2.2.5 Pointers, Arrays, and Loops

An array of elements of type **char** can be declared like this:

```
char v[6];          // array of 6 characters
```

Similarly, a pointer can be declared like this:

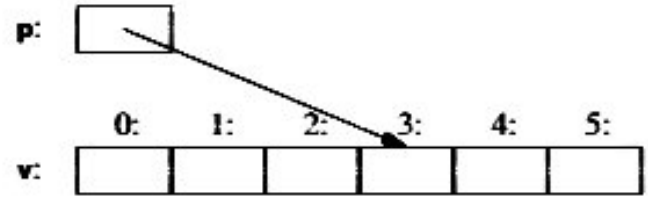
```
char* p;            // pointer to character
```

In declarations, **[]** means "array of" and * means "pointer to." All arrays have **0** as their lower

bound, so **v** has six elements, **v[0]** to **v[5]**. The size of an array must be a constant expression (§2.2.3). A pointer variable can hold the address of an object of the appropriate type:

```
char* p = &v[3];    // p points to v's fourth element
char x = *p;        // *p is the object that p points to
```

In an expression, prefix unary * means "contents of" and prefix unary **&** means "address of." We can represent the result of that initialized definition graphically:



Consider copying ten elements from one array to another:

```
void copy_fct()
{
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10];                 // to become a copy of v1

    for (auto i=0; i!=10; ++i)  // copy elements
        v2[i]=v1[i];
    // ...
}
```

This **for**-statement can be read as "set **i** to zero; while **i** is not **10**, copy the **i**th element and increment **i**." When applied to an integer variable, the increment operator, **++**, simply adds **1**. C++ also offers a simpler **for**-statement, called a range-**for**-statement, for loops that traverse a sequence in the simplest way:

```
void print()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto x : v)            // for each x in v
        cout << x << '\n';

    for (auto x : {10,21,32,43,54,65})
        cout << x << '\n';
    // ...
}
```

The first range-for-statement can be read as "for every element of v, from the first to the last, place a copy in x and print it." Note that we don't have to specify an array bound when we initialize it with a list. The range-for-statement can be used for any sequence of elements (§3.4.1).

If we didn't want to copy the values from v into the variable x, but rather just have x refer to an element, we could write:


```
void increment()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto& x : v)
        ++x;
    // ...
}
```

In a declaration, the unary suffix & means "reference to." A reference is similar to a pointer, except that you don't need to use a prefix * to access the value referred to by the reference. Also, a reference cannot be made to refer to a different object after its initialization. When used in declarations, operators (such as &, *, and []) are called *declarator operators*:

```
T a[n];     // T[n]: array of n Ts (§7.3)
T* p;       // T*: pointer to T (§7.2)
T& r;       // T&: reference to T  (§7.7)
T f(A);     // T(A): function taking an argument of type A returning a result of type T (§2.2.1)
```

We try to ensure that a pointer always points to an object, so that dereferencing it is valid. When we don't have an object to point to or if we need to represent the notion of "no object available" (e.g., for an end of a list), we give the pointer the value nullptr ("the null pointer"). There is only one nullptr shared by all pointer types:

```
double* pd = nullptr;
Link<Record>* lst = nullptr;    // pointer to a Link to a Record
int x = nullptr;                // error: nullptr is a pointer not an integer
```

It is often wise to check that a pointer argument that is supposed to point to something, actually points to something:

```
int count_x(char* p, char x)
    // count the number of occurrences of x in p[]
    // p is assumed to point to a zero-terminated array of char (or to nothing)
{
    if (p==nullptr) return 0;
    int count = 0;
    for (; *p!=0; ++p)
        if (*p==x)
            ++count;
    return count;
}
```