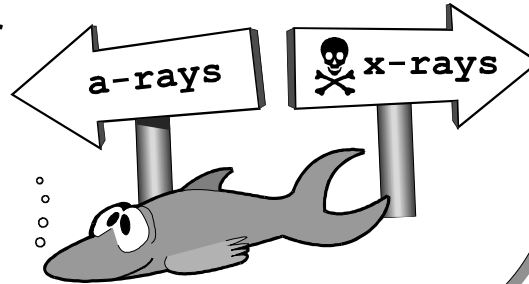


## Arrays in C

- Declaring arrays
- Accessing elements
- Passing arrays into functions
- Using pointers to access arrays
- Strings
- The null terminator



© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 1

---

## Arrays in C

This chapter discusses arrays in C.

## Declaring Arrays

- An array is a collection of data items (called *elements*) all of the same type
- It is declared using a type, a variable name and a **CONSTANT** placed in square brackets
- C always allocates the array in a single block of memory
- The size of the array, once declared, is fixed forever - there is no equivalent of, for instance, the “redim” command in BASIC

---

## Declaring Arrays

An important fact to understand about arrays is that they consist of the same type all the way through. For instance, an array of 10 **int** is a group of 10 integers all bunched together. The array doesn't change type half way through so there are 5 **int** and 5 **float**, or 1 **int**, 1 **float** followed by 1 **int** and 1 **float** five times. Data structures like these could be created in C, but an array isn't the way to do it.

Thus to create an array we merely need a type for the elements and a count. For instance:

```
long a[5];
```

creates an array called “a” which consists of 5 **long ints**. It is a rule of C that the storage for an array is physically contiguous in memory. Thus wherever, say, the second element sits in memory, the third element will be adjacent to it, the fourth next to that and so on.

## Examples

```
#define SIZE 10
int a[5]; /* a is an array of 5 ints */
long int big[100]; /* big is 400 bytes! */
double d[100]; /* but d is 800 bytes! */
long double v[SIZE]; /* 10 long doubles, 100 bytes */
```

```
int a[5] = { 10, 20, 30, 40, 50 };
double d[100] = { 1.5, 2.7 };
short primes[] = { 1, 2, 3, 5, 7, 11, 13 };
long n[50] = { 0 };
```

all five  
elements  
initialised

first two elements  
initialised,  
remaining ones  
set to zero

compiler fixes  
size at 7  
elements

```
int i = 7;
const int c = 5;

int a[i];
double d[c];
short primes[];
```

quickest way of setting  
ALL elements to zero

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 3

## Examples

Above are examples of declaring and initializing arrays. Notice that C can support arrays of any type, including structures (which will be covered the next chapter), except **void** (which isn't a type so much as the absence of a type). You will notice that a *constant* must appear within the brackets so:

```
long int a[10];
```

is fine, as is:

```
#define SIZE 10
long int a[SIZE];
```

But:

```
int size = 10;
long int a[size];
```

and

```
const int a_size = 10;
long int a[a_size];
```

will NOT compile. The last is rather curious since "a\_size" is obviously constant, however, the compiler will not accept it. Another thing to point out is that the number provided must be an integral type, "int a[5.3]" is obviously nonsense.

### Initializing Arrays

1. The number of initializing values is exactly the same as the number of elements in the array. In this case the values are assigned one to one, e.g. `int a[5] = { 1, 2, 3, 4, 5 };`
2. The number of initializing values is less than the number of elements in the array. Here the values are assigned "one to one" until they run out. The remaining array elements are initialized to zero, e.g. `int a[5] = { 1, 2 };`
3. The number of elements in the array has not been specified, but a number of initializing values has. Here the compiler fixes the size of the array to the number of initializing values and they are assigned one to one, e.g. `int a[] = { 1, 2, 3, 4, 5 };`

## Accessing Elements

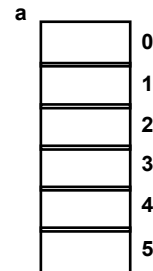
- The elements are accessed via an integer which ranges from 0..size-1
- There is no bounds checking

```
int main(void)
{
    int    a[6];
    int    i = 7;

    a[0] = 59;
    a[5] = -10;
    a[i/2] = 2;

    a[6] = 0;
    a[-1] = 5;

    return 0;
}
```



© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 4

## Accessing Elements

### Numbering Starts at Zero

THE most important thing to remember about arrays in C is the scheme by which the elements are numbered. The FIRST element in the array is element number ZERO, the second element is number one and so on. The LAST element in the array "a" above is element number FIVE, i.e. the total number of elements less one.

