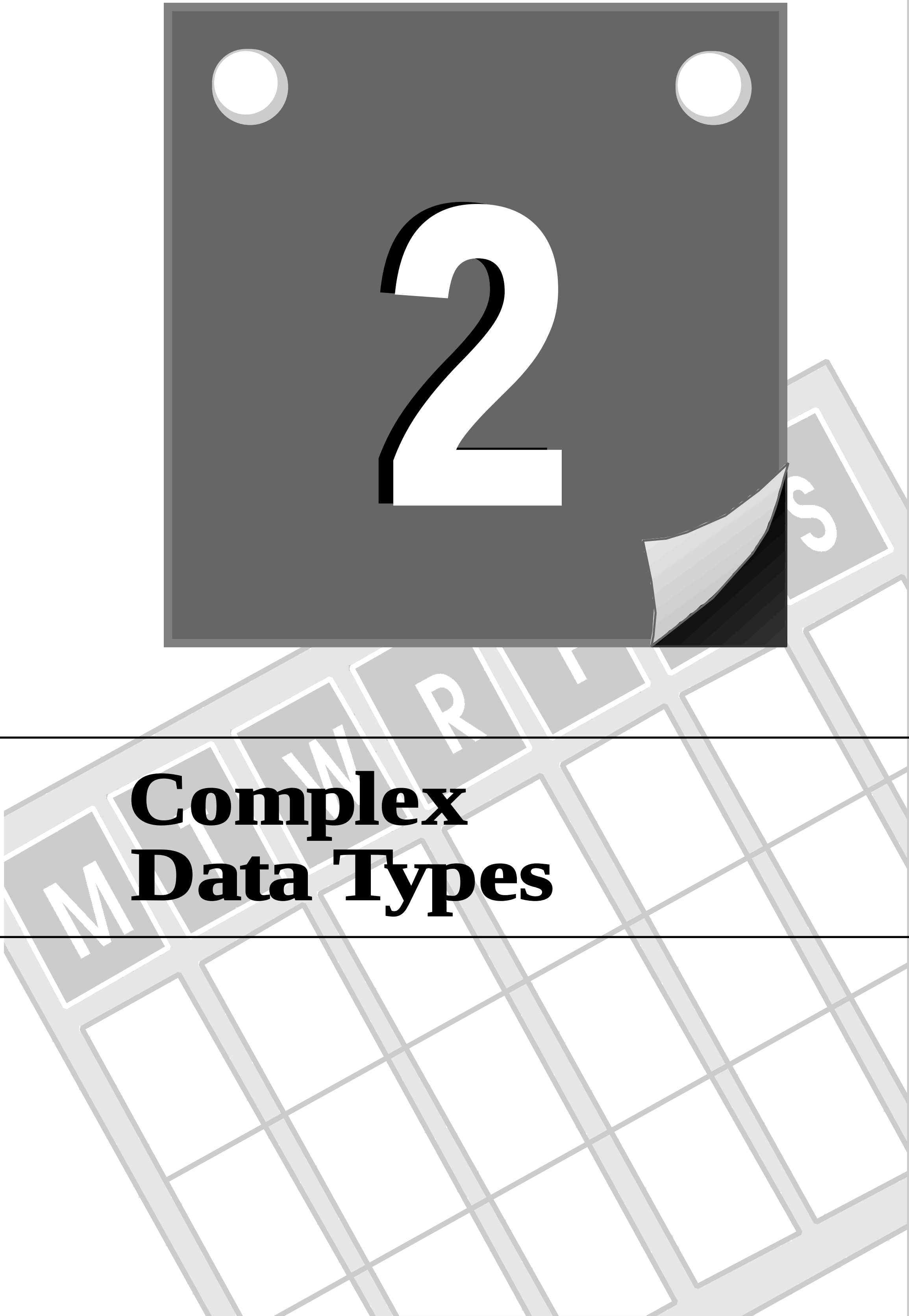


2

Complex Data Types

The basic building blocks of a program are variables. Variables are created with data types. The basic data types are presented in every beginning C book. Today you will go beyond these basic data types. Today you learn:

- ☐ What are considered the basic data types.
- ☐ How to use single and multiple dimensioned arrays.
- ☐ How to use pointers in complex situations, including:
    - ☐ A review of pointers.
    - ☐ Using pointers with other pointers.
    - ☐ Using pointers with functions.
    - ☐ Using pointers with structures.
- ☐ What complex data types can be created with structures and unions.
- ☐ What dynamic or variable-length structures are.

What Are Data Types?

There are many basic data types in C. These are presented in virtually every C book. Several of these data types were used in the listings in Day 1. In addition, virtually every C program will use data types to declare variables. The basic data types are presented in Table 2.1.

Table 2.1. The basic data types.

Data Type	Description
`char`	character
`int`	integers
`short`	short integers
`long`	long integers
`float`	decimal numbers
`double`	larger decimal numbers

The basic data types are used in creating variables, which in turn store information for your programs to use. The data type you use when declaring the variable determines the type of information that can be stored as specified by the description in Table 2.1.

Note: A *variable* is a name given to a location in memory. The data type used when naming, or creating, the variable determines how much space is reserved.

The Values of the Basic Data Types

You should be familiar with the basic data types and the modifiers that can be applied to them from your previous experience with C. Most important are the `signed` and `unsigned` modifiers. Table 2.2 shows the minimum and maximum values that each of the basic data types can hold and their typical memory size in bytes.

Table 2.2. Maximum and minimum values of basic data types.

Variable Type	Size	Minimum	Maximum
signed character	1 byte	–128	128
unsigned character	1 byte	0	255
signed integer	2 bytes	–32,768	32,767
unsigned integer	2 bytes	0	65,535
signed short integer	2 bytes	–32,758	32.767
unsigned short integer	2 bytes	0	65,535
signed long integer	4 bytes	–2,147,483,648	2,147,483,647
unsigned long integer	4 bytes	0	4,294,967,295
float	4 bytes	3.4E-38	3.4E38
double	8 bytes	1.7E-308	1.7E308

Creating Your Own Data Types

C provides a command that enables you to create your own data types. Actually, you don't create them; you simply rename the existing types using the `typedef` keyword. The following presents the `typedef`'s use:

```
typedef existing_type new_type_name;
```

For example:

```
typedef float decimal_number;
```

When the previous type definition is declared, `decimal_number` can be used just like any of the other variable types. For example,

```
char  x;
int   y;
decimal_number z;
```

By using the `typedef` keyword, you can increase the readability of your code. In addition, as you work with the more intricate types, using type definitions can make the code seem a little less complex.

