7.2 POINTER VARIABLES AND MEMORY ADDRESSES

7.2.1 Getting Started

We have been looking at variables that hold different types of data with different values.

```
int myAge = 39;
float mySpeed = 54.52;
char myAnswer = 't';
```

where, how

We know that those variables are stored in memory somewhere and that the different types require different amounts of memory. At this point, we should be asking several questions. How does one know *where* the variables are stored? How does one know *how* the variables are stored? Are large data objects stored the same way as smaller ones? How should one pass a large data object into a function? These questions are more relevant to embedded applications than to those on the desktop because often in embedded applications, one is trying to optimize the amount, organization, and use of memory in the system. The desktop application typically does not have such concerns. Let's see how to begin to answer these questions.

defined

Each variable that is *defined* has a storage place somewhere in memory and a value in that location independent of whether the variable is initialized or assigned. That place has an address. As each of the program's variables is stored, the system memory will begin to look like that in Figure 7.7.

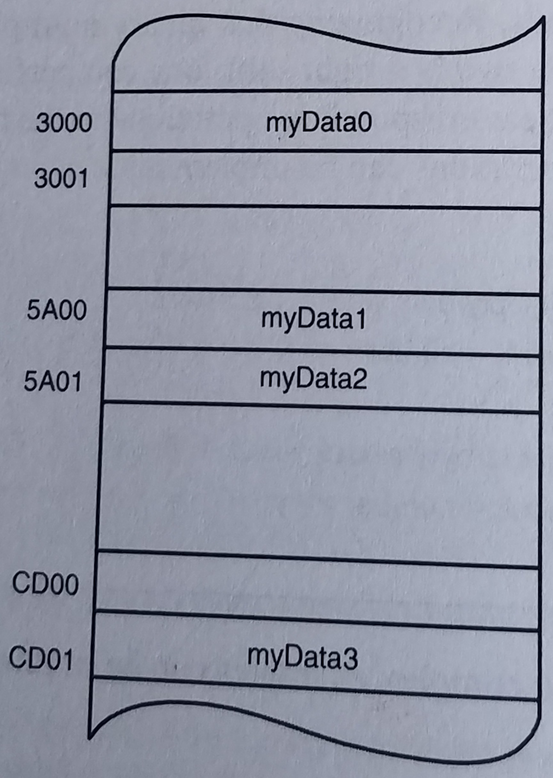


Figure 7.7 A Typical View of Data Storage in Memory

When one works with data, that is, variables, one can read their values, modify their values, or ask where they are stored. We have learned that variables hold different types of values, integers, floats, chars, and so on. In C, the type of the value of a variable can be a memory address as well. An address is just another variable with a value. Up to that point, it is nothing special. However, now some confusion can potentially arise.

As we learned when we first started working with the intrinsic types, they are simply a collection of bits in memory. It was not until a type was associated with the collection that it took on a meaning as an integer or char or float. If we look at the 16-bit quantity, 0xFE89, there is nothing to distinguish it as a variable's value or a variable's address. The compiler is no smarter. Therefore, to reduce the confusion, each variable that holds an address is

pointers given a distinguishing name and type—*pointers*. Thus, a pointer is a variable whose values are addresses in memory—nothing more and nothing less.

Consider the standard C declaration, definition, and initialization in the context of the memory fragment given above.

```
unsigned int myData0 = 0x3;
```

In response to such a declaration, the compiler

myData0
- Allocates 16 bits of memory to hold the variable *myData0*, assuming that we are working with 16-bit integers in our microprocessor. This is both a declaration and a definition.
- Places value 3 (0x3) into those 16 bits.
- Associates an address such as 0x3000 in memory where the data is stored with the variable *myData0*.

myData0 Subsequently, when we write *myData0*, we are actually referring to the data at location 0x3000. If we wish to know where data is stored, that is, at which address in memory, we simply ask. We use the *address of* operator &, and the compiler responds with the appropriate information. If we write

address of

```
&myData0; // read address of myData0
```

the compiler returns 0x3000 as the place where data is stored. Now, if we write

```
int *myData0Ptr = &myData0;
```

the following occurs. The compiler

myData0Ptr
- Allocates 16 bits of memory to hold the (pointer-type) variable *myData0Ptr*.

myData0
- Finds the address of *myData0* (which is 0x3000) and places that value into the 16 bits at that address.

