**POINTER AND ARRAYS**

**1) POINTERS AND ADDRESSES**

A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. A pointer is a group of cells that can hold an address. So if **c** is a char and **p** is a pointer that points to it, we could represent the situation this way:



The unary operator **&** gives the “**address of**” an object, so the statement

p = &c;

assigns the address of **c** to the variable **p**, and **p** is said to “**point to**” **c**. The **&** operator only applies to objects in memory. It cannot be applied to expressions, constants, or register variables.

The unary operator **\*** is the **indirection** or **deferencing** operator; when applied to a pointer, it accesses the object the pointer points to.

int x = 1, y = 2, z[10];

int \*ip; /\* ip is a pointer to int \*/

ip = &x; /\* ip now points to x\*/

y = \*ip; /\* y is now 1\*/

\*ip = 0; /\* x in now 0 \*/

ip = &z[0]; /\* ip now points to z[0] \*/

If **ip** points to the integer **x,** then **\*ip** can occur in any context where **x** could, so

x = x + 10;

could be written as

\*ip = \*ip + 10;

**2) POINTERS AND ARRAYS**

The declaration

int a[10];

defines an array a of size 10, that is, a block of 10 consecutive objects named **a[0], a[1], a[2],.....a[9].**



The notation **a[i]** refers to the **i-th** element of the array.

If **pa** is a pointer to an integer declared as

int \*pa;

then the assignment

pa = &a[10];

sets **pa** to point to element zero of **a**; that is, **pa** contains the address of **a[0]**.



Now the assignment

x = \*pa;

will copy the contents of **a[0]** into **x**.

If **pa** points to a particular element of an array, then by definition **pa+1** points to the next element, **pa+i** points **i** elements after **pa**, and **pa-i** points **i** elements before. Thus **pa** points to **a[0]**,

\*(pa + 1)

refers to the contents of **a[1]**, **pa+i** is the address of **a[i]**, and **\*(pa+i)** is the contents of **a[i]**.



pa = &a[0];

can also be written as

pa = a;

a[i] ≡\*(a+i)

apply & operator to both sides

&a[i] ≡a+i

pa[i] ≡\*(pa+i)

In short, an array-and-index expression is equivalent to one written as a pointer and offset.

When an array name is passed to a function, what is passed is the location of the initial element.

**3) ADDRESS ARITHMETIC**

If **p** is a pointer to some element of an array, then **p++** increments **p** to point to the next element, and **p+=i** increments it to point **i** elements beyond where it currently does.

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language.

Let us illustrate by writing a rudimentary storage allocator. There are two routines.

1) **alloc( n ),** returns a pointer **p** to **n** consecutive character positions, which can be used by the caller

of **alloc** for storing characters.

2) **afree( p ),** releases the storage thus acquired so it can be re-used later.

The routines are “rudimentary” because the calls to **afree** must be made in the opposite order to the calls made on **alloc**. That is, the storage managed by **alloc** and **afree** is a stack, or last-in, first-out list. The standard library provides analogous functions called **malloc** and **free** that have no such restrictions. The easiest implementation is to have **alloc** hand out pieces of a large character array that we will call **allocbuf**. This array is private to **alloc** and **afree**. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared **static** in the source file containing **alloc** and **afree**, and thus be invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained by calling **malloc** or by asking the operating system for a pointer to some unnamed block of storage. The other information needed is how much of **allocbuf** has been used. We use a pointer, called **allocp**, that points to the next free element. When **alloc** is asked for **n** characters, it checks to see if there is enough room left in **allocbuf**. If so, **alloc** returns the current value of **allocp** (i.e., the beginning of the free block), then increments it by **n** to point to the next free area. If there is no room, **alloc** returns zero. **afree(p)** merely sets **allocp** to **p** if **p** is inside **allocbuf**.



#define ALLOCSIZE 10000 /\* size of available space \*/

static char allocbuf[ALLOCSIZE]; /\* storage for alloc \*/

static char \*allocp = allocbuf; /\* next free position \*/

char \*alloc(int n) /\* return pointer to n characters \*/

{

if (allocbuf + ALLOCSIZE - allocp >= n)

{ /\* it fits \*/

allocp += n;

return allocp - n; /\* old p \*/

}

else /\* not enough room \*/

return 0;

}

void afree(char \*p) /\* free storage pointed to by p \*/

{

if (p >= allocbuf && p < allocbuf + ALLOCSIZE)

allocp = p;

}

C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event.

