Programming I

Arrays

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Arrays

Sometimes we want to hold in memory many objects of the same type.

e.g.

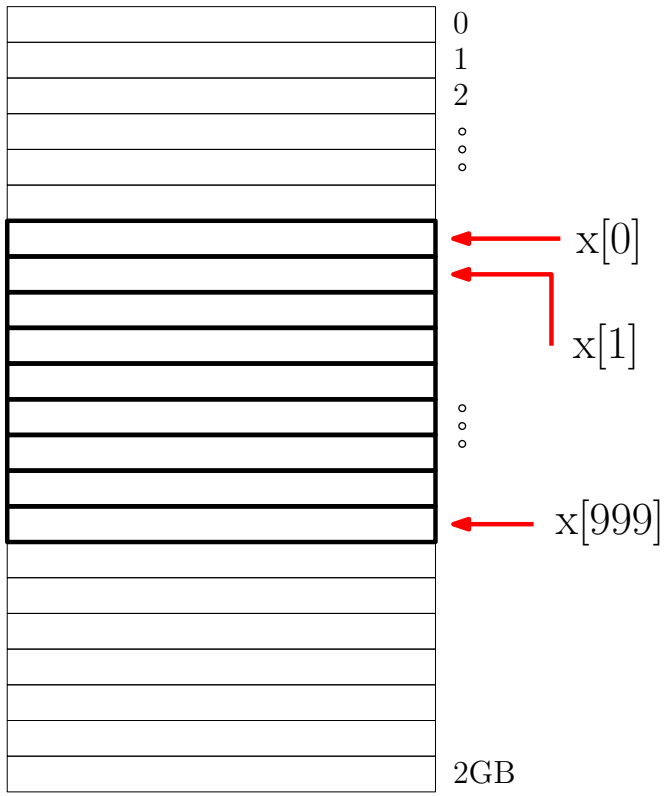
- maybe we want to read 1000 integers and sort them in non-decreasing order
- maybe we want to store 100 student names along with their grades

Arrays

An array is a group of continous memory locations which have the same name and same type.

Array Definition in C

```
int main() {
    int x[1000];
}
```



Basic Array Characteristics in C

```
int main() {
    int x[1000];
}
```

1. The first element is always element zero (0)
2. Elements are always continous in memory
3. The size of an array must be integral

Accessing an Array Element

In order to read or write an array element we must provide:

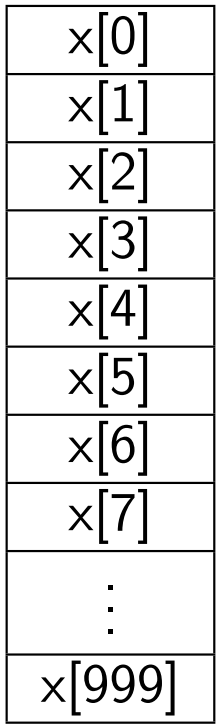
- array name
- location index

e.g.

```
x[3] = 100;
```

or

```
printf("%d", x[3]);
```



The elements of an array of size n are accessible using integer indices from 0 to $n-1$.

Accessing an Array Element

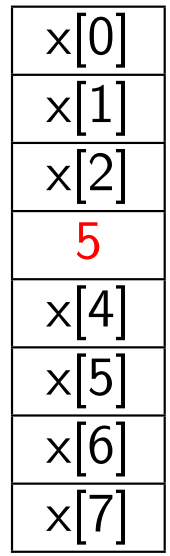
```
int main() {
    int i;
    int x[1000];

    for(i = 0; i < 1000; i++) {
        x[i] = 0;
    }

    return 0;
}
```

The elements of an array of size n are accessible using integer indices from 0 to $n - 1$.

Array Elements

The array elements are and can be used as normal variables.

