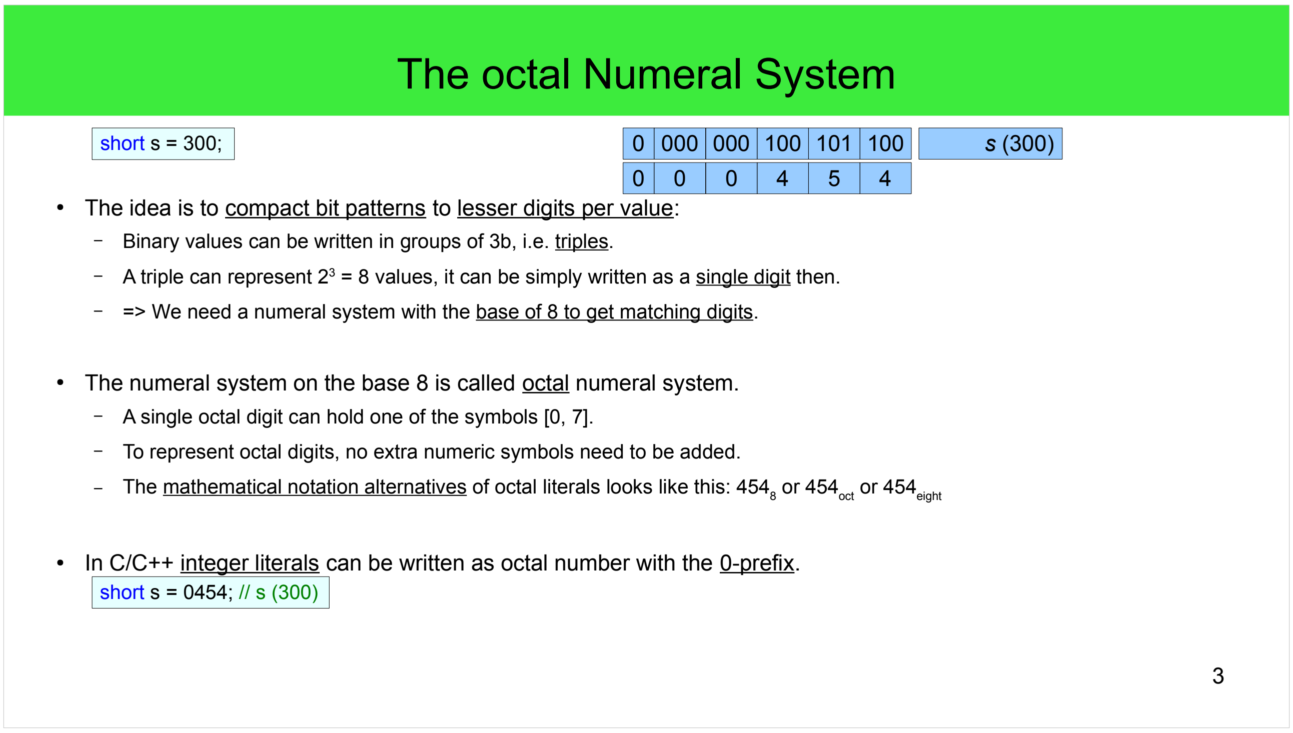
(8) Basics of the C++ Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

- Sources:
  - Bjarne Stroustrup, The C++ Programming Language
  - Charles Petzold, Code
  - Rob Williams, Computer System Architecture
  - Jerry Cain, Stanford Course CS 107



The octal Numeral System

```
short s = 300;
```

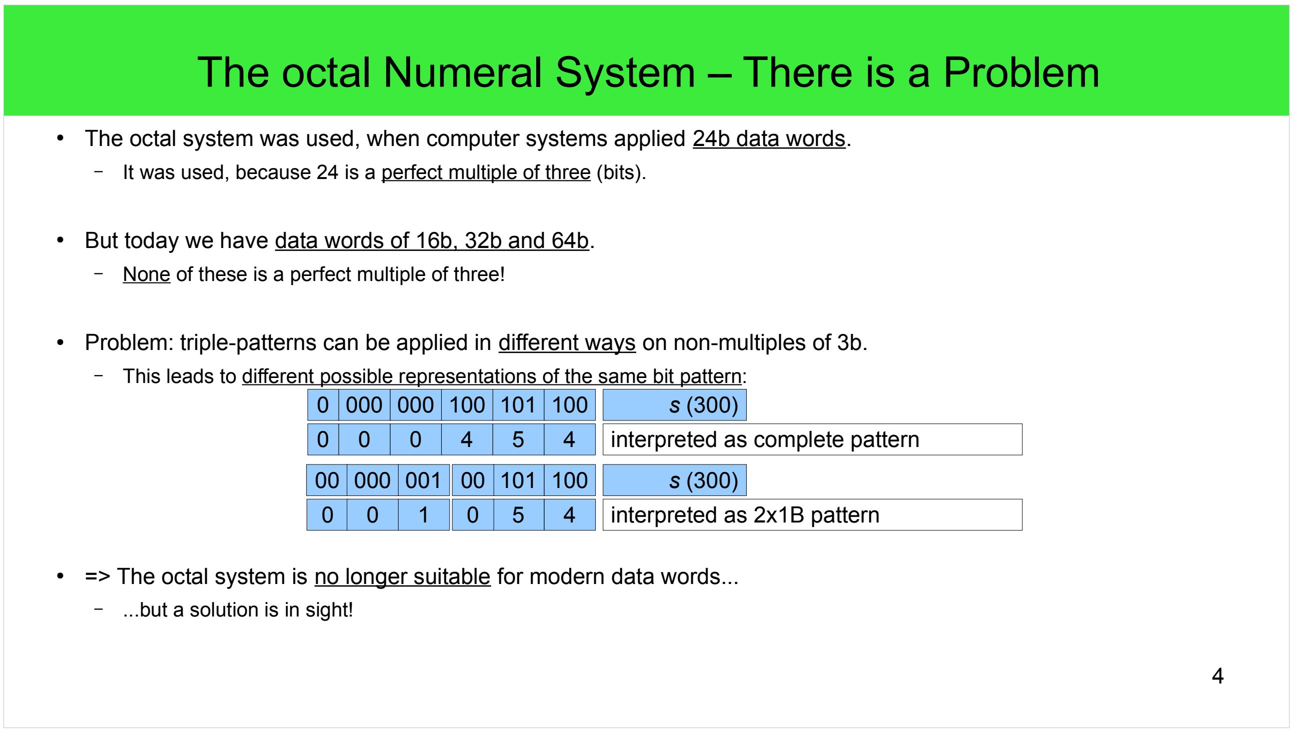
0	000	000	100	101	100	*s* (300)
0	0	0	4	5	4	

- The idea is to compact bit patterns to lesser digits per value:
  - Binary values can be written in groups of 3b, i.e. triples.
  - A triple can represent $2^3 = 8$ values, it can be simply written as a single digit then.
  - => We need a numeral system with the base of 8 to get matching digits.
- The numeral system on the base 8 is called octal numeral system.
  - A single octal digit can hold one of the symbols [0, 7].
  - To represent octal digits, no extra numeric symbols need to be added.
  - The mathematical notation alternatives of octal literals looks like this: 454_8 or 454_{oct} or 454_{eight}
- In C/C++ integer literals can be written as octal number with the 0-prefix.

```
short s = 0454; // s (300)
```

3

- Why do we introduce the octal numeral system here?
  - Binary numbers can be quickly transformed into octal digits and vice versa. The conversion of groups of 3b makes this possible.
- It is used in some computer systems to enhance the readability of values under certain circumstances (24b data words) – It is esp. used in Unix' file permission system.



The octal Numeral System – There is a Problem

- The octal system was used, when computer systems applied 24b data words.
  - It was used, because 24 is a perfect multiple of three (bits).
- But today we have data words of 16b, 32b and 64b.
  - None of these is a perfect multiple of three!
- Problem: triple-patterns can be applied in different ways on non-multiples of 3b.
  - This leads to different possible representations of the same bit pattern:

0	000	000	100	101	100	*s* (300)
0	0	0	4	5	4	interpreted as complete pattern

00	000	001	00	101	100	*s* (300)
0	0	1	0	5	4	interpreted as 2x1B pattern

- => The octal system is no longer suitable for modern data words...
  - ...but a solution is in sight!

- Sometimes the octal numeral system does also play a role in encoding of (serial) data communication, where 3b-wise encoding can be found often.
- Notice, that after the 2x1B pattern application only the rightmost triples retain their representation.

The hexadecimal Numeral System

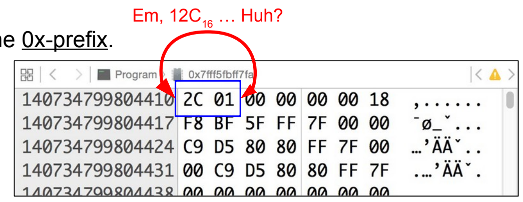
```
short s = 300;
```

0000 0001		0010 1100		s (300)
0	1	2	C	

- In previous pictures, binary values have been written in groups of 4b very often.
  - We call these "half bytes" nibbles or tetrads. Two tetrads make one octet – the byte!
  - A tetrad can represent 2^4 = 16 values, it can simply be written as a single digit then.
  - => We need a numeral system with the base of 16 to get matching digits.
- The numeral system on the base 16 is called hexadecimal (short "hex") numeral system.
  - A single hexadecimal digit can hold one of the symbols [0, 9] or [A, F] (or [a, f], the case does not matter).
  - And a single hexadecimal digit can be directly represented by a tetrad.
  - The mathematical notation alternatives of literals look like this: $12C_{16}$ $12C_{hex}$ or $12C_{sixteen}$, 12Ch, x12C, #12C or $12C
- In C/C++ integer literals can be written as hex number with the 0x-prefix.

```
short s = 0x12C; // s (300)
```

- In the depicted memory view there is the 300 in hex! But... something is wrong: the digits are somehow "mirrored"?
  - We'll discuss the "effect" of byte order in short!