Pointers and integers are not interchangeable. **Zero** is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic constant **NULL** is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. **NULL** is defined in **<stdio.h>.**

Pointers may be compared under certain circumstances. If **p** and **q** point to members of the same array, then relations like ==, !=, <, >=, etc., work properly.

For example ,

p < q

is true if **p** points to an earlier member of the array than **q** does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behaviour is undefined for arithmetic or comparisons with pointers that do not point to members of the same array.

p + n

means the address of the **n-th** object beyond the one **p** currently points to . This is true regardless of the kind of object **p** points to; **n** is scaled according to the size of the objects **p** points to, which is determined by the declaration of **p**. If an int is of four bytes, for example, the int will be scaled by four.

All the pointer manipulations automatically take into account the size of the object pointed to.

The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal.

**4) CHARACTER POINTER AND FUNCTIONS**

A string constant, written as “I am a string” is an array of characters. In the internal representation, the array is terminated with the null character ‘\0’ so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

If **pmessage** is declared as

char \*pmessage;

then the statement

pmessage = "now is the time";

assigns to **pmessage** a pointer to the character array. This is *not* a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions:

char amessage[] = “now is the time”; /\* an array \*/

char \*pmessage = “now is the time”; /\* a pointer \*/

**amessage** is an array, just big enough to hold the sequence of characters and '\0' that initializes it. Individual characters within the array may be changed but **amessage** will always refer to the same storage. On the other hand, **pmessage** is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.



We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is **strcpy(s,t)**, which copies the string **t** to the string **s**. It would be nice just to say **s=t** but this copies the pointer, not the characters. To copy the characters, we need a loop.

The array version first:

/\* strcpy: copy t to s; array subscript version \*/

void strcpy(char \*s, char \*t)

{

int i;

i = 0;

while ((s[i] = t[i]) != '\0')

i++;

}

For contrast, here is a version of strcpy with pointers:

/\* strcpy: copy t to s; pointer version \*/

void strcpy(char \*s, char \*t)

{

int i;

i = 0;

while ((\*s = \*t) != '\0')

{

s++;

t++;

}

}

Because arguments are passed by value, **strcpy** can use the parameters s and t in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the **'\0'** that terminates **t** has been copied into **s**. In practice, **strcpy** would not be written as we showed it above.

Experienced C programmers would prefer

/\* strcpy: copy t to s; pointer version 2 \*/

void strcpy(char \*s, char \*t)

{

while ((\*s++ = \*t++) != '\0')

;

}

This moves the increment of **s** and **t** into the test part of the loop. The value of **\*t++** is the character that **t** pointed to before **t** was incremented; the postfix **++** doesn't change **t** until after this character has been fetched. In the same way, the character is stored into the old **s** position before **s** is incremented. This character is also the value that is compared against **'\0'** to control the loop. The net effect is that characters are copied from t to s, up and including the terminating **'\0'**.

As the final abbreviation, observe that a comparison against **'\0'** is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

/\* strcpy: copy t to s; pointer version 3 \*/

void strcpy(char \*s, char \*t)

{

while (\*s++ = \*t++)

;

}

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs.

**5) MULTI-DIMENSIONAL ARRAY**

In C, a two- dimensional array is really a one-dimensional array, each of whose elements is an array.

a[i][j] /\* [row][col]\*/

elements are stored by rows

int a[2][4] =

{

{1, 2, 3, 4},

{5, 6, 7, 8}

};

More generally, only the first dimension (subscript) of an array is free; all the others have to be specified.

Since brackets [] have higher precedence than \*.

int \*a[4] /\* array of 4 pointers to integers \*/

int (\*a)[4] /\* pointer to an array of 4 integers \*/

Compare the declaration and picture for an array of pointers with those for a two-dimensional array. The important advantage of the pointer array is that the rows of the array may be of different lengths.

char \*name[] = { "Illegal month", "Jan", "Feb", "Mar" };



char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };



**6)** **POINTERS TO FUNCTIONS**

int \*func( int, int ) /\* func is a function having two integers as its arguments and is returning pointer to an integer \*/

int ( \*func )( int, int ) /\* func is a pointer to a function that has two integer arguments and returns an integer\*/