This scheme, together with the fact that there is no bounds checking in C accounts for a great deal of errors where array bounds accessing is concerned. It is all too easy to write "**a[6] = 0**" and index one beyond the end of the array. In this case whatever variable were located in the piece of memory (maybe the variable "i") would be corrupted.

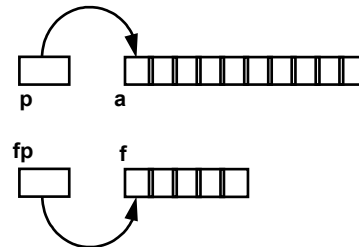
Notice that the array access **a[i/2]** is fine, since "i" is an integer and thus **i/2** causes integer division.

## Array Names

- There is a special and unusual property of array names in C
- The name of an array is a pointer to the start of the array, i.e. the zeroth element, thus

$a == \&a[0]$

```
int    a[10];  
int    *p;  
  
float  f[5];  
float  *fp;  
  
p = a;    /* p = &a[0] */  
fp = f;   /* fp = &f[0] */
```



© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 5

## Array Names

### A Pointer to the Start

In C, array names have a rather unusual property. The compiler treats the name of an array as an address which may be used to initialize a pointer without error. The address is that of the first element (i.e. the element with index 0).

### Cannot Assign to an Array


Note that the address is a *constant*. If you are wondering what would happen with the following:

```
int    a[10];  
int    b[10];  
  
a = b;
```

the answer is that you'd get a compiler error. The address that "a" yields is a constant and thus it cannot be assigned to. This makes sense. If it were possible to assign to the name of an array, the compiler might "forget" the address at which the array lived in memory.

## Passing Arrays to Functions

- When an array is passed to a function a pointer to the zeroth element is passed across
- The function may alter any element
- The corresponding parameter may be declared as a pointer, or by using the following special syntax



```
int add_elements(int a[], int size)
{
```

```
int add_elements(int *p, int size)
{
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 6

## Passing Arrays to Functions

If we declare an array:

```
int a[60];
```

and then pass this array to a function:

```
function(a);
```

the compiler treats the name of the array "a" in exactly the same way it did before, i.e. as a pointer to the zeroth element of the array. This means that a pointer is passed to the function, i.e. the array is NOT passed by value.

### Bounds Checking Within Functions

One problem with this strategy is that there is no way for the function to know how many elements are in the array (all the function gets is a pointer to one integer, this could be one lone integer or there could be one hundred other integers immediately after it). This accounts for the second parameter in the two versions of the `add_elements` function above. This parameter must be provided by us as the valid number of elements in the array.

Note that there is some special syntax which makes the parameter a pointer. This is:

```
int a[]
```

This is one of very few places this syntax may be used. Try to use it to declare an array and the compiler will complain because it cannot determine how much storage to allocate for the array. All it is doing here is the same as:

```
int * a;
```

Since pointers are being used here and we can write down pointers, any element of the array may be changed.

## Example

```
#include <stdio.h>

void sum(long [], int);

int main(void)
{
    long primes[6] = { 1, 2,
                      3, 5, 7, 11 };

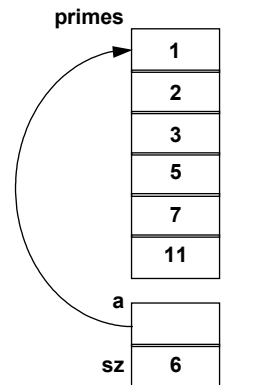
    sum(primes, 6);

    printf("%li\n", primes[0]);

    return 0;
}

void sum(long a[], int sz)
{
    int i;
    long total = 0;
    for(i = 0; i < sz; i++)
        total += a[i];

    a[0] = total;
}
```



provides bounds checking

the total is written over  
element zero

© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 7

## Example

### A Pointer is Passed

In the example above the array "primes" is passed down to the function "sum" by way of a pointer. "a" is initialized to point to primes[0], which contains the value 1.

Within the function the array access a[i] is quite valid. When "i" is zero, a[0] gives access to the value 1. When "i" is one, a[1] gives access to the value 2 and so on. Think of "i" as an offset of the number of **long ints** beyond where "a" points.

### Bounds Checking

The second parameter, "sz" is 6 and provides bounds checking. You will see the **for** loop:

```
for(i = 0; i < sz; i++)
```

is ideally suited for accessing the array elements. a[0] gives access to the first element, containing 1. The last element to be accessed will be a[5] (because "i" being equal to 6 causes the loop to exit) which contains the 11.

Notice that because call by reference is used, the sum function is able to alter any element of the array. In this example, element a[0], in other words prime[0] is altered to contain the sum.

## Using Pointers - Example

```
#include <stdio.h>

long sum(long*, int);

int main(void)
{
    long primes[6] = { 1, 2,
                      3, 5, 7, 11 };

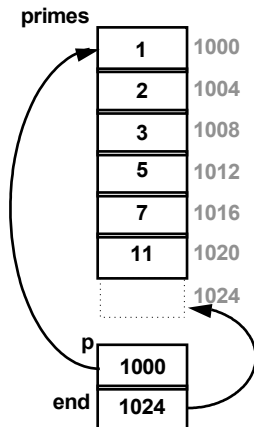
    printf("%li\n", sum(primes, 6));

    return 0;
}

long sum(long *p, int sz)
{
    long *end = p + sz;
    long total = 0;

    while(p < end)
        total += *p++;

    return total;
}
```



© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 11

## Using Pointers - Example

Above is an example of using pointers to handle an array. In the statement:

```
sum(primes, 6)
```

the use of the name of the array "primes" causes the address of the zeroth element, 1000, to be copied into "p". The 6 is copied into "sz" and provides bounds checking.

The initialization: 

```
long *end = p + sz;
```

sets the pointer "end" to be 1000 + 6 \* 4 (since **long int** is 4 bytes in size), i.e. 1024. The location with address 1024 lies one beyond the end of the array, hence

```
while(p < end)
```

and NOT:

```
while(p <= end)
```

The statement:

```
total += *p++;
```

adds into "total" (initially zero) the value at the end of the pointer "p", i.e. 1. The pointer is then incremented, 4 is added, "p" becoming 1004. Since 1004 is less than the 1024 stored in "end", the loop continues and the value at location 1004, i.e. 2 is added in to total. The pointer increases to 1008, still less than 1024. It is only when all the values in the array have been added, i.e. 1, 2, 3, 5, 7 and 11 that the pointer "p" points one beyond the 11 to the location whose address is 1024. Since the pointer "p" now contains 1024 and the pointer "end" contains 1024 the condition:

```
while(p < end)
```

is no longer true and the loop terminates. The value stored in total, 34, is returned.



## Strings

- C has no native string type, instead we use arrays of `char`
- A special character, called a “null”, marks the end (don’t confuse this with the `NULL` pointer)
- This may be written as `'\0'` (zero not capital ‘o’)
- This is the only character whose ASCII value is zero
- Depending on how arrays of characters are built, we may need to add the null by hand, or the compiler may add it for us

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 14

---

## Strings

The sudden topic change may seem a little strange until you realize that C doesn't really support strings. In C, strings are just arrays of characters, hence the discussion here.

C has a special marker to denote the last character in a string. This character is called the null and is written as `'\0'`. You should not confuse this null with the `NULL` pointer seen in the pointers chapter. The difference is that `NULL` is an invalid pointer value and may be defined in some strange and exotic way. The null character is entirely different as it is always, and is guaranteed to be, zero.

Why the strange way of writing `'\0'` rather than just `0`? This is because the compiler assigns the type of `int` to `0`, whereas it assigns the type `char` to `'\0'`. The difference between the types is the number of bits, `int` gives 16 or 32 bits worth of zero, `char` gives 8 bits worth of zero. Thus, potentially, the compiler might see a problem with:

```
char c = 0;
```

since there are 16 or 32 bits of zero on the right of “=”, but room for only 8 of those bits in “c”.

## Example

