

1 Arrays

Sometimes, you want to define something that will hold more than one of a given type. For example, if we wanted to store a list of integers, or a mathematical vector whose components are doubles, we would need such a thing. Those are called *arrays* in C.

1.1 Declaring and Using Arrays

Here is the general form of an array declaration:

```
<type> <variable-name>[<size-of-array>];
```

Here the square brackets must be literally present, they do not denote “optional”. Here are a few examples of array declarations:

```
int a[10];
float b[3];
double c[10];
```

For example, the declaration `int a[10];` means that `a` is an array of 10 integers. In other words, `a` can be used to store 10 different integers.

The question is, how do we refer to the individual elements of an array? The answer is you refer to the element by its index, as in:

```
a[0] = 5;
a[2] = 10/3;
```

So, in order to refer to an element of an array, you follow the array name by the *index* of the element in square brackets. The index runs from zero to the size of the array minus one. So, for example, if the array `a` is declared as `int a[10]`, you can refer to `a[0]` through `a[9]`.

What happens if you refer to an index that is larger than the size of the array (or that is negative)? First, C will definitely not stop you from doing so at compile time. Second, probably very bad things will happen at runtime. So, you should really know what you are doing when you use arrays.

1.2 Initializing Arrays

Just like variables, arrays will also contain random values when they are declared. It is possible to give arrays initial values in declaration as in the following example:

```
int a[3] = {2, 5, 3};
```

This declaration means that `a[0]` will have a value of 2, `a[1]` will have a value of 5, and `a[2]` will have a value of 3. This general form is the same for other types of arrays, except that the values must be of the correct type.

Note that the number of elements in the initializer matches the declared size of the array. Giving too many initializers is an error. But, if you have an initializer with too few elements, the remaining elements are automatically assigned to be zero.

In case you declare an array with an initializer, you do not have to specify the length; the length will just be the number of elements in the initializer. So, the following declaration is a valid one:

```
int a[] = {2, 5, 3};
```

1.3 Multi-Dimensional Arrays

The arrays you have seen so far are known as “one-dimensional arrays” because they have only one index. Multi-dimensional arrays are actually arrays of arrays, in other words arrays whose elements are arrays, which can have further arrays as elements. So, here is an example of a declaration of a two dimensional array:

```
float x[10][20];
```

This declares a two-dimensional array which has a total of 200 elements. For example, `x[2]` is an array itself which has 20 elements. So, `x` is an array of 10 elements, and those elements are arrays of 20 elements themselves.

How do you initialize such an array? Here is an example (with fewer elements):

```
int a[3][2] = {{1, 2}, {5, 7}, {6, 9}};
```

So it is three arrays of two elements each. Can you omit the size in the declaration in this case? Yes, but you can only omit the very first index. So, the following is possible:

```
int a[][2] = {{1, 2}, {5, 7}, {6, 9}};
```

If you think that is just too many braces to type, you can omit the inner braces as follows:

```
int a[][2] = {1, 2, 5, 7, 6, 9};
```

But, I think the initializer is more clear with the braces left in. It is a matter of convenience, and normally you will not have too many multi dimensional array with initializers in a program. Therefore, this should not be a problem.

2 Strings in C

The C language does not have a “string” type. However, you can clearly have string constants in a program. You have already seen such constants, in expressions such as:

```
printf("Please enter a number:");
```

So, any number of characters enclosed in double quotes is a string constant in C. However, how do we define string variables if we do not have a string data type? Strings in C are null-terminated arrays of `chars`. What does null terminated mean? There is no way C can know the length of any given array, so any string (character array) must be terminated with a character whose value is zero (null). So, if you want to be able to store strings that are 20 characters long, you should declare a character array of length 21. So, you will be able to store 20-character strings in the following array:

```
char str[21];
```

How do you initialize such an array? There are a few ways. one of them is initializing it using numbers as in the last section. Another is doing the following:

```
char str[3] = {'a', 'b', 0};
```

This makes `str` a null terminated string containing “ab”. However, you will almost never see such a thing in any program. What you *will* see is the following:

```
char str[3] = "ab";
```

This has the same effect as the previous declaration. But, it looks more readable. Note again that we have defined an array of length 3, therefore we can store a string of 2 characters in length in it.