Several new types are generally created using type definitions. The following are some of the type definitions that have been included with a wide variety of compilers:

```
typedef int                BOOL;
typedef unsigned char      BYTE;
typedef unsigned short     WORD;
typedef unsigned int       UINT;
typedef signed long        LONG;
typedef unsigned long      DWORD;
typedef char far*          LPSTR;
typedef const char far*    LPCSTR;
typedef int                HFILE;
typedef signed short int   SHORT;
```

You should notice that the types that have been created here are in all uppercase letters. This is so you don't confuse them with variables. If you have the Microsoft or Borland compilers, you'll find the preceding type definitions in the VER.H file and several other files.

Expert Tip: You should be aware of the preceding type definitions; many advanced programs use them.

Advanced Data Usage: Grouping Data

By using the basic data types, you can create complex or advanced data types. You create new ways of associating and accessing data when you group basic data types. There are three basic groupings of data types that are commonly used in C programs. Because these groups are common, their usage is detailed in most basic C books. They will be briefly reviewed here. The three types of groupings are:

☐ Arrays

☐ Structures

☐ Unions

Arrays

An *array* allows a group of similar data to be accessed using a single variable. This data is all of the same data type. The data type can be one of the basic data types presented earlier, or it can be any of the complex data types presented in this chapter. This even includes other arrays. Listing 2.1 shows how basic arrays can be accessed.

Listing 2.1. Accessing arrays and their data.

```
/* Program:  array.c
 * Author:   Bradley L. Jones
 * Purpose:  Demonstration of accessing arrays and their
 *           data.
 *=========================================================*/

#include <stdio.h>

char text[15] = {‘B’,’R’,’A’,’D’,’L’,’E’,’Y’,’ ‘,
                 ‘J’,’O’,’N’,’E’,’S’, 0, ‘X’ };

int number_array[7] = { 66, 82, 65, 68,
                        76, 69, 89 };

void main( void )
{
   int ctr;

   printf( “\n\nPrint Character array...\n\n”);

   for (ctr = 0; ctr < 15; ctr++)
```

continues

Listing 2.1. continued

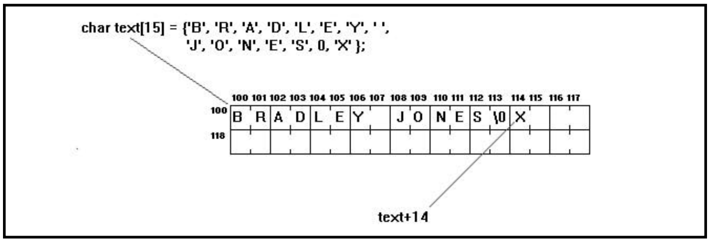
```
    {
        printf( “%c”, text[ctr] );
    }

    printf( “\n\nReprint text with offsets...\n\n”);

    for (ctr = 0; ctr < 15; ctr++)
    {
        printf( “%c”, *(text+ctr));
    }

    printf( “\n\nReprint using string function...\n\n”);
    printf( “%s”, text );

    printf( “\n\nPrint number array...\n\n”);

    for (ctr = 0; ctr < 7; ctr++)
    {
        printf( “%d “, number_array[ctr] );
    }
}
```

Output

```
Print Character array...

BRADLEY JONES X

Reprint text with offsets...

BRADLEY JONES X

Reprint using string function...

BRADLEY JONES

Print number array...

66 82 65 68 76 69 89
```

Analysis

This program shows how single dimension arrays can be defined, preinitialized, and displayed. When an array is created, each element is similar to an individual variable. The difference between individual variables and the array elements is that the array elements have the same name. In addition, array elements are stored one after the other in memory, whereas individual variables could be stored anywhere. Figure 2.1 shows what memory might look like for the `text` character array in Listing 2.1.



```
char text[15] = {'B', 'R', 'A', 'D', 'L', 'E', 'Y', ' ',
                 'J', 'O', 'N', 'E', 'S', 0, 'X' };
```

Figure 2.1. *The* `text` *character array in memory.*

Listing 2.1 should be review. Lines 9 and 10 declare a character array called `text` and initialize it. Lines 12 and 13 declare and initialize a numeric array, `number_array`. These two arrays are then used in the program. Lines 21 through 24 use a `for` loop to print each value of the text character array using a subscript. As you can see, the subscripts in C start with 0 not 1. Other languages, such as BASIC, use 1 as the first subscript.

Review Tip: C uses 0 (not 1!) as the first subscript.

Lines 28 through 31 offer a different method for printing the same character array. Rather than subscripts, a memory offset (pointer) is used. The value of `text` by itself is the address of the first element of the `text` array. By using the dereference operator (`*`), the value at the address stored in `text` (or stated in computerese, *pointed to* by `text`) can be printed. Using Figure 2.1 as a reference, you can see the value of `text` might be 100. The first time line 30 is reached, the value of `text + ctr` is equal to `text`, or 100, because `ctr` is zero. Looking at the `printf()`, you see that the character at address 100 is printed. In the case of `text`, this character is `B`. The next time through the `for` loop, the value pointed to at `text + ctr`—or at 100 + 1—is printed. The value at 101 is `R`. This continues through the end of the `for` loop. If these concepts are new to you, or if you are unsure of what dereferencing is, read the pointer review section that follows later today. If it is still confusing, you may want to consult a beginning C book for a review on pointer basics. *Teach Yourself C in 21 Days* has two chapters on pointers and their use.

Line 34 prints the `text` character array a third way. This time the character array is printed as a string. You should understand the difference between a character array and a string—the difference is sometimes subtle. A *string* is basically a null-terminated character array, which means it ends at the first null character. When line 34 prints, it stops printing at the 15th character—the `x` is not printed. This is because the end of the string is considered to be the null in position 14.

Review Tip: A null character is a character with a numeric value of 0.

The numeric array, `numeric_array`, is printed in lines 38 through 41. As you can see, this is just like the character array only numbers, rather than characters, are being printed.

Structures

Unlike arrays, which allow data of the same type to be grouped together, *structures* allow data of different types to be stored. The following is an example of a structure delcaration.

```
struct date_tag {
    int month;
    char breaker1;
    int day;
    char breaker2;
    int year;
};
```

By grouping different data types, you can see that it's easy to create new types. With the preceding structure, you can declare a structure variable that will hold a date. This single variable will contain a month, day, year, and two breakers. Each of these parts of the declared variable can then be accessed. Listing 2.2 shows the use of a structure.