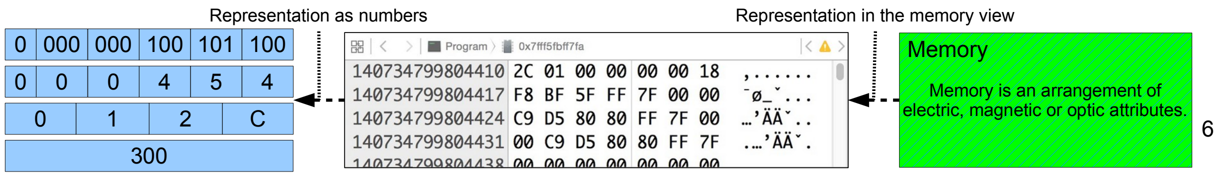
```
140734799804410 2C 01 00 00 00 00 18   ,......
140734799804417 F8 BF 5F FF 7F 00 00   ¯ø_ˇ...
140734799804424 C9 D5 80 80 FF 7F 00   …'ÄÄˇ..
140734799804431 00 C9 D5 80 80 FF 7F   .…'ÄÄˇ.
140734799804438 00 00 00 00 00 00 00
```

5

- Sometimes in long binary numbers the tetrads are separated by dashes to enhance the readability.
- Why do we introduce the hexadecimal numeral system here?
  - Binary numbers can be quickly transformed into hexadecimal digits and vice versa. The conversion of groups of 4b makes this possible.
  - Esp. we need it to understand the contents of the memory view (i.e. memory dumps or "raw" views of data (e.g. network traffic) in the IDE's debugger), which mostly uses a hexadecimal presentation.
- C++14 introduced binary integer literals with the 0b-prefix. – Before that, programmers could use special libraries like Boost, which provide macros to use binary literals. (Binary integer literals can be defined in Java 7 with the 0b-prefix as well.)
- Where do we need the hexadecimal and octal numeral system?
  - Well, the decimal numeral system uses values that are often not dividable by a power of two. As binary numbers can be written in blocks, which are powers of two, the hexadecimal and octal numeral system can compress the projection of numbers very nicely.
  - Memory addresses are almost always written as hexadecimal numbers.
  - Media access control (MAC) addresses are typically written as six dash-separated two-digit hex numbers.
- Do you know other numeral systems?
  - There exists the simple unary system, in which the count of symbols simply represents the number in question. E.g. $1_1 = 1_{10}$, $11_1 = 2_{10}$, $111_1 = 3_{10}$ This system is know from tally sheets (German "Strichlisten", sometimes "Bierdeckelnotation"): **I** = 1_{10}, **II** = 2_{10}, **卌** = 5_{10}
  - Esp. important: the roman system and the sexagesimal system, based on the value 60 used e.g. by the babylonians, who also introduced the 360°, taken, because of a year having about 360 days (still used today to display clock time and angles).
  - The DNA is the biological 4-system using the symbols cytosine (C), guanine (G), adenine (A), and thymine (T).
- The hexadecimal representation of numbers will be used occasionally in this course.

Numbers and the Memory

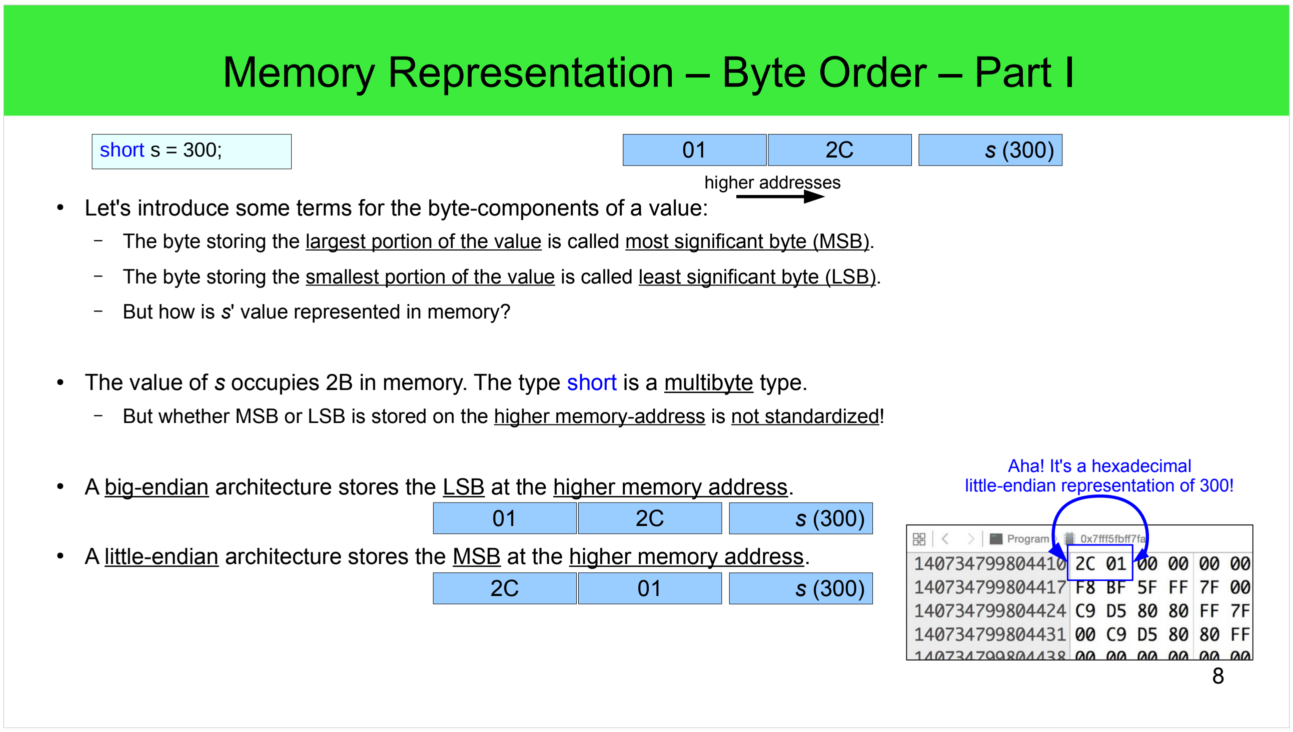
- I've just overheard this statement "Numbers are stored hexadecimally in a computer's memory!" – No! That's nonsense!
- The next "correction" was "Oh, I mean numbers are stored binary in a computer's memory!" – No, it is also nonsense!
- Folks! A computer stores data as tiny pieces of electric attributes, e.g. voltage, in electronic circuits. That's all!
  - Permanent memory also uses magnetic (e.g. magnetic tape), optic (e.g. DVD) or meanwhile also electric (flash) attributes.
- If these attributes have discretely distinguishable values, these are interpreted as different states, often just "on" and "off".
- Computer scientists, programmers and software interpret "on" and "off" as information.
  - Software can interpret series of "on" and "off" information-pieces and groups as numbers.
  - Sure, the same number can be written in the binary, octal, hexadecimal or decimal numeral system!


Representation as numbers
0 | 000 | 000 | 100 | 101 | 100
0 | 0 | 0 | 4 | 5 | 4
0 | 1 | 2 | C
300
Representation in the memory view
Program
0x7fff5fbff7fa
Memory
Memory is an arrangement of electric, magnetic or optic attributes.

```
140734799804410 2C 01 00 00 00 00 18   ,......
140734799804417 F8 BF 5F FF 7F 00 00   ¯ø_ˇ...
140734799804424 C9 D5 80 80 FF 7F 00   …’ÄÄˇ..
140734799804431 00 C9 D5 80 80 FF 7F   .…’ÄÄˇ.
140734799804438 00 00 00 00 00 00 00
```

Why do I need to understand Memory Representation?

- Today, programming is very comfortable.
- Understanding "the metal" helps understanding pointers.
- Understanding "the metal" helps understanding errors.
- Understanding the past helps understanding presence and future.
- Don't believe in magic! Try to understand memory!



Memory Representation – Byte Order – Part I

```
short s = 300;
```

01	2C	*s* (300)

higher addresses →

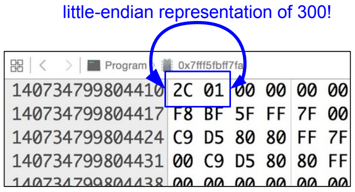
- Let's introduce some terms for the byte-components of a value:
  - The byte storing the largest portion of the value is called most significant byte (MSB).
  - The byte storing the smallest portion of the value is called least significant byte (LSB).
  - But how is *s*' value represented in memory?
- The value of *s* occupies 2B in memory. The type short is a multibyte type.
  - But whether MSB or LSB is stored on the higher memory-address is not standardized!
- A big-endian architecture stores the LSB at the higher memory address.

01	2C	*s* (300)

- A little-endian architecture stores the MSB at the higher memory address.

