(9) Basics of the C++ Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

- (9) C++ Basics
  - Introducing CPU Registers
  - Function Stack Frames and the Decrementing Stack
  - Function Call Stacks, the Stack Pointer and the Base Pointer
  - C/C++ Calling Conventions
  - Stack Overflow, Underflow and Channelling incl. Examples
  - How variable Argument Lists work with the Stack
  - Static versus automatic Storage Classes
  - The static Storage Class and the Data Segment

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

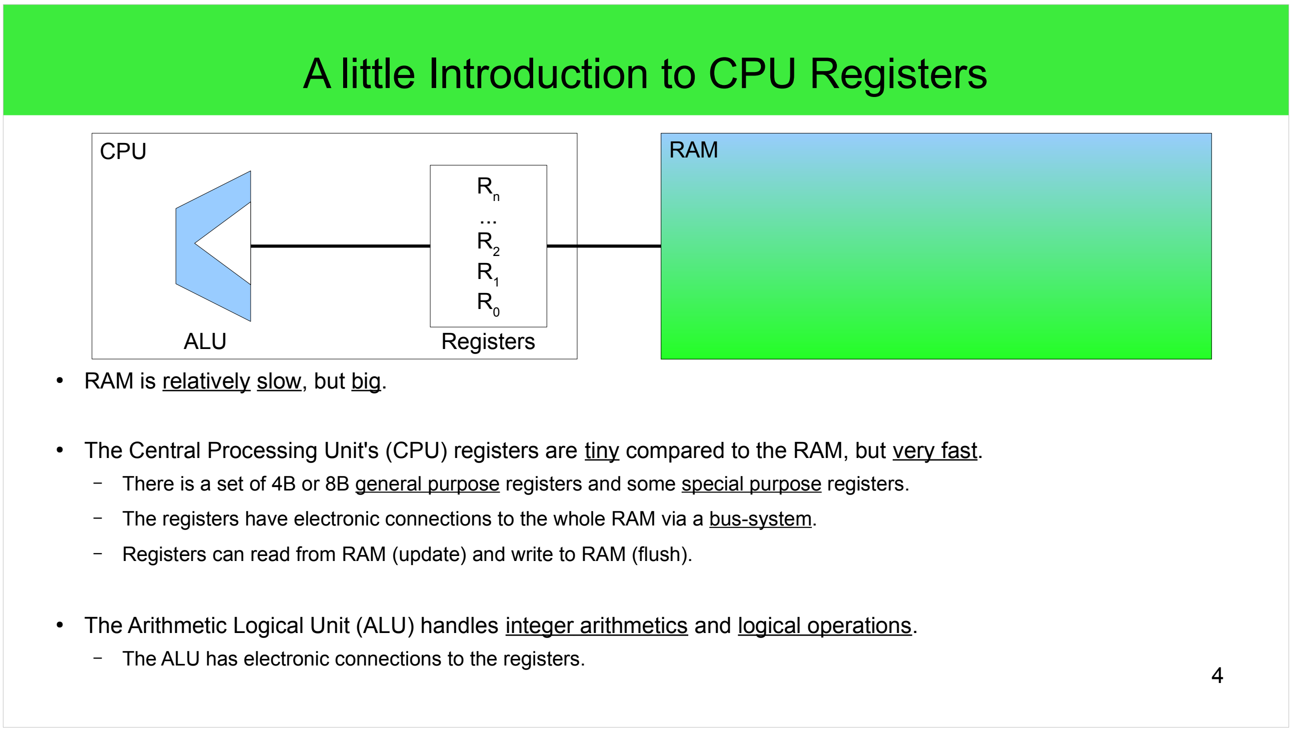
Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

