Storage Space: Working with Memory

WEEK
1

How you use memory can make or break a program. As you write bigger and more dynamic programs, you need a better understanding of how your C programs use memory. Today will you learn:

- ☐ Why first understanding memory is important.
- ☐ What kind of storage different C variable types require.
- ☐ What dynamic memory allocation is.
- ☐ How to dynamically allocate memory with `malloc()`.
- ☐ What other memory functions will be of use.

It is important that you understand the complexities involved with memory. Starting with Day 2, "Complex Data Types," you will be writing programs that may require dynamically allocating memory.

Why Start the 21 Days with Memory Usage?

It is assumed that you have a basic understanding of the C language. In most beginning C books and beginning C classes, the overall objective is to teach C. The approach most often taken is to present the C keywords and explain what they can do. In addition, many of the common functions, such as `printf()`, that come with C compilers are presented along with what they can do. In presenting these basic concepts, relatively small programs are used, making memory conservation irrelevant.

As you begin to write bigger programs, you will find memory management to be of vital importance. You will find that to avoid running out of memory, your programs will be required to manage the memory needed. This book will quickly get into topics where it will be best to only use the memory you need. In addition, you may not always know how much memory you need until you run the program. This is definitely the case when you work with variable length structures on Day 2. You will find that understanding memory usage is almost mandatory when working with some complex data types and advanced C topics.

Variable Storage Requirements

Everything in C uses memory in one way or another. Listing 1.1 is an expanded version of a program that can be found in virtually all beginning C books. This

program displays the amount of storage space (memory) needed for different variable types and other C constructs.

Type

Listing 1.1. C variable sizes.

```
1:  /* Filename:  LIST0101.c
2:   * Author  :  Bradley L. Jones
3:   * Purpose :  This program prints the size of various C
4:   *            variable types and constructs.
5:   *=========================================================*/
6:
7:  #include <stdio.h>
8:
9:  void main(void)
10: {
11:    char   *char_ptr;
12:    int    *int_ptr;
13:    long   *long_ptr;
14:    short  *short_ptr;
15:    float  *float_ptr;
16:    double *double_ptr;
17:    char   far *far_char_ptr;
18:
19:    struct test_tag {
20:       char a;
21:       int  b;
22:    }test_struct;
23:
24:    printf( "\n Type            Size " );
25:    printf( "\n=========================" );
26:    printf( "\n character         %d ", sizeof( char  ));
27:    printf( "\n integer           %d ", sizeof( int   ));
28:    printf( "\n short             %d ", sizeof( short ));
29:    printf( "\n long              %d ", sizeof( long  ));
30:    printf( "\n float             %d ", sizeof( float ));
31:    printf( "\n double            %d ", sizeof( double));
32:    printf( "\n char pointer      %d ", sizeof( char_ptr  ));
33:    printf( "\n int pointer       %d ", sizeof( int_ptr   ));
34:    printf( "\n short pointer     %d ", sizeof( short_ptr ));
35:    printf( "\n long pointer      %d ", sizeof( long_ptr  ));
36:    printf( "\n float pointer     %d ", sizeof( float_ptr ));
37:    printf( "\n double pointer    %d ", sizeof( double_ptr));
38:    printf( "\n far char pointer  %d ", sizeof( far_char_ptr));
39:    printf( "\n test_structure    %d ", sizeof( test_struct));
40:    printf( "\n=========================");
41: }
```

Output

```
 Type              Size
==========================
 character          1
 integer            2
 short              2
 long               4
 float              4
 double             8
 char pointer       2
 int pointer        2
 short pointer      2
 long pointer       2
 float pointer      2
 double pointer     2
 far char pointer   4
 test_structure     4
==========================
```

Analysis

As you can see from the output, different variable types are different sizes. This program simply uses the `sizeof` operator in conjunction with `printf()` to display the sizes of different variable types. Lines 26 through 31 print the standard C variable types. To make the program clearer, lines 11 to 17 declare pointer variables to be used in the `printf()` calls in lines 32 to 38. In line 39, a `test_struct` structure is printed. This structure was declared in lines 19 to 22. Depending on your compiler's settings, this may be off by 1 byte in size due to byte alignment. When the output was compiled, byte alignment was on. Byte alignment is covered on Day 2.

Review Tip: Remember, `sizeof` is a C keyword used to determine the size of any item in C.

Depending on what size machine you are using, your output—the sizes printed—may be different. For example, a mainframe C compiler may interpret an integer as being 4 bytes long instead of two. For virtually all IBM PC-compatible compilers, the sizes should match those given in the preceding output, with a possible exception being the structure—if byte alignment is off, it may be three instead of four.