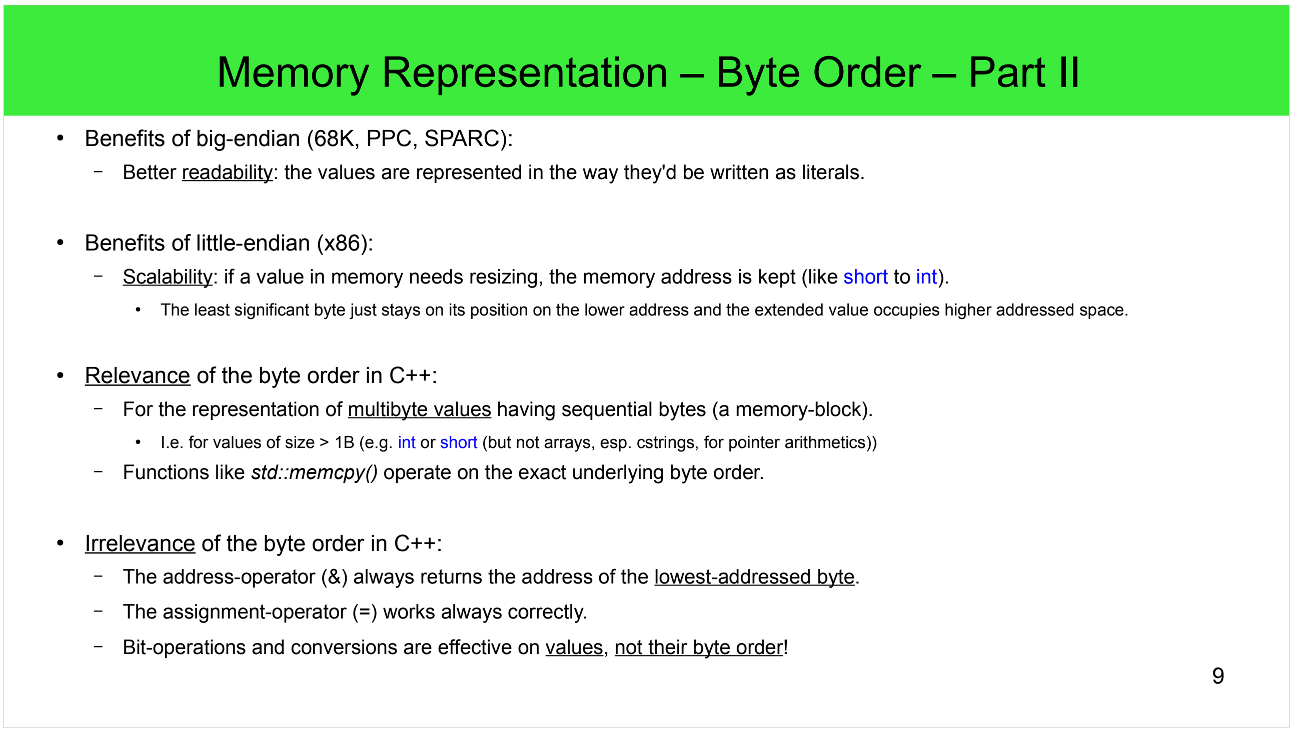
2C	01	*s* (300)

Aha! It's a hexadecimal little-endian representation of 300!



```
140734799804410 2C 01 00 00 00 00
140734799804417 F8 BF 5F FF 7F 00
140734799804424 C9 D5 80 80 FF 7F
140734799804431 00 C9 D5 80 80 FF
140734799804438 00 00 00 00 00 00
```

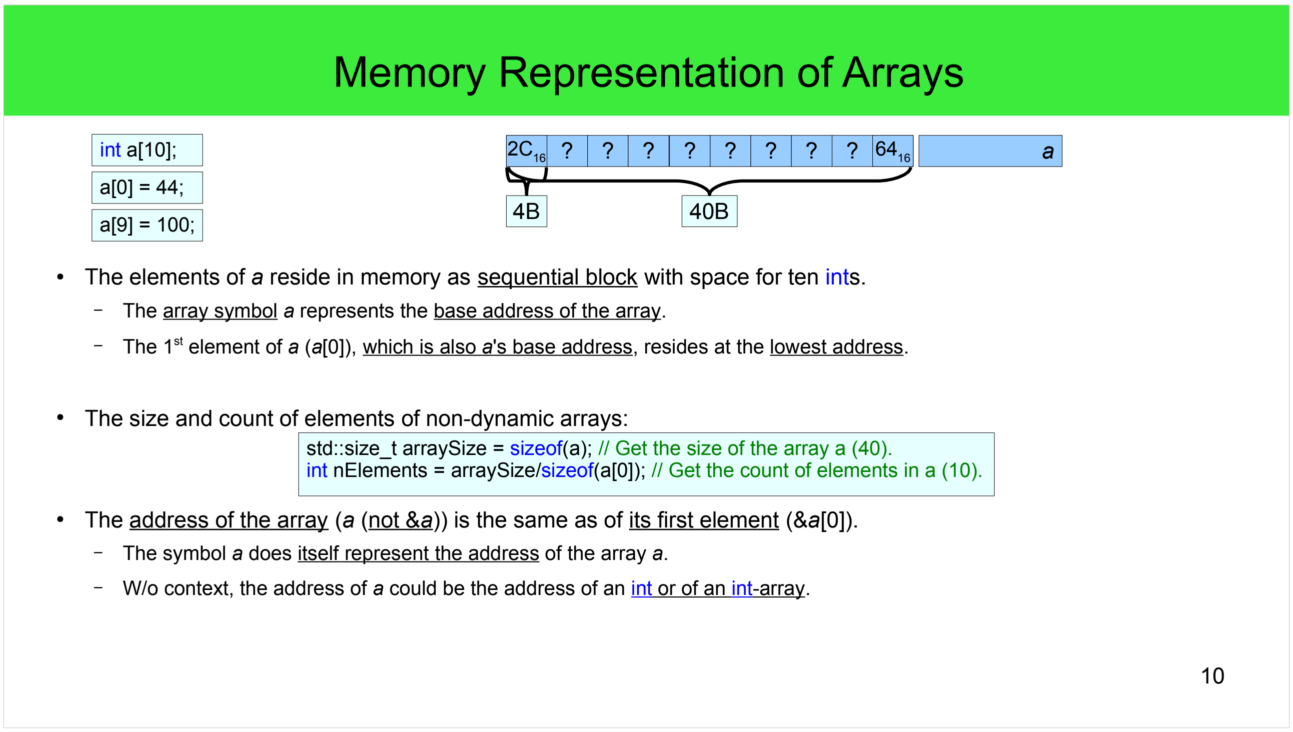
- In English, "normal" numbers are written in big-endian order, but read in little-endian order, i.e. the most significant parts of a value are read first (thousands before hundreds etc.).
- Another example for the significance of the parts of a value: for the clock time represented as hh:mm:ss the hh-portion is the most significant part of the clock time value. → The clock time is encoded as big-endian.
- Binary numbers, although written in big-endian, i.e. "normal number notation", they are read from right to left to avoid misunderstandings.
- Interestingly date values are written in completely different orders depending on the culture/country/locale:
  - US: MM/DD/YYYY (middle-endian)
  - Germany: DD.MM.YYYY (little-endian)
  - ISO-8601: YYYY-MM-DD (big-endian)
- The terms big/little-endians stem from the book "Gulliver's Travels" (Jonathan Swift). In the story the folk of Lilliput is required to open boiled eggs on the small end (little-endians), whereas in the rival kingdom of Belfuscu the folk is required to open them on the big end (big-endians). – The picture was used by the engineer Danny Cohen to explain the difficulties of byte order.



Memory Representation – Byte Order – Part II

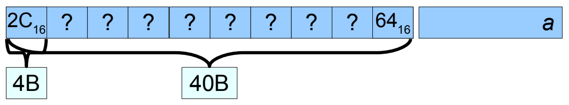
- Benefits of big-endian (68K, PPC, SPARC):
  - Better readability: the values are represented in the way they'd be written as literals.
- Benefits of little-endian (x86):
  - Scalability: if a value in memory needs resizing, the memory address is kept (like short to int).
    - The least significant byte just stays on its position on the lower address and the extended value occupies higher addressed space.
- Relevance of the byte order in C++:
  - For the representation of multibyte values having sequential bytes (a memory-block).
    - I.e. for values of size > 1B (e.g. int or short (but not arrays, esp. cstrings, for pointer arithmetics))
  - Functions like *std::memcpy()* operate on the exact underlying byte order.
- Irrelevance of the byte order in C++:
  - The address-operator (&) always returns the address of the lowest-addressed byte.
  - The assignment-operator (=) works always correctly.
  - Bit-operations and conversions are effective on values, not their byte order!

- The war of byte orders of multibyte data is fought since the 1970s when Intel (x86) and Motorola (68K) presented their products.
- The ARM architecture uses little endian by default, but is compatible to either.
- The readability of big-endians is relevant for memory dumps.
- There are also mixed byte orders (bi-endian) and (rare) completely different byte orders.
  - Bi-endians (IA-64) can switch the byte order and can have a different byte order in different memory segments.
  - Different byte orders store integral and floating point number in different byte orders, or they store floating point values as big/little-endian mix.
  - The date format in the US uses middle-endian: MM/DD/YYYY.
- Cstrings are always stored as big-endians.
- Relevance in other areas:
  - For registers the byte order is not relevant, the rightmost byte is always the LSB.
  - As the byte order influences the stack layout, certain sorts of bugs show up differently depending on the byte order. We'll discuss this in a future lecture.
  - If systems having different byte orders exchange data via a network, the byte order conversion (just on the byte copy layer) takes place in the network drivers. The byte order of the network protocol needs to be fixed to make this function. The fields (binary integers of the headers) in the protocols of the "Internet protocol suite" (IPv4, IPv6, TCP, and UDP) use big-endian, so even little-endian machines need to convert information to big-endian to communicate with each other! Because the byte order of the IP-family of protocols is big-endian, big-endian is also called the "network byte order". On the same machine there is of course no problem.
    - Socket APIs to program TCP/IP communication, e.g. WinSock, provide functions to convert byte orders in a portable way: *htons()* (host-to-network short) and *ntohl()* (network-to-host long)
  - I/O on files from such different system is implemented via a compatibility layer.
  - A byte order mark (BOM, the 2B sequences "FE FF" for big-endian and "FF FE" for little-endian) is used at the beginning of a stream of text encoded as UTF-16 or UTF-32, it allows the receiver to interpret the text correctly.