Listing 2.2. Use of the date structure.

```
/* Program:  STRUCT.C
 * Author:   Bradley L. Jones
 * Purpose:  Program to use a date structure
 *==================================================*/

#include <stdio.h>

struct date_tag {
   int  month;
   char breaker1;
   int  day;
   char breaker2;
   int  year;
} date1;

```

```
void main(void)
{
  struct date_tag date2 = { 1, ‘/’, 1, ‘/’, 1998 };

  printf(“\n\nEnter information for date 1: “);
  printf(“\n\nEnter month: “);
  scanf(“%d”, &date1.month);
  printf(“\nEnter day: “);
  scanf(“%d”, &date1.day);
  printf(“\nEnter year: “);
  scanf(“%d”, &date1.year);

  date1.breaker1 = ‘-’;
  date1.breaker2 = ‘-’;

  printf(“\n\n\nYour dates are:\n\n”);
  printf(“Date 1: %d%c%d%c%d\n\n”, date1.month,
                                   date1.breaker1,
                                   date1.day,
                                   date1.breaker2,
                                   date1.year );

  printf(“Date 2: %d%c%d%c%d\n\n”, date2.month,
                                   date2.breaker1,
                                   date2.day,
                                   date2.breaker2,
                                   date2.year );

  printf(“\n\n\nSize of date_tag structure: %d”,
  sizeof(date1));
}
```

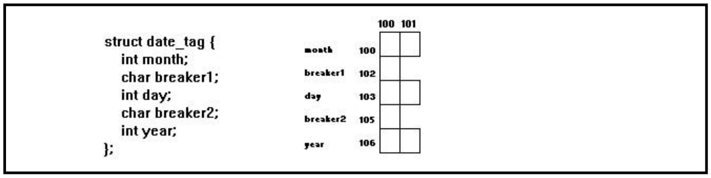
```
Enter information for date 1:

Enter month: 12

Enter day: 25

Enter year: 1994



Your dates are:

Date 1: 12-25-1994

Date 2: 1/1/1998


Size of date_tag structure: 8
```

Analysis This listing should also provide a review. Later today, you will see advanced use of structures—structures with variable lengths. You can see in lines 8 through 14 that the date structure has been defined. In fact, you can see that a variable, `date1`, has been declared. In line 18, another variable, `date2`, is declared using the `struct` keyword along with the previously defined structure's tag, `date_tag`. At the time of `date2`'s declaration, each of its elements is also initialized.

Lines 20 through 26 enable the user to enter the information for `date1`. Using `printf()`, the user is prompted for each of the numeric elements of the structure. Lines 28 and 29 set the breaker values to dashes. These could have been slashes or any other values. Lines 32 through 42 print the values from the individual dates.

Word Alignment

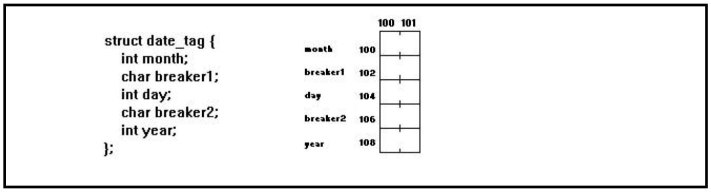
In Listing 2.2, lines 44 and 45 were added to help demonstrate word alignment. How big is the `date` structure? Typically, the size of the structure is the size of its elements added together. In the case of the `date` structure, this would consist of three integers (each two bytes) and two characters (each a single byte). This adds up to a total of eight as the preceding output presented. Figure 2.2 demonstrates the memory placement of each of the date structure's elements.



```
struct date_tag {
    int month;
    char breaker1;
    int day;
    char breaker2;
    int year;
};
```

Figure 2.2. *The date structure in memory with word alignment off.*

You may have gotten the answer of 10 instead of 8 in the output for Listing 2.2. If you did, you compiled with the word-alignment option on. Word alignment is an option that can be set when compiling. If this alignment is on, each element of a structure is stored starting at the beginning of a byte. The exception to this is character elements following other characters. In this case, they can start on an odd byte. A word is the same as an `unsigned short`—two bytes. Check your compiler's compile options to see how to compile the listing with word alignment on and off. Run Listing 2.2 again after compiling each way. Figure 2.3 shows the memory placement for each of the date structure's elements with word alignment on.



```
struct date_tag {
    int month;
    char breaker1;
    int day;
    char breaker2;
    int year;
};
```

Figure 2.3. *The date structure in memory with word alignment.*

DO	**DON'T**

DO capitalize the new types you create with type definitions so that you don't confuse them with variables.

DO understand word alignment. At some point, your data won't seem to be what you think it is. This could be because you are dealing with data that has been aligned differently from how you are accessing it.

DO review a beginning C book, such as *Teach Yourself C in 21 Days,* if you don't understand structures and arrays.

Unions

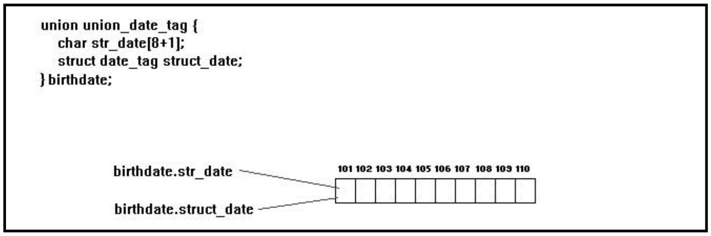
Unions are a special kind of structure. Instead of storing the individual items one after the other, each element is stored in the same location of memory. The following is an example of a union declaration for a date string and the previously declared date structure.

```
union union_date_tag {
    char str_date[8+1];
    struct date_tag struct_date;
} birthdate;
```

Figure 2.4 shows what this looks like in memory.

Each of the values in the union can be accessed; however, only one can be accessed at a time. This is not a concern in the case of the `birthdate` union, but consider a union that has mixed values, as in the following:

```
union mix_tag {
    int number;
    char string[10];
    char float;
}values;
```



```
union union_date_tag {
   char str_date[8+1];
   struct date_tag struct_date;
} birthdate;
```

Figure 2.4. *The birthdate union.*

This union contains values that require different amounts of storage space. Only one of these values can be used at a time. If you have an integer value stored in `number`, you cannot have a value stored in the `string` or the `float` variables. This is because these values share a location in memory. Each of the variables is independent. Listing 2.3 helps to demonstrate the independence of these variables.

