```
for(size_t i = 0 ; i < total ; i ++)
{
  if(!(i%5U))
    printf("\n");                    /* Newline after every 5         */
  printf ("%12lu", *(primes+i));
}
printf("\n");                        /* Newline for any stragglers    */
```

The `for` loop will output `total` number of primes. The `printf()` that displays each prime value just appends the output to the current line, but the `if` statement outputs a newline character after every fifth iteration, so there will be five primes displayed on each line. Because the number of primes may not be an exact multiple of five, you output a newline after the loop ends to ensure that there's always at least one newline character at the end of the output.

Memory Allocation with the calloc() Function

The `calloc()` function that is declared in the `<stdlib.h>` header offers a couple of advantages over the `malloc()` function. First, it allocates memory as an array of elements of a given size, and second, it initializes the memory that is allocated so that all bits are zero. The `calloc()` function requires you to supply two argument values, the number of elements in the array, and the size of the array element, both arguments being of type `size_t`. The function still doesn't know the type of the elements in the array so the address of the area that is allocated is returned as type `void *`.

Here's how you could use `calloc()` to allocate memory for an array of 75 elements of type `int`:

```
int *pNumber = (int *) calloc(75, sizeof(int));
```

The return value will be `NULL` if it was not possible to allocate the memory requested, so you should still check for this. This is very similar to using `malloc()` but the big plus is that you know the memory area will be initialized to zero.

To make Program 7.11 use `calloc()` instead of `malloc()` to allocate the memory required, you only need to change one statement, shown in bold. The rest of the code is identical:

```
/* Allocate sufficient memory to store the number of primes required */
primes = (unsigned long *)calloc(total, sizeof(unsigned long));
if (primes == NULL)
{
   printf("\nNot enough memory. Hasta la Vista, baby.\n");
   return 1;
}
```

Releasing Dynamically Allocated Memory

When you allocate memory dynamically, you should always release the memory when it is no longer required. Memory that you allocate on the heap will be automatically released when your program ends, but it is better to explicitly release the memory when you are done with it, even if it's just before you exit from the program. In more complicated situations, you can easily have a **memory leak**. A memory leak occurs when you allocate some memory dynamically and you do not retain the reference to it, so you are unable to release the memory. This often occurs within a loop, and because you do not release the memory when it is no longer required, your program consumes more and more of the available memory and eventually may occupy it all.

Of course, to free memory that you have allocated using `malloc()` or `calloc()`, you must still be able to use the address that references the block of memory that the function returned. To release

the memory for a block of dynamically allocated memory whose address you have stored in the pointer pNumber, you just write the statement:

```
free(pNumber);
```

The free() function has a formal parameter of type void *, and because any pointer type can be automatically converted to this type, you can pass a pointer of any type as the argument to the function. As long as pNumber contains the address that was returned by malloc() or calloc() when the memory was allocated, the entire block of memory that was allocated will be freed for further use.

If you pass a null pointer to the free() function the function does nothing. You should avoid attempting to free the same memory area twice, as the behavior of the free() function is undefined in this instance and therefore unpredictable. You are most at risk of trying to free the same memory twice when you have more than one pointer variable that references the memory you have allocated, so take particular care when you are doing this.

Let's modify the previous example so that it uses calloc() and frees the memory at the end of the program.

TRY IT OUT: FREEING DYNAMICALLY ALLOCATED MEMORY

You'll implement this program using pointers and dynamic memory allocation:

```
/* Program 7.11A  Allocating and freeing memory         */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main(void)
{
  unsigned long *primes = NULL;       /* Pointer to primes storage area    */
  unsigned long trial = 0;            /* Integer to be tested              */

  bool found = false;                 /* Indicates when we find a prime    */
  size_t total = 0;                   /* Number of primes required         */
  size_t count = 0;                   /* Number of primes found            */

  printf("How many primes would you like - you'll get at least 4? ");
  scanf("%u", &total);                /* Total is how many we need to find */
  total = total<4U ? 4U:total;        /* Make sure it is at least 4        */

  /* Allocate sufficient memory to store the number of primes required */
  primes = (unsigned long *)calloc(total, sizeof(unsigned long));
  if (primes == NULL)
  {
     printf("\nNot enough memory. Hasta la Vista, baby.\n");
     return 1;
  }

  /* Code to determine the primes as before...*/
```

```
  /* Display primes 5-up */
  for(int i = 0 ; i < total ; i ++)
  {
    if(!(i%5U))
      printf("\n");                    /* Newline after every 5         */
    printf ("%12lu", *(primes+i));
  }
  printf("\n");                        /* Newline for any stragglers    */

  free(primes);                        /* Release the memory         */
  return 0;
}
```

The output from the program will be the same as the previous version, given the same input. Only the two lines in bold font are different from the previous version. The program now allocates memory using `calloc()` with the first argument as the size of type `long`, and the second argument as `total`, which the number of primes required. Immediately before the `return` statement that ends the program, you free the memory that you allocated previously by calling the `free()` function with `primes` as the argument.

Reallocating Memory

The `realloc()` function enables you to reuse memory that you previously allocated using `malloc()` or `calloc()` (or `realloc()`). The `realloc()` function expects two argument values to be supplied: a pointer containing an address that was previously returned by a call to `malloc()`, `calloc()` or `realloc()`, and the size in bytes of the new memory that you want allocated.

The `realloc()` function releases the previously allocated memory referenced by the pointer that you supply as the first argument, then reallocates the same memory area to fulfill the new requirement specified by the second argument. Obviously the value of the second argument should not exceed the number of bytes that was previously allocated. If it is, you will only get a memory area allocated that is equal to the size of the previous memory area.

Here's a code fragment illustrating how you might use the `realloc()` function:

```
long *pData = NULL;                    /* Stores the data                */
size_t count = 0;                      /* Number of data items           */
size_t oldCount = 0;                   /* previous count value           */
while(true)
{
  oldCount = count;                    /* Save previous count value      */
  printf("How many values would you like?  ");
  scanf("%u", &count);                 /* Total is how many we need to find */

  if(count == 0)                       /* If none required, we are done  */
  {
    if(!pData)                         /* If memory is allocated         */
      free(pData);                     /* release it                     */
    break;                             /* Exit the loop                  */
  }

  /* Allocate sufficient memory to store count values */
  if((pData && (count <= oldCount)  /* If there's big enough old memory... */
    pData = (long *)realloc(pData, sizeof(long)*count); /* reallocate it.  */
```

```
  else
  {                                  /* There wasn't enough old memory   */
    if(pData)                        /* If there's old memory...         */
      free(pData);                   /* release it.                      */

    /* Allocate a new block of memory */
    pData = (long *)calloc(count, sizeof(long));
  }
  if (pData == NULL)                 /* If no memory was allocated...    */
  {
     printf("\nNot enough memory.\n");
     return 1;                       /* abandon ship!                    */
  }
  /* Read and process the data and output the result... */
}
```

This should be easy to follow from the comments. The loop reads an arbitrary number of items of data, the number being supplied by the user. Space is allocated dynamically by reusing the previously allocated block if it exists and if it is large enough to accommodate the new requirement. If the old block is not there, or is not big enough, the code allocates a new block using `calloc()`.

