(6) Basics of the C++ Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

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

Arrays have a Compile Time fixed Length

- We begin our discussion by remembering a limitation of arrays: they must have a length, which is fixed at compile time.

- Actually, we cannot define a C++ array with a length, which is set during run time, e.g. via user input

```
int count = 0;
std::cout<<"How many numbers do you want to enter?"<<std::endl;
std::cin>>count;
if (0 < count) {
	int numbers[count]; // Invalid (in C++)! The symbol count must be a compile-time constant!
	for (int i = 0; i < count; ++i) {
		std::cout<<"Enter number "<<(i + 1)<<": "<<std::endl;
		std::cin>>numbers[i];
	}
}
```

Good to know
Actually, some compilers, e.g. g++ accept this code, and the code also works. Those compilers have been extended to offer variable length arrays (VLAs), which can be created with a dynamic length. – But VLAs are no standard C++17 feature!

- More exactly, C++ does not support automatic arrays with a dynamic length!

The Correct Way of creating dynamic Arrays in Code

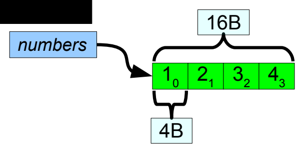
- Instead of using an automatic array, we have to create a dynamic array, where we can specify a dynamic length:

```cpp
#include <cstdlib>

int count = 0;
std::cout<<"How many numbers do you want to enter?"<<std::endl;
std::cin>>count;
if (0 < count) {   // Create a properly sized block in heap. The function std::malloc() returns a generic
                   // pointer (void*) and we have to cast this generic pointer to the type we need.
    int* numbers = static_cast<int*>(std::malloc(sizeof(int) * count));
    if (numbers) { // Check, whether std::malloc() was successful.
        for (int i = 0; i < count; ++i) { // Loop over the dynamically created array:
            std::cout<<"Enter number "<<(i + 1)<<": "<<std::endl;
            // Use the block like an ordinary array, e.g. with the []-operator:
            std::cin>>numbers[i];
        }
        std::free(numbers); // When done with the array, it must be freed!
    }
}
```

Terminal

```
NicosMBP:src nico$ ./main
How many numbers do you want to enter?
4
Enter number 1:
1
Enter number 2:
2
Enter number 3:
3
Enter number 4:
4
NicosMBP:src nico$
```



- The core new thing in this code is the usage of the functions *std::malloc()* and *std::free()* from <cstdlib>:
  - *std::malloc()* takes over the creation of the array, accepting the length, which was specified by the user at run time.
  - *std::malloc()* returns a void* representing the created array, this generic pointer must be cast to an int* (to make it usable as int[]).
  - Because *numbers* is no longer an automatic array, we must remove it from memory manually with *std::free()*.
  - With automatic arrays, the memory consumed by the array was given back to the system automatically, when it left its scope.

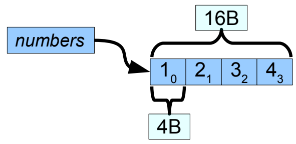
4

- Consequently check the success of *std::malloc*()! – I.e. program defensively!
- Why do we need to cast here?
  - This is the first time we really need to cast. – Here we need to cast a pointer to memory of raw type to a pointer to memory of the type we need.
  - The void* represents a pointer to memory of unknown type; casting is required to get a type with which we can work. A void* is irrelevant by itself, we can't even dereference it. The only thing we can do with it is comparing it with other pointers or 0.
  - Objects of type void* are generally provided by the C++ (memory) system and must be "given back" to the system, after the programmer no longer needs it.
  - Interestingly the conversion from void* to another pointer type is seen as conversion between related types, so a static_cast is sufficient.

Graphical Representation of Stack- and Heap-Memory

- The arrays we have discussed up to now are so called automatic arrays.
  - Automatic arrays must have a length, which is defined at compile time!

```
const int length = 4;
int numbers[length];
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;
numbers[3] = 4;
```

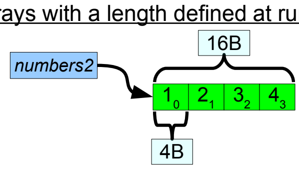


  - The memory occupied by an automatic array resides on the stack, therefor, here we use blue boxes to depict them.

- The new way to create arrays, i.e. via *std::malloc()*, allows us to create arrays with a length defined at run time:

```
#include <cstdlib>

int length = 4; // Assume value from run time ...
int* numbers2 = static_cast<int*>(std::malloc(sizeof(int) * length));
if (numbers) {
	numbers2[0] = 1;
	numbers2[1] = 2;
	numbers2[2] = 3;
	numbers2[3] = 4;
	std::free(numbers2);
}
```



  - The memory occupied by an array created dynamically with *std::malloc()* resides in the heap memory,
  - Hence we use green boxes to depict portions of heap memory.
  - Because this array is not automatically served back to the system, we have to call *std::free()* to remove it manually from memory.

- In this course we make a distinction between heap and free store. This distinction is not defined by the C++ standard. Instead it is used colloquially to tell the memory managed by *std::malloc()*/*std::free()* from that managed by new/delete, because they could work in different memory locations each.

std::malloc() in Focus

- In opposite to automatic memory, acquiring memory from the heap is a more explicit process.
  - The explicit acquiring of heap-memory is called memory allocation, hence the function named *std::malloc()* for memory allocation.

```
// <cstdlib>
namespace std {
	void* malloc(size_t size);
}
```

Good to know
from latin *allocare* or *locare*: "to place something"

- *std::malloc()*'s parameter *size* expects the size of the memory to allocate from the heap in byte (= sizeof(char)).

- However, the returned value is somewhat special: The returned value is of type void*. – Why that?
  - void* represents the generic pointer type. A void* can point to any kind of memory from any source (stack or heap).
  - *std::malloc()*'s function is to allocate a block of raw memory from the heap. Raw means that any type could be stored in this block.
  - In C++ we represent blocks of memory as arrays: an array is a list of elements, that reside in a contiguous block of memory.
  - The elements of this raw array have no type, they are just raw bytes (having sizeof(char)).
  - An array of elements without type, cannot decay to a pointer of a certain type, but can decay to the generic pointer-type void*.