Memory Representation of Arrays

```
int a[10];
a[0] = 44;
a[9] = 100;
```



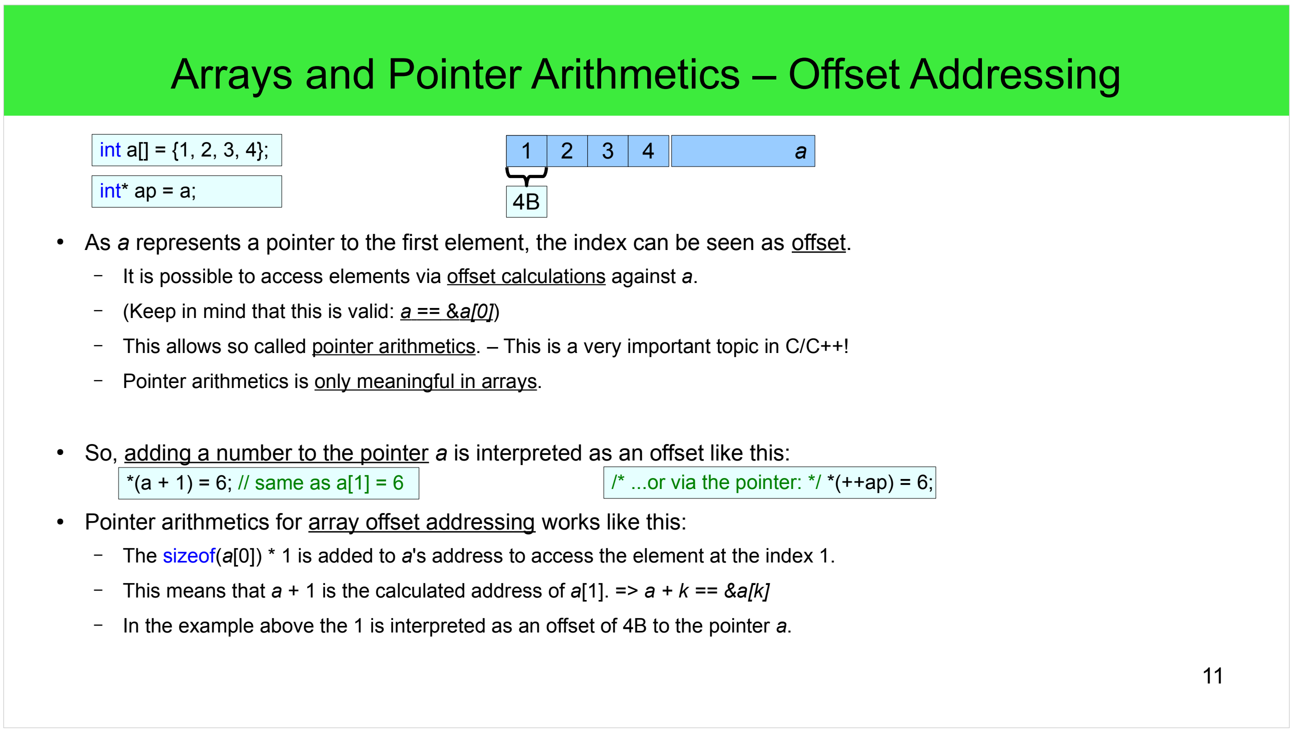
- The elements of *a* reside in memory as sequential block with space for ten ints.
  - The array symbol *a* represents the base address of the array.
  - The 1st element of *a* (*a*[0]), which is also *a*'s base address, resides at the lowest address.
- The size and count of elements of non-dynamic arrays:

```
std::size_t arraySize = sizeof(a); // Get the size of the array a (40).
int nElements = arraySize/sizeof(a[0]); // Get the count of elements in a (10).
```

- The address of the array (*a* (not &*a*)) is the same as of its first element (&*a*[0]).
  - The symbol *a* does itself represent the address of the array *a*.
  - W/o context, the address of *a* could be the address of an int or of an int-array.

10

- 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).
- Instead of sizeof(*a*) we could also get the value of sizeof(int[10]) as both values need to be equal (in the latter form we are required to write the argument for sizeof in parentheses, because int[] is a type).
- We can not get the address of the array *a* by writing &*a*, because the symbol *a* is not an lvalue. The symbol *a* itself represents the address of the array *a*.
- When we pass an array to a function, we are really passing its address (which is also the address of the first element) to the function, this is what we call array decay. – We need to pass the length of an array separately.



Arrays and Pointer Arithmetics – Offset Addressing

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

```
int* ap = a;
```

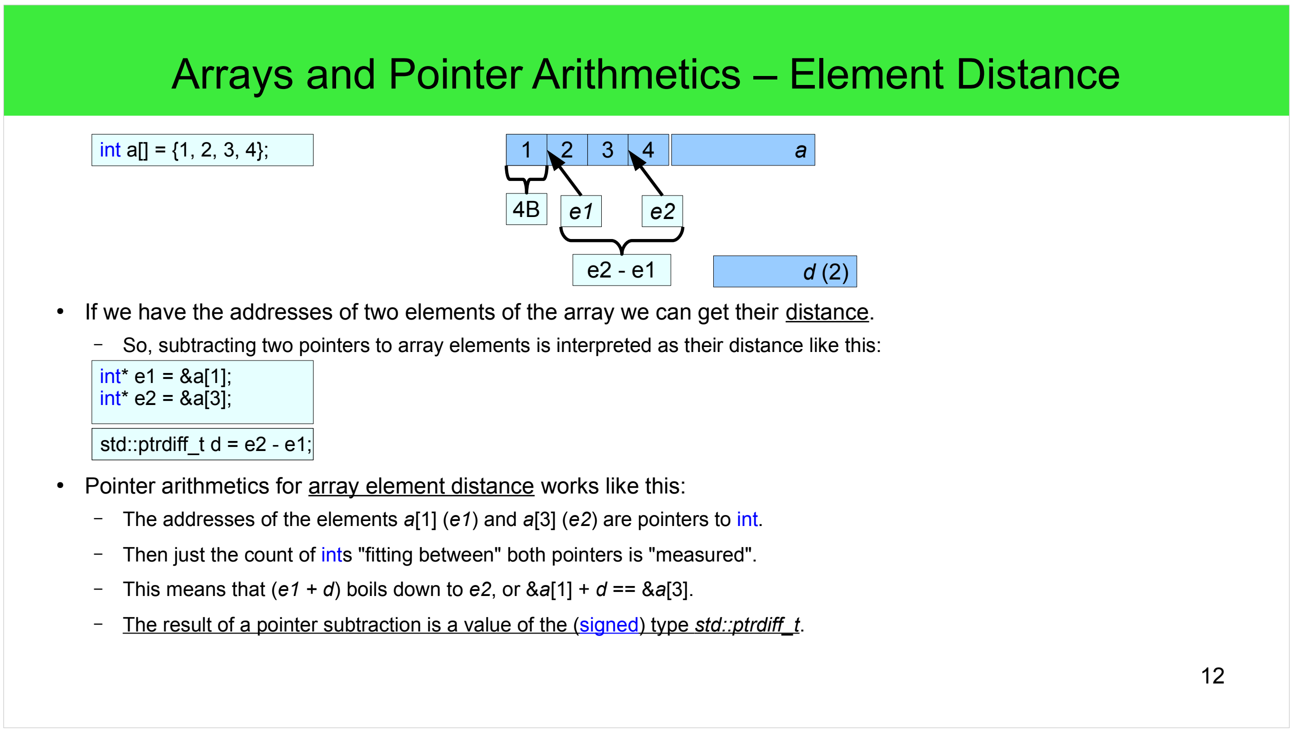
- As *a* represents a pointer to the first element, the index can be seen as offset.
  - It is possible to access elements via offset calculations against *a*.
  - (Keep in mind that this is valid: *a == &a[0]*)
  - This allows so called pointer arithmetics. – This is a very important topic in C/C++!
  - Pointer arithmetics is only meaningful in arrays.
- So, adding a number to the pointer *a* is interpreted as an offset like this:

```
*(a + 1) = 6; // same as a[1] = 6
```

```
/* ...or via the pointer: */ *(++ap) = 6;
```

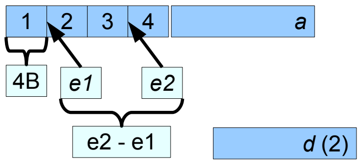
- Pointer arithmetics for array offset addressing works like this:
  - The sizeof(*a*[0]) * 1 is added to *a*'s address to access the element at the index 1.
  - This means that *a* + 1 is the calculated address of *a*[1]. => *a* + *k* == *&a[k]*
  - In the example above the 1 is interpreted as an offset of 4B to the pointer *a*.

- In C/C++ the argument of the subscript is virtually not an index, it is rather an offset from the address of the array, which is the address of the very first array element.
- The pointer arithmetics works, because the scalar number being added to or subtracted from a pointer is interpreted to scale for the type of the pointer we are operating on.
- Pointer arithmetics is an important topic in C++, because it abstracts the functionality of builtin types to be idiomatically compatible to the Standard Template Library (STL).