As you see from the code fragment, there's quite a lot of work involved in reallocating memory because you typically need to be sure that an existing block is large enough for the new requirement. Most of the time in such situations it will be best to just free the old memory block explicitly and allocate a completely new block.

Here are some basic guidelines for working with memory that you allocate dynamically:

- Avoid allocating lots of small amounts of memory. Allocating memory on the heap carries some overhead with it, so allocating many small blocks of memory will carry much more overhead than allocating fewer larger blocks.
- Only hang on to the memory as long as you need it. As soon as you are finished with a block of memory on the heap, release the memory.
- Always ensure that you provide for releasing memory that you have allocated. Decide where in you code you will release the memory when you write the code that allocates it.
- Make sure you do not inadvertently overwrite the address of memory you have allocated on the heap before you have released it; otherwise your program will have a memory leak. You need to be especially careful when allocating memory within a loop.

Handling Strings Using Pointers

You've used array variables of type `char` to store strings up to now, but you can also use a variable of type "pointer to `char`" to reference a string. This approach will give you quite a lot of flexibility in handling strings, as you'll see. You can declare a variable of type "pointer to `char`" with a statement such as this:

```
char *pString = NULL;
```

At this point, it's worth noting yet again that a pointer is just a variable that can store the address of another memory location. So far, you've created a pointer but not a place to store a string. To store a string, you need to allocate some memory. You can declare a block of memory that you intend to use to store string data and then use pointers to keep track of where in this block you've stored the strings.

String Input with More Control

It's often desirable to read text with more control than you get with the `scanf()` function. The `getchar()` function that's declared in `<stdio.h>` provides a much more primitive operation in that it reads only a single character at a time, but it does enable you to control when you stop reading characters. This way, you can be sure that you don't exceed the memory you have allocated to store the input.

The `getchar()` function reads a single character from the keyboard and returns it as type `int`. You can read a string terminated by `'\n'` into an array, `buffer`, like this:

```
char buffer[100];                          /* String input buffer */
char *pbuffer = buffer;                    /* Pointer to buffer   */
while((*pbuffer++ = getchar()) != '\n');

*pbuffer = '\0';                           /* Add null terminator */
```

All the input is done in the `while` loop condition. The `getchar()` function reads a character and stores it in the current address in `pbuffer`. The address in `pbuffer` is then incremented to point to the next character. The value of the assignment expression, `((*pbuffer++ = getchar())`, is the value that was stored in the operation. As long as the character that was stored isn't `'\n'`, the loop will continue. After the loop ends, the `'\0'` character is added in the next available position. Note that this retains the `'\n'` character as part of the string. If you don't want to do this, you can adjust the address where you store the `'\0'` to overwrite the `'\n'`.

This doesn't prevent the possibility of exceeding the 100 bytes available in the array, so you can use this safely only when you're sure that the array is large enough. However, you could rewrite the loop to check for this:

```
size_t index = 0;
for(; index<sizeof(buffer) ; i++)
  if((*(pbuffer+index) = getchar()) == '\n')
  {
    *(pbuffer + index++) = '\0';
    break;
  }
if( (index ==sizeof(buffer) &&( (*(pbuffer+index-1) != '\0) )
    {
      printf("\nYou ran out of space in the buffer.");
      return 1;
    }
```

The `index` variable indicates the next available element in the `buffer` array. The read operations now take place in a `for` loop that terminates, either when the end of the `buffer` array is reached, or when a `'\n'` character is read and stored. The `'\n'` character is replaced by `'\0'` within the loop. Note that `index` is incremented after `'\0'` is stored. This ensures that `index` still reflects the next available position in `buffer`, although of course, if you fill the buffer, this will be beyond the last element in the array.

When the loop ends, you have to determine why; it could be because you finished reading the string, but it also could be because you ran out of space in `buffer`. When you run out of space, index will be equal to the number of elements in buffer and the last element in buffer will not be a terminating null. Therefore the left operand of the `&&` operation in the `if` expression will be true if you have filled `buffer`, and the right operand will be true if the last element in `buffer` is not a terminating null. It is possible that you read a string that exactly fits, in which case the last element will be a terminating null, in which case the `if` expression will be false, which is the way it should be.

Using Arrays of Pointers

Of course, when you are dealing with several strings, you can use an array of pointers to store references to the strings on the heap. Suppose that you wanted to read three strings from the keyboard and store them in the buffer array. You could create an array of pointers to store the locations of the three strings:

```
char *pS[3] = { NULL };
```

This declares an array, `pS`, of three pointers. You learned in Chapter 5 that if you supply fewer initial values than elements in an array initializer list, the remaining elements will be initialized with 0. Thus just putting a list with one value, `NULL`, will initialize all the elements of an array of pointers of any size to `NULL`.

Let's see how this works in an example.

TRY IT OUT: ARRAYS OF POINTERS

The following example is a rewrite of the previous program, and it demonstrates how you could use an array of pointers to achieve the same result:

```
/* Program 7.12 Arrays of Pointers to Strings */
#include <stdio.h>
const size_t BUFFER_LEN = 512;               /* Size of input buffer     */

int main(void)
{
  char buffer[BUFFER_LEN];                   /* Store for strings        */
  char *pS[3] = { NULL };                    /* Array of string pointers */
  char *pbuffer = buffer;                    /* Pointer to buffer        */
  size_t index = 0;                          /* Available buffer position*/

  printf("\nEnter 3 messages that total less than %u characters.",
                                                        BUFFER_LEN-2);

  /* Read the strings from the keyboard */
  for(int i=0 ; i<3 ; i++)
  {
    printf("\nEnter %s message\n", i>0? "another" : "a" );
    pS[i] = &buffer[index];                  /* Save start of string         */

    /* Read up to the end of buffer if necessary */
    for( ; index<BUFFER_LEN ; index++)       /* If you read \n ... */
      if((*(pbuffer+index) = getchar()) == '\n')
        {
          *(pbuffer+index++) = '\0';         /* ...substitute \0   */
          break;
        }
```

```
    /* Check for buffer capacity exceeded */
    if((index == BUFFER_LEN) && ((*(pbuffer+index-1) != '\0') || (i<2)))
    {
      printf("\nYou ran out of space in the buffer.");
      return 1;
    }
  }


  printf("\nThe strings you entered are:\n\n");
  for(int i = 0 ; i<3 ; i++)
    printf("%s\n", pS[i]);

  printf("The buffer has %d characters unused.\n",
                   BUFFER_LEN-index);
  return 0;
}
```

Here's some sample output from this program:

```
Enter a message
Hello World!

Enter another message
Today is a great day for learning about pointers.

Enter another message
That's all.

The strings you entered are:
Hello World!
Today is a great day for learning about pointers.
That's all.
The buffer has 437 characters unused.
```

How It Works

The first thing of note in this example is that you use the variable `BUFFER_LEN` defined at global scope as the dimension for the array `buffer`:

```
const size_t BUFFER_LEN = 512;                /* Size of input buffer     */
```

The variable must be defined as `const` to allow you to use it as an array dimension; array dimensions can only be specified by constant expressions.