- The important take away for the time being: dealing with heap memory means dealing with pointers.

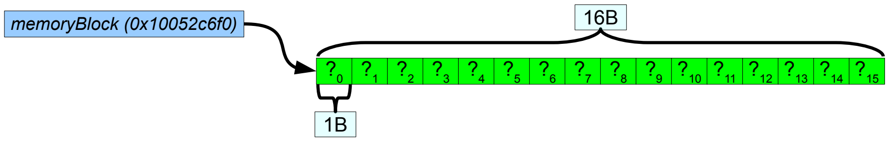
- The other take away is, that this pointer in of type void*. – What can we do with it? How to "type" it?

std::malloc() and the generic Pointer Type (void*)

- All right, so we get a void* from *std::malloc()*. – How to progress from that?

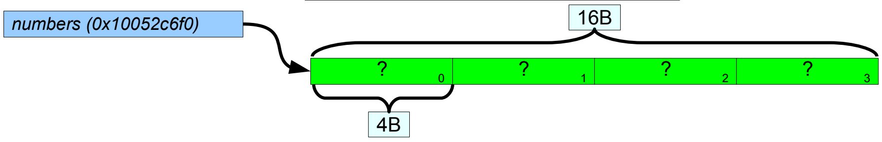
- The first thing we have to do is telling the void*-referenced memory its type, currently, it is just a byte-array:

```
void* memoryBlock = std::malloc(sizeof(int) * 4);
```



- We do this by casting the void* to a specific type:

```
int* numbers = static_cast<int*>(memoryBlock);
```



  - The cast to a specific pointer type tells *numbers*, that the referenced memory should be treated like an array of int!
  - i.e. each element in *numbers* actually is an int, and has a size of 4(B).
  - The cast works like a contact lens, which focuses the pointer from a blurry raw memory to an int array!
  - Contact lens: Notice, that *memoryBlock* and *number* refer the same address, *numbers* just has a "sharper" view as int array.

Checking Heap-Memory Allocation and Freeing

- After *std::malloc()* returns a pointer, we have to check it for being a nullptr!
  - If *std::malloc()* didn't succeed, it'll return a nullptr.
  - *std::malloc()* fails, if there is no more heap memory for the requested allocation.
  - *std::malloc()* fails, if there is no contiguous portion in the heap memory to fit the requested block's in size.
  - (The nullptr-check can be done on the void* or the cast pointer "wearing the contact lens".)

```
int* numbers = static_cast<int*>(std::malloc(sizeof(int) * 4));
if (numbers) { // If the allocation succeeded ...
	// ...
}
```

- If *std::malloc()* succeeded, i.e. its result is not nullptr, we can use the result like any other array.
  - Esp. we can use the []-operator to access/modify elements. I.e. the []-operator can directly be used on a pointer (which is no void*).

```
if (numbers) { // If the allocation succeeded ... we can safely use numbers:
	numbers2[0] = 1;
	numbers2[1] = 2;
	numbers2[2] = 3;
	numbers2[3] = 4;
	// ...
}
```

- When we are done with the dynamic array *numbers*, we have to remove it from the heap with *std::free()*.
  - Because *std::free()* doesn't work with nullptrs, we must only call free, if *std::malloc()* succeeded!

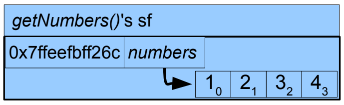
```
if (numbers) { // If the allocation succeeded ... we can safely use numbers:
	// ...
	std::free(numbers); // free numbers from memory
}
```


- The []-operator dereferences the pointer, but void* cannot be dereferenced, because the type to which is points is undefined!
- When we discussed *std::memcpy()*, we already had to deal with functions accepting void*s. But this time we have to work with returned void*s explicitly, which forces us to cast to other types, because void*s are pointing to raw memory. *std::malloc()*, *std::free()* and *std::memcpy()* are all low level functions.
- Can we assume that *std::malloc()* returns a pointer to a gap-free portion in the heap?
  - As far as our current knowledge is concerned: yes! It is required, because *std::malloc()* can be used to create arrays, and arrays need to represent contiguous blocks of memory.

Example: Why dynamic Memory is needed: Returning Arrays – Part I

- Of course the example we discussed up to now does not reflect a real life scenario.
- Consider following situation, in which we're going to return an array from a function.

```
int* getNumbers() {                 // Defining a function that returns a pointer to a
    int numbers[] = {1, 2, 3, 4}; // locally defined array (created on the stack).
    return numbers;               // This pointer points to the 1st item of values.
}
```

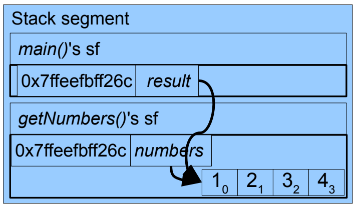


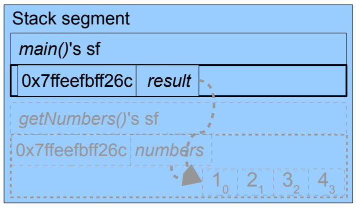
  - When *getNumbers()* is called, it creates an int[4] on the stack.
  - Then *getNumbers()* returns this array, it decays to int*, hence the return type.

- When *main()* calls *getNumbers()* it stores the returned int* in the local variable *result*.

```
int* result = getNumbers();                          // Semantically wrong! result points to
std::cout<<"2. number is: "<<result[1]<<std::endl;  // already discarded memory.
// The array "values" is gone away, result points to its scraps, probably rubbish!
```

  - This won't work! When *getNumbers()* returns, its stackframe will be popped from the stack.
  - When the sf is popped, all its locals will be removed from the stack as well, hence we call it autom. memory!
  - *result* in *main()* will contain an invalid address pointing to unknown content!

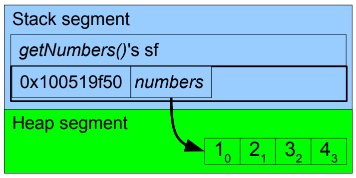




Example: Why dynamic Memory is needed: Returning Arrays – Part II

- One way to make returning of array from a function possible is using dynamic memory.
- Let's rewrite *getNumbers()* to create the array in question in the dynamic memory.

