EXAMPLE 7.11 Examining the Addresses of Array Elements

```
int main()
{ short a[] = {22, 33, 44, 55, 66};
  cout << "a = " << a << ", *a = " << *a << endl;
  for (short* p = a; p < a + 5; p++)
    cout << "p = " << p << ", *p = " << *p << endl;
}
```

```
    a = 0x3fffd08, *a = 22
    p = 0x3fffd08, *p = 22
    p = 0x3fffd0a, *p = 33
    p = 0x3fffd0c, *p = 44
    p = 0x3fffd0e, *p = 55
    p = 0x3fffd10, *p = 66
```

Initially, `a` and `p` are the same: they are both pointers to `short` and they have the same value (`0x3fffd08`). Since `a` is a constant pointer, it cannot be incremented to traverse the array. Instead, we increment `p` and use the exit condition `p < a + 5` to terminate the loop. This computes `a + 5` to be the hexadecimal address `0x3fffd08 + 5*sizeof(short) = 0x3fffd08 + 5*2 = 0x3fffd08 + 0xa = 0x3fffd12`, so the loop continues as long as `p < 0x3fffd12`.

The array subscript operator `[]` is equivalent to the dereference operator `*`. They provide direct access into the array the same way:

```
a[0] == *a
a[1] == *(a + 1)
a[2] == *(a + 2), etc.
```

So the array a could be traversed like this:

```
for (int i = 0; i < 8; i++)
cout << *(a + i) << endl;
```

The next example illustrates how pointers can be combined with integers to move both forward and backward in memory.

EXAMPLE 7.12 Pattern Matching

In this example, the `loc` function searches through the first `n1` elements of array `a1` looking for the string of integers stored in the first `n2` elements of array `a2` inside it. If found, it returns a pointer to the location within `a1` where `a2` begins; otherwise it returns the `NULL` pointer.

```
short* loc(short* a1, short* a2, int n1, int n2)
{ short* end1 = a1 + n1;
  for (short* p1 = a1; p1 < end1; p1++)
    if (*p1 == *a2)
    { int j;
      for (j = 0; j < n2; j++)
        if (p1[j] != a2[j]) break;
      if (j == n2) return p1;
    }
  return 0;
}

int main()
{ short a1[9] = {11, 11, 11, 11, 11, 22, 33, 44, 55};
```