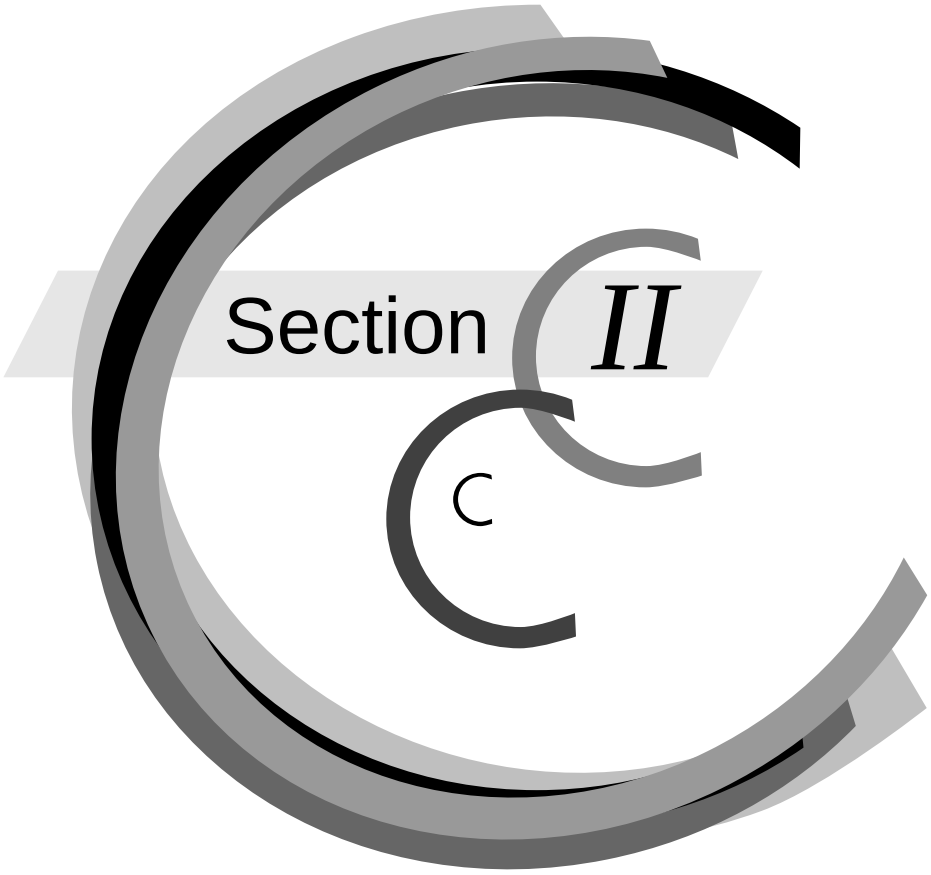

Arrays and Pointers

Arrays and Pointers

Arrays of Characters

Arrays of Structures

Arrays of Arrays

Arrays of Pointers

Arrays and Pointers

Offset Expressions, Increments, and Indexes

Array Delcarations and Initialization

Array References and Array Notation

Arrays and Pointer Operations

The Array Name as Pointer

Pointer Offset Expressions

Increments: Pointer Assignments

Indexes: Subtraction of Pointers

Arrays and Functions

Section 2: Arrays and Pointers

10. Arrays and Pointers

An array is a collection of objects, all of the same data type. Any one data type can be used in an array. You can declare an array of integers, an array of characters, an array of structures, and even an array of pointers. Though an array may contain objects, an array is not an object itself. The declaration of an array reserves memory, which is then managed by pointers. Array objects themselves are actually referenced through pointer indirections. In this respect, there is a special relationship between arrays and pointers. That relationship will be carefully explored throughout this chapter.

There are three pointer operations designed specifically to manage arrays. They are described in this text as the offset, increment, and index operations. The offset operation references an object in an array. The increment operation advances sequentially from one object to the next. The index operation provides the position of an object in an array.

An array object can also be referenced using array subscripts. An array subscript is a notation consists of a set of brackets and the index of an object. Array subscripts looks much like array references in other languages. It is often clearer and more elegant to use. However, an array subscript is equivalent to the pointer offset operation. It is, in fact, just another way of writing a pointer offset operation.

Array Declarations and Initializations

An array declaration consists of four parts: the object data type, the array name, the array data type, and the number of objects in the array. The array data type is represented with opening and closing brackets, `[]`. An integer value is placed within the brackets to specify the number of objects being declared. Figure 10.1 describes the different components of an array declaration. The declaration shown here declares an array of 5 integers. The array name is `mynums`.

```
int mynums[5];
```

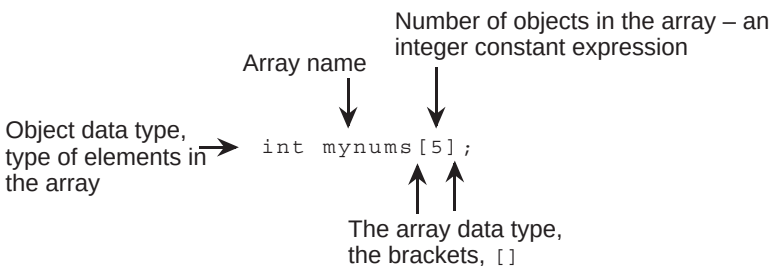

Figure 10.1. Array Declaration.

The number of objects in an array is determined by an integer constant expression. In the declarations that follow, the number of objects for `mynums` is determined by the constant 5. The number of objects in the array totals is determined by a constant arithmetic expression, 3 * 5.

```
int totals [3*5];

int main(void)
      {
      int nums [5];

      }
```

You can declare many different kinds of arrays, each having its own data type and number of objects. In the next example, the arrays `mynums` and name have the same number of objects, 5, but different types. `mynums` is an array of integers, and name is an array of characters. Though the arrays `mynums` and totals have the same type of objects, they have a different number of objects. `mynums` has 5 objects, and totals has 10 objects.

```
int mynums [5];
int totals [10];
char name[5];
```

You can think of the objects in an array as being numbered sequentially. The numbering begins with zero and ends with one less than the number of objects in the array. The array `mynums` is an array of 5 integers, each numbered sequentially from 0 to 4 (see figure 10.2).

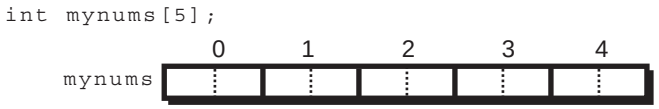

Figure 10.2. Array elements numbered from 0.

The array itself is identified by its array name. Each object in the array is referenced through the array name. The number of an object's place in the sequence, together with the array name, is used to reference the object.

Array Initialization

Recall that when you declare a variable you can also initialize it with a value. In the declaration char mychar='E', the variable mychar is initialized with the character value 'E'. In much the same way, when you declare an array, you can also initialize its elements. You list the initialization values within a set of braces and separate them sequentially by commas. The first value will be assigned to the first element, the second value to the second element, and so on. In the next example, the `mynums` array is declared as

an array of five integer objects, each initialized by a value in the initialization list. The first object is assigned the first value in the initialization list, in this case 3. The second object is assigned the second value, 4, and so on.

```
int mynums[5] = { 3,4,5,6,7 };
```

You can even use the initialization part of the array declaration to specify the actual number of objects in the array. To do so, you simply leave out the number that you would usually place within the brackets to specify the number of objects in the array, leaving you with a set of empty brackets followed by a list of initialization values. The number of objects in the array will then be determined by the number of values in the initialization list. In the next example, the myletters array is declared as an array of three integers. Notice the empty brackets. The number of objects in the array is determined the number of values in the initialization list, in this case three.

```
int myletters[] = { C,D,E };
```

In the **arinit.c** program in Listing 10.1, both the arrays letters and totals use this form of array declaration. `myletters` is declared as an array of three characters. The array declaration uses empty brackets and has three values in its initialization list. `totals` is declared as an array of four integers, having four values in its initialization list.

LISTING 10.1

arinit.c

```
#include <stdio.h>

int main(void)
      {
      char myletters[] = {'C','D','E'};
      int totals[] = {23, 8, 11, 31};

      putchar (myletters[1]);

      return 0;
      }
```

There are variations on array initialization which are unique to the type of arrays declared. Arrays of characters allow certain kinds of initialization using string constants, as do arrays of pointers to characters. Arrays of arrays and arrays of structures use nested lists of array values. Each variation will be presented when those types of arrays are discussed.