Type

Listing 2.3. Unions in memory.

```
/* Program:  UNION.C
 * Author:   Bradley L. Jones
 * Purpose:  Demonstrate a Union
 *=====================================================*/

#include <stdio.h>

   struct date_struct_tag {
      int  month;
      char separater1;
      int  day;
      char separater2;
      int  year;
   };

union date_tag {
   char full_date[8+1];
   struct date_struct_tag date;
};

void main(void)
{

  union date_tag date1;
  union date_tag date2;

  printf(“\n\nEnter string date (1) (Format: MM/DD/YY): “);
  gets(date1.full_date);

  printf(“\n\nEnter information for structure date 2: “);
  printf(“\n\nEnter month: “);
  scanf(“%d”, &date2.date.month);
  printf(“\nEnter day: “);
  scanf(“%d”, &date2.date.day);
  printf(“\n\nEnter year: “);
  scanf(“%d”, &date2.date.year);

  date2.date.separater1 = ‘-’;
  date2.date.separater2 = ‘-’;

  printf(“\n\n\nYour dates are:\n\n”);

  printf(“String - Date 1: %s\n\n”, date1.full_date);

  printf(“Structure - Date 2: %0d%c%0d%c%d\n\n”,
                                   date2.date.month,
                                   date2.date.separater1,
                                   date2.date.day,
                                   date2.date.separater2,
                                   date2.date.year );


  printf(“String - Date 1”
         “(printed as a Structure): %0d%c%0d%c%d\n\n”,
                                    date1.date.month,
                                    date1.date.separater1,
                                    date1.date.day,
                                    date1.date.separater2,
                                    date1.date.year );

  printf(“Structure - Date 2 (printed as String): %s\n\n”,
                                   date2.full_date);
}
```

```
Enter string date (1) (Format: MM/DD/YY): 12/15/63


Enter information for structure date 2:

Enter month: 01

Enter day: 21


Enter year: 1998
```

```
Your dates are:

String - Date 1: 12/15/63

Structure - Date 2: 1-21-1998

String - Date 1 (printed as a Structure): 12849/13617/13110

Structure - Date 2 (printed as a String): J_
```

Analysis The UNION.C program enables you to store a date as either a string date or a date composed of a structure. It uses a union to enable you to store them in the same area of memory.

The structure that is to be used in the union is defined in lines 8 through 14. This structure contains three integer values and two character values. Lines 16 through 19 actually define the union. The character string portion of the union will be called `full_date`. The structure will be called `date`. Notice that this is just a definition of the union. The union is not actually declared until lines 24 and 25. Two unions are declared, `date1` and `date2`. Lines 27 and 28 fill in the first date union, `date1`, with a string value. Lines 30 through 39 fill in the second union, `date2`, by prompting for three integers and then filling in the separators with dashes.

Lines 43 through 50 print the values of `date1` and `date2` so that you can see them. More importantly, lines 53 through 61 reprint the values, but this time the opposite part of the union is used. `date1` was entered as the string part of the date union. Lines 53 through 58 print this string date using the structure portion of the union. Notice that the output doesn't appear to be correct. This is because a string is stored differently than numeric values.

Pointers in Review

Pointers were mentioned previously when talking about printing the character arrays. Because understanding pointers is required for a solid understanding of C, every beginning C book covers them in some detail. A pointer is a special kind of variable. A pointer is a numeric variable used to hold a memory address. When you declare a pointer, you specify the type of variable that is located at the address stored in the pointer variable. The following are the declarations for three different pointers.

Review Tip: A pointer is a numeric variable that is used to hold a memory address. For example:

```
char *char_ptr;
int *int_ptr
struct date_tag *date_ptr;
```

As you can see, pointers are declared like other variables with the exception of the asterisk. `char_ptr` is defined as a pointer to a character, `int_ptr` is a pointer to an integer, and `date_ptr` is a pointer to a `date_tag` structure. You know these are declared as pointers by the asterisks preceding them. (An asterisk in a declaration always means the variable is a pointer.)

As already stated, `char_ptr` is a pointer to a character. The `char_ptr` variable does not hold a character. Instead, it will hold a number, and more exactly `char_ptr` will hold the number of a memory address used to store a character. Figure 2.5 shows a pointer and its variable in memory.

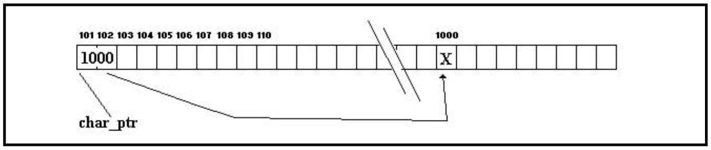


Figure 2.5. *A pointer in memory.*

Finding the Address of a Variable

If a pointer is used to hold a memory address, then it's important to be able to find addresses. In Figures 2.1 through 2.5, you saw how different variables were stored in memory. In most of the examples, the starting memory address was 100. These are simple examples. When variables are actually stored in memory, they can be stored in a variety of locations. In addition, the locations may vary each time the program is run. To truly know where a variable is stored, you need to find its address. C provides, an "address of" operator, which is an ampersand (&), to enable you to determine the address of any variable. Listing 2.4 demonstrates using both a pointer and the "address of" operator.

Listing 2.4. A pointer review.

```
/* Program:  Pointer.c
 * Author:   Bradley L. Jones
 * Purpose:  To show the basics of pointers
 *=================================================*/

#include <stdio.h>

void main(void)
{
   int a_number;
   int *ptr_to_a_number;
   int **ptr_to_ptr_to_a_number;

   a_number = 500;

   ptr_to_a_number = &a_number;

   ptr_to_ptr_to_a_number = &ptr_to_a_number;


   printf(“\n\nThe number is: %d”, a_number);
   printf(“\nThe value of the address of the number is %ld”,
          &a_number);
   printf(“\n\nThe value of the pointer to a number is %ld”,
          ptr_to_a_number);
   printf(“\nThe value of the address of the ptr to a nbr is %ld”,
          &ptr_to_a_number);
   printf(“\n\nThe value of the ptr to a ptr to a nbr is %ld”,
          ptr_to_ptr_to_a_number);

   printf(“\n\n\nThe indirect value of the ptr to a nbr is %d”,
          *ptr_to_a_number);
   printf(“\nThe indirect value of the ptr to a ptr to a nbr is %ld”,
          *ptr_to_ptr_to_a_number);
   printf(“\nThe double indirect value of the ptr to a ptr to a nbr”
           “is %d”,
          **ptr_to_ptr_to_a_number);
}
```

```
The number is: 500
The value of the address of the number is 78053364

The value of the pointer to a number is 78053364
The value of the address of the ptr to a nbr is 78053362
The value of the ptr to a ptr to a nbr is 78053362


The indirect value of the ptr to a nbr is 500
The indirect value of the ptr to a ptr to a nbr is 78053364
The double indirect value of the ptr to a ptr to a nbr is 500
```

Analysis

Listing 2.4 declares three variables. Line 10 declares an integer variable, `a_number`, which is assigned the value of 500 in line 14. Line 11 declares a pointer to an integer called `ptr_to_a_number`. This is assigned a value in line 16. As you can see, it is assigned the address of (&) the previously declared number, `a_number`. The third declared variable, `ptr_to_ptr_to_a_number`, is a pointer to a pointer. This pointer is assigned the address of the pointer, `ptr_to_a_number` (line 18).