Allocating at Compile Time

In Listing 1.1, the storage space required for all the variables was determined at the time the program was compiled. For a program to compile and later run, enough memory will have to be available for the program and all its variable allocations. In a

small program such as Listing 1.1, this is not a concern because there should always be more than enough available memory for the few variables being allocated.

Consider the case of having the following declaration in your program:

```
char buffer[1000];
```

When the program is compiled, a requirement of 1000 bytes of memory will be added to the memory requirements of the rest of the program. If the total of the program's memory requirements is not available, the program will not run. When the program runs, 1000 bytes of memory will be set aside to be used as the `buffer` character array. If `buffer` is never used to store information that is 1000 bytes long, memory space is wasted.

Consider a second example such as the following array:

```
char user_name[???];
```

If `user_name` is going to store my first name, "Bradley," it needs to be 8 characters long. Of course, I am assuming that only my first name is going to be stored, but what if my last name is supposed to be included also? If `user_name` is 8 characters long, there isn't room for my full name. In addition, if `user_name` is going to be used to store other names, it will need to be big enough to hold the largest.

Review Tip: Don't forget that "Bradley" takes 8 bytes to store—the eighth byte is for the Null terminator.

A name is a simplistic example because the number of bytes are minimal; however, it easily illustrates the point. Most programmers will set a certain size for a name field and allow the space to be wasted. What if you have an array of names? Listing 1.2 demonstrates how memory can be wasted.

Listing 1.2. A program showing wasted memory.

```
1:   /* Filename:  LIST0102.c
2:    * Author   :  Bradley L. Jones
3:    * Purpose :  This provides an example of allocating memory
4:    *            for an array.
5:    *============================================================*/
6:
7:   #include <stdio.h>
8:
```

continues

Listing 1.2. continued

```
#define MAX 100

void main(void)
{
   char student_name[MAX][35];

   long x;

   printf("\nEnter student names, a blank line will end\n");

   for( x = 0; x < MAX; x++ )
   {
      printf("Enter student %5.5d: ", x+1);
      gets(student_name[x]);

      if( student_name[x][0] == '\0')
         x = MAX;
   }

   printf("\n\nYou entered the following:\n");

   for ( x = 0; student_name[x][0] != '\0' && x < MAX; x++ )
   {
      printf("\nStudent %5.5d:", x+1);
      printf(" %s", student_name[x]);
   }

}
```

Output

```
Enter student names, a blank line will end
Enter student 00001: Connie Crank
Enter student 00002: Deanna Alexander
Enter student 00003: Dawn Johnson
Enter student 00004: Bruce Crouch
Enter student 00005: Sherman Denman
Enter student 00006:


You entered the following:

Student 00001: Connie Crank
Student 00002: Deanna Alexander
Student 00003: Dawn Johnson
Student 00004: Bruce Crouch
Student 00005: Sherman Denman
```

Analysis This program allows you to enter up to MAX number of students. In this case, MAX is defined as 100 in line 9. If you are entering only the number of students in a single class, you may only need 20 to 30, but what if you are entering the names of all the students in the United States? In this case, you are going to need more than 100 students. To increase the value of MAX from 100, you have to modify the code and recompile. This means the program, at present, is only useful to people wanting to enter 100 or fewer names.

There is another problem that can be illustrated with this listing. Change line 9 to define MAX as 100,000. This is not large enough to enter all the students in the United States, but it should handle all the students in a large city. Recompile the program. You may find that you can't recompile! You will probably get an error similar to the following:

```
Error LIST0102.C ##: Array size too large
```

The compiler knows that you are trying to use too much memory. Because of the way the computer addresses memory, you are limited to 64K of memory for data. Later today, you will see how to get around this limitation.

Dynamic Memory Allocation

One way to avoid trying to second guess the number of variables or the amount of mem-ory you require is to dynamically allocate memory. Dynamic allocation of memory means that memory is requested when it is needed. Instead of determining memory requirements at the time the program is compiled, memory is requested as the program is running. This means that if your program doesn't need the memory, the memory is left available. In addition, the program will only need to request the memory it requires. If you are collecting the names of only 352 students, you only reserve memory for the 352 students.

There are several general functions that are used when allocating memory dynamically. To be effective in your use of dynamically allocated memory, you should understand the purpose of each of these. These include:

- `malloc()`
- `free()`
- `realloc()`
- `calloc()`

The *malloc()* Function

Among the more popular functions to allocate memory is `malloc()`. The `malloc()` function enables you to set aside a specified number of bytes of memory. The function prototype for `malloc()` is

```
void *malloc(size_t size);
```

This prototype is found in the STDLIB.H header file of most compilers. This memory allocation function sets aside `size` bytes of memory and returns a pointer to the starting byte of the newly allocated memory. This pointer can then be used as a reference to the allocated memory. If there is not enough memory available, then `NULL` is returned. Listing 1.3 presents an example of `malloc()`'s use.