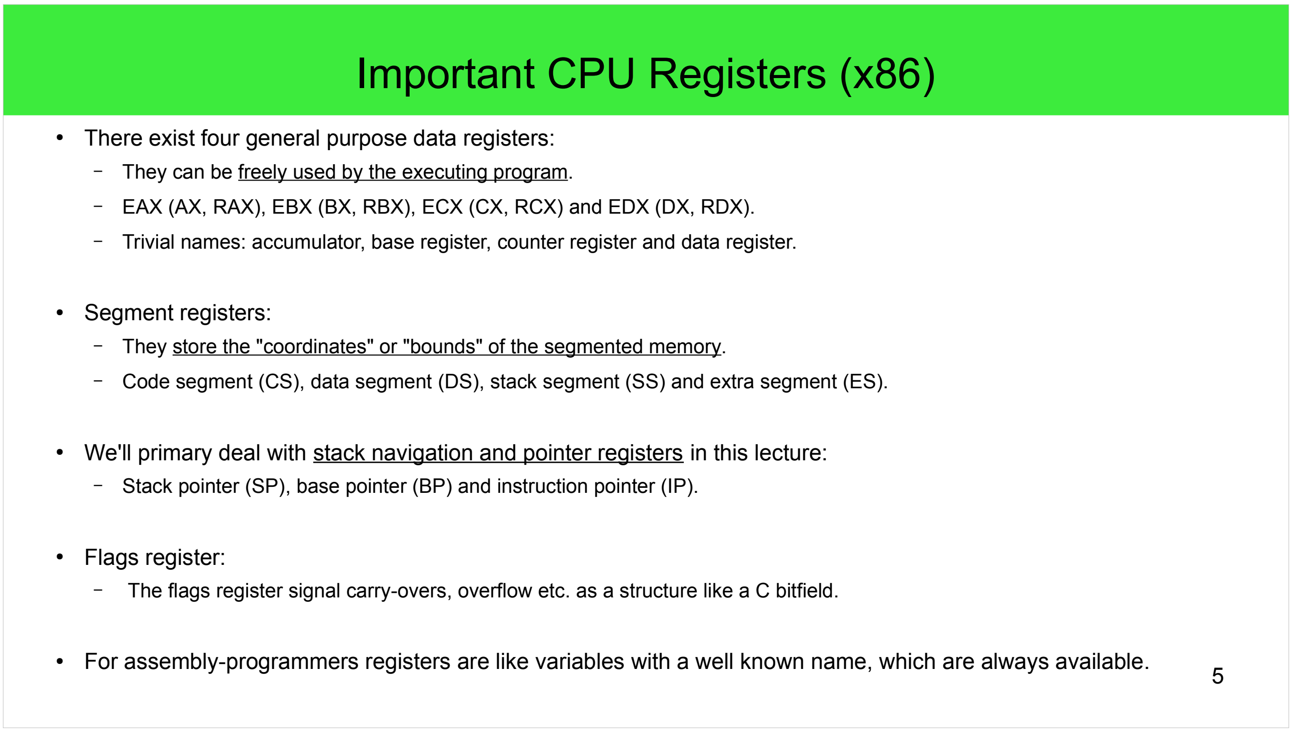
Have fun!



A little Introduction to CPU Registers

- RAM is relatively slow, but big.
- The Central Processing Unit's (CPU) registers are tiny compared to the RAM, but very fast.
  - There is a set of 4B or 8B general purpose registers and some special purpose registers.
  - The registers have electronic connections to the whole RAM via a bus-system.
  - Registers can read from RAM (update) and write to RAM (flush).
- The Arithmetic Logical Unit (ALU) handles integer arithmetics and logical operations.
  - The ALU has electronic connections to the registers.