Lines 21 through 36 print the values of the previously declared and initialized variables. Line 21 prints the value of the initial integer value, `a_number`. Line 22 prints the address of `a_number` as a long value. Since line 16 assigned this value to `ptr_to_a_number`, it should equal the value printed in line 24. Line 26 prints the address of the pointer, `ptr_to_a_number`. This helps to demonstrate that a pointer is just another variable that is located at an address in memory and contains an individual value. Line 31 prints the indirect value of `ptr_to_a_number`. The indirect value of a pointer is the value at the location stored in the pointer variable. In this case, the value will be 500.

DO **DON'T**

DON'T confuse a pointer's address with its stored value.

Pointers to Pointers

A pointer to another pointer was declared in Listing 2.4. Because a pointer is itself a variable, there is no reason why you can't declare one pointer to hold the address of another pointer. As shown in Listing 2.4, you can declare multiple layers of pointers by simply stacking the asterisks in the declaration. Figure 2.6 shows a variable, `var`, declared in memory with a value of 500. It also shows a pointer, `ptr1`, to this variable. Additionally, the figure shows a pointer, `ptr2ptr`, that points to `ptr1`, which actually makes it a pointer to a pointer. Listing 2.4 presented an example of this using slightly different variable names.

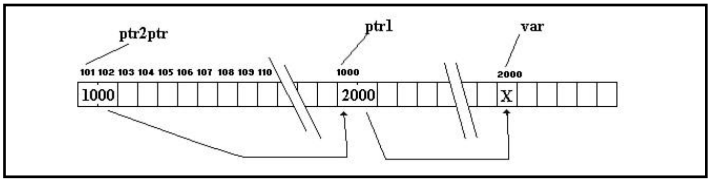


Figure 2.6. *Representation of pointers.*

Pointers to Functions

You can create pointers to functions because they are stored in memory also. These pointers must be declared differently than variable pointers. A declaration for a pointer to a function would be declared as:

```
return_type (*func_pointer)(parameter_list);
```

where `func_pointer` is the name of the function pointer. `parameter_list` is the parameters that are to be passed to the function being pointed to. `return_type` is the data type that the function will return. Listing 2.5 presents a simple example of using a function pointer.

Type

Listing 2.5. Using a function pointer.

```
/* Program:  list0205.c
 * Author:   Bradley L. Jones
 * Purpose:  Demonstrate a function pointer
 *=================================================*/

#include <stdio.h>

void main(void)
{
   int (*func_ptr)();

   func_ptr = printf;

   (*func_ptr)("\n\nHello World!\n\n");
}
```

Output

```
Hello World!
```

Line 10 is the declaration for the function pointer, `func_ptr`. Notice that the parentheses are around the asterisk and the pointer's name, `func_ptr`. If you leave these parentheses off, you get different results:

```
int *func_ptr();
```

This is the same declaration without the parentheses. Instead of being a prototype for a pointer to a function, it is a prototype for a function that returns a pointer to an integer.

You can see in line 12, `printf` was assigned to `func_ptr`. When this assignment was made, the parentheses were left off the `printf`. This was so the address of the `printf` would be passed to `func_ptr` and not the return value a call to `printf()`. Although not necessary, this could also have been done by using the "address of" operator as follows:

```
func_ptr = &printf;
```

Line 14 is the call to the function. Notice that once again the asterisk is used to dereference the function. Because the parentheses to the right have a higher precedence, you need to include an extra set of parentheses around the function name and asterisk. Line 14 is equivalent to calling `printf()`.

The Practical Use of a Function Pointer

There are several occasions when you'll want to pass a function pointer to another function. The `qsort()` function is a prime example of such an instance. The ANSI C function, `qsort()`, expects a pointer to a function to be passed to it. The `qsort()` function sorts the entries in a table by calling a user-defined function. Following is the prototype for `qsort()`:

```
void qsort( void *base, size_t nelem, size_t width,
            int (*fcmp)(const void *, const void *));
```

Notice that the third parameter in this prototype, `fcmp`, is a pointer to a function. Listing 2.6 shows how different functions can be passed in this third parameter.

Listing 2.6. Passing pointers to functions.

```
/* Program:  sort.c
 * Author:   Bradley L. Jones
 * Purpose:  Demonstrate a function pointer being
 *           passed as an argument
 *=================================================*/

#include <stdio.h>
```

continues

Listing 2.6. continued

```
#include <stdlib.h>
#include <string.h>

int sort_a_to_z(const void *first, const void *second);
int sort_z_to_a(const void *first, const void *second);

void main(void)
{
   int  ctr = 0;
   int  total;
   char list[10][256];

   printf(“\n\nPress <Enter> after each word. Enter QUIT to
          end\n”);

   gets(list[ctr]);

   while( stricmp(list[ctr], “QUIT”) != NULL )
   {
      ctr++;
      if(ctr == 10)
        break;

      gets(list[ctr]);
   }
   total = ctr;

   qsort((void *) list, total, sizeof(list[0]), sort_a_to_z);

   printf(“\nThe items sorted A to Z\n”);
   for(ctr = 0; ctr < total; ctr++ )
   {
      printf(“\n%s”, list[ctr]);
   }

   qsort((void *) list, total, sizeof(list[0]), sort_z_to_a);

   printf(“\n\nThe items sorted Z to A\n”);
   for(ctr = 0; ctr < total; ctr++ )
   {
      printf(“\n%s”, list[ctr]);
   }
}

int sort_a_to_z( const void *first, const void *second)
{
   return( strcmp((char*)first, (char *)second) );
}

```

```
56:  int sort_z_to_a( const void *first, const void *second)
57:  {
58:     return( strcmp((char*)second, (char *)first) );
59:  }
```

Output

```
Press <Enter> after each word. Enter QUIT to end
Mississippi
Illinois
Indiana
Montana
Colorado
South Dakota
Florida
California
Alaska
Georgia

The items sorted A to Z

Alaska
California
Colorado
Florida
Georgia
Illinois
Indiana
Mississippi
Montana
South Dakota

The items sorted Z to A

South Dakota
Montana
Mississippi
Indiana
Illinois
Georgia
Florida
Colorado
California
Alaska
```

Analysis

This program enables the user to enter up to ten words or phrases. It then sorts the words into ascending and descending orders using `qsort()`. As you saw in the preceding prototype, `qsort()` takes a pointer to a function as the third parameter. A function name without the parentheses is the address of the function. Two different functions are passed in the SORT.C listing. In line 34, `qsort()` is called with `sort_a_to_z` as the third parameter. In line 42, `sort_z_to_a` is passed. Notice that these functions were prototyped in lines 11 and 12. It was necessary to prototype them before using them.