The declarations at the beginning of `main()` are as follows:

```
  char buffer[BUFFER_LEN];                    /* Store for strings        */
  char *pS[3] = { NULL };                     /* Array of string pointers */
  char *pbuffer = buffer;                     /* Pointer to buffer        */
  size_t index = 0;                           /* Available buffer position*/
```

buffer is an array of BUFFER_LEN elements of type char that will hold all the input strings. pS is an array of three pointers that will store the addresses of the strings in buffer. pbuffer is a pointer that is initialized with the address of the first byte in buffer. You'll use pbuffer to move through the buffer array as you fill it with input characters. The index variable records the position of the currently unused element in buffer.

The first for loop reads in three strings. The first statement in the loop is this:

```
printf("\nEnter %s message\n", i>0? "another" : "a" );
```

Here, you use a snappy way to alter the prompt in the printf() after the first iteration of the for loop, using your old friend the conditional operator. This outputs "a" on the first iteration, and "another" on all subsequent iterations.

The next statement saves the address currently stored in pbuffer:

```
pS[i] = pbuffer;                          /* Save start of  string    */
```

This assignment statement is storing the address stored in the pointer pbuffer in an element of the pS pointer array.

The statements for reading the string and appending the string terminator are the following:

```
for( ; index<BUFFER_LEN ; index++)
  if((*(pbuffer+index) = getchar()) == '\n')    /* If you read \n ... */
    {
      *(pbuffer+index++) = '\0';                /* ...substitute \0   */
      break;
    }
```

This is the for loop you saw in the previous section that will read up to the end of buffer if necessary. If a '\n' is read, it is replaced by '\0' and the loop ends. After the loop you check for buffer being full without reaching the end of the string:

```
if((index == BUFFER_LEN) && ((*(pbuffer+index-1) != '\0') || (i<2)))
{
  printf("\nYou ran out of space in the buffer.");
  return 1;
}
```

This is the code you saw explained in the previous section.

By reading the strings using getchar() you have a great deal of control over the input process. This approach isn't limited to just reading strings; you can use it for any input where you want to deal with it character by character. You could choose to remove spaces from the input or look for special characters such as commas that you might use to separate one input value from the next.

```
printf("\nThe strings you entered are:\n\n");
for(int i = 0 ; i<3 ; i++)
  printf("%s\n", pS[i]);
```

In the loop, you output the strings that each of the elements in the pS point to.

If you were to develop this example just a little further, you would be able to allow input of any number of messages, limited only by the number of string pointers provided for in the array.

In the last printf(), you output the number of characters left in the string:

```
printf("The buffer has %d characters unused.\n",
                    BUFFER_LEN-index);
```

Subtracting index from the number of elements in the buffer array gives the number of elements unused.

At the outset, you initialize your pointers to NULL. You can also initialize a pointer with the address of a constant string:

```
char *pString = "To be or not to be";
```

This statement allocates sufficient memory for the string, places the constant string in the memory allocated and, after allocating space for it, sets the value of the pointer pS as the address of the first byte of the string. The problem with this is that there is nothing to prevent you from modifying the string with a statement such as the following:

```
*(pString+3) = 'm';
```

The const keyword doesn't help in this case. If you declare pString as const, it's the pointer that is constant, not what it points to.

TRY IT OUT: GENERALIZING STRING INPUT

Let's try rewriting the example to generalize string input. You can extend the program to read an arbitrary number of strings up to a given limit and ensure that you don't read strings that are longer than you've provided for. Here's the program:

```
/* Program 7.13 Generalizing string input */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const size_t BUFFER_LEN = 128;              /* Length of input buffer    */
const size_t NUM_P = 100;                   /* maximum number of strings */

int main(void)
{
  char buffer[BUFFER_LEN];                  /* Input buffer              */
  char *pS[NUM_P] = { NULL };               /* Array of string pointers */
  char *pbuffer = buffer;                   /* Pointer to buffer         */
  int i = 0;                                /* Loop counter              */

  printf("\nYou can enter up to %u messages each up to %u characters.",
                                                NUM_P, BUFFER_LEN-1);

  for(i = 0 ; i<NUM_P ; i++)
  {
    pbuffer = buffer ; /* Set pointer to beginning of buffer */
    printf("\nEnter %s message, or press Enter to end\n",
                                                          i>0? "another" : "a");

    /* Read a string of up to BUFFER_LEN characters */
    while((pbuffer - buffer < BUFFER_LEN-1) &&
           ((*pbuffer++ = getchar()) != '\n'));

    /* check for empty line indicating end of input */
    if((pbuffer - buffer) < 2)
      break;
```

```
    /* Check for string too long */
    if((pbuffer - buffer) == BUFFER_LEN && *(pbuffer-1)!= '\n')
    {
      printf("String too long - maximum %d characters allowed.",
                                                  BUFFER_LEN);
      i--;
      continue;
    }
    *(pbuffer - 1) = '\0';                  /* Add terminator            */

    pS[i] = (char*)malloc(pbuffer-buffer);  /* Get memory for string     */
    if(pS[i] == NULL)                       /* Check we actually got some…*/
    {
      printf("\nOut of memory - ending program.");
      return 1;                             /* …Exit if we didn't        */
    }

    /* Copy string from buffer to new memory */
    strcpy(pS[i], buffer);
  }

  /* Output all the strings */
  printf("\nIn reverse order, the strings you entered are:\n");
  while(--i >= 0)
  {
     printf("\n%s", pS[i] );            /* Display strings last to first     */
     free(pS[i]);                       /* Release the memory we got         */
     pS[i] = NULL;                      /* Set pointer back to NULL for safety*/
  }
  return 0;
}
```

The output is very similar to the previous two examples:

```
Enter a message, or press Enter to end
Hello

Enter another message, or press Enter to end
World!

Enter another message, or press Enter to end

In reverse order, the strings you entered are:
World!
Hello
```

How It Works

This has expanded a little bit, but there are quite a few extras compared to the original version. You now handle as many strings as you want, up to the number that you provide pointers for, in the array `pS`. The dimension of this array is defined at the beginning, to make it easy to change:

```
const size_t NUM_P = 100;                   /* maximum number of strings */
```

If you want to alter the maximum number of strings that the program will handle, you just need to change the value set for this variable.

At the beginning of `main()`, you have the declarations for the variables you need:

```
char buffer[BUFFER_LEN];                /* Input buffer             */
char *pS[NUM_P] = { NULL };             /* Array of string pointers */
char *pbuffer = buffer;                 /* Pointer to buffer        */
int i = 0;                              /* Loop counter             */
```

The `buffer` array is now just an input buffer that will contain each string as you read it. Therefore, the `#define` directive for `BUFFER_LEN` now defines the maximum length of string you can accept. You then have the declaration for your pointer array of length `NUM_P`, and your pointer, `pbuffer`, for working within `buffer`. Finally, you have a couple of loop control variables.