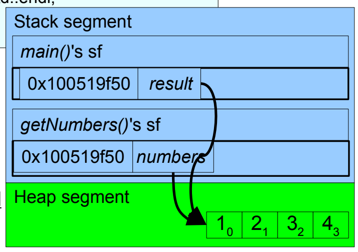
```
int* getNumbers() {
	int* numbers = static_cast<int*>(std::malloc(sizeof(int) * 4));
	if (numbers) {
		numbers[0] = 1; numbers[1] = 2; numbers[2] = 3; numbers[3] = 4;
	}
	return numbers;
}
```

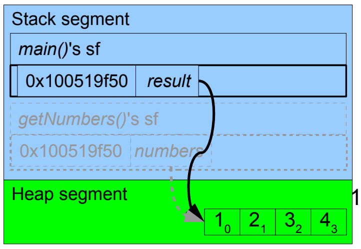


  - When *getNumbers()* is called, it creates an int[] on the heap.
- When *main()* calls *getNumbers()* it stores the returned int* in the local *result*.

```
int* result = getNumbers();
if (result) {
	std::cout<<"2. number is: "<<result[1]<<std::endl;
	std::free(result);
}
```

  - This works! When *getNumbers()* returns, its stackframe will be popped from the stack.
  - But *getNumbers()*' return value is still a valid pointer!
  - The locals in *getNumbers()* are popped from the stack, but the heap memory survived this!





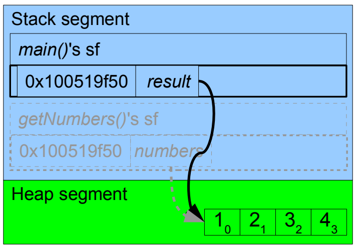
- Once again: Consequently check the success of *std::malloc()*! – Program defensively!
- RAII means that a resource, which requires e.g. dynamic memory or other operating system resources, will be initialized and freed analogous to the lifetime of a variable. – Practically it means that a resource from dynamic memory can be controlled by a variable on the stack! – This can be implemented with user defined types.

Example: Why dynamic Memory is needed: Returning Arrays – Part III

- The graphical representation of the stack- and heap-situation shows that the heap is kind of shared among functions:

```
int* getNumbers() {
	int* numbers = static_cast<int*>(std::malloc(sizeof(int) * 4));
	if (numbers) {
		numbers[0] = 1; numbers[1] = 2; numbers[2] = 3; numbers[3] = 4;
	}
	return numbers;
}
```

```
int* result = getNumbers();
if (result) {
	std::cout<<"2. number is: "<<result[1]<<std::endl;
	std::free(result);
}
```

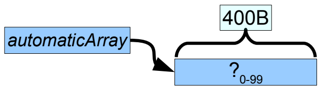


- Notice following points:
  - We have a pointer, that is a variable on the stack, but this time it holds the address to memory on the heap!
  - When we have an array from the heap, we can access/modify its elements with the []-operator, but other aspects have changed:
  - Using the heap comes for a price: since no stackframe controls the lifetime of memory, we have to control it manually!
  - I.e. we have to remove the memory we have alloced on the heap ourselves using explicit code, namely calling *std::free()*.
  - Also mind, that we have to check, whether allocation succeeded with a nullptr-check!

Wrap up: automatic and dynamic Arrays in Code

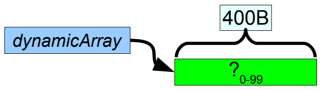
- Creation of automatic arrays:

```
int automaticArray[100];
// This is a simple automatic array with a compile time constant size of 100.
```



- Creation and freeing of dynamic arrays:

```
int* dynamicArray = static_cast<int*>(std::malloc(sizeof(int) * 100));
// - Indeed the syntax looks weird, not even similar to the autoArray example.
// - The type of the variable we assign to is int-pointer.
// - The function std::malloc() is used to create a raw memory-block in heap.
// - std::malloc() returns a generic pointer (void*) to this raw memory-block, it does so, because
//   it doesn't know, what the programmer wants to do.
// - So, as programmers we need to tell C/C++ that we want to use the allocated memory
//   block as int-array, therefor we need to cast the generic pointer (void*) to int-pointer. - We
//   "put contact lenses on".
if (dynamicArray) {
	std::free(dynamicArray); // Free the dynamically created array in the right place.
}
```



- Dynamically created arrays can have the length 0!

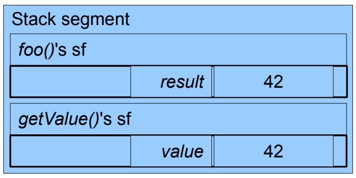
- Decay makes automatic arrays indistinguishable from dynamic arrays as function parameters!
  - E.g. we cannot blindly free a pointer passed to a function in the code of this function.
  - Specific implementation strategies and good documentation is required!

- Whether objects are stored on the heap or the stack is an implementation detail in many other languages, not so in C++, because the programer explicitly decides, where an object will be created. (Annotated C# Language Reference, Anders Hejlsberg et. al.)

But, how does returning of "normal" Variables compute?

- Arrays can't be returned by value, so they can't be copied! – This is exactly the problem we solved with the heap!
- But taking a look back, how is the situation with non-arrays, i.e. "normal" automatic locals?
- When a local variable is returned from a function, it will be simply copied.

```
int getValue() { // Just returns an int:
	int value = 42;  // value is an automatic variable on the stack.
	return value;    // Returns value.
}
```



  - When *getValue()* ends, *value* will be popped from *getValue()*'s stack.
  - But returning the content of *value* will push its content to the stackframe of the caller function. – This is the copy activity!
  - => In effect *value* will be copied to its caller when *getValue()* returns.

```
void foo() { // Calls getValue():
	int result = getValue();    // The returned int was pushed on foo()'s stack by getValue()
	                            // and will be copied into result.
}
```

- When using automatic memory, there is no connection between the sfs, other than copying activities.
  - Esp. addresses are irrelevant, because *value* is getting copied and not shared somehow!