The rest of this program shouldn't present anything new. In line 20, the user is given a prompt to enter words. In lines 22 through 31, the program gets the information from the user. Line 32 assigns the counter, `ctr`, to `total.ctr`, and now `total`, contains the number of items that were entered. This is necessary for knowing how many items are to be sorted and printed. Lines 36 through 40 and 44 through 48 print the sorted values in the `list` array after each call to `qsort()`.

Complex Data Types (or Complex Data Combinations)

Virtually all of the basic data types presented so far today can be combined. When you combine the data types, you create what could be termed complex data types. These data types aren't really any more complex than the basic types. It's just that you have to be aware of what combinitations you have made. Some of the more common combinations of data types are:

- ☐ Pointers to structures.
- ☐ Arrays of structures.
- ☐ Variable-length structures.

Pointers to Structures

There are two commonly used ways to pass information from one function to another. The first and most common way is *passing by value* in which a value is passed to the called function. The second way, known as *passing by reference*, is to pass a pointer that references the data being passed. When you pass a pointer to a function, the new function has the capability to modify the original data. In addition, less computer resources are used in passing a pointer (reference) than in passing a copy (value). In the case of structures and other large data constructs, this can become important. The amount of space available to pass information is limited. In addition, there are a variety of instances where you'll want to modify the original data and not a copy of the data. Because of this, pointers to data and pointers to structures can be very important. Listing 2.7 presents an example of a pointer to a structure.

Note: An additional use of pointers to structures is in linked lists, which are covered in Day 3.

Listing 2.7. Using a pointer to a structure.

```
/* Program:  list0207.C
 * Author:   Bradley L. Jones
 * Purpose:  Program to demonstrate a pointer to a
 *           structure
 *=====================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct name_tag {
   char last[25+1];
   char first[15+1];
   char middle;
};

char *format_name( struct name_tag *name );

void main(void)
{
  struct name_tag name;

  char input_string[256];
  char *full_name;

  printf(“\nEnter name:”);
  printf(“\n\nFirst name     ==> “);
  gets(input_string);
  strncpy(name.first, input_string, 15);
  name.first[15] = NULL;

  printf(“\nMiddle initial ==> “);
  gets(input_string);
  name.middle = input_string[0];

  printf(“\nLast name      ==> “);
  gets(input_string);
  strncpy(name.last, input_string, 25);
  name.last[25] = NULL;

```

continues

Listing 2.7. continued

```
  full_name = format_name( &name );

  printf(“\n\nThe name you entered: %s”, full_name);
  free(full_name);
}

char *format_name( struct name_tag *name )
{
    char *full_name;
    char tmp_str[3];

    full_name = malloc( 45 * sizeof(char) );

    if( full_name != NULL )
    {
       strcpy( full_name, name->first );
       strcat( full_name, “ “ );

       tmp_str[0] = name->middle;
       tmp_str[1] = ‘.’;
       tmp_str[2] = NULL;

       strcat( full_name, tmp_str);
       if(name->middle != NULL)
          strcat( full_name, “ “);

       strcat( full_name, name->last );
    }

    return(full_name);
}
```

Output

```
Enter name:

First name     ==> Bradley

Middle initial ==> Lee

Last name      ==> Jones


The name you entered: Bradley L. Jones
```

Analysis This program enables you to enter information into a name structure. The structure, `name_tag`, holds a first, middle, and last name. This structure is declared in lines 11 through 15. Once the name is entered in lines 26 to 39, the name is then formatted into the `full_name` variable. The function, `format_name()`, is used to do the formatting. This function was prototyped in line 17. `format_name()` receives a pointer to a structure as its only parameter and returns a character pointer for the name string. Once the name is formatted, line 43 prints it. Line 44 concludes the program by freeing the space that was allocated for the name by `format_name()`.

Lines 47 through 71 contain the `format_name()` function. This function formats the name into the format of first name, space, middle initial, period, space, last name. Line 52 allocates memory with the `malloc()` command. (This should be familiar from Day 1.) Line 54 verifies that memory was allocated for the full name. If it was, the members of the name structure are formatted. Notice that this function uses the indirect membership operator (`->`). This function could have used indirection. The following would be an equivalent to line 59:

```
tmp_str[0] = (*name).middle;
```

Notice that there are parentheses around the structure pointer. This is because the member operator (`.`) has a higher precedence than the indirection operator (`*`).

Arrays of Structures

Arrays of structures are also commonly used. An array of structures operates just like an array of any other data type. Listing 2.8 is an expansion of Listing 2.7. This listing allows multiple names to be entered and placed into an array of name structures.

Type **Listing 2.8. Using an array of structures.**

```
1:   /* Program:  list0208.C
2:    * Author:   Bradley L. Jones
3:    * Purpose:  Program to demonstrate an array of
4:    *           structures
5:    *=====================================================*/
6:
7:   #include <stdio.h>
8:   #include <stdlib.h>
9:   #include <string.h>
10:
11:  #define NAMES  2
12:
```

continues

Listing 2.8. continued

```
struct name_tag {
   char last[25+1];
   char first[15+1];
   char middle;
};

char *format_name( struct name_tag *name );

void main(void)
{
  struct name_tag names[NAMES];

  int  ctr = 0;
  char input_string[256];
  char *full_name;

  printf(“\n\nThis program allows you to enter %d names.”, NAMES);
  for( ctr = 0; ctr < NAMES; ctr++)
  {
     printf(“\nEnter name %d.”, ctr+1);
     printf(“\n\nFirst name     ==> “);
     gets(input_string);
     strncpy(names[ctr].first, input_string, 15);
     names[ctr].first[15] = NULL;

     printf(“\nMiddle initial ==> “);
     gets(input_string);
     names[ctr].middle = input_string[0];

     printf(“\nLast name      ==> “);
     gets(input_string);
     strncpy(names[ctr].last, input_string, 25);
     names[ctr].last[25] = NULL;
  }

  printf(“\n\nThe names: \n”);

  for( ctr = 0; ctr < NAMES; ctr++ )
  {
     full_name = format_name( &names[ctr] );
     printf(“\nName %d: %s”, ctr+1, full_name);
     free(full_name);
  }
}

char *format_name( struct name_tag *name )
{
    char *full_name;
    char tmp_str[3];

   full_name = malloc( 45 * sizeof(char) );

   if( full_name != NULL )
   {
      strcpy( full_name, name->first );
      strcat( full_name, " " );

      tmp_str[0] = name->middle;
      tmp_str[1] = '.';
      tmp_str[2] = NULL;

      strcat( full_name, tmp_str);
      if(name->middle != NULL)
         strcat( full_name, " ");

      strcat( full_name, name->last );
   }

   return(full_name);
}
```