```
char first_name[5] = { 'J', 'o', 'h', 'n', '\0' };
char last_name[6] = "Minor";
char other[] = "Tony Blurt";
char characters[7] = "No null";
```

this special case specifically  
excludes the null terminator

first_name	'J'	'o'	'h'	'n'	0							
last_name	'M'	'i'	'n'	'o'	'r'	0						
other	'T'	'o'	'n'	'y'	32	'B'	'l'	'u'	'r'	't'	0	
characters	'N'	'o'	32	'n'	'u'	'l'	'l'					

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 15

## Example

The example above shows the two ways of constructing strings in C. The first requires the string to be assembled by hand as in:

```
char first_name[5] = { 'J', 'o', 'h', 'n', '\0' };
```

### Character Arrays vs. Strings

Each character value occupies a successive position in the array. Here the compiler is not smart enough to figure we are constructing a string and so we must add the null character `'\0'` by hand. If we had forgotten, the array of characters would have been just that, an array of characters, not a string.

### Null Added Automatically

The second method is much more convenient and is shown by:

```
char last_name[6] = "Minor";
```

Here too the characters occupy successive locations in the array. The compiler realizes we are constructing a string and *automatically adds the null terminator*, thus 6 slots in the array and NOT 5.

As already seen, when providing an initial value with an array, the size may be omitted, as in:

```
char other[] = "Tony Blurt";
```

Here, the size deduced by the compiler is 11 which includes space for the null terminator.

### Excluding Null

A special case exists in C when the size is set to exactly the number of characters used in the initialization *not including the null character*. As in:

```
char characters[7] = "No null";
```

Here the compiler deliberately excludes the null terminator. Here is an array of characters and not a string.

## Printing Strings

- Strings may be printed by hand
- Alternatively printf supports “%s”

```
char other[] = "Tony Blurt";
```

```
char *p;  
p = other;  
while(*p != '\0')  
    printf("%c", *p++);  
printf("\n");
```

```
int i = 0;  
while(other[i] != '\0')  
    printf("%c", other[i++]);  
printf("\n");
```

```
printf("%s\n", other);
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 16

## Printing Strings

### printf “%s” Format Specifier

Strings may be printed by hand, character by character until the null is found or by using the “%s” format specifier to **printf**. **scanf** understands this format specifier too and will read a sequence of characters from the keyboard.

Consider the way: **printf("%s\n", other);**

actually works. Being an array, “other” generates the address of the first character in the array. If this address were, say, 2010 the “%s” format specifier tells **printf** to print the character stored at location 2010. This is the character “T”.

**printf** then increments its pointer to become 2011 (because **char** is being dealt with, there is no scaling of the pointer). The value at this location “o” is tested to see if it null. Since it is not, this value is printed too. Again the pointer is incremented and becomes 2012. The character in this location “n” is tested to see if it is null, since it is not, it is printed.

This carries on right through the “y”, space, “B”, “l”, “u”, “r” and “t”. With “t” the pointer is 2019. Since the “t” is not null, it is printed, the pointer is incremented. Now its value is 2020 and the value “\0” stored at that location is tested. **printf** breaks out of its loop and returns to the caller.

Consider also the chaos that would result if the array “characters” defined previously were thrown at **printf** and the “%s” format specifier. This array of characters did not contain the null terminator and, since there is no bounds checking in C, **printf** would continue printing characters randomly from memory until by the laws of chance found a byte containing zero. This is a very popular error in C programs.

## Null Really Does Mark the End!

```
#include <stdio.h>

int main(void)
{
    char other[] = "Tony Blurt";

    printf("%s\n", other);

    other[4] = '\0';

    printf("%s\n", other);

    return 0;
}
```

even though the rest of  
the data is still there,  
printf will NOT move  
past the null terminator

Tony Blurt  
Tony

other

'T'	'o'	'n'	'y'	32	'B'	'l'	'u'	'r'	't'	0
-----	-----	-----	-----	----	-----	-----	-----	-----	-----	---

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 17

## Null Really Does Mark the End!

The example here shows how **printf** will not move past the null terminator. In the first case, 11 characters are output (including the space).

When the null terminator is written into the fifth position in the array only the four characters before it are printed. Those other characters are still there, but simply not printed.

