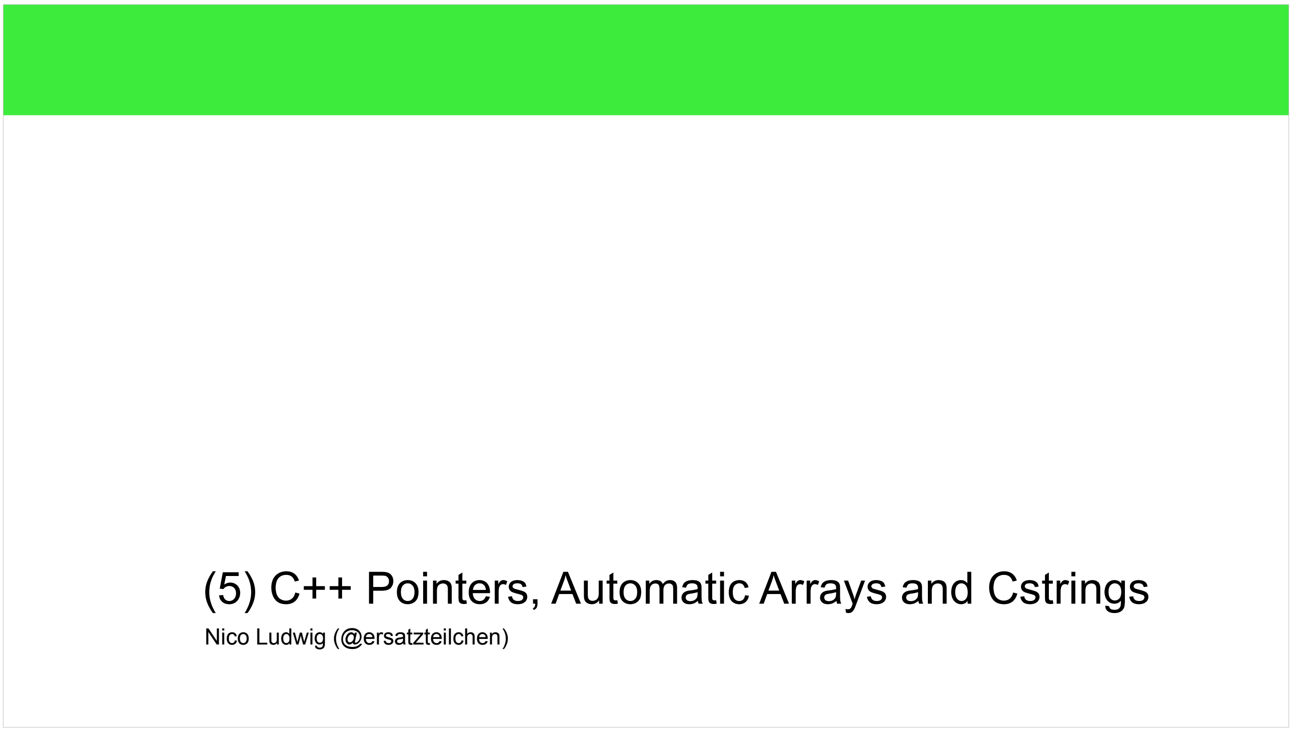


(5) C++ Pointers, Automatic Arrays and Cstrings

Nico Ludwig (@ersatzteilchen)

TOC

- (5) C++ Pointers, Automatic Arrays and Cstrings
  - Pointers: Call by Value versus Call by Reference
  - Automatic Arrays
  - Arrays and Pointer Decay
  - Pointers to Pointers
  - Cstrings as Arrays and their Memory Representation
  - Basic Cstring Functions

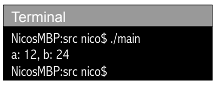
- Sources:
  - Bruce Eckel, Thinking in C++ Vol I
  - Bjarne Stroustrup, The C++ Programming Language

Passing Function Arguments by Value – Motivation

- By default, function arguments are getting passed by value in C++.
  - This means that in a function a parameter is a copy of the passed argument.
  - Assigning to the parameter in the function won't affect the original argument!
  - A function like *swap()* that should swap the contents of its arguments cannot be implemented like so:

```
// Suspicious implementation of swap()!
void swap(int first, int second) {
	int temp = first;
	// Assignment affects only the parameter!
	first = second;
	// Assignment affects only the parameter!
	second = temp;
}
```

```
int main() {
	int a = 12, b = 24;
	swap(a, b);
	// … but the values of a and b won't be swapped:
	std::cout<<"a: "<<a<<", b: "<<b<<std::endl;
}
```



```
NicosMBP:src nico$ ./main
a: 12, b: 24
NicosMBP:src nico$
```

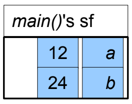
- To better understand, what's happening here, we have to take a look in how memory is involved!

Passing Function Arguments by Value – Stackframes

- Each function gets a portion of automatic memory, in which all its locals are stored.
  - The automatic memory is located in the so-called stack memory, we say, that a function has a stackframe (sf).
  - The memory is "automatic", because this memory is automatically given back to the CPU, when the function returns.

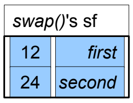
- Here, *main()* defines two local ints, *a* and *b*. *a* and *b* are existing in the sf of *main()*.
  - When *main()* calls *swap()*, it passes *a* and *b* as arguments:

```
int main() {
        int a = 12, b = 24;
        swap(a, b);
        std::cout<<"a: "<<a<<", b: "<<b<<std::endl;
        // >a: 12, b: 24
}
```



- When *swap()* is called, the passed arguments are copied into *swap()*s int params *first* and *second*.
  - *swap()*'s parameters are nothing but local variables of *swap()*, i.e. they "live" in *swap()*'s sf.
  - When *first* or *second* are modified in *swap()*, those modifications will only affect *swap()*'s sf!

```
// Suspicious implementation of swap()!
void swap(int first, int second) {
        int temp = first;
        first = second;
        second = temp;
}
```

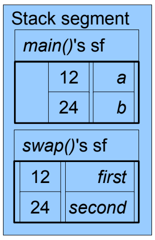


Passing Function Arguments by Value – Stackframes and Copying

- Technically, call by value appears, because caller and callee store their locals on different stackframes.

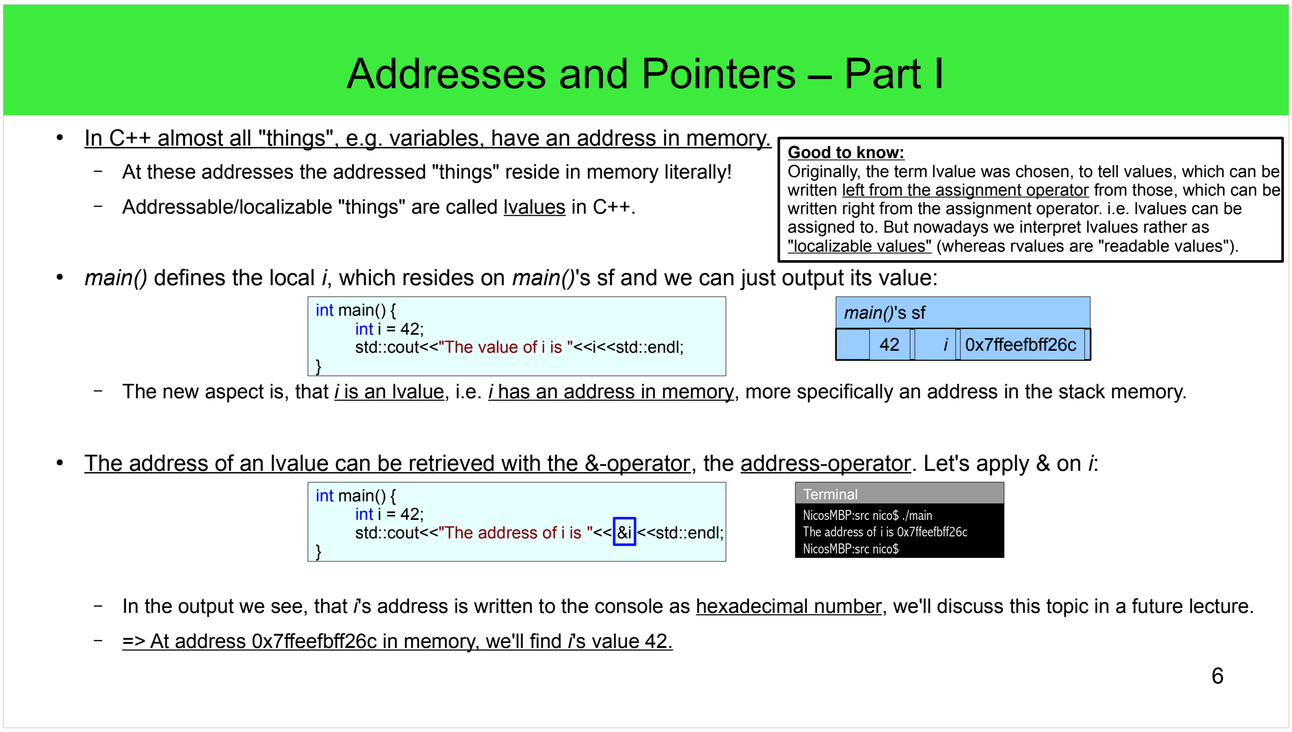
- Call by value means, that functions don't communicate by one function directly accessing another function's stackframe.
  - The communication only works via parameters and return values, which are copied among stackframes.

- The stackframe of a caller- and a callee-function exist at the same time and for the full live time of the function call.
  - These "active stackframes" exist in the same stack segment of the stack memory of the "program", but in adjacent regions:



  - Stack memory is highlighted in light blue color in the box graphics of this course.

- Actually, C++ allows callee-functions to access caller-functions's stackframes via so called pointers.



Addresses and Pointers – Part I

- In C++ almost all "things", e.g. variables, have an address in memory.
  - At these addresses the addressed "things" reside in memory literally!
  - Addressable/localizable "things" are called lvalues in C++.

Good to know:
Originally, the term lvalue was chosen, to tell values, which can be written left from the assignment operator from those, which can be written right from the assignment operator. i.e. lvalues can be assigned to. But nowadays we interpret lvalues rather as "localizable values" (whereas rvalues are "readable values").

- *main()* defines the local *i*, which resides on *main()*'s sf and we can just output its value:

```
int main() {
        int i = 42;
        std::cout<<"The value of i is "<<i<<std::endl;
}
```

main()'s sf

42	*i*	0x7ffeefbff26c

  - The new aspect is, that *i* is an lvalue, i.e. *i* has an address in memory, more specifically an address in the stack memory.

- The address of an lvalue can be retrieved with the &-operator, the address-operator. Let's apply & on *i*:

```
int main() {
        int i = 42;
        std::cout<<"The address of i is "<<&i<<std::endl;
}
```

Terminal

```
NicosMBP:src nico$ ./main
The address of i is 0x7ffeefbff26c
NicosMBP:src nico$
```

  - In the output we see, that *i*'s address is written to the console as hexadecimal number, we'll discuss this topic in a future lecture.
  - => At address 0x7ffeefbff26c in memory, we'll find *i*'s value 42.

- Literals are objects that have no address in memory, so they are no lvalues! Cstring literals are lvalues because of compatibility reasons.

Addresses and Pointers – Part II

- The way we used the address of *i* was just an expression, that wrote *i*'s address to the console:

```
int main() {
        int i = 42;
        std::cout<<"The address of i is "<<&i<<std::endl;
}
```

- Apart from this, we can also store the address of *i* in another variable, because an address is also a value!

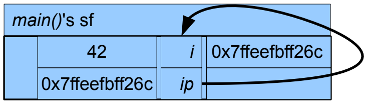
```
int main() {
        int i = 42;         // The addressed type is int.
        int* ip = &i;       // Take the address of i and store it into the int-pointer ip.
        std::cout<<"The address of i is "<<ip<<std::endl; // &-operator not used
}
```

  - As can be seen, we have just retrieved *i*'s address and stored it into another variable *ip* of type int*.

- *ip* is an int-pointer, syntactically the type is written as int*.

Definition
A variable that stores the address of an lvalue is called pointer.
Simplified:
A pointer is a variable that stores the address of another variable.

- Notice, that *ip* points to a piece of memory in the same sf where *i*'s value is stored:



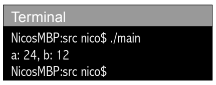
  - Notice also, that the value stored in *ip* is the address of *i*!

Passing Function Arguments by Reference

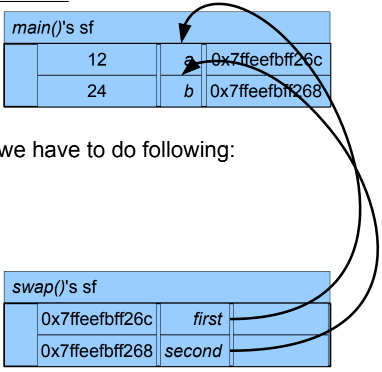
- An important application of pointers is the implementation of call by reference. – Now we can implement *swap()* correctly.

- The idea is that a pointer as parameter of a function holds the address of another variable.
  - Then the caller-function can pass the address of a variable of its stackframe to the callee-function:

```
int main() {
        int a = 12, b = 24;
        swap(&a, &b);
        std::cout<<"a: "<<a<<", b: "<<b<<std::endl;
}
```



```
Terminal
NicosMBP:src nico$ ./main
a: 24, b: 12
NicosMBP:src nico$
```



- The callee-function must now deal with addresses as arguments, e.g. within *swap()* we have to do following:
  - (1) Get the value at address *first* and use it to overwrite the value at address *second*.
  - (2) Get the value at address *second* and use it to overwrite the value at address *first*.
  - Notice, that we still have to use a temporary variable!

```
// Correct implementation of swap()!
void swap(int* first, int* second) {
        int temp = *first;
        *first = *second;
        *second = temp;
}
```

- To get the value, which is stored at the address stored in a pointer, we have to prefix the pointer-variable with a '*'.
  - When the *-operator is used as prefix of a pointer variable to get the "referenced" value we call it the dereferencing operator.

Pointers, Indirection and Dereferencing

- To make *swap()* work with call by reference, we used following features of C++ to span function stackframes:
  - We take the addresses of two variables in *main()*'s sf using the prefix &-operator, also called address-operator.
  - We pass these addresses to pointer params of *swap()*. So, *swap()* has access to addresses to variables of *main()*'s sf.
  - In *swap()* we use the prefix *-operator, the dereferencing-operator, on the pointer params to indirectly read/write the values in *main()*'s sf.

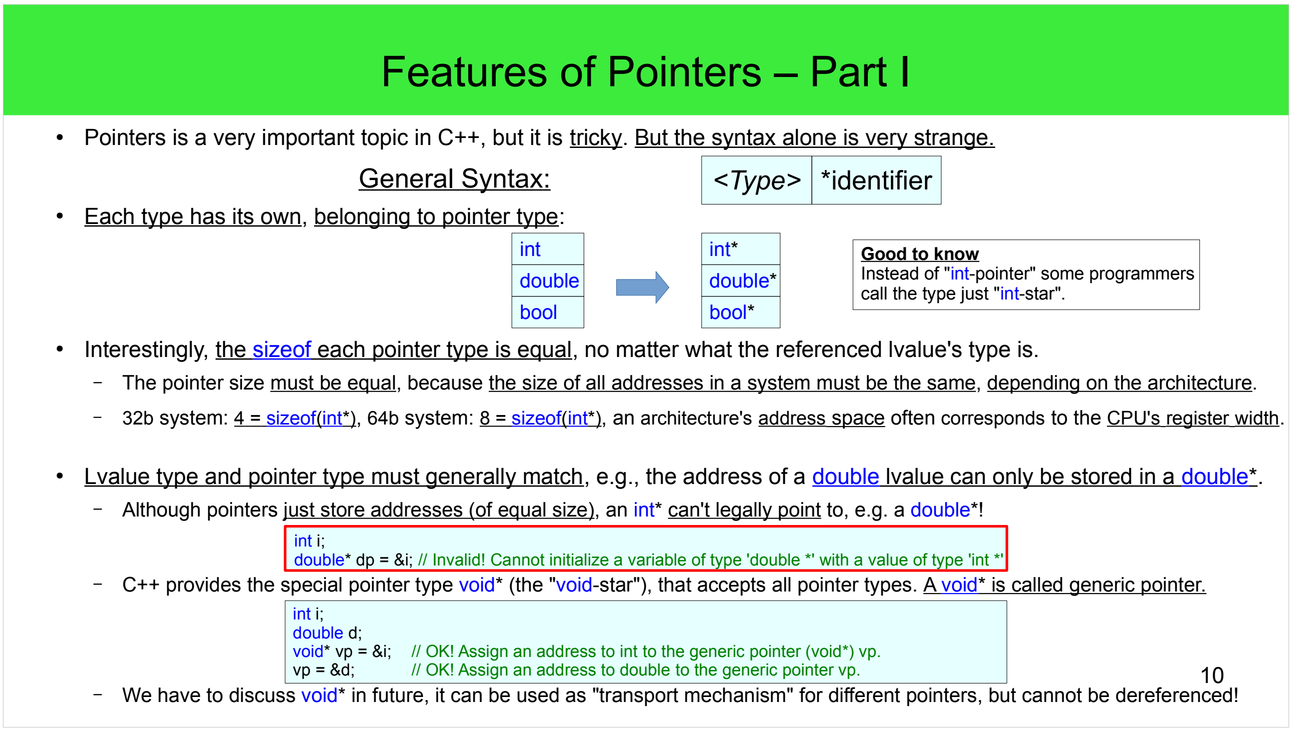
Notice
swap(int, int*)* doesn't really support call by reference! – Pointers will still be passed by value, but the value to which the pointer points to can be modified by dereferencing the pointer, which allows indirect access to the value.

- Using pointers for indirect access to lvalues:
  - Indirection: Pointers allow indirect access to referenced lvalues.
  - Dereferencing: The prefix *-operator (the dereference-operator) allows to read/write the referenced lvalue.

```
int i = 42;
std::cout<<"The value of i is: "<<i<<std::endl; // i's original value is 42.
int* ip = &i; // Take the address of i and store it into the int-pointer ip.
*ip = 300;    // Dereference the pointer and assign 300 to the original lvalue.
std::cout<<"The value of i is: "<<i<<std::endl; // Will print 300 as i's value!
```

- Pointers, indirection and dereferencing actually reflect real world scenarios:
  - (A) Indirection: blow up a ballon, and just pass around the cord.
  - (B) Dereferencing: means to coil up the ballon's cord, and catch the ballon.
  - (A) Indirection: tell a friend a URL to a site, instead of mailing a copy of the webpage.
  - (B) Dereferencing: loading the URL in a browser.
  - Semantic picture: Pointers are shortcuts to the objects they're pointing to!

Notice
In some examples, we are using the 'p'-prefix to highlight, that a variable is a pointer. This is no mandatory convention and will basically only be done in the introductory lectures.



Features of Pointers – Part I

- Pointers is a very important topic in C++, but it is tricky. But the syntax alone is very strange.

General Syntax: <*Type*> *identifier

- Each type has its own, belonging to pointer type:

int	int*
double	double*
bool	bool*

Good to know
Instead of "int-pointer" some programmers call the type just "int-star".

- Interestingly, the sizeof each pointer type is equal, no matter what the referenced lvalue's type is.
  - The pointer size must be equal, because the size of all addresses in a system must be the same, depending on the architecture.
  - 32b system: 4 = sizeof(int*), 64b system: 8 = sizeof(int*), an architecture's address space often corresponds to the CPU's register width.
- Lvalue type and pointer type must generally match, e.g., the address of a double lvalue can only be stored in a double*.
  - Although pointers just store addresses (of equal size), an int* can't legally point to, e.g. a double*!

```
int i;
double* dp = &i; // Invalid! Cannot initialize a variable of type 'double *' with a value of type 'int *'
```

  - C++ provides the special pointer type void* (the "void-star"), that accepts all pointer types. A void* is called generic pointer.

```
int i;
double d;
void* vp = &i;    // OK! Assign an address to int to the generic pointer (void*) vp.
vp = &d;          // OK! Assign an address to double to the generic pointer vp.
```

  - We have to discuss void* in future, it can be used as "transport mechanism" for different pointers, but cannot be dereferenced!

- In which "unit" does the sizeof operator return its result?
  - In *std::size_t*, a *std::size_t* of value 1 represents the sizeof(char). The type *std::size_t* is defined in <cstddef>.
- We can assign pointers of any pointer type to a void* without casting, this is why it is called "generic" pointer type. Exceptions: function pointers that need to be converted to void* with a reinterpret_cast and const pointers that need to be converted with a const_cast.

Features of Pointers – Part II

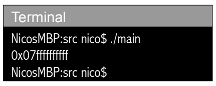
- The syntactic idiosyncrasy of pointers shows esp., when we define multiple pointers of the same type in one statement:

```
int *ip2, *ip3, *ip4; // Pointer declarators in a declaration list.
```

  - The *-symbol seems not to belong to the type, but to the identifier! More exactly, the *-symbol is part of the pointer-declarator.