13

- Arrays can also be generated as static arrays instead of automatic arrays. A static array can be returned from a function. The lifetime of a static array is not restricted to a function's local scope.

Stack vs. Heap: It's not a Mystery, just two Concepts

- The stack is a conceptual place, where local (auto) variables reside.
  - This is a little oversimplification, but each function has its own portion of the stack, the so called stackframe.
  - The lifetime of a stack variable is visible by its scope (i.e. it is automatic: auto).
  - The stack is controlled by hardware and owned by hardware.

- The heap is a conceptual place, where all dynamic contents reside.
  - All functions of a program generally use the same heap.
  - Dynamic content must be created by the programmer manually.
  - The heap is controlled by software, the heap manager (*std::malloc()*, *std::free()* etc.).
  - There is always an "entity" that is in charge for the allocated heap memory.
  - This "entity" is responsible for explicit freeing the allocated heap memory.
  - In the end, the lifetime of a dynamic content is controlled by the entity's programmer.

- We'd try to control as less memory as possible manually: using the stack is preferred!

- What is a scope?
- See RAII!

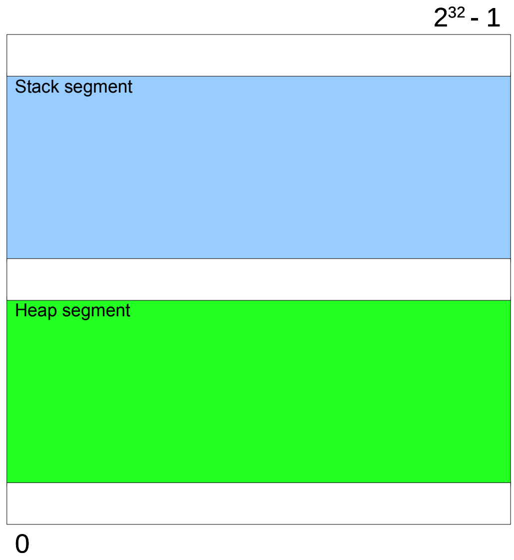
Stack vs. Heap: In Memory (RAM)

- There is the illusion that all the machine's memory is owned by a program.
  - This is not true, each program uses its own portion of memory respectively.
  - But in the following graphics we stick to this illusion.

- The memory is segmented, different segments have different functions.

- Esp. the stack and heap segment often have special "geometric" properties:
  - The heap segment resides at lower addresses than the stack segment.
  - The addresses of subsequent stack variables are decreasing.
    - This is called "descending stack".
  - The stack evolves/grows to lower, the heap to greater addresses.
    - In fact, stack and heap grow to meet each other halfway!
  - Compared to the stack, the heap is very big: dynamic content is typically bigger than automatic content (e.g. local variables).

Stack vs. Heap: Conventional Locations in Memory



- Why 2^{32}-1?
  - Well 2^{32} is 4,294,967,296 (ca. 4GB), but we have to subtract 1 in order to get space for the address 0!

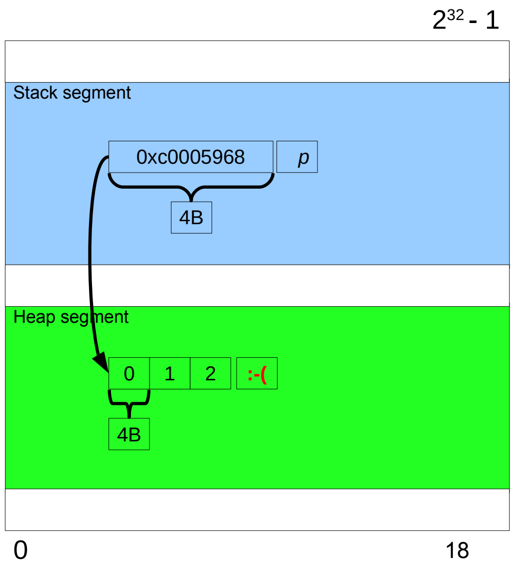
The lost Pointer to the Heap Memory in Code

```
void f(int count) {
	// Allocating an array of three ints. f() is in charge of the dynamic content, to which
	// p points!
	int* p = static_cast<int*>(std::malloc(sizeof(int) * count));
	if (p) { // Check std::malloc()'s success.
		for (int i = 0; i < count; ++i) {
			p[i] = i;
		}
	}
	// The auto variable p will go out of scope and will be popped from stack. But the
	// referred dynamic content is still around!
}
//-------------------------------------------------------------------------------------------------------------------
// Calling f():
f(3);
// Oops, nobody did free the dynamic content, to which p was pointing to! Now there is
// no pointer to the dynamic content in avail. This is a semantic error, a memory leak of
// sizeof(int) * 3. The compiler will not see any problem here!
```

The lost Pointer to the Heap Memory in Memory

```
void f(int count) { // Allocating an array of three ints.
	int* p = static_cast<int*>(std::malloc(sizeof(int) * count));
	if (p) { // Check std::malloc()'s success.
		for (int i = 0; i < count; ++i) {
			p[i] = i;
		}
	}
}
```

```
// Calling f():
f(3);
// After f() did run: oops! The pointer to the allocated three
// ints is lost, the allocated memory is orphaned. We have a
// memory leak of 12B.
```