Listing 1.3. Using the `malloc()` function.

```
1:  /* Filename:  LIST0103.c
2:   * Author  :  Bradley L. Jones
3:   * Purpose :  This provides an example of allocating memory
4:   *            for an array dynamically.
5:   *=========================================================*/
6:
7:  #include <stdio.h>
8:  #include <stdlib.h>
9:
10: void main(void)
11: {
12:    long nbr_students = 0;
13:    long ctr;
14:
15:    char *student_name;
16:    char trash[80];      /* to clear keyboard buffer */
17:
18:    while( nbr_students < 1 || nbr_students > 2000000000 )
19:    {
20:       printf("\nHow many students will be entered? ==> ");
21:       scanf("%ld", &nbr_students);
22:       gets(trash);       /* clear out keyboard buffer */
23:    }
24:
25:    student_name = (char *) malloc( 35*nbr_students);
26:
27:    if( student_name == NULL )    /* verify malloc() was successful
*/
28:    {
29:       printf( "\nError in line %3.3d: Could not allocate memory.",
30:               __LINE__);
31:       exit(1);
32:    }
```

```
33:
34:     for( ctr = 0; ctr < nbr_students; ctr++ )
35:     {
36:        printf("\nEnter student %5.5d: ", ctr+1);
37:        gets(student_name+(ctr*35));
38:     }
39:
40:     printf("\n\nYou entered the following:\n");
41:
42:     for ( ctr = 0; ctr < nbr_students; ctr++ )
43:     {
44:        printf("\nStudent %5.5d:", ctr+1);
45:        printf(" %s", student_name+(ctr*35));
46:     }
47:
48:     /* this program does not release allocated memory! */
49:  }
```

Output

```
How many students will be entered? ==> 5

Enter student 00001: Connie Crank

Enter student 00002: Deanna Alexander

Enter student 00003: Dawn Johnson

Enter student 00004: Bruce Crouch

Enter student 00005: Sherman Denman


You entered the following:

Student 00001: Connie Crank
Student 00002: Deanna Alexander
Student 00003: Dawn Johnson
Student 00004: Bruce Crouch
Student 00005: Sherman Denman
```

Analysis

This program differs from Listing 1.2 in that it leaves it to the person running the program to determine how many names are going to be entered. You don't have to guess how many student names there will be. Lines 20 and 21 prompt the user for the number of names that are going to be entered. Line 25 attempts to use `malloc()` to dynamically allocate the memory. Notice that the program allocates the specific amount requested. If there is not enough memory available, an error message is displayed in line 29; otherwise, the user is allowed to enter the names in lines 34 to 38.

Some other items in this program deserve mentioning. Line 16 declares a character array, or string, called `trash`. This string also could have been dynamically allocated using the `malloc()` function. The `trash` string is used in line 22 to remove any remaining keystrokes that may still be in the keyboard buffer.

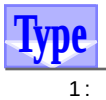
Another point worth mentioning is the defined constant in line 30. The `__LINE__` constant is replaced by the compiler with the current line number. You may be wondering why `__LINE__` was used in line 30 instead of the number `30`. The answer is simple: If the program is changed and recompiled, the line number for the parameter to the `printf()` may change. By using `__LINE__`, the programmer does not need to worry about making a change. If 30 had been used, the line number would need to be manually changed any time the program changed.

Review Tip: The `__LINE__` preprocessor directive is a defined constant. When the program is compiled, it is replaced with the current line number. Another popular preprocessor directive is `__FILE__`. This constant is replaced by the current source file's name.

The *free()* Function

Allocating memory allows memory to be set aside for when it is needed; however, to complete the cycle, the memory should be deallocated, or freed, when it is no longer needed. When a program ends, the operating system is generally able to clean up memory; however, many programs will want to deallocate the memory so that other parts of the program can use it. Listing 1.4 shows what can happen when memory allocations are not cleaned up.

Warning: The following listing may cause your machine to lock up.



