8.5 Arrays of pointers to string

In Chapter 7 we learned that an initialized string can be declared, for example, in the following way:

```
char  some_string[]  =  "Some initialization text." ;
```

While processing the above string declaration, the compiler reserves enough memory space for **some_string** so that it can store the given string literal. The string literal is then placed into the memory space of **some_string**. In the above case, the size of **some_string** would be 26 because 25 bytes are needed for the visible characters inside double quotes, and one byte is needed for the terminating NULL character.

In C++, a pointer to type **char** can be considered either as a pointer to a single character or a pointer to a string of characters. It is possible to declare a pointer and initialize it with an address of a string. When we write

```
char*  type_in_your_name_text  =  "Type in your name: " ;
```

we declare a pointer to type **char**, and make it point to the first character of the string literal. In the above declaration, no string space is reserved for storing the characters of the string literal. The ASCII codes of the string literal are placed somewhere among the executable machine instructions. The above declaration reserves memory only to store an address, which means 4 bytes of memory.

Pointers to strings are sometimes useful when the pointers are in an array. The declaration statement

```
char*  array_of_string_pointers[ 10 ] ;
```

specifies an array of pointers to type **char**. This array is actually an array of pointers to string, and it can hold 10 addresses to strings. The above kind of array is not needed very often, but in some programs in this book we shall use an initialized array of this type. For example, the array

```
char*  names_of_days_of_week[]  =

       {"Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday", "Saturday", "Sunday" } ;
```

is an initialized array of pointers to type **char**. The array stores addresses to strings, and therefore it is said to be an array of pointers to string. Every array element is initialized with a memory address. The first array element points to the first character of "Monday", the second array element points to T in "Tuesday", etc. The array has 7 array elements since seven initialization values are given.

Program **months.cpp** demonstrates how an initialized array of pointers to string can be used. These kinds of arrays are useful when your program needs a list of strings where each string is associated with a whole number. Program **months.cpp** has an array of names of months. Each month can also be described with a number, and the number can be used as an index for the array.

The last part of program **months.cpp** shows how an array of pointers to string looks like in the main memory of a computer. By studying the output of **months.cpp**, you can find out that the array of pointers resides in a different memory area than the pointed to strings. The pointed to strings are string literals which are placed in the same memory area where the compiler puts the executable machine instructions. The array of pointers, on the other hand, is located in the memory area where the variables, arrays, and other data of programs generally reside. It is usual that executable machine instructions and data reside in different memory areas when a program is being executed by a computer. Executable machine instructions are not modified during program execution. Because the month names in **months.cpp** also don't need to be modified, they are put to the same memory area as the executable machine instructions.

This loop prints the addresses and contents of the first four elements in the array **names_of_months**. Each element is an address to a month name stored elsewhere in the memory.

This array contains addresses to the string literals that are given inside braces { }. Because this is an initialized array of pointers, there is no size-specifying literal written in brackets [ ]. The array size depends on how many initializers are given inside braces. Because there are twelve months, this array of pointers contains addresses to those twelve strings.

This is a reference to the address of text "July" in the computer's memory. The output stream **cout** prints characters starting from this address until it encounters the NULL (zero) at the end.

```
//  months.cpp

#include  <iostream.h>

int main()
{
   char*  names_of_months[]  =

      { "January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November",
        "December"  }  ;

   cout <<  "\n The first month of year is "
        <<  names_of_months[ 0 ]  <<  "." ;

   cout <<  "\n\n The seventh month, " << names_of_months[ 6 ]
        <<  ", is named after Julius Caesar.\n" ;

   cout << hex << "\n Let's explore the memory: \n" ;

   for ( int month_index  =  0 ;
             month_index  <  4 ;
             month_index  ++  )
   {
      cout<< "\n Address " << (long) &names_of_months[ month_index ]
          << " contains "  << (long)  names_of_months[ month_index ]
          << " (Address of \""  <<    names_of_months[ month_index ]
          << "\")" ;
   }

   cout  <<  "\n" ;
   char*  memory_pointer  =  names_of_months[ 0 ] ;

   while ( *memory_pointer != 'b' )
   {
      cout << "\n Address " << (long) memory_pointer
           << " contains  " << (int) *memory_pointer
           << "   "         <<       *memory_pointer ;
      memory_pointer  ++ ;
   }
}
```

months.cpp - 1.+ Demonstration of an initialized array of pointers to string.

```
char*  memory_pointer  =  names_of_months[ 0 ] ;

while ( *memory_pointer != 'b' )
{
   cout << "\n Address " << (long) memory_pointer
        << " contains  " << (int) *memory_pointer
        << "   "         <<       *memory_pointer ;
   memory_pointer  ++ ;
}
```

This **while** loop prints the string literals in the memory. **memory_pointer** is first set to point to the letter J in "January". The addresses of memory locations and the contents of memory locations are printed until **memory_pointer** points to the letter b in "February".

The address stored in **memory_pointer** must be converted to type **long**. If it were not converted, the output stream would print a string starting from the given address.