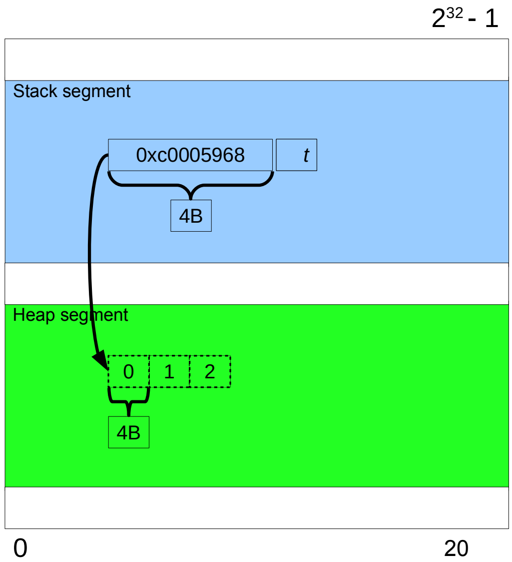
How to handle dynamic Content responsibly in Code

```
/**
 * Returns an int[] holding the first 'count' natural numbers.
 *
 * @param count the count of natural numbers to put into the result
 * @return a pointer to the int[] created on the heap, the caller must nullptr-check it
 * and std::free() it!
 */
int* g(int count) {
	// Allocating an array of three ints. g() is in charge of the dynamic content, to which
	// p points to!
	int* p = static_cast<int*>(std::malloc(sizeof(int) * count));
	if (p) { // Check std::malloc()'s success.
		for (int i = 0; i < count; ++i) {
			p[i] = i;
		}
	}
	// Returning p. Then g()'s caller is in charge of the dynamic content!
	return p; // The stack variable p will go out of scope and will be popped from the
		  // stack. But the referred dynamic content is still around!
}
//-----------------------------------------------------------------------------------------------------------------
// Calling g():
int* t = g(3);
if (t) { // Fine! The returned pointer will be checked and freed correctly.
	std::free(t);
}
```

Handling dynamic Content responsibly in Memory

```
/**
 * Returns an int[] holding the first 'count' natural numbers.
 *
 * @param count the count of natural numbers to put into the result
 * @return a pointer to the int[] created on the heap, the caller must nullptr-check it
 * and std::free() it!
 */
int* g(int count) { // Allocating an array of three ints.
	int* p = static_cast<int*>(std::malloc(sizeof(int) * count));
	if (p) { // Check std::malloc()'s success.
		for (int i = 0; i < count; ++i) {
			p[i] = i;
		}
	}
	return p; // This time: return p!
}
```

```
int* t = g(3);		// Call g() and receive the pointer.
if (t) {		// Check and free the content (i.e. the
	std::free(t);	// memory from the heap).
}
// The local variable t is still on the stack.
```



Potential Problems with Heap Memory

- It is needed to check, whether allocation was successful!
- It is needed to free dynamic content in the right place manually.
  - We have to keep in mind that there is no garbage collection in C/C++.
  - So, we should not forget to free dynamically created content!
  - We should free dynamically created content as early as possible, but not too early!
  - We should not free dynamically created content more than once!
  - We should not free dynamically created content that we don't own.
- It's impossible to distinguish pointers to the stack from pointers to the heap.
  - Don't free pointers to the stack (i.e. pointers not from the heap)! -> It will result in undefined behavior!
- Secondary problem: we can't use array initializers when creating dynamic memory!
- Wherever functions deal with dynamically content, it must be documented where this memory must be freed!
  - Who's the owner of the memory?
  - Who's in charge of freeing the memory?

More Information about Heap Memory

- There exist two further useful functions to deal with heap memory (<cstdlib>):
  - *std::realloc()* resizes a given block of heap memory.
  - *std::calloc()* allocates a block of size * count and initiates all "items" with 0.
  - The returned value must be checked for nullptr and freed with *std::free()* respectively.

- Free store in C++:
  - In C++ the heap segment can also be used as free store.
  - The operators new and delete act as interface to the free store.
  - These operators represent C++' way to dynamically allocate/deallocate user defined types.
  - In general, C's heap memory and C++' free store are incompatible.

- Often 3rd party libraries invent own allocation/deallocation mechanisms:
  - to deal with platform specialities,
  - and/or to encapsulate usage of dynamic contents.

- *std::realloc*():
  - Present items will be preserved up to the passed length.
  - Possibly a pointer to another location in the heap is returned, rendering the passed address invalid.
  - The returned pointer should be checked for 0, before it is assigned to the passed pointer variable. Don't do it like so: *p = std::realloc(p, 5000)*! If reallocation didn't succeed the memory to which the passed pointer *p* refers won't be touched.

Cstring Limitations: Assignment and Extension

- Assigning arrays to copy their content, esp. cstrings is not allowed in C++:

```
char aString[] = "Malta";
const char aString2[] = "Teneriffa";
aString = aString2; // Invalid! Array type 'char [6]' is not assignable
```

  - What we want to have: copy the content of *aString2* to *aString*. – Mind, that arrays offer no copy semantics on assignment!
  - Even if this would work at compile time, it wouldn't at run time, because *aString* has too few elements!
  - (At least we defined *aString* being non-const, so that copying would at least be possible logically...)

- We cannot "extend" a cstring by appending or adding further cstrings to it:

```
char aString[] = "Weyland Yutani";
aString += " at LV-426"; // Invalid! Invalid operands to binary expression ('char [15]' and 'const char [11]')
```

  - Even if this would work at compile time, it wouldn't at run time, because *aString* has too few elements!
  - (At least we defined *aString* being non-const, so that extending/appending would at least be possible logically...)

- In both cases show similar problems:
  - We have to allocate memory on demand to create cstrings, which are larger than the the original cstrings.
  - The dilemma: we know the effective lengths of these cstrings only at run time.
  - The solution: we must create this memory dynamically in the heap.

Dynamic Cstrings – Putting Heap Memory to work with Cstrings

- As cstrings are char arrays underneath, they share the limits of other arrays:
  - (1) We cannot resize/extend or assign cstrings.
  - (2) We cannot return an automatic cstring variable from a function.

- These limitations can be solved by usage of the heap memory:
  - To overcome limitation (1): Create a dynamically sized char[] on the heap to hold a modified/resized copy of the original cstring.
  - To overcome limitation (2): Copy a cstring into a dynamically sized char[] on the heap and return the pointer from a function.
  - Of course somebody must free the dynamically created memory, when it is no longer required.

- Sidebar: Peculiarities of cstrings, not directly shared with other arrays:
  - Cstrings are 0-terminated, so the very last char[] item contains a 0.
  - We can get a cstring's length (*std::strlen()*), this is impossible with other arrays.
  - As cstrings are const char-arrays, we can't modify them.
  - Indeed we can return a cstring literal from a function!

- Why is it possible to return a literal cstring from a function?
  - Cstring literals are stored in the static portion of memory (they are an example of a static arrays, which were mentioned earlier)! We'll discuss static memory in a future lecture.