Next you display a message explaining what the input constraints are:

```
printf("\nYou can enter up to %u messages each up to %u characters.",
                                                NUM_P, BUFFER_LEN-1);
```

The maximum input message length allows for the terminating null to be appended.

The first `for` loop reads the strings and stores them. The loop control is as follows:

```
for(i = 0 ; i<NUM_P ; i++)
```

This ensures that you can input only as many strings as there are pointers that you've declared. Once you've entered the maximum number of strings, the loop ends and you fall through to the output section of the program.

Within the loop, a string is entered using a similar mechanism with `getchar()` to those that you've seen before but with an additional condition:

```
/* Read a string of up to BUFFER_LEN characters */
while((pbuffer - buffer < BUFFER_LEN-1) &&
       ((*pbuffer++ = getchar()) != '\n'));
```

The whole process takes place in the condition for the continuation of the `while` loop. A character obtained by `getchar()` is stored at the address pointed to by `pbuffer`, which starts out as the address of `buffer`. The `pbuffer` pointer is then incremented to point to the next available space, and the character that was stored as a result of the assignment is compared with `'\n'`. If it's `'\n'`, the loop terminates. The loop will also end if the expression `pbuffer-buffer < BUFFER_LEN-1` is false. This will occur if the next character to be stored will occupy the last position in the `buffer` array.

The input process is followed by the check in the following statement:

```
if((pbuffer - buffer) < 2)
  break;
```

This detects an empty line because if you just press the Enter key only one character will be entered: the `'\n'`. In this case, the `break` statement immediately exits the `for` loop and begins the output process.

The next `if` statement checks whether you attempted to enter a string longer than the capacity of `buffer`:

```
if((pbuffer - buffer) == BUFFER_LEN && *(pbuffer-1)!= '\n')
{
  printf("String too long - maximum %d characters allowed.",
                                                BUFFER_LEN);
  i--;
  continue;
}
```

Because you end the `while` loop when the last position in the `buffer` array has been used, if you attempt to enter more characters than the capacity of `buffer`, the expression `pbuffer-buffer` will be equal to `BUFFER_LEN`. Of course, this will also be the case if you enter a string that fits exactly, so you must also check the last character in `buffer` to see if it's `'\n'`. If it isn't, you tried to enter too many characters, so you decrement the loop counter after displaying a message and go to the next iteration.

The next statement is the following:

```
*(pbuffer - 1) = '\0';                  /* Add terminator             */
```

This places the `'\0'` in the position occupied by the `'\n'` character, because `pbuffer` was left pointing to the first free element in the array `buffer`.

Once a string has been entered, you use the `malloc()` function to request sufficient memory to hold the string exactly:

```
pS[i] = (char*)malloc(pbuffer-buffer);  /* Get memory for string      */
if (pS[i] == NULL)                      /* Check we actually got some */
{
  printf("\nOut of memory - ending program.");
  return 0;                             /* …Exit if we didn't         */
}
```

The number of bytes required is the difference between the address currently pointed to by `pbuffer`, which is the first vacant element in `buffer`, and the address of the first element of `buffer`. The pointer returned from `malloc()` is stored in the current element of the `pS` array, after casting it to type `char`. If you get a `NULL` pointer back from `malloc()`, you display a message and end the program.

You copy the string from the buffer to the new memory you obtained using the following statement:

```
strcpy(pS[i], buffer);
```

This uses the library function, `strcpy()`, to copy the contents of `buffer` to the memory pointed to by `pS[i]`. Take care not to confuse the arguments when using the `strcpy()` function; the second argument is the source and the first argument is the destination for the copy operation. Getting them mixed up is usually disastrous, because copying continues until a `'\0'` is found.

Once you exit the loop, either because you entered an empty string or because you used all the pointers in the array `pS`, you generate the output:

```
printf("\nIn reverse order, the strings you entered are:\n");
while(--i >= 0)
{
   printf("\n%s", pS[i] );              /* Display strings last to first */
   free(pS[i]);                         /* Release the memory we got     */
   pS[i] = NULL;                  /* Set pointer back to NULL for safety */
}
```

The index `i` will have a value one greater than the number of strings entered. So after the first loop condition check, you can use it to index the last string. The loop continues counting down from this value and the last iteration will be with `i` at 0, which will index the first string.

You could use the expression `*(pS + i)` instead of `pS[i]`, but using array notation is much clearer.

You use the function, `free()`, after the last `printf()`. This function is complementary to `malloc()`, and it releases memory previously allocated by `malloc()`. It only requires the pointer to the memory allocated as an argument. Although memory will be freed at the end of the program automatically, it's good practice to free memory as soon as you no longer need it. Of course, once you have freed memory in this way, you can't use it, so it's a good idea to set the pointer to `NULL` immediately, as was done here.

■ **Caution** Errors with pointers can produce catastrophic results. If an uninitialized pointer is used to store a value before it has been assigned an address value, the address used will be whatever happens to be stored in the pointer location. This could overwrite virtually anywhere in memory.

TRY IT OUT: SORTING STRINGS USING POINTERS

Using the functions declared in the `string.h` header file, you can demonstrate the effectiveness of using pointers through an example showing a simple method of sorting:

```
/* Program 7.14 Sorting strings */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#define BUFFER_LEN 100                /* Length of input buffer            */
#define NUM_P 100                     /* maximum number of strings         */

int main(void)
{
  char buffer[BUFFER_LEN];            /* space to store an input string    */
  char *pS[NUM_P] = { NULL };         /* Array of string pointers          */
  char *pTemp = NULL;                 /* Temporary pointer                 */
  int i = 0;                          /* Loop counter                      */
  bool sorted = false;                /* Indicated when strings are sorted */
  int last_string = 0;                /* Index of last string entered      */

  printf("\nEnter successive lines, pressing Enter at the"
                " end of each line.\nJust press Enter to end.\n\n");
  while((*fgets(buffer, BUFFER_LEN, stdin) != '\n') && (i < NUM_P))
  {
    pS[i] = (char*)malloc(strlen(buffer) + 1);
    if(pS[i]==NULL)                   /* Check for no memory allocated     */
    {
      printf(" Memory allocation failed. Program terminated.\n");
      return 1;
    }
    strcpy(pS[i++], buffer);
  }
  last_string = i;                    /* Save last string index            */
```

```
  /* Sort the strings in ascending order */
  while(!sorted)
  {
    sorted = true;
    for(i = 0 ; i<last_string-1 ; i++)
      if(strcmp(pS[i], pS[i + 1]) > 0)
      {
        sorted = false;              /* We were out of order             */
        pTemp= pS[i];                /* Swap pointers pS[i]              */
        pS[i] = pS[i + 1];           /*       and                        */
        pS[i + 1]  = pTemp;          /*     pS[i + 1]                    */
      }
  }

  /* Displayed the sorted strings */
  printf("\nYour input sorted in order is:\n\n");
  for(i = 0 ; i<last_string ; i++)
  {
    printf("%s\n", pS[i] );
    free( pS[i] );
    pS[i] = NULL;
  }
  return 0;
}
```