The byte in each memory location is printed both in numerical and in character form. The output stream **cout** works so that it recognizes the types of the data items that are output with operator <<. When **cout** receives type **char**, it prints a character corresponding to the received ASCII code. If type **char** is typecast to **int**, the output stream prints the ASCII code in numerical form.

months.cpp - 1 - 1. The last loop of the program.

```
D:\book2cpp>months

 The first month of year is January.

 The seventh month, July, is named after Julius Caesar.

 Let's explore the memory:

 Address 12ff5c contains 41a1a8 (Address of "January")
 Address 12ff60 contains 41a1b0 (Address of "February")
 Address 12ff64 contains 41a1b9 (Address of "March")
 Address 12ff68 contains 41a1bf (Address of "April")

 Address 41a1a8 contains  4a   J
 Address 41a1a9 contains  61   a
 Address 41a1aa contains  6e   n
 Address 41a1ab contains  75   u
 Address 41a1ac contains  61   a
 Address 41a1ad contains  72   r
 Address 41a1ae contains  79   y
 Address 41a1af contains  0
 Address 41a1b0 contains  46   F
 Address 41a1b1 contains  65   e
```

The string data where the pointers point to start in memory location 41A1A8H. This data is constant data which resides in the same memory area where the executable machine instructions are. (Remember that these memory addresses are likely to be different when you run this program on your own computer.

months.cpp - X. Addresses and characters printed together with their addresses.

8.6 Chapter summary

This may be the most technical (and terrifying) C++ chapter of the whole book because the example programs print so many hexadecimal numbers. I believe, though, that all those hexadecimal numbers can help you to understand the nature of pointers. Once you have understood how pointers work, you do not need any hexadecimal numbers to use them. Pointers will be used in some of the example programs in the following chapters. You should return to this chapter if you find something that you do not understand about pointers.

The essential points of this chapter are the following:

- The address operator & can be used to find out what is the memory address of a variable, an array element, or some other type of data item.
- An asterisk * is used both to declare pointers and as an indirection operator to refer to the data items in memory locations pointed to by a pointer. The statement

  ```
  double*  some_pointer ;
  ```

  declares a pointer to type **double**, whereas the expressions

  ```
  *some_pointer
  *(some_pointer + 1)
  ```

  are references to data items in the memory.
- A pointer name alone refers to the memory address stored in the pointer. Supposing that we have declared the pointer

  ```
  int*  pointer_to_integer ;
  ```

  the statement

  ```
  pointer_to_integer  =  &array_of_integers[ 3 ] ;
  ```

  makes **pointer_to_integer** point to the fourth element in **array_of_integers**.
- Operators +, -, ++, and -- can be used with pointers. The effect they have on the address value stored in a pointer depends on the type to which the pointer is declared to point. If a pointer is declared in the following way

  ```
  type*  pointer_name ;
  ```

  the statement

  ```
  pointer_name  ++  ;
  ```

  increments the address stored in the pointer by **sizeof**(type). In the declaration above, type can be **char**, **int**, **long**, **float**, **double**, etc.
- Regardless to whether a pointer is a pointer to **char**, a pointer to **int**, or a pointer to some other type, the pointer always needs 4 bytes (32-bits) of memory. (Table 5-1 and its footnote discuss the sizes of the basic types in C++.)
- On its own, the name of an array refers to the address of the first element of the array. The expressions

  ```
  *some_array
  *( some_array + 1 )
  ```

  refer to the first and second elements of **some_array**.

Exercises with pointers and strings

Exercise 8-4. Write a program that asks for a string from the keyboard, and, by using a pointer variable, explores each character in the given string and counts how many uppercase letters, lowercase letters, numbers, and other characters there are in the given string. Use the function **`getline()`** to read the string from the keyboard. You can use normal integer variables to count different types of characters, but use a pointer to **`char`** to point to the characters in the string. Studying program **ifascii.cpp** in Chapter 6 may help you in this exercise.

Exercise 8-5. Write a program that asks for a string from the keyboard, and then prints the string in "shrinking pieces". For example, if the user types in the string "Hello", the program should print

```
Hello
ello
llo
lo
o
```

The program must have a loop which prints smaller and smaller pieces of the string. You can use the address operator & to print a piece of a string. For example, the output statement

```
cout << &some_string[ 2 ] ;
```

would print **`some_string`** so that the first two characters from the beginning are not printed.

Exercise 8-6. Write a program that asks for a month number from a range of 1 to 12, and prints a sentence according to the given number. For example, if number 8 is given to the program, it must print "August is the month of Emperor Augustus". The program must print a sentence that describes the history of a month. Most of the words for the sentence must be taken from arrays of pointers to string. You can copy the array **`names_of_months`** from **months.cpp**. In addition, you can use the following array

```
char*  history_of_months[] =
{  "month of Roman god Janus",      // January
   "last month in Roman calendar",  // February
   "month of Roman war god Mars",   // March
   "month of Roman goddess Venus",  // April
   "month of goddess Maia",         // May
   "month of Roman goddess Juno",   // June
   "month of Julius Caesar",        // July
   "month of Emperor Augustus",     // August
   "7th Roman month",               // September
   "8th Roman month",               // October
   "9th Roman month",               // November
   "10th Roman month"  } ;          // December
```