- The value of an uninitialized pointer is undefined:

```
int* ip;
std::cout<<ip<<std::endl; // The value of ip, i.e. the address stored ip is undefined.
```



```
Terminal
NicosMBP:src nico$ ./main
0x07ffffffffff
NicosMBP:src nico$
```

  - Dereferencing of uninitialized pointers leads to undefined behavior:

```
int* ip;
std::cout<<*ip<<std::endl; // Dereferencing an uninitialized pointer is undefined.
```

- To explicitly mark a pointer as "currently not used" we can let it "point to nothing", which can be better than letting it uninitialized.
  - To do this, we can initialize or assign a pointer with a null-value. – We can use the values nullptr or 0 or *NULL* (<cstddef>).
  - Dereferencing null-pointers is also undefined:

```
int* ip = nullptr;
std::cout<<*ip<<std::endl; // Dereferencing a null-pointer is undefined.
```

  - To avoid this problem we can check a potential null-pointer for nullptr before dereferencing:

```
if (ip != nullptr) { // Great! Check for null-pointer.
	std::cout<<*ip<<std::endl;
}
```

  - There is an implicit conversion from all pointer-types to bool. null-pointers evaluate to false, all other pointers evaluate to true:

```
if (ip) { // Implicit conversion from pointer-type to bool.
	std::cout<<*ip<<std::endl;
}
```

Features of Pointers – Part III

- Because pointers are just addresses as values bound to variables, they are itself lvalues with an address.
  - This leads to the fact, that we can also have pointers to pointers! A pointer to pointer just uses '**' in its declarator:

```
int i = 42;
int* ip = &i;
int** ipp = &ip; // Taking the address of a int-pointer, which is stored in an int-pointer-pointer.
std::cout<<ip<<std::endl;
```

General Syntax:

<*Type*>	**identifier

  - Potentially, we could also have further pointer indirection levels, but the two-level pointer to pointer is often the maximum.
  - However, we cannot directly get the address of an address! The explanation is simple: it is not an lvalue!

```
ipp = &(&i); // Invalid! Cannot take the address of a non lvalue of type 'int *'
```

- It is possible to define function pointers, i.e. pointers to functions.
  - We'll discuss this topic in a future lecture. It is not a complicated, but a rather advanced topic.
  - Having pointers to functions, we can pass such a function pointer to yet another function as argument!
  - The declarators of function pointers are looking really weird:

```
double sum(double a, double b) {
   return a + b;
}
double (*fp)(double, double);    // Declare the function pointer named fp of type double(*)(double, double).
fp = sum;                        // When sum() is assigned to fp, sum() will decay to its pointer type.
double result = fp(.3, 4);       // Dereference fp and call sum(), to which fp is pointing.
```

Features of Pointers – Part IV – Constness

- Pointers, being just variables holding the address of another variable, can also be defined as constant in C++:

```
int i = 42, j = 15;
int* const constip = i&;
```

  - Following the idea of a constant, after we have initialized the pointer constant, we cannot assign any other addresses or pointers:

```
int j = 15;
constip = j&;
```

- The syntax of the type "pointer constant" is a little weird on a first look, basically the const qualifier follows the '*'-symbol:

```
// An int-pointer constant:
int* const constip = i&;
```

VS

```
// An int constant:
const int constint = 200;
```

- The constness of a pointer refers only to the pointer itself, so we can modify the value, to which the pointer is pointing:

```
*constip = 383; // OK! Deference a const pointer and change the value at the referenced address.
```

- A pointer is a compound type (a term yet to discuss): the pointer and the referenced value can have a separate constness.
  - C++ allows to declare a pointer to be const on its own value (i.e. the contained address) and constness of the value it is pointing to:

```
// An int-pointer constant to a const int:
const int* const constpconsti = i&;
```

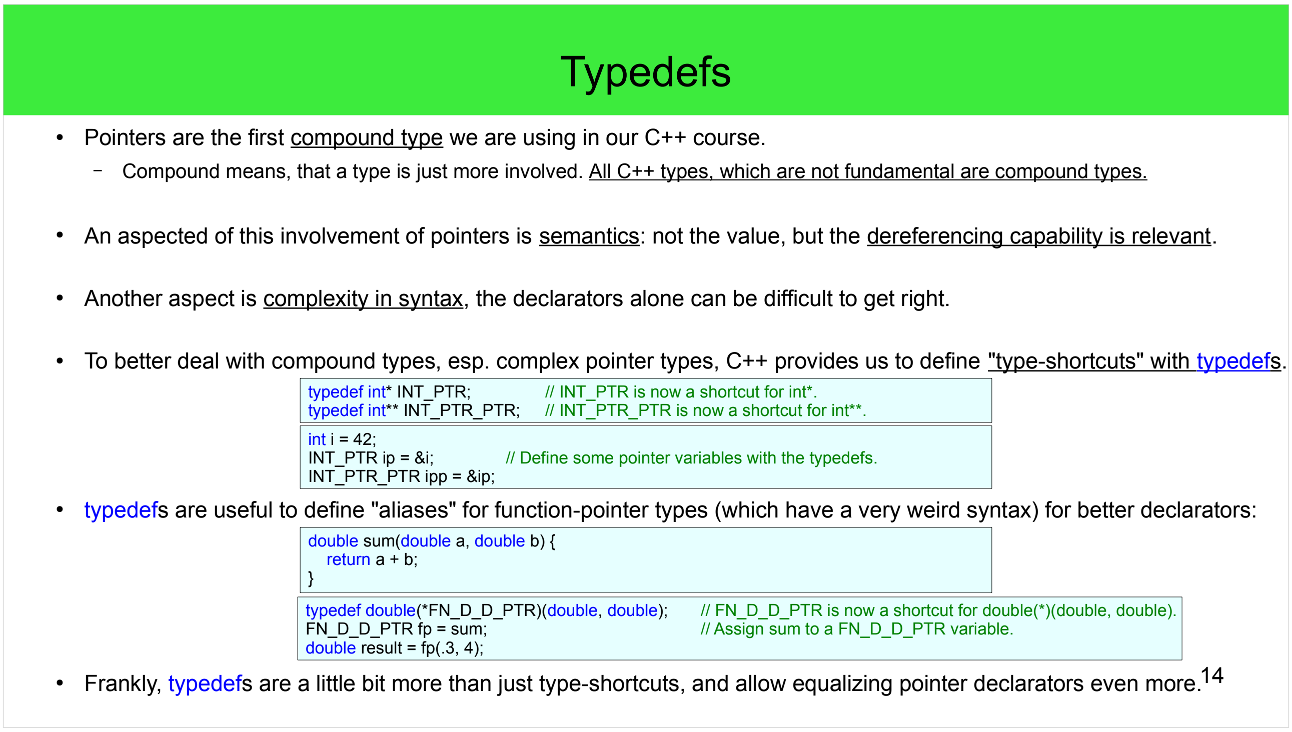
```
*constpconsti = 383; // Invalid! Read-only variable is not assignable
```

- It should be said, that the pointer-constant syntax is actually reasonable, i.e. writing the const qualifier after the type.
  - Mind, that C++ generally allows to put the const qualifier after the type, so we can also code this:

```
// An int-pointer constant to a const int:
const int* const constpconsti = i&;
```

```
// An int-pointer constant to a const int:
int const * const constpconsti = i&;
```

  - Reading this aloud puts the reason into the syntax: "an int constant referred to by a pointer constant"



Typedefs

- Pointers are the first compound type we are using in our C++ course.
  - Compound means, that a type is just more involved. All C++ types, which are not fundamental are compound types.
- An aspected of this involvement of pointers is semantics: not the value, but the dereferencing capability is relevant.
- Another aspect is complexity in syntax, the declarators alone can be difficult to get right.
- To better deal with compound types, esp. complex pointer types, C++ provides us to define "type-shortcuts" with typedefs.

```
typedef int* INT_PTR;              // INT_PTR is now a shortcut for int*.
typedef int** INT_PTR_PTR;     // INT_PTR_PTR is now a shortcut for int**.
```

```
int i = 42;
INT_PTR ip = &i;                // Define some pointer variables with the typedefs.
INT_PTR_PTR ipp = &ip;
```

- typedefs are useful to define "aliases" for function-pointer types (which have a very weird syntax) for better declarators:

```
double sum(double a, double b) {
    return a + b;
}
```

```
typedef double(*FN_D_D_PTR)(double, double);      // FN_D_D_PTR is now a shortcut for double(*)(double, double).
FN_D_D_PTR fp = sum;                                           // Assign sum to a FN_D_D_PTR variable.
double result = fp(.3, 4);
```

- Frankly, typedefs are a little bit more than just type-shortcuts, and allow equalizing pointer declarators even more.

- typedefs play an important role in the STL to hide the virtual type behind a typedef to allow library vendors/compiler builders to create individual solutions.
- Another kind of types, whose full names can be shortcut nicely with typedefs, are template instances. This idiom is also present in the STL.

Once again the Problem of Code Repetition

- Let's consider following code to read three numbers from the console and output their sum:

```
int promptAndReadNumber() {
	std::cout<<"Please enter a number:"<<std::endl;
	std::cout<<"The number should be greater than ten:"<<std::endl;
	int number;
	std::cin>>number;
	return number;
}
```

```
// Reading three numbers from the console:

int a, b, c;
a = promptAndReadNumber();
b = promptAndReadNumber();
c = promptAndReadNumber();
std::cout<<"The sum: "<<(a + b + c)<<"!"<<std::endl;
```

- But what to do, if we need to sum more than three numbers? A piece of cake! Just add another prompt and variable:

```
// Reading four numbers from the console:

int a, b, c, d;
a = promptAndReadNumber();
b = promptAndReadNumber();
c = promptAndReadNumber();
d = promptAndReadNumber();
std::cout<<"The sum: "<<(a + b + c + d)<<"!"<<std::endl;
```

- … and, what to do if want to sum ten numbers? Add six more prompts and six more variables? Hm?
  - We already heard about the principle of DRY! What if we apply a loop to solve this problem?
  - Let's do that, it should solve our problem.

Reducing Code Repetition with Arrays – Part I

- All right, give loops a chance, we'll use a for loop! But … it doesn't compute, we cannot formulate the required code:

```
// Reading four numbers from the console:

int a, b, c, d;
for (int i = 0; i < 4; ++i) {
    a??? Huh? = promptAndReadNumber();
}
```

- The loop allows to formulate the repeating prompt, but we cannot assign to the four variables to sum up!

- The basic problem are the variables, more exactly, the need to assign four individual values.

- We can solve this problem by using a variable, which can store multiple values at once, by keeping a list of variables.

- In C++, variables holding multiple values are called arrays. Let's rewrite the code reading four values from the console:

```
// Reading four numbers from the console:

int a, b, c, d;
a = promptAndReadNumber();
b = promptAndReadNumber();
c = promptAndReadNumber();
d = promptAndReadNumber();
std::cout<<"The sum: "<<(a + b + c + d)<<"!"<<std::endl;
```

```
// Reading four numbers from the console using an array:

int numbers[4];
numbers[0] = promptAndReadNumber();
numbers[1] = promptAndReadNumber();
numbers[2] = promptAndReadNumber();
numbers[3] = promptAndReadNumber();
std::cout<<"The sum: "<<(numbers[0] + numbers[1] + numbers[2] + numbers[3])<<"!"<<std::endl;
```

Reducing Code Repetition with Arrays – Part II

- Let's review the example using the array:

```
// Reading four numbers from the console using an array:

int numbers[4];
numbers[0] = promptAndReadNumber();
numbers[1] = promptAndReadNumber();
numbers[2] = promptAndReadNumber();
numbers[3] = promptAndReadNumber();
std::cout<<"The sum: "<<(numbers[0] + numbers[1] + numbers[2] + numbers[3])<<"!"<<std::endl;
```

  - The new aspect in this code is, that it stores values in the array *numbers* and not in individual variables (e.g. *a*, *b*, *c* and *d*).

- With arrays, we can regard the DRY principle: we use a for loop to read multiple values from the console and summing them up.

```
// Reading four numbers from the console and sum them up with a loop:

int numbers[4];
int sum = 0;
for (int i = 0; i < 4; ++i) {
	numbers[i] = promptAndReadNumber();
	sum += numbers[i];
}
std::cout<<"The sum: "<<sum<<"!"<<std::endl;
```

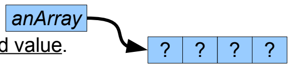
  - Just the value 4 (which is used twice here) controls how many numbers are asked from the user and summed up.

- Of course, this information about arrays is really overwhelming, so let us discuss the details about arrays now.

Introduction to Arrays – Part I

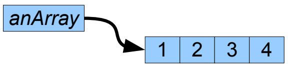
- What is an array?
  - In brief: arrays are like lists of values, an array is a kind of "container": it stores a bucket of values.
  - An array variable represents multiple variables kept under one symbol held in one object in memory!
- Let's start having a first glimpse on the definition of an array variable:

```
int anArray[4];
```



  - This statement creates the array, *anArray* of 4 elements and each element has an undefined value.
  - The count of elements in the array is specified in the brackets.
  - => *anArray* actually represents 4 variables, which are just called elements, which all have an undefined value.
- Because *anArray* has only uninitialized values, we should give the elements some values:

```
for (int i = 0; i < 4; ++i) {
        anArray[i] = i + 1;
}
```



  - This time we use the []-syntax as an operator to write values into each individual element in *anArray* via their indexes.
  - We use a loop to generate index numbers to access each element in *anArray* exactly. for loops are excellent to work with arrays.
  - Notice, that the indexes are incremented from 0 to 3 (*i* < 4), because counting up the indexes starts at index 0 (not 1).
  - The individual 4 variables aggregated in *anArray* are accessible as *anArray[0]*, *anArray[1]*, *anArray[2]* and *anArray[3]*.
  - After the loop is done the elements have these values: *anArray[0]* = 1, *anArray[1]* = 2, *anArray[2]* = 3 and *anArray[3]* = 4.

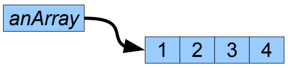
Introduction to Arrays – Part II

```
int anArray[4];
```

- The length of an array can be specified, when the array is created. The length is of type int.

- The length of an array is of type int. => Arrays are datatypes, which need another data of type int: the length.

```
for (int i = 0; i < 4; ++i) {
        anArray[i] = i + 1;
}
```

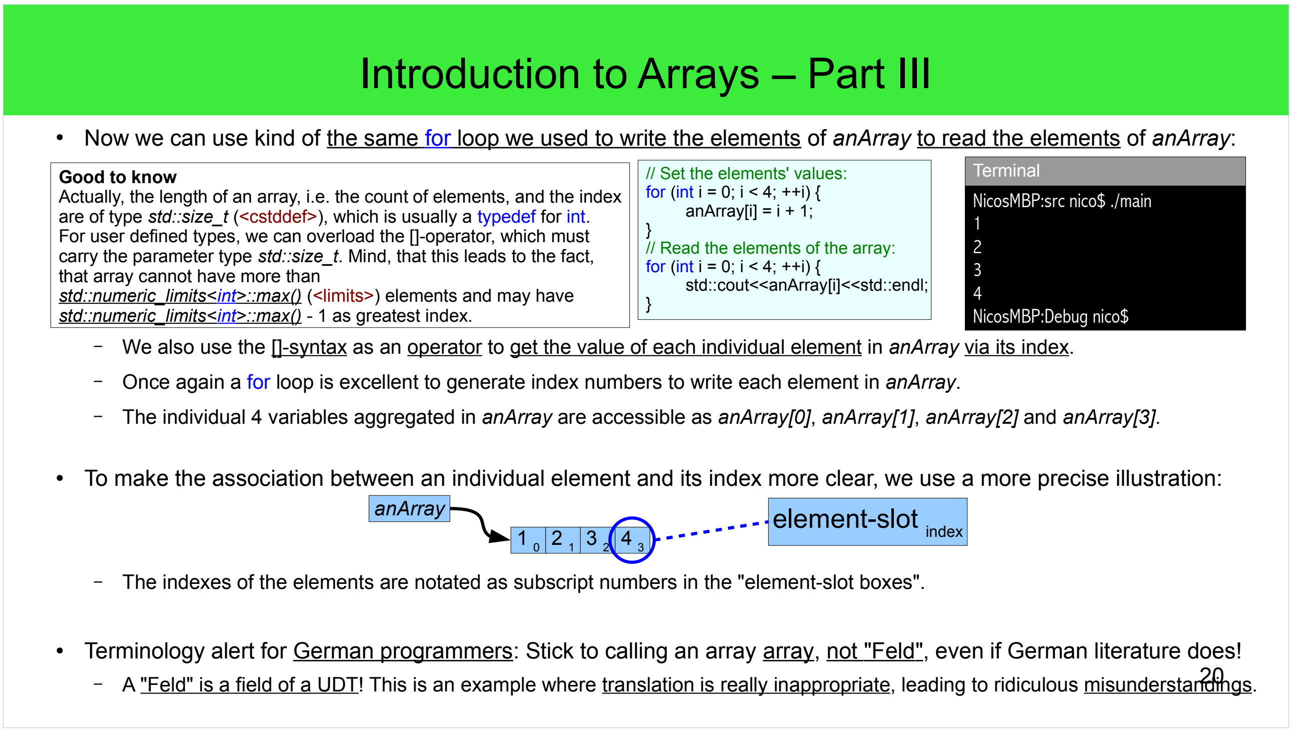


- Because an array is like a list, we need two sorts of data to use it: the array object and the position in the array.

- The position in an array is also of type int. => Arrays are datatypes, which use another data of type int: the position.
  - We use the counter variable *i* as "position-int" for the []-operator to access each individual element on its position *i* in the array.
  - The value we use as "position-int" for an array is called index. The counter variable *i* is named as *i* for index.

- After we discussed pointers, arrays are another compound type we have to understand.

- Each element can be modified/assigned to with the []-operator and an index just by using assignment operations.
  - The []-operator is usually called index-operator, element-access-operator or array-subscript-operator.



Introduction to Arrays – Part III

- Now we can use kind of the same for loop we used to write the elements of *anArray* to read the elements of *anArray*:

Good to know
Actually, the length of an array, i.e. the count of elements, and the index are of type *std::size_t* (<cstddef>), which is usually a typedef for int. For user defined types, we can overload the []-operator, which must carry the parameter type *std::size_t*. Mind, that this leads to the fact, that array cannot have more than *std::numeric_limits<int>::max()* (<limits>) elements and may have *std::numeric_limits<int>::max()* - 1 as greatest index.

```
// Set the elements' values:
for (int i = 0; i < 4; ++i) {
        anArray[i] = i + 1;
}
// Read the elements of the array:
for (int i = 0; i < 4; ++i) {
        std::cout<<anArray[i]<<std::endl;
}
```

Terminal

```
NicosMBP:src nico$ ./main
1
2
3
4
NicosMBP:Debug nico$
```

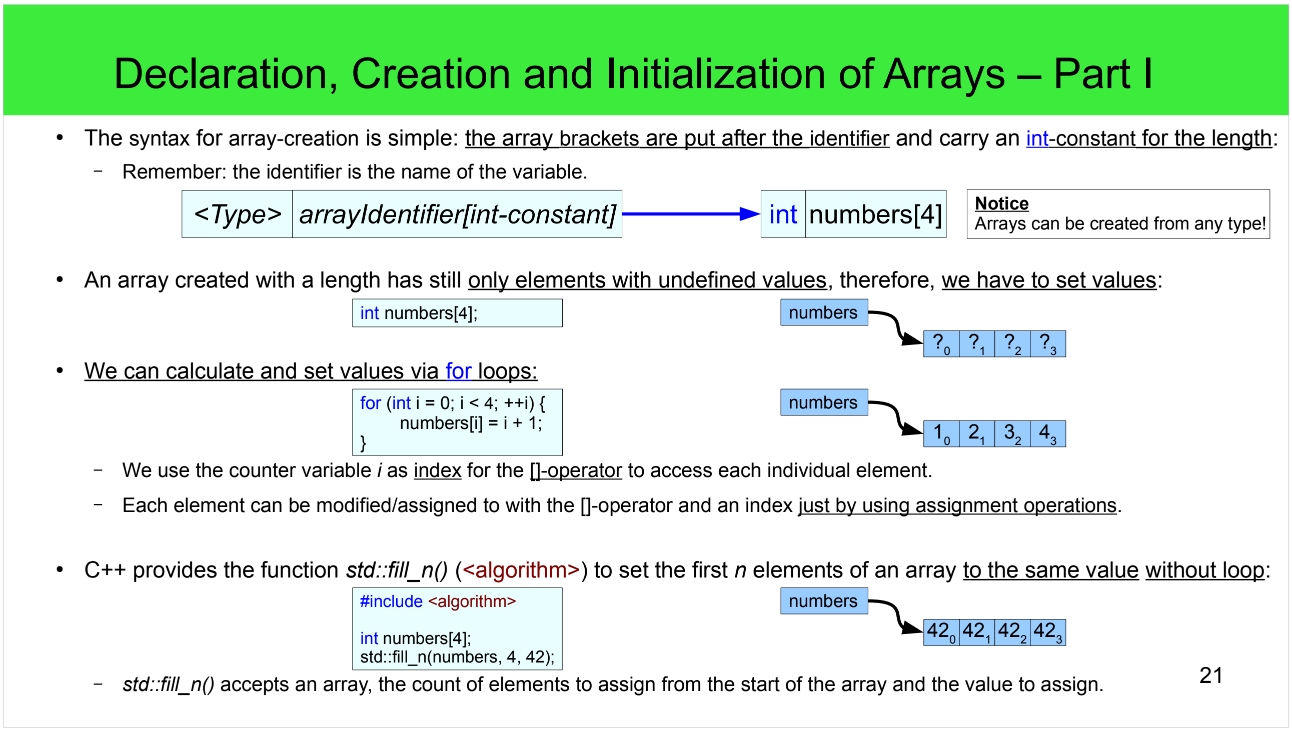
  - We also use the []-syntax as an operator to get the value of each individual element in *anArray* via its index.
  - Once again a for loop is excellent to generate index numbers to write each element in *anArray*.
  - The individual 4 variables aggregated in *anArray* are accessible as *anArray[0]*, *anArray[1]*, *anArray[2]* and *anArray[3]*.