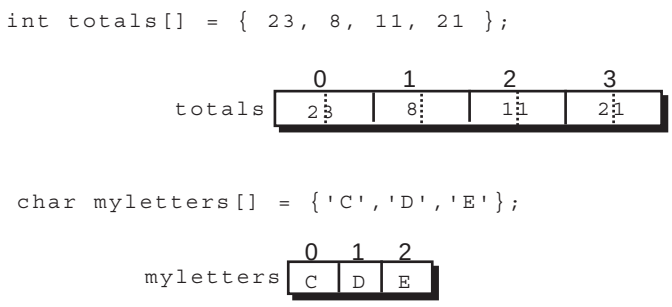

Figure 10.3. Array initialization using empty brackets.

Array References and Array Subscripts

Once you have declared an array, you can reference its objects, and then use those objects in expressions. To reference an array object, you need to specify its position in the array. As previously noted, objects in an array are arranged in sequence, starting from zero. In this sense, objects are numbered according to their position in that sequence. You can then use this numbering to reference individual objects in the array. You can reference an object in the third position using the number 2. The object in the second position is reference using the number 1. Remember that the numbering starts from 0, not 1, so that the first object is referenced using the number 0 (not 1). The number of an object's place in an array sequence is often referred to as either the object's index or subscript.

You reference an array object by using a combination of the array name together with the position of that object in the array. However, the actual reference can take two different forms: that of array subscripts or of a pointer offset operation (the pointer offset operation is discussed later in this chapter.) An array subscript is very similar to the way in which arrays are referenced in other programming languages. With an array subscript, a particular object in an array is referenced with its array name and the number of the object's position in the array, which is placed within brackets. `mynums[2]` refers to the third object in the `mynums` array; `mynums[0]` refers to the first object. The numbering always begins with 0.

```
mynums[0]          /*first object */
mynums[2]          /*third object */
mynums[4]          /*fifth object */
```

Once you have referenced an object, you can use it as you would any other object of that type. An integer object can be used like any integer variable. You can think of a reference of an object in an array of integers, as a reference an integer variable. Just as variables can be assigned values, each object of an array can be assigned a value. You can use an array reference to an integer object in an arithmetic expression, just as you would use an integer variable. In the next example the fourth object in the `mynums` array is first assigned a value, and then used in an addition operation, assigning the result to the first object.

```
      int mynums[5];

      mynums[3] = 23

      mynums[0] =  ( 2 * mynums[3]);
```

The term element is often used to refer to an object in an array. In an array of integers, the integer objects making up the array are referred to as elements of the array. `mynums[2]` references the third element in the `mynums` array (see figure 10.4). However, the term element can be misleading. An element is an object itself, not merely a piece of a larger construct. A reference of an element in an array of integers references an integer variable. In listing 10.2, each element of the `mynums` array is referenced and used in an assignment operation. Figure 10.4 shows the `mynums` array with its assigned values.

LISTING 10.2

element.c

```
#include <stdio.h>

int main(void)
      {
      int mynums[5];

      mynums[0] = 9;
      mynums[1] = 87;
      mynums[2] = 95;
      mynums[3] = 23;
      mynums[4] = 45;

      return 0;
      }
```

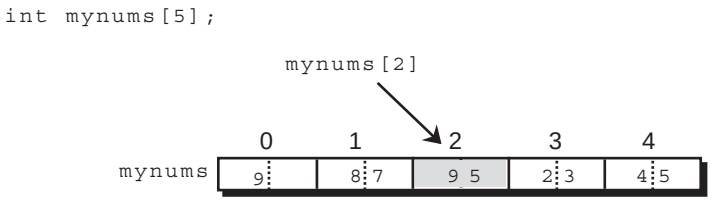

Figure 10.4. Referencing array elements.

Array Management and Loops

An array is only a collection of objects. It is not an object itself. This means that you cannot perform an operation on an array as a whole. Instead, you need to perform a separate operation on each individual element, dealing with them one by one. For example, to assign a set of values to an array, you need to reference and assign a value to each individual element of the array.

In this respect, an array initialization can appear misleading. The format of an array initialization looks like a single assignment operation as it is written. This may wrongly lead you to infer that the array is being treated as a whole It is not. In fact, each element of the array is initialized its own value. For example, the array initialization shown in Listing 10.3 may appear to be a single operation on the array `mynums`, but it is not. You should think of the array initialization:

```
int mynums[3]={12, 5, 27};
```

as if it were three separate assignment operations, one for each element of the 3 element `mynums` array:

```
mynums[0]= 12
mynums[1]= 5
mynums[2]= 27
```

LISTING 10.3

mynums.c

```
#include <stdio.h>

int main(void)
      {
      int mynums[3]={12, 5, 27};
      int i;

      for (i = 0; i < 3; i++)
            {
            printf("%d \n", mynums[i]);
            }

      return 0;
      }
```

Often you will have a task in which you need to reference all elements in an array, performing the same operation on each element. Simply printing out an array is an example of such a situation. Each element is referenced and printed out. You could write a print statement for each individual element, but it is far more practical to simply use a loop control structure such as a while or for. Inside the loop you would then need only one print statement, and each time through the loop the next element would be printed, progressing from one element to the next. The integer variable used to control the count of the loop would also be used to index and reference each array element in turn.

In the **mynums.c** program in Listing 10.3, a for loop is used to print out all the values in an array. Within the loop each element is individually referenced and printed out. Since arrays are numbered from 0, care is taken to initialize the counting variable i to 0. The index of the last element of the letters array is 2. The loop must cut out after 2, before 3. For this reason, the test is (i<3). It could just as easily have been (i<=2).

A common rule of thumb is that the test for the end of an array consists of the less-than operator, <, tested against the number of objects declared in the array. If you declare an array to have 3 objects, then, in a loop, you would test against 3, (i<3). To insure this fact, it is a common practice to define a symbolic constant that represents the number of elements in a given array. You could then use this same symbol in loop control structures as the cuttoff in the loop's test. In the **maxnums.c** program in Listing 10.4, the same symbolic constant, MAX, is used in both the array declaration and the test for the

last array object in the `for` loop. Such a strategy has the distinct advantage of letting you easily change the size of an array. Just change the number specified for the symbolic constant.

LISTING 10.4

maxnums.c

```
#include <stdio.h>
#define MAX 3

int main(void)
      {
      int mynums[MAX]={12, 5, 27};
      int i;

      for (i = 0; i < MAX; i++)
            {
            printf("%d ", mynums[i]);
            }

      return 0;
      }
```

You can also use loops to perform operations such as copying one array to another. Copying an array requires that you individually reference the value of each element and assign it to an element in another array. The **copynums.c** program in Listing 10.5 uses a for loop to copy the `mynums` array to the newnums by individually copying each elements, one by one.

LISTING 10.5

copynums.c

```
#include <stdio.h>
#define MAX 3

int main(void)
      {
      int mynums[MAX]={12, 5, 27};
      int newnums[MAX];
      int i;

      for (i = 0; i < MAX; i++)
            {
            newnums[i] = mynums[i];
            }

      return 0;
      }
```

Array References and Pointer Operations

An array subscript allows you to write a reference to an array element in much the same way as you would write a simple variable reference. A variable is reference by it name, and an array element is

referenced by the array name and an index (an integer representing the position of the element in that array). You can use such array subscripts references in the same way as you would use any variable reference. However, the array subscript reference of an element only appears similar to that of a variable. In fact, an array subscript is merely a notation, hiding the true underlying operation taking place, a pointer operation. Arrays are actually managed entirely by pointers. What appears to be a simple variable-like array element reference, is actually a pointer operations using a pointer and indirection. To truly understand how arrays work in C, you need to know the pointer operations that actually manage them. There are three pointer operations used to manage arrays: the offset, increment, and index operations. Array subscript is actually translated into a pointer offset and indirection operation.

So far we have only examined how you can use pointers to reference a variable (as noted in Chapters 5, 6, and 7). You can reference a variable either with its name or with an indirection operation on a pointer that holds the address of that variable. However, array elements are not themselves variables, and the pointer operations on arrays are not quite the same as those used on variables. To understand the differences, it is important to remember that the indirection operation does not require that pointer's address be the address of a variable. The indirection operation references the memory at any address as if it were a variable. An indirection operation on a pointer to an integer references memory as if it were reserved for an integer variable. Remember how the function malloc works with pointers. In the example discussed in Chapter 5, the function malloc set aside memory in the heap and returned its address. There was no variable declaration. The address was assigned to a pointer to an integer. Indirection on that pointer referenced that memory in the heap as if it were the memory of an integer variable. A variable requires a type, an address, and memory reserved at that address. The pointer provided the type and address. malloc provided the reserved memory.