Arrays and Pointer Arithmetics – Element Distance

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



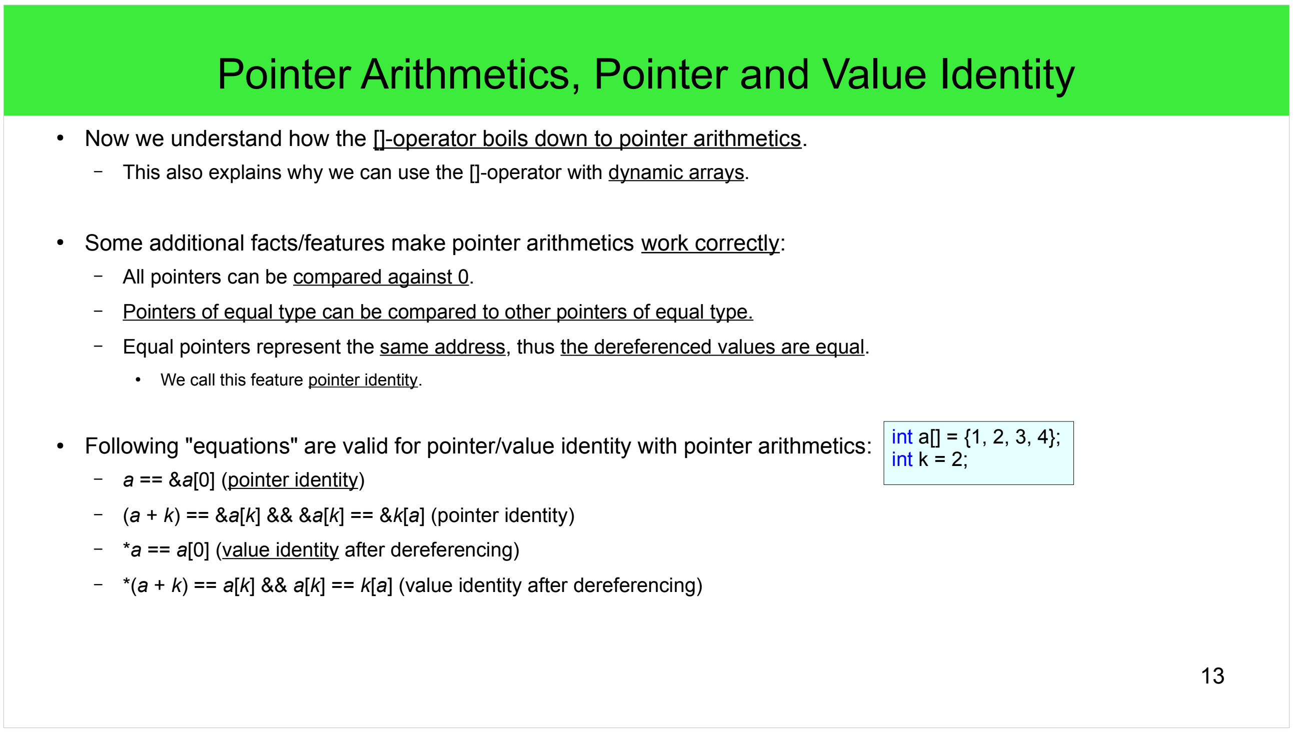
- If we have the addresses of two elements of the array we can get their distance.
  - So, subtracting two pointers to array elements is interpreted as their distance like this:

```
int* e1 = &a[1];
int* e2 = &a[3];
```

```
std::ptrdiff_t d = e2 - e1;
```

- Pointer arithmetics for array element distance works like this:
  - The addresses of the elements *a*[1] (*e1*) and *a*[3] (*e2*) are pointers to int.
  - Then just the count of ints "fitting between" both pointers is "measured".
  - This means that (*e1* + *d*) boils down to *e2*, or &*a*[1] + *d* == &*a*[3].
  - The result of a pointer subtraction is a value of the (signed) type *std::ptrdiff_t*.

- The type *std::ptrdiff_t* is defined in <cstddef>.



Pointer Arithmetics, Pointer and Value Identity

- Now we understand how the []-operator boils down to pointer arithmetics.
  - This also explains why we can use the []-operator with dynamic arrays.
- Some additional facts/features make pointer arithmetics work correctly:
  - All pointers can be compared against 0.
  - Pointers of equal type can be compared to other pointers of equal type.
  - Equal pointers represent the same address, thus the dereferenced values are equal.
    - We call this feature pointer identity.
- Following "equations" are valid for pointer/value identity with pointer arithmetics:
  - *a* == &*a*[0] (pointer identity)
  - (*a* + *k*) == &*a*[*k*] && &*a*[*k*] == &*k*[*a*] (pointer identity)
  - **a* == *a*[0] (value identity after dereferencing)
  - *(*a* + *k*) == *a*[*k*] && *a*[*k*] == *k*[*a*] (value identity after dereferencing)

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

13

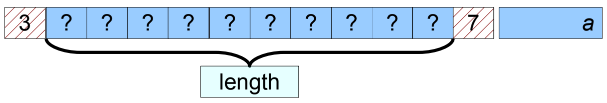
- The way pointer arithmetics work does also explain why the void* pointing to array created on the heap needs to be casted to a concrete type. – Why?
  - Because the operations with pointer arithmetics need to know the size of the elements to calculate correct offsets, a void* is just a generic pointer to a block of memory. It explains also why a void* can not be dereferenced.

Arrays and foreign Memory – Writing

```
const int length = 10;
int a[length];
```

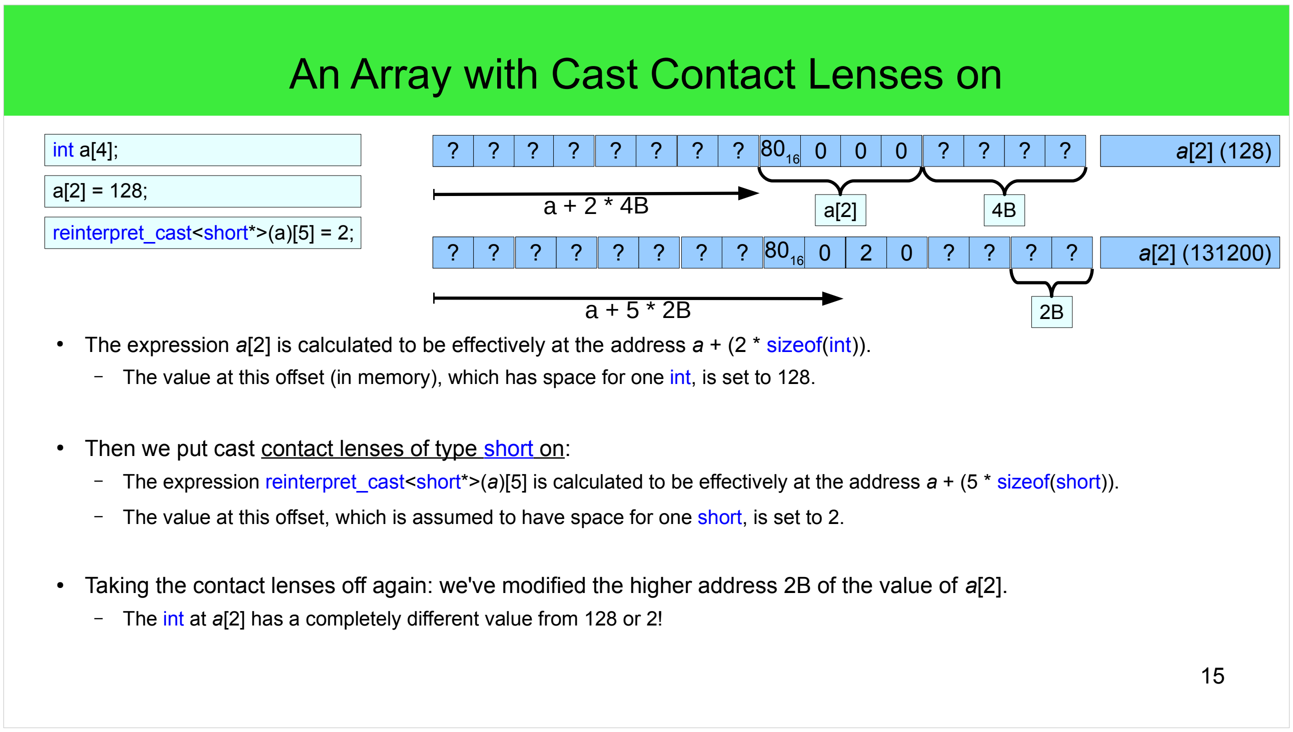
```
a[length] = 7;
```

```
a[-1] = 3;
```



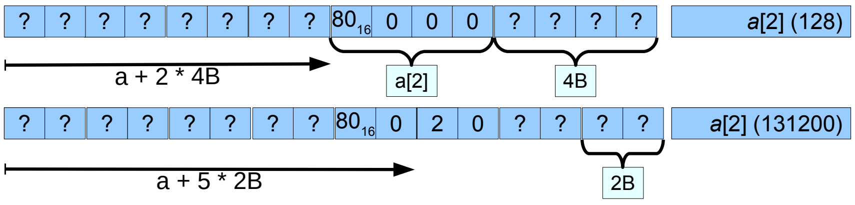
- We call the pointer to the array element *a*[*length*] the past-the-end-pointer.
  - It points to the first "element" not belonging to the array.
  - It has a special meaning in C/C++.
  - C/C++ support no bounds checking, but the past-the-end-pointer can be safely read.