- To make the association between an individual element and its index more clear, we use a more precise illustration:

  - The indexes of the elements are notated as subscript numbers in the "element-slot boxes".

- Terminology alert for German programmers: Stick to calling an array array, not "Feld", even if German literature does!
  - A "Feld" is a field of a UDT! This is an example where translation is really inappropriate, leading to ridiculous misunderstandings.

- If a "data container" should store more than *std::numeric_limits<int>::max()* elements, another type must be used, which uses keys of a type different from int. – Such "data containers" are called associative containers (e.g. *std::map*) in C++, they can potentially solve this limitation.



Declaration, Creation and Initialization of Arrays – Part I

- The syntax for array-creation is simple: the array brackets are put after the identifier and carry an int-constant for the length:
  - Remember: the identifier is the name of the variable.

<Type> *arrayIdentifier[int-constant]* → int numbers[4]

Notice
Arrays can be created from any type!

- An array created with a length has still only elements with undefined values, therefore, we have to set values:

```
int numbers[4];
```

numbers → $?_0$ $?_1$ $?_2$ $?_3$

- We can calculate and set values via for loops:

```
for (int i = 0; i < 4; ++i) {
        numbers[i] = i + 1;
}
```

numbers → 1_0 2_1 3_2 4_3

  - We use the counter variable *i* as index for the []-operator to access each individual element.
  - Each element can be modified/assigned to with the []-operator and an index just by using assignment operations.

- C++ provides the function *std::fill_n()* (<algorithm>) to set the first *n* elements of an array to the same value without loop:

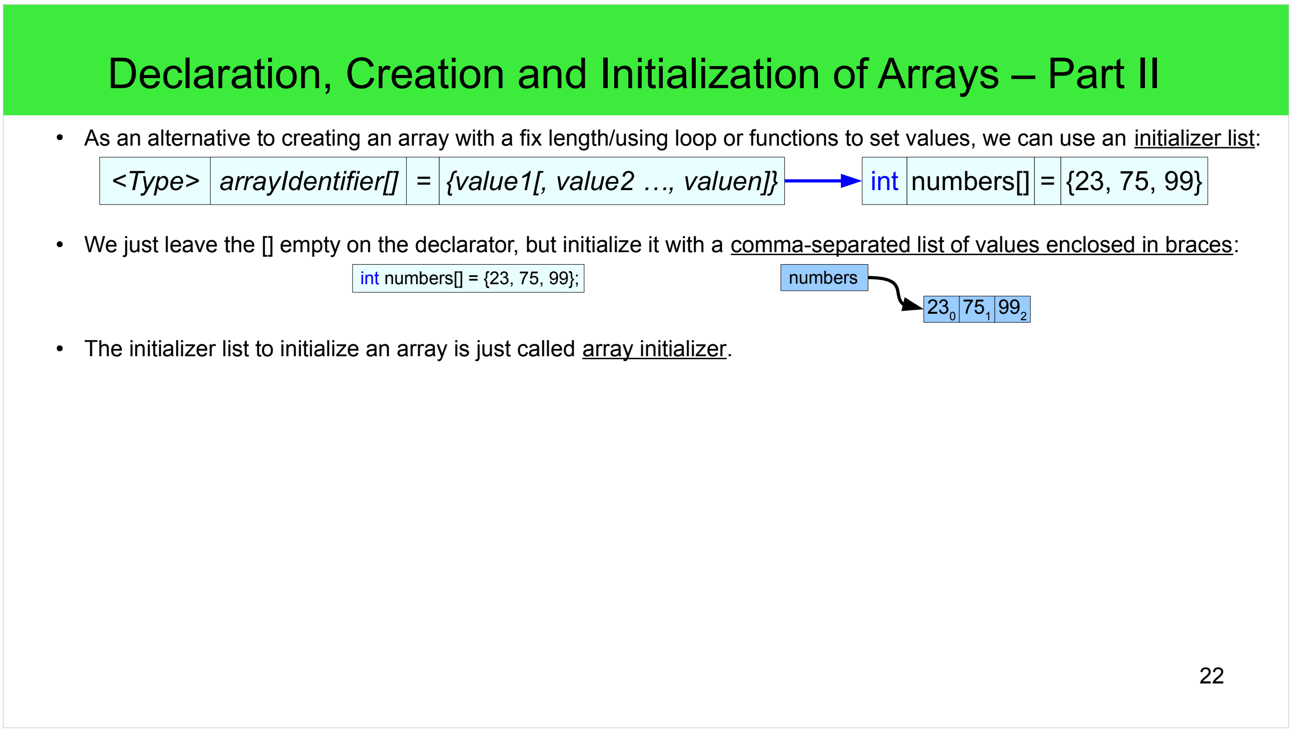
```
#include <algorithm>

int numbers[4];
std::fill_n(numbers, 4, 42);
```

numbers → 42_0 42_1 42_2 42_3

  - *std::fill_n()* accepts an array, the count of elements to assign from the start of the array and the value to assign.

- Mind that we can only create (dimension (verb)) automatic arrays of const length with our current knowledge. In the next lecture we learn how to handle arrays of dynamic size.
  - It should be mentioned that C99 does support variable length arrays (VLAs) on the stack.



Declaration, Creation and Initialization of Arrays – Part II

- As an alternative to creating an array with a fix length/using loop or functions to set values, we can use an initializer list:

<Type>	*arrayIdentifier[]*	=	*{value1[, value2 …, valuen]}*	→	int	numbers[]	=	{23, 75, 99}

- We just leave the [] empty on the declarator, but initialize it with a comma-separated list of values enclosed in braces:

```
int numbers[] = {23, 75, 99};
```

- The initializer list to initialize an array is just called array initializer.

- Mind that we can only create (dimension (verb)) automatic arrays of const length with our current knowledge. In the next lecture we learn how to handle arrays of dynamic size.
  - It should be mentioned that C99 does support variable length arrays (VLAs) on the stack.

Accessing and Modifying Array Elements

- Accessing and modification of array elements is done with the []-operator, we have already used it in many loops:

```
int numbers[4];
for (int i = 0; i < 4; ++i) {
        numbers[i] = i + 1;                     // modify/write value from numbers at index i
        std::cout<<numbers[i]<<std::endl;  // access/read value from numbers at index i
}
```

  - Each element is addressed via its index in the array. Syntactically, the index is the argument, which is passed to the []-operator.
  - Because the array's length is of type int, the index must also be of type int.

- The array-indexes are 0-based. – So the indexes' range is [0, length[. The first valid index is 0 and the last is length - 1!
  - This is why indexes are incremented starting from 0 upwards to *i* < length in for loops.

```
for (int i = 0; i < 4; ++i) {
        std::cout<<numbers[i]<<std::endl;
}
```

Good to know:
Most programming languages or "computer-oriented" notations use 0-based indexes.
Notable exceptions are early BASIC-dialects and Cascading Style Sheets (CSS).

- After an array was created and its elements filled with values, we can rewrite the values of those elements at any time.

```
// rewrite all elements in numbers2:
int numbers2[] = {232, 6789, 3};
for (int i = 0; i < 3; ++i) {
        numbers2[i] = i * i;
}
```

  - The notable fact here: array elements are not in any kind "read only".

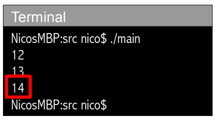
Array Length – Part I

- Let's review this example:

```
int numbers[] = {12, 13, 14};
for (int i = 0; i < 3; ++i) {
        std::cout<<numbers[i]<<std::endl;
}
```

  - Sure, this code will write the three ints 12, 13 and 14 to the console. In the next step, we append the int 15 to the initializer list:

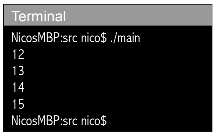
```
int numbers[] = {12, 13, 14, 15};
for (int i = 0; i < 3; ++i) {
        std::cout<<numbers[i]<<std::endl;
}
```



```
NicosMBP:src nico$ ./main
12
13
14
NicosMBP:src nico$
```

  - But only three ints 12, 13 and 14 are written to console! – Right, we forgot to tell the loop to iterate the 4th element (3rd index)!

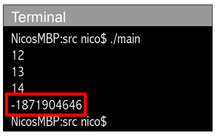
```
int numbers[] = {12, 13, 14, 15};
for (int i = 0; i < 4; ++i) {
        std::cout<<numbers[i]<<std::endl;
}
```



```
NicosMBP:src nico$ ./main
12
13
14
15
NicosMBP:src nico$
```

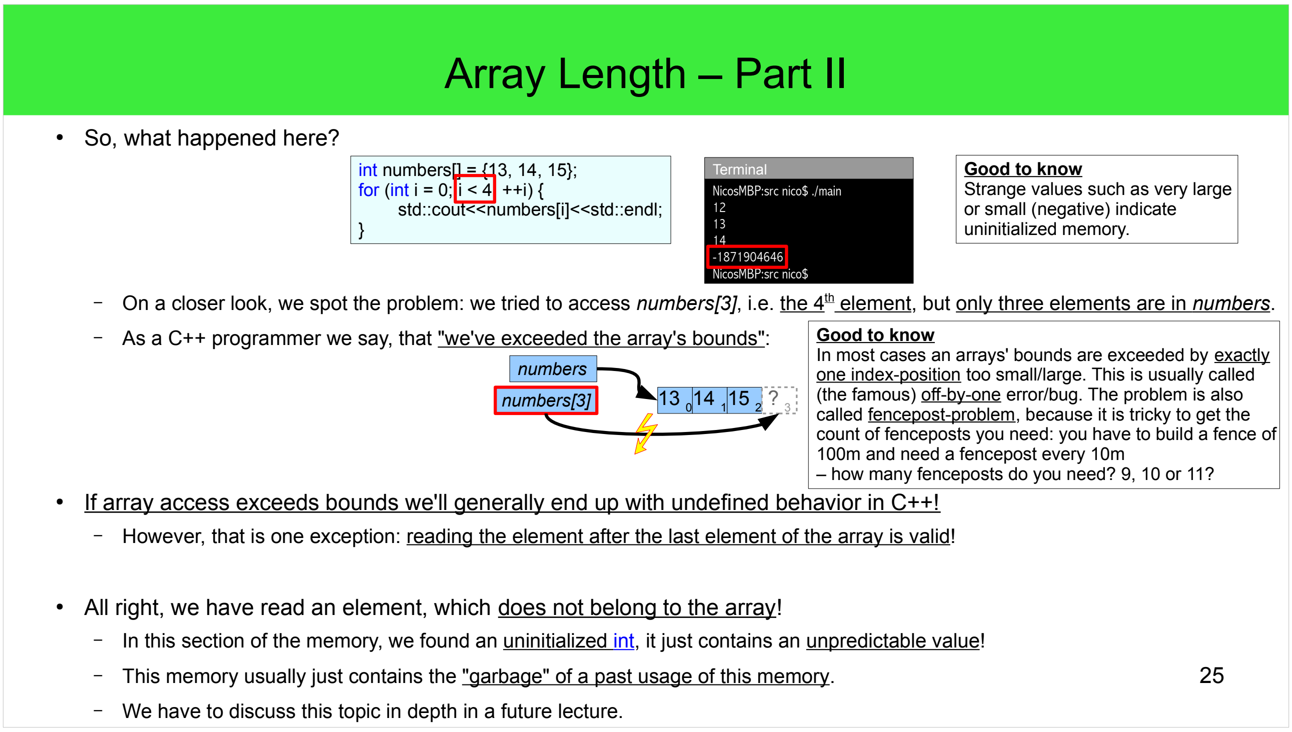
  - Now we will remove the int 12 from *numbers* and write its elements to console:

```
int numbers[] = {13, 14, 15};
for (int i = 0; i < 4; ++i) {
        std::cout<<numbers[i]<<std::endl;
}
```



```
NicosMBP:src nico$ ./main
12
13
14
-1871904646
NicosMBP:src nico$
```

  - What happened now? – A run time error appeared!



Array Length – Part II

- So, what happened here?

```
int numbers[] = {13, 14, 15};
for (int i = 0; i < 4; ++i) {
        std::cout<<numbers[i]<<std::endl;
}
```

Terminal

```
NicosMBP:src nico$ ./main
12
13
14
-1871904646
NicosMBP:src nico$
```

<u>Good to know</u>
Strange values such as very large or small (negative) indicate uninitialized memory.

  - On a closer look, we spot the problem: we tried to access *numbers[3]*, i.e. <u>the 4th element</u>, but <u>only three elements are in *numbers*</u>.
  - As a C++ programmer we say, that <u>"we've exceeded the array's bounds"</u>:

<u>Good to know</u>
In most cases an arrays' bounds are exceeded by <u>exactly one index-position</u> too small/large. This is usually called (the famous) <u>off-by-one</u> error/bug. The problem is also called <u>fencepost-problem</u>, because it is tricky to get the count of fenceposts you need: you have to build a fence of 100m and need a fencepost every 10m – how many fenceposts do you need? 9, 10 or 11?

- <u>If array access exceeds bounds we'll generally end up with undefined behavior in C++!</u>
  - However, that is one exception: <u>reading the element after the last element of the array is valid</u>!

- All right, we have read an element, which <u>does not belong to the array</u>!
  - In this section of the memory, we found an <u>uninitialized int</u>, it just contains an <u>unpredictable value</u>!
  - This memory usually just contains the <u>"garbage" of a past usage of this memory</u>.
  - We have to discuss this topic in depth in a future lecture.

- It should be said that bounds checked arrays are a milestone in stable programming compared to C/C++, where, e.g. modification of regions exceeding an arrays bounds is undefined. Where we had to debug for hours in C/C++ only to find the bug in the code, spotting and correcting such an error in Java is a piece of cake due to exceptions!
  - On the other hand, with the introduction of Java exceeding the bounds of an array was so serious, that Java ends program execution, if such an exception is thrown.

Array Length – Part III

- **Accessing invalid memory is the mayor gate for security leaks in software!**
  - C++ operates very near the memory, array excess is extremely (!) dangerous and usually leads to a disaster.

- All right, the question is, what can we do about this? Answer: we've to avoid array excess by defensive programming!

- But, what was the problem exactly? What do we have to get right?
  - The problem is, that the array variable and the bounds used for the iteration are separately managed in our code.
  - If data is managed separately, which has indeed a logical connection, things will fail, when only a single piece of data is changed.
  - E.g. we have removed one element from *numbers*, but we kept the iteration from 0 to the old count of elements: 4 (3rd index):

```
int numbers[] = {13, 14, 15};
for (int i = 0; i < 4; ++i) {
        std::cout<<numbers[i]<<std::endl;
}
```

- The bounds of array access are depending on the array length. How can we get the length of an array?

- The bad news is, that C++' arrays, do not know and do not expose their length!

Array Length – Part IV

- In C++ the inability of getting or tracking the length of an arrays is a very big disadvantage!

- When we program with arrays in C++, we have care ourselves for using arrays regarding their length!

- Let's discuss the array problematic on our loop example: A first step is to give the "magic" int 4 a better name:

```
int numbers[] = {13, 14, 15};
const int numbersLength = 4;
for (int i = 0; i < numbersLength; ++i) {
      std::cout<<numbers[i]<<std::endl;
}
```

Good to know
A magic number is a literal value in the source code, of which we do not understand the meaning, unless we analyze the complete code section. Therefor, we should avoid magic numbers and instead defined variables or constants with speaking names.

  - The bug is still in the code! – We have not stabilized the loop!
  - With the better name *numbersLength* we could quickly spot the discrepancy to the count of elements in *numbers*.
  - Because we know that an array's bounds are [0, *numbersLength*[, we have at least a chance to spot the problem!

- Rule: As soon as we know the length of an array, we should introduce a const, that stores this value!
  - The definition of this const should be very close to the definition of the array!

Array Length – more Examples

- Because handling of C++ arrays is so cumbersome, here some handy examples.

- Real life case: getting the first and last element of an array:

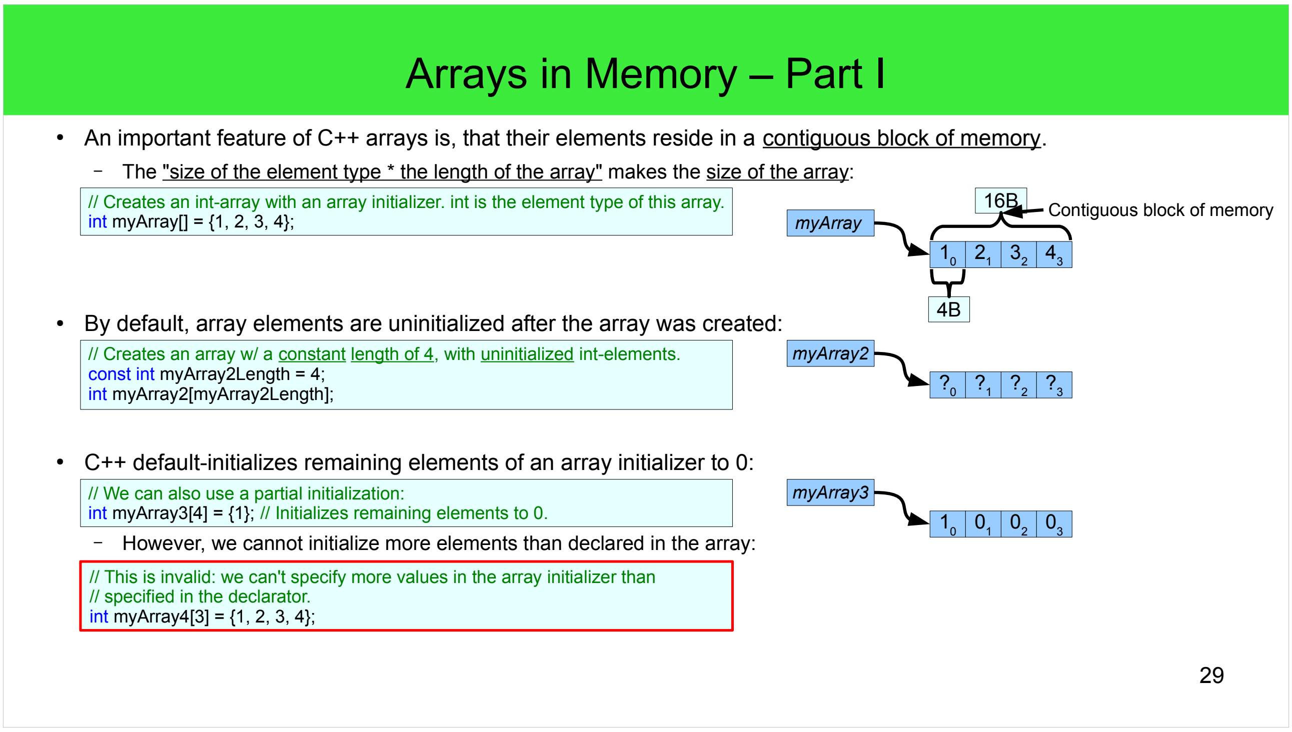
```
int numbers[] = {1, 2, 3, 4};
const int numbersLength = 4;
int firstElementsValue = numbers[0];
int lastElementsValue = numbers[numbersLength – 1];
```

- Real life case: regarding the length of passed array.
  - Like any other type, arrays can be a parameter type in functions.
  - With array parameters we've exactly a case, in which we don't know the length of arrays in advance:

```
void printArray(int numbers[], int nNumbers) {
	for (int i = 0; i < nNumbers; ++i) {
		std::cout<<numbers[i]<<std::endl;
	}
}
```

Good to know
A handy naming convention for constants just holding the length of a belonging to array is to use the name of the array and prefix it with 'n' for number (regarding camelCase).
So, the length of numbers is kept in *nNumbers*.