A similar process takes place for arrays. An array declaration reserves memory. The array name used in the declaration then functions as a pointer holding type and address information. In this respect, an array name radically differs from variable names,. It is, in fact, a pointer. The array name itself is the address of the first byte in that memory. Indirection operations can reference different parts of that memory as if they were different variables. In this sense, an array declaration does not actually create objects. Though you can treat array elements as variables, they are actaully referenced through pointer operations on the array's memory.

An array declaration reserves enough memory for the number of elements declared. An array of 10 integers will reserve 20 bytes, 2 to an integer. An array of 5 characters will reserve 5 bytes, one for each character. You then use an indirection operation to reference each element. Before you can reference a particular element in an array, you need to first calculate the address of that element's memory. The offset and increment pointer operations can calculate the address of an array element. Once calculated, an indirection operation can then reference the element. The index operation allows you to uses two addresses to calculate an element's index, its position in an array.

The pointer operations were designed specifically to work on arrays. It is possible for them to work on any chunk of memory. However, they only make sense when applied to arrays.

The Array Name as Pointer

The use of an array name in an array's declaration can be misleading. An array declaration is made using the array name and the array data type, []. There is no specific pointer type in the declaration, *. However, when an array name is used in an expression, its type is that of a pointer. In expressions, the array name becomes a pointer that holds the address of the first element in the array. Any operations using the array name, including array subscript references, are really pointer operations. In the Figure

10.5, the `mynums` array declaration defines an array of five character bytes that are consecutively set aside together in memory. In any expressions, the array name `mynums` is a pointer representing the address of the first byte.

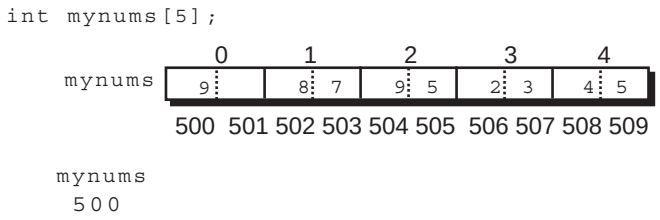

Figure 10.5. Arrays and address.

An array name, when used in expressions, operates like of a pointer constant. You can think of it as a symbolic constant that represents the address of the first array element. An array name, however, cannot have its address changed. It is not a pointer variable. Nor can the address operator operate on it. The address operation `&mynums` is invalid.

You can use an array name as the pointer operand in an indirection operation, just like a pointer variable. Indirection on the array name itself references the first element in the array, as shown in Figure 10.6. The indirection operation below references the first element in the `mynums` array.

```
*mynums
```

In the following statement, the call to `printf` prints out the contents of the first integer element in the `mynums` array, in this case 9.

```
printf("%d",*mynums);
```

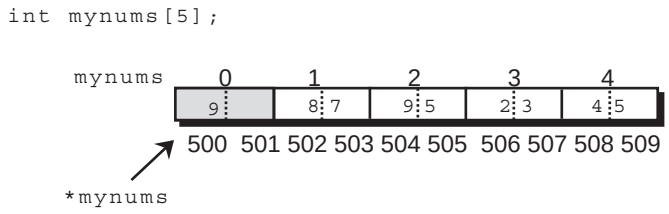

Figure 10.6. Using indirection on an array name to reference the first element in the array.

Though, in almost every expression, an array name is a pointer, there is one important exception. As an operand of the sizeof operator, the array name does not function as a pointer. Instead it maintains the original array data type defined in its array declaration. The array data type will specify the type of object and number of objects in the array. The sizeof operator will result in the total number of bytes in the array. sizeof calculates the size by multiplying the size of the type of object in the array by the

number of objects. In the **sizenums.c** program in Listing 10.6, the size of the `mynums` array is 6, and the size of the last array is 5.

LISTING 10.6

sizenums.c

```
#include <stdio.h>

int main(void)
      {
      char last[5] = {'R','I','C','H','\0'};
      int mynums[3] = {12, 5, 27};
      int numsize, lastsize;

      numsize = sizeof mynums;
      lastsize = sizeof last;
      printf ("Size of mynums is %d\n", numsize);
      printf ("Size of last is %d\n", lastsize);

      return 0;
      }


      Size of mynums is 6
      Size of last is 5
```

Pointer Offset Expressions

A pointer expression is an operation that results in an address. A pointer expression can be a primary expression consisting of a pointer variable or a pointer constant. A pointer expression can also be a function call, such as `malloc`, that returns an address. A pointer expression can even be an assignment operation, in which an address is assigned to a pointer variable. The address assigned is the resulting value of the expression.

There is another pointer operation, which is technically referred to as a pointer arithmetic operation. It appears similar to an arithmetic additive operation. However, it is not a simple arithmetic process. It is designed to calculate the addresses of array objects. A better name for it may be the pointer offset operation. It uses an offset to calculate an object's address in an array.

The pointer offset expression consists of two operands: a pointer and an integer. The integer is added to (or subtracted from) the pointer, resulting in a new address. This integer will be referred to here as the offset. The pointer operand is any pointer expression. It can be a pointer variable, an array name, a pointer assignment expression, or even another pointer offset expression.

(pointer + integer)

The offset in the pointer offset expression actually refers to the number of objects, not the number of bytes. If an offset of 1 is added to an initial address, the address of the next object in the array will be referenced. It is tempting to think of a pointer offset operation as adding the offset to the pointer address. However, the offset is not added directly to the address. In pointer offset expressions, there is a further hidden, implied calculation using the pointer's data type. The offset is always multiplied by the size of the pointer's data type. The result of this calculation is the number that is then added to the

pointer address. In Figure 10.7 an offset of 3 is added to the array name `mynums`. The integer 3 will be used to calculate an offset of 3 integers from the base address. The pointer's data type is an integer, and the size of an integer is 2 bytes. An offset of 3 on an integer array actually adds 6, (3 * 2). In the expression `(mynums+3 )`, the hidden multiplication by the size of the integer type results in the address of the fourth element, 506.

```
(mynums + 3)
```

The address that results from the pointer offset expression can then be used in an indirection operation to reference the object at that address. Remember that an address is not merely an address, but an address of a type of object. In this sense, an address can be thought of as a pointer itself. An address of an array element calculated by the pointer offset operations, will hold the data type of that element. An indirection operation on the address of an array element can then reference that element.

```
(mynums + 3)    is    506
                      (500 + (3 * 2) );
                      (500 + 6)   is 506
```

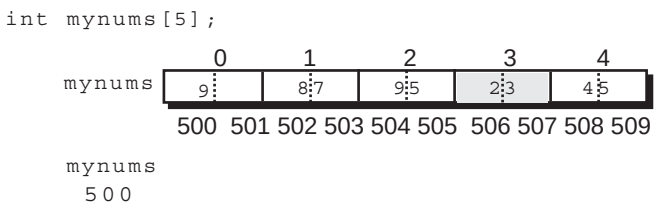

Figure 10.7. Pointer offset expression.

You can apply the indirection operator directly to a pointer offset operation. Using an array name as the pointer operand in an offset operation, a combined pointer offset and indirection operation can reference a particular element in the array. For example, the combined expression `*(mynums+2)` first calculates the address of the third object in the array and then references that object with an indirection operation.

```
*(mynums + 2)
```

Notice that the integer used in the pointer offset operation is the index of the object referenced. Figure 10.8, shows how a pointer offset operation derives the address of an array element, and how an indirection operation on this address references the element.

```
(mynums + 3)    *(mynums + 3)
(500 + 3)       *(500 + 3)
(506)           *(506)
 506            *506
          23
```

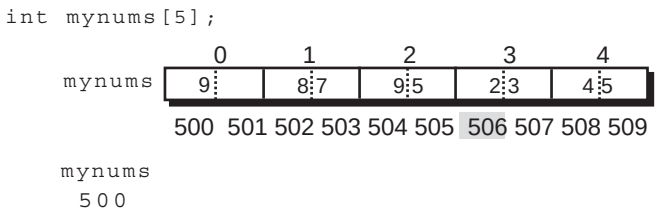

Figure 10.8 Pointer offset and indirection to reference array element.

In the program **offnums.c** in Listing 10.7, the programs presented in Listings 10.5 and 10.6 are combined and implemented using combined pointer offset and indirection operations, instead of array subscripts. Notice how the integer used in the pointer offset expressions is derived from a pointer variable, i, not a constant.