Listing 1.4. Allocating memory without freeing it.

```
1:   /* Filename:  LIST0104.c
2:    * Author   :  Bradley L. Jones
3:    * Purpose :  This program shows what happens when dynamically
4:    *            allocated memory is not released.
5:
```

```
*================================================================*/
6:
7:   #include <stdio.h>
8:   #include <stdlib.h>
9:
10:  int do_a_book_page( long );
11:
12:  void main(void)
13:  {
14:     int rv;
15:     unsigned long nbr_pages = 0;
16:     unsigned long page = 0;
17:
18:     printf("\n\nEnter number of pages to do ==> ");
19:     scanf( "%d", &nbr_pages );
20:
21:     for( page = 1; page <= nbr_pages; page++)
22:     {
23:        rv = do_a_book_page( page );
24:
25:        if (rv == 99)
26:        {
27:           printf("\nAllocation error, exiting...");
28:           exit(1);
29:        }
30:     }
31:
32:     printf( "\n\nDid all the pages!\n" );
33:  }
34:
35:  int do_a_book_page( long page_nbr )
36:  {
37:     char *book_page;          /* pointer to assign allocation to */
38:
39:     book_page = (char *) malloc( 1000 );
40:
41:     if( book_page == NULL )
42:     {
43:        printf( "\nError in line %3.3d: Could not allocate memory.",
44:                __LINE__);
45:        return(99);
46:     }
47:     else
48:     {
49:        printf( "\nAllocation for book page %d is ready to use...",
50:                page_nbr);
51:     }
52:
53:     /*********************************************************
```

continues

Listing 1.4. continued

```
54:      * code to get information and assign it at the location  *
55:      * that was previously obtained with malloc().            *
56:      * Code might then write the page to disk or something.   *
57:      ********************************************************/
58:
59:     /*** WARNING: This function does not release allocated memory! ***/
60:
61:     return(0);
62:  }
```

Output

```
Enter number of pages to do ==> 4

Allocation for book page 1 is ready to use...
Allocation for book page 2 is ready to use...
Allocation for book page 3 is ready to use...
Allocation for book page 4 is ready to use...

Did all the pages!
```

Second run:

```
Enter number of pages to do ==> 1000

Allocation for book page 1 is ready to use...
Allocation for book page 2 is ready to use...
Allocation for book page 3 is ready to use...
Allocation for book page 4 is ready to use...
Allocation for book page 5 is ready to use...
Allocation for book page 6 is ready to use...
Allocation for book page 7 is ready to use...
Allocation for book page 8 is ready to use...

                  . . .

Allocation for book page 46 is ready to use...
Allocation for book page 47 is ready to use...
Allocation for book page 48 is ready to use...
Allocation for book page 49 is ready to use...
Allocation for book page 50 is ready to use...
Error in line 044: Could not allocate memory.
Allocation error, exiting...
```

Analysis

Notice that after a certain number of allocations, the program stops. In addition, you may find your entire machine locks up. Now replace the comment in line 59 with the following:

```
free( book_page );
```

When you rerun the program, you should no longer run out of memory to allocate for the book page. When you enter a number such as 1000, the output displays all the way to page 1000.

Notice that there was not a memory allocation error. The `free()` function releases the memory so that it can be reused.

Note: It is good programming practice to always deallocate any memory that is dynamically allocated. Memory allocation errors can be very hard to find in large programs.

DO	DON'T

DO check the return value from `malloc()` to ensure that memory was allocated.

DO avoid using `malloc()` for allocating small amounts of memory. Each memory allocation made with `malloc()` contains some overhead—typically 16 bytes or more.

DON'T forget to free allocated memory with the `free()` function. If you don't free the memory, and the pointer to it goes out of scope, the memory will be unavailable during the execution of the rest of the program.

The *realloc()* Function

In addition to being able to allocate blocks of memory, you may also want to change the size of the memory block. The function `realloc()` was developed for this specific reason. `realloc()` increases or decreases the size of an allocated block of memory. The prototype for `realloc()` as described in the STDLIB.H header file will be similar to the following:

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

The `size` parameter for the `realloc()` function is just like the one in the `malloc()` function—it is the number of bytes that is being requested. The `originalblock` parameter is the pointer to the block of memory that had previously been allocated.

Once the reallocation of memory is completed, the `originalblock` pointer may or may not exist. If the allocation fails, which would occur in the case of a request for a larger block of memory that is unavailable, the pointer will be retained. In this failed case, the `realloc()` function will return a `NULL` pointer. If the allocation is completed successfully, the `originalblock` pointer could be gone. The reason for this is that the reallocation may not be in the same location. If necessary, `realloc()` will move the allocated memory to another location. Listing 1.5 presents an example of `realloc()`.

