# Sams Teach Yourself C in 24 Hours



# Hour 17 - Allocating Memory

It's just as unpleasant to get more than you bargain for as to get less.

**—G. B. Shaw**

So far you've learned how to declare and reserve a piece of memory space before it is used in your program. For instance, you have to specify the size of an array in your program (or the compiler has to figure out the size if you declare an unsized array) before you assign any data to it at runtime. In this lesson you'll learn to allocate memory space dynamically when your program is running. The four dynamic memory allocation functions covered in this lesson are

- The malloc() function
- The calloc() function
- The realloc() function
- The free() function

## Allocating Memory at Runtime

There are many cases when you do not know the exact sizes of arrays used in your programs, until much later when your programs are actually being executed. You can specify the sizes of arrays in advance, but the arrays can be too small or too big if the numbers of data items you want to put into the arrays change dramatically at runtime.

Fortunately, C provides you with four dynamic memory allocation functions that you can employ to allocate or reallocate certain memory spaces while your program is running. Also, you can release allocated memory storage as soon as you don't need it. These four C functions, malloc(), calloc(), realloc(), and free(), are introduced in the following sections.

## The malloc() Function

You can use the malloc() function to allocate a specified size of memory space.

The syntax for the malloc() function is

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

Here size indicates the number of bytes of storage to allocate. The malloc() function returns a void pointer.

Note that the header file, stdlib.h, has to be included before the malloc() function can be called. Because the malloc() function returns a void pointer, its type is automatically converted to the type of the pointer on the left side of an assignment operator.

If the malloc() function fails to allocate a piece of memory space, it returns a null pointer. Normally, this happens when there is not enough memory. Therefore, you should always check the returned pointer from malloc() before you use it.

Listing 17.1 demonstrates the use of the malloc() function.

**TYPE**

**Listing 17.1. Using the malloc() function.**

```
1:  /* 17L01.c: Using the malloc function */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  #include <string.h>
5:  /* function declaration */
6:  void StrCopy(char *str1, char *str2);
7:  /* main() function */
8:  main()
9:  {
10:    char str[] = "Use malloc() to allocate memory.";
11:    char *ptr_str;
12:    int result;
13:    /* call malloc() */
14:    ptr_str = malloc( strlen(str) + 1);
15:    if (ptr_str != NULL){
16:       StrCopy(str, ptr_str);
17:       printf("The string pointed to by ptr_str is:\n%s\n",
18:              ptr_str);
19:       result = 0;
20:    }
21:    else{
22:       printf("malloc() function failed.\n");
23:       result = 1;
24:    }
25:    return result;
26: }
27: /* function definition */
28: void StrCopy(char *str1, char *str2)
29: {
30:    int i;
31:
```

```
32:    for (i=0; str1[i]; i++)
33:       str2[i] = str1[i];
34:    str2[i] = `\0';
35: }
```

The following output is shown on the screen after the executable, 17L01.exe, of the program in Listing 17.1 is created and executed:

**OUTPUT**

```
C:\app>17L01
The string pointed to by ptr_str is:
Use malloc() to allocate memory.
C:\app>
```

**ANALYSIS**

The purpose of the program in Listing 17.1 is to use the malloc() function to allocate a piece of memory space that has the same size as a character string. Then, the content .of the string is copied to the allocated memory referenced by the pointer returned from the malloc() function. The content of the memory is displayed on the screen to prove that the memory space does contain the content of the string after the allocation and duplication.

Note that two more header files, stdlib.h and string.h, are included in lines 3 and 4, respectively, for the functions malloc() and strlen(), which are called in line 14.

Line 10 declares a char array, str, that is initialized with a character string of "Use malloc() to allocate memory.". A char pointer variable, ptr_str, is declared in line 11.

The statement in line 14 allocates a memory space of strlen(str)+1 bytes by calling the malloc() function. Because the strlen() function does not count the null character at the end of a string, adding 1 to the value returned by strlen(str) gives the total number of bytes that need to be allocated. The value of the returned pointer is assigned to the char pointer variable ptr_str after the malloc() function is called in line 14.

The if-else statement in lines 15_24 checks the returned pointer from the malloc() function. If it's a null pointer, an error message is printed out, and the return value of the main() function is set to 1 in lines 22 and 23. (Remember that a nonzero value returned by the return statement indicates an abnormal termination.)

But if the returned pointer is not a null pointer, the start address of the str array and the pointer ptr_str are passed to a subfunction called StrCopy() in line 16. The StrCopy() function, whose definition is given in lines 28_35, copies the content of the str array to the allocated memory pointed to by ptr_str. Then, the printf() function in lines 17 and 18 prints out the copied content in the allocated memory. Line 19 sets the return value to 0 after the success of the memory allocation and string duplication.