You can also leave the size unspecified, in which case the array will be just large enough to store the string:

```
char str[] = "ES 112 : Algorithms and Computer Programming";
```

It is possible to define a larger array to store a shorter string in it:

```
char str[256] = "Hello, world";
```

What can we do with such strings? Well, we can use them for a variety of purposes. We can print them. We can get user input into them. And, we can perform operations on them. Unfortunately, none of the operators work on strings in C. For that, we need to use the C library functions.

3 String Library Functions

The functions that can be used to operate on strings are defined in the standard C library header file `string.h`. So, if you use any of these functions in your program, you need to `#include <string.h>` at the top of your file.

3.1 String Copying

NAME

`strcpy`, `strncpy` - copy a string

SYNOPSIS

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

```
char *strcpy(char *dest, const char *src);
```

```
char *strncpy(char *dest, const char *src, size_t n);
```

DESCRIPTION

The `strcpy()` function copies the string pointed to by `src` (including the terminating `'\0'` character) to the array pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy.

The `strncpy()` function is similar, except that not more than `n` bytes of `src` are copied. Thus, if there is no null byte among the first `n` bytes of `src`, the result will not be null-terminated.

In the case where the length of `src` is less than that of `n`, the remainder of `dest` will be padded with nulls.

RETURN VALUE

The `strcpy()` and `strncpy()` functions return a pointer to the destination string `dest`.

BUGS

If the destination string of a `strcpy()` is not large enough (that is, if the programmer was stupid/lazy, and failed to check the size before copying) then anything might happen. Overflowing fixed length strings is a favourite cracker technique.

3.2 String Concatenation

NAME

`strcat`, `strncat` - concatenate two strings

SYNOPSIS

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

```
char *strcat(char *dest, const char *src);
```

```
char *strncat(char *dest, const char *src, size_t n);
```

DESCRIPTION

The `strcat()` function appends the `src` string to the `dest` string overwriting the `'\0'` character at the end of `dest`, and then adds a terminating `'\0'` character. The strings may not overlap, and the `dest` string must have enough space for the result.

The `strncat()` function is similar, except that only the first `n` characters of `src` are appended to `dest`.

RETURN VALUE

The `strcat()` and `strncat()` functions return a pointer to the resulting string `dest`.

3.3 Comparing Strings

NAME

`strcmp`, `strncmp` - compare two strings

SYNOPSIS

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

```
int strcmp(const char *s1, const char *s2);
```

```
int strncmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

The `strcmp()` function compares the two strings `s1` and `s2`. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.

The `strncmp()` function is similar, except it only compares the first `n` characters of `s1`.

RETURN VALUE

The `strcmp()` and `strncmp()` functions return an integer less than, equal to, or greater than zero if `s1` (or the first `n` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`.

3.4 Finding the Length of a String

NAME

`strlen` - calculate the length of a string

SYNOPSIS

```
#include <string.h>

size_t strlen(const char *s);
```

DESCRIPTION

The `strlen()` function calculates the length of the string `s`, not including the terminating `'\0'` character.

RETURN VALUE

The `strlen()` function returns the number of characters in `s`.

3.5 Reading Strings from the Keyboard

NAME

`gets`

SYNOPSIS

```
#include <stdio.h>
char *gets(char *s);
```

DESCRIPTION

`gets()` reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or EOF, which it replaces with `'\0'`. No check for buffer overrun is performed (see BUGS below).

RETURN VALUE

`gets()` returns `s` on success, and `NULL` on error or when end of file occurs while no characters have been read.

BUGS

Never use `gets()`. Because it is impossible to tell without knowing the data in advance how many characters `gets()` will read, and because `gets()` will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security.

4 Examples

4.1 Example 1

```
#include <stdio.h>

int main(void)
{
    char fname[80];
    char lname[80];

    printf("Enter your first name: ");
    gets(fname);
    printf("Enter your last name: ");
    gets(lname);
    printf("Your name is: %s %s\n", fname, lname);

    return 0;
}
```

4.2 Example 2

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

int main(void)
{
    char password1[80];
    char password2[80];
    int done = 0;

    while (!done) {
        printf("Please enter a password: ");
        gets(password1);

        printf("Enter the password again: ");
        gets(password2);

        if (strcmp(password1, password2)) {
            printf("The passwords do not match. Try again\n");
        } else {
            done = 1;
        }
    }
    return 0;
}
```