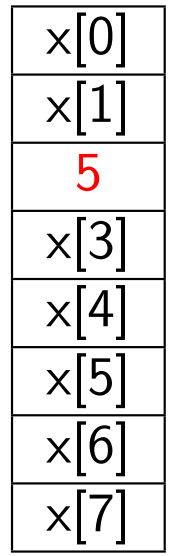
```
int main() {
    int x[8];

    x[3]= 5;

    printf("%d\n", x[3]);

    return 0;
}
```



Array Elements

We can provide any expression which produces an integer as an array index.

```
int main() {
    int x[8], j;

    j = 1;
    x[j+1] = 5;

    if(x[4-2] == 5) {
        printf("x[2] = 5\n");
    }

    return 0;
}
```



Expressions as Array Indices

Before an array is accessed the expression inside the square brackets [ ] must be calculated.

```
int cyclenext8(int i) {
    if (i >= 8||i < 0) {
        return 0;
    } else {
        return i+1;
    }
}

int main() {
    int i, y[8], x[8];

    for(i=0; i < 8; i++)
        x[i] = i;

    for(i = 0; i < 8; i++)
        y[i] = x[cyclenext8(i)];

    return 0;
}
```

Operators Precedence

Square brackets [ ] have the highest priority.

1. **parentheses:** (), [], `expr++`, `expr--`
   Calculated first, from left to right. Nested are calculated first.
2. **unary operators:**
   +, −, `++expr` or `--expr` (prefix) Calculated from right to left.
3. **multiplication, division and remainder:** ∗, /, or %
   Calculated from left to right.
4. **addition, subtraction:** + or −
   If there are many, calculated from left to right.
5. **relational:** <, >, <=, >=
   Calculated from left to right.
6. **equality:** ==, ! =
   Calculated from left to right.
7. **boolean AND:** && Calculated from left to right.
8. **boolean OR:** || Calculated from left to right.
9. **assignment:** =, + =, − =, ∗ =, / =, % =
   From right to left.

Array Initialization

Like all local variables in C, arrays need to be initialized.

```
int main() {
    int i;
    int x[1000];

    for(i = 0; i < 1000; i++) {
        x[i] = 0;
    }
}
```

Array Initialization

In case an array is small enough, we can initialize it directly when we define it.

```
int main() {
    int x[10] = { 32, 27, 64, 18, 95,
                  14, 90, 70, 60, 37 };
}
```

32
27
64
18
95
14
90
70
60
37

Array Initialization

An array does not get initialized automatically.

```
int main() {
    int x[1000] = { 0 };
}
```

If, however, we initialize fewer elements than its size, the compiler makes sure that the rest of the elements get initialiazed with zero values.

Array Initialization

The compiler can also understand the array size from its static initialization list.

```
int main() {
    int x[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
}
```

The array `x[]` is `x[9]`.

Array Size and Defines

Recall how we can define constants using the pre-processor.

```c
#include <stdio.h>

#define SIZE 100

int main() {
    int a[SIZE];
    for(int i = 0; i < SIZE; i++) {
        a[i] = i;
    }

    for(int i = 0; i < SIZE; i++) {
        printf("%d", a[i]);

        if (i < SIZE-1)
            printf(" ");
        else
            printf("\n");
    }
    return 0;
}
```

Array Size and Defines

We will later see that sometimes it is not possible to know the size of an array during compilation.

For this reason the C language supports:

1. static arrays
2. dynamic arrays

Dynamic arrays will be handled later. For now we only care about static ones (whose size is known during compilation).

