Arrays and Strings

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Arrays. . .

An array is a group of variables of a particular type occupying a contiguous region of memory.

Arrays. . .

An array is a group of variables of a particular type occupying a contiguous region of memory.

In *C*, array elements are numbered from `0`, so that an array of size `N` is indexed from `0` to `N−1`.

Arrays. . .

An array is a group of variables of a particular type occupying a contiguous region of memory.

In *C*, array elements are numbered from `0`, so that an array of size `N` is indexed from `0` to `N−1`.

An array must contain at least one element, and it is an error to define an empty array.

Arrays. . .

An array is a group of variables of a particular type occupying a contiguous region of memory.

In *C*, array elements are numbered from `0`, so that an array of size `N` is indexed from `0` to `N−1`.

An array must contain at least one element, and it is an error to define an empty array.

A variable of type array is declared by adding square brackets (`[`, `]`) after it:

```
int array[]
```

Arrays. . .

An array is a group of variables of a particular type occupying a contiguous region of memory.

In *C*, array elements are numbered from 0, so that an array of size N is indexed from 0 to N−1.

An array must contain at least one element, and it is an error to define an empty array.

A variable of type array is declared by adding square brackets ([,]) after it:

```
int array[]
```

The number of elements in the array can be declared:

```
double array[SIZE]
```

Array initialisation. . .

Remember: As for any other type of variable, arrays may have local, external or static scope!

Array initialisation...

Remember: As for any other type of variable, arrays may have local, external or static scope!

Arrays with static extent have their elements initialised to zero by default

Array initialisation. . .

Remember: As for any other type of variable, arrays may have local, external or static scope!

Arrays with static extent have their elements initialised to zero by default, but arrays with local extent are not initialised by default, so their elements have arbitrary values.

Array initialisation...

Remember: As for any other type of variable, arrays may have local, external or static scope!

Arrays with static extent have their elements initialised to zero by default, but arrays with local extent are not initialised by default, so their elements have arbitrary values.

An array is explicitly initialised by using an initialiser list (a list of values of the appropriate type enclosed in braces and separated by commas):

```
int days[12] =
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Array initialisation...

If the number of values in the initialiser list is less than the size of the array, the remaining elements of the array are initialised to zero:

```
int allZero[10] = { 0 };
```

Array initialisation. . .

If the number of values in the initialiser list is less than the size of the array, the remaining elements of the array are initialised to zero:

```
int allZero[10] = { 0 };
```

It is an error to have more initialisers than the size of the array!

Array initialisation...

If the number of values in the initialiser list is less than the size of the array, the remaining elements of the array are initialised to zero:

```
int allZero[10] = { 0 };
```

It is an error to have more initialisers than the size of the array!

If the size of an array with an initialiser list is not specified, the array will automatically be allocated memory to match the number of elements in the list:

```
int days[] = //The size of this array is 12!
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Size of an array. . .

Function sizeof can be used to determine the size of the array.

Size of an array. . .

Function `sizeof` can be used to determine the size of the array.

Warning: the result of `sizeof (e)` may be confusing:

- if `e` is a value or a type, it indicates the amount of memory needed to store that value (or type);

Size of an array...

Function `sizeof` can be used to determine the size of the array.

Warning: the result of `sizeof (e)` may be confusing:

- if `e` is a value or a type, it indicates the amount of memory needed to store that value (or type);
- if `e` is a variable, it indicates the amount of memory allocated to store that variable.

Size of an array. . .

Function `sizeof` can be used to determine the size of the array.

Warning: the result of `sizeof(e)` may be confusing:

- if `e` is a value or a type, it indicates the amount of memory needed to store that value (or type);
- if `e` is a variable, it indicates the amount of memory allocated to store that variable.

Size of an array. . .

Function `sizeof` can be used to determine the size of the array.

Warning: the result of `sizeof(e)` may be confusing:

- if `e` is a value or a type, it indicates the amount of memory needed to store that value (or type);
- if `e` is a variable, it indicates the amount of memory allocated to store that variable.

The size of an array is the amount of memory allocated for all the elements in the array!

