Arrays in C/C++

Ein Array ist eine Kombination mehrerer Variablen gleichen Typs. Die Elemente des Arrays werden über ihre Positionsnummer angesprochen.

Der Begriff "Array" wird in der deutschen Fachliteratur mit dem Wort "Feld" übersetzt. Leider ist dieser Begriff nicht sehr markant, so dass viele Programmierer den Begriff "Array" bevorzugen.

Realitätsabbildung

Sie können mit einem Array also eine Reihe gleicher Elemente ansprechen. Ein Beispiel sind die Lottozahlen. Es werden immer sechs ganze Zahlen gezogen. Wenn Sie also die Lottozahlenziehung in einem Programm simulieren wollen, bietet es sich an, ein Array von sechs Integern zu verwenden.

lotto[0] lotto[1] lotto[2] lotto[3] lotto[4] lotto[5]

In der Abbildung sehen Sie wie die Lottozahlen nebeneinander stehen. Unter jedem Kasten befindet sich in eckigen Klammern die Position jeder Lottozahl. Entgegen der Gewohnheit der meisten Menschen beginnt ein Array immer an der Position 0. Wenn also sechs Zahlen im Array stehen, befindet sich die letzte an Position 5.

Deklaration von Arrays

Um das Array für die Lottozahlen im Programm zu deklarieren, geben Sie zunächst den Typ eines einzelnen Elements an. Als nächstes kommt der Name des Arrays. Es folgt in eckigen Klammern die Anzahl der Elemente, die das Array maximal aufnehmen können soll.

```
int lotto[6];
```

Zugriff

Wenn Sie auf ein Element des Arrays zugreifen wollen, nennen Sie zuerst den Namen des Arrays. Es folgt in eckigen Klammern die Position des Elements, auf das Sie zugreifen wollen. Denken Sie dabei daran, dass die Array-Positionen immer bei 0 beginnen. **`lotto[0]`**

Initialisierung

```
lotto[0] = 5;
lotto[1] = 12;
lotto[2] = 13;
lotto[3] = 28;
lotto[4] = 32;
lotto[5] = 43;
```

Die Zeile zeigt, wie das 1. Element mit einer Zahl zwischen 1 und 49 gefüllt wird.

Zugriff

Wenn Sie auf ein Element des Arrays zugreifen wollen, nennen Sie zuerst den Namen des Arrays. Es folgt in eckigen Klammern die Position des Elements, auf das Sie zugreifen wollen.

```
cout << lotto[0];
```

Es wird das erste Element auf dem Bildschirm ausgegeben.

Eckige Klammern

Das syntaktische Erkennungszeichen eines Arrays sind die eckigen Klammern. Sie werden bei der Definition eines Arrays verwendet, um anzugeben, wie viele Variablen zu dem Array gehören. Die eckigen Klammern werden auch verwendet, wenn auf ein Element zugegriffen werden soll. Darum werden die eckigen Klammern Indexoperator genannt. Das Beispiel zeigt, wie das Lottozahlen-Array mit Zufallszahlen gefüllt wird:

```
int lotto[6];

lotto[0] = rand();
lotto[1] = rand();
lotto[2] = rand();
lotto[3] = rand();
lotto[4] = rand();
lotto[5] = rand();
```

Die Zufallsfunktion rand() erstellt Pseudozufallszahlen!
Dieser Zufall wiederholt sich, außer man verwendet

```
srand(time(NULL)); // benötigt time.h
```

Lottoautomat Projekt

1. Erstellen Sie ein Projekt mit dem Namen Lottoautomat und schreiben ein Programm, was ein Lottoarray mit 6 Zufallszahlen füllt.

2. Geben Sie alle Lottozahlen mit Hilfe einer for-Schleife aus. Verbessern Sie die Ziehung oben ebenfalls mit einer for-Schleife, so dass rand() nur einmal im Quelltext steht.

3. Verbessern Sie das Programm, so dass nur Zahlen zwischen 1 und 49 gezogen werden.

4. Verbessern Sie das Programm, so dass keine Zahlen doppelt gezogen werden können.

5. Ein Benutzer soll 6 Zahlen zwischen 1 und 49 eingeben. Anschließend wird die Lottoziehung durchgeführt und danach wird dem Benutzer mitgeteilt, wieviel Richtige er hat und welche Zahlen richtig waren.

Zusatzaufgabe:

Sortieren Sie die Lottoarrays z.B. mit dem Bubblesort-Algorithmus.
Dabei werden 2 Nachbarfelder verglichen und eventuell vertauscht. Dieses Vertauschen dauert maximal 5 Durchläufe mit 5 mal Vertauschen.