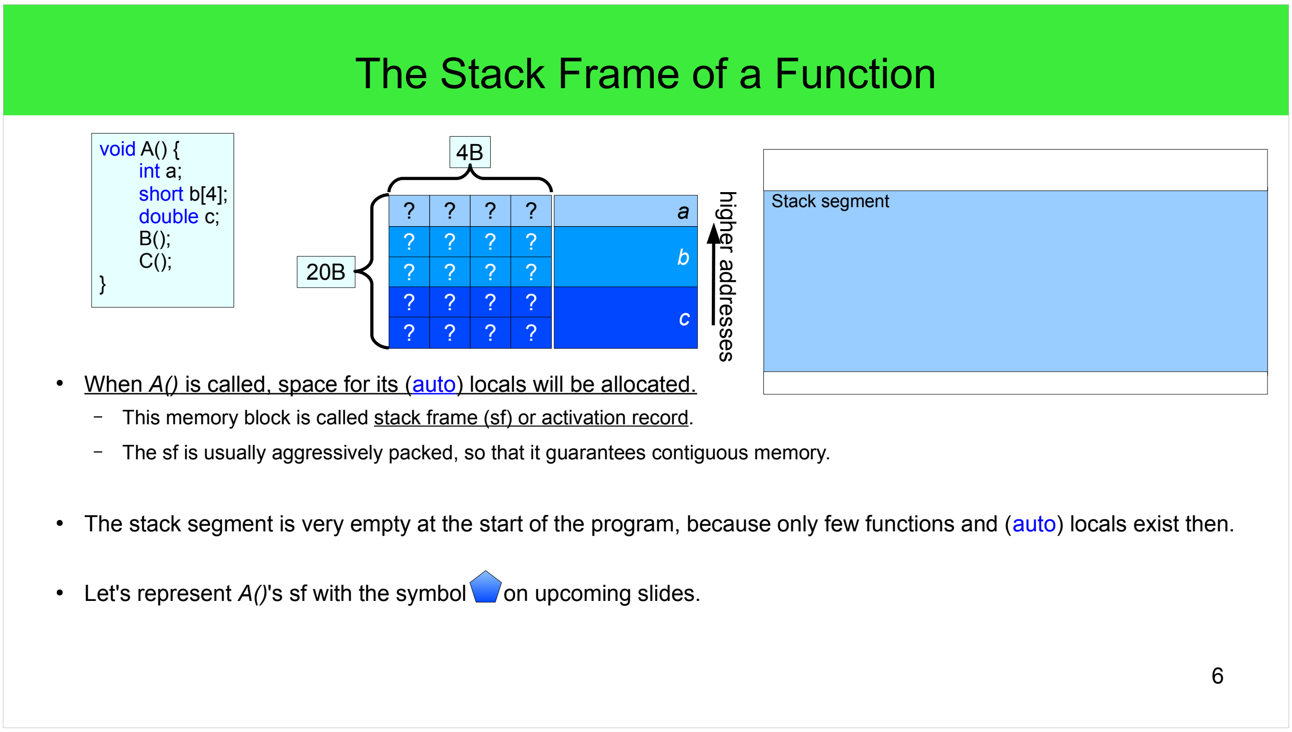
- As we're going to discuss the stack, which is only managed by the hardware. We need to know a little more about the hardware in this respect.
- The shape of the ALU (like a Y) underscores the idea of having two or more operands and one result.
- The basic parts of a CPU are the registers, the ALU and the control unit (CU), which controls program execution (the fetch-execute cycle).
- The dimensions of the RAM and the registers in the graphic are not realistic. The registers are very much smaller than the RAM. The graphic presents less than a microprocessor minimal system, we concentrate only on required details.
  - Indeed there is a kind of memory hierarchy:
    - The registers and the CPU caches are of very small, but very fast (register: 0.x ns, cache: some ns) and very expensive static memory (=SRAM, for Static RAM memory units are built of flip-flops, which need not to be refreshed allowing very fast access, however, SRAM is getting really hot in comparison to (S)DRAM)
    - RAM is much slower than CPUs, therefor the function of caches is to shield the CPU from the RAM's bad performance.
    - RAM is of moderate size, speed and prize (dynamic memory (=(S)DRAM, for (Synchronous) Dynamic RAM) a single memory unit is built from a capacitor and a transistor, which needs to be periodically refreshed, during refreshes (S)DRAM cannot be accessed)
    - And finally the memory of solid state drives (SSDs), magnetic and optical devices is huge, relatively slow and cheap.
- A new upcoming computer-memory architecture is the non-uniform memory access (NUMA), in which each CPU has local memory as well as all CPUs sharing a common memory.
- The shown connections are part of the CPU-internal bus system.
- The CPU-internal bus between the registers and the ALU defines the architecture-classification of the CPU (a 32b-bus makes a CPU a 32b-CPU as 32b can be processed in one CPU cycle). – Also the width of the external and internal data bus and of the registers plays a role in the classification.
- Why are registers either 4B or 8B big?
  - It depends on the CPU, a whole data word should be storable by a register: 4B for a 32b machine, 8B for a 64b machine.
- Normally arithmetic operations are not directly performed in the RAM.
  - Connecting the ALU direct to the memory would be slow and/or expensive.
  - Actually registers are filled with data from the RAM (update), then taken to the ALU where the operation takes place, then the result is sent back to the registers and finally copied to the RAM (flush).



Important CPU Registers (x86)

- There exist four general purpose data registers:
  - They can be freely used by the executing program.
  - EAX (AX, RAX), EBX (BX, RBX), ECX (CX, RCX) and EDX (DX, RDX).
  - Trivial names: accumulator, base register, counter register and data register.
- Segment registers:
  - They store the "coordinates" or "bounds" of the segmented memory.
  - Code segment (CS), data segment (DS), stack segment (SS) and extra segment (ES).
- We'll primary deal with stack navigation and pointer registers in this lecture:
  - Stack pointer (SP), base pointer (BP) and instruction pointer (IP).
- Flags register:
  - The flags register signal carry-overs, overflow etc. as a structure like a C bitfield.
- For assembly-programmers registers are like variables with a well known name, which are always available.