Dynamic Cstrings – The Roles of const char[] and char[]

- When we need to create an array that must be filled afterwards, we cannot use arrays with const elements.
  - Instead we need arrays as modifiable buffers.
  - This is needed, cause we need to assign to the elements, in order to modify the content!

- So, cstrings are of type const char[], their matching buffer type is char[].
  - The buffers we allocate dynamically for char-based cstrings are always of type char[].

- Where cstring functions accept const char*-params, we can safely pass char[] buffers; they will be decayed to const char*.

- To sum up (char-based cstrings):
  - Cstrings are of type const char[], they're not modifiable
  - Cstring buffers are of type char[], they're modifiable.

Dynamic Cstrings – Non-Dynamic Assignment/Copying

- For completeness, this code shows a naïve way to implement cstring assignment/copying with an automatic array:

```
const char* const initialString = "Malta";
const char* const aString2 = "Teneriffa";

char aString[256]; // A large buffer, with compile time fixed length.
std::strcpy(aString, initialString); // The "assignment"!
std::cout<<aString<<std::endl;

std::strcpy(aString, aString2);  // The "assignment"!
std::cout<<aString<<std::endl;
```

Terminal

```
NicosMBP:src nico$ ./main
Malta
Teneriffa
NicosMBP:src nico$
```

- The code copies *initialString* and then *aString2* into *aString*.

- However, the problem is that *aString* has a length, which is fixed at compile time.
  - Problem 1: If a cstring we copy into *aString* is smaller than the capacity of *aString*, we waste memory!
  - Problem 2: If a cstring we copy into aString is larger than the capacity of *aString*, we overwrite memory and risk a buffer overrun!

- In most scenarios we don't know, how large the buffer must be to hold the effective result.
  - We could define a very large (automatic) buffer as shown above, but this is neither efficient nor safe.
  - The preferred way is to calculate the required buffer length exactly and create it dynamically!
  - On the following slides, we'll see some examples, were buffers must be created dynamically. – We begin with rewriting this code!

Dynamic Cstrings – Dynamic Assignment/Copying

- All right. Copying a cstring using the heap is a pretty simple case, but requires a lot of code:

```
const char* const initialString = "Malta";
const char* const aString2 = "Teneriffa";

char* aString = static_cast<char*>(std::malloc(sizeof(char) * (std::strlen(initialString) + 1)));
if (aString) {
        std::strcpy(aString, initialString); // The "assignment"!
        std::cout<<aString<<std::endl;
        std::free(aString);
}
aString = static_cast<char*>(std::malloc(sizeof(char) * (std::strlen(aString2) + 1)));
if (aString) {
        std::strcpy(aString, aString2);  // The "assignment"!
        std::cout<<aString<<std::endl;
        std::free(aString);
}
```

Terminal

```
NicosMBP:src nico$ ./main
Malta
Teneriffa
NicosMBP:src nico$
```

- The code copies *initialString* and then *aString2* into *aString*, *aString* is dynamically sized to fit its new content respectively.
  - The code bloat makes the check of the correct allocation after *std::malloc()* and the need to free the allocated memory afterwards!

- The assignment/copying operation is a rather unimpressive call to *std::strcpy()*.
  - Notice: we could also have used *std::memcpy()* to do the copying, but this had required more size-calculation efforts in the code!

Dynamic Cstrings – Modified Copy

- A rich set of functions dealing with individual chars can be found in <cctype>.
- E.g. there are functions to change the casing of a char (these functions await an int/char and return an int/char):
  - *std::tolower()* and *std::toupper()*, their result must be cast to char for presentation.

```
char ch = 'X';
// If ch is no upper case letter, the same char will be returned.
std::cout<<ch<<" as lower case: "<<static_cast<char>(std::tolower(ch))<<std::endl;
// >X as lower case: x
```

- With *std::toupper()* we can create an "upper-cased" variant of another cstring by copying and modifying that copy:

```
const char* oldText = "home"; // The original plain c-string.
// Create a buffer in the heap, large enough to hold the original cstring. The buffer needs a size of:
// sizeof(char) * (count of chars/letters + one byte for the 0-termination).
char* newText = static_cast<char*>(std::malloc(sizeof(char) * (std::strlen(oldText) + 1)));
if (newText) { // Check std::malloc()'s success.
	// Loop over the original cstring and store the upper case variant of each char and the
	// 0-termination into newText at the same index.
	for (int i = 0; i < std::strlen(oldText) + 1; ++i) {
		newText[i] = std::toupper(oldText[i]); // Modifies the non-const buffer.
	}
	std::cout<<"The modified text: "<<newText<<std::endl;
	// >The modified text: HOME"
	std::free(newText); // Free the buffer.
}
```

  - Strategy: create a new buffer dynamically with the length of the original cstring + 1 (0-termination).
  - Iterate over the buffer and the original cstring and set the upper-cased char of the original cstring at the new buffer's index.

- If we pass a non-letter char to *std::tolower()*/*std::toupper()* the passed char will just be returned.
- *std::tolower()*/*std::toupper()* return either the lower case or upper case variant of the passed character or the character itself, if there is no lower/upper variant. Therefor it is safe to pass the 0-termination, because it'll survive the function call and the 0-termination will be copied accordingly.
- Mind, that we can modify the copy of the cstring, because it resides in a modifiable/non-const char-buffer.

Dynamic Cstrings – String Concatenation – Part I

- Often it is required to dynamically compose cstrings by extending or concatenating other cstrings.

```
const char* s1 = "Weyland Yutani", *s2 = " at " *s3 = "LV-426";
```

  - Concatenation (concat) means, that multiple cstrings are "jammed together" to build another cstring:

"Weyland Yutani" ∘ " at " ∘ "LV-426" = "Weyland Yutani at LV-426"

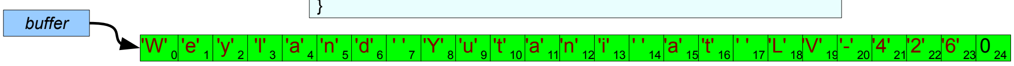
  - Concat is not an addition, because a "real" mathematical/numerical addition is a commutative operation, string concat is surely not!