We use the symbol * to tell the compiler that this variable is a pointer. We use a distinguishing name such as *myData0Ptr* to tell us (or people working with the code) that this variable is a pointer. This is good coding style.

When we are dealing with pointers and pointer declarations, knowing how to read them sometimes helps to make their role a little more clear.

EXAMPLE 7.7

Starting with the declaration and assignment,

```
int *myData0Ptr = &myData0;
```

the code fragment on the left of the assignment operator is read in several steps from right to left:

1. *myData0Ptr*—the first part
2. *is a pointer*—* the second part
3. *to an integer*—int the third part
4. myData0Ptr is a pointer to an integer

One more time,

```
float* anOtherPtr = &myFloatVar;
```

we read

1. *anOtherPtr*—the first part
2. *is a pointer*—* the second part
3. *to a float*—float the third part
4. anOtherPtr is a pointer to a float

If we assume that the compiler put the pointer variable *myData0Ptr* (remember, it's just another variable) at memory address 0xCD00, we now have the picture shown in Figure 7.8.

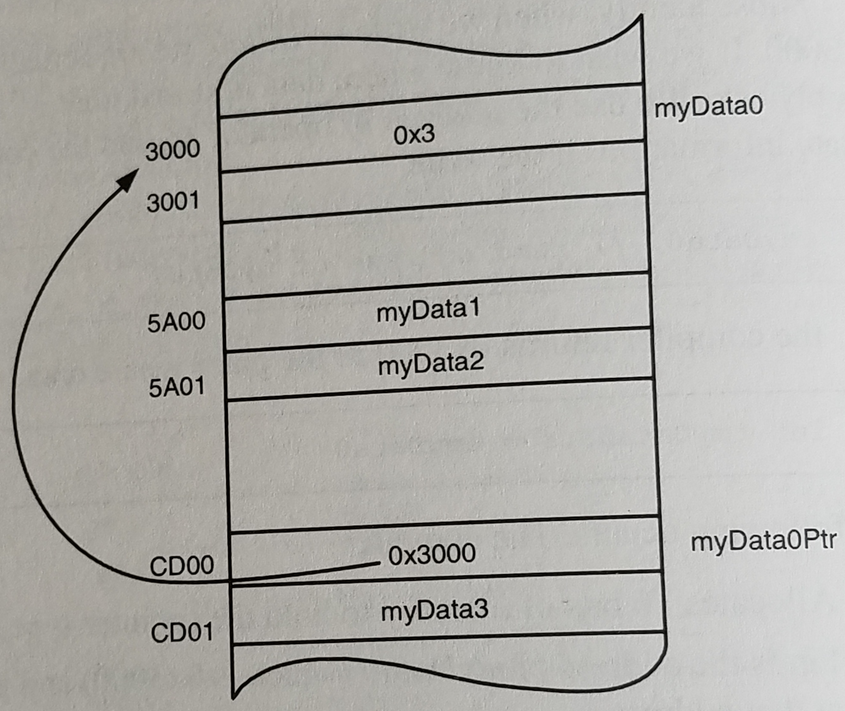


Figure 7.8 A Pointer Refers to a Variable in Memory

myData0Ptr, dereference operator

We can retrieve the value that *myData0Ptr* refers to by using the *dereference operator*, *, preceding the pointer variable. Thus, if we write

```
int myData1 = *myData0Ptr;
```