- Real life case: length of an array returned from a function.
  - Alas, it makes no sense to discuss returning arrays right now, we'll deal only with arrays and its length known on the caller side.
  - We have to revisit this topic, when we talk about dynamic arrays.
  - => The arrays we discuss in this lecture, so called automatic arrays, cannot be returned from functions!



Arrays in Memory – Part I

- An important feature of C++ arrays is, that their elements reside in a contiguous block of memory.
  - The "size of the element type * the length of the array" makes the size of the array:

```
// Creates an int-array with an array initializer. int is the element type of this array.
int myArray[] = {1, 2, 3, 4};
```

- By default, array elements are uninitialized after the array was created:

```
// Creates an array w/ a constant length of 4, with uninitialized int-elements.
const int myArray2Length = 4;
int myArray2[myArray2Length];
```

- C++ default-initializes remaining elements of an array initializer to 0:

```
// We can also use a partial initialization:
int myArray3[4] = {1}; // Initializes remaining elements to 0.
```

  - However, we cannot initialize more elements than declared in the array:

```
// This is invalid: we can't specify more values in the array initializer than
// specified in the declarator.
int myArray4[3] = {1, 2, 3, 4};
```

- Array, pointer and const types of other types need not to be declared ahead like other UDTs. As compound types they are declared by their usage with the []- and *-declarators, respectively.

Arrays in Memory – Part II

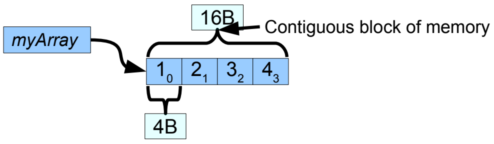
- The size of an array is defined as size of element type * length, so we can re-arrange the formula to get the array-length:

$$arraySize = elementSize \cdot arrayLength$$

$$\underline{arrayLength} = \frac{arraySize}{elementSize}$$

- Sure, we can program this in C++:

```
int myArray[] = {1, 2, 3, 4};
// getting the count of elements in myArray:
int nElements = sizeof(myArray)/sizeof(int);
// nElements = 4
```



- But, this solves our problem! Now we can get the count of elements in a C++ array! – Yes, but only in rare cases.
  - In other words: in most cases we cannot use this formula!
  - It works in this case, because the C++ compiler "sees": "*myArray* is an array and it was initialized with 4 elements".

- The calculation of an array's length via sizeof only works, if the compiler sees an array declaration.

Arrays in Memory – Part III

- Interestingly, we cannot "directly" assign arrays!

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

```
int myArray2[4];
myArray2 = myArray; // Array type 'int [4]' is not assignable
```

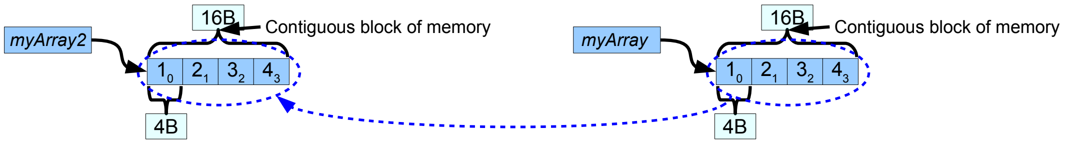
  - In C++ the semantics of assignment via '=' is copying.
  - Although in this example the compiler "sees" the array lengths and the elements' size, it won't perform a copy operation on '='!

- To "assign" arrays, we must copy them in memory ourselves! We can do this with *std::memcpy()* (<cstring>):

```
std::memcpy(myArray2, myArray, sizeof(int) * 4);
```

  - *std::memcpy()* accepts the destination array, the source array and the count of bytes to copy from source to destination.
  - And we want to copy all 4 elements, which consume sizeof(int) each, so we must copy sizeof(int) * 4 bytes.

- *std::memcpy()* works for us, because arrays are just continuous blocks in memory.
  - And this is what *std::memcpy()* does, it copies continuous blocks in memory of the specified size:



Array Decay

- Arrays and pointers have a special connection in C++: arrays can be represented with a pointer to their first element.
  - The pointer to the 1st element of an array is a pointer to the element type.
  - E.g. a pointer to the 1st element of an int array is of type int*.

- How to establish an array's pointer-representation? – It is simple: there is an implicit conversion from an array to "its" pointer:

```
int myArray[] = {1, 2, 3, 4};
int* pointerTo1stElement = myArray; // Implicit conversion from array to pointer
```

  - The implicit conversion from an array to a pointer has an appropriate name, it is called decay.

- Interestingly, an array's pointer can exactly be used like the declared array symbol:

```
for (int i = 0; i < 4; ++i) {
        std::cout<<pointerTo1stElement[i]<<std::endl;
}
```

- Pointer decay is the key to understand how arrays are passed to functions and how pointer arithmetics works.
  - Now we'll discuss how array arguments work with functions.
  - The discussion of pointer arithmetics follows in a future lecture, because we need more background knowledge.

Passing Arrays to Functions

- In the lecture about procedural programming we learned, that all args are passed to their parameters by value.
  - Remember, this means the arguments are copied when a function is called.

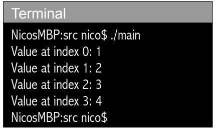
- When we pass an array to a function, will it also be copied? It could be very costly to copy all elements of an array!

- The good news: this is not happening in C++! Arrays are not getting copied, when passed to a function!

```
int myArray[] = {1, 2, 3, 4};
const int nMyArray = 4;
printArray(myArray, nMyArray); // myArray is decayed to "its" pointer
```

Good to know
The signature *printArray(int*, int)* is identically to *printArray(int[], int)*. This means, that both signatures don't overload! Using a pointer-type parameter to accept an array is the "syntactic tradition" in C++.

```
// printArray() outputs the first length elements of the content of array.
void printArray(int* array, int length) {
        for (int i = 0; i < length; ++i) {
                std::cout<<"Value at index "<<i<<": "<<array[i]<<std::endl;
        }
}
```


Terminal

```
NicosMBP:src nico$ ./main
Value at index 0: 1
Value at index 1: 2
Value at index 2: 3
Value at index 3: 4
NicosMBP:src nico$
```

  - The bottom line is, that only a pointer to the first element of array is copied to a function, but not all elements!
  - This means, that only the address of the first element in memory is copied to the parameter.
  - Mind, that we still have call by value, but pointer decay leads to effectively only copying a pointer, not the full array!

- Avoiding copies of arrays during function calls by pointer decay is a key concept behind C++' high run time performance!

Why Arrays cannot be passed by Value

- Decay is universal! When defining arrays of different lengths, C++ assumes them being of different type:

```
#include <typeinfo>

int myArray[4];  // myArray is of type int[4]
int myArray2[3];// myArray2 is of type int[3]

std::cout<<std::boolalpha<<(typeid(myArray) == typeid(myArray2))<<std::endl;
// >false
```

Good to know
The operator typeid yields an object of type *std::type_info*, which contains meta data of a C++ type of an object (e.g. a variable). *std::type_info* itself is a compound type, i.e. not a fundamental type.

  - The operator typeid (<typeinfo>) yields the type info of the specified type, we can compare *std::type_info*s with the ==-operator.
  - If int[4] and int[3] would not be decayed to int*, we needed to provide a lot of overloads of *printArray()*:

```
void printArray(int array[4]) {
        f
        }
}
```

```
void printArray(int array[3]) {  // Invalid! Redefinition of 'printArray'
        f
        }
}
```

```
void printArray(int array[2]) { // Invalid! Redefinition of 'printArray'
        for (int i = 0; i < 4; ++i) {
                std::cout<<"Value at index "<<i<<": "<<array[i]<<std::endl;
        }
}
```

For the C++ compiler all overloads are identical to *printArray(int*)*! The error cases hurt the ODR.

- Arrays have no copy semantics! How should C++ pass an array by value? Pass by value has copy semantics!
  - We already know, that C++ doesn't define copy semantics for arrays!
  - C++ needed to use *std::memcpy()*! But how to use it? – C++ needed to know the length of the array/size of the memory to copy!

Const Pointers avoid Modification

- Using pointers as params opens a gate for a kind of potential trouble: a function could change the addressed memory!
  - Remember, this kind of modification is what we wanted to have, when we implemented *swap()*!
  - But, this is not what we want for *printArray()*! – It could potentially modify the source int[] via the passed pointer:

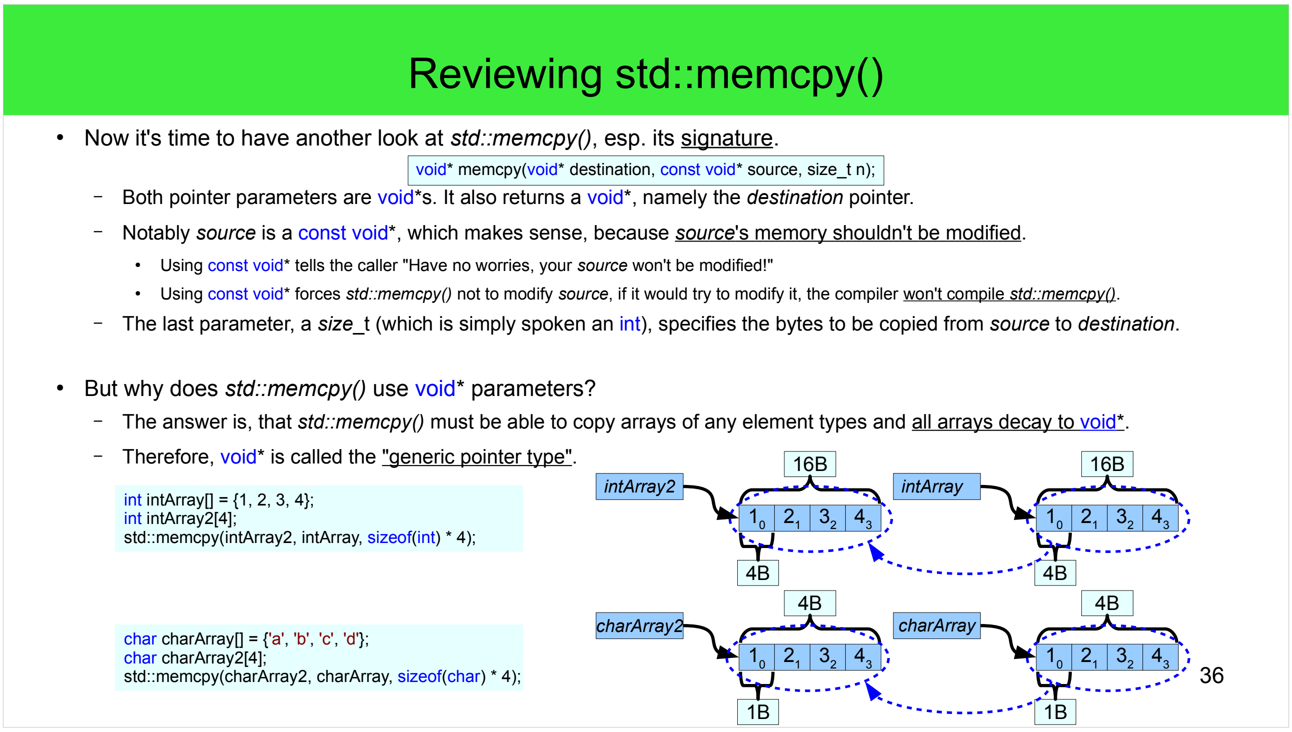
```
// printArray() prints the array to the console, but also sets all array elements to 0 as a side effect.
void printArray(int* array, int length) {
	for (int i = 0; i < length; ++i) {
		std::cout<<"Value at index "<<i<<": "<<array[i]<<std::endl;
		array[i] = 0; // Modifies the original array in the caller's sf!
	}
}
```

```
int myArray[] = {1, 2, 3, 4};
const int nMyArray = 4;
printArray(myArray, nMyArray);
// myArray = {0, 0, 0, 0} myArray was modified, all elements set to 0!
```

- Actually, we often have the need to just pass an address to avoid excessive copying, but don't want to modify the source!
  - To express this, e.g. in a function signature and to avoid accidental modification via the pointer, we define a const pointer param:

```
void printArray(const int* array, int length) {
	for (int i = 0; i < length; ++i) {
		std::cout<<"Value at index "<<i<<": "<<array[i]<<std::endl;
		array[i] = 0; // Invalid! Read-only variable is not assignable
	}
}
```

- Effectively, *array* is now a pointer, which doesn't allow modification of the value it is pointing to!



Reviewing std::memcpy()

- Now it's time to have another look at *std::memcpy()*, esp. its signature.

```
void* memcpy(void* destination, const void* source, size_t n);
```

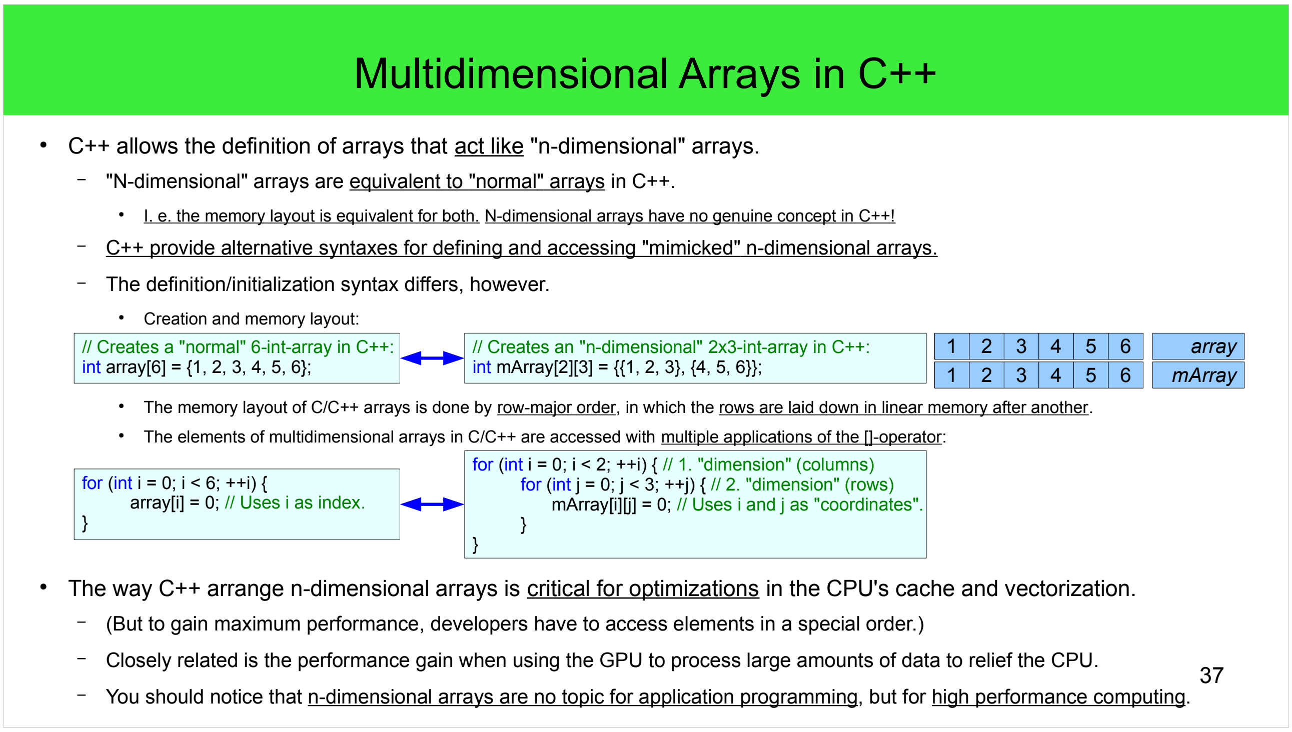
  - Both pointer parameters are void*s. It also returns a void*, namely the *destination* pointer.
  - Notably *source* is a const void*, which makes sense, because *source*'s memory shouldn't be modified.
    - Using const void* tells the caller "Have no worries, your *source* won't be modified!"
    - Using const void* forces *std::memcpy()* not to modify *source*, if it would try to modify it, the compiler won't compile *std::memcpy()*.
  - The last parameter, a *size*_t (which is simply spoken an int), specifies the bytes to be copied from *source* to *destination*.

- But why does *std::memcpy()* use void* parameters?
  - The answer is, that *std::memcpy()* must be able to copy arrays of any element types and all arrays decay to void*.
  - Therefore, void* is called the "generic pointer type".

```
int intArray[] = {1, 2, 3, 4};
int intArray2[4];
std::memcpy(intArray2, intArray, sizeof(int) * 4);
```

```
char charArray[] = {'a', 'b', 'c', 'd'};
char charArray2[4];
std::memcpy(charArray2, charArray, sizeof(char) * 4);
```

- We can assign any pointer type to a void* without casting, this is why it is called "generic" pointer type. Exceptions: function pointers that need to be converted to void* with a reinterpret_cast and const pointers that need to be converted with a const_cast.



Multidimensional Arrays in C++

- C++ allows the definition of arrays that act like "n-dimensional" arrays.
  - "N-dimensional" arrays are equivalent to "normal" arrays in C++.
    - I. e. the memory layout is equivalent for both. N-dimensional arrays have no genuine concept in C++!
  - C++ provide alternative syntaxes for defining and accessing "mimicked" n-dimensional arrays.
  - The definition/initialization syntax differs, however.
    - Creation and memory layout:

```
// Creates a "normal" 6-int-array in C++:
int array[6] = {1, 2, 3, 4, 5, 6};
```

```
// Creates an "n-dimensional" 2x3-int-array in C++:
int mArray[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

1	2	3	4	5	6	*array*
1	2	3	4	5	6	*mArray*

- The memory layout of C/C++ arrays is done by row-major order, in which the rows are laid down in linear memory after another.
- The elements of multidimensional arrays in C/C++ are accessed with multiple applications of the []-operator:

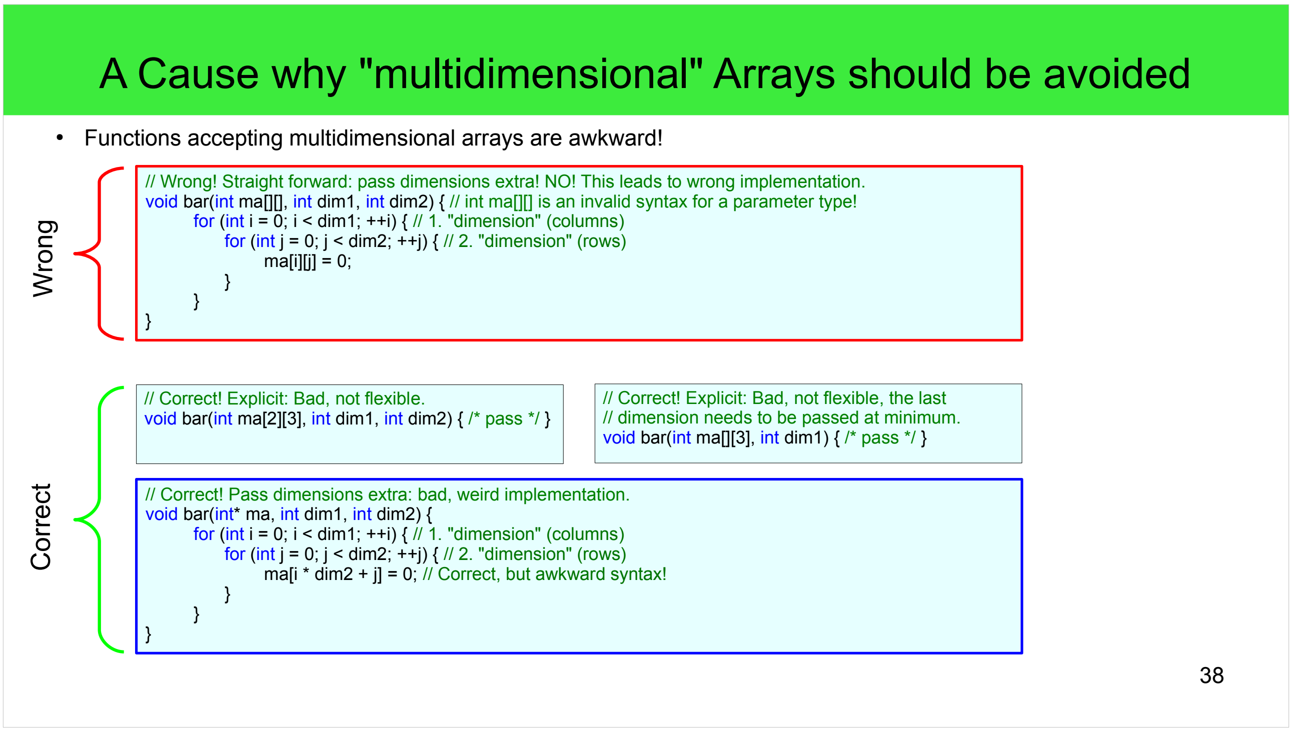
```
for (int i = 0; i < 6; ++i) {
	array[i] = 0; // Uses i as index.
}
```