- The effective result we want to reach is to have a new cstring, which holds *s1*, *s2* and *s3* as concatenated cstring
  - To make this work, we create a char[]-buffer, which is large enough to hold all chars of *s1*, *s2* and *s3* and the 0-termination!
  - If we'd like to have this buffer exactly sized, we must use a dynamically allocated char[]!

```
// Calculate the length of the resulting cstring:
std::size_t  countOfChars = std::strlen(s1) + std::strlen(s2) + std::strlen(s3);
// Allocate buffer with the exact size, sufficient for our situation (the lengths + 1):
char* buffer = static_cast<char*>(std::malloc(sizeof(char) * (countOfChars + 1)));
```

  - Then, we have to concatenate the cstrings *s1*, *s2* and *s3* into the buffer with the function ***std::strcat()*** from <cstring>:

```
if (buffer) { // Check std::malloc()'s success then do the concatenation:
        buffer[0] = 0; // Set the 0-char at first char in the buffer.
        std::strcat(buffer, s1);
        std::strcat(buffer, s2);
        std::strcat(buffer, s3);
        std::cout<<buffer<<std::endl;
        // >Weyland Yutani at LV-426"
        std::free(buffer); // Free buffer.
}
```

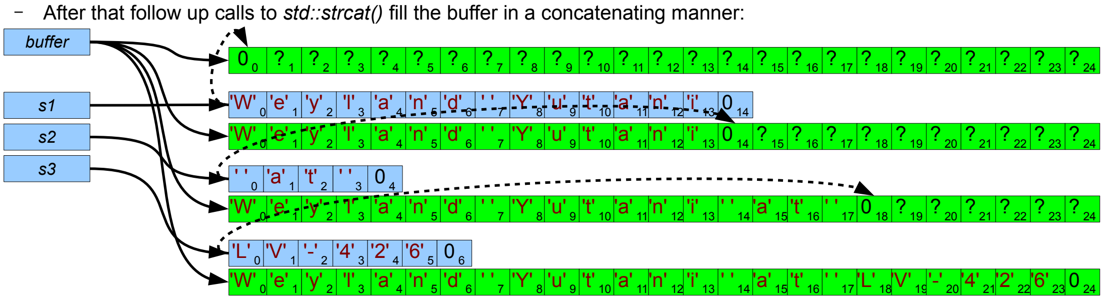


Dynamic Cstrings – String Concatenation – Part II

- `char* strcat(char* destination, const char* source);` :
  - *std::strcat()* reads *destination* starting with the first char, until it finds the first 0-char.
  - Then it copies all chars from *source* into buffer until *source*'s 0-termination is found incl. *source*'s 0-termination.

```
char* buffer = static_cast<char*>(std::malloc(sizeof(char) * (countOfChars + 1)));
if (buffer) {
        buffer[0] = 0; // Set the 0-char at first char in the buffer.
        std::strcat(buffer, s1);
        std::strcat(buffer, s2);
        std::strcat(buffer, s3);
        std::free(buffer);
}
```

- Right after the *buffer* is created and *buffer[0]* set to 0, we have an empty array with a capacity of 25 elements.
  - After that follow up calls to *std::strcat()* fill the buffer in a concatenating manner:



- The memory representation is not precise on showing, how *buffer* changes. – Following the animation one could assume, that *buffer* is *std::free()*'d and created anew from the heap for multiple times, but instead it is modified in place by *std::strcpy()*!
- *std::strcat()* returns *destination*.

Dynamic Cstrings – Formatting – Part I

- Besides "simple" concatenation of cstrings we can also <u>format</u> cstrings dynamically.
  - Formatting is useful, if we want to guarantee a special and reusable look of a text with minimal effort.
  - Formatting is only about <u>formatting of bare text data</u>, <u>it has nothing to do with font styles</u> like "bold" or "underlined"!

- To format cstrings, we will use the function *std::sprintf()* from <cstdio>.
  - *std::sprintf()* awaits a <u>buffer</u> into which the formatted result will be stored, a <u>format string</u> and the <u>values to be formatted</u> as cstring:

```
char fullText[256];
std::sprintf(fullText, "Duration: %dms %s", 3 * 4, "speed-test");
// fullText = "Duration: 12ms speed-test"
```

- The format string works like a <u>template</u>, which defines the <u>result cstring with placeholders</u>, which are replaced by values.
  - The placeholders in the <u>format string template</u> are denoted by the <u>%-character</u> and a so called <u>format specifier</u>.
  - The placeholders are effective to the values in the order they are written: **%d** to 3 *4 and **%s** to "speed-test":

```
std::sprintf(fullText, "Duration: %dms %s", 3 * 4, "speed-test");
```

  - Here we have two placeholders **%d** and **%s**, which address the arguments 3 * 4 and *testName* respectively.
  - **%d** tells *std::sprintf()* to handle its corresponding value (3 * 4) as decimal integer, **%s** denotes a cstring ("speed-test").

- *std::sprintf()* can principally deal with an <u>unlimited amount of placeholders</u> and can replace an unlimited amount of values.
  - But, this would mean that *std::sprintf()* deals with an unlimited amount of argument, how can that work?

- String-formatting: integrals and floats are promoted to int/double, when passed to …, therefor %d and %g can accept either.

Dynamic Cstrings – Formatting – Part II

```
std::sprintf(fullText, "Duration: %dms %s", 3 * 4, "speed-test");
```

- Here we call *std::sprintf()* with 4 args, which can be split into buffer and format as first 2 args and a list of values.
  - The signature of *std::sprintf()* looks like this:

```
int sprintf(char* buffer, const char* format, ...);
```

```
char fullText[256];
std::sprintf(fullText, "Duration: %dms %s", 3 * 4, "speed-test");
```

  - As can be seen the parameter format stores the format string and the variable arity parameter args stores the remaining arguments.