LISTING 10.7

offnums.c

```
#include <stdio.h>
#define MAX 3

int main(void)
      {
      int mynums[MAX]={12, 5, 27};
      int newnums[MAX];
      int i;

      for (i = 0; i < MAX; i++)
            {
            printf("%d\n", *(mynums + i) );
            }

      for (i = 0; i < MAX; i++)
            {
            *(newnums + i) = *(mynums + i);
            }

      return 0;
      }
```

The integer in the pointer offset expression need not be a constant. The integer is literally derived from an integer expression, which could just as easily be a variable, an arithmetic expression, or a function call returning an integer. In the program **offsets.c** in Listing 10.8, there are several ways in which the integer of the pointer offset expression is calculated. All these offset operations reference the fifth object. Notice that in the last statement the integer is obtained from an integer variable referenced through a pointer.

LISTING 10.8

offsets.c

```
#include <stdio.h>

int square(int);

int main(void)
            {
            char mynums[13];
            int myoffset;
            int square(int);
            int *ptr;

            myoffset = 4;
            *(mynums + myoffset);

            *(mynums + (2 * 2));
            *(mynums + (8 / ((int) 2.786)) );

            *(mynums + square(2) );

            ptr = &myoffset;
            *(mynums + *ptr);

            return 0;
            }

int square(int num)
      {
      return (num * num);
      }
```

Array Subscripts as Pointer Offsets

As previously noted, array subscripts consist of brackets enclosing an index and placed next to an array name. The brackets used in a declaration have a different meaning than those used in array subscripts. The brackets in an array declaration represent the array data type. However, in array subscripts, the brackets are only a convenient and optional representation of a combined pointer offset and indirection operation. In fact, your compiler will strip away brackets used in an array subscripts and replace them by an offset and indirection operation. The array subscript `mynums[i]` actually represents the offset and indirection operation `*(mynums+i)`. For example, the programs in Listing 10.9 are exactly of equivalent. The **poffset.c** program in Listing 10.9A uses array subscripts, whereas the **notation.c** program in Listing 10.9B uses the combined offset and indirection operations.

LISTING 10.9A

poffset.c

```
#include <stdio.h>

int main(void)
{
int i;
int mynums[2] = {12,44};


i = 0;
while(i < 2)
      {
      printf("%d",*(mynums+i));
      i++;
      }
}
```

LISTING 10.9B

notation.c

```
#include <stdio.h>

int main(void)
{
int i;
int mynums[2] = {12,44};


i = 0;
while(i < 2)
      {
      printf("%d ",mynums[i]);
      i++;
      }
}
```

One often confusing feature of array subscripts is that it doesn't have to be used with just array names. It can also be used with regular pointer variables. Array subscripts works equally well for both pointer variables and array names. Since array subscript is really a pointer offset expression, the address used in the offset expression can just as easily be obtained from a pointer variable as from an array name. In Listing 10.10 there are two versions of the same program. Both use a pointer variable. The pointer variable, `numptr`, is first assigned the beginning address of the `mynums` array, `numptr = mynums`. The array name `mynums` evaluates to the beginning address of the array. The **ptr_off.c** version in Listing 10.10A uses the standard pointer offset operation. The **ptr_note.c** version in Listing 10.10B uses array subscripts with a pointer variable.

LISTING 10.10A

ptr_off.c ptr_note.c

```
#include <stdio.h>


int main(void)
{
int mynums[2] = {12,44};
int *numptr;
int i;


numptr = mynums;
i = 0;
while(i < 2)
      {
      printf("%d",*(numptr+i));
      i++;
      }
}
```

LISTING 10.10B

```
#include <stdio.h>


int main(void)
{
int mynums[2] ={12,44};
int *numptr;
int i;


numptr = mynums;
i = 0;
while(i < 2)
      {
      printf("%d ",numptr[i]);
      i++;
      }
}
```

Pointer Arithmetic

Kernighan and Ritche describe a set of pointer expressions by the term pointer arithmetic, which can be misleading. Arithmetic operations, as such, cannot be performed on pointers1. You cannot divide, multiply, or add pointers. Technically, you can subtract pointers. But even pointer subtraction is not the same as arithmetic subtraction, and is more accurately referred to a pointer difference operation.

Though a pointer cannot be added to another pointer, a pointer can be added to an integer. An integer can also be subtracted from a pointer. However, other types, such as floats, doubles, and longs, cannot be added to or subtracted from a pointer. Furthermore, pointers cannot be multiplied or divided by integers. Multiplication or division of any kind is strictly prohibited with pointers. These restrictions eliminate most arithmetic operations. Only the following two operations are permitted:

1. The addition or subtraction of an integer to a pointer.
2. The subtraction of a pointer from another pointer.

The addition or subtraction of an integer to a pointer is referred to here as the pointer offset operation. This operation is used almost exclusively to reference objects in an array. The subtraction of one pointer from another is usually used to determine the integer index of an object in an array. For that reason it is referred to here as the index operation.

Increments: Pointer Assignments

You may recall the increment operator, ++, and its corresponding decrement operator, --, as described in Chapter 3. When applied to an integer variable, the increment operator increments the integer variable by 1, performing a combined addition and assignment operation. Remember that:

`i++` is equivalent to `i=i+1`

The decrement operator would decrement the integer by 1, performing a subtraction instead of an addition. i-- is equivalent to i=i-1.

In much that same way, you can also apply the increment and decrement operators to pointer variables. But in this case, the increment and decrement operations are equivalent to pointer expressions, not simple addition and subtraction. When you use an increment operator, ++, on a pointer variable, the increment operation can be thought of as incrementing the pointer variable's address. This pointer increment operation is the equivalent of a combined pointer offset and pointer assignment operation. In the pointer offset operation, the pointer operand is the pointer variable, and the integer is the constant 1, which is added to the address held by the pointer variable. The resulting address is then assigned back to the same pointer variable.

`numptr++` is equivalent to `numptr=numptr+1`

1. Kernighan, B. and D. Ritchie. The C Programming Language. (2nd Edition.) Englewood Cliffs: N.J.: Prentice Hall, 1978. pp.100-103.

It is important to realize that an offset of 1 includes a hidden multiplication by the size of the pointer's data type. In Figure 10.9, an increment of `numptr` increments by the size of an integer, 2 bytes. Given this fact, you can now see how such pointer increments can be very useful in referencing array elements. If a pointer variable holds the address of an array element, then an increment operation on that pointer will give it the address of the next element in the array, effectively moving from one

element to the next. An increment of such a pointer is always an increment of the pointer to the address of the next element in the array.

```
numptr++  is          502
                      (500 +  (1 * sizeof(int)))
                      (500 + (1 * 2) );
                      (500 + 2)  is 502
```

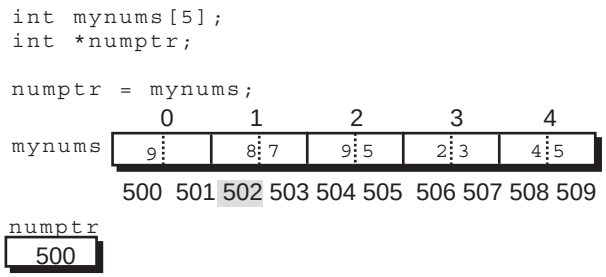

Figure 10.9. Increment on pointer variable, referencing array elements.

Once the increment operation changes the pointer variable to the address of the next array element, then you can reference that element by performing an indirection operation on that pointer. For example, as shown in Figure 10.9, the increment of the `numptr` variable changed the address it held to 502, the address of the second element. An indirection operation on `numptr` will then reference that second element, `*numptr`. In this case, the following `printf` out display the value of the second element in the `mynums` array, 87.

```
printf("%d\n",*numptr);
```

The increment operation applied to a pointer variable is often used to advance down an array, element by element. Such a pointer variable is often referred to either as a working pointer or as a temporary pointer. The pointer variable is first assigned the beginning address of the array, the address of the first element. Then subsequent increment operations will move the pointer from one element's address to the next. A indirection operation on that address will reference that element. In the program **ar_inc.c** in Listing 10.11, `numptr` is an example of such a working pointer. `numptr` is used to advance down the `mynums` array. Each integer in the `mynums` array is printed out using a pointer, `numptr`. `numptr` is incremented by the size of an integer. It is consecutively set to the address of each integer element in the array.