- There are more registers on a 64b CPU. Lesser 64b data fit into the caches, so the caches will become filled quite soon (therefor 64b CPUs have bigger caches). Acceleration is esp. noticeable on Intel Macs, where we have more registers. Not so much on Power PC Macs, where we already have the full set of registers, important is the increased addressable memory here primarily.
- Normally the types "pointer" and long will be influenced by 64-bitness (LP64) (some type field characters for string formatting must be modified (%d (32 only) -> %ld and %x -> %p) and also sizeofs should be used instead of constants as 4 (e.g. on calling memory-functions)).

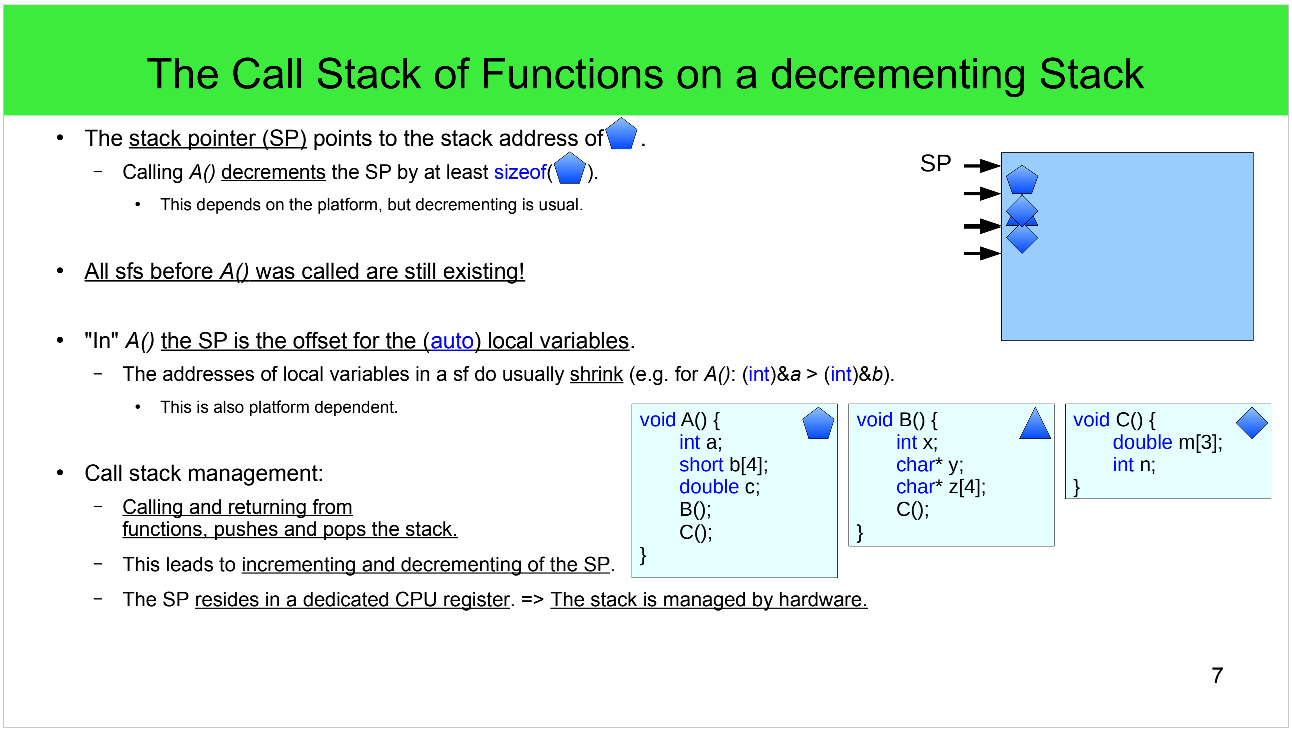


The Stack Frame of a Function

```
void A() {
        int a;
        short b[4];
        double c;
        B();
        C();
}
```

- When *A()* is called, space for its (auto) locals will be allocated.
  - This memory block is called stack frame (sf) or activation record.
  - The sf is usually aggressively packed, so that it guarantees contiguous memory.
- The stack segment is very empty at the start of the program, because only few functions and (auto) locals exist then.
- Let's represent *A()*'s sf with the symbol ⬟ on upcoming slides.