```
for (int i = 0; i < 2; ++i) { // 1. "dimension" (columns)
	for (int j = 0; j < 3; ++j) { // 2. "dimension" (rows)
		mArray[i][j] = 0; // Uses i and j as "coordinates".
	}
}
```

- The way C++ arrange n-dimensional arrays is critical for optimizations in the CPU's cache and vectorization.
  - (But to gain maximum performance, developers have to access elements in a special order.)
  - Closely related is the performance gain when using the GPU to process large amounts of data to relief the CPU.
  - You should notice that n-dimensional arrays are no topic for application programming, but for high performance computing.

37

- Data vectorization means that blocks of data, such as arrays, are not manipulated element-wise in loops, but manipulated as a whole. CPUs provide special instructions to apply vectorization.



A Cause why "multidimensional" Arrays should be avoided

- Functions accepting multidimensional arrays are awkward!

Wrong

```
// Wrong! Straight forward: pass dimensions extra! NO! This leads to wrong implementation.
void bar(int ma[][], int dim1, int dim2) { // int ma[][] is an invalid syntax for a parameter type!
        for (int i = 0; i < dim1; ++i) { // 1. "dimension" (columns)
             for (int j = 0; j < dim2; ++j) { // 2. "dimension" (rows)
                    ma[i][j] = 0;
             }
        }
}
```

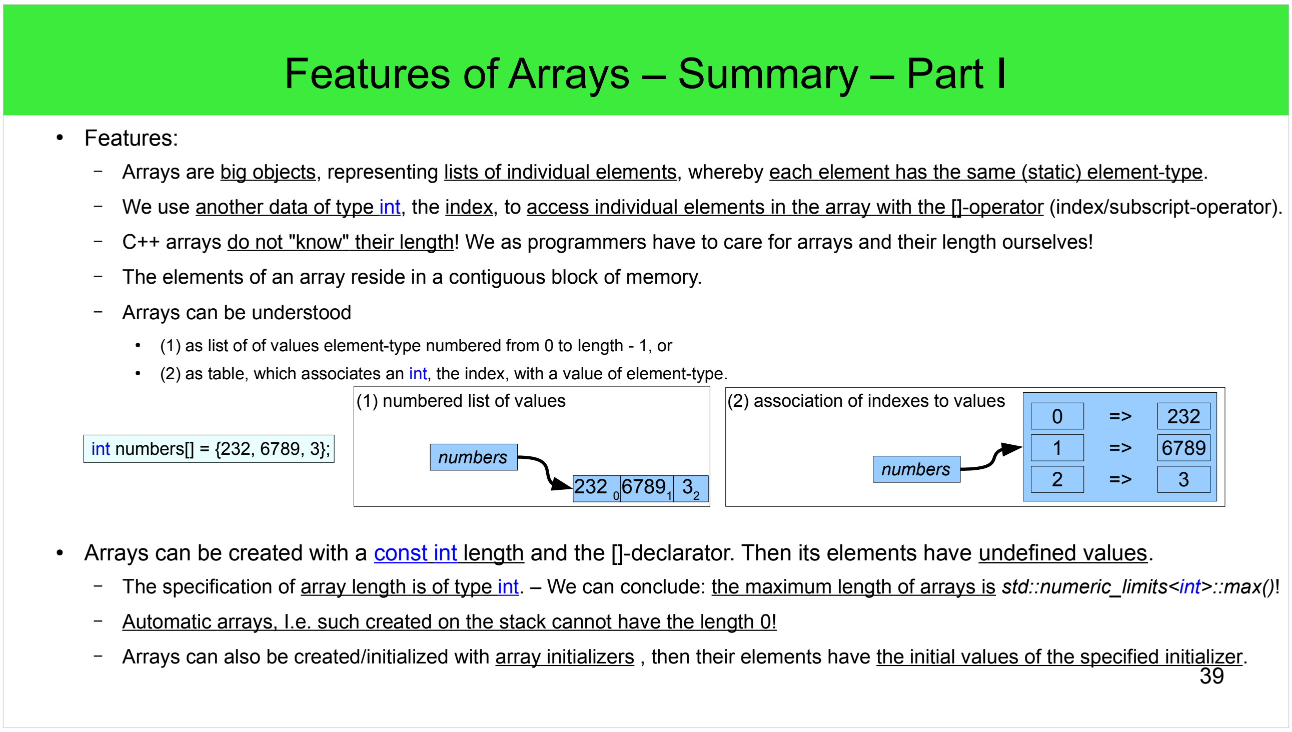
Correct

```
// Correct! Explicit: Bad, not flexible.
void bar(int ma[2][3], int dim1, int dim2) { /* pass */ }
```

```
// Correct! Explicit: Bad, not flexible, the last
// dimension needs to be passed at minimum.
void bar(int ma[][3], int dim1) { /* pass */ }
```

```
// Correct! Pass dimensions extra: bad, weird implementation.
void bar(int* ma, int dim1, int dim2) {
        for (int i = 0; i < dim1; ++i) { // 1. "dimension" (columns)
             for (int j = 0; j < dim2; ++j) { // 2. "dimension" (rows)
                    ma[i * dim2 + j] = 0; // Correct, but awkward syntax!
             }
        }
}
```

- As can be seen in the last correct example, accessing an array defines, whether it is a "normal" one or a multidimensional one.



Features of Arrays – Summary – Part I

- Features:
  - Arrays are big objects, representing lists of individual elements, whereby each element has the same (static) element-type.
  - We use another data of type int, the index, to access individual elements in the array with the []-operator (index/subscript-operator).
  - C++ arrays do not "know" their length! We as programmers have to care for arrays and their length ourselves!
  - The elements of an array reside in a contiguous block of memory.
  - Arrays can be understood
    - (1) as list of of values element-type numbered from 0 to length - 1, or
    - (2) as table, which associates an int, the index, with a value of element-type.

```
int numbers[] = {232, 6789, 3};
```

(1) numbered list of values

(2) association of indexes to values

0	=>	232
1	=>	6789
2	=>	3

- Arrays can be created with a const int length and the []-declarator. Then its elements have undefined values.
  - The specification of array length is of type int. – We can conclude: the maximum length of arrays is *std::numeric_limits<int>::max()*!
  - Automatic arrays, I.e. such created on the stack cannot have the length 0!
  - Arrays can also be created/initialized with array initializers , then their elements have the initial values of the specified initializer.

- If a "data container" should store more than *std::numeric_limits<int>::max()* elements, another type must be used, which uses keys of a type different from int. – Such "data containers" are called associative containers (e.g. *std::map*) in C++, they can potentially solve this limitation.

Features of Arrays – Summary – Part II

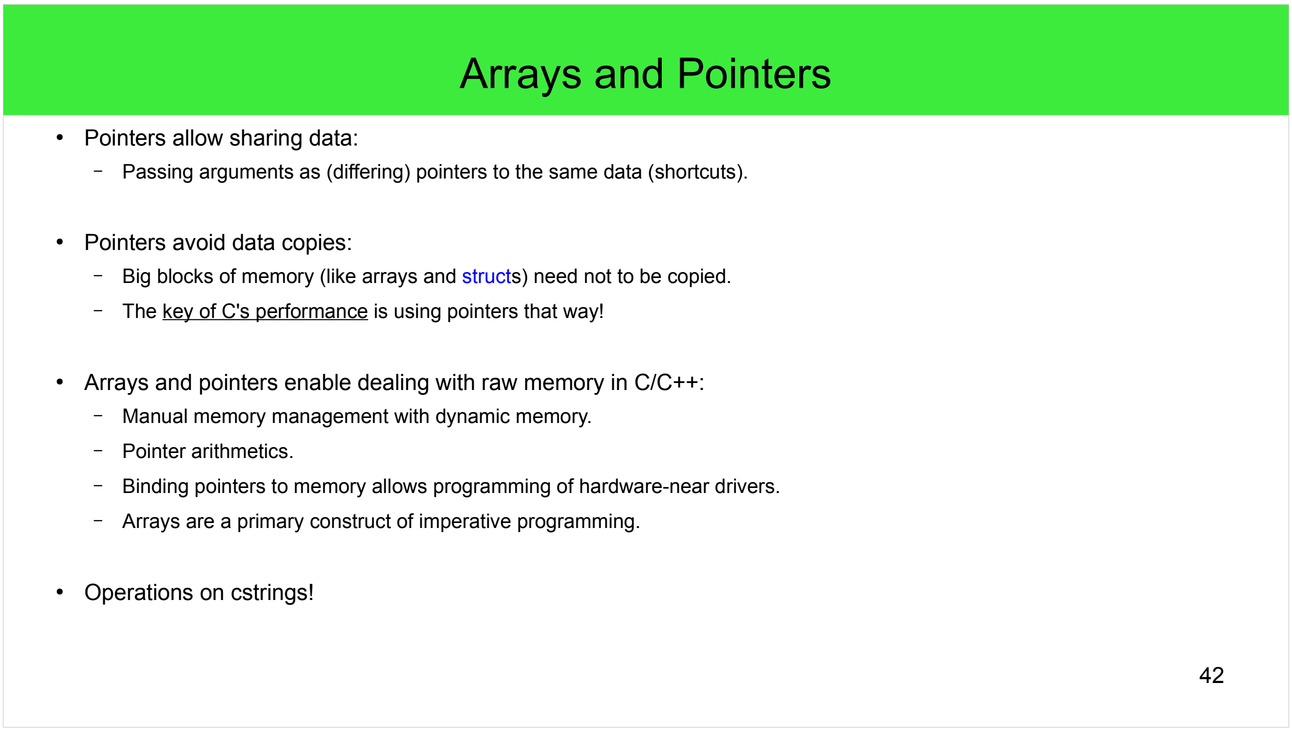
- Array elements can be initialized/filled/set with loops (esp. with for loops) using an index and the []-operator.
  - Alternatively functions like *std::fill_n()* can be used.
- The array index is of type int. – We can conclude: the highest index of arrays is *std::numeric_limits<int>::max() - 1*!
  - The array-indexes are 0-based. – So the indexes' range is [0, length[.
  - If array access exceeds this range, the behavior is undefined.
- Arrays don't have copy semantics, this technically impossible for the general case and would be expensive.
  - When arrays are passed to functions, they decay to a pointer pointing to the first element.
  - Arrays can not be assigned. However pointers to array can be assigned, which leads to aliasing.
  - Automatic arrays can't be returned from functions.
- C++ supports syntactic support to mimic rectangular "multidimensional" arrays on the stack.
- Up to now, we have discussed automatic arrays, which are created on the stack with a compile time defined length.
  - In a future lecture we'll discuss and dynamic arrays, which are created on the heap or freestore with a run time defined length.

Features of Arrays – Summary – Part III

- Random access: index-access to each individual element is allowed at any time in any order!
  - E.g. it is not required to access the element at index 2 only after the elements at 1 and 0 have been accessed.

- Constant complexity access: index-access to each individual element takes the same amount of time!
  - E.g. []-accessing the element at index 5 takes the same amount of time as accessing the element at index 2.
  - Arrays can neither grow nor shrink! I.e. we can not remove or add elements and an array's length is immutable.

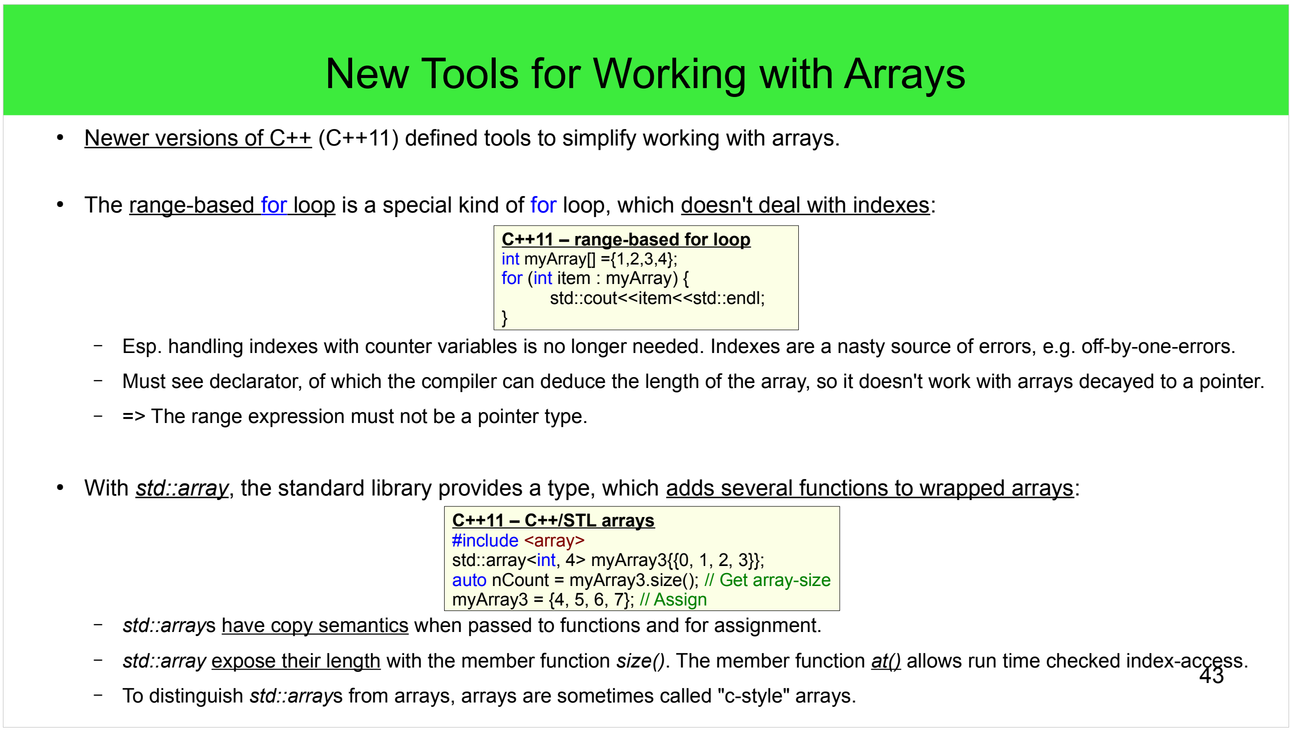
- Terminology alert for German programmers: Stick to calling an array array, not "Feld", even if German literature does!
  - A Feld is a field of a UDT! This is an example where translation is really inappropriate, leading to ridiculous misunderstandings.



Arrays and Pointers

- Pointers allow sharing data:
  - Passing arguments as (differing) pointers to the same data (shortcuts).
- Pointers avoid data copies:
  - Big blocks of memory (like arrays and structs) need not to be copied.
  - The key of C's performance is using pointers that way!
- Arrays and pointers enable dealing with raw memory in C/C++:
  - Manual memory management with dynamic memory.
  - Pointer arithmetics.
  - Binding pointers to memory allows programming of hardware-near drivers.
  - Arrays are a primary construct of imperative programming.
- Operations on cstrings!

- What is pointer arithmetics?
  - Arithmetics is the theory of basic calculation with numbers. Elementary arithmetics describe the operators +, - ,* and /.
  - Arithmetics with pointers deal with elementary arithmetic operators with pointers.



New Tools for Working with Arrays

- Newer versions of C++ (C++11) defined tools to simplify working with arrays.

- The range-based for loop is a special kind of for loop, which doesn't deal with indexes:

C++11 – range-based for loop

```
int myArray[] ={1,2,3,4};
for (int item : myArray) {
        std::cout<<item<<std::endl;
}
```

  - Esp. handling indexes with counter variables is no longer needed. Indexes are a nasty source of errors, e.g. off-by-one-errors.
  - Must see declarator, of which the compiler can deduce the length of the array, so it doesn't work with arrays decayed to a pointer.
  - => The range expression must not be a pointer type.

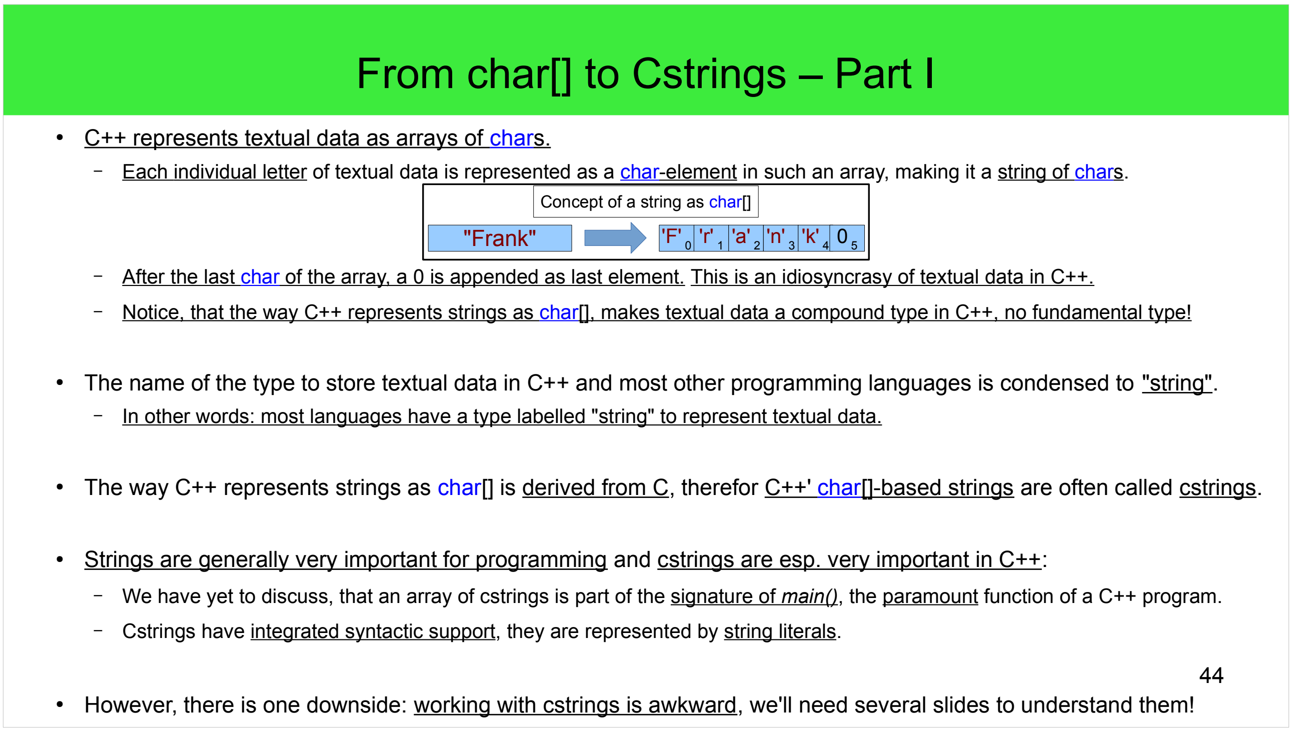
- With *std::array*, the standard library provides a type, which adds several functions to wrapped arrays:

C++11 – C++/STL arrays

```
#include <array>
std::array<int, 4> myArray3{{0, 1, 2, 3}};
auto nCount = myArray3.size(); // Get array-size
myArray3 = {4, 5, 6, 7}; // Assign
```

  - *std::array*s have copy semantics when passed to functions and for assignment.
  - *std::array* expose their length with the member function *size()*. The member function *at()* allows run time checked index-access.
  - To distinguish *std::array*s from arrays, arrays are sometimes called "c-style" arrays.

- However, that the range-based for loop does not work with an index also means, that we don't know on which index the current element resides in the array. – In such a case an explicit counter must be used, or … just use the good old "counting" for loop.



From char[] to Cstrings – Part I

- C++ represents textual data as arrays of chars.
  - Each individual letter of textual data is represented as a char-element in such an array, making it a string of chars.

Concept of a string as char[]

"Frank"

'F' 0 'r' 1 'a' 2 'n' 3 'k' 4 0 5

  - After the last char of the array, a 0 is appended as last element. This is an idiosyncrasy of textual data in C++.
  - Notice, that the way C++ represents strings as char[], makes textual data a compound type in C++, no fundamental type!

- The name of the type to store textual data in C++ and most other programming languages is condensed to "string".
  - In other words: most languages have a type labelled "string" to represent textual data.

- The way C++ represents strings as char[] is derived from C, therefor C++' char[]-based strings are often called cstrings.

- Strings are generally very important for programming and cstrings are esp. very important in C++:
  - We have yet to discuss, that an array of cstrings is part of the signature of *main()*, the paramount function of a C++ program.
  - Cstrings have integrated syntactic support, they are represented by string literals.

- However, there is one downside: working with cstrings is awkward, we'll need several slides to understand them!

- Concerning the term "string" mind the German term "Zeichenkette", which means "string of characters".

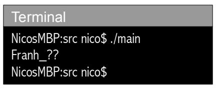
From char[] to Cstrings – Part II

- Cstrings are hard to use, but on the next slides we'll discuss them and apply the knowledge we have gained meanwhile.

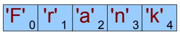
- Individual letters can be held by char/wchar_t elements each and whole strings by char[]/wchar_t[].
  - We'll discuss the type wchar_t in short.

- To distinguish bare char/wchar_t arrays from such representing cstrings, "cstring-char[]" need to be 0-terminated!
  - This means that the very last item of the char[] needs to be a 0 (or '\0')!
  - Example: if we create an "ordinary" char[] containing some letters "making up" a string, printing it to console shows weird results:

```
char chars[] = {'F', 'r', 'a', 'n', 'k'}; // a char array.
// Output to console:
std::cout<<chars<<std::endl;     // This will not work as intended!
```



```
Terminal
NicosMBP:src nico$ ./main
Franh_??
NicosMBP:src nico$
```

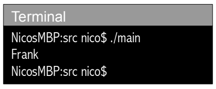


  - The last letters/chars of {'F', 'r', 'a', 'n', 'k'} are not correctly printed! The problem: C++ doesn't get, that it's a cstring, not a bare char[].
  - Technically, C++ doesn't see, where *chars* terminates being a belonging together string! We have to 0-terminate the char[].
  - We can either append a 0 as last element after the last letter, or just put 'Frank' into a string literal:

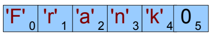
```
char aString[] = {'F', 'r', 'a', 'n', 'k', 0};
```

```
char aString[] = "Frank";
```

```
std::cout<<aString<<std::endl;   // This _will_ work as intended!
```



```
Terminal
NicosMBP:src nico$ ./main
Frank
NicosMBP:src nico$
```



  - As can be seen a 0-terminated char[] can be successfully printed to the console.

From char[] to Cstrings – Part III

```
char hello[] = "Hello, World!";
```

- String literals have to be written as a text enclosed in double quotes!
  - The examples of this course highlight string literals in brown color.

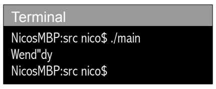
- Escape sequences are symbol-sequences in a string literal with a special meaning.

- E.g. within a string literal we can't use the "-char directly, it must be "escaped".
  - Why is it impossible? Because the compiler has to know the limits of a string literal.