LISTING 10.11

ar_inc.c

```
#include <stdio.h>

int main(void)
      {
      int i = 0;
      int mynums[5] = { 9, 87, 95, 23, 45 };
      int *numptr;

      numptr = mynums;
      while (i < 5)
            {
            printf("%d %d %p \n", i, *numptr, numptr);
            numptr++;
            i++;
            }

      return 0;
      }
```

Assuming that the address of the `mynums` array in Listing 10.11 is 500, the program would print out:

```
0       9       500
1       87      502
2       95      504
3       23      506
4       45      508
```

In the **ar_inc.c** program in Listing 10.11, a counter, i, had to be tested against the number of objects in the array, 5, in order to detect the end of the array. The counter is not needed for anything else. There is a way to do away with this overhead, and use the pointer variable instead of an integer counter to detect the end of the array. To do so you need to determine the end address of the array The working pointer can then be tested against this end address. You can easily determine the end address of an array by a pointer offset operation in which the number of objects in an array is added to the array name. For example, given the declaration `mynums[5]`, you can calculate the end address of the `mynums` array with the offset expression `mynums+5`. Assuming that the `mynums` array name is the address 500, `mynums+5` will result in the address 510. You could then test a working pointer against this address to detect the end of the array. In Listing 10.12, `numptr` tests for the end of the array with the expression `numptr<(mynums+5)`. Notice that there is now no longer any need for a counter variable. The entire loop is managed using pointers and pointer operations such as offsets, increments, and indirections.

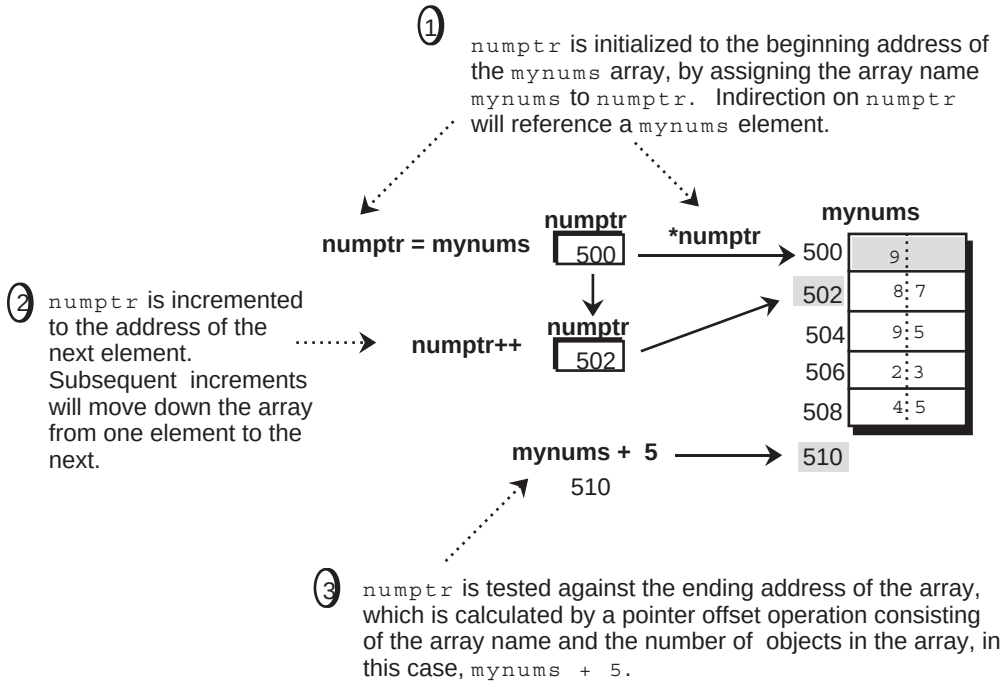

Figure 10.10. Using working pointers on an array.

Remember that `mynums` as an array name is a pointer, and that `mynums+5` is a standard pointer offset operation. As such there is a hidden multiplication by the size of the pointer data type, in this case an integer. Assuming that an integer is 2 bytes, the `mynums+5` is equivalent to `mynums + (5*2)`. If `mynums` is 500, this gives us 500 + 10, 510, the end address of the array.

Listing 10.12

ptrinc.c

```
#include <stdio.h>

int main(void)
      {
      int mynums[5] = { 9, 87, 95, 23, 45 };
      int *numptr;

      numptr = mynums;
      while (numptr < (mynums + 5))
            {
            printf("%d  %p \n", *numptr, numptr);
            numptr++;
            }

      return 0;
      }
```

A decrement operator performs the same kind of operation, except that a pointer is decremented to the address of the previous object. The decrement is actually a combination of the subtraction and assignment operations.

`numptr--` is equivalent to `numptr=numptr-1`

Like the increment operation, the decrement contains a hidden multiplication by the size of the pointer data type. The result is the subtracted from the pointer variable. Using the `mynums` array in Figure 10.9 as an example, if `numptr` is set to 508, then `numptr--` will set `numptr` to 506. Since `numptr` is a pointer to an integer, there is a hidden multiplication by the size of an integer, 2.

```
int *numptr;
numptr--     evaluates to          numptr = numptr - (1 * 2)
```

In the **ptrdec.c** program in Listing 10.13, the `mynums` array is printed out in reverse using the decrement operator. Notice how `numptr` is initialized to the last object in the array. The offset of the last object is always the number of objects in the array minus 1.

LISTING 10.13

ptrdec.c

```
#include <stdio.h>
#define MAX 5

int main(void)
      {
      int mynums[MAX] = { 9, 87, 95, 23, 45 };
      int *numptr;

      numptr = mynums + (MAX - 1);
      while (numptr >= mynums)
            {
            printf("%d  %p \n", *numptr, numptr);
            numptr--;
            }

      return 0;
      }
```

You can, if you wish, combine the pointer increment and indirection operations. Such combinations can be difficult to interpret so you may want to avoid them. However, they do allow the development of very compact code. One key point to remember in such combination is that the increment can be either a postfix or prefix operator. When placed before a variable it is prefix operation, being performed before any other operation in the expression. When placed after the variable, it is a postfix operator and is only performed after all other operations in the expression. Commonly the increment is combined with indirection as a postfix operation. The indirection operator is placed before the pointer variable and the increment operator after it.

```
*numptr++
```

The other key point in such combinations is to keep in mind that the increment operation can operate either on the pointer or on the object pointed to by the pointer. This all depends upon which

operation is evaluated first. If the increment is evaluated first, then the pointer is incremented. But if the indirection is evaluate first, then the increment operates on the object referenced by that indirection.

You should think of `*numptr++` as actually two operations in one, `*numptr` and `numptr++`. Since both the indirection and increment operators have the same precedence, their associativity will determine the order in which they are evaluated. This means that where you position the increment and indirection operators is crucial. The indirection and increment operators associate from right to left. The right-most operator will be evaluated first. In the case of `*numptr++`, the increment, `++`, is first evaluated and applied to the variable `numptr`. This means that the address in `numptr` will be incremented. Incrementation, in this case, applies to the pointer variable. However, the increment operator is placed after the pointer, making it a postfix operation. The increment will be performed after all other operations in the expressions, including the indirection operation. Then the indirection operation is evaluated. Since the incrementation is postfix, the indirection operation is performed before the address is incremented.

In the following example, the combined indirection and increment operations are broken down into their equivalent statements. Doing this can often help you keep straight what is being incremented and when.

```
*numptr++;          *numptr;
                    numptr = numptr + 1;
```

The differences become a bit more clear when you use the combination as part of a larger expression.

```
*numptr++ = 5;          *numptr = 5;
                        numptr = numptr + 1;
```

To clarify such operations, it is advisable to use parenthesis to determine the sequence of evaluation, instead of relying on assciativity. Parenthesis will force the evaluation of one operation before another. In the next example, parenthesis aroud the indirection operation clearly indicate that the indirection will be performed before the increment.

```
(*numptr)++ = 5;
```

If you place the increment before the pointer variable, it becomes a prefix operation. In this case it is executed before any other operations in the expression. In the following example, the pointer variable is incremented first. Indirection will then operate on the new address. Notice that, though the increment is place before the variable, it still comes after the increment operator, `*++numptr`. The right to left associativity will still evaluate the increment operation first, making the increment an operation on the pointer variable.

```
*++numptr;          numptr = numptr + 1
                    *numptr;
```

Again, use of parenthesis will clarify the sequence of operations.

```
*(++numptr);
```

In Listing 10.14, there are two examples of this combination of indirection and postfix increment of a pointer. The increment and indirection on `numptr` now both take place in the `printf` statement. However, in the **postinc.c** program in 10.14A, the increment is a postfix operation, and in

the **preinc.c** program in 10.14B, the increment is a prefix operation. Notice that in 10.14B, the increment takes place before the first element is printed out. The first element is never printed.