Out-of-bounds Error

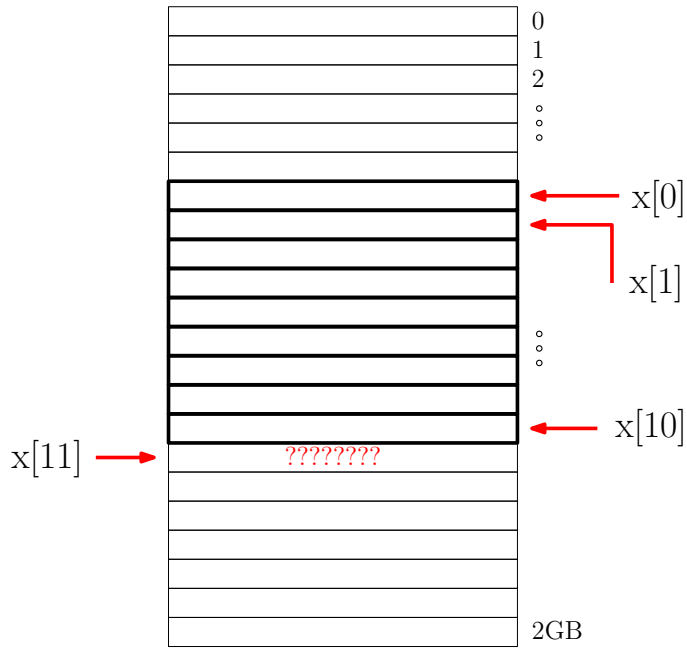
What happens if we define an array of 10 elements and by accident access its 11th element?

```c
#include <stdio.h>

int main() {
    int a[10] = { 10, 1, 3, 4, 5, 9, 2, 9, 10, 10 };

    for(int i = 0; i < 11; i++) {  /* RUN-TIME ERROR */
        printf("%d\n", a[i]);
    }

    return 0;
}
```

The result is undefined, anything can happen!

Out-of-bounds Error

What happens if we define an array of 10 elements and by accident access its 11th element?



Usually we either get "garbage" or the program terminates abnormally (crash).

Example

Sum

```c
#include <stdio.h>
#define SIZE 12

int main() {
    int a[SIZE] = { 1, 3, 5, 4, 7, 2, 99,
                    16, 45, 67, 89, 45 };
    int i, total = 0;

    for(i = 0; i < SIZE; i++) {
        total += a[i];
    }

    printf("Total of array elements is %d\n", total);

    return 0;
}
```

prints

Total of array elements is 383

Example

Average

```c
#include <stdio.h>
#define SIZE 6

int main() {
    double a[SIZE] = { 1.0, 2.3, 1.3333, 7.102, 1.523, 1.0001 };
    int i;
    double avg = 0.0;

    for(i = 0; i < SIZE; i++) {
        avg += a[i];
    }

    avg /= SIZE;

    printf("average is %lf\n", avg);

    return 0;
}
```

prints

Character Arrays

Strings in C are arrays of characters with the special characteristic that the last character is the special null character `'\0'`.

```
int main() {
    char string[6];

    string[0] = 'f';
    string[1] = 'i';
    string[2] = 'r';
    string[3] = 's';
    string[4] = 't';
    string[5] = '\0';
}
```

The null character is necessary in order for various functions to be able to tell when a string ends without the need for an additional user parameter.

Character Arrays

The program below

```
int main() {
    char string[6];

    string[0] = 'f';
    string[1] = 'i';
    string[2] = 'r';
    string[3] = 's';
    string[4] = 't';
    string[5] = '\0';

    printf( "%s\n", string );
    printf( "first\n" );
}
```

prints twice the word *first*.

Character Arrays

We can also initialize the character array faster.

```
int main() {
    char string[6] = "first";

    printf("%s\n", string);
    printf("first\n");
}
```

the compiler adds automatically the `'\0'`.

Character Arrays

We can also omit the array size.

```
int main() {
    char string[] = "first";

    printf("%s\n", string);
    printf("first\n");
}
```

the compiler finds out automatically.

Notice that the size of the array is one more than the size of the string.

Character Arrays

'\0' position

The position of the null character designates the end of the string.

```
#include <stdio.h>

int main() {
    char string[11] = { 't', 'w', 'o', '\0', ' ', 'w',
                        'o', 'r', 'd', 's', '\0' };

    printf("%s\n", string);
}
```

The program above prints *two* and not *two words*, since `printf` stops when it first finds the null character.