myData1

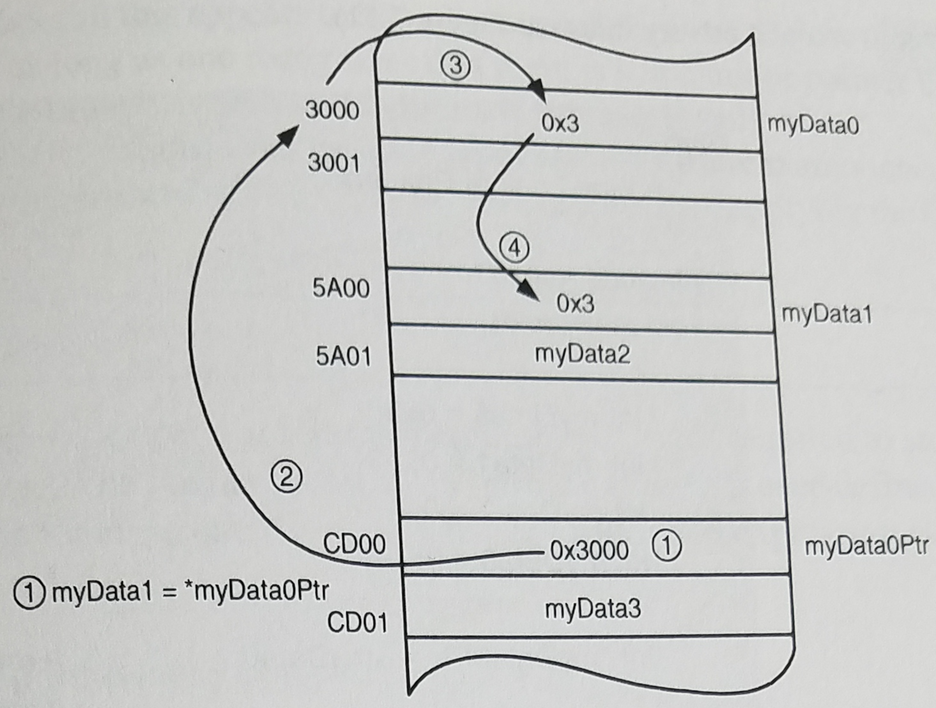
The value 0x3 is assigned to *myData1* using the steps shown in Figure 7.9.

Similarly, if we write

```
myData3 = 0x4;
*myData0Ptr = myData3;
```

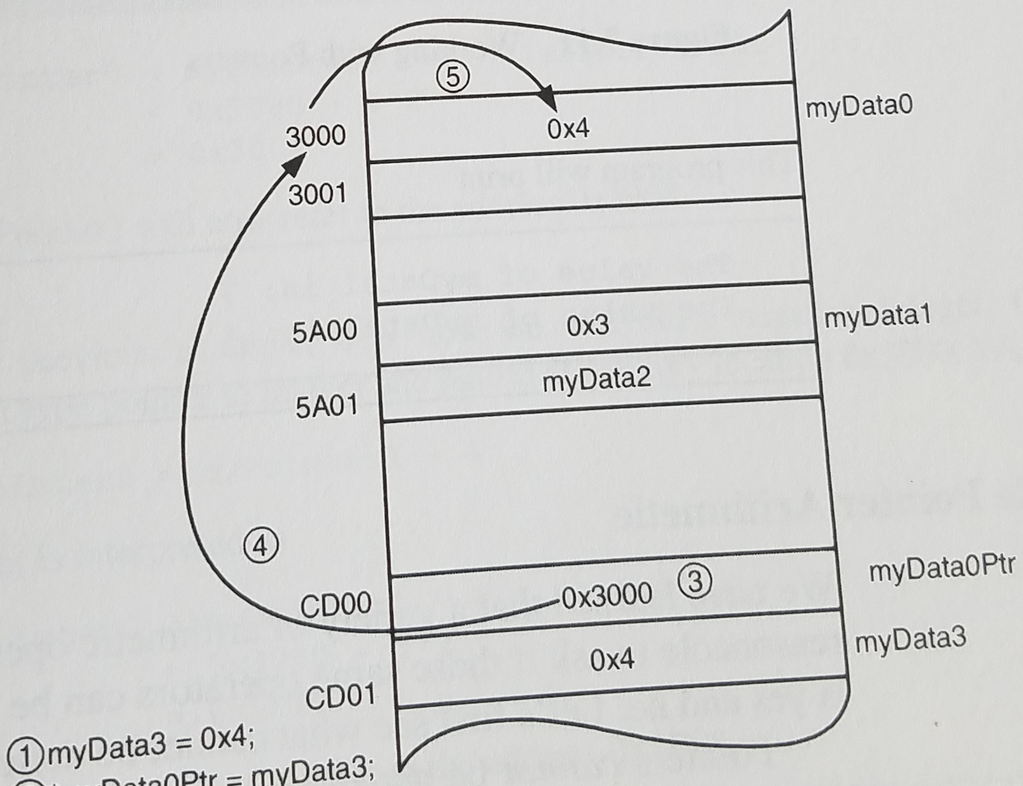
myData0Ptr

The value 0x4 is assigned to the memory location referred to by the pointer variable *myData0Ptr* using the steps listed in Figure 7.10.



① Get the value contained in the pointer variable *myData0Ptr*, 0x3000.
② Go to the address (0x3000) specified, pointed, or referred to by that value.
③ Get the value of the variable, *myData0*, at memory address 0x3000 – this will be the value 0x3.
④ Assign that value (0x3) to the variable *myData1*.