```
char text[] = "Wen"dy"; // Compiler in trouble: can't interpret decl.: is that "Wen" or "Wen"dy"?
```

  - In C++ there exists a set of escape sequences, one of them, '\"' solves this problem:

```
char text[] = "Wen\"dy"; // OK, just use the escape sequence \". "Wen"dy"
std::cout<<text<<std::endl;
```



Terminal

```
NicosMBP:src nico$ ./main
Wend"dy
NicosMBP:src nico$
```

- A //-comments within a string-literal becomes part of the string-literal:

```
char aString[] = "Hello, World! // comment";
```

Good to know
What does "escape" mean, when we talk about strings? An "escape-character" tells the interpreter "Notice, next, there will be a character, that does not belong to the "ordinary character-/typeset", because it is a "control character".

From char[] to Cstrings – Part IV

- Some "letters" have just no "human readable representation", escape sequences will help us here as well.
  - Such "letters" are often so called control codes.
  - Control codes can usually not be entered via the keyboard. Let's examine some of them:

```
// Inserting a newline into a string literal (\r\n means "carriage return" and "newline"):
std::cout<<"Hello,\r\nWorld"<<std::endl; // \r\n will add a blank line below "Hello," on the console:
// >Hello,
// >World
```

```
// Inserting a tab into a string literal (\t):
std::cout<<"a\tb"<<std::endl; // \t will add a tab between a and b on the console:
// >a    b
```

```
// To avoid misinterpretation of \ as a character, it must be escaped as well, so have \\:
std::cout<<"Hello\\World"<<std::endl; // \\ will add a backslash between "Hello" and "World":
// >Hello\World
```

Good to know
String literals will carry a lot of backslashes, if there are many characters to escape, e.g. for Windows file paths:
"C:\\foo\\bar\\text.txt"
(as raw string literal: R"(C:\foo\bar\text.txt)")
Or regular expressions:
"[^\"]*\"([^\"]*)\"\\s*\\(([^\\(]*)\\)"
(as raw string literal: R"_([^"]*"([^"]*)"\s*\(([^\(]*)\))_")
This visual effect in the code is called the "leaning toothpick syndrome".

- C++11 provides so called raw string literals, which allow leaving away a lot of backslashes, improving readability.

```
// A complicated string literal of a Windows file path with many escaped \s:
std::cout<<"C:\\foo\\bar\\text.txt"<<std::endl;
// >C:\foo\bar\text.txt
```

  - When using raw string literals, e.g. enclosing the literal with R"()" lets us leave all the escaping away:

```
// Enclosing the path into a raw string literal, avoids the many escapes:
std::cout<<R"(C:\foo\bar\text.txt)"<<std::endl;
// >C:\foo\bar\text.txt
```

From char[] to Cstrings – Part V

- String literals can be spread over multiple lines without any operator, they are concatenated by the C++ compiler:

```
const char* text1 = "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et"
" dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita"
" kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur"
" sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam"
" voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata"
" sanctus est Lorem ipsum dolor sit amet.";
```

- Alternatively, some C++ compilers allow using the \<newline> escape sequence to extend a string literal (C89 standard):

```
const char* text2 = "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et\
 dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita\
 kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur\
 sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam\
 voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata\
 sanctus est Lorem ipsum dolor sit amet.";
```

- With raw string literals we can literally write newlines into the literal, which will be taken over as actual newlines:

```
const char* text3 = R"(Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et
 dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita
 kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur
 sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam
 voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata
 sanctus est Lorem ipsum dolor sit amet.)";
```

  - With raw string literals, the text will be, e.g., written to the console, exactly as formatted as given in the literal.

From char[] to Cstrings – Part VI

- Initialization of cstring variables as char[]:
  - We use = to initialize cstring variables, as with other types.
  - A char[] must be initialized, e.g. with a string literal!
  - A char[] cannot be assigned!

```
char name[] = "Arthur";    // initialization of name
```

```
char otherName[];          // Invalid! uninitialized char[]
```

```
char otherName[] = "Mary";
otherName = name;          // Invalid! assigning another variable
otherName = "Jamie";       // Invalid! assigning a value
```

- Because a cstring is just a char[], it can be interpreted as a sequence of characters, like a text is a sequence of letters.
  - An individual letter of a cstring is represented by a value of type char.
  - We can get an individual letter of a cstring by using the char[]'s []-operator:

```
char firstLetter = name[0];
// firstLetter = 'A'
```

- When accessing a cstring as char[] the same rules as for other arrays are valid:
  - The element type is char.
  - We don't know the length of the array. – But we can remedy this problem for cstrings as we'll see soon.
  - Exceeding index-access leads to undefined behavior.

Individual chars and ASCII – Part I

- C++' cstrings can be represented by char[], however char[] can only store letters, which can be represented by 1B.
  - This constrains the letters we can store in cstrings, in that we can only store 256 different sorts of chars in cstrings.
  - More exactly, we can only store chars, which are representable by 1B/8b, which makes 256 combinations/letters.
  - Those combinations are defined as American Standard Code for Information Interchange, abbreviated as ASCII.
  - Actually, ASCII only covers 7b-code, i.e. only 128 individual combinations/letters, which are written down as a table.
  - Here a part of the ASCII table:

ASCII Code	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol
…	…	…	…	75	**K**	86	**V**	102	**f**	113	**q**
48	**0**	65	**A**	76	**L**	87	**W**	103	**g**	114	**r**
49	**1**	66	**B**	77	**M**	88	**X**	104	**h**	115	**s**
50	**2**	67	**C**	78	**N**	89	**Y**	105	**i**	116	**t**
51	**3**	68	**D**	79	**O**	90	**Z**	106	**j**	117	**u**
52	**4**	69	**E**	80	**P**	…	…	107	**k**	118	**v**
53	**5**	70	**F**	81	**Q**	97	**a**	108	**l**	119	**w**
54	**6**	71	**G**	82	**R**	98	**b**	109	**m**	120	**x**
55	**7**	72	**H**	83	**S**	99	**c**	110	**n**	121	**y**
56	**8**	73	**I**	84	**T**	100	**d**	111	**o**	122	**z**
57	**9**	74	**J**	85	**U**	101	**e**	112	**p**	…	…

Individual chars and ASCII – Part II

ASCII Code	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol
…	…	…	**…**	75	**K**	86	**V**	102	**f**	113	**q**
48	**0**	65	**A**	76	**L**	87	**W**	103	**g**	114	**r**
49	**1**	66	**B**	77	**M**	88	**X**	104	**h**	115	**s**
50	**2**	67	**C**	78	**N**	89	**Y**	105	**i**	116	**t**
51	**3**	68	**D**	79	**O**	90	**Z**	106	**j**	117	**u**
52	**4**	69	**E**	80	**P**	…	**…**	107	**k**	118	**v**
53	**5**	70	**F**	81	**Q**	97	**a**	108	**l**	119	**w**
54	**6**	71	**G**	82	**R**	98	**b**	109	**m**	120	**x**
55	**7**	72	**H**	83	**S**	99	**c**	110	**n**	121	**y**
56	**8**	73	**I**	84	**T**	100	**d**	111	**o**	122	**z**
57	**9**	74	**J**	85	**U**	101	**e**	112	**p**	…	…

- Digit symbols and letter symbols have increasing and adjacent ASCII codes following their lexicographic or numeric order.
  - The ASCII code of 'Q' is a smaller value then the ASCII code of 'S'. This is handy, because Q is lexicographically less then S in a dictionary.
  - The ASCII code of '1' is a smaller value then the ASCII code of '2'. This is handy, because 1 < 2 in the set of integer numbers.
- Digit symbols and letter symbols have a gap in the ASCII table at]58, 64[.
- Upper case letter symbols and lower case symbols have a gap in the ASCII table at]91, 96[.
- Upper case letter symbols have smaller ASCII codes than lower case letter symbols.

Individual chars and ASCII – Part III

- Because ASCII only defines 128 characters, there is still some "space" to fill the 1B of a full char element.

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

  - However, some vendors produced incompatible ASCII versions, in which some characters were replaced by locale specific ones.

- The American National Standards Institute (ANSI) equalized ASCII as follows:
  - The first 7b, i.e. 128th characters have all been fixed.
  - The 8thb was also used: beyond the 128th character to the 256th character the character set was opened to locale-specific variants.
  - One of the variants is ISO/IEC 8859-1, corresponding to ASCII plus characters for western European languages such as German.
  - => Effectively, with the ANSI extension, we can store 256 different kinds of characters in a char.

- But meanwhile 256 different ones are not enough: wider character sets where introduced to cover up to 32b characters.
  - ANSI is still a compromise, because, e.g. western Europe and Greek characters couldn't coexist, 256 characters were just too few!

- An extension and replacement of the ANSI character set is the Unicode character set, which can handle 32b characters.

- But ... the type char can't handle 32b characters, but only 8b characters! Therefor C++ provides the type wchar_t.

char[] and wchar_t[] in Memory – Part I

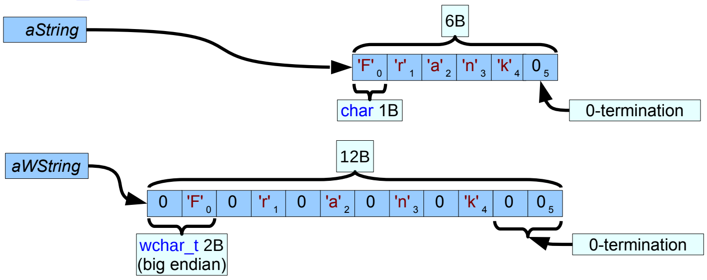
- Simply spoken, wchar_t is a wider version of char, hence its name "wide char type".

- The sizeof(wchar_t) must at least be 2, i.e. 2B. This allows to store 65.536 characters of different kind at least!

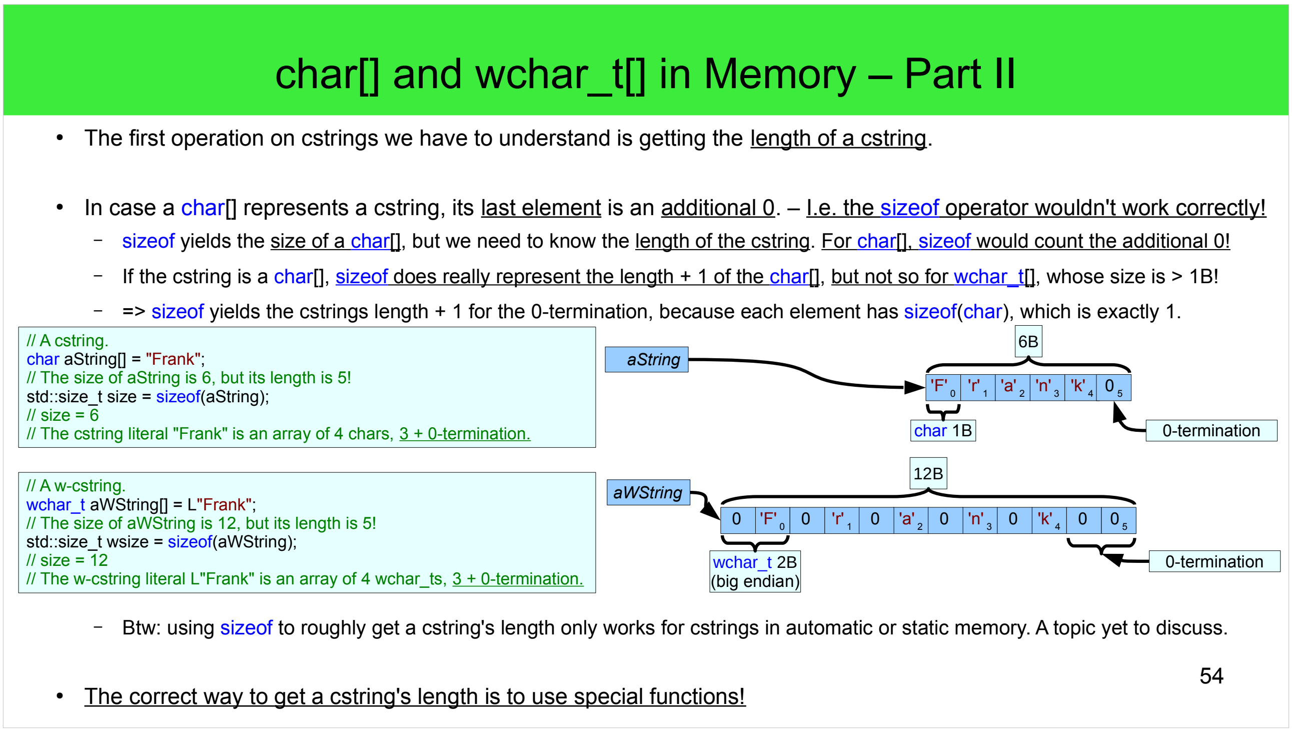
- Let's compare the same cstring with char and wchar_t:

```
// A cstring.
char aString[] = "Frank";
std::size_t size = sizeof(aString);
// size = 6
```

```
// A w-cstring.
wchar_t aWString[] = L"Frank";
std::size_t wsize = sizeof(aWString);
// size = 12
```


aString
6B
'F' 0 | 'r' 1 | 'a' 2 | 'n' 3 | 'k' 4 | 0 5
char 1B
0-termination
aWString
12B
0 | 'F' 0 | 0 | 'r' 1 | 0 | 'a' 2 | 0 | 'n' 3 | 0 | 'k' 4 | 0 | 0 5
wchar_t 2B
(big endian)
0-termination

  - wchar_t-based cstring literals are written with the L-prefix, e.g. L"Frank" instead of "Frank".
  - In this case wchar_t has a size (width) of 2B, therefor it requires twice the memory space of char.
  - If characters fit into the first byte of the wchar_t (2B), the remaining byte will be set to the value 0.



char[] and wchar_t[] in Memory – Part II

- The first operation on cstrings we have to understand is getting the length of a cstring.
- In case a char[] represents a cstring, its last element is an additional 0. – I.e. the sizeof operator wouldn't work correctly!
  - sizeof yields the size of a char[], but we need to know the length of the cstring. For char[], sizeof would count the additional 0!
  - If the cstring is a char[], sizeof does really represent the length + 1 of the char[], but not so for wchar_t[], whose size is > 1B!
  - => sizeof yields the cstrings length + 1 for the 0-termination, because each element has sizeof(char), which is exactly 1.

```
// A cstring.
char aString[] = "Frank";
// The size of aString is 6, but its length is 5!
std::size_t size = sizeof(aString);
// size = 6
// The cstring literal "Frank" is an array of 4 chars, 3 + 0-termination.
```

```
// A w-cstring.
wchar_t aWString[] = L"Frank";
// The size of aWString is 12, but its length is 5!
std::size_t wsize = sizeof(aWString);
// size = 12
// The w-cstring literal L"Frank" is an array of 4 wchar_ts, 3 + 0-termination.
```

  - Btw: using sizeof to roughly get a cstring's length only works for cstrings in automatic or static memory. A topic yet to discuss.
- The correct way to get a cstring's length is to use special functions!

- What is "big endian"?
  - When we have data that is composed of multiple bytes (i.e. integers greater than 1B or character types greater than 1B (But not multiple chars of a cstring!)), these bytes can be arranged in memory in different byte orders.
- On big endian byte order, the most significant bits resides (i.e. the bits contributing the highest amount to the value) on lowest address of the value in memory. This order directly reflects the "reading direction" of the value. -> 68000 and PowerPC (default) A cstring is always stored in big endian order (but not necessarily its char/wchar_t elements), because pointer arithmetics must work w/ all kinds of arrays.
  - On little endian byte order, the most significant bits reside on the highest address of the value in memory. This order reverses the byte sequence and the "reading direction" of the value in memory. -> Intel

Cstrings: Length and Element Access

- The correct way of getting the length of a cstring is using the function *std::strlen()* in <cstring>:

```
char aString[] = "bar"; // A cstring.
std::size_t size = sizeof(aString); // The size of aString is 4, but its length is 3!
std::size_t length = std::strlen(aString); // Correct: The length of aString is 3.
```

Good to know
For w-cstrings, the function *std::wcslen()* (<cwchar>) must be used.

- Having the correct length of the cstring, we can access each element (read "each letter") of *aString* via the []-operator:

```
for (int i = 0; i < length; ++i) {      // Virtually, i should be of std::size_t, but declaring i
                                        // as int is the canonical form...
      std::cout<<aString[i]<<std::endl; // Accessing the cstring as array.
}
```

- *std::strlen()* could be implemented like so (returning int instead of *std::size_t*):

```
int strlen(const char* input) {
      int position = 0;
      while (input[position] != 0) { // While the 0-termination is not hit: continue counting!
            ++position;
      }
      return position;
}
```

- *std::strlen()*'s algorithm works, because it knows, that a cstring starts at the passed address and has a 0 as last element.
  - Passing a not-0-terminated char[] to *std::strlen()* would lead it to counting along the memory until a 0 is found: this is usually a bug.

- All C++'s cstring-related functions work that way, relying on the fact, that cstrings are 0-terminated.

- Cstring-related functions are not null-aware, i.e. they behave undefined, if a passed pointer is a nullptr.

Cstrings and Constness

- We've just discussed, that arrays and pointers have strong connections via pointer-decay, this is also true for cstrings:

```
// Decay from char[] of array-initializer:
char aString[] = {'F', 'r', 'a', 'n', 'k', 0};
char* aStringDecayed = aString;
```

```
// Decay from char[] of string literal:
char aString[] = "Frank";
char* aStringDecayed = aString;
```

  - Of course, handling constant cstrings as string literals is the more traditional and compact form.

- The other tradition in C++ is to accept pointer types in favor to array types, but there is a problem with string literals:

```
// Invalid: decay from string literal to char*:
char* aStringDecayed = "Frank";
```

  - It doesn't work! Instead, C++ assumes all cstrings, not only string literals, to represent a constant region in memory!
  - What we have to do: string literals must be represented as const char*:

```
// Invalid: decay from string literal to char*:
const char* aStringDecayed = "Frank";
```

- All cstring functions, such as *std::strlen()* do not await a char* but instead a const char*.

- Hence, we will represent all cstrings in our code as const char*.

- All right, but using const char* as type for cstrings unleashed another feature of cstrings: cstrings are immutable!

```
// Invalid: Read-only variable is not assignable:
aStringDecayed[1] = 'u';
```

Cstrings and their Array Nature

- Because of the representation of cstrings as const char* after decay, some rules must be discussed.

- When in const char* form, a cstring can be uninitialized and we could pass nullptr, where const char* is expected.

```
const char* name;                 // OK! but name is uninitialized
```

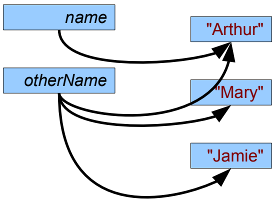
  - The value of *name* is undefined, it will point to any address in memory.