The output on my screen shows that a piece of memory has been allocated and that the string has been copied to the memory.

There is a potential problem if you keep allocating memory, because there is always a limit. You can easily run out of memory when you just allocate memory without releasing it. In the next section, you'll learn how to use the free() function to free up memory spaces allocated for you when you don't need them.

## Releasing Allocated Memory with free()

Because memory is a limited resource, you should allocate an exactly sized piece of memory right before you need it, and release it as soon as you don't need it.

The program in Listing 17.2 demonstrates how to release allocated memory by calling the free() function.

**TYPE**

**Listing 17.2. Using the free() and malloc() functions together.**

```
 1:  /* 17L02.c: Using the free() function */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:  /* function declarations */
 5:  void DataMultiply(int max, int *ptr);
 6:  void TablePrint(int max, int *ptr);
 7:  /* main() function */
 8:  main()
 9:  {
10:     int *ptr_int, max;
11:     int termination;
12:     char key = `c';
13:
14:     max = 0;
15:     termination = 0;
16:     while (key != `x'){
17:        printf("Enter a single digit number:\n");
18:        scanf("%d", &max);
19:
20:        ptr_int = malloc(max * max * sizeof(int));  /* call malloc() */
21:        if (ptr_int != NULL){
22:           DataMultiply(max, ptr_int);
23:           TablePrint(max, ptr_int);
24:           free(ptr_int);
25:        }
26:        else{
27:           printf("malloc() function failed.\n");
28:           termination = 1;
29:           key = `x';  /* stop while loop */
30:        }
31:        printf("\n\nPress x key to quit; other key to continue.\n");
```

```
32:       scanf("%s", &key);
33:    }
34:    printf("\nBye!\n");
35:    return termination;
36: }
37: /* function definition */
38: void DataMultiply(int max, int *ptr)
39: {
40:    int i, j;
41:
42:    for (i=0; i<max; i++)
43:       for (j=0; j<max; j++)
44:          *(ptr + i * max + j) = (i+1) * (j+1);
45: }
46: /* function definition */
47: void TablePrint(int max, int *ptr)
48: {
49:    int i, j;
50:
51:    printf("The multiplication table of %d is:\n",
52:            max);
53:    printf(" ");
54:    for (i=0; i<max; i++)
55:       printf("%4d", i+1);
56:    printf("\n ");
57:    for (i=0; i<max; i++)
58:       printf("----", i+1);
59:    for (i=0; i<max; i++){
60:       printf("\n%d|", i+1);
61:       for (j=0; j<max; j++)
62:         printf("%3d ", *(ptr + i * max + j));
63:    }
64: }
```

While the executable 17L02.exe is being run, I enter two integers, 4 and 2 (highlighted in the following output), to obtain a multiplication table for each; then I quit running the program by pressing the x key:

**OUPUT**

```
C:\app>17L02
Enter a single digit number:
4
The multiplication table of 4 is:
   1   2   3   4
 --------------
1| 1   2   3   4
2| 2   4   6   8
3| 3   6   9  12
4| 4   8  12  16
Press x-key to quit; other key to continue.
C
Enter a single digit number:
2
The multiplication table of 2 is:
   1   2
 ------
1| 1   2
2| 2   4
Press x-key to quit; other key to continue.
x
Bye!
C:\app>
```

**ANALYSIS**

The purpose of the program in Listing 17.2 is to build a multiplication table based on the integer given by the user. The program can continue building multiplication tables until the user presses the x key to quit. The program also stops execution if the malloc() function fails.

To show you how to use the free() function, the program allocates a temporary memory storage to hold the items of a multiplication table. As soon as the content of a multiplication table is printed out, the allocated memory is released by calling the free() function.

Lines 5 and 6 declare two functions, DataMultiply() and TablePrint(), respectively. The former is for performing multiplication and building a table, whereas the latter prints out the table on the screen. The definitions of the two functions are given in lines 38_45 and lines 47_64, respectively.

Inside the main() function, there is a while loop in lines 16_33 that keeps asking the user to enter an integer number (see lines 17 and 18) and then building a multiplication table based on the integer.

To hold the result of the multiplication, the statement in line 20 allocates a memory storage that has the size of max*max*sizeof(int), where the int variable max contains the integer value entered by the user. Note that the sizeof(int) expression gives the byte number of the int data type of the computer on which the program is being run.

