1][2][3][4]
- calloc ensures that the buffer is completely empty of sensitive information, avoiding some kinds of security bugs[5] (but, unfortunately, this would not have prevented the Heartbleed bug).

Error checking

When we want to use `malloc`, we have to be mindful that the pool of memory available to the programmer is *finite*. Even if a modern PC will have at least an entire gigabyte of memory, it is still possible and conceivable to run out of it! In this case, `malloc` will return `NULL`. In order to stop the program crashing from having no more memory to use, one should always check that malloc has not returned `NULL` before attempting to use the memory; we can do this by

```
int *pt = malloc(3 * sizeof(int));
if(pt == NULL)
{
   fprintf(stderr, "Out of memory, exiting\n");
   exit(1);
}
```

Of course, suddenly quitting as in the above example is not always appropriate, and depends on the problem you are trying to solve and the architecture you are programming for. For example, if the program is a small, non critical application that's running on a desktop quitting may be appropriate. However if the program is some type of editor running on a desktop, you may want to give the operator the option of saving their tediously entered information instead of just exiting the program. A memory allocation failure in an embedded processor, such as might be in a washing machine, could cause an automatic reset of the machine. For this reason, many embedded systems designers avoid dynamic memory allocation altogether.

The `calloc` function

The `calloc` function allocates space for an array of items and initializes the memory to zeros. The call `mArray = calloc( count, sizeof(struct V))` allocates `count` objects, each of whose size is sufficient to contain an instance of the structure `struct V`. The space is initialized to all bits zero. The function returns either a pointer to the allocated memory or, if the allocation fails, `NULL`.

The `realloc` function

```
void * realloc ( void * ptr, size_t size );
```

The `realloc` function changes the size of the object pointed to by `ptr` to the size specified by `size`. The contents of the object shall be unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If `ptr` is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by the `calloc`, `malloc`, or `realloc` function, or if the space has been deallocated by a call to the `free` or `realloc` function, the behavior is undefined. If the space cannot be allocated, the object pointed to by `ptr` is unchanged. If `size` is zero and `ptr` is not a null pointer, the object pointed to is freed. The `realloc` function returns either a null pointer or a pointer to the possibly moved allocated object.

The `free` function

Memory that has been allocated using `malloc`, `realloc`, or `calloc` must be released back to the system memory pool once it is no longer needed. This is done to avoid perpetually allocating more and more memory, which could result in an eventual memory allocation failure. Memory that is not released with `free` is however released when the current program terminates on most operating systems. Calls to `free` are as in the following example.

```
int *myStuff = malloc( 20 * sizeof(int));
if (myStuff != NULL)
{
   /* more statements here */
   /* time to release myStuff */
   free( myStuff );
}
```

free with recursive data structures

It should be noted that `free` is neither intelligent nor recursive. The following code that depends on the recursive application of free to the internal variables of a struct does not work.

```
typedef struct BSTNode
{
   int value;
   struct BSTNode* left;
   struct BSTNode* right;
} BSTNode;

// Later: ...

BSTNode* temp = calloc(1, sizeof(BSTNode));
temp->left = calloc(1, sizeof(BSTNode));

// Later: ...

free(temp); // WRONG! don't do this!
```

The statement "`free(temp);`" will **not** free `temp->left`, causing a memory leak. The correct way is to define a function that frees *every* node in the data structure:

```
void BSTFree(BSTNode* node){
    if (node != NULL) {
        BSTFree(node->left);
        BSTFree(node->right);
        free(node);
    }
}
```

Because C does not have a garbage collector, C programmers are responsible for making sure there is a `free()` exactly once for each time there is a `malloc()`. If a tree has been allocated one node at a time, then it needs to be freed one node at a time.

Don't free undefined pointers

Furthermore, using `free` when the pointer in question was never allocated in the first place often crashes or leads to mysterious bugs further along.

To avoid this problem, always initialize pointers when they are declared. Either use `malloc` at the point they are declared (as in most examples in this chapter), or set them to `NULL` when they are declared (as in the "delayed allocation" example in this chapter). [6]

Write constructor/destructor functions

One way to get memory initialization and destruction right is to imitate object-oriented programming. In this paradigm, objects are constructed after raw memory is allocated for them, live their lives, and when it is time for them to be destructed, a special function called a destructor destroys the object's innards before the object itself is destroyed.

For example:

```
#include <stdlib.h> /* need malloc and friends */

/* this is the type of object we have, with a single int member */
typedef struct WIDGET_T {
  int member;
} WIDGET_T;

/* functions that deal with WIDGET_T */

/* constructor function */
void
WIDGETctor (WIDGET_T *this, int x)
{
  this->member = x;
}

/* destructor function */
void
WIDGETdtor (WIDGET_T *this)
{
  /* In this case, I really don't have to do anything, but
     if WIDGET_T had internal pointers, the objects they point to
     would be destroyed here.  */
  this->member = 0;
}

/* create function - this function returns a new WIDGET_T */
WIDGET_T *
WIDGETcreate (int m)
{
  WIDGET_T *x = 0;

  x = malloc (sizeof (WIDGET_T));
  if (x == 0)
    abort (); /* no memory */
  WIDGETctor (x, m);
  return x;
}

/* destroy function - calls the destructor, then frees the object */
void
WIDGETdestroy (WIDGET_T *this)
{
  WIDGETdtor (this);
  free (this);
}

/* END OF CODE */
```

References

1. "calloc when multiply overflows" (https://drj11.wordpress.com/2008/06/04/calloc-when-multiply-overflows/)
2. "Why does calloc exist?" (https://news.ycombinator.com/item?id=33579884).
3. "MEM07-C. Ensure that the arguments to calloc(), when multiplied, do not wrap" (https://wiki.sei.cmu.edu/confluence/display/c/MEM07-C.+Ensure+that+the+arguments+to+calloc%28%29%2C+when+multiplied%2C+do+not+wrap).
4. "ps buffer overflow - CVE 2023-4016" (https://www.freelists.org/post/procps/ps-buffer-overflow-CVE-2023-4016).
5. "MEM03-C. Clear sensitive information stored in reusable resources" (https://wiki.sei.cmu.edu/confluence/display/c/MEM03-C.+Clear+sensitive+information+stored+in+reusable+resources).
6. "Bug 478901 ... libpng-1.2.34 and earlier might free undefined pointers" (https://bugzilla.mozilla.org/show_bug.cgi?id=478901)

- Memory Management

Retrieved from "https://en.wikibooks.org/w/index.php?title=C_Programming/Memory_management&oldid=4404894"

This page was last edited on 4 June 2024, at 22:34.

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.