Introdução à Programação 2020/2021

António J. R. Neves
Daniel Corujo

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

an@ua.pt / dcorujo@ua.pt

http://elearning.ua.pt/

Topics

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

- Arrays
- Pointers

- An array contains one or more objects of a given type, stored consecutively in a continuous memory block.
- The individual objects are called the elements of an array.
- The definition of an array determines its name, the type of its elements, and the number of elements in the array.

```
type name[ number_of_elements ];
```

- The number of elements, between square brackets (`[]`), must be an integer expression whose value is greater than or equal zero. Here is an example:

```
char buffer[5];
```

- You can determine the size, in bytes, of the memory block that an array occupies using the **`sizeof`** operator. The array's size in memory is always equal to the size of one element times the number of elements in the array.

- Most array definitions specify the number of array elements as a constant expression. An array so defined has a fixed length.
- The four array definitions in the following example are all valid:

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

void func(void) {
   static int c[10];
   int d[10];
   /* ... */
}
```

- It is possible to define an array using a non constant expression for the number of elements if the array has automatic storage duration. Such an array is then called a variable-length array.
- The array definition in the following example is valid:

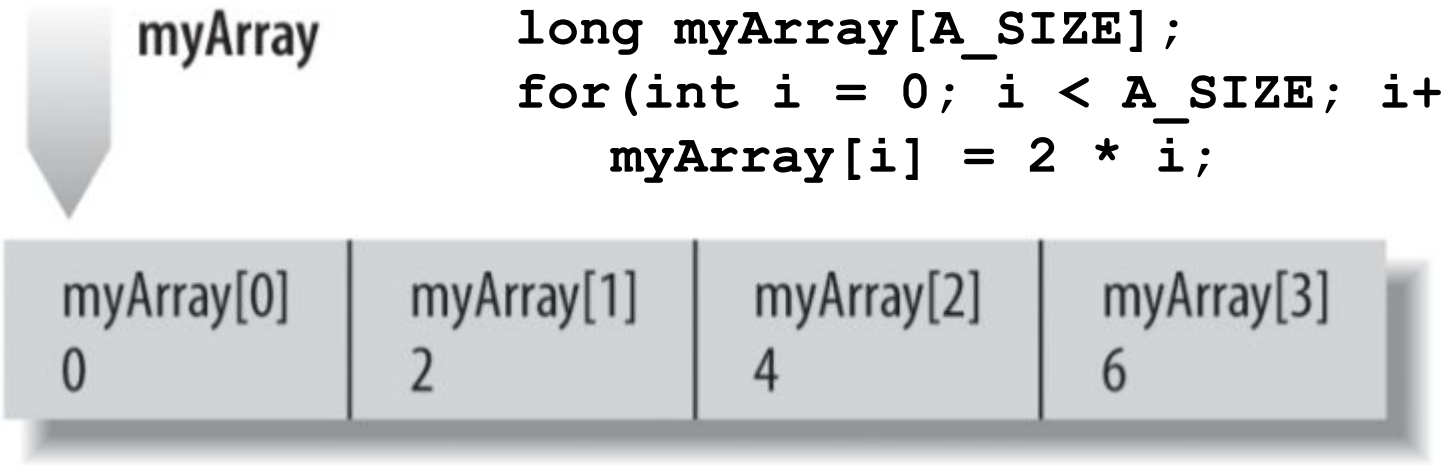
```
void func( int n ) {
   int vla[2 * n]; // DANGEROUS when n < 0
   ...
```

- Like any other automatic variable, a variable-length array is created each time the program flow enters the block containing its definition. As a result, the array can have a different length at each such instantiation.

- Storage for automatic objects (those declared inside a function without the **`static`** keyword) is allocated on the stack, and is released when the program flow leaves the block.
- For this reason, variable-length array definitions are useful only for small, temporary arrays.
- To create larger arrays dynamically, you should generally allocate storage space explicitly using the **`new`** expression.
- The storage duration of such arrays then ends with the end of the program or when you release the allocated memory with the **`delete[]`** expression.
- **We will get to this later in the course...**

- The subscript operator `[]` provides an easy way to address the individual elements of an array by way of an index.
- The following code fragment defines the array **`myArray`** and assigns a value to each element.

```
const int A_SIZE = 4;
long myArray[A_SIZE];
for(int i = 0; i < A_SIZE; i++)
   myArray[i] = 2 * i;
```



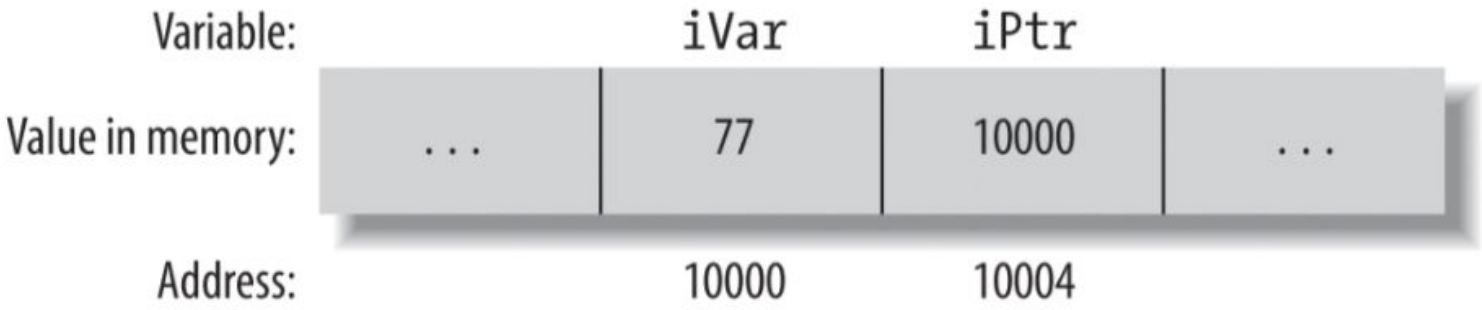
1. To initialize an array explicitly when you define it, you must use an initialization list:
```
int a[4] = { 1,2,4,8 };
```
2. You cannot include an initialization in the definition of a variable-length array.
3. If the array has static storage duration, then the array initializers must be constant expressions.
4. You may omit the length of the array in its definition if you supply an initialization list.
```
int a[] = { 1, 2, 4 }; // An array with three elements
```
5. Element designators allow you to associate initializers with specific elements:
```
int a[20] = { 1, [12] = 2, 4, [5] = 8 }; // a[13] = 4
```

- A pointer is a reference to a data object or a function.
- Very often the only efficient way to manage large volumes of data is to manipulate not the data itself but pointers to the data (ex: sort large structures, pass data to a function, etc.).
- A pointer represents both the address and the type of an object or function.
- A simple example:

```
int iVar = 77;
int *iPtr;     // Declare iPtr as a pointer to int.
iPtr = &iVar; // iPtr now "points" to the variable iVar
```

- The asterisk (*) means "pointer to." The type **`int`** is the type of object that the pointer **`iPtr`** can point to. The ampersand (&) here means: get the address of the variable (in this case **`iVar`**).

- Possible arrangement of the variables **`iVar`** and **`iPtr`** in memory:



- It is often useful to output addresses for verification and debugging purposes. The **`printf()`** functions provide a format specifier for pointers: **`%p`**.
- The size of a pointer in memory is the same regardless of the type of object addressed. On 32-bit processors, pointers are usually four bytes long; on 64-bit processors, they usually are eight bytes long.

- By convention, a pointer that does not point to anything (a so-called) null pointer, has the value **`NULL`**.
- A pointer to **`void`** is a pointer with the type **`void *`**; it is used as the all-purpose pointer type.
- A **`void`** pointer can represent the address of any object. To access an object in memory, you must always convert a pointer to **`void`** into an appropriate object pointer.
- **`NULL`** is a special pointer to **`void`**. It points to the start of the memory address space (address 0), which is an illegal address.

- You can initialize a pointer (to an object) with the following kinds of initializers:
  - A **`NULL`** pointer constant;
  - A pointer to the same type, or to a less qualified version of the same type;
  - A pointer to **`void`**, if the pointer being initialized is not a function pointer;
- When you initialize a pointer, no implicit type conversion takes place except in the cases above.

```
double x = 1.5;
char *cPtr = &x; // Error: type mismatch
char *cPtr = (char *)&x; // OK: cPtr points to the
                         //   start of the memory area
                         //   where x is stored
```

- You can perform the following operations on pointers to objects (to be explained in detail later on):
  - Adding an integer to, or subtracting an integer from, a pointer.
  - Subtracting one pointer from another.
  - Comparing two pointers.
- However, the most important operation with pointers is accessing the object that the pointer refers to. If **`ptr`** is a pointer, then **`*ptr`** designates the object that **`ptr`** points to.

```
double x, y, *ptr; // double variables and a pointer to double
ptr = &x;          // Let ptr point to x
                   // (now x and *ptr are the same thing)
*ptr = 7.8;        // Assign the value 7.8 to the variable x
*ptr *= 2.5;       // Multiply x by 2.5
y = *ptr + 0.5;    // Assign to y the result of x + 0.5
```

- Pointers occur in many C++ programs as references to arrays, and also as elements of arrays. A pointer to an array type is called an array pointer.

```
int a[3];      // Array of three integers
int *ptr = a; // The array name is a pointer to the
               // first element; i.e., the first row
ptr[0] = 5;    // Assign the value 5 to the first element
```

- The expression **`a + i`** is a pointer to **`a[i]`**, and the value of **`*(a+i)`** is the element **`a[i]`**.
- According to the rules, the expression **`arrPtr[i]`** is equivalent to **`*(arrPtr + i)`**.

```
*(ptr + 2) = 1; // Assign the value 1 to the third element
                //   it is the same as prt[2] = 1;
```

- A final example:

```
int a[10];       // Array of 10 integers
int *b = &a[7]; // From now on, b[0] is the same as a[7],
                //   hence b[i] is the same as a[7 + i]
int x = b[-3];  // Since b[-3] is the same as a[7 - 3], assign
                //   to x the value stored in a[4]
```

- Using, by mistake, an out-of-range index usually gives rise to a program crash (segmentation fault) or to unexpected program behavior. The code generated by the C++ compiler **does not check for them** (the -Wall compiler option may detect some cases). In the previous code, **`a[i]`** has a well defined value only when 0 <= i < 10, and **`b[i]`** has a well defined value only when -7 <= i < 3. Using any other index value is a conceptual error.

- When the element type of an array is another array, it is said that the array is multidimensional:

```
// array of 2 arrays of 3 int each
int a[2][3] = {{1, 2, 3}, //can be viewed as a 2×3 matrix
               {4, 5, 6}};

// printing the content of the array
for (int r = 0; r < 2; r++){
    for (int c = 0; c < 3; c++) {
        cout << m[r][c] << " ";
    }

    cout << endl;
}
```