- Writing memory, which we don't own is still illegal.
  - The local variables of a function are packed into the stack frame.
  - Writing to array-foreign memory could modify the values of adjacent variables in the sf.



An Array with Cast Contact Lenses on

```
int a[4];
a[2] = 128;
reinterpret_cast<short*>(a)[5] = 2;
```



- The expression *a*[2] is calculated to be effectively at the address *a* + (2 * sizeof(int)).
  - The value at this offset (in memory), which has space for one int, is set to 128.
- Then we put cast contact lenses of type short on:
  - The expression reinterpret_cast<short*>(*a*)[5] is calculated to be effectively at the address *a* + (5 * sizeof(short)).
  - The value at this offset, which is assumed to have space for one short, is set to 2.
- Taking the contact lenses off again: we've modified the higher address 2B of the value of *a*[2].
  - The int at *a*[2] has a completely different value from 128 or 2!

15

- In the following examples we'll have to use reinterpret_casts (static_casts can not convert from int* to short* as in this example).
- Why are the memory portions modified in the presented order?
  - Because of the byte order. (We assume little endian in this case, as the MSB is right from the LSB.)
- We could go really crazy with casting and navigation through memory, the possible combinations are endless. But it is often also pointless and maybe dangerous.

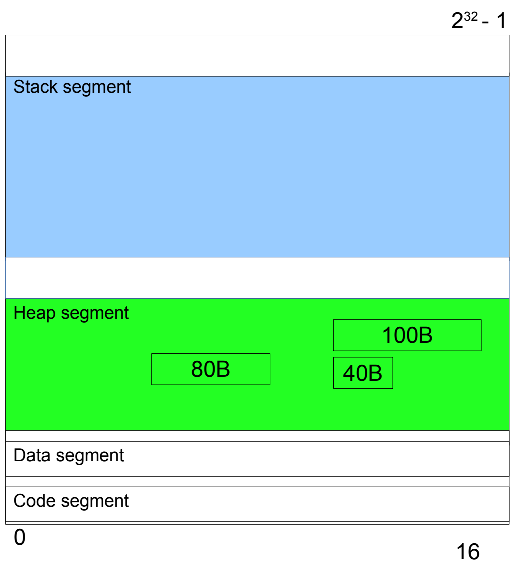
Memory Segmentation – Geometric Memory Properties

- Let's assume a pointer has a size of 4B.
- The stack segment
  - stores local (auto) variables,
  - manages the stack of function calls and
  - is owned and managed by hardware.
- The heap segment
  - is managed by the heap manager and
  - is owned and managed by software.
    - *std::malloc()*, *std::realloc()*, *std::free()* etc.

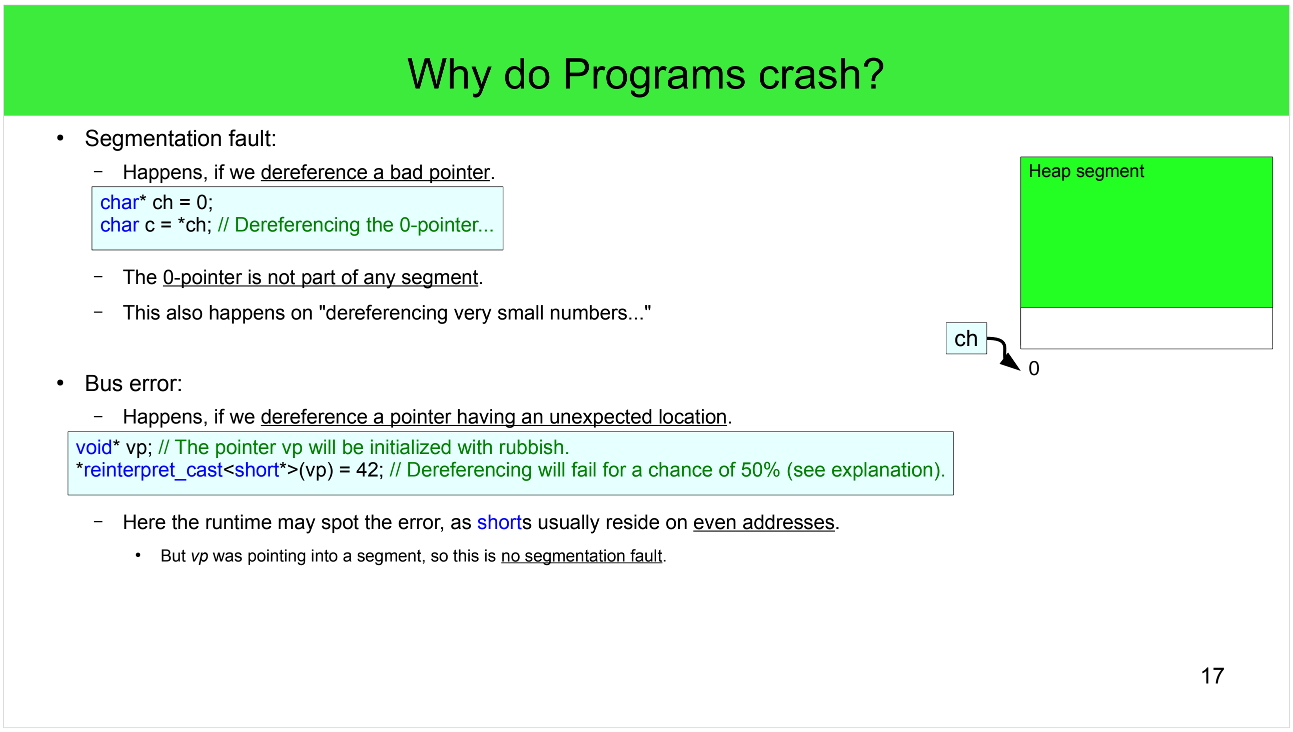
```
void* a = std::malloc(80);
```

```
void* b = std::malloc(40);
```

```
void* c = std::realloc(b, 100);
```



- <u>Why do we draw a memory of size 2^{32}?</u>
  - If the pointer's size is 4B, the width of the address bus (count of address-"wires") must be 32b, this makes 2^{32} different bytes to be addressable.
- This memory model:
  - Keeping data and code in the same memory is an important aspect of the "von Neumann architecture".
  - The depicted memory model exists since the early 70s and was introduced with the "Real" processor mode. Safety was introduced with the protected mode .
  - The dimensions of the segments are not realistic. The stack is rather small and the heap is rather big.
  - In assembly languages it is possible to use direct segmentation to define which data resides in the data and in the code segment.
- When a program is loaded the start and end address of the heap are passed to the heap manager.
- <u>What is the code segment (text segment)?</u>
  - In this portion of the memory the object code or assembly code resides.
- The C/C++ types for which memory is allocated don't matter to the heap manger; it rather manages requests of portions of bytes.
- The function *std::realloc()* may or may not extend the portion of memory (in "higher address direction") that is designated by the passed pointer. This means that the address returned by *std::realloc()* needs not to be the same as the one passed.
- If a memory block is reallocated and can not be extended, because there is not enough adjacent space, another matching memory block will be allocated.

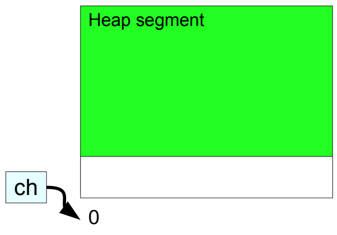


Why do Programs crash?

- Segmentation fault:
  - Happens, if we dereference a bad pointer.

```
char* ch = 0;
char c = *ch; // Dereferencing the 0-pointer...
```

  - The 0-pointer is not part of any segment.
  - This also happens on "dereferencing very small numbers..."

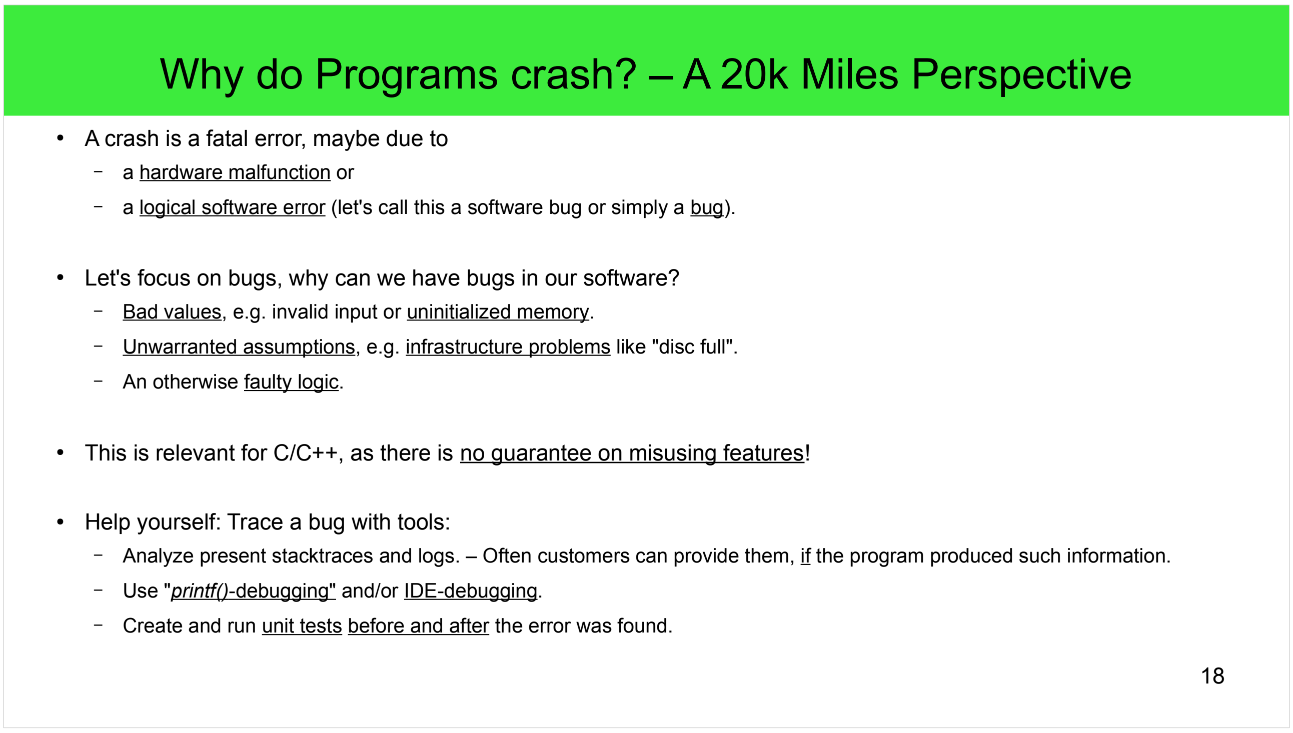


- Bus error:
  - Happens, if we dereference a pointer having an unexpected location.

```
void* vp; // The pointer vp will be initialized with rubbish.
*reinterpret_cast<short*>(vp) = 42; // Dereferencing will fail for a chance of 50% (see explanation).
```