- Why do we discuss the stack after we have discussed the heap?
  - Because the handling of the stack is more difficult to understand, as the hardware algorithms that control the stack need to be understood. – Esp. the order of elements on the stack and the order of actions on the stack is very relevant! The heap is rather simple: the programmers are responsible to handle it, they have to define conventions!
- The sf does also contain memory for variables defined in blocks, but this can be optimized on some compilers. Up to C99 it was not allowed to put the definition of variables in blocks.
- The discussed stack is aligned, it means that some bytes are sacrificed in order to get simple access to stack elements having addresses on a multiple of the word size. – This is compiler- and settings-specific, but here we have a simple model to explain the stack, by assuming a 4B alignment.
- This is a simple view of the sf, we'll refine it during the next slides.



The Call Stack of Functions on a decrementing Stack

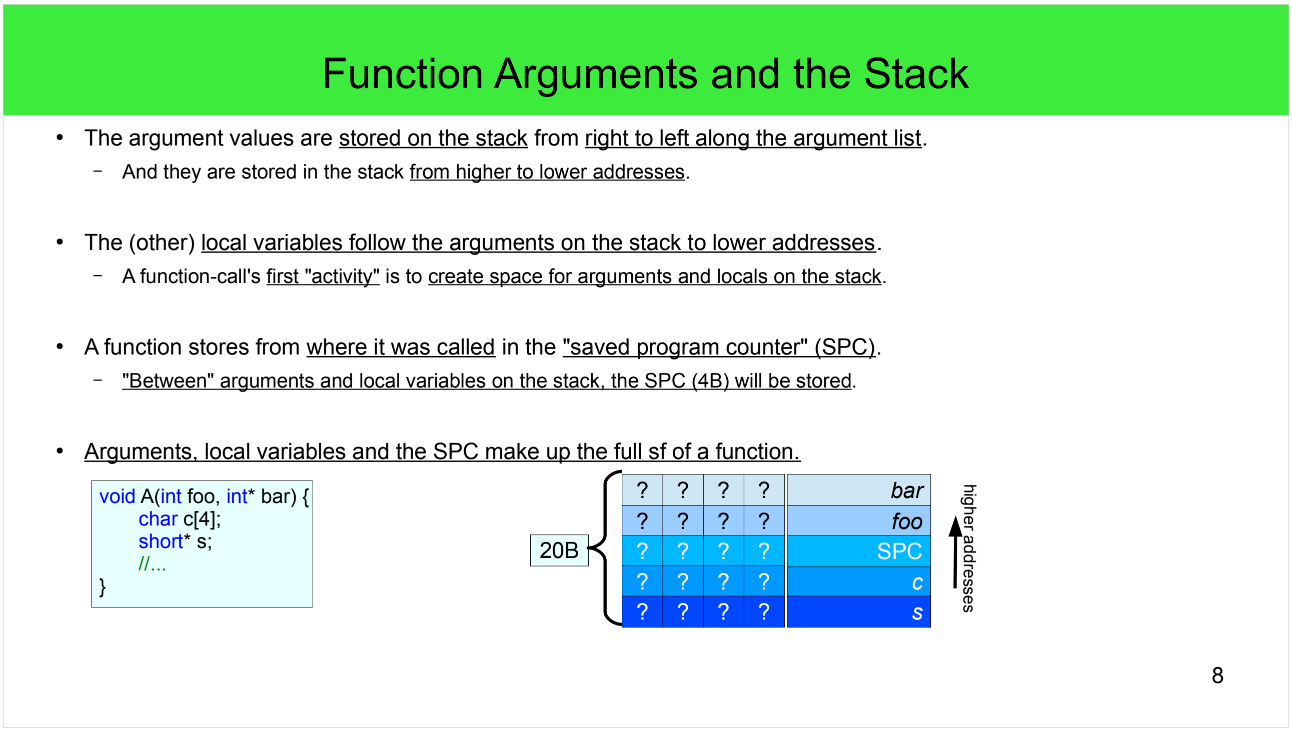
- The stack pointer (SP) points to the stack address of ⬟.
  - Calling *A()* decrements the SP by at least sizeof(⬟).
    - This depends on the platform, but decrementing is usual.
- All sfs before *A()* was called are still existing!
- "In" *A()* the SP is the offset for the (auto) local variables.
  - The addresses of local variables in a sf do usually shrink (e.g. for *A()*: (int)&*a* > (int)&*b*).
    - This is also platform dependent.
- Call stack management:
  - Calling and returning from functions, pushes and pops the stack.
  - This leads to incrementing and decrementing of the SP.
  - The SP resides in a dedicated CPU register. => The stack is managed by hardware.

```
void A() {                ⬟
        int a;
        short b[4];
        double c;
        B();
        C();
}
```

```
void B() {                ▲
        int x;
        char* y;
        char* z[4];
        C();
}
```

```
void C() {                ◆
        double m[3];
        int n;
}
```