Listing 1.5. The use of `realloc()`.

```
1:   /* Filename:  LIST0105.c
2:    * Author   :  Bradley L. Jones
3:    * Purpose :  This provides an example of using realloc() to
4:    *            get additional dynamic memory as needed
5:
*=================================================================*/
6:
7:   #include <stdio.h>
8:   #include <stdlib.h>
9:
10:  #define NAME_SIZE  35
11:
12:  void main(void)
13:  {
14:     long student_ctr = 0;
15:     long ctr;
16:     char *student_name = NULL;
17:     while( (student_name =
18:          realloc( student_name,
                    (NAME_SIZE * (student_ctr+1)))) != NULL )
19:     {
20:        printf("\nEnter student %5.5d: ", student_ctr+1);
21:        gets(student_name+( student_ctr * NAME_SIZE));
22:        if( student_name[student_ctr * NAME_SIZE] == NULL )
23:        {
24:
25:           break;
26:        }
27:        else
28:        {
29:           student_ctr++;
30:        }
31:     }
32:
33:     printf("\n\nYou entered the following:\n");
34:
35:     for ( ctr = 0; ctr < student_ctr; ctr++ )
36:     {
37:        printf("\nStudent %5.5d:", ctr+1);
```

```
38:         printf(" %s", student_name+(ctr*NAME_SIZE));
39:     }
40:
41:     free(student_name);
42: }
```

Output

```
Enter student 00001: Mario Andretti

Enter student 00002: A. J. Foyt

Enter student 00003: Rick Mears

Enter student 00004: Michael Andretti

Enter student 00005: Al Unser, Jr.

Enter student 00006:


You entered the following:

Student 00001: Mario Andretti
Student 00002: A. J. Foyt
Student 00003: Rick Mears
Student 00004: Michael Andretti
Student 00005: Al Unser, Jr.
```

Analysis

This program follows the flow of some of the previous listings, except that it reallocates the storage space for the names. Each time there is a new name, a call to `realloc()` (line 18) attempts to increase the size of the `student_name`. Users will be able to enter names until there is not enough memory left to allocate, or until they choose not to enter a name. Notice that `realloc()` is used to allocate the first instance of `student_name`. The first time that line 18 is reached, `student_name` is `NULL`. Passing a `NULL` string as the first parameter of `realloc()` is equivalent to calling `malloc()`.

The *calloc()* Function

The `calloc()` function is quite similar to `malloc()`. There are two differences. The first difference is in the initialization of the allocated memory. When `malloc()` allocates memory, it does not initialize, or clear, the newly allocated memory; whatever was previously stored at the allocated memory location will still be there. With a call to `calloc()`, the allocated memory is cleared by initializing the block with zeros.

The second difference is that `calloc()` allows for an additional parameter. The prototype for `calloc()`, which is in the STDLIB.H header file, should be similar to the following:

```
void *calloc( size_t number_items, size_t block_size );
```

The `block_size` parameter is the same as the size value passed to the `malloc()` function. It is the number of bytes that you want allocated in your memory block. The `number_items` parameter is the number of blocks you want allocated. For example, if you did the following call, the `calloc()` would try to allocate 10 blocks of 100 bytes of memory, or 1000 bytes in total:

```
pointer = calloc( 10, 100 )
```

This would have the same result as a `malloc( 1000 )` in regard to the amount of space allocated. The following two calls would attempt to allocate the same amount of memory. The difference would be that `calloc()` would also initialize the memory to zeros.

```
pointer = malloc( 1000 );
pointer = calloc( 1, 1000);
```

You might be asking why you need the extra parameter. Up to this point, you have been dealing with character data only. With character data, typically, each character requires 1 byte, so it is easy to see from the `malloc()` call how many characters are going to be stored. If you were going to store integers, calling `malloc()` with 1000 does not make it clear that only 500 integers are going to be allocated. In addition, this is making an assumption that an integer will be 2 bytes. Listing 1.6 is an example of how `calloc()` is used:

Listing 1.6. Using `calloc()`.

```
1:   /* Filename:  LIST0106.c
2:    * Author   :  Bradley L. Jones
3:    * Purpose :  This provides an example of allocating memory for an
4:    *            array dynamically with the calloc function.
5:    * Descript:  Program allows grades to be entered before printing
6:    *            an average.
7:    *=================================================================*/
8:
9:   #include <stdio.h>
10:  #include <stdlib.h>
11:
12:  void main( void )
13:  {
14:     int  nbr_grades = 0;
```

```
15:     int  total = 0;
16:     int  ctr;
17:     int  *student_grades;
18:     char trash[80];              /* to clear keyboard buffer */
19:
20:     while( nbr_grades < 1 || nbr_grades >= 10000 )
21:     {
22:        printf("\nHow many grades will be entered? ==> ");
23:        scanf("%ld", &nbr_grades);
24:        gets(trash);              /* clear out keyboard buffer */
25:     }
26:
27:     student_grades = (int *) calloc( nbr_grades, sizeof(int));
28:
29:     if( student_grades == NULL )
30:     {
31:        printf( "\nError in line %3.3d: Could not allocate memory.",
32:                __LINE__);
33:        exit(1);
34:     }
35:
36:     for( ctr = 0; ctr < nbr_grades; ctr++ )
37:     {
38:        printf("\nEnter grade %4.4d: ", ctr+1);
39:        scanf("%d", student_grades+ctr);
40:     }
41:
42:     printf("\n\nYou entered the following:\n");
43:
44:     for ( ctr = 0; ctr < nbr_grades; ctr++ )
45:     {
46:        printf("\nGrade %4.4d:", ctr+1);
47:        printf(" %d", *(student_grades+ctr));
48:
49:        total += *(student_grades+ctr);
50:     }
51:
52:     printf("\n\nThe average grade is: %d\n\n", (total/nbr_grades));
53:
54:     /* Free allocated memory */
55:
56:     free(student_grades);
57:  }
```