Output

```
This program allows you to enter 2 names.
Enter name 1.

First name     ==> Boris

Middle initial ==>

Last name      ==> Yeltsin

Enter name 2.

First name     ==> Bill

Middle initial ==> E

Last name      ==> Clinton


The names:

Name 1: Boris Yeltsin
Name 2: Bill E. Clinton
```

Analysis

As stated before, this listing is quite similar to Listing 2.7. The listing allows names to be entered into an array. When all the names are entered, they are formatted and printed.

The difference in this listing should be easy to follow. In line 11, a defined constant, `NAMES`, is declared. This constant contains the number of names that will be entered. By using a defined constant, it becomes easy to change the number of names to be entered. Line 23 is also different. Instead of just declaring a name structure, an array is being declared. Lines 30 and 50 contain `for` loops that are used to get the multiple occurrences of names.

Like regular arrays, structure arrays will also use subscripts to access each element. This can first be seen in line 35. Notice that the subscript goes on the structure name, `names`, not the member name. Other than adding the subscript, this program will operate similar to Listing 2.7.

Variable-Length Structures

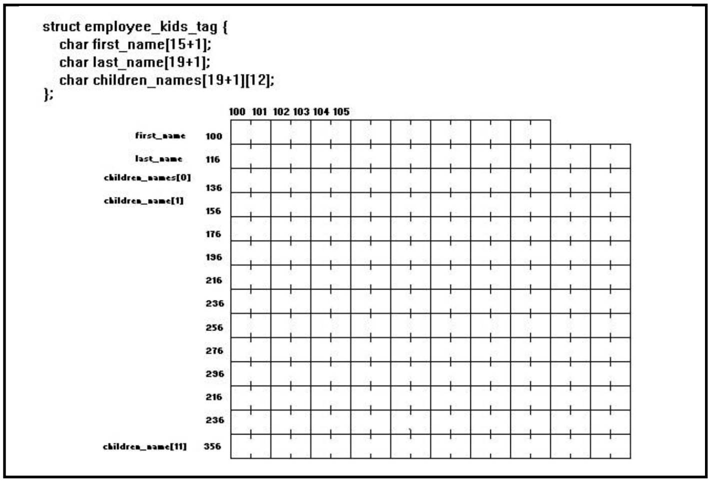
Variable-length structures are simply structures that can vary in size. In a normal structure, such as the following, you can determine the size:

```
struct employee_kids_tag{
    char first_name[15+1];
    char last_name [19+1];
    char child1_name[19+1];
    char child2_name[19+1];
};
```

You should be able to determine that variables declared with the `employee_kids_tag` will be 76 characters long. This is calculated by adding the sizes of the individual elements, plus any byte alignment that may occur. The `employee_kids_tag` structure would work well for all the employees who have two children or fewer, but what if John Smith has 12 kids? You wouldn't be able to store all his kids' names in a single structure. There are several solutions to get around this. The first is to modify the structure to declare 12 names for the children rather than 2. The following code shows this being done with an array instead of individual child name variables:

```
struct employee_kids_tag{
    char first_name[15+1];
    char last_name [19+1];
    char children_names[19+1][12];
};
```

This structure enables you to store the first and last name of the employee along with up to 12 children's names. As long as the array size is big enough to hold the most children that any employee has, this structure works; however, this is typically not an optimal solution. Figure 2.7 shows what the memory usage of this structure would be when a variable is declared. Notice that there is a great deal of memory allocated. If most of the employees have only one or two kids, the majority of this allocated space will never be used.



```
struct employee_kids_tag {
   char first_name[15+1];
   char last_name[19+1];
   char children_names[19+1][12];
};
```

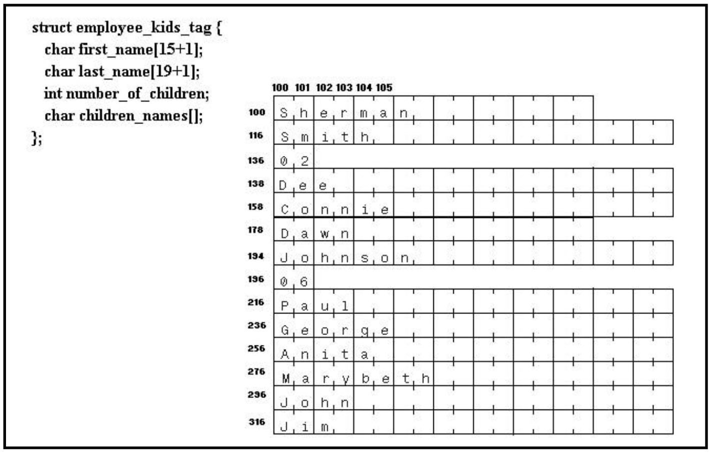
Figure 2.7. *The* `employee_kid_tag` *structure for 12 kids—potentially a lot of wasted space.*

Variable-length structures offer a solution to this type of problem by enabling you to change the number of elements in the array. If employee Sherman Smith has two kids, you declare the array with two elements. If Dawn Johnson has six kids, you declare the structure with six elements. Figure 2.8 shows how the memory usage should be.

As you can see in Figure 2.8, memory isn't completely conserved because we have the spaces at the end of each name; however, we did save space by only storing names when we have to.

Declaring the structure so that there can be different numbers of children would be done as follows:

```
struct employee_kids_tag{
    char first_name[15+1];
    char last_name [19+1];
    int  number_of_children;
    char child_name[];
};
```



```
struct employee_kids_tag {
   char first_name[15+1];
   char last_name[19+1];
   int number_of_children;
   char children_names[];
};
```

Figure 2.8. *Variable-length structures—conserving memory usage.*

Notice that two things have changed. First, we added an integer, `number_of_children`, stating how many children there are. Without this, we would have a much harder time knowing how many kids there are, and thus how big the structure is. The second change is in the `child_name` array. There isn't a number in the array! This signals that it is a variable-length array. It will be up to you to determine how big it is. (Hence the previously mentioned `number_of_children`.)