- Because the SP needs to be decremented for each stack variable, for each function call and for the sf construction this stack is called "decrementing stack".
- As the stack grows to lower addresses (i.e. against free memory) it grows "against" the heap. The heap may grow to higher addresses (i.e. also "against" free memory). When the stack and heap meet each other in memory, memory is exhausted. Usually the stack is exhausted first.
- In this example we can also inspect the nature of the stack as "last in first out" (LIFO) container. – The last item that was put into the stack will be the next item that will be taken from the stack.
- Similar to the heap, the values being left by already popped sfs do stay in the memory as long as they have not been overwritten by the next function call, they are just no longer legally accessible. – This can also be a source of bugs.
- Notice how the picture showing the call stack also makes clear how recursive functions an quickly consume much stack space and overflow.



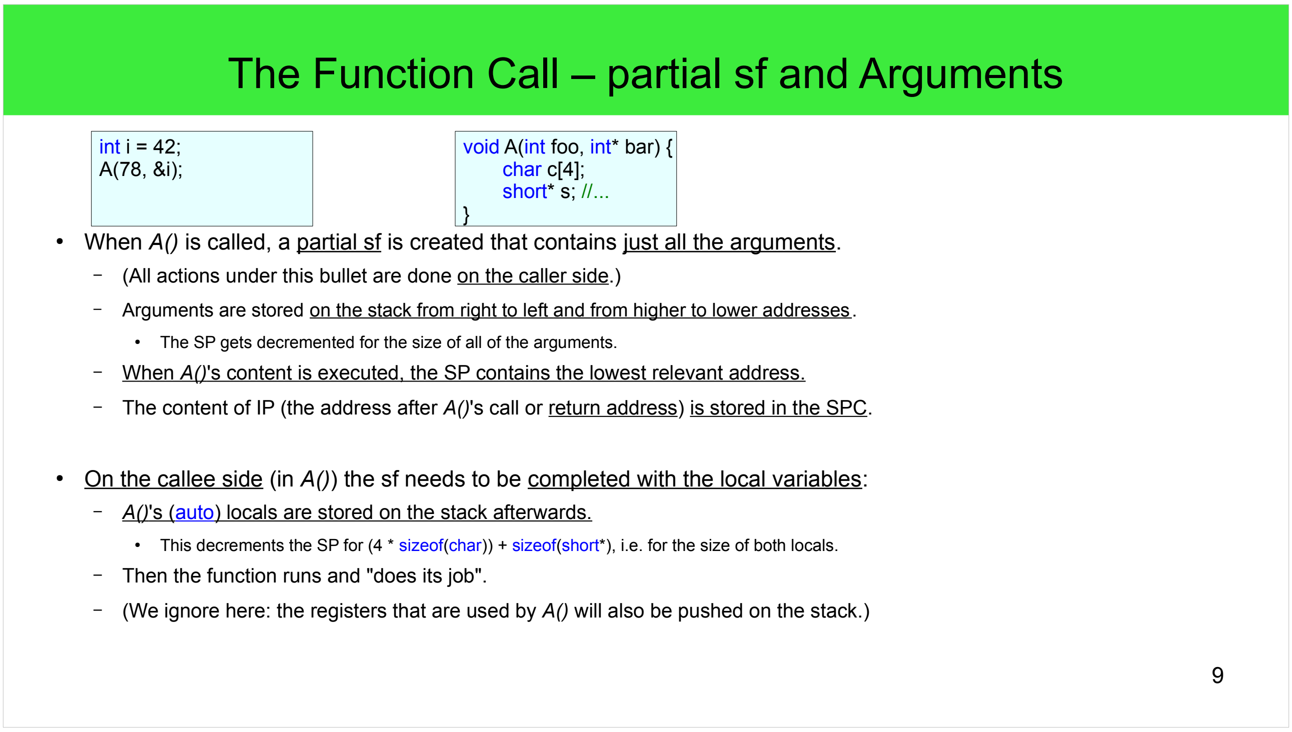
Function Arguments and the Stack

- The argument values are stored on the stack from right to left along the argument list.
  - And they are stored in the stack from higher to lower addresses.
- The (other) local variables follow the arguments on the stack to lower addresses.
  - A function-call's first "activity" is to create space for arguments and locals on the stack.
- A function stores from where it was called in the "saved program counter" (SPC).
  - "Between" arguments and local variables on the stack, the SPC (4B) will be stored.
- Arguments, local variables and the SPC make up the full sf of a function.

```
void A(int foo, int* bar) {
        char c[4];
        short* s;
        //...
}
```