Assuming you enter the same input data, the output from this program is as follows:

```
Enter successive lines, pressing Enter at the end of each line.
Just press Enter to end.

Many a mickle makes a muckle.
A fool and your money are soon partners.
Every dog has his day.
Do unto others before they do it to you.
A nod is as good as a wink to a blind horse.


Your input sorted in order is:

A fool and your money are soon partners.
A nod is as good as a wink to a blind horse.
Do unto others before they do it to you.
Every dog has his day.
Many a mickle makes a muckle.
```

How It Works

This example will really sort the wheat from the chaff. You use the input function `fgets()`, which reads a complete string up to the point you press Enter and then adds `'\0'` to the end. Using `fgets()` rather than the `gets()` function ensures that the capacity of buffer will not be exceeded. The first argument is a pointer to the memory area where you want the string to be stored, the second is the maximum number of character that can be stored, and the third argument is the stream to be read, the standard input stream in this case. Its return value is either the address where the input string is stored—`buffer`, in this case—or `NULL` if an error occurs. Don't forget, `fgets()` differs from

gets() in that it stores the newline character that terminates the input before appending the string terminator '\0' whereas fgets() does not.

The overall operation of this program is quite simple, and it involves three distinct activities:

- Read in all the input strings.
- Sort the input strings in order.
- Display the input strings in alphabetical order.

After the initial prompt lines are displayed, the input process is handled by these statements:

```
while((*fgets(buffer, BUFFER_LEN, stdin) != '\n') && (i < NUM_P))
{
  pS[i] = (char*)malloc(strlen(buffer) + 1);
  if(pS[i]==NULL)                  /* Check for no memory allocated     */
  {
    printf(" Memory allocation failed. Program terminated.\n");
    return 1;
  }
  strcpy(pS[i++], buffer);
}
```

The input process continues until an empty line is entered or until you run out of space in the pointer array. Each line is read into buffer using the fgets() function. This is inside the while loop condition, which allows the loop to continue for as long as fgets() doesn't read a string containing just '\n' and the total number of lines entered doesn't exceed the pointer array dimension. The string just containing '\n' will be a result of you pressing Enter without entering any text. You use the * to get at the contents of the pointer address returned by fgets(). This is the same as dereferencing buffer, of course.

As soon as you collect each input line in buffer, you allocate the correct amount of memory to accommodate the line by using the malloc() function. You get the count of the number of bytes that you need by using the strlen() function and adding 1 for the '\0' at the end. After verifying that you did get the memory allocated, you copy the string from buffer to the new memory using the library function strcpy().

You then save the index for the last string:

```
last_string = i;                    /* Save last string index            */
```

This is because you're going to reuse the loop counter i, and you need to keep track of how many strings you have.

Once you have all your strings safely stowed away, you sort them using the simplest, and probably the most inefficient, sort going—but it's easy to follow. This takes place within these statements:

```
while(!sorted)
{
  sorted = true;
  for(i = 0 ; i < last_string - 1 ; i++)
    if(strcmp(pS[i], pS[i + 1]) > 0)
    {
      sorted = false;              /* We were out of order              */
      pTemp= pS[i];                /* Swap pointers pS[i]               */
      pS[i] = pS[i + 1];           /*       and                         */
      pS[i + 1]  = pTemp;          /*     pS[i + 1]                     */
    }
}
```

The sort takes place inside the `while` loop that continues as long as `sorted` is false. The sort proceeds by comparing successive pairs of strings using the `strcmp()` function inside the `for` loop. If the first string is greater than the second string, you swap pointer values. Using pointers, as you have here, is a very economical way of changing the order. The strings themselves remain undisturbed exactly where they were in memory. It's just the sequence of their addresses that changes in the pointer array, `pS`. This mechanism is illustrated in Figure 7-4. The time needed to swap pointers is a fraction of that required to move all the strings around.

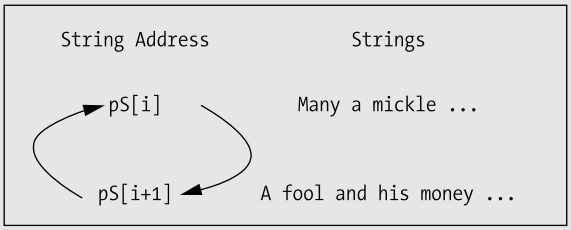

Figure 7-4. *Sorting using pointers*

The swapping continues through all the string pointers. If you have to interchange any strings as you pass through them, you set `sorted` to `false` to repeat the whole thing. If you repeat the whole thing without interchanging any, then they're in order and you've finished the sort. You track the status of this with the `bool` variable `sorted`. This is set to `true` at the beginning of each cycle, but if any interchange occurs, it gets set back to `false`. If you exit a cycle with `sorted` still `true`, it means that no interchanges occurred, so everything must be in order; therefore, you exit from the `while` loop.

The reason this sort is none too good is that each pass through all the items only moves a value by one position in the list. In the worst case, when you have the first entry in the last position, the number of times you have to repeat the process is one less than the number of entries in the list. This inefficient but nevertheless famous method of sorting is known as a **bubble sort**.

Handling strings and other kinds of data using pointers in this way is an extremely powerful mechanism in C. You can throw the basic data (the strings, in this case) into a bucket of memory in any old order, and then you can process them in any sequence you like without moving the data at all. You just change the pointers. You could use ideas from this example as a base for programs for sorting any text. You had better find a better sort of sort, though.

Designing a Program

Congratulations! You made it through a really tough part of the C language, and now I can show you an application using some of what you've learned. I'll follow the usual process, taking you through the analysis and design, and writing the code step by step. Let's look at the final program for this chapter.

The Problem

The problem you'll address is to rewrite the calculator program that you wrote in Chapter 3 with some new features, but this time using pointers. The main improvements are as follows:

- Allow the use of signed decimal numbers, including a decimal point with an optional leading sign, – or +, as well as signed integers.
- Permit expressions to combine multiple operations such as 2.5 + 3.7 - 6/6.

- Add the ^ operator, which will be raised to a power, so 2 ^ 3 will produce 8.
- Allow a line to operate on the previous result. If the previous result is 2.5, then writing =*2 + 7 will produce the result 12. Any input line that starts with an assignment operator will automatically assume the left operand is the previous result.

You're also going to cheat a little by not taking into consideration the precedence of the operators. You'll simply evaluate the expression that's entered from left to right, applying each operator to the previous result and the right operand. This means that the expression

```
1 + 2*3 - 4*-5
```

will be evaluated as

```
((1 + 2)*3 - 4)*(-5)
```

The Analysis

You don't know in advance how long an expression is going to be or how many operands are going to be involved. You'll get a complete string from the user and then analyze this to see what the numbers and operators are. You'll evaluate each intermediate result as soon as you have an operator with a left and a right operand.

