3-COMPOUND DATA TYPES

ARRAYS

There is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent.
int foo [ ] = { 10, 20, 30 } ;
int foo [ ] { 10, 20, 30 };

If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types.)

Accessing The Values Of An Array

exceeding = aşma, aşırı, ölçüsüz

Two uses that bracket [ ] have related to arrays. They perform two different tasks:
1- Specify the size of arrays when they are declared; **int foo [5];**
2- Specify indices for concrete array elements when they are accessed. **foo[2] = 75;**

foo [0] = a;
foo [a] = 75;
b = foo [a + 2];
foo [foo [a]] = foo [2] + 5;

Multidimensional Arrays

The amount of memory needed for an array increases exponentially with each dimension.
char century [100] [365] [24];

int jimmy [3] [5]; //It is equivalent to
int jimmy [15]; // (3 * 5 = 15)

The following two pieces of code produce the exact same result, but one uses a bidimensional array while the other uses a simple array:

multidimensional array	pseudo-multidimensional array

`#define WIDTH 5`
`#define HEIGHT 3`
 	
`int jimmy [HEIGHT][WIDTH];`
`int n,m;`
 	
`int main ()`
`{`
`  for (n=0; n<HEIGHT; n++)`
`    for (m=0; m<WIDTH; m++)`
`    {`
`      jimmy[n][m]=(n+1)*(m+1);`
`    }`
`}`
	`#define WIDTH 5`
	`#define HEIGHT 3`
	
	`int jimmy [HEIGHT * WIDTH];`
	`int n,m;`
	
	`int main ()`
	`{`
	`  for (n=0; n<HEIGHT; n++)`
	`    for (m=0; m<WIDTH; m++)`
	`    {`
	`      jimmy[n*WIDTH+m]=(n+1)*(m+1);`
	`    }`
	`}`

The code uses defined constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

snippet = ufak parça, kırıntı

Arrays As Parameters
It is not possible to pass the entire block of memory represented by an array to a function directly as an argument.

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

```
5 10 15
2 4 6 8 10
```

In the code above, the first parameter (**int arg [ ]**) accepts an array whose elements are of type **int**, whatever its length. For that reason, we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter.

void procedure (**int** myarray [ ] [3] [4])
The first brackets [ ] are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

In a way, passing an array as argument **always loses a dimension**. Arrays cannot be directly copied, and thus what is really passed is a pointer.

novice = acemi

Library Arrays
decay = çürümek, bozulma, parçalanmak

C++ provides an alternative array type as a standard container. It is a type template (a class template, in fact) defined in header **<array>**. It allows to be copied the array directly, but it put away. (expensive operation)

language built-in array	container library array

`#include <iostream>`
 	
`using namespace std;`
 	
`int main()`
`{`
`  int myarray[3] = {10,20,30};`
 	
`  for (int i=0; i<3; ++i)`
`  ++myarray[i];`
 	
`  for (int elem : myarray)`
`  cout << elem << '\n';`
`}`
	`#include <iostream>`
	`#include <array>`
	`using namespace std;`
	
	`int main()`
	`{`
	`  array<int,3> myarray {10,20,30};`
	
	`  for (int i=0; i<myarray.size(); ++i)`
	`  ++myarray[i];`
	
	`  for (int elem : myarray)`
	`  cout << elem << '\n';`
	`}`

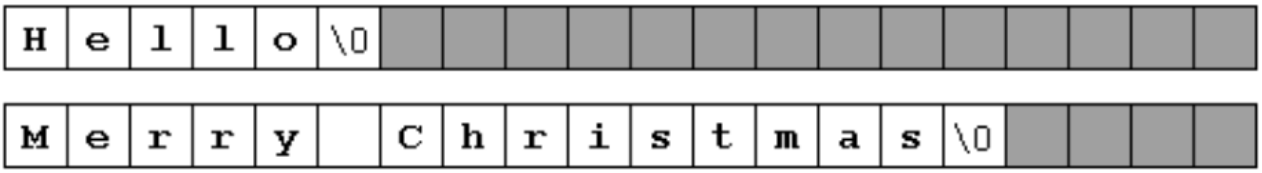
As you can see, both kinds of arrays use the same syntax to access its elements: **myarray[i]**. Other than that, the main differences lay on declaration of the array, and the inclusion of an additional header for the **library array.** Notice also how it is easy to access the size of the **library array.**

CHARACTER SEQUENCES

plain = yalın

by convention = genel kabul, anlaşma ile

The end of strings in character sequences is signaled by a special character: the null character (\0).



The panels in gray color represent **char** elements with undetermined values.

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.

String literals always have a null character (\0) automatically appended at the end.

char myword [ ] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’ };

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.

String literals are regular arrays, they have the same restrictions as these, and cannot be assigned values.

Expressions (once myword has already been declared as above), such as:

myword = “Bye”;

myword [ ] = “Bye”;

would not be valid, like neither would be:

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 individiually. This would be correct:

myword [0] = ‘B’;
myword [1] = ‘y’;
myword [2] = ‘e’;
myword [3] = ‘\0’;

Strings And Null-Terminated Character Sequences

```
// strings and NTCS:
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  char question1[] = "What is your name? ";
  string question2 = "Where do you live? ";
  char answer1 [80];
  string answer2;
  cout << question1;
  cin >> answer1;
  cout << question2;
  cin >> answer2;
  cout << "Hello, " << answer1;
  cout << " from " << answer2 << "!\n";
  return 0;
}
```

```
What is your name? Homer
Where do you live? Greece
Hello, Homer from Greece!
```

Strings have a dynamic size determined during runtime, while the size of arrays is determined on compilation, before the program runs.

```
char myntcs[] = "some text";
string mystring = myntcs;  // convert c-string to string
cout << mystring;          // printed as a library string
cout << mystring.c_str();  // printed as a c-string
```

Null-terminated character sequences can be transformed into strings implicitly, and strings can be transformed into null-terminated character sequences by using either of string’s member functions **c_str** or **data.**

POINTERS

Adres asla negatif bir sayı olamaz! (unsigned)
Intel işlemciler Little Endian’a göre sayıyı yerleştirir.
Mesela, 5 sayısını;
00000000 00000000 00000000 00000101

00000101 (8 bit, 1 byte) (başı)
00000000 (8 bit, 1 byte)
00000000 (8 bit, 1 byte)
00000000 (8 bit, 1 byte) (sonu)
Olmak üzere 4 byte (int) yerleştirir.

int a = 5;
int * p = &a;
p işaretçisi a yı gösteriyor.

```
int Degistir ( int* , int* );
int main ( ) {
        int sayi1, sayi2;
sayi1 = 3;
sayi2 = 5;
int sonuc = Degistir ( &sayi1, &sayi2 );
}

int Degistir ( int* sayi1, int* sayi2 ) {
        *sayi1 = 20;
        *sayi2 = 40;
cout<< sayi1 << sayi2;
        return *sayi1 + * sayi2;
}
```

p [0] ile *(p+0) aynı.
p [1] ile *(p+1) aynı.

```
int a[ ] = {1,2,3,4,5,6,7,8,9,10 } ;
int * p = a;

*p++;  //p işaretçisinin tuttuğu adresi artırır.
(*p)++ //p işaretçisinin tuttuğu değeri artırır.
```