?	?	?	?	*bar*
?	?	?	?	*foo*
?	?	?	?	SPC
?	?	?	?	*c*
?	?	?	?	*s*

- On Reduced Instruction Set Computing (RISC) CPUs there exist so called "Register Windows" to project different stacks into the current stack frame with a single operation, so it's a fast way to pass arguments to functions. The general idea with RISC CPUs is to reduce memory access and stack operations.
- There exist architectures that have no stack at all (we discuss only the ones having a stack).



The Function Call – partial sf and Arguments

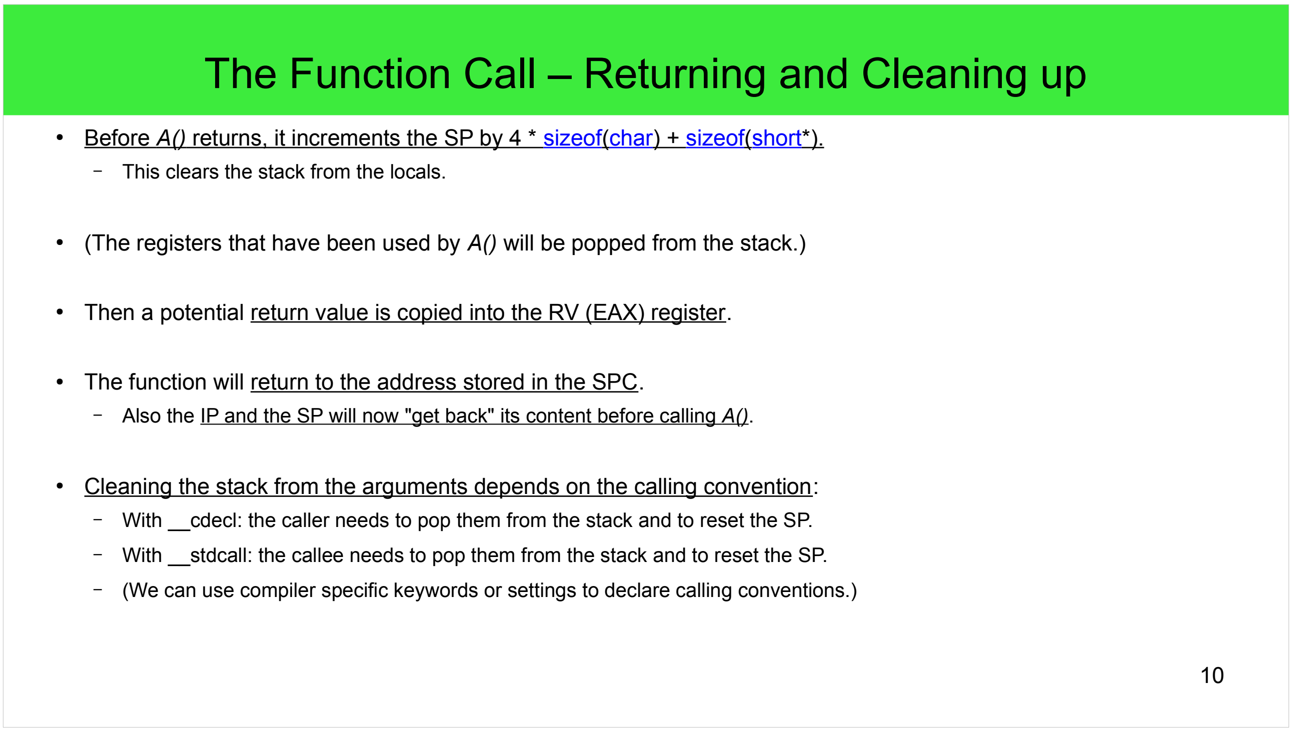
```
int i = 42;
A(78, &i);
```

```
void A(int foo, int* bar) {
      char c[4];
      short* s; //...
}
```

- When *A()* is called, a partial sf is created that contains just all the arguments.
  - (All actions under this bullet are done on the caller side.)
  - Arguments are stored on the stack from right to left and from higher to lower addresses.
    - The SP gets decremented for the size of all of the arguments.
  - When *A()*'s content is executed, the SP contains the lowest relevant address.
  - The content of IP (the address after *A()*'s call or return address) is stored in the SPC.