If the malloc() function returns a null pointer, the return value of the main() function is set to 1 to indicate an abnormal termination (see line 28), and the while loop is stopped by assigning the key variable with `x' in line 29.

Otherwise, if the malloc() function allocates a memory storage successfully, the DataMultiply() function is called in line 22 to calculate each multiplication. The results are saved into the memory storage pointed to by the ptr_int pointer. Then the multiplication table is printed out by calling the TablePrint() function in line 23.

As soon as I no longer need to keep the multiplication table, I call the free() function in line 24 to release the allocated memory storage

pointed to by the ptr_int pointer.

If I did not release the memory, the program would take more and more memory as the user keeps entering integer numbers to build more multiplication tables. Eventually, the program would either crash the operating system or be forced to quit. By using the free() and malloc() functions, I am able to keep running the program by taking the exact amount of memory storage I need, no more and no less.

## The calloc() Function

Besides the malloc() function, you can also use the calloc() function to allocate a memory storage dynamically. The differences between the two functions are that the latter takes two arguments and that the memory space allocated by calloc() is always initialized to 0. There is no such guarantee that the memory space allocated by malloc() is initialized to 0.

The syntax for the calloc() function is

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

Here nitem is the number of items you want to save in the allocated memory space. size gives the number of bytes that each item takes. The calloc() function returns a void pointer too.

If the calloc() function fails to allocate a piece of memory space, it returns a null pointer.

Listing 17.3 contains an example of using the calloc() function. The initial value of the memory space allocated by calloc() is printed out.

**TYPE**

**Listing 17.3. Using the calloc() function.**

```
1:  /* 17L03.c: Using the calloc() function */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  /* main() function */
5:  main()
6:  {
7:     float *ptr1, *ptr2;
8:     int i, n;
9:     int termination = 1;
10:
11:    n = 5;
12:    ptr1 = calloc(n, sizeof(float));
13:    ptr2 = malloc(n * sizeof(float));
14:    if (ptr1 == NULL)
15:       printf("malloc() failed.\n");
16:    else if (ptr2 == NULL)
17:       printf("calloc() failed.\n");
18:    else {
19:       for (i=0; i<n; i++)
20:          printf("ptr1[%d]=%5.2f,  ptr2[%d]=%5.2f\n",
21:           i, *(ptr1 + i), i, *(ptr2 + i));
22:       free(ptr1);
23:       free(ptr2);
24:       termination = 0;
25:    }
26:    return termination;
27: }
```

The following output appears on the screen after running the executable 17L03.exe:

**OUTPUT**

```
C:\app>17L03
ptr1[0] = 0.00,  ptr2[0] = 7042.23
ptr1[1] = 0.00,  ptr2[1] = 1427.00
ptr1[2] = 0.00,  ptr2[2] = 2787.14
ptr1[3] = 0.00,  ptr2[3] =    0.00
ptr1[4] = 0.00,  ptr2[4] = 5834.73
C:\app>
```

**ANALYSIS**

The purpose of the program in Listing 17.3 is to use the calloc() function to allocate a piece of memory space. To prove that the calloc() function initializes the allocated memory space to 0, the initial values of the memory are printed out. Also, another piece of memory space is allocated by using the malloc() function, and the initial values of the second memory space is printed out too.

As you see in line 12, the calloc() function is called with two arguments passed to it: the int variable n and the sizeof(float) expression. The float pointer variable ptr1 is assigned the value returned by the calloc() function.

Likewise, the malloc() function is called in line 13. This function only takes one argument that specifies the total number of bytes that the allocated memory should have. The value returned by the malloc() function is then assigned to another float pointer variable, ptr2.

From lines 12 and 13, you can tell that the calloc() and malloc() functions actually plan to allocate two pieces of memory space with the same size.

The if-else-if-else statement in lines 14_25 checks the two values returned from the calloc() and malloc() functions and then prints out the initial values from the two allocated memory spaces if the two return values are not null.

I ran the executable program in Listing 17.3 several times. Each time, the initial value from the memory space allocated by the calloc() function was always 0. But there is no guarantee for the memory space allocated by the malloc() function. The output shown here is one of

the results from running the executable program on my machine. You can see that there is some "garbage" in the memory space allocated by the malloc() function. That is, the initial value in the memory is unpredictable. (Sometimes, the initial value in a memory block allocated by the malloc() function is 0. But it is not guaranteed that the initial value is always 0 each time when the malloc() function is called.)

## The realloc() Function

The realloc() function gives you a means to change the size of a piece of memory space allocated by the malloc() function, the calloc() function, or even itself.