Figure 7.9 Using a Pointer to Retrieve the Value of a Variable in Memory



① Assign the value 0x4 to the variable *myData3*.
② Get the value contained in the variable *myData3* – 0x4.
③ Get the value of the pointer variable *myData0Ptr*, 0x3000.
④ Go to the address (0x3000) specified or pointed to by that value.
⑤ Assign the value contained in the variable, *myData3* to the variable at memory address 0x3000, *myData0*—this will be the value 0x4.

Figure 7.10 Using a Pointer to Change a Variable in Memory

EXAMPLE 7.8 Let's try this (see Figure 7.11).

```
/*
 * A First Look at Pointers
 */
#include <stdio.h>
void main(void)
{
      int myData0 = 0x3;
      int myData1 = 0;
      int myData2 = 0;
      int myData3 = 0;

      int *myData0Ptr = &myData0;          // myData0Ptr is a pointer to int
                                           // initialized to point to myData0
      myData1 = *myData0Ptr;               // myData1 now contains the value 3

      printf ("The value of myData1 is: %d\n", *myData0Ptr);
      myData3 = 0x4;                       // myData3 now contains the value 4
      *myData0Ptr = myData3;               // myData0 now contains the value 4 as well

      printf ("The value of myData3 is: %d\n", *myData0Ptr);
      return;
}
```

Figure 7.11 Working with Pointers

This program will print

```
The value of myData1 is: 3
The value of myData3 is: 4
```

7.2.2 Simple Pointer Arithmetic

We have learned that a variety of arithmetic operators can be applied to a C variable. It is reasonable to ask if those same operators can be applied to a pointer variable. The answer is yes and no. Let's first see what cannot be done.

Pointers *cannot* be added, multiplied, or divided. With a little bit of thought, these restrictions make sense. Pointers *cannot* be added, multiplied, or divided by a scalar. These, too, make sense. In both cases, one has to ask: If such operations were legal, does the result of the arithmetic operation give a meaningful answer?

On the other hand, a pointer variable and a scalar *can* be added. The result is a pointer variable—specifically, a pointer variable that refers to a memory address that is offset from the original address by the size of the offset in bytes. Conversely, two pointers can be subtracted; the result is a scalar. The value of the scalar is the size of the offset (in bytes) separating the two pointers.

On the surface, all this appears to be rather straightforward. Below the surface, it is still straightforward as long as one recognizes that there is some minor pointer magic going on. Let's see what that magic is and, at this point, if it really is only minor.

As we have learned, different variable types occupy different amounts of memory. The number of bytes required to store a type can be determined by applying the C *sizeof* operator to that type.

```
sizeof(type)
```

The application of the operator returns the number of bytes required to store an operand of the specified type. The operand may be an intrinsic type or a user-defined type.

What is happening under the hood when the following expression is written

```
myPointer0 = myPointer1+ anInteger
```

is that the compiler actually computes

```
myPointer0 = myPointer1+ anInteger * sizeof(type of myPointer1)
```

Such an operation gives a new pointer value that refers to a memory location that is separated from the original by *aScaler* number of variable instances of the specified type. Let's take a look in memory.

EXAMPLE 7.9

Let myPointer1 be of type pointer to integer and placed at memory location 0x3000. Let's further assume that an integer is 16 bits – 2 bytes on the machine. If we write

```
int* myPointer0 = myPointer1 + 4;
```

the code fragment is interpreted as

```
int* myPointer0 = myPointer1 + 4 * (sizeof ( int ) );
                = 0x3000 + 4*2
                = 0x3008
```

The variable myPointer1 will now refer to the address 0x3008.

EXAMPLE 7.10

Let's repeat the previous example; only now we will use a negative integer. Once again, let myPointer1 be of type pointer to integer and placed at memory location 0x3000. We write

```
int* myPointer0 = myPointer1 - 4;
```

the code fragment is interpreted as

```
int* myPointer0 = myPointer1 - 4 * (sizeof ( int ) );
                = 0x3000 - 4*2
                = 0x2FF8
```

The variable myPointer1 will now refer to the address 0x2FF8.

Returning to the problem of subtracting pointers, we should now see that the difference between two pointers is computed by the compiler as

```
pointer1 - pointer0 / sizeof(type)
```

Remember that when we write the name of a variable—in this case, a pointer— it is synonymous with writing its value and that value is an address.