So dass anstatt	38	5	2	49	20	11	
	2	5	11	20	38	49	steht

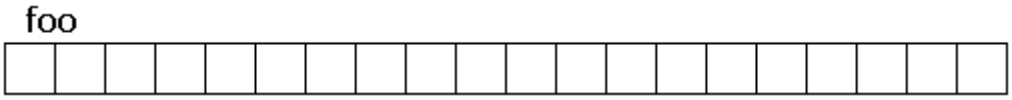
Character sequences (Strings)

Strings are, in fact, sequences of characters, we can represent them also as plain arrays of elements of a character type.

For example, the following array:

```
char foo [20];
```

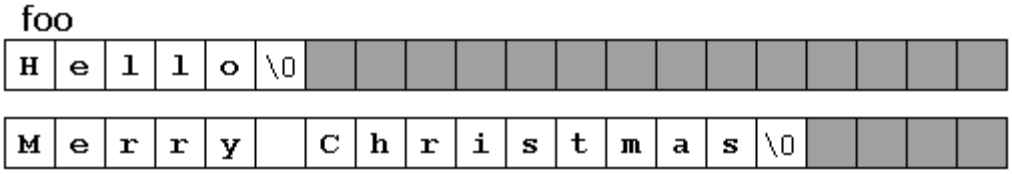
is an array that can store up to 20 elements of type `char`. It can be represented as:



Therefore, this array has a capacity to store sequences of up to 20 characters. But this capacity does not need to be fully exhausted: the array can also accommodate shorter sequences. For example, at some point in a program, either the sequence `"Hello"` or the sequence `"Merry Christmas"` can be stored in `foo`, since both would fit in a sequence with a capacity for 20 characters.

By convention, the end of strings represented in character sequences is signaled by a special character: the *null character*, whose literal value can be written as `'\0'` (backslash, zero).

In this case, the array of 20 elements of type `char` called `foo` can be represented storing the character sequences `"Hello"` and `"Merry Christmas"` as:



Notice how after the content of the string itself, a null character (`'\0'`) has been added in order to indicate the end of the sequence. The panels in gray color represent `char` elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays, they follow the same rules as these. For example, to initialize an array of characters with some predetermined sequence of characters, we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The above declares an array of 6 elements of type `char` initialized with the characters that form the word `"Hello"` plus a *null character* `'\0'` at the end.

But arrays of character elements have another way to be initialized: using *string literals* directly.

For example:

```
"the result is: "
```

This is a *string literal*.

Sequences of characters enclosed in double-quotes (") are *literal constants*. And their type is, in fact, a null-terminated array of characters. This means that string literals always have a null character ('\0') automatically appended at the end.

Therefore, the array of char elements called myword can be initialized with a null-terminated sequence of characters by either one of these two statements:

```
1 char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 char myword[] = "Hello";
```

In both cases, the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello", plus a final null character ('\0'), which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Please notice that here we are talking about initializing an array of characters at the moment it is being declared, and not about assigning values to them later (once they have already been declared). In fact, because string literals are regular arrays, they have the same restrictions as these, and cannot be assigned values.

Nicht erlaubt ist folgendes nach der Deklaration:
Expressions (once myword has already been declared as above), such as:

```
1 myword = "Bye";
2 myword[] = "Bye";
```

would **not** be valid, like neither would be:

```
myword = { 'B', 'y', 'e', '\0' };
myword[] = { 'B', 'y', 'e', '\0' };
```

This is because arrays cannot be assigned values. Note, though, that each of its elements can be assigned a value individually. For example, this would be correct:

Erlaubt ist nach der Deklaration nur noch:

```
myword[0] = 'B';
myword[1] = 'y';
myword[2] = 'e';
myword[3] = '\0';
```

Pointers and string literals

String literals are arrays containing null-terminated character sequences. In earlier sections, string literals have been used to be directly inserted into `cout`, to initialize strings and to initialize arrays of characters.

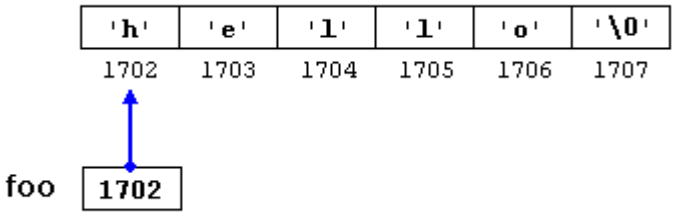
But they can also be accessed directly. String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type `const char` (as literals, they can never be modified). For example:

```
char* foo = "hello";
```

besser daher:

```
const char* foo = "hello";
```

This declares an array with the literal representation for `"hello"`, and then a pointer to its first element is assigned to `foo`. If we imagine that `"hello"` is stored at the memory locations that start at address 1702, we can represent the previous declaration as:



Note that here `foo` is a pointer and contains the value 1702, and not `'h'`, nor `"hello"`, although 1702 indeed is the address of both of these.

The pointer `foo` points to a sequence of characters. And because pointers and arrays behave essentially in the same way in expressions, `foo` can be used to access the characters in the same way arrays of null-terminated character sequences are. For example:

```
1 *(foo+4)
2 foo[4]
```