The steps are as follows:

1. Read an input string entered by the user and exit if it is `quit`.
2. Check for an = operator, and if there is one skip looking for the first operand.
3. Search for an operator followed by an operand, executing each operator in turn until the end of the input string.
4. Display the result and go back to step 1.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

As you saw earlier in this chapter, the `scanf()` function doesn't allow you to read a complete string that contains spaces, as it stops at the first whitespace character. You'll therefore read the input expression using the `gets()` function that's declared in the `<stdio.h>` library header file. This will read an entire line of input, including spaces. You can actually combine the input and the overall program loop together as follows:

```
/* Program 7.15 An improved calculator */
#include <stdio.h>                          /* Standard input/output  */
#include <string.h>                         /* For string functions   */
#define BUFFER_LEN 256                      /* Length of input buffer */

int main(void)
{
  char input[BUFFER_LEN];                   /* Input expression       */
```

```
  while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
  {
    /* Code to implement the calculator */
  }
  return 0;
}
```

You can do this because the function `strcmp()` expects to receive an argument that's a pointer to a string, and the function `fgets()` actually returns a pointer to the string that the user has typed in—`&input[0]` in this case. The `strcmp()` function will compare the string that's entered with `"quit\n"` and will return 0 if they're equal. This will end the loop.

You set the input string to a length of 256. This should be enough because most computers keyboard buffers are 255 characters. (This refers to the maximum number of characters that you can type in before having to press Enter.)

Once you have your string, you could start analyzing it right away, but it would be better if you removed any spaces from the string. Because the input string is well-defined, you don't need spaces to separate the operator from the operands. Let's add code inside the `while` loop to remove any spaces:

```
/* Program 7.15 An improved calculator */
#include <stdio.h>                          /* Standard input/output  */
#include <string.h>                         /* For string functions   */
#define BUFFER_LEN 256                      /* Length of input buffer */

int main(void)
{
  char input[BUFFER_LEN];                   /* Input expression       */
  unsigned int index = 0; /* Index of the current character in input   */
  unsigned int to = 0;    /* To index for copying input to itself      */
  size_t input_length = 0;       /* Length of the string in input      */
  while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
  {
    input_length = strlen(input);       /* Get the input string length */
    input[--input_length] = '\0';       /* Remove newline at the end   */

    /* Remove all spaces from the input by copy the string to itself */
    /* including the string terminating character                    */
    for(to = 0, index = 0 ; index<=input_length ; index++)
    if(*(input+index) != ' ')               /* If it is not a space    */
      *(input+to++) = *(input+index);       /* Copy the character      */

    input_length = strlen(input);      /* Get the new string length    */
    index = 0;                         /* Start at the first character */

    /* Code to implement the calculator */
  }
  return 0;
}
```

You've added declarations for the additional variables that you'll need. The variable `input_length` has been declared as type `size_t` to be compatible with the type returned by the `strlen()` function. This avoids possible warning messages from the compiler.

The `fgets()` function stores a newline character when you press the Enter to end entry of as line. You don't want the code that analyzes the string to be looking at the newline character, so you

overwrite it with '\0'. The index expression for the input array decrements the length value returned by the strlen() and uses the result to reference the element containing newline.

You remove spaces by copying the string stored in input to itself. You need to keep track of two indexes in the copy loop: one for the position in input where the next nonspace character is to be copied to, and one for the position of the next character to be copied. In the loop you don't copy spaces; you just increment index to move to the next character. The to index gets incremented only when a character is copied. After the loop is entered, you store the new string length in input_length and reset index to reference to the first character in input.

You could equally well write the loop here using array notation:

```
for(to = 0, index = 0 ; index<=input_length ; index++)
  if(input[index] != ' ')                  /* If it is not a space */
    input[to++] = input[index];            /* Copy the character   */
```

For my taste, the code is clearer using array notation, but you'll continue using pointer notation as you need the practice.

Step 2

The input expression has two possible forms. It can start with an assignment operator, indicating that the last result is to be taken as the left operand, or it can start with a number with or without a sign. You can differentiate these two situations by looking for the '=' character first. If you find one, the left operand is the previous result.

The code you need to add next in the while loop will look for an '=', and if it doesn't find one it will look for a substring that is numeric that will be the left operand:

```
/* Program 7.15 An improved calculator */
#include <stdio.h>                /* Standard input/output                    */
#include <string.h>               /* For string functions                     */
#include <ctype.h>                /* For classifying characters               */
#include <stdlib.h>               /* For converting strings to numeric values */
#define BUFFER_LEN 256            /* Length of input buffer                   */

int main(void)
{
  char input[BUFFER_LEN];                     /* Input expression       */
  char number_string[30];        /* Stores a number string from input         */
  unsigned int index = 0;        /* Index of the current character in input   */
  unsigned int to = 0;           /* To index for copying input to itself      */
  size_t input_length = 0;       /* Length of the string in input             */
  unsigned int number_length = 0;    /* Length of the string in number_string */
  double result = 0.0;           /* The result of an operation                */

  while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
  {
    input_length = strlen(input);             /* Get the input string length */
    input[--input_length] = '\0';             /* Remove newline at the end   */

    /* Remove all spaces from the input by copying the string to itself */
    /* including the string terminating character                       */
    for(to = 0, index = 0 ; index<=input_length ; index++)
    if(*(input+index) != ' ')                    /* If it is not a space    */
      *(input+to++) = *(input+index);            /* Copy the character      */
```

```
    input_length = strlen(input);             /* Get the new string length    */
    index = 0;                                /* Start at the first character */

    if(input[index]== '=')                  /* Is there =?                    */
      index++;                              /* Yes so skip over it            */
    else
    {                                       /* No - look for the left operand */
      /* Look for a number that is the left operand for */
      /* the first operator                             */

      /* Check for sign and copy it */
      number_length = 0;                             /* Initialize length      */
      if(input[index]=='+' || input[index]=='-')     /* Is it + or -?          */
        *(number_string+number_length++) = *(input+index++); /* Yes so copy it */

      /* Copy all following digits */
      for( ; isdigit(*(input+index)) ; index++)             /* Is it a digit? */
       *(number_string+number_length++) = *(input+index);   /* Yes - Copy it  */

      /* copy any fractional part */
      if(*(input+index)=='.')                        /* Is it decimal point? */
      { /* Yes so copy the decimal point and the following digits */
        *(number_string+number_length++) = *(input+index++);    /* Copy point */

        for( ; isdigit(*(input+index)) ; index++)           /* For each digit */
             *(number_string+number_length++) = *(input+index);    /* copy it */
      }
      *(number_string+number_length) = '\0';      /* Append string terminator */

      /* If we have a left operand, the length of number_string */
      /* will be > 0. In this case convert to a double so we    */
      /* can use it in the calculation                          */
      if(number_length>0)
        result = atof(number_string);         /* Store first number as result */
    }

    /* Code to analyze the operator and right operand */
    /* and produce the result                         */
  }
  return 0;
}
```

You include the <ctype.h> header for the character analysis functions and the <stdlib.h> header because you use the function atof(), which converts a string passed as an argument to a floating-point value. You've added quite a chunk of code here, but it consists of a number of straightforward steps.