LISTING 10.14A

postinc.c

```
#include <stdio.h>

int main(void)
{
int mynums[2] = {12,44};
int *numptr;


numptr = mynums;
while(numptr<(mynums+2))
      {
      printf("%d",*numptr++);
      }
}
```

LISTING 10.14B

preinc.c

```
#include <stdio.h>

int main(void)
{
int mynums[2] = {12,44};
int *numptr;


numptr = mynums;
while(numptr<(mynums+2))
      {
      printf("%d",*++numptr);
      }
}
```

If you should place the increment operator to the left of the indirection operator, the element referenced is incremented instead of the pointer variable. The right to left associativity will the first evaluate the indirection operation, referencing the object. The variable pointed to is referenced. The increment operation is then evaluated and operates on the referenced object. That variable is then incremented. In the next example, the integer variable referenced by `numptr`, not the address in `numptr`, is incremented.

`++*numptr;` is equivalent to `(*numptr) = (*numptr) + 1;`

The same kind of combinations can work for decrements. For example:

```
*numptr--;        *numptr;
                  numptr = numptr - 1;
```

Like the increment operator, the decrement can be either a postfix or prefix operation. Here is an example of the prefix decrement operation.

```
*--numptr;        numptr = numptr - 1
                  *numptr;
```

If you should place the decrement operator before the indirection operator, then the object pointed to is decremented, not the pointer.

`--*numptr;` is equivalent to `(*numptr) = (*numptr) - 1;`

Using parenthesis will clarify the sequence.

```
--(*numptr);
```

In the **predec.c** program in Listing 10.15, a combined decrement an indirection operation is used in the `printf` statement to both move to the previous element in the array and to reference that element, `*--numptr`. Notice, that, unlike Listing 10.14, `numptr` is set to the end address of the array, not the address of the last element. This can be done because the decrement is a prefix operation taking effect before the first indirection. `numptr` is first decremented back to the address of the last array element and only then does indirection references that element. Using the `mynums` example in Figure 10.9, `numptr` is set to 510, then decremented to 508 (the address of the last element), before the indirection takes place.

Notice also that the test for the beginning of the array uses only a > operator instead of a >= operator in Listing 10.14. With postfix and prefix operations you need to be very careful to not to either stop too soon or stop too late at the end of the array.

LISTING 10.15

predec.c

```
#include <stdio.h>
#define MAX 5

int main(void)
      {
      int mynums[MAX] = { 9, 87, 95, 23, 45 };
      int *numptr;

      numptr = mynums + MAX;
      while (numptr > mynums)
            {
            printf("%d\n", *--numptr);
            }

      return 0;
      }
```

Indexes: Pointer Differences

An index is the position of an element in an array. In C, array elements are indexed from zero. For example, the third element of an array has an index of 2, and the first element has an index of 0. You could find yourself in a situation in which you have the address of an element, but not its index. In such a case you can use the pointer difference operation to calculate that element's index. In a pointer difference operation one pointer is subtracted from another. The subtraction results in an integer value that is the difference between two pointers. It does not result in an address. Technically, this integer value is the number of objects between two addresses. Though pointer difference can operate between any two addresses, it was designed to operate on arrays, determining the index of an element given only its address. To do so the array name is used as one operand and the element address as the other. The array name is then subtracted from the address of an element. The result is the index of that element in the array.

working_pointer – array name

Usually you will have a working pointer that holds the address of the element. You then subtract the array name from the working pointer. In the next example, the index is calculated by subtracting the array name `mynums` from the pointer variable `numptr`. The result is an integer and this value is assigned to the integer variable rindex.

```
      rindex = numpter - mynums;
```

The **index.c** program in Listing 10.16 performs the same calculation. First, the pointer variable `numptr` is assigned the address of the fourth element, `mynums[3]`. The index of the element is then determined by a subtraction of `mynums` from `numptr`. This index result is assigned to the integer rindex. which is then used to in array subscripts to reference and print out the fourth element. Notice that rindex is declared as an integer, not a pointer. The result of a pointer difference expression is an integer, not a pointer value.

LISTING 10.16

index.c

```
#include <stdio.h>

int main(void)
      {
      int mynums[5] = { 9, 87, 95, 23, 45 };
      int rindex;
      int *numptr;

      numptr = mynums + 3;

      rindex = numptr - mynums;

      printf("%d %d\n", mynums[rindex] ,mynums[3]);

      printf("%d %d\n", *(mynums+rindex), *(mynums+3));

      return 0;
      }
```

Assuming the `mynums` array as depicted in Figure 10.11, the **index.c** program in Listing 10.16 would print out the value of the fourth array element.

```
23    23
23    23
```

The offset expression `mynums+3` assigns the address 506 to `numptr`. 506 is the address of the fourth element in the `mynums` array. The pointer subtraction of the array name `mynums` from `numptr` results in the integer value 3, which is the offset and the index for the fourth element of the `mynums` array. Notice that in this situation, `mynums[rindex]`, `mynums[3]`, `*(mynums+rindex)`, `*(mynums+3)`, and `*numptr` all reference the same array element.

In a pointer difference operation, it is important to realize that there is a hidden division by the size of the pointer's data type. In Figure 10.11, a subtraction of `mynums` from `numptr` further divides the difference by the size of an integer, 2 bytes. In this sense, pointer difference always results in the

number of elements between two addresses, not the number of bytes. As shown in figure 10.11, the subtraction of `mynums` from `numptr` results in the integer value 2.

```
(numptr - mynums)    is          2
                                 (504 - 500) / sizeof(int)
                                 4 / sizeof(int);
                                 4 / 2 is 2
```

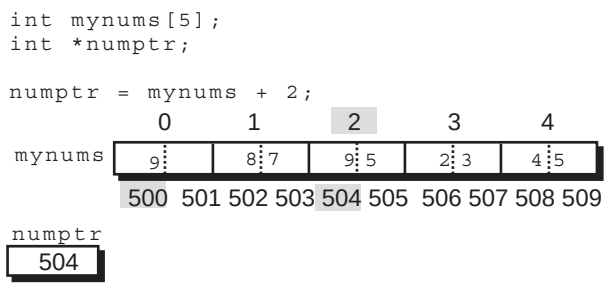

Figure 10.11. The index operation: pointer difference.

The **numidx.c** program in Listing 10.17 uses pointer difference to calculate each array index and then uses that index to reference and print out each element in the `mynums` array. The address of each element is held by the pointer `numptr`. `mynums` is then subtracted from `numptr`, and the result is used as an array index. The array index is used with pointer notation to print out an element of the array.

LISTING 10.17

numidx.c

```
#include <stdio.h>

int main(void){
      int i = 0;
      int mynums[5] = { 9, 87, 95, 23, 45 };
      int *numptr;
      int index;

      numptr = mynums;

      while (numptr < (mynums + 5))
            {
            index = numptr - mynums;
            printf("%d %d %p %p\n",
                  index, mynums[index], numptr, mynums);
            numptr++;
            }

      return 0;
      }
```

Assuming that the address of the `mynums` array is 500, the **numidx.c** program in Listing 10.17 would print out:

```
0     9     500   500
1     87    502   500
2     95    504   500
3     23    506   500
4     45    508   500
```

TABLE I OFFSETS, INCREMENTS, and INDEXES

OFFSETS

Addition or subtraction of pointer with arithmetic integer value. Implied multiplication by `sizeof(`*type*`)` of arithmetic value.

pointer + *integer*

pointer − *integer*

```
(numptr + 2)     numptr + (2 * sizeof(int))
(numptr - 2)     numptr - (2 * sizeof(int))
```

INCREMENTS AND DECREMENTS

Addition or Subtraction of 1 multiplied by `sizeof(`*type*`)`. Increment to next element in array, or decrement to previous element. Actual increment or decrement by size of type.

pointer_variable++

pointer_variable−−

```
numptr++;  numptr = numptr + sizeof(int);
numptr--;  numptr = numptr - sizeof(int);
```

INDEXES

Difference of two pointers or pointer values. Implied division by `sizeof(`*type*`)`. Subtract a array name from working pointer to obtain index of an element.

working_pointer − *array name*

pointer expression − *pointer expression*

```
(numptr - mynums)      (numptr - mynums) / sizeof(int)
(numptr-(mynums+2))
(numptr - (mynums + (2 * sizeof (int))) / sizeof(int)
```

Pointer difference is not limited to using array names to calculate indexes. The addresses used in the pointer difference operation can be derived from any pointer expression. However, there is one important limitation. The pointers used as operands in a pointer difference operation must have the same data type. You could not subtract the address of an integer from the address of a float. In the

indexop.c program in Listing 10.18, several different pointer expression are used as operands in a pointer difference operation. The result assigned to `rindex` is always 4. First an array name is subtracted from a pointer variable, then a pointer variable is subtracted from another pointer variable, and finally an array name is subtracted from the address resulting from a pointer offset expression.