Output

```
How many grades will be entered? ==> 5

Enter grade 0001: 100

Enter grade 0002: 80
```

```
Enter grade 0003: 85

Enter grade 0004: 90

Enter grade 0005: 95


You entered the following:

Grade 0001: 100
Grade 0002: 80
Grade 0003: 85
Grade 0004: 90
Grade 0005: 95
The average grade is: 90
```

Analysis This program allows a number of grades to be entered and printed, along with an average grade. This program uses `calloc()` in line 27 in a way that makes the program portable from one computer type to another. The objective of line 27 is to allocate 5 integers. Regardless of what computer this program is compiled on, there will be enough room for the 5 integers. This is accomplished by using the `sizeof` operator to determine the size of an integer. Granted, `malloc()` could be used to do the same thing, but the `calloc()` is clearer. The following is a function call that uses `malloc()` to accomplish the same task:

```
ptr = malloc( sizeof(int) * 5 );
```

Listing 1.6 follows the same flow as the other programs presented so far. Lines 20 through 25 prompt the user for how many grades will be entered. Line 23 uses the `scanf()` function to accept a number. Line 24 clears any remaining information that may be in the keyboard buffer, including the carriage return the user typed when entering the grades in line 23. The `while` statement keeps reprompting the user for the number of grades as long as the number that is entered is less than 1 or greater than 10,000. Line 27 then allocates the space for the grades to be stored.

As should be done with any call to a dynamic memory allocation function, line 29 ensures the space was indeed allocated. If not, an error message is displayed and, in line 33, the program exits. Lines 36 to 40 use a `for` loop to prompt the user to enter the previously specified number of grades. Because the counter is started at 0, when prompting the user (in line 38), 1 is added. A user will be more familiar with the first grade being 1 instead of 0. Line 39 actually gets the grade using `scanf()`. Notice that the grade is placed at a position within the memory previously allocated. The offset into this memory is based on the `ctr` value.

Once all the grades are entered, the program uses an additional `for` loop (lines 44 to 50) to print the grades. As the `for` loop prints each grade, line 49 adds the grade to the total. Notice that in lines 47 and 49 the array is being dereferenced with the asterisk (`*`) to actually obtain the values. (If this dereferencing is confusing, a quick review of pointers, which will be covered on Day 2, should help.) Line 56 is one of the most important lines in the program. Notice that this is where the `student_grade` array is deallocated. At this point, all the memory that was allocated dynamically is released back.

Expert Tip: It does not matter whether you use `malloc()` or `calloc()` to allocate memory dynamically. If you are allocating a single area, then `malloc()` is easiest to use. If you are allocating several items of the same size, then `calloc()` is a better choice.

DO	DON'T

DO use the `sizeof` operator when allocating memory to help make your programs portable, that is, `malloc( sizeof(int) * value).`

DON'T assume that a dynamic allocation function always works. You should always check the return value to ensure the memory was obtained.

DON'T assume that a call to `realloc()` will allocate memory in the same location. If there is not enough memory at the original location, `realloc()` may move the original block to a different location.

Allocating More than 64K

All the functions described so far have a maximum allocation capability. They cannot allocate more than 64K of memory. You saw this in Listing 1.4, which lacked a `free()` statement. The program always stopped before it allocated more than 64 times (64,000 bytes, 1,000 bytes at a time). The way around this limitation is to use functions that allow additional memory to be allocated. This additional memory is termed *far memory*.

Unlike the general memory allocation functions, the far memory allocation functions are not ANSI standard. This means that each compiler may have different functions that perform these allocations. For the Borland compilers, these functions are:

- `farmalloc()`

  Prototype:

  ```
  void far *farmalloc( unsigned long size);
  ```

- `farcalloc()`

  Prototype:

  ```
  void far *farcalloc( unsigned long nbr_units, unsigned long size);
  ```

- `farrealloc()`

  Prototype:

  ```
  void far *farrealloc( void far *old_block_ptr, unsigned long size);
  ```

- `farfree()`

  Prototype:

  ```
  void far *farfree( void far*block_ptr);
  ```

For the Microsoft compiler, the available far functions are:

- `_fmalloc()`

  Prototype:

  ```
  void __far *fmalloc( size_t size);
  ```

- `_fcalloc()`

  Prototype:

  ```
  void __far *_fcalloc( size_t nbr_units, size_t size);
  ```

- `_frealloc()`

  Prototype:

  ```
  void __far *_frealloc( void __far *old_block_ptr, size_t size);
  ```

- `_ffree()`

  Prototype:

  ```
  void __far *_ffree( void __far*block_ptr);
  ```

These functions work nearly identically to the general memory allocation functions; however, there are a few exceptions. First, you must include a different header file. If you are using a Microsoft compiler, you must include the malloc.h header. If you are using a Borland compiler, you include the alloc.h header file. For other compilers, you should consult the library reference section of the manuals. In addition, a difference in the Borland functions is that instead of taking an unsigned integer parameter (`size_t`), they take an unsigned long parameter.

Listing 1.7 is a rewrite of Listing 1.4 using the `farmalloc()` function. Notice that the other listings in this chapter could be similarly modified to use the far functions. For specific usage of your compiler's far memory allocation functions, you should consult your compiler's library reference manual.

Warning: The ability to use the far memory allocations is dependent upon which memory model you are using to compile your programs. Tiny model programs cannot use the far functions. In compact, large, and huge memory models, the far functions are similar to their counterparts, except they take unsigned long parameters. In addition, memory allocated with far functions require that the pointers used to access them be declared as `far`.

Listing 1.7. Use of far memory allocation functions.

```
1:  /* Filename:  LIST0107.c
2:   * Author   :  Bradley L. Jones
3:   * Purpose :  This program shows what happens when memory allocated
4:   *            dynamically with farmalloc() is not released.
5:   *=================================================================*/
6:  #include <alloc.h>
7:  #include <stdio.h>
8:  #include <stdlib.h>
9:
10: int do_a_book_page( long );
11:
12: void main()
13: {
14:    int rv;
15:    unsigned long nbr_pages = 0;
16:    unsigned long page = 0;
17:
18:    printf("\n\nEnter number of pages to do ==> ");
```

continues

Listing 1.7. continued

```
19:     scanf( "%d", &nbr_pages );
20:
21:     for( page = 1; page <= nbr_pages; page++)
22:     {
23:        rv = do_a_book_page( page );
24:
25:        if (rv == 99)
26:        {
27:           printf("\nAllocation error, exiting...");
28:           exit(1);
29:        }
30:     }
31:
32:     printf( "\n\nDid all the pages!\n" );
33:  }
34:
35:  int do_a_book_page( long page_nbr )
36:  {
37:     char far * book_page;      /* pointer to assign allocation to */
38:
39:     book_page = (char *) farmalloc( 1000 );
40:
41:     if( book_page == NULL )
42:     {
43:        printf( "\nError in line %3.3d: Could not allocate memory.",
44:                __LINE__);
45:        return(99);
46:     }
47:     else
48:     {
49:        printf( "\nAllocation for book page %ld is ready to use...",
50:                page_nbr);
51:     }
52:
53:     /*********************************************************
54:      * code to get information and assign it at the location  *
55:      * that was previously obtained with malloc().            *
56:      * Code might then write the page to disk or something.   *
57:      *********************************************************/
58:
59:  /*** WARNING: This function does not release allocated memory! ***/
60:
61:     return(0);
62:  }
```

For Microsoft compilers, replace the following lines:

```
6:      #include <malloc.h>
39:     book_page = (char *) _fmalloc( 1000 );

Enter number of pages to do ==> 4
```

Output

```
Allocation for book page 1 is ready to use...
Allocation for book page 2 is ready to use...
Allocation for book page 3 is ready to use...
Allocation for book page 4 is ready to use...

Did all the pages!
```

Second run:

```
Enter number of pages to do ==> 10000

Allocation for book page 1 is ready to use...
Allocation for book page 2 is ready to use...
Allocation for book page 3 is ready to use...
Allocation for book page 4 is ready to use...
Allocation for book page 5 is ready to use...
Allocation for book page 6 is ready to use...
Allocation for book page 7 is ready to use...
Allocation for book page 8 is ready to use...

                . . .

Allocation for book page 9995 is ready to use...
Allocation for book page 9996 is ready to use...
Allocation for book page 9997 is ready to use...
Allocation for book page 9998 is ready to use...
Allocation for book page 9999 is ready to use...
Allocation for book page 10000 is ready to use...

Did all the pages!
```

Analysis

This program has just a few changes from Listing 1.4. In line 6, a new include file was added for the far allocation function. In line 37, the addition of the `far` keyword for the `book_page` variable enables it to be used to address far memory. The final change is in line 39, where the appropriate far memory allocation function has replaced the `malloc()` function in Listing 1.4.