```
const char* name = nullptr;     // OK! name is initialized to nullptr
```

  - Sure, when we pass nullptr to a function, it should better be null aware.

- Assigning const char* also works, but it doesn't really copy a cstring:

```
const char* name = "Arthur";
const char* otherName = "Mary";
otherName = name;       // OK! assigning another pointer, but leads to aliasing
otherName = "Jamie";    // OK! assigning a new value
```



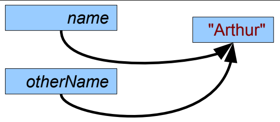
  - The problem with assignment: it leaves two pointers containing the same address!
  - So, the assignment doesn't copy the array, it just copies the pointer.
  - We say, we only have a shallow copy of the array, not a deep copy!

- Why can we assign at all? *otherName* is const! – No, the pointer is not const, only the memory it is pointing to!

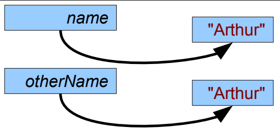
Deep-Copying Cstrings – Part I

- C++ also provides means to create a deep copy of a cstring. Because a cstring is an array, we can use *std::memcpy()*:

```
const char* name = "Arthur";
const char* otherName = nullptr;
// Assigning to another pointer, leads to a shallow copy and aliasing:
otherName = name;
```



```
const char* name = "Arthur";
char otherName[7];
// Memory copy to another buffer, leads to a deep copy:
std::memcpy(otherName, name, sizeof(char) * (std::strlen(name) + 1));
```



- There are important differences:
  - *otherName* must be of type *char[length+1]*, which acts as buffer.
    - This creates an uninitialized char[] with enough space to keep the source cstring of length and the 0-termination.
    - Mind, that we cannot create this char[] with the length of name calculated at run time, e.g. via *std::strlen()*!
  - The last param of *std::memcpy()* gets the calculated size of the memory to copy in bytes as *sizeof(char) * (std::strlen(name) + 1)*.

- Mind, that we don't assign pointers, but create an uninitialized buffer and copy a section of memory into that buffer.

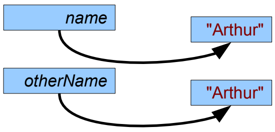
- Once again: mind, that this buffer must have enough space to hold the source memory!

Deep-Copying Cstrings – Part II

- Using *std::memcpy()* is cumbersome to use and also potentially dangerous.
  - (1) We have to pass the size of the memory section to copy, and we have to get the calculation of this size absolutely correct.
  - (2) *std::memcpy()* accepts void*, so we could accidentally pass pointers to different types: copying could end in a disaster!
  - => Getting those aspects wrong can lead to accidentally overwriting memory, which we don't own!
  - => Getting those aspects wrong can lead to accidentally copying memory, we do not want to be copied!

- C++ provides a special function to make assigning of cstring a little bit better: *std::strcpy()* (<cstring>):

```
const char* name = "Arthur";
char otherName[7];
// String copy to another buffer, leads to a deep copy of a cstring:
std::strcpy(otherName, name);
```



name

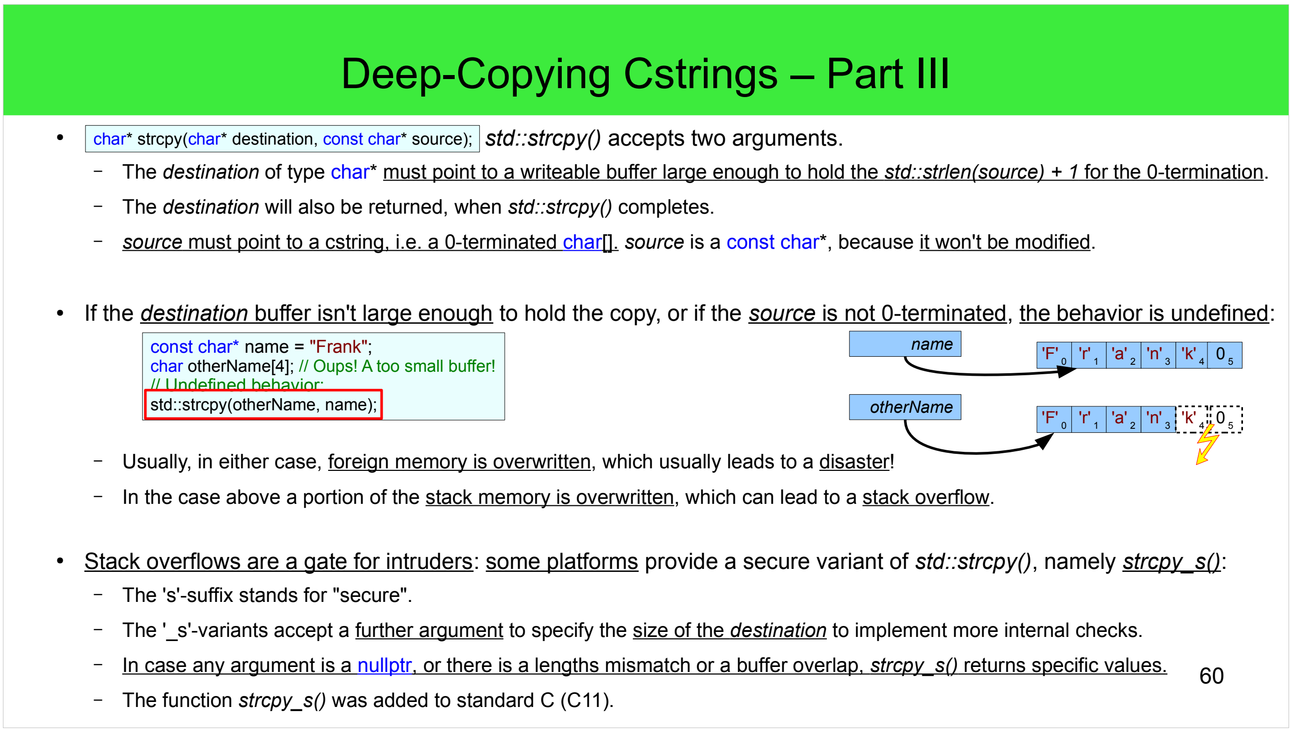
"Arthur"

otherName

"Arthur"

- There are some important differences:
  - We still have to create a destination buffer of type *char[length+1]*, we cannot create an array with a run time-calculated length.
  - *std::strcpy()* doesn't need to know the count of bytes to copy: it counts chars until the 0-termination is reached, sizeof(char) is fix.
  - => So, the simplification: *std::strcpy()* does automatically count the bytes to be copied from source to destination.
  - => *std::strcpy()* has char*/const char* parameters, so we cannot accidentally pass differing pointed-to types!

- There is still potential to use *std::strcpy()* wrongly: the buffer could be too small or the source might not be 0-terminated!



Deep-Copying Cstrings – Part III

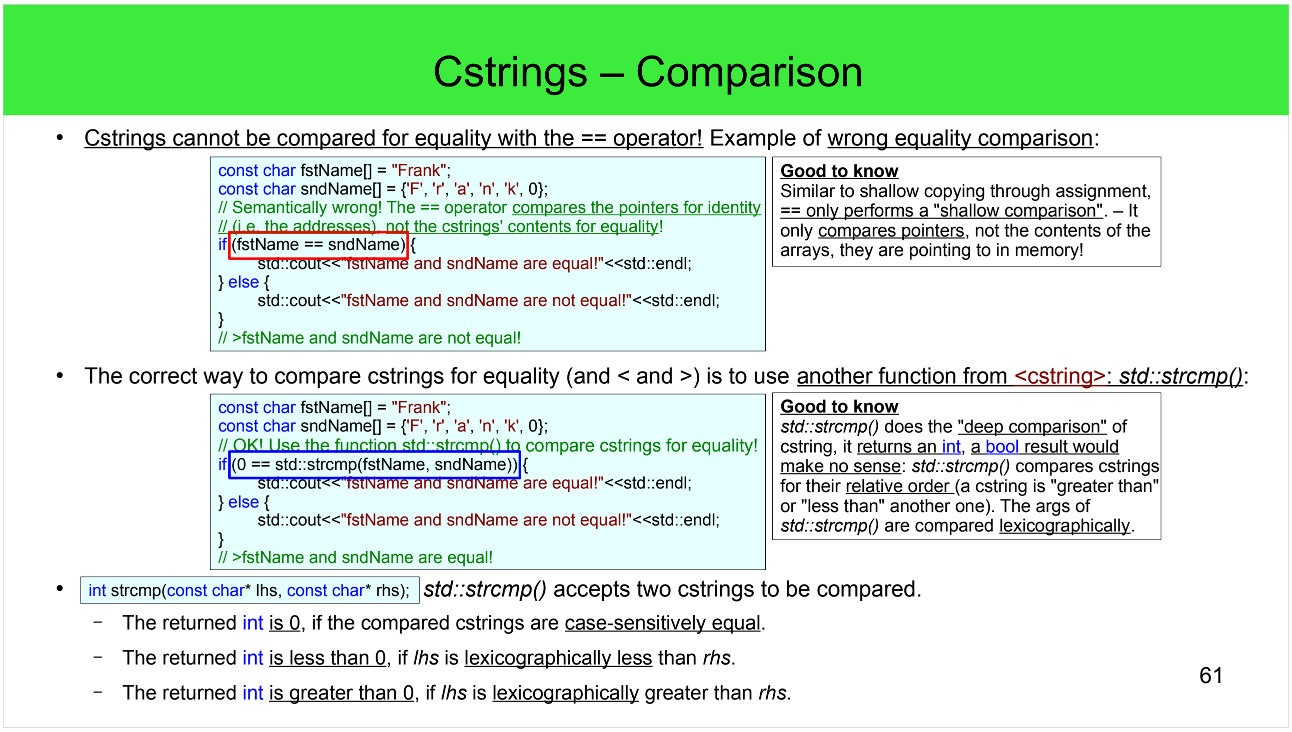
- `char* strcpy(char* destination, const char* source);` *std::strcpy()* accepts two arguments.
  - The *destination* of type char* must point to a writeable buffer large enough to hold the *std::strlen(source) + 1* for the 0-termination.
  - The *destination* will also be returned, when *std::strcpy()* completes.
  - *source* must point to a cstring, i.e. a 0-terminated char[]. *source* is a const char*, because it won't be modified.
- If the *destination* buffer isn't large enough to hold the copy, or if the *source* is not 0-terminated, the behavior is undefined:

```
const char* name = "Frank";
char otherName[4]; // Oups! A too small buffer!
// Undefined behavior:
std::strcpy(otherName, name);
```

name
'F' 0 'r' 1 'a' 2 'n' 3 'k' 4 0 5
otherName
'F' 0 'r' 1 'a' 2 'n' 3 'k' 4 0 5

  - Usually, in either case, foreign memory is overwritten, which usually leads to a disaster!
  - In the case above a portion of the stack memory is overwritten, which can lead to a stack overflow.
- Stack overflows are a gate for intruders: some platforms provide a secure variant of *std::strcpy()*, namely *strcpy_s()*:
  - The 's'-suffix stands for "secure".
  - The '_s'-variants accept a further argument to specify the size of the *destination* to implement more internal checks.
  - In case any argument is a nullptr, or there is a lengths mismatch or a buffer overlap, *strcpy_s()* returns specific values.
  - The function *strcpy_s()* was added to standard C (C11).

- C++ also provides *std::strncpy()*, which accepts the count of chars to be copied, which can also lead to more security (if the source cstring has more than the chars to copy, *std::strncpy()* won't set a 0-temination!). – But still, the destination could be too small, therefore some platforms also define *std::strncpy_s()*, which allows to specify destination's length as well.



Cstrings – Comparison

- Cstrings cannot be compared for equality with the == operator! Example of wrong equality comparison:

```
const char fstName[] = "Frank";
const char sndName[] = {'F', 'r', 'a', 'n', 'k', 0};
// Semantically wrong! The == operator compares the pointers for identity
// (i.e. the addresses), not the cstrings' contents for equality!
if (fstName == sndName) {
	std::cout<<"fstName and sndName are equal!"<<std::endl;
} else {
	std::cout<<"fstName and sndName are not equal!"<<std::endl;
}
// >fstName and sndName are not equal!
```

Good to know
Similar to shallow copying through assignment, == only performs a "shallow comparison". – It only compares pointers, not the contents of the arrays, they are pointing to in memory!

- The correct way to compare cstrings for equality (and < and >) is to use another function from <cstring>: *std::strcmp()*:

```
const char fstName[] = "Frank";
const char sndName[] = {'F', 'r', 'a', 'n', 'k', 0};
// OK! Use the function std::strcmp() to compare cstrings for equality!
if (0 == std::strcmp(fstName, sndName)) {
	std::cout<<"fstName and sndName are equal!"<<std::endl;
} else {
	std::cout<<"fstName and sndName are not equal!"<<std::endl;
}
// >fstName and sndName are equal!
```

Good to know
std::strcmp() does the "deep comparison" of cstring, it returns an int, a bool result would make no sense: *std::strcmp()* compares cstrings for their relative order (a cstring is "greater than" or "less than" another one). The args of *std::strcmp()* are compared lexicographically.

- `int strcmp(const char* lhs, const char* rhs);` *std::strcmp()* accepts two cstrings to be compared.
  - The returned int is 0, if the compared cstrings are case-sensitively equal.
  - The returned int is less than 0, if *lhs* is lexicographically less than *rhs*.
  - The returned int is greater than 0, if *lhs* is lexicographically greater than *rhs*.

- In this example we didn't assign the same literal cstring to *fstName* and *sndName* … why? – Many C++ compilers are clever: in case they spot the very same string literal for multiple times, they'll just store this data for one time in the static memory, this optimization is called string pooling. – If the compiler does this, *fstName* and *sndName* will actually point to the same cstring in the static memory and comparing them using == would actually evaluate to true, because both pointers hold the very same address!
- The exact numeric result of *std::strcmp()* has no meaning! Only whether their result is less than, greater than or equal to zero is relevant.
- Upper case letters are considered "less than" lower case letters.
- C/C++ don't provide a way to perform a case-insensitive cstring comparison. We have to code it ourself (e.g. via *std::tolower()* or *std::toupper()*).

Cstrings – Searching individual chars

- Yet another basic function is searching a specific letter in a cstring, this can be done with *std::strchr()* (<cstring>).

- Searching an individual char works like this:

```
const char* aString = "bar"; // Search the first 'a' in aString,
const char* result = std::strchr(aString, 'a'); // result will point to substring "ar".
result = std::strchr(aString, 'z'); // There is no 'z' in aString, so result is nullptr.
```

- `const char* strchr(const char* input, int _char);` *std::strchr()* accepts a cstring to be searched in and a char to search in the first argument.
  - The function returns a const char* pointing to the first substring starting with the char to be found.
  - The function returns nullptr, if the char to be found is not contained in the passed cstring.
  - There also exists the function *std::strrchr()* (with the same signature), which searches in reverse order.

- To count all occurrences of an individual char we can use *std::strchr()* repeatedly in a loop:

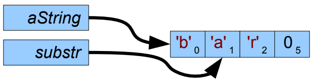
```
const char* aString = "bananas";
const char* result = std::strchr(aString, 'a'); // Search the first 'a' in aString.
int occurrences = 0;
while (nullptr != result) { // While the substring addressed by result is not 0:
        ++occurrences;                         // 1. Increment occurrences.
        ++result;                              // 2. Advance result to next char.
        result = std::strchr(result, 'a');     // 3. Search next 'a' in result.
}
// occurences = 3
```

Cstrings – Searching Substrings

- We have somewhat jumped over the discussion of *std::strchr()*. We mentioned the term substring, what is a "substring"?

- A substring is simply a part or portion of a "full" string, let's use *std::strchr()* to show this again

```
const char* aString = "bar"; // Search the first 'a' in aString.
const char* substr = std::strchr(aString, 'a'); // substr points to the substring "ar".
```



  - *aString* points to a "full" cstring.
  - *substr* points to a portion of *aString*, *substr* and *aString* share some chars and the 0-termination.
  - Thus, *substr* is a substring of *aString*.

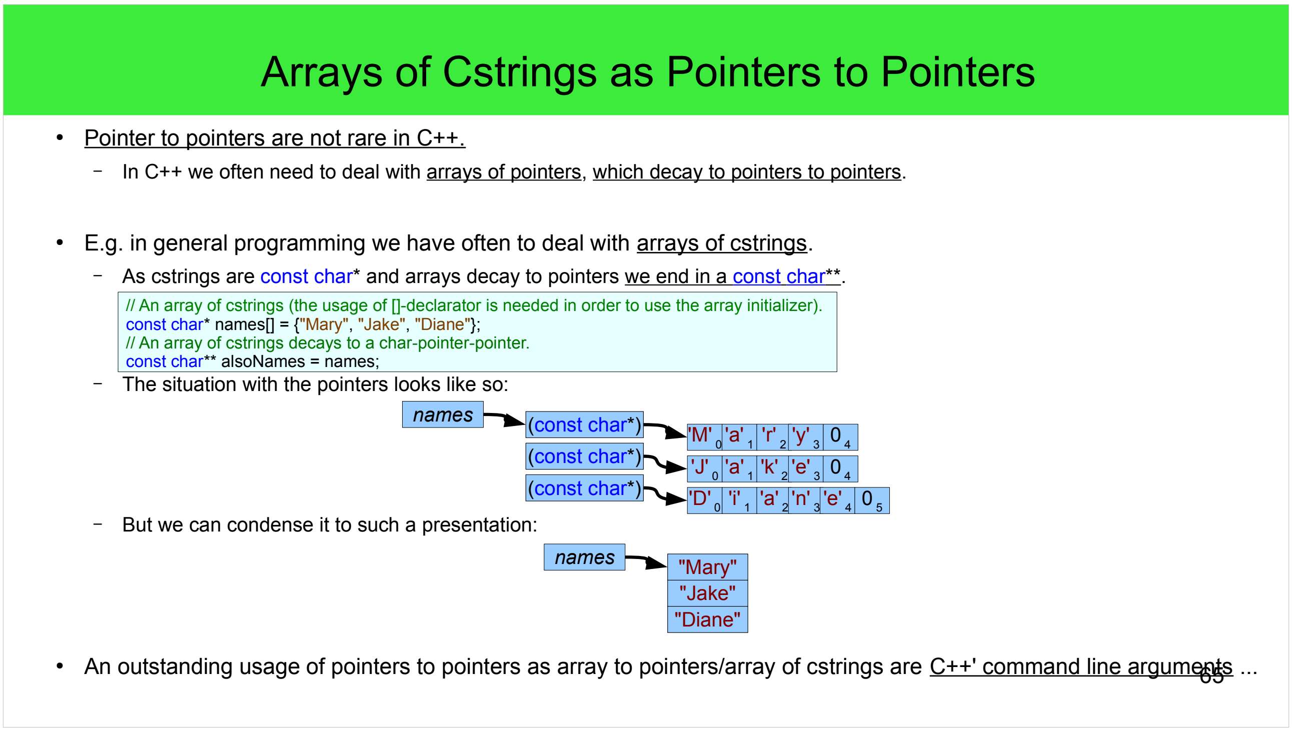
- C++ also allows to search substrings in other strings with *std::strstr()*:

```
const char* aString = "thinking"; // Search substring "ki" in aString,
const char* result = std::strstr(aString, "ki"); // result will point to substring "king".
result = std::strstr(aString, "zap"); // There is no "zap" in aString, so result is nullptr.
```

- `const char* strstr(const char* input, const char* substr);` *std::strstr()* accepts a cstring to be searched in and a substring to search in the first arg.
  - The function returns a const char* pointing to the first substring starting with the substring to be found.
  - The function returns nullptr, if the substring to be found is not contained in the passed cstring.
  - There exists no "reverse-version" of *std::strstr()*.

Summary: Cstring-related Functions

- Cstring-related functions working on char-based cstrings are declared in <cstring>.
- Cstring-related functions also for exist wchar_t-based cstrings, they are declared in <cwchar>.
  - *std::strlen()* -> *std::wcslen(), std::strcpy()* -> *std::wcscpy()*, *std::strcmp()* -> *std::wcscmp()*, *std::strstr()* -> *std::wcsstr()* etc.
  - There also exist wchar_t-based output- and input-streams *std:wcout* and *std::wcin* (<iostream>).
- These functions are not null-aware! Passing nullptrs as arguments leads to undefined behavior.
- If passed buffers are not large enough or if passed cstrings are not 0-terminated, the behavior is undefined.
- The passed buffers must be writeable memory.
  - E.g. char[], which are created on a function's stack with a fixed length at compile time (i.e. the automatic memory).
  - E.g. a dynamically allocated region of memory with a length calculated at run time in the heap memory or the free store.
    - A bigger topic, we'll discuss soon.
- There exist more cstring-related functions, we didn't discuss in this lecture.



Arrays of Cstrings as Pointers to Pointers

- Pointer to pointers are not rare in C++.
  - In C++ we often need to deal with arrays of pointers, which decay to pointers to pointers.

- E.g. in general programming we have often to deal with arrays of cstrings.
  - As cstrings are const char* and arrays decay to pointers we end in a const char**.