Example. . .

```c
int x = 10;
int *px = &x;
int array[] = { 1, 2 , 3 , 4 , 5 };

printf("Val: %lu\n",sizeof(10));
printf("Var: %lu\n", sizeof(x));
printf("Pointer: %lu\n", sizeof(px));
printf("Array: %lu\n", sizeof(array));
```

Example...

```
int x = 10;
int *px = &x;
int array[] = { 1, 2 , 3 , 4 , 5 };

printf("Val: %lu\n",sizeof(10));
printf("Var: %lu\n", sizeof(x));
printf("Pointer: %lu\n", sizeof(px));
printf("Array: %lu\n", sizeof(array));
```

Result:

```
Val: 4
Var: 4
Pointer: 8
Array: 20
```

Number of elements in an array...

The general pattern used to get the number of elements in an array `a` is:

```c
int length = sizeof(a)/sizeof(a[0]);
```

Number of elements in an array...

The general pattern used to get the number of elements in an array `a` is:

```
int length = sizeof(a)/sizeof(a[0]);
```

Warning: when an array is passed to a function it is automatically converted to a pointer!

Number of elements in an array. . .

The general pattern used to get the number of elements in an array `a` is:

```c
int length = sizeof(a)/sizeof(a[0]);
```

Warning: when an array is passed to a function it is automatically converted to a pointer!

```c
int count_days(int days[], int len)
    {
        int total=0;
        /* assert will fail: sizeof(days)
           equals sizeof(int *) and len equals 12 */
        assert(sizeof(days) / sizeof(days[0]) == len);
        while(len--)
            total += days[len];
        return total;
}
```

Character Arrays and Strings

Character arrays are special. They have certain initialisation properties not shared with other array types because of their relationship with strings.

Character Arrays and Strings

Character arrays are special. They have certain initialisation properties not shared with other array types because of their relationship with strings.

Character arrays can be initialised in the normal way using an initialiser list.

```
char letters = { 'a' , 'b' , 'c' , 'd' , 'e' };
```

Character Arrays and Strings

Character arrays are special. They have certain initialisation properties not shared with other array types because of their relationship with strings.

Character arrays can be initialised in the normal way using an initialiser list.

```
char letters = { 'a' , 'b' , 'c' , 'd' , 'e' };
```

But they may also be initialised using a string constant, as follows.

```
char letters[] = "abcde";
```

Character Arrays and Strings

Character arrays are special. They have certain initialisation properties not shared with other array types because of their relationship with strings.

Character arrays can be initialised in the normal way using an initialiser list.

```
char letters = { 'a' , 'b' , 'c' , 'd' , 'e' };
```

But they may also be initialised using a string constant, as follows.

```
char letters[] = "abcde";
```

The string initialisation automatically appends a `\0` character, so the above array is of size 6, not 5. It is equivalent to writing,

```
char letters = { 'a' , 'b' , 'c' , 'd' , 'e' , '\0' };
```

Character Arrays and Strings

An important property of string constants is that they are allocated memory.

Character Arrays and Strings

An important property of string constants is that they are allocated memory.

This means that they have an address and may be referred to by a `char*` pointer.

Character Arrays and Strings

An important property of string constants is that they are allocated memory.

This means that they have an address and may be referred to by a `char*` pointer.

Warning: For constants of any other type, it is not possible to assign a pointer because these constants are not stored in memory and do not have an address.

Character Arrays and Strings

An important property of string constants is that they are allocated memory.

This means that they have an address and may be referred to by a char* pointer.

Warning: For constants of any other type, it is not possible to assign a pointer because these constants are not stored in memory and do not have an address.

```
double *pval = 9.6;        /* Invalid. Won't compile. */
int *parray = { 1, 2, 3 }; /* Invalid. Won't compile. */
char *str = "Hello World!\n"; /* Correct. Read-only array. */
```

Strings and the Standard Library

The standard library contains many functions for manipulating strings:

`size_t strlen(const char *s)`. Returns the number of characters in string s, excluding the terminating \0 character.

`char *strcpy(char *s, const char *t)`. Copies the string t into character array s, and returns a pointer to s.