  - Here the runtime may spot the error, as shorts usually reside on even addresses.
    - But *vp* was pointing into a segment, so this is no segmentation fault.

- Why does the explained bus error only appear in 50% of the cases?
  - By a chance of 50% the dereferenced address is odd.
- Typically ints reside on addresses being a multiple of four.
- Typically there is no address restriction for bytes/chars.



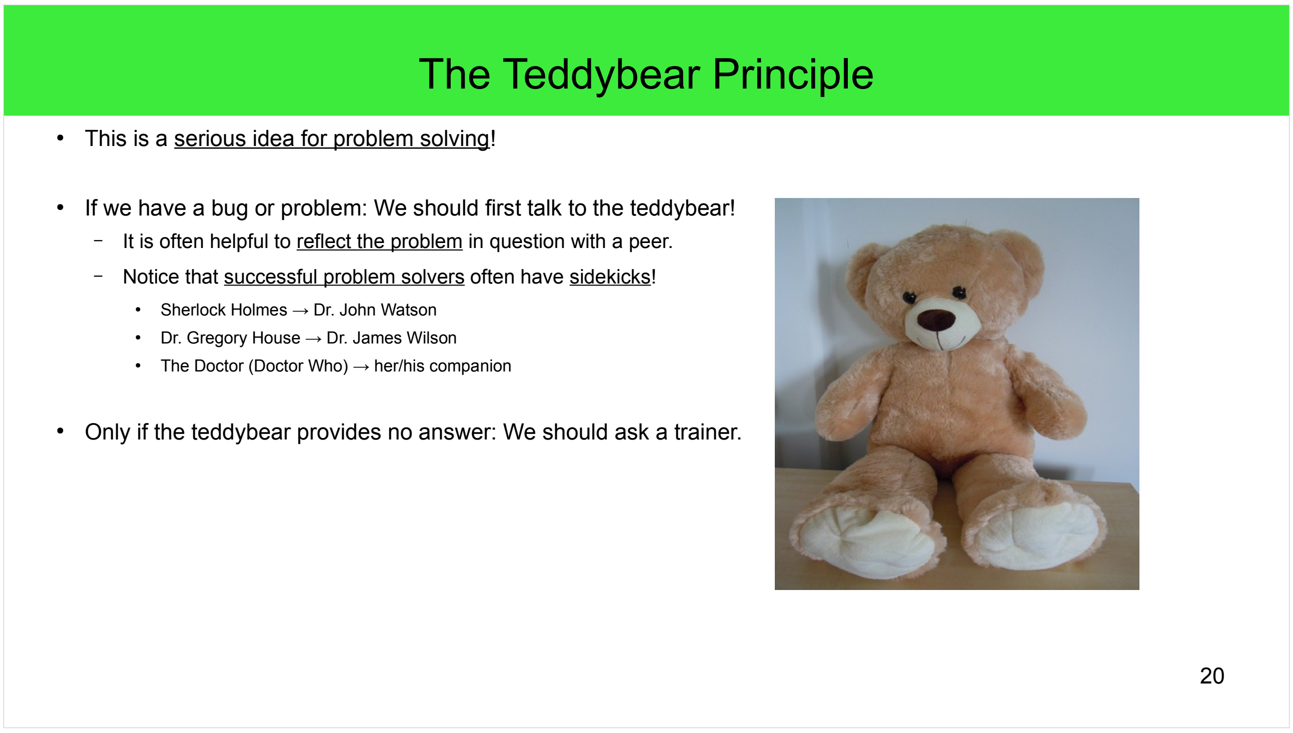
Why do Programs crash? – A 20k Miles Perspective

- A crash is a fatal error, maybe due to
  - a hardware malfunction or
  - a logical software error (let's call this a software bug or simply a bug).
- Let's focus on bugs, why can we have bugs in our software?
  - Bad values, e.g. invalid input or uninitialized memory.
  - Unwarranted assumptions, e.g. infrastructure problems like "disc full".
  - An otherwise faulty logic.
- This is relevant for C/C++, as there is no guarantee on misusing features!
- Help yourself: Trace a bug with tools:
  - Analyze present stacktraces and logs. – Often customers can provide them, if the program produced such information.
  - Use "*printf()*-debugging" and/or IDE-debugging.
  - Create and run unit tests before and after the error was found.

- After a crash we may be forced to remove the crashed process or to reboot the system.
- C/C++ is a programming language for programmers: there is no guarantee on misusing features. – There is no elaborate exception strategy like in .NET or Java. In C++ we can use/consume exceptions, but they are only present in STL APIs, for other APIs we have to cope with undefined behavior.

Nailing down Bugs – Simple but useful Tips

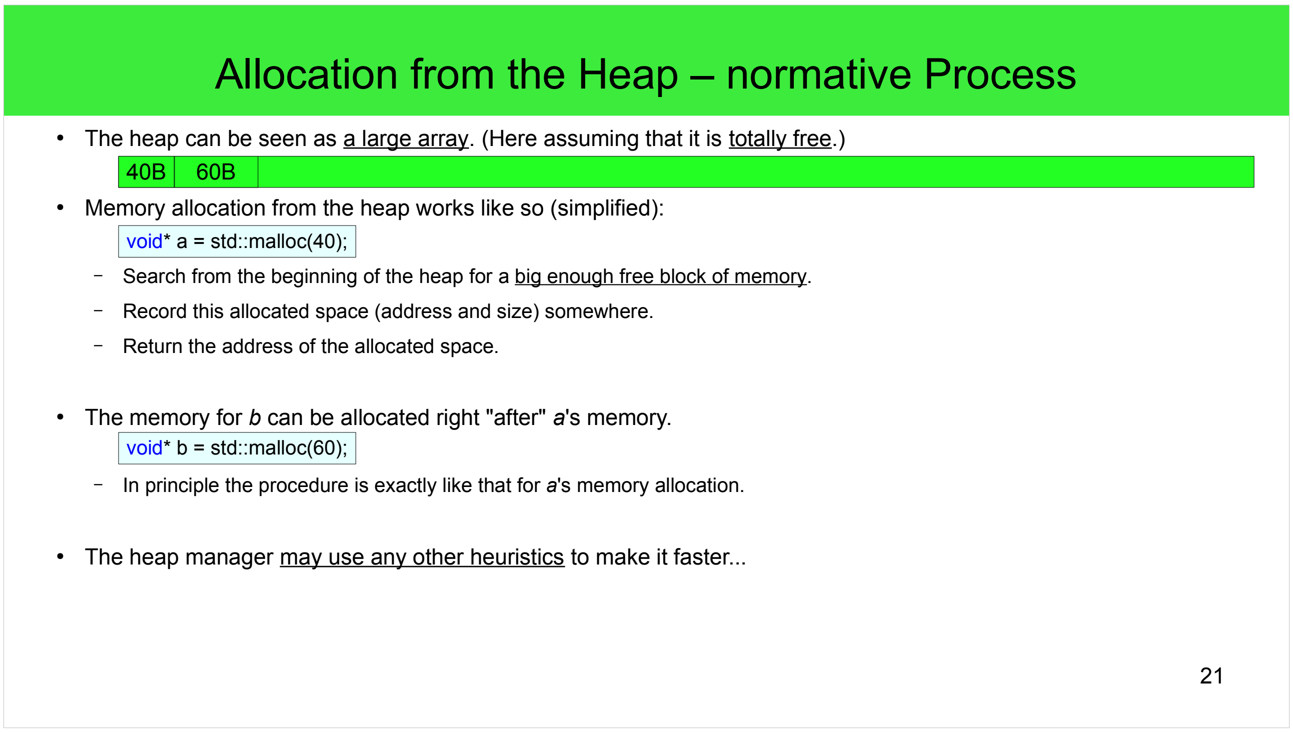
- 1. Bug-finding needs to be done systematically.
- 2. Make assumptions about the bug or problem!
- 3. We should be utterly critical about our assumptions and hold nothing for granted!
  - We have always to check user input and the return value of relevant functions like *std::malloc()*!
  - Forgetting to program defensively is a major source of errors!
  - Compile with highest warning level!
- 4. Don't panic!
- 5. Do it with pair programming.
- 6. Explain the bug or problem to another person, or...