As with Listing 1.4, after a certain number of allocations, the program stops. As before, you may find that your entire machine locks up. However, as you will see, this program does not run out of memory as quickly as Listing 1.4. With the `farmalloc()` or `_fmalloc()` statement, you have access to much more of the computer's available RAM.

In using the far version of the `malloc()` statement, you had to also make one additional change. In line 6, an additional header file needed to be included. Depending on your

compiler, this may have been either alloc.h or malloc.h.

In Listing 1.4, you replaced the comment that was in line 59 with the following:

```
free( book_page );
```

Because you are using a far memory allocation function, you want to use the appropriate far function in Listing 1.7. For Microsoft compilers, this is `_ffree()`; for Borland compilers, use the `farfree()` function instead.

DO	DON'T
DON'T forget that the far memory allocation functions are not ANSI standard functions. This means they may not be portable from one compiler to another.	
DO use the far free function to allocate memory that was allocated with a far function.	

Summary

Today you learned about the dynamic memory allocation functions. These functions enable you to allocate specific amounts of memory at runtime rather than at compile time. The capability to dynamically allocate memory can help to make more versatile programs. Not only were the general, ANSI standard allocation functions presented, but so were several additional non-ANSI standard functions. The non-ANSI standard functions are used to allocate larger chunks of memory than are allowed by the general functions, such as `malloc()`.

Q&A

Q Why is a topic such as memory allocation covered on Day 1?

A As you begin to write bigger programs and develop more advanced programs, you will find that memory management is of vital importance. As early as Day 2, you will see several advanced concepts that require memory to be allocated dynamically. These include variable length structures and linked lists.

Q What are memory models?

A Memory models deal with the amount of space that is available for a program and its data. When compiling a C program, you must specify to the compiler which memory model is being used. Typically this value—either tiny, small, compact, medium, large, or huge—is set to a default value. Appendix A contains a detailed explanation of the differences among the memory models.

Q **Should I be worried if I did not understand everything in this chapter?**

A No! Although programs as early as Day 2 will be using the dynamic memory functions, these programs will explain what they are doing at the time. As you work on these new programs, the function presented becomes clearer.

Q **What are common reasons for using dynamic memory allocation functions?**

A There are two major uses for dynamic memory allocation functions. The first is to keep a program's initial memory requirements (or allocation) as small as possible. This allows the program to load and execute even if it is under another program's control. Secondly, a major reason for using dynamic memory allocation functions is to assure the memory can be released when it is no longer needed.

Workshop

The Workshop consists of quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. Why is understanding memory important?
2. What is dynamically allocated memory?
3. What is the most memory you can dynamically allocate?
4. How do you release memory that has been dynamically allocated?
5. What is the difference between `malloc()` and `calloc()`?
6. What is the difference between `malloc()` and `farmalloc()`?

Exercises

1. Write a code fragment that dynamically allocates memory to hold 10 integers using the `malloc()` command.
2. Rewrite your answer in Exercise 1 using the `calloc()` function.
3. Write a function to allocate enough memory to hold 20,000 `long` values. This function should return a pointer to the array, or `NULL` if the memory could not be allocated.
4. **BUG BUSTER:** Is there anything wrong with the following code fragment?

```
#include <stdlib.h>
#include <stdio.h>
#define MAX  100
void main(void)
{
    string = malloc( MAX );
    printf( "Enter something: " );
    gets( string );
    puts( string );        /* do something like printing */
    free( string );
}
```

5. **BUG BUSTER:** Is there anything wrong with the following function?

```
/* Day 1: Exercise 5 */
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
   long *long_array;
   long  total = 0;
   int   ctr;

   long_array = calloc( sizeof(long), 10 );
   printf( "Enter 10 numbers: " );
   for( ctr = 0; ctr < 10; ctr++ )
   {
       scanf( "%ld", long_array+ctr );
```

```
        total += *(long_array+ctr);
    }

    printf( "\n\nTotal of numbers is: %ld", total );
}
```

6. **ON YOUR OWN:** Write a program that declares a structure containing a first name, last name, and middle initial. Do not limit the number of names the user is able to enter. Print the names after the user is done entering them. Use a dynamic memory function.

Note: Answers are not provided for the ON YOUR OWN exercises. You are on your own!

Tip: ON YOUR OWN programs typically require rewriting listings presented within the day's materials.

7. **ON YOUR OWN:** Write a function that allocates a given amount of memory and initializes the memory to a specified value. Following is an example prototype and an example call:

   Prototype:

   ```
   void * initialize_memory( void * pointer, size_t size, char
   initializer);
   ```

   Calling the function:

   ```
   char *name;
   /* initialize name to Xs */
   name = (char *) initialize_memory( (char *)name, 35, 'X');
   ```