The meaning and result of any of the following code fragments should now start to become clear.

```
int* myPtr = anAddress;
myPtr + 1;
myPtr++;
myPtr + 3;
myPtr - 1;
--myPtr;
```

Each of the operations algebraically adds a scalar value to a pointer. However, one must be careful when using the auto increment (decrement) operator and know when the pointers are dereferenced. For the auto increment (decrement) operator, prefix placement says to do the arithmetic and then evaluate the pointer variable, while postfix placement says to evaluate the pointer variable and then do the arithmetic.

The code fragments in Figure 7.12 do not all accomplish the same thing, nor do they leave the value of the pointer in the same state. Let's assume that the value of the pointer starts out at 0x3000, that the value 0x3 is stored at location 0x3000, and that we are working with 16-bit integers.

```
int aVal = *myPtr++;
```

Evaluation

1. myPtr will be dereferenced and will return 0x3000 because * is higher precedence than ++.
2. 0x3 will be assigned to aVal.
3. myPtr will be incremented by the size of one integer to 0x3002.

```
int aVal = *(myPtr++);
```

Evaluation

1. The operation inside the parentheses will be evaluated first. myPtr will be evaluated as 0x3000 and this will be the return value from the operation.
2. Before the return, myPtr will be incremented by the size of one integer to 0x3002.
3. The value 0x3000 is returned—the value *before* the increment.
4. 0x3 will be assigned to aVal because this is the value stored at memory location 0x3000.

Figure 7.12a Precedence with Pointers and Pointer Operations

```
int aVal = *myPtr+1;
```

Evaluation

1. myPtr will be evaluated as 0x3000 and dereferenced.
2. The value at memory location 0x3000 will be returned.
3. 1 will be added to the value returned from memory location 0x3000 to yield 0x4.
4. 0x4 will be assigned to aVal.

```
int aVal = *(myPtr+1);
```

Evaluation

1. myPtr will be evaluated as 0x3000.
2. 1 will be added to 0x3000 to give 0x3002.
3. The value at memory location 0x3002 will be returned and assigned to aVal.

Figure 7.12b Precedence with Pointers and Pointer Operations

Pointer Comparison

Another form of arithmetic on pointers is comparison. In reality, what is being compared are addresses. Such a comparison is meaningful only if the addresses are in the same address space.

Legal comparisons are as follows.

```
==, !=
The two pointer values are or are not the same address
< , <=, >=, >
The two pointer values are or are not referring to higher or lower
addresses
```

Const Pointers

The const qualifier applied to pointers can be somewhat confusing. This confusion occurs because there are three different interpretations based on placement of the const keyword,

Pointer is const

Pointer is const

Value of the pointer cannot be changed and must be initialized at the time of declaration.

Thing pointed to is const
Value of the object cannot be changed.

Both are const

as illustrated in Figure 7.13. Stated alternatively, if the keyword const appears to the left of the *, the object pointed to is constant; if the keyword appears to the right of the *, the pointer is constant.

char myChar = 'a';

Object is Constant	*	Pointer is Constant
char	*	ptr = &myChar
const char	*	ptr = &myChar
char	*	const ptr = &myChar
const char	*	const ptr = &myChar

Figure 7.13 Pointers and const

Reading a pointer declaration from right to left helps to lessen the confusion a bit. Looking at the second and third entries in the previous table, we have

```
const char* ptr = &myChar;
```

Read this as

```
ptr is a pointer to a character constant - the character is constant,
it can't be changed.
```

```
char* const ptr = &myChar;
```

Read this as

```
ptr is a constant pointer to a character - the pointer is constant, it
can't be changed.
```

EXAMPLE 7.11 The simple program in Figure 7.14 illustrates how we work with pointers and the const qualifier.

```
#include <stdio.h>

void main(void)
{
    // declare some working variables

    const char myChar0 = 'a';
    char myChar1 = 'b';
    const char* ptr0 = &myChar0;
    char* const ptr1 = &myChar1;

    // *ptr0 = 'c';          // illegal ptr0 points to a constant
    *ptr1 = 'd';             // ok, the pointer not the object is const

    ptr0 = &myChar0;         // ok, the object is const not the pointer
    // ptr1 = &myChar1;      // illegal, the pointer is const
    return;
}
```

Figure 7.14 Working with Pointers and const