The Teddybear Principle

- This is a serious idea for problem solving!
- If we have a bug or problem: We should first talk to the teddybear!
  - It is often helpful to reflect the problem in question with a peer.
  - Notice that successful problem solvers often have sidekicks!
    - Sherlock Holmes → Dr. John Watson
    - Dr. Gregory House → Dr. James Wilson
    - The Doctor (Doctor Who) → her/his companion
- Only if the teddybear provides no answer: We should ask a trainer.

- The teddybear principle: http://talkaboutquality.wordpress.com/2010/08/30/tell-it-to-your-teddy-bear/



Allocation from the Heap – normative Process

- The heap can be seen as a large array. (Here assuming that it is totally free.)

40B	60B	

- Memory allocation from the heap works like so (simplified):

```
void* a = std::malloc(40);
```

  - Search from the beginning of the heap for a big enough free block of memory.
  - Record this allocated space (address and size) somewhere.
  - Return the address of the allocated space.

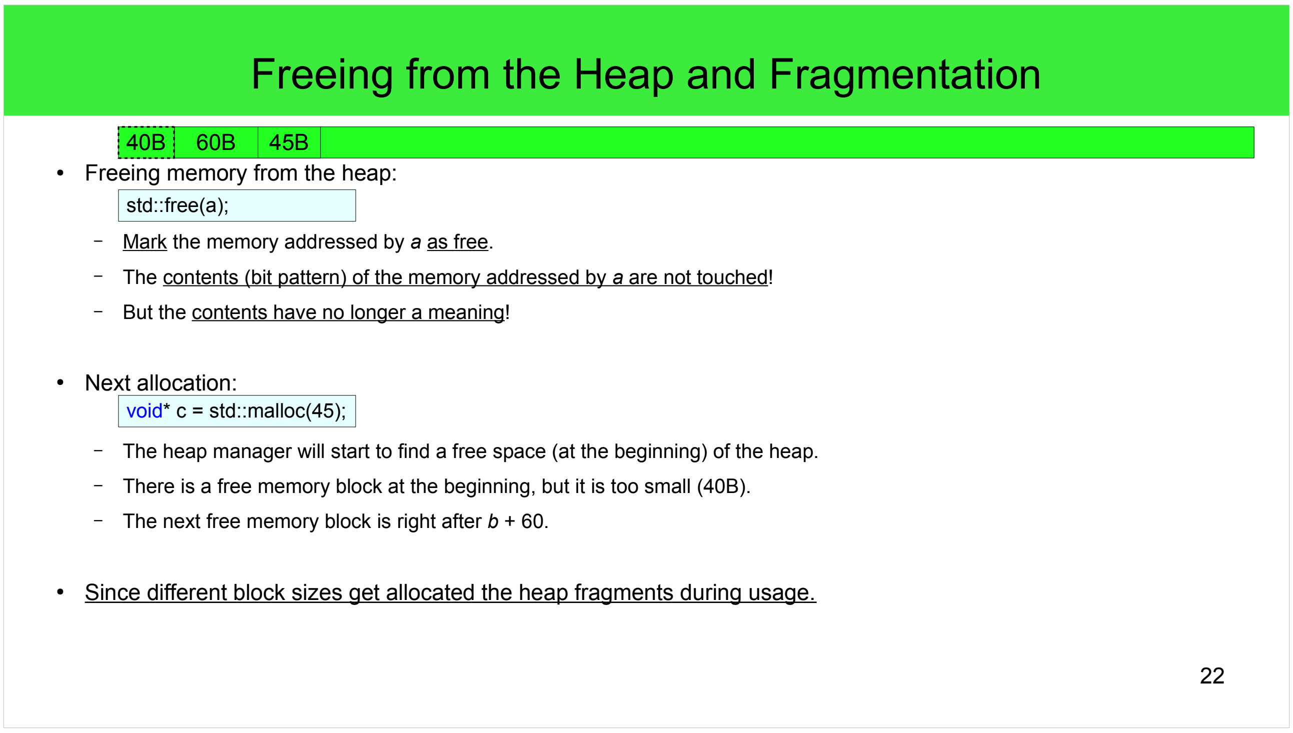
- The memory for *b* can be allocated right "after" *a*'s memory.

```
void* b = std::malloc(60);
```

  - In principle the procedure is exactly like that for *a*'s memory allocation.

- The heap manager may use any other heuristics to make it faster...

- This is not exactly what happens, but close enough to understand it basically.



Freeing from the Heap and Fragmentation

40B 60B 45B

- Freeing memory from the heap:

```
std::free(a);
```

  - Mark the memory addressed by *a* as free.
  - The contents (bit pattern) of the memory addressed by *a* are not touched!
  - But the contents have no longer a meaning!

- Next allocation:

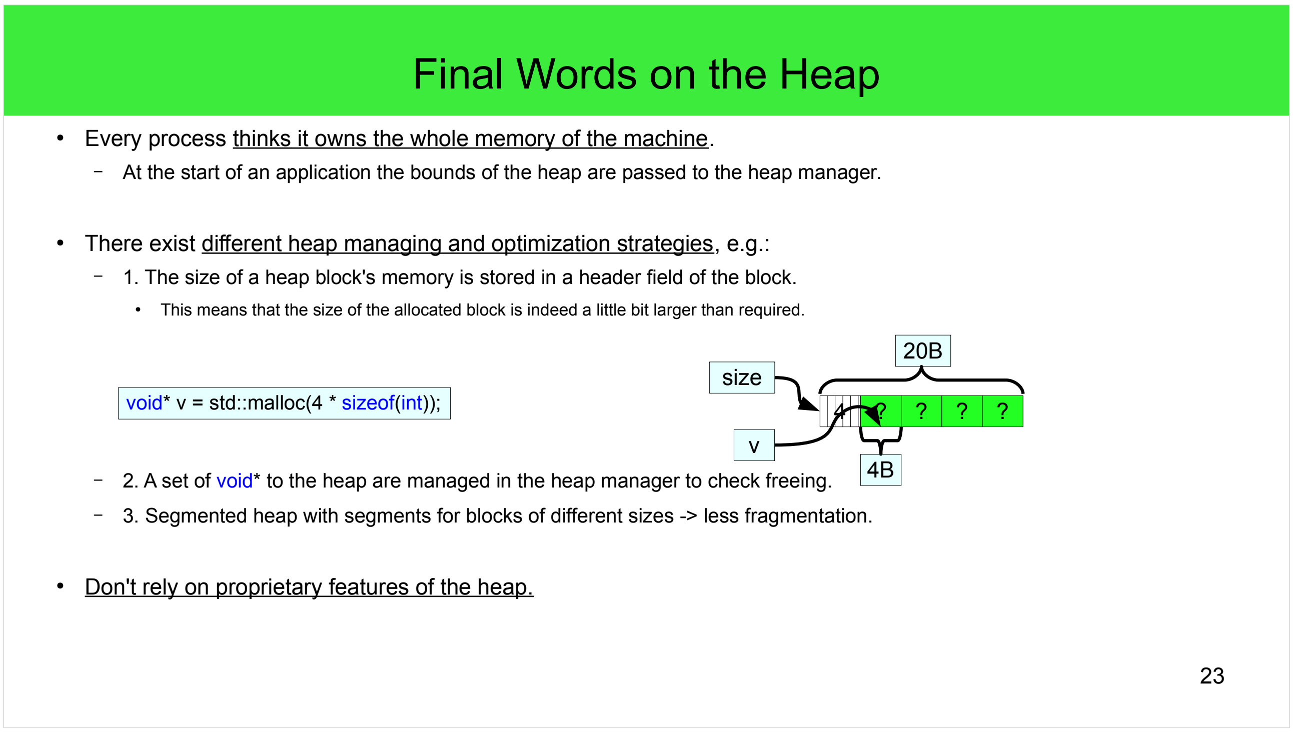
```
void* c = std::malloc(45);
```

  - The heap manager will start to find a free space (at the beginning) of the heap.
  - There is a free memory block at the beginning, but it is too small (40B).
  - The next free memory block is right after *b* + 60.

- Since different block sizes get allocated the heap fragments during usage.

22

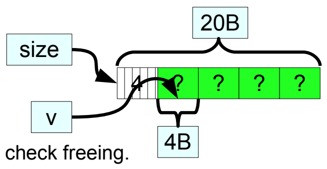
- The heap fragmentation evolves like a parking bay w/o marks: Cars of different dimensions enter and leave the bay. After a while, spaces occupied by wide cars will be reused by small cars. The surplus space is wasted leaving a fragmented parking bay.
- After *a* has been freed, the formerly occupied memory could be reused by next allocations.
- Not all implementations of the heap manager start finding free memory at the beginning of the heap.
- The heap manager can record the available gaps of free memory as a linked list to their pointers. The pointers are called "free nodes" and the linked list is called "free list".



Final Words on the Heap

- Every process thinks it owns the whole memory of the machine.
  - At the start of an application the bounds of the heap are passed to the heap manager.
- There exist different heap managing and optimization strategies, e.g.:
  - 1. The size of a heap block's memory is stored in a header field of the block.
    - This means that the size of the allocated block is indeed a little bit larger than required.

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



  - 2. A set of void* to the heap are managed in the heap manager to check freeing.
  - 3. Segmented heap with segments for blocks of different sizes -> less fragmentation.
- Don't rely on proprietary features of the heap.

23

- A concrete example for a segmented heap is the "Low Fragmentation Heap" (LFH), which can be used in Windows Vista and newer Windows versions. It stores memory blocks of different sizes in dedicated buckets to optimize search time for free memory and to lower the heap fragmentation of course.

Thank you!