LISTING 10.18

indexop.c

```
#include <stdio.h>

int main(void)
      {
      int mynums[5] = { 9, 87, 95, 23, 45 };
      int *lastptr;
      int *firstptr;
      int rindex;

      lastptr = (mynums + 4);
      rindex = lastptr - mynums;
      firstptr = mynums;
      rindex = lastptr - firstptr;
      rindex = (mynums + 4) - mynums;

      return 0;
      }
```

Arrays and Functions

Often you will want to pass an array from one function to another, either to reference its values or to work on its elements. However arrays cannot be passed from one function to another as variables are. The values of an array as a whole cannot be passed to a function. Arrays are not variables. This means that call-by-value operations cannot be performed on an array as a whole. Call-by-value assumes that there is a corresponding parameter variable in which to place the argument's values. The array name is only an address, it is not a variable.

However, any function could access elements in an array using pointer offset operations if the beginning address is known. The array name is that beginning address. You can pass this beginning address into a function that has a parameter declared as a pointer variable. This effectively implements a call-by-reference operation for the array, passing the address of the array from one function to another.

When an array name is used as an argument, it is passing a reference for the array. It is not passing a value. This means that a parameter can never be declared as an array. This fact is confused by the fact that there is an array subscript that is often used in parameter declarations. You can declare a parameter using a set of empty brackets. But this notation is simply translated into a pointer declaration. Through this pointer, you can then reference the elements of the array in the calling function.

`int parray[]` is equivalent to `int *parray`

LISTING 10.19

printnum.c

```
#include <stdio.h>
#define MAX 3

void printarray(int[], int);

int main(void)
      {
      int mynums[MAX] = {12, 5, 27};

      printarray(mynums, MAX);

      return 0;
      }

void printarray(int parray[], int max)
      {
      int i;

      for (i = 0; i < max; i++){
            printf("%d\n", parray[i]);
            }
      }
```

In the **printnum.c** program in Listing 10.19, the array name `mynums` is an argument in the function call to printarray. The array name is the address of the `mynums` array. This address is passed to parray, which is a pointer variable. Here, however, the array subscript is used to declare the pointer variable. Array subscript is then used in the function to reference elements of the `mynums` array.

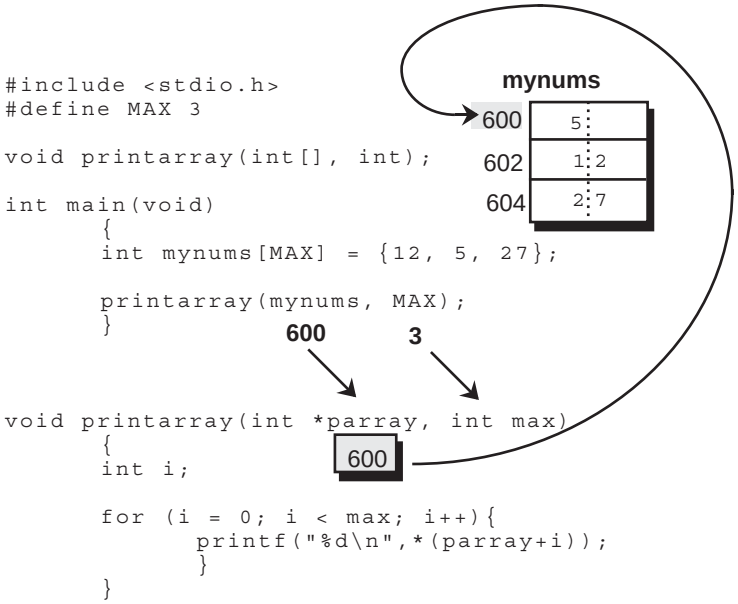

Figure 10.12. Arrays and Functions, passing an address of an array to a function.

The array subscript used in parameter declarations is only a notation. It is really only an alternative way of writing the declaration of a pointer variable. Array subscript and pointer offset operations can be applied interchangeably to a pointer declared in either form. In Listing 10.20, an array name is passed to a parameter pointer. In the **parptr.c** program in Listing 10.20A the parameter variable is declared using the pointer type. In the **parnote.c** program in Listing 10.20B, the parameter variable is declared using the array subscript. Offset operations and array subscripts are used in each case to reference array elements.

LISTING 10.20A

parptr.c

```
#include <stdio.h>

void add (int*);

int main(void)
      {
      int mynums[5];

      add(mynums);
      }

void add (int *numptr)
      {
      *(numptr + 2) = 35;
      numptr[2] = 35;
      }
```

10.20B

parnote.c

```
#include <stdio.h>

void add(int[]);

int main(void)
      {
      int mynums[5];

      add(mynums);
      }

void add(int numptr[])
      {
      numptr[2] = 35;
      *(numptr + 2) = 35;
      }
```

In the **ptrprint.c** program Listing 10.21, the program in Listing 10.19 is rewritten using a pointer type for the parameter declaration instead of array subscripts. Elements of the array are referenced with a combined offset and indirection operation.

LISTING 10.21

ptrprint.c

```
#include <stdio.h>
#define MAX 3

void printarray(int[], int);

int main(void)
      {
      int mynums[MAX] = {12, 5, 27};

      printarray(mynums, MAX);

      return 0;
      }

void printarray(int *parray, int max)
      {
      int i;

      for (i = 0; i < max; i++){
            printf("%d\n",*(parray+i));
            }
      }
```

The pointer declared as a parameter is a variable itself. As a variable, its values can be changed. In several situations you can use the parameter pointer that references an array as a working pointer to advance down the array from one element to the next. The parameter pointer will be incremented using the increment operator. Of course, the original beginning address of the array is lost. However, if the function only requires a single sequential use of the array elements, it will not matter if the beginning address is lost. On the other hand, in most situations the beginning address should be preserved. In the **incprint.c** program in Listing 10.22, the parray pointer is incremented as a working pointer advancing from one element to the next.

LISTING 10.22

incprint.c

```
#include <stdio.h>
#define MAX 3

void printarray(int[], int);

int main(void)
      {
      int mynums[MAX] = {12, 5, 27};

      printarray(mynums, MAX);

      return 0;
      }

void printarray(int *parray, int max)
      {
      int i;

      for(i=0; i < max; i++)
            {
            printf("%d\n",*parray);
            parray++;
            }
      }
```

In the **constprt.c** program in Listing 10.23, the address in the parameter pointer is preserved, and a separate working pointer that will advance through the array is declared. Notice that the parameter pointer is declared with the const qualifier. This prevents the writing of any code that attempts to modify the parameter pointer. It becomes a constant pointer. The const qualifier is placed after the pointer's data type and before the pointer type, *. This indicates that the pointer, not the object it points to, is being qualified. The address in the pointer itself cannot be changed, not the object it points to.

LISTING 10.23

constprt.c

```
#include <stdio.h>
#define MAX 3

void printarray(int const*, int);

int main(void)
      {
      int mynums[MAX] = {12, 5, 27};

      printarray(mynums, MAX);

      return 0;
      }

void printarray(int const *parray, int max)
      {
      int *wkptr;

      for(wkptr=(int*)parray; wkptr<(parray+max); wkptr++)
            {
            printf("%d\n",*wkptr);
            }
      }
```

Controlling modification of Arrays: const

Because you can only pass the address of an array, not its values, an array is open to modification by any function given that address. An array's call-by-reference operation allows a programmer to the write code that modifies it, in any function it is passed to. Sometimes the array elements need to be changed, as in the case of the copy program in Listing 10.24. The elements of the array copynums in main will have their values changed through pointer references in the function copyarray.

However, there are situations in which the elements of an array should not be changed in a function. In the copy program, code that changes the elements of the source array, in this case `mynums`, should never be written. You can forcibly prevent such modification by qualifying the parameter pointer declaration with the const type qualifier. The `const` type qualifies a variable as a constant, preventing its value from being changed. It can easily be applied to array elements. In the **constcpy.c** program in Listing 10.24, the `const` modifier is used to prevent the `sarray` pointer from ever changing the source array. In effect, only the value of the source array, `mynums`, can be referenced.

The `const` qualifier can be used in two different ways: one to qualify the pointer itself, and the other to qualify the object pointed to.