The syntax for the realloc() function is

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

Here block is the pointer to the start of a piece of memory space previously allocated. size specifies the total byte number you want to change to. The realloc() function returns a void pointer.

The realloc() function returns a null pointer if it fails to reallocate a piece of memory space.

The realloc() function is equivalent to the malloc() function if the first argument passed to realloc() is NULL. In other words, the following two statements are equivalent:

```
ptr_flt = realloc(NULL, 10 * sizeof(float));
ptr_flt = malloc(10 * sizeof(float));
```

Also, you can use the realloc() function as the free() function. You do this by passing 0 to realloc() as its second argument. For instance, to release a block of memory pointed to by a pointer ptr, you can either call the free() function like this:

```
free(ptr);
```

or use the realloc() function in the following way:

```
realloc(ptr, 0);
```

The program in Listing 17.4 demonstrates the use of the realloc() function in memory reallocation.

**TYPE**

**Listing 17.4. Using the realloc() function.**

```
1:  /* 17L04.c: Using the realloc() function */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  #include <string.h>
5:  /* function declaration */
6:  void StrCopy(char *str1, char *str2);
7:  /* main() function */
8:  main()
9:  {
10:    char *str[4] = {"There's music in the sighing of a reed;",
11:                    "There's music in the gushing of a rill;",
12:                    "There's music in all things if men had ears;",
13:                    "There earth is but an echo of the spheres.\n"
14:                   };
15:    char *ptr;
16:    int i;
17:
18:    int termination = 0;
19:
20:    ptr = malloc((strlen(str[0]) + 1) * sizeof(char));
21:    if (ptr == NULL){
22:      printf("malloc() failed.\n");
23:      termination = 1;
24:    }
25:    else{
26:      StrCopy(str[0], ptr);
27:      printf("%s\n", ptr);
28:      for (i=1; i<4; i++){
29:        ptr = realloc(ptr, (strlen(str[i]) + 1) * sizeof(char));
30:        if (ptr == NULL){
31:          printf("realloc() failed.\n");
32:          termination = 1;
33:          i = 4;    /* break the fro loop */
34:        }
35:        else{
36:          StrCopy(str[i], ptr);
37:          printf("%s\n", ptr);
38:        }
39:      }
40:    }
41:    free(ptr);
42:    return termination;
43: }
44: /* function definition */
45: void StrCopy(char *str1, char *str2)
46: {
47:    int i;
48:
49:    for (i=0; str1[i]; i++)
50:       str2[i] = str1[i];
```

```
51:    str2[i] = `\0';
52: }
```

The following output is obtained by running the executable 17L04.exe:

**OUTPUT**

```
C:\app>17L04
There's music in the sighing of a reed;
There's music in the gushing of a rill;
There's music in all things if men had ears;
There earth is but an echo of the spheres.
C:\app>
```

**ANALYSIS**

The purpose of the program in Listing 17.4 is to allocate a block of memory space to hold a character string. There are four strings in this example, and the length of each string may vary. I use the realloc() function to adjust the size of the previously allocated memory so it can hold a new string.

As you can see in lines 10_13, there are four character strings containing a lovely poem written by Lord Byron. (You can tell that I love Byron's poems.) Here I use an array of pointers, str, to refer to the strings.

A piece of memory space is first allocated by calling the malloc() function in line 20. The size of the memory space is determined by the (strlen(str[0])+1)*sizeof(char) expression. As mentioned earlier, because the C function strlen() does not count the null character at the end of a string, you have to remember to allocate one more piece of memory to hold the full size of a string. The sizeof(char) expression is used here for portability, although the char data type is 1 byte long on most computers.

Exercise 4 at the end of this lesson asks you to rewrite the program in Listing 17.4 and replace the malloc() and free() functions with their equivalent formats of the realloc() functions.

If the malloc() function doesn't fail, the content of the first string pointed to by the str[0] pointer is copied to the block memory allocated by malloc(). To do this, a function called StrCopy() is called in line 26. Lines 45_52 give the definition of StrCopy().

The for loop, in lines 28_39, copies the remaining three strings, one at a time, to the block of memory pointed to by ptr. Each time, the realloc() function is called in line 29 to reallocate and adjust the previously allocated memory space based on the length of the next string whose content is about to be copied to the memory block.

After the content of a string is copied to the memory block, the content is also printed out (see lines 27 and 37).

In this example, a block of memory space is allocated and adjusted based on the length of each of the four strings. The realloc() function, as well as the malloc() function, does the memory allocation and adjustment dynamically.