The if statement checks for '=' as the first character in the input:

```
    if(input[index]== '=')                  /* Is there =?                    */
      index++;                              /* Yes so skip over it            */
```

If you find one, you increment index to skip over it and go straight to looking for the operand. If '=' isn't found, you execute the else, which looks for a numeric left operand.

You copy all the characters that make up the number to the array number_string. The number may start with a unary sign, '-' or '+', so you first check for that in the else block. If you find it, then you copy it to number_string with the following statement:

```
if(input[index]=='+' || input[index]=='-')             /* Is it + or -?  */
  *(number_string+number_length++) = *(input+index++); /* Yes so copy it */
```

If a sign isn't found, then index value, recording the current character to be analyzed in input, will be left exactly where it is. If a sign is found, it will be copied to number_string and the value of index will be incremented to point to the next character.

One or more digits should be next, so you have a for loop that copies however many digits there are to number_string:

```
for( ; isdigit(*(input+index)) ; index++)             /* Is it a digit? */
 *(number_string+number_length++) = *(input+index);   /* Yes - Copy it  */
```

This will copy all the digits of an integer and increment the value of index accordingly. Of course, if there are no digits, the value of index will be unchanged.

The number might not be an integer. In this case, there must be a decimal point next, which may be followed by more digits. The if statement checks for the decimal point. If there is one, then the decimal point and any following digits will also be copied:

```
if(*(input+index)=='.')                          /* Is it decimal point? */
{ /* Yes so copy the decimal point and the following digits */
  *(number_string+number_length++) = *(input+index++);    /* Copy point */

  for( ; isdigit(*(input+index)) ; index++)           /* For each digit */
       *(number_string+number_length++) = *(input+index);    /* copy it */
}
```

You must have finished copying the string for the first operand now, so you append a string-terminating character to number_string.

```
*(number_string+number_length) = '\0';      /* Append string terminator */
```

While there may not be a value found, if you've copied a string representing a number to number_string, the value of number_length must be positive because there has to be at least one digit. Therefore, you use the value of number_length as an indicator that you have a number:

```
if(number_length>0)
  result = atof(number_string);         /* Store first number as result */
```

The string is converted to a floating-point value of type double by the atof() function. Note that you store the value of the string in result. You'll use the same variable later to store the result of an operation. This will ensure that result always contains the result of an operation, including that produced at the end of an entire string. If you haven't stored a value here, because there is no left operand, result will already contain the value from the previous input string.

Step 3

At this point, what follows in the input string is very well-defined. It must be an operator followed by a number. The operator will have the number that you found previously as its left operand, or the previous result. This "op-number" combination may also be followed by another, so you have a possible succession of op-number combinations through to the end of the string. You can deal with this in a loop that will look for these combinations:

```
/* Program 7.15 An improved calculator */
#include <stdio.h>                /* Standard input/output                    */
#include <string.h>               /* For string functions                     */
#include <ctype.h>                /* For classifying characters               */
#include <stdlib.h>               /* For converting strings to numeric values */
#define BUFFER_LEN 256            /* Length of input buffer                   */

int main(void)
{
  char input[BUFFER_LEN];              /* Input expression                  */
  char number_string[30];              /* Stores a number string from input */
  char op = 0;                         /* Stores an operator                */

  unsigned int index = 0;        /* Index of the current character in input   */
  unsigned int to = 0;           /* To index for copying input to itself      */
  size_t input_length = 0;       /* Length of the string in input             */
  unsigned int number_length = 0;    /* Length of the string in number_string */
  double result = 0.0;           /* The result of an operation                */
  double number = 0.0;           /* Stores the value of number_string         */

  while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
  {
    input_length = strlen(input);             /* Get the input string length */
    input[--input_length] = '\0';             /* Remove newline at the end   */

    /* Remove all spaces from the input by copying the string to itself */
    /* including the string terminating character                       */
    /* Code to remove spaces as before...                               */

    /* Code to check for '=' and analyze & store the left operand as before.. */

    /* Now look for 'op number' combinations */
    for(;index < input_length;)
    {
      op = *(input+index++);                              /* Get the operator */
      /* Copy the next operand and store it in number */
      number_length = 0;                           /* Initialize the length  */

      /* Check for sign and copy it */
      if(input[index]=='+' || input[index]=='-')            /* Is it + or -? */
      *(number_string+number_length++) = *(input+index++); /* Yes - copy it. */

      /* Copy all following digits */
      for( ; isdigit(*(input+index)) ; index++)            /* For each digit */
       *(number_string+number_length++) = *(input+index);   /* copy it.       */

      /* copy any fractional part */
      if(*(input+index)=='.')                      /* Is it a decimal point? */
      { /* Copy the  decimal point and the following digits */
        *(number_string+number_length++) = *(input+index++);   /* Copy point */
        for( ; isdigit(*(input+index)) ; index++)          /* For each digit */
          *(number_string+number_length++) = *(input+index);   /* copy it.   */
      }
```

```
      *(number_string+number_length) = '\0';             /* terminate string */

      /* Convert to a double so we can use it in the calculation */
      number = atof(number_string);
    }
  /* code to produce result */
  }
  return 0;
}
```

In the interest of not repeating the same code ad nauseam, there are some comments indicating where the previous bits of code that you added are located in the program. I'll list the complete source code with the next addition to the program.

The `for` loop continues until you reach the end of the input string, which will be when you have incremented `index` to be equal to `input_length`. On each iteration of the loop, you store the operator in the variable `op` of type `char`:

```
      op = *(input+index++);                             /* Get the operator */
```

With the operator out of the way, you then extract the characters that form the next number. This will be the right operand for the operator. You haven't verified that the operator is valid here, so the code won't spot an invalid operator at this point.

The extraction of the string for the number that's the right operand is exactly the same as that for the left operand. The same code is repeated. This time, though, the `double` value for the operand is stored in `number`:

```
     number = atof(number_string);
```

You now have the left operand stored in `result`, the operator stored in `op`, and the right operand stored in `number`. Consequently, you're now prepared to execute an operation of the form

```
result=(result op number)
```

When you've added the code for this, the program will be complete.