`const int *sptr;`	Pointed-to object cannot change, but pointer can change
`int const *sptr;`	Pointed-to object can change, but pointer cannot
`const int const *ptr`	Neither pointed-to object nor pointer can change

LISTING 10.24

constcpy.c

```
#include <stdio.h>
#define MAX 3

void copyarray(int [], const int [], int);

 int main(void)
      {
      int mynums[MAX] = {12, 5, 27};
      int copynums[MAX];

      copyarray (copynums, mynums, MAX);

      return 0;
      }

 void copyarray(int tarray[],const int sarray[],int num)
      {
      int i;
      for (i = 0; i < num; i++)
            {
            tarray[i] = sarray[i];
            }
      }
```

In the **constsrc.c** program in Listing 10.25, the program in Listing 10.24 is rewritten using pointer declarations for the parameters in the definition of `copyarray`. In the declaration of `sarray`, the const modifier affects the elements referenced, not sarray itself. The source array is referenced by the pointer `sarray`. The `const` qualifier placed before the pointer's data type will prevent the pointer from modifying the object it points to. `sarray` can change its value. It can be set to another address. However, the elements it references through indirection cannot be changed. The expression `*sarray = 10;` is not allowed.

LISTING 10.25

constsrc.c

```
#include <stdio.h>
#define MAX 3
void copyarray(int*, const int*, int);

 int main(void)
      {
      int mynums[MAX] = {12, 5, 27};
      int copynums[MAX];

      copyarray (copynums, mynums, MAX);

      return 0;
      }

void copyarray(int *tarray, const int *sarray, int num)
      {
      int i;
      for (i = 0; i < num; i++)
            {
            tarray[i] = sarray[i];
            }
      }
```

Passing Array Elements: call-by-value and call-by-reference

Though the values of an array as a whole cannot be passed, you can pass the values of individual array elements. In a one-dimensional array, the reference of an array element is equivalent to a variable reference. When used as an argument, the value of the referenced element is passed. In this case, the corresponding parameter is a variable with the same data type as that of the element. In the **printele.c** program in Listing 10.26, an element of an array is used as an argument. The value of the first element, 12, is passed into an integer parameter variable.

LISTING 10.26

printele.c

```
#include <stdio.h>

#define MAX 3

void printnum(int);

int main(void)
      {
      int mynums[MAX] = {12, 5, 27};

      printnum(mynums[1]);

      return 0;
      }

void printnum(int num)
            {
            printf("%d ", num);
            }
```

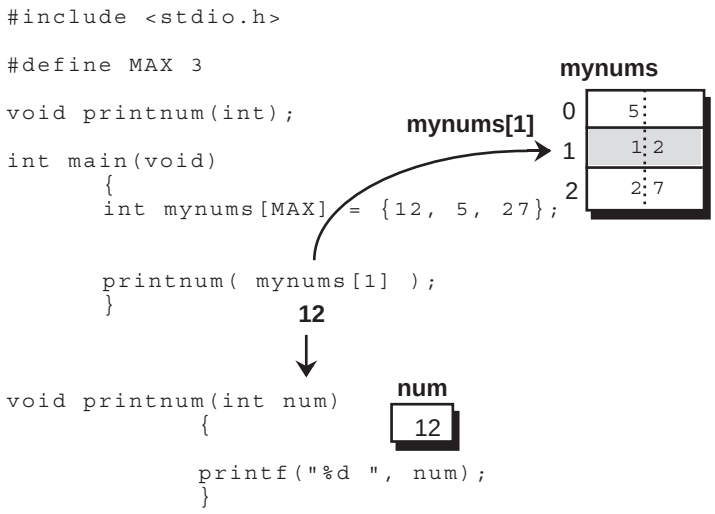

Figure 10.13. Array elements as arguments.

There may be situations in which you may need to modify an array element. In this case, you would have to perform a call-by-reference operation on the array element, passing its address to a pointer variable. You can obtain the address of an element by applying the address operation to a reference of the element. Usually you only need to place the address operator before an array subscript reference of the element. The following example obtains the address of the third element in the `mynums` array.

```
      &mynums[2]
```

In the program **getnums.c** in Listing 10.27, the user is allowed to enter in the values for the `mynums` array. This requires a call to scanf. However, scanf always requires the address of an object. In this case, the object is first referenced with array subscript, and then its address is obtained with the address operator.

LISTING 10.27

getnums.c

```
#include <stdio.h>

int main(void)
            {
            int mynums[5];
            int i;

            for (i = 0; i < 5; i++)
                  {
                  printf("Enter number :");
                  scanf("%d", &mynums[i]);
                  }

            for (i = 0; i < 5; i++)
                  {
                  printf("%d \n", mynums[i]);
                  }

            return 0;
            }
```

There is, of course a shortcut version of this process. Array subscripts is really equivalent to a pointer offset and indirection operation. In the program **getnoff.c** in Listing 10.28, the previous program is written with the pointer offset `mynums+i` rather than with the array subscript and address operation `&num[i]`. The offset operation `mynums+i` results in the address of that element. scanf only requires the address. It does not matter how the address is obtained.

`num[i]`	is equivalent to	`*(mynums + i)`
`&num[i]`	is equivalent to	`&(*(mynums + i))`
`&(*(mynums+i))`	is equivalent to	`(mynums + i)`

LISTING 10.28

getnoff.c

```
#include <stdio.h>

int main(void)
      {
      int mynums[5];
      int i;

      for (i = 0; i < 5; i++)
            {
            printf("Enter number :");
            scanf("%d", (mynums + i));
            }

      for (i = 0; i < 5; i++)
            {
            printf("%d \n",  *(mynums + i));
            }

      return 0;
      }
```

Chapter Summary: Arrays and Pointers

An array is a collection of objects, but it is not an object itself. The objects in an array are referred to as elements. Elements are numbered starting from zero. An array of five elements is numbered 0 to 4. Each element is referenced according to its position in the array. An element can be referenced using array subscripts, which consists of the array name and the position of the element enclosed in brackets.

An array is declared with a data type, an array name, the array data type, and the number of objects in the array. The array data type is a set of brackets. Brackets mean different things depending upon where they are used. Brackets are used in declarations, statements, and parameter declarations. In each situation, brackets have a different meaning. In declarations, brackets denote the array data type used to declare an array. In statements, brackets constitute an array subscript that represents a pointer offset and indirection operation. In parameter declarations, the array declaration is converted to a pointer declaration.

The array name itself is an address of the first element in the array. The array name can be thought of as a pointer constant. You use the array name in pointer operations to reference objects in the array.

Pointer offsets and indexes were designed to work on arrays. An offset is an expression in which an integer is either added to or subtracted from a pointer. The increment operation advances a pointer from one element to the next in an array. An increment simply consist of resetting a pointer to itself with an offset of 1. Indexes are derived by subtracting one pointer from another. The result is the difference, in terms of the number of elements between the two pointers.

The increment operation is often used for a working pointer that advances down the array from one object to the next. The end address of the array can be calculated with an offset expression consisting of the array name and the number of objects in the array. The end address of the array declared as `int mynums[5]` is `mynums + 5`. The working pointer can be tested against this end address as it advances down the array.

Arrays can only be passed to functions in a call-by-reference operation. However, you can use the const qualifier to either protect the parameter pointer that holds the beginning address or to protect the array in the calling function whose address was passed.

Exercises

Using the program that follows, replace the variable i altogether. Instead of comparing i to 10, compare `numptr` to a pointer offset operation that yields the last address in the array. The first address in the array is `mynums`. The last is `mynums` plus an offset of 10. Also, implement the increment of `numptr` with the increment operator, ++.

Offsets can be negative as well as positive. Rewrite the program, filling up the array in reverse order (decrement from 9 to 0). Use the decrement operator, --.

Add a search process which requests a number to be searched and then prints out the index of that number in the array. Remember that the subtraction of two pointers will result in an index to an array. The address of the array is subtracted from the address of an element in the array. If `numptr` is pointing to the third element, then the expression (`numptr-mynums`) results in the integer 2. This is often helpful in a search.

```
#include <stdio.h>

int main(void)
      {
      int mynums[10], i = 0;
      int * numptr;

      numptr = mynums;
      while (i < 10)
            {
            printf("Enter number :");
            scanf("%d", numptr);
            numptr = numptr + 1;
            i++;
            }

      for(numptr=mynums,i=0;i<10;numptr=numptr+1,i++)
            {
            printf("%d \n", *numptr);
            }

      return 0;
      }
```