```
// An array of cstrings (the usage of []-declarator is needed in order to use the array initializer).
const char* names[] = {"Mary", "Jake", "Diane"};
// An array of cstrings decays to a char-pointer-pointer.
const char** alsoNames = names;
```

  - The situation with the pointers looks like so:

names
(const char*)
(const char*)
(const char*)
'M' 0 'a' 1 'r' 2 'y' 3 0 4
'J' 0 'a' 1 'k' 2 'e' 3 0 4
'D' 0 'i' 1 'a' 2 'n' 3 'e' 4 0 5

  - But we can condense it to such a presentation:

- An outstanding usage of pointers to pointers as array to pointers/array of cstrings are C++' command line arguments ...

- Array initializers can not be used to initialize a char**.

Review of the main()-Function and Command Line Arguments Processing – Part I

- After we have a better understanding of arrays, cstring and arrays of cstrings, we'll have another look at the function *main()*:

```
int main(int argc, const char** argv) {
        // pass
}
```

  - Indeed, *main()* can offer a signature, which deals with the command line arguments, that are passed when the program is started.
  - I.e. up to now, we have had just no use of this optional signature of *main()*.

- There can be only one *main()* either with the signature *main()* or *main(int, const char**)*.
  - I.e. C++ disallows overloading *main()*!

- However, *main()* offers two parameters:
  - the int *argc* contains the count of passed command line arguments, and
  - the const char** *argv* contains the command line arguments as array of cstrings. The length of this array is represented by *argc*!

- The names *argc* and *argv* are not mandatory for *main()*'s params, but quite the traditional naming.
  - *argv* can be read as "argument vector".

- C++ does not define, in which memory (e.g. automatic or dynamic) command line args are stored.

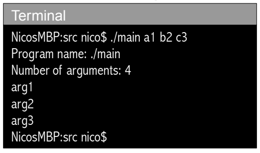
Review of the main()-Function and Command Line Arguments Processing – Part II

- Now we can process the command line arguments via *main()*'s parameter *argv*

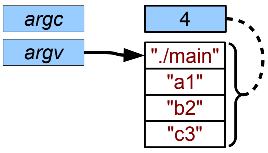
```
int main(int argc, char** argv) {
        std::cout<<"Program name: "<<argv[0]<<std::endl;
        std::cout<<"Number of arguments: "<<argc<<std::endl;

        for (int i = 1; i < argc; ++i) {
                std::cout<<argv[i]<<std::endl;
        }
}
```

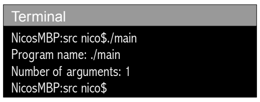
- Let's start this program with three arguments, which will be printed to the console:



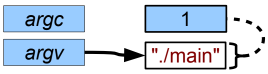
```
Terminal
NicosMBP:src nico$ ./main a1 b2 c3
Program name: ./main
Number of arguments: 4
arg1
arg2
arg3
NicosMBP:src nico$
```



  - We can also pass no command line arguments at all, then nothing will be printed to the console:



```
Terminal
NicosMBP:src nico$./main
Program name: ./main
Number of arguments: 1
NicosMBP:src nico$
```

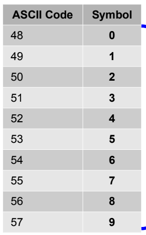


- Mind, that the first element in the argument list is always the name of the program itself!
  - This means, that the variable *argc* will always have a value <= 1.

Working with individual chars

- Some slides ago, we introduced the ASCII table, which maps a numeric code to (almost) every character.
  - A remarkable fact is, that the characters at codes [48, 57] are digits.

- In C++ it is easy to get the ASCII code of a char as int: we just use C++' implicit conversion from char to int!
  - With this information, we can write a function, which tells us, if the passed char is a digit:



ASCII Code	Symbol
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9

```
// Checks, whether ch is in the "range"
// of the ASCII codes of digits:
bool isDigit(char ch) {
        int asciiCode = ch; // Assign char to int variable.
        return asciiCode >= 48 && asciiCode <= 57;
}
```

```
// Alternatively, the param itself can be of type int
// the implicit conversion is performed, when a char
// argument is passed.
bool isDigit(int ch) {
        return ch >= 48 && ch <= 57;
}
```

Good to know
Functions, which return a logical information (true/false) about a passed argument are often called predicates. Predicates are simple to use, and often simple to code: they are highly reusable functions.

  - Using *isDigit()* is simple:

```
std::cout<<std::boolalpha<<isDigit('4')<<std::endl;
// >true
```

- Knowing about the ASCII table, we can develop other predicates, analyzing chars having codes in a specific range.
  - Besides *isDigit()* we could write *isUpperCase()* or *isLowerCase()*.
  - C++ already provides such predicates in <cctype>, which analyze chars in that way. – We don't have to write them ourselves!
  - Examples: *std::isdigit()*, *std::isupper()*, *std::islower()* and more.

Cstrings – Cstrings containing Numbers

- Sometimes, textual data contains other data, which can be interpreted as data of fundamental type, e.g.:

```
const char* stringWithNumber = "5297";
```

  - The content of *stringWithNumber* can be interpreted as a number of type int.

- How can we extract the number's value from the cstring as an int? We could scan the cstring and generate the int:

ASCII Code	Symbol
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9

```
int result = 0;

const int nDigits = std::strlen(stringWithNumber);
for (int i = 0; i < nDigits; ++i) {
        // The weight of the letter is its 1-based position in the string
        int digitWeight = nDigits - i - 1;
        // The digitValue is the ASCII code of the char minus 48 (See the ASCII table!).
        int digitValue = stringWithNumber[i] - 48;
        // The digitValue multiplied with the scale of the digitWeight contributes to the result:
        result += digitValue * std::pow(10, digitWeight);
}
// result = 5297
```

- But this solution has many downsides:
  - It doesn't work for negative numbers.
  - It only works with decimal numbers and not, e.g., with hexadecimal numbers.
  - The code uses a lot of magic numbers.
  - It will also hurt the DRY principle, because such processing of a cstring to get contained int is required quite often!

Cstrings – Parsing – Part I

- In the just presented code we read, interpret and process an object (const char*) and convert it into another object (int).

- The operation-chain "read-interprete-calculate-convert" is called parsing among programmers (and grammar-lawyers).
  - The function *std::atoi()* (<cstdlib>), "alpha to integer", parses an int from a cstring (this line replaces our former code completely):

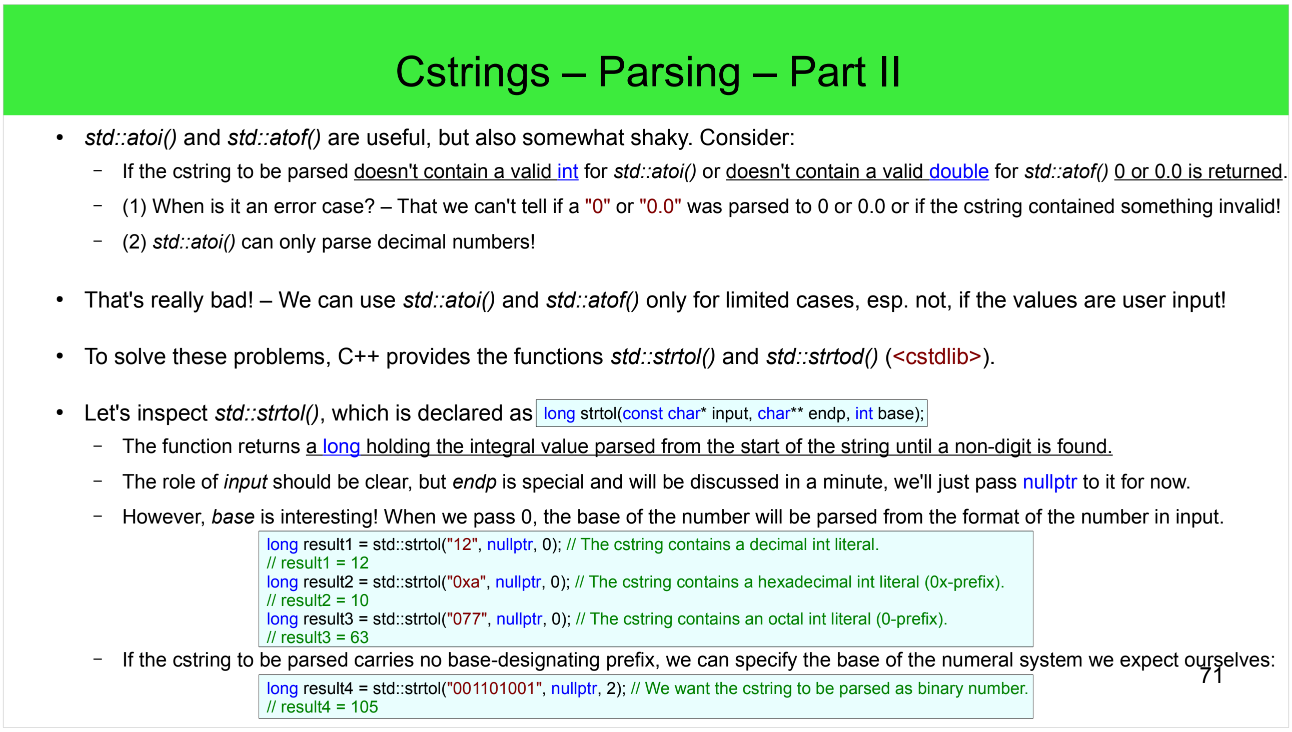
```
int result = std::atoi(stringWithNumber);
// result = 5297
```

  - C++ has no implicit conversion from cstrings to fundamental types, because cstrings are unrelated to fundamental types.
  - Hence, the cstring must be parsed and the contained int extracted. – Parsing is a relatively costly operation!

- Similar to *std::atoi()*, we can use the function *std::atof()* (<cstdlib>), "alpha to float" to parse double values from cstrings:

```
// A cstring that can be interpreted as double:
const char* aDouble = "1.786";
// Parse the double from aDouble:
double theDouble = std::atof(aDouble);
// aDouble = 1.786
```

- Behavior of *std::atoi()* and *std::atof()*:
  - Positive case: If the passed cstring can be parsed to an int for *std::atoi()* or a double for *std::atof()* this value will be returned.
  - If the cstring to be parsed doesn't contain a valid int for *std::atoi()* or doesn't contain a valid double for *std::atof()* 0 or 0.0 is returned.
  - If the argument to be parsed is no valid cstring, e.g. a char[], which not 0-terminated, the behavior is undefined.
  - If the converted value cannot to be represented by an int for *std::atoi()* or a double for *std::atof()*, the behavior is undefined.



Cstrings – Parsing – Part II

- *std::atoi()* and *std::atof()* are useful, but also somewhat shaky. Consider:
  - If the cstring to be parsed doesn't contain a valid int for *std::atoi()* or doesn't contain a valid double for *std::atof()* 0 or 0.0 is returned.
  - (1) When is it an error case? – That we can't tell if a "0" or "0.0" was parsed to 0 or 0.0 or if the cstring contained something invalid!
  - (2) *std::atoi()* can only parse decimal numbers!

- That's really bad! – We can use *std::atoi()* and *std::atof()* only for limited cases, esp. not, if the values are user input!

- To solve these problems, C++ provides the functions *std::strtol()* and *std::strtod()* (<cstdlib>).

- Let's inspect *std::strtol()*, which is declared as `long strtol(const char* input, char** endp, int base);`
  - The function returns a long holding the integral value parsed from the start of the string until a non-digit is found.
  - The role of *input* should be clear, but *endp* is special and will be discussed in a minute, we'll just pass nullptr to it for now.
  - However, *base* is interesting! When we pass 0, the base of the number will be parsed from the format of the number in input.

```
long result1 = std::strtol("12", nullptr, 0); // The cstring contains a decimal int literal.
// result1 = 12
long result2 = std::strtol("0xa", nullptr, 0); // The cstring contains a hexadecimal int literal (0x-prefix).
// result2 = 10
long result3 = std::strtol("077", nullptr, 0); // The cstring contains an octal int literal (0-prefix).
// result3 = 63
```

  - If the cstring to be parsed carries no base-designating prefix, we can specify the base of the numeral system we expect ourselves:

```
long result4 = std::strtol("001101001", nullptr, 2); // We want the cstring to be parsed as binary number.
// result4 = 105
```

- *std::strtol()* allows bases in [2, 32] and 0 for default behavior.

Cstrings – Parsing – Part III

- *std::strtol()*'s parameter *endp* deals with the sophisticated error handling *std::strtol()* offers, mind that *endp* is a char**!

- Let's inspect following examples:

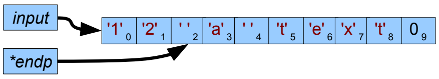
```
long strtol(const char* input, char** endp, int base);
```

Parsing a valid int	`int result1 = std::atoi("12 a text");`
	`// result1 = 12`
		`char* endp1;`
		`long result1 = std::strtol("12 a text", &endp1, 0);`
		`// result1 = 12, *endp1 = " a text"`

Parsing an invalid int	`int result2 = std::atoi("a text");`
	`// result2 = 0`
		`char* endp2;`
		`long result2 = std::strtol("a text", &endp2, 0);`
		`// result2 = 0, *endp2 = "a text"`

Parsing a valid 0	`int result3 = std::atoi("0"); // Also 0!`
	`// result3 = 0`
		`char* endp3;`
		`long result3 = std::strtol("0", &endp3, 0);`
		`// result3 = 0, *endp3 = ""`

  - Remember: the function returns a long holding the integral value parsed from the start of the string until a non-digit is found.
  - The parameter *endp* is a pointer to a substring, thus a char**. – It will either
    - point to a substring of input, which could not be parsed as int and the long parsed in front of the substring is returned, or
    - point to an empty cstring, i.e. *std::strlen(*endp) == 0*, if input contained only a valid int and the fully parsed int is returned.



Input is "12 a text", so 12 is returned and " a text" is the remaining "unparsed" cstring.

- Now we can distinguish error cases from ordinary parsing of 0:
  - If a "0" was parsed the result is 0 and *endp* points to an empty string (length = 0).
  - If the contained value is no parseable int, the result is also 0, but *endp* points to input.

Cstrings – Parsing – Part IV

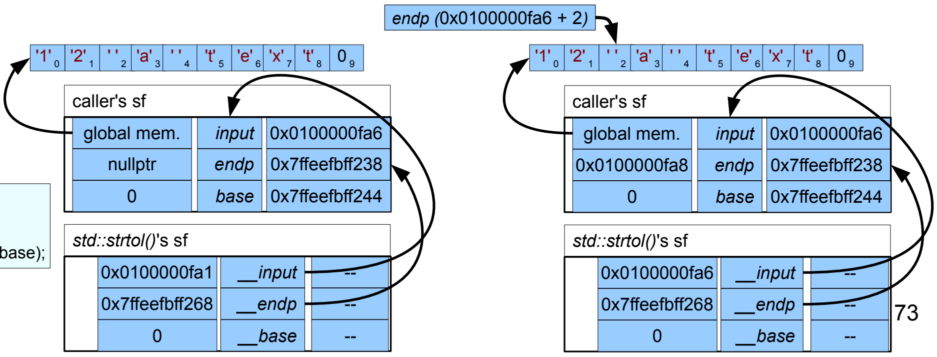
- *Why do we have to pass a char** to std::strtol()*'s?

```
namespace std {
        long strtol(const char* __input, char** __endp, int __base) {
                // pass
        }
}
```

- The answer is, that *std::strtol()* actually returns two pieces of information, the parsed long and the remaining unparsed.

- A C++ function can only return one value, but via params, we can write many values into the caller's sf with pointers!
  - And this is exactly, what *std::strtol()* does!

```
const char* input = "12 a text";
char* endp = nullptr;
int base = 0;
long result = std::strtol(input, &endp, base);
```



'1' 0 | '2' 1 | ' ' 2 | 'a' 3 | ' ' 4 | 't' 5 | 'e' 6 | 'x' 7 | 't' 8 | 0 9

caller's sf

global mem.	*input*	0x0100000fa6
nullptr	*endp*	0x7ffeefbff238
0	*base*	0x7ffeefbff244

std::strtol()'s sf

0x0100000fa1	*__input*	--
0x7ffeefbff268	*__endp*	--
0	*__base*	--

endp (0x0100000fa6 + 2)

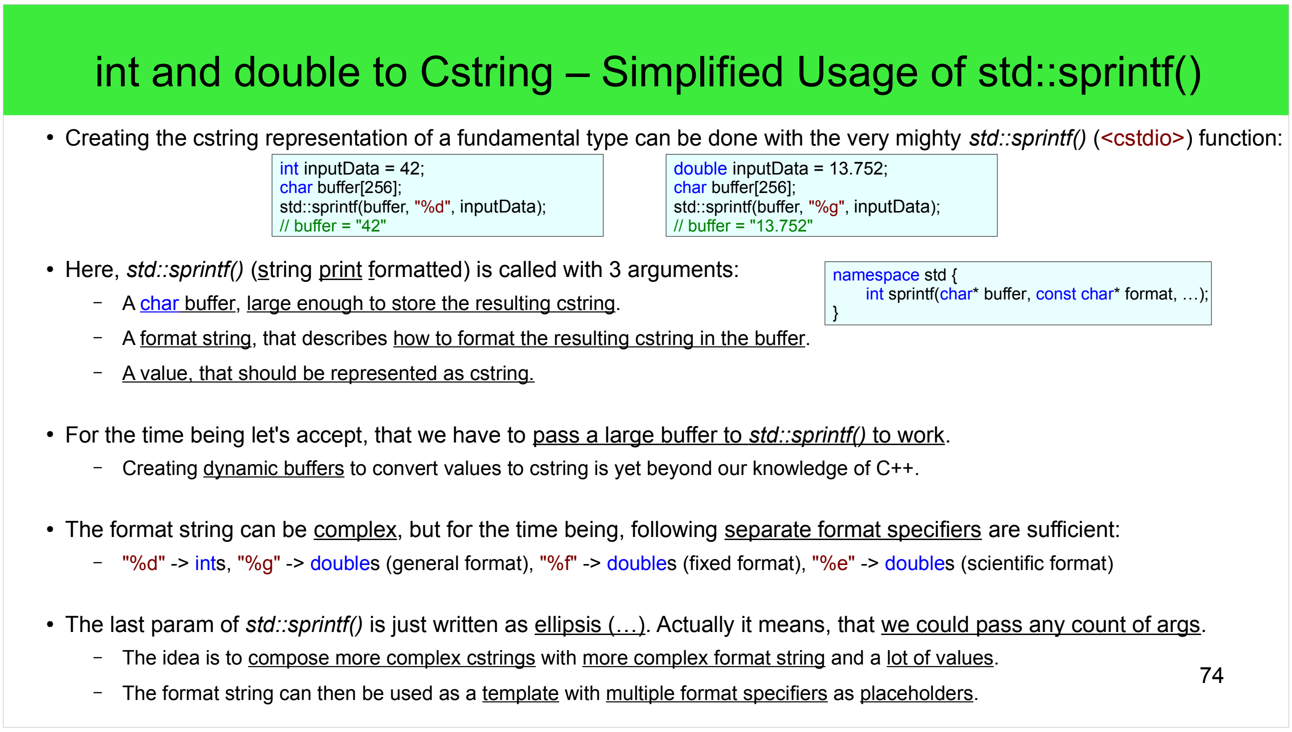
'1' 0 | '2' 1 | ' ' 2 | 'a' 3 | ' ' 4 | 't' 5 | 'e' 6 | 'x' 7 | 't' 8 | 0 9

caller's sf

global mem.	*input*	0x0100000fa6
0x0100000fa8	*endp*	0x7ffeefbff238
0	*base*	0x7ffeefbff244

std::strtol()'s sf

0x0100000fa6	*__input*	--
0x7ffeefbff268	*__endp*	--
0	*__base*	--



int and double to Cstring – Simplified Usage of std::sprintf()

- Creating the cstring representation of a fundamental type can be done with the very mighty *std::sprintf()* (<cstdio>) function:

```
int inputData = 42;
char buffer[256];
std::sprintf(buffer, "%d", inputData);
// buffer = "42"
```

```
double inputData = 13.752;
char buffer[256];
std::sprintf(buffer, "%g", inputData);
// buffer = "13.752"
```

- Here, *std::sprintf()* (string print formatted) is called with 3 arguments:
  - A char buffer, large enough to store the resulting cstring.
  - A format string, that describes how to format the resulting cstring in the buffer.
  - A value, that should be represented as cstring.

```
namespace std {
	int sprintf(char* buffer, const char* format, ...);
}
```

- For the time being let's accept, that we have to pass a large buffer to *std::sprintf()* to work.
  - Creating dynamic buffers to convert values to cstring is yet beyond our knowledge of C++.
- The format string can be complex, but for the time being, following separate format specifiers are sufficient:
  - "%d" -> ints, "%g" -> doubles (general format), "%f" -> doubles (fixed format), "%e" -> doubles (scientific format)
- The last param of *std::sprintf()* is just written as ellipsis (...). Actually it means, that we could pass any count of args.
  - The idea is to compose more complex cstrings with more complex format string and a lot of values.
  - The format string can then be used as a template with multiple format specifiers as placeholders.

- *std::sprintf()* returns the count of chars written into the buffer (excl. the 0-termination). On failure, it returns a negative number.
- As stated on the slide, we have to discuss dealing with dynamic memory in C++, which is basis of more real-world handling of cstrings created at run time.

Thank you!