Step 4

You can use a `switch` statement to select the operation to be carried out based on the operand. This is essentially the same code that you used in the previous calculator. You'll also display the output and add a prompt at the beginning of the program on how the calculator is used. Here's the complete code for the program, with the last code you're adding in bold:

```
/* Program 7.15 An improved calculator */
#include <stdio.h>                /* Standard input/output                    */
#include <string.h>               /* For string functions                     */
#include <ctype.h>                /* For classifying characters               */
#include <stdlib.h>               /* For converting strings to numeric values */
#include <math.h>                 /* For power() function                     */
#define BUFFER_LEN 256            /* Length of input buffer                   */

int main(void)
{
  char input[BUFFER_LEN];              /* Input expression                 */
  char number_string[30];              /* Stores a number string from input */
  char op = 0;                         /* Stores an operator               */
```

```
unsigned int index = 0;        /* Index of the current character in input   */
unsigned int to = 0;           /* To index for copying input to itself      */
size_t input_length = 0;       /* Length of the string in input             */
unsigned int number_length = 0;  /* Length of the string in number_string   */
double result = 0.0;           /* The result of an operation                */
double number = 0.0;           /* Stores the value of number_string         */

printf("\nTo use this calculator, enter any expression with"
                                    " or without spaces");
printf("\nAn expression may include the operators:");
printf("\n          +, -, *, /, %%, or ^(raise to a power).");
printf("\nUse = at the beginning of a line to operate on ");
printf("\nthe result of the previous calculation.");
printf("\nUse quit by itself to stop the calculator.\n\n");

/* The main calculator loop */
while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
{
  input_length = strlen(input);             /* Get the input string length */
  input[--input_length] = '\0';             /* Remove newline at the end   */

  /* Remove all spaces from the input by copying the string to itself */
  /* including the string terminating character                       */
  for(to = 0, index = 0 ; index<=input_length ; index++)
  if(*(input+index) != ' ')                    /* If it is not a space  */
    *(input+to++) = *(input+index);            /* Copy the character    */

  input_length = strlen(input);               /* Get the new string length */
  index = 0;                                 /* Start at the first character */

 if(input[index]== '=')                      /* Is there =?                  */
   index++;                                  /* Yes so skip over it          */
 else
 {                                          /* No - look for the left operand */
   /* Look for a number that is the left operand for the 1st operator */

   /* Check for sign and copy it */
   number_length = 0;                            /* Initialize length      */
   if(input[index]=='+' || input[index]=='-')             /* Is it + or -?  */
     *(number_string+number_length++) = *(input+index++); /* Yes so copy it */

   /* Copy all following digits */
   for( ; isdigit(*(input+index)) ; index++)             /* Is it a digit? */
    *(number_string+number_length++) = *(input+index);   /* Yes - Copy it  */

   /* copy any fractional part */
   if(*(input+index)=='.')                          /* Is it decimal point? */
   { /* Yes so copy the decimal point and the following digits */
     *(number_string+number_length++) = *(input+index++);    /* Copy point */

     for( ; isdigit(*(input+index)) ; index++)           /* For each digit */
       *(number_string+number_length++) = *(input+index);  /* copy it      */
   }
```

```
  *(number_string+number_length) = '\0';      /* Append string terminator */

  /* If we have a left operand, the length of number_string */
  /* will be > 0. In this case convert to a double so we    */
  /* can use it in the calculation                          */
  if(number_length>0)
    result = atof(number_string);         /* Store first number as result */
}

 /* Now look for 'op number' combinations */
 for(;index < input_length;)
 {
   op = *(input+index++);                              /* Get the operator */
   /* Copy the next operand and store it in number */
   number_length = 0;                            /* Initialize the length  */

   /* Check for sign and copy it */
   if(input[index]=='+' || input[index]=='-')           /* Is it + or -?  */
   *(number_string+number_length++) = *(input+index++); /* Yes - copy it. */

   /* Copy all following digits */
   for( ; isdigit(*(input+index)) ; index++)            /* For each digit */
     *(number_string+number_length++) = *(input+index); /* copy it.       */

   /* copy any fractional part */
   if(*(input+index)=='.')                       /* Is it a decimal point? */
   { /* Copy the  decimal point and the following digits */
     /* Copy point     */
     *(number_string+number_length++) = *(input+index++);
     for( ; isdigit(*(input+index)) ; index++)          /* For each digit */
       *(number_string+number_length++) = *(input+index); /* copy it.     */
   }
   *(number_string+number_length) = '\0';             /* terminate string */

   /* Convert to a double so we can use it in the calculation */
   number = atof(number_string);

   /* Execute operation, as 'result op= number' */
   switch(op)
   {
      case '+':                                        /* Addition        */
        result += number;
        break;
      case '-':                                        /* Subtraction     */
        result -= number;
        break;
      case '*':                                        /* Multiplication  */
        result *= number;
        break;
      case '/':                                        /* Division         */
        /* Check second operand for zero */
        if(number == 0)
          printf("\n\n\aDivision by zero error!\n");
```

```
            else
              result /= number;
            break;
          case '%':                          /* Modulus operator - remainder  */
            /* Check second operand for zero */
            if((long)number == 0)
              printf("\n\n\aDivision by zero error!\n");
            else
              result = (double)((long)result % (long)number);
            break;
          case '^':                              /* Raise to a power          */
            result = pow(result, number);
            break;
          default:                           /* Invalid operation or bad input */
            printf("\n\n\aIllegal operation!\n");
            break;
        }
      }
      printf("= %f\n", result);                        /* Output the result */
    }
    return 0;
}
```

The `switch` statement is essentially the same as in the previous calculator program, but with some extra cases. Because you use the power function `pow()` to calculate `resultnumber`, you have to add an `#include` directive for the header file `math.h`.

Typical output from the calculator program is as follows:

```
To use this calculator, enter any expression with or without spaces
An expression may include the operators:
                +, -, *, /, %, or ^(raise to a power).
Use = at the beginning of a line to operate on
the result of the previous calculation.
Use quit by itself to stop the calculator.

2.5+3.3/2
= 2.900000
= *3
= 8.700000
= ^4
= 5728.976100
1.3+2.4-3.5+-7.8
= -7.600000
=*-2
= 15.200000
= *-2
= -30.400000
= +2
= -28.400000
quit
```

And there you have it!

Summary

This chapter covered a lot of ground. You explored pointers in detail. You should now understand the relationship between pointers and arrays (both one-dimensional and multidimensional arrays) and have a good grasp of their uses. I introduced the `malloc()`, `calloc()`, and `realloc()` functions for dynamically allocating memory, which provides the potential for your programs to use just enough memory for the data being processed in each run. You also saw the complementary function `free()` that you use to release memory previously allocated by `malloc()`, `calloc()`, or `realloc()`. You should have a clear idea of how you can use pointers with strings and how you can use arrays of pointers.

The topics I've discussed in this chapter are fundamental to a lot of what follows in the rest of the book, and of course to writing C programs effectively, so you should make sure that you're quite comfortable with the material in this chapter before moving on to the next chapter. The next chapter is all about structuring your programs.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 7-1. Write a program to calculate the average for an arbitrary number of floating-point values that are entered from the keyboard. Store all values in memory that's allocated dynamically before calculating and displaying the average. The user shouldn't be required to specify in advance how many values there will be.

Exercise 7-2. Write a program that will read an arbitrary number of proverbs from the keyboard and store them in memory that's allocated at runtime. The program should then output the proverbs ordered by their length, starting with the shortest and ending with the longest.

Exercise 7-3. Write a program that will read a string from the keyboard and display it after removing all spaces and punctuation characters. All operations should use pointers.

Exercise 7-4. Write a program that will read a series of temperature recordings as floating-point values for an arbitrary number of days, in which six recordings are made per day. The temperature readings should be stored in an array that's allocated dynamically and that's the correct dimensions for the number of temperature values that are entered. Calculate the average temperature per day, and then output the recordings for each day together, with the average on a single line with one decimal place after the point.