## Summary

In this lesson you've learned the following:

- In C, there are four functions that can be used to allocate, reallocate, or release a block of memory space dynamically at runtime.
- The malloc() function allocates a block of memory whose size is specified by the argument passed to the function.
- The free() function is used to free up a block of memory space previously allocated by the malloc(), calloc(), or realloc() function
- The calloc() function can do the same job as the malloc() function. In addition, the calloc() function can initialize the allocated memory space to 0.
- The realloc() function is used to reallocate a block of memory that has been allocated by the malloc() or calloc() function.
- If a null pointer is passed to the realloc() function as its first argument, the function acts like the malloc() function.
- If the second argument of the realloc() function is set to 0, the realloc() function is equivalent to the free() function that releases a block of allocated memory.
- You have to include the header file stdlib.h before you can call the malloc(), calloc(), realloc(), or free() function.
- You should always check the values returned from the malloc(), calloc(), or realloc() function, before you use the allocated memory made by these functions.

In the next lesson you'll learn more about data types in C.

## Q&A

**Q** Why do you need to allocate memory at runtime?

**A** Very often, you don't know the exact sizes of arrays until your program is being run. You might be able to estimate the sizes for those arrays, but if you make those arrays too big, you waste the memory. On the other hand, if you make those arrays too small, you're going to lose data. The best way is to allocate blocks of memory dynamically and precisely for those arrays when their sizes are determined at runtime. There are four C library functions, malloc(), calloc(), realloc(), and free(), which you can use in memory allocation at runtime.

**Q** What does it mean if the malloc() function returns a null pointer?

**A** If the malloc() function returns a null pointer, it means the function fails to allocate a block of memory whose size is specified by the argument passed to the function. Normally, the failure of the malloc() function is caused by the fact that there is not enough memory to allocate. You should always check the value returned by the malloc() function to make sure that the function has been successful before you use the block of memory allocated by the function.

**Q** What are the differences between the calloc() and malloc() functions?

**A** Basically, there are two differences between the calloc() and malloc() functions, although both of them can do the same job. The first difference is that the calloc() function takes two arguments, while the malloc() function takes only one. The second one

is that the calloc() function initializes the allocated memory space to 0, whereas there is no such guarantee made by the malloc() function.

**Q** Is the free() function necessary?

**A** Yes. The free() function is very necessary, and you should use it to free up allocated memory blocks as soon as you don't need them. As you know, memory is a limited resource in a computer. Your program shouldn't take too much memory space when it allocates blocks of memory. One way to reduce the size of memory taken by your program is to use the free() function to release the unused allocated memory in time.

## Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

### Quiz

1. Provided that the char data type is 1 byte, the int data type is 2 bytes, and the float data type is 4 bytes, how many bytes of memory do the following functions try to allocate?
   - malloc(100 * sizeof(int))
   - calloc(200, sizeof(char))
   - realloc(NULL, 50 * sizeof(float))
   - realloc(ptr, 0)
2. Given an int pointer, ptr, that is pointing to a block of memory that can hold 100 integers, if you want to reallocate the memory block to hold up to 150 integers, which of the two following statements do you use?
   - ptr = realloc(ptr, 50 * sizeof(int));
   - ptr = realloc(ptr, 150 * sizeof(int));
3. After the following statements are executed successfully, what is the final size of the allocated memory block pointed to by the ptr pointer?

```
. . .
ptr = malloc(300 * sizeof(int));
. . .
ptr = realloc(ptr, 500 * sizeof(int));
. . .
ptr = realloc(ptr, 60 * sizeof(int));
```

4. What is the final size of the allocated memory block pointed to by the ptr pointer, if the following statements are executed successfully?

```
. . .
ptr = calloc(100 * sizeof(char));
. . .
free(ptr);
ptr = realloc(NULL, 200 * sizeof(char));
. . .
ptr = realloc(ptr, 0);
```

### Exercises

1. Write a program to ask the user to enter the total number of bytes he or she wants to allocate. Then, initialize the allocated memory with consecutive integers, starting from 1. Add all the integers contained by the memory block and print out the final result on the screen.
2. Write a program that allocates a block of memory space to hold 100 items of the float data type by calling the calloc() function. Then, reallocate the block of memory in order to hold 50 more items of the float data type.
3. Write a program to ask the user to enter the total number of float data. Then use the calloc() and malloc() functions to allocate two memory blocks with the same size specified by the number, and print out the initial values of the two memory blocks.
4. Rewrite the program in Listing 17.4. This time, use the two special cases of the realloc() function to replace the malloc() and free() functions.