- *std::sprintf()* can be used as alternative to *std::strcat()* to concat cstrings with less code:

```
const char* s1 = "Weyland Yutani", *s2 = " at ", *s3 = "LV-426";
std::size_t  countOfChars = std::strlen(s1) + std::strlen(s2) + std::strlen(s3);
char* buffer = static_cast<char*>(std::malloc(sizeof(char) * (countOfChars + 1)));

if (buffer) {
        buffer[0] = 0;
        std::strcat(buffer, s1);
        std::strcat(buffer, s2);
        std::strcat(buffer, s3);
        std::cout<<buffer<<std::endl;
        // >Weyland Yutani at LV-426"
        std::free(buffer);
}
```

```
const char* s1 = "Weyland Yutani", *s2 = " at ", *s3 = "LV-426";
std::size_t  countOfChars = std::strlen(s1) + std::strlen(s2) + std::strlen(s3);
char* buffer = static_cast<char*>(std::malloc(sizeof(char) * (countOfChars + 1)));

if (buffer) {
        std::sprintf(buffer, "%s%s%s", s1, s2, s3);
        std::cout<<buffer<<std::endl;
        // >Weyland Yutani at LV-426"
        std::free(buffer);
}
```

Excursus: Variable Length Argument Lists in C

- Esp. C is well known for its feature of functions, which can cope with variable argument lists (vargs):

```c
#include <stdarg.h>
// C variadic function example
int sum(int nNumbers, ...) {
	int sum = 0;
	va_list args;
	va_start(args, nNumbers);
	for (int i = 0; i < nNumbers; ++i) {
		nSum += va_arg(args, int);
	}
	va_end(args);
	return sum;
}
```

```c
const int full_sum = sum(3, 1, 2, 3);
// full_sum = 6
```

Good to know
All standard C/C++ functions have the calling convention __cdecl. Only __cdecl allows variable argument lists, because only the caller knows the argument list and only the caller can then pop the arguments. __stdcall functions execute a little bit faster than __cdecl functions, because the stack needs not to be cleaned on the callee's side (i.e. within a __stdcall function).

  - Featured by the ubiquitous function *std::sprintf()*, applied via the ...-operator (ellipsis-operator).
  - The mandatory vargs' first argument must be interpreted, to guess how many vargs follow.
  - Vargs are harmful: It's a way to introduce security leaks through stack overruns. (Just call *sum(**4**, 1, 2, 3)* and see what happens.)

- How does it work? The vargs-features works very near the metal:
  - The compiler calculates the required stack depending on the arguments and decrements the stack pointer by the required offset.
  - As arguments are laid down on the stack from right to left, *nNumbers* is on offset 0.
  - Then *nNumbers* is analyzed and the awaited offsets are read from the stack. Here an offset of, e.g., 4B for each int passed to *sum()*.

33

- When we think about *std::sprintf()*, we don't pass the count of arguments via an explicit argument, instead *std::sprintf()* analyzes the passed format-string and determines the amount of placeholders.
- The calling convention __cdecl is a C/C++ compiler's default, __stdcall is the calling convention of the Win32 API, because it works better with non-C/C++ languages. __cdecl requires to prefix a function's name with an underscore when calling it (this is the exported name, on which the linker operates). A function compiled with __stdcall carries the size of its parameters in its name (this is also the exported name). – Need to encode the size of bytes or the parameters: If a __cdecl function calls a __stdcall function, the __stdcall function would clean the stack and after the __stdcall function returns the __cdecl function would clean the stack again. - The naming of the exported symbol of __stdcall functions allow the caller to know how many bytes to "hop", because they've already been removed by the __stdcall function. Carrying the size in a function name is not required with __cdecl, because the caller needs to clean the stack. - This feature allowed C to handle variadic functions with __cdecl (nowadays the platform independent variadic macros can be used in C and C++).
- Other calling conventions:
  - pascal: This calling convention copies the arguments to the stack from left to right, the callee needs to clean the stack.
  - fastcall: This calling convention combines __cdecl with the usage of registers to pass parameters to get better performance. It is often used for inline functions. The callee needs to clean the stack. The register calling convention is often the default for 64b CPUs.
  - thiscall: This calling convention is used for member functions. It combines __cdecl with passing a pointer to the member's instance as if it was the leftmost parameter.
- In this example the RV (EAX on x86) register can only store values of 4B. In reality the operation can be more difficult.
  - For floaty results the FPU's stack (ST0) is used.
  - User defined types (e.g. structs) are stored to an address that is passed to the function silently.
- It is usually completely different on micro controllers.

Formatting Cstring – more Examples

- For string formatting we can use a lot of format specifiers different from %s or %d, esp. for formatting floaty values.

- The general format for floaty values, such as double is specified with the **%g** placeholder as format specifier:

```
char buffer[256];
double value = 2234567890;
std::sprintf(buffer, "value: %g", value);
// buffer = "2.23457e+09"
```

  - The default precision, which is effective on all digits is 6. We can change this to a precision of 10, with the **%.10g** format specifier:

```
std::sprintf(buffer, "value: %.10g", value);
// buffer = "2234567890"
```

- With the format specifier **%e** we specify the scientific format for the corresponding floaty value:

```
double value2 = 2.23456789;
std::sprintf(buffer, "value2: %e", value2);
// buffer = "2.234568e+00"
```

```
// Scientific notation with precision of 2:
std::sprintf(buffer, "value2: %.2e", value2);
// buffer = "2.23e+00"
```

- With the format specifier **%f** we specify the fixed point format for the corresponding floaty value:

```
std::sprintf(buffer, "value2: %f", value2);
// buffer = "2.234568"
```

```
// Floating point value rounded to two digits:
std::sprintf(buffer, "value2: %.2f", value2);
// buffer = "2.23"
```


- *std::sprintf()*:
  - Superfluous arguments in the varargs will be ignored.
  - Too few arguments result in undefined behavior.
  - If the format-string and the arguments don't match the behavior is undefined.
  - There are different ways to control the format in a more detailed fashion. Besides the precision, additional flags control alignment and padding, also the width and the length can be controlled. – But the resulting format strings quickly become difficult to read.
  - %p can be used to output pointers.
  - Downsides:
    - It should be said, that *std::sprintf()* must also be used with care, because one can easily overwrite the passed buffer by getting format specifiers wrong. – It can be tricky to calculate the correct size of the buffer.
    - Currently, C++ does not allow to "invent" new format specifiers of own UDTs. However, this issue was addressed by the possibility to overload operator-<< for own UDTs.

Thank you!