## Assigning to Strings

- Strings may be initialised with "=", but not assigned to with "="
- Remember the name of an array is a **CONSTANT** pointer to the zeroth element

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char who[] = "Tony Blurt";
    who = "John Minor";
    strcpy(who, "John Minor");
    return 0;
}
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 18

## Assigning to Strings

Don't make the mistake of trying to assign values into strings at run time as in:

```
who = "John Minor";
```

By trying to assign to "who" the compiler would attempt to assign to the address at which the "T" is stored (since "who" is the name of an array and therefore the address of the zeroth element). This address is a constant. Instead the Standard Library function **strcpy** should be used as in:

```
strcpy(who, "John Minor");
```

notice how the format is: **strcpy(destination, source);**

this routine contains a loop (similar to that contained in **printf**) which walks the string checking for the null terminator. While it hasn't been found it continues copying into the target array. It ensures the null is copied too, thus making "who" a valid string rather than just an array of characters. Notice also how **strcpy** does absolutely no bounds checking, so:

```
strcpy(who, "a really very long string indeed");
```

would overflow the array "who" and corrupt the memory around it. This would very likely cause the program to crash. A safer option entirely is to use **strncpy**'s cousin **strncpy** which is count driven:

```
strncpy(who, "a really very long string indeed",
        sizeof(who));
```

This copies either up to the null terminator, or up to the count provided (here 11, the number of bytes yielded by **sizeof**). Unfortunately when **strncpy** hits the count first, it fails to null terminate. We have to do this by hand as in:

```
who[sizeof(who) - 1] = '\0';
```

## Multidimensional Arrays

- C does not support multidimensional arrays
- However, C does support arrays of any type including arrays of arrays

```
float rainfall[12][365];
```

"rainfall" is an array of 12  
arrays of 365 float

```
short exam_marks[500][10];
```

"exam\_marks" is an array of  
500 arrays of 10 short int

```
const int brighton = 7;  
int day_of_year = 238;  
  
rainfall[brighton][day_of_year] = 0.0F;
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 21

## Multidimensional Arrays

Sometimes a simple array just isn't enough. Say a program needed to store the rainfall for 12 places for each of the 365 days in the year (pretend it isn't a leap year). 12 arrays of 365 reals would be needed. This is exactly what:

```
float rainfall[12][365];
```

gives. Alternatively imagine a (big) college with up to 500 students completing exams. Each student may sit up to 10 exams. This would call for 500 arrays of 10 integers (we're not interested in fractions of a percent, so whole numbers will do). This is what:

```
short exam_marks[500][10];
```

gives. Although it may be tempting to regard these variables as multi dimensional arrays, C doesn't treat them as such. Firstly, to access the 5th location's rainfall on the 108th day of the year we would write:

```
printf("rainfall was %f\n", rainfall[5][108]);
```

and NOT (as in some languages):

```
printf("rainfall was %f\n", rainfall[5, 108]);
```

which wouldn't compile. In fact, C expects these variables to be initialized as arrays of arrays. Consider:

```
int    rainfall_in_mm_per_month[4][12] = {  
    { 17, 15, 20, 25, 30, 35, 48, 37, 28, 19, 18, 10 },  
    { 13, 13, 18, 20, 27, 29, 29, 26, 20, 15, 11, 8 },  
    { 7, 9, 11, 11, 12, 14, 16, 13, 11, 8, 6, 3 },  
    { 29, 35, 40, 44, 47, 51, 59, 57, 42, 39, 35, 28 },  
};
```

where each of the four arrays of twelve floats are initialized separately.

## Review

- How many times does the following program loop?

```
#include <stdio.h>

int main(void)
{
    int i;
    int a[10];

    for(i = 0; i <= 10; i++) {
        printf("%d\n", i);
        a[i] = 0;
    }

    return 0;
}
```

---

## Review

Time for a break. How many times will the loop execute?

## Summary

- **Arrays are declared with a type, a name, “[ ]” and a CONSTANT**
- **Access to elements by array name, “[ ]” and an integer**
- **Arrays passed into functions by pointer**
- **Pointer arithmetic**
- **Strings - arrays of characters with a null terminator**
- **Sometimes compiler stores null for us (when double quotes are used) otherwise we have to store it ourselves**

---

## Summary