`int strcmp(const char *s, const char *t)`. Performs a lexicographical comparison of strings s and t, and returns a negative value if $s < t$, a positive value if $s > t$, and zero if $s == t$.

Strings and the Standard Library

char *strcat(char *s, const char *t). Concatenates the string t onto the end of string s. The first character of t overwrites the '\0' character at the end of s.

char *strchr(const char *s, int c). Returns a pointer to the first occurrence of character c in string s. If c is not present, then NULL is returned.

char *strrchr(const char *s, int c). Performs the same task as strchr() but starting from the reverse end of s.

char *strstr(const char *s, const char *t). Searches for the first occurrence of sub-string t in string s. If found, it returns a pointer to the beginning of the substring in s, otherwise it returns NULL.

Arrays of Pointers

Since pointers are themselves variables, they can be stored in arrays just as other variables can:

Arrays of Pointers

Since pointers are themselves variables, they can be stored in arrays just as other variables can:

```
double *parray[N];
```

Arrays of Pointers

Since pointers are themselves variables, they can be stored in arrays just as other variables can:

```
double *parray[N];
```

Each pointer in an array of pointers behaves as any ordinary pointer would:

```
double val = 9.7;
double array[] = { 3.2, 4.3, 5.4 };
double *pa[] = { &val, array+1, NULL };
```

Arrays of Pointers

Since pointers are themselves variables, they can be stored in arrays just as other variables can:

```
double *parray[N];
```

Each pointer in an array of pointers behaves as any ordinary pointer would:

```
double val = 9.7;
double array[] = { 3.2, 4.3, 5.4 };
double *pa[] = { &val, array+1, NULL };
```

In the above example, element `pa[i]` is a pointer to a double, and `*pa[i]` is the double variable that it points to.

Arrays of Pointers

If an element in an array of pointers also points to an array, the elements of the pointed-to array may be accessed in a variety of different ways:

```c
int a1[] = { 1, 2, 3, 4 };
int a2[] = { 5, 6, 7 };
/* pa stores pointers to beginning of each array. */
int *pa[] = { a1, a2 };
/* Pointer-to-a-pointer holds address of beginning of pa. */
int **pp = pa;
int*p= pa[1]; /* Pointer to the second array in pa. */
int val;
val = pa[1][1]; /* equivalent operations: val = 6 */
val = pp[1][1];
val = *(pa[1] + 1);
val = *(pp[1] + 1);
val = *(*(pp+1) + 1));
val = p[1];
```

Multi-dimensional Arrays

A multi-dimensional array is defined using multiple adjacent square brackets, and the elements of the array may be initialised with values enclosed in curly braces:

Multi-dimensional Arrays

A multi-dimensional array is defined using multiple adjacent square brackets, and the elements of the array may be initialised with values enclosed in curly braces:

```
float matrix[3][4] = {
        { 2.4, 8.7, 9.5, 2.3 },
        { 6.2, 4.8, 5.1, 8.9 },
        { 7.2, 1.6, 4.4, 3.6 }
};
```

Multi-dimensional Arrays

A multi-dimensional array is defined using multiple adjacent square brackets, and the elements of the array may be initialised with values enclosed in curly braces:

```
float matrix[3][4] = {
        { 2.4, 8.7, 9.5, 2.3 },
        { 6.2, 4.8, 5.1, 8.9 },
        { 7.2, 1.6, 4.4, 3.6 }
};
```

As for one dimensional arrays, multi-dimensional arrays may be defined without a specific size. However, only the left-most subscript is free:

Multi-dimensional Arrays

A multi-dimensional array is defined using multiple adjacent square brackets, and the elements of the array may be initialised with values enclosed in curly braces:

```
float matrix[3][4] = {
        { 2.4, 8.7, 9.5, 2.3 },
        { 6.2, 4.8, 5.1, 8.9 },
        { 7.2, 1.6, 4.4, 3.6 }
};
```

As for one dimensional arrays, multi-dimensional arrays may be defined without a specific size. However, only the left-most subscript is free:

```
float matrix[][4] = {        /* The 4 must be specified. */
        { 2.4, 8.7, 9.5, 2.3 },
        { 6.2, 4.8, 5.1, 8.9 }
};
```

To be continued. . .