Warning. Some compilers won't allow you to declare an array without a value. In these cases, you should leave the last member, `child_name[]`, out of the structure.

When Are Variable-Length Structures Used?

You might be thinking that it would be much easier to use a consistent number of array elements. As long as you selected a size that meets a majority of your needs, you could be happy. While this may suffice, there are many instances that you may find that you need to use the variable-length structures. Although small or simple programs can afford the luxury of wasting space, more complex programs cannot. In addition, many programs that work with disk files require that you work with variable-length structures.

There are several examples of programs that use variable-length structures. Calendar programs, programs that modify executable (EXE) files, programs that work with bit-mapped graphics files, and word processor programs are just a few.

A Variable-Length Structure Example

The best way to understand the usage of variable-length files is to use them. The following example will use a variable-length structure to create a journal entry. The format of the journal entry will be as follows:

```
struct journal_entry_tag {
     int text_size;
     char text_entry[];
}
```

As you can see, this is a relatively simple structure. The first member of the structure is an integer that tells the size of the following character array, `text_entry`. Listing 2.9 uses this structure.

Listing 2.9. Program using variable-length structure.

```
/*  Program: list0209.c
 *  Author:  Bradley L. Jones
 *  Purpose: Demonstrates a variable length file.
 *=================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
   int text_size;
   char text_entry[];
} JOURNAL_ENTRY;

void main( void )
{
     JOURNAL_ENTRY entry[10];

     char  buffer[256];
     int   ctr;

     FILE *out_file;

     if(( out_file = fopen( “TMP_FILE.TXT”, “w+”)) == NULL )
     {
        printf(“\n\nError opening file.”);
```

continues

Listing 2.9. continued

```
         exit(99);
      }

      printf(“\n\nYou will be prompted to enter 10 strings.\n”);

      for( ctr = 0; ctr < 10; ctr++ )
      {
         printf(“\nEnter string %d:\n”, ctr+1);
         gets(buffer);
         entry[ctr].text_size = strlen(buffer);
         fwrite( &entry[ctr].text_size, 1, sizeof( int), out_file);
         fwrite( buffer, 1, entry[ctr].text_size, out_file);
      }

      printf(“\n\nTo view your file, type the following:”);
      printf(“\n\n   TYPE TMP_FILE.TXT”);

      fclose(out_file);
  }
```

Output

```
You will be prompted to enter 10 strings.

Enter string 1:
aaaa

Enter string 2:
BBBBBBBBBBBB

Enter string 3:
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

Enter string 4:
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD

Enter string 5:
EEE

Enter string 6:
FFFFFFFFFFFFFFFFFFFFF

Enter string 7:
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG

Enter string 8:
HHHHHHHH
```

```
Enter string 9:
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII

Enter string 10:
JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ


To view your file, type the following:

   TYPE TMP_FILE.TXT
```

Typing the TMP_FILE.TXT file displays the following:

```
__aaaa__BBBBBBBBBBB__CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC__
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
__EEE__FFFFFFFFFFFFFFFFFFF__GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG__
HHHHHHH__IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII__
JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
```

You should note that the underscores in the output from typing `TMP_FILE.TXT` are actually numeric values. These will appear as unusual symbols on your screen. This listing isn't exactly clear on using the variable-length structure. To accurately demonstrate variable-length structures will take several pages of code. Later in this book, several variable-length structures will be used. The previous output demonstrates how a file can be created that applies to a simplistic variable-length structure. You could easily reverse this program so that it reads the file that was created. You could read each of these into the `JOURNAL_ENTRY` structures. You would need to dynamically allocate space for the character array within the structure.

This program presents some interesting items. In line 22, a file pointer is declared. This pointer is used in line 24 to point to the TMP_FILE.TXT file. This is the file for the variable-length journal entries. Lines 37 and 38 write the information out to the file. In line 37, the portions of the structure that are constant in size are written. In this case, it is a single field. In line 38, the variable length portion is written out. An exercise at the end of this chapter asks you to write a program that reads this file and prints out the information.

Summary

Although a lot of what was presented today should have been familiar, some of the material might have been new to you. Today began with a review of the basic data types, the definition of the `typedef` statement and its use, followed by advanced data types (or advanced groupings of data types). This included working with arrays, structures, and unions. A review of pointers was also provided—simple pointers, pointers to pointers, pointers to functions, and pointers to structures. The day moved to arrays of structures before concluding with variable-length structures.

Q&A

Q Can you program in C without fully understanding pointers?

A Yes; however, you'll be extremely limited in what you'll be able to do. Pointers are found in virtually every real-world application.

Q How many dimensions can an array have?

A The number of levels an array can have is compiler-dependent. It becomes more confusing to use multidimensional arrays the more levels you use. Most applications rarely need more than three levels.

Q How many levels of indirection can you have with pointers?

A As with arrays, you don't want too many levels of indirection. You rarely go more than three levels deep with pointers (pointer to a pointer to a pointer). Although going two levels deep is not unusual (pointer to pointer), anything more should be avoided if possible.

Workshop

The Workshop provides 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. What are considered the basic data types?
2. What C keyword can be used to help increase the readability of your code?
3. What is the data type of a variable declared as a DWORD?
4. What are three ways of grouping data?
5. What is the value of NULL or the Null character?
6. Why is it important to know about word alignment?

7. What is a pointer?
8. How do you determine the address of a variable?
9. What is the difference between the following two prototypes?

```
return_type (*name)(parameter_list);

return_type *(name)(parameter_list);
```

10. What is a benefit of using variable-length structures?

Exercises

1. Write the code for the data type necessary to store a social security number of the format 999-99-9999. Don't use a simple string (character array).
2. Write the code for the data type necessary to store a type `long` variable called `number` in the same area of memory as a string variable called `string`. The length of `string` should be four characters.
3. Write a program that declares a pointer variable. The pointer variable should be assigned its own address.
4. Write a program that declares and initializes a double-dimensioned array. The program should also print the array to the screen.
5. How would you create a data element that could hold 10 social security numbers (from Exercise 1)?
6. Modify Listing 2.1 to print the numeric array as characters. What is printed?
7. **ON YOUR OWN:** Write a program that reads the information in the TMP_FILE.TXT created by Listing 2.9.
8. **ON YOUR OWN:** Write a program that sorts the TMP_FILE.TXT text entries and writes them back out.
9. **ON YOUR OWN:** If you are using a Windows compiler, consult your documentation for information on the Microsoft Windows file formats. Most of the file formats used by Microsoft Windows employ variable-length structures similar to those presented today. Another good reference book would be Tom Swan's book, *Inside Windows File Formats* from SAMS Publishing.