Character Arrays

Initialization to an empty string

The position of the null character determines the end of the string.

```c
#include <stdio.h>

int main() {
    char string[11] = { 't', 'w', 'o', '\0', ' ', 'w',
                        'o', 'r', 'd', 's', '\0' };

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

    string[0] = '\0';
}
```

In order to create an empty string we can set the first character to `'\0'`.

Arrays and Functions

We can declare a function which accepts an array as parameter.

```
void swap(int array[], int i, int j) {
    /* code to swap element array[i] and array[j] */
}
```

end then call it like

```
int main() {
    int a[5] = { 1, 2, 2, 3, 1 };

    swap(a, 2, 3);
}
```

giving only the name of the array as a parameter.

Arrays and Call-By-Value

Although C always uses **call-by-value**, the values of the elements of any array can change inside a function.

```c
#include <stdio.h>

void increment(int[] a, int n) {
    int i;
    for(i = 0; i < n; i++) {
        a[i]++;
    }
}

int main() {
    int a[] = { 1, 2, 3 };

    increment(a, 3);

    /* here a[] = { 2, 3, 4 } */
}
```

The reason is that C actually passes only the location of the array to the function, but we will have this discussion later on.

Arrays and Call-By-Value

The use of keyword **const** tells the compiler to not allow any change of array elements inside the function.

```
#include <stdio.h>

void increment(const int[] a, int n) {  /* const here */
    int i;
    for(i = 0; i < n; i++) {
        a[i]++;   /* COMPILATION ERROR */
    }
}

int main() {
    int a[] = { 1, 2, 3 };

    increment(a, 3);
}
```

Example

Swapping Array Elements

```c
#include <stdio.h>

void swap(int a[], int i, int j);

int main() {
    int n[] = { 1, 2, 3 };
    swap(n, 0, 1);
    swap(n, 1, 2);
}

void swap(int a[], int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Example

Reversing Array Elements

```c
#include <stdio.h>

void swap(int a[], int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

void reverse(int a[], int n) {
    int i;
    for(i = 0; i < n/2; i++) {
        swap(a, i, n-i-1);
    }
}

int main() {
    int a[] = { 1, 2, 3, 4, 5 };
    reverse(a, 5);
}
```

Multi-Dimensional Arrays

C allows to define multi-dimensional arrays.

```
int marray[3][4];
```

e.g. an array with 3 lines and 4 columns which is called `marray`.

Multi-Dimensional Arrays

$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$

and a classic **loop** in order to initialize the array

```
for(i = 0; i < 3; i++) {
    for(j = 0; j < 4; j++) {
        a[i][j] = 0;
    }
}
```

Static Initialization of Multi-Dimensional Arrays

```
int a[2][2] = { { 1, 2 }, { 3, 4 } };
```

Values are groups per row using curly brackets.

Multi-Dimensional Arrays and Functions

```c
#include <stdio.h>
#define COLUMNS  3

int max(const int a[][COLUMNS], int rows, int columns) {
    int i,j;
    int max = a[0][0];

    for(i = 0; i < rows; i++) {
        for(j = 0; j < columns; j++) {
            max = a[i][j] > max?a[i][j]:max;
        }
    }

    return max;
}

int main() {
    const int a[2][COLUMNS] = { { 1, 2, 3 }, { 1000, 0, 1 } };
    printf("maximum = %d\n", max(a, 2, COLUMNS));
}
```

It is possible to omit the first dimension from the compiler, but not the second since the compiler needs to know the shape of the array.

Multi-Dimensional Arrays and Functions

Two-dimensional arrays are implemented in C using one dimensional, e.g. for an array `int a[3][4]` we have

- `a[i][j] = a[4*i + j]`

For this reason the compiler needs to known the number of columns (4 in the above example) when we provide an array inside a function.

Two-Dimensional Arrays Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{pmatrix}$$

```
for(i = 0; i < n; i++) {
    for(j = 0; j < p; j++) {
        for(k = 0, c[i][j] = 0.0; k < m; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```