- On the callee side (in *A()*) the sf needs to be completed with the local variables:
  - *A()*'s (auto) locals are stored on the stack afterwards.
    - This decrements the SP for (4 * sizeof(char)) + sizeof(short*), i.e. for the size of both locals.
  - Then the function runs and "does its job".
  - (We ignore here: the registers that are used by *A()* will also be pushed on the stack.)

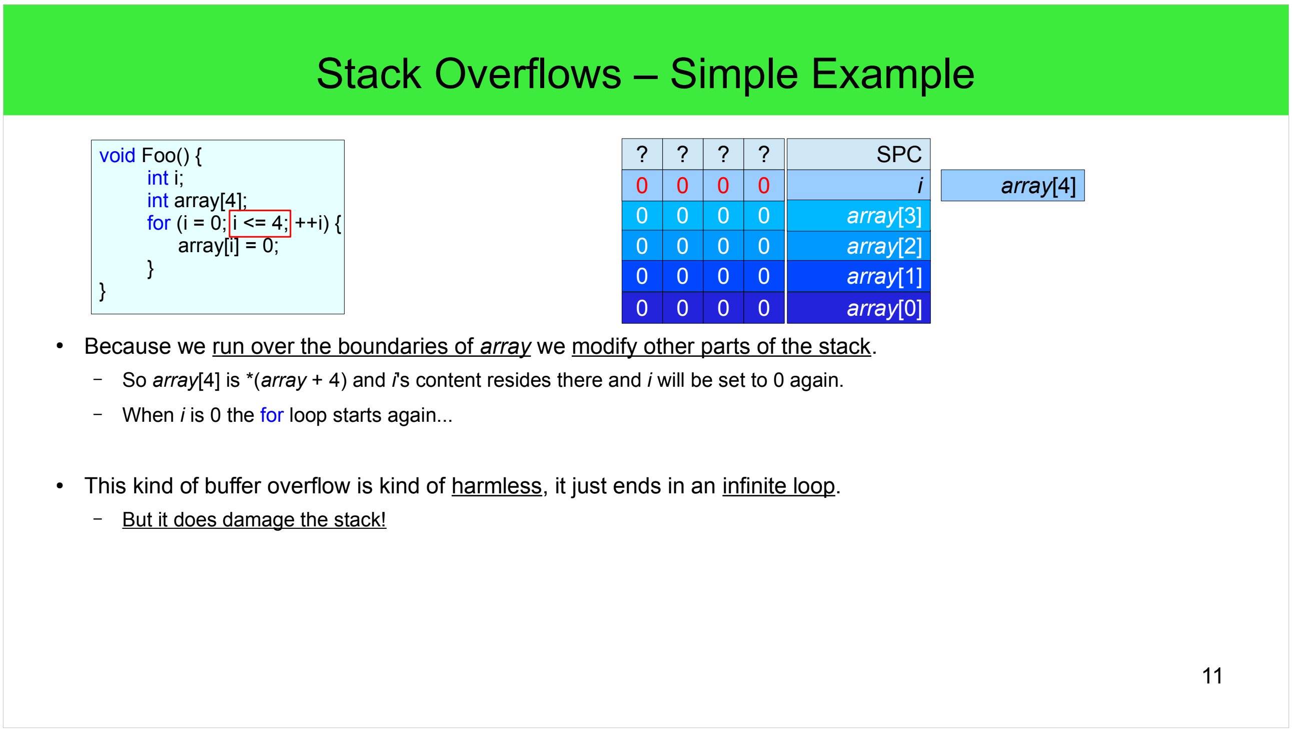
- The caller needs to fill the "argument part" of the sf, because only the caller knows all the arguments. The callee needs to fill the "local auto part", because only the caller knows all the local auto variables.
- Normally the content of the SP register is stored in the base pointer (BP) register (also called environment pointer) in the function. From the BP then the offsets to the local variables are calculated. The SP contains the offset address (within the stack segment) to the next item in the stack during execution.



The Function Call – Returning and Cleaning up

- Before *A()* returns, it increments the SP by 4 * sizeof(char) + sizeof(short*).
  - This clears the stack from the locals.
- (The registers that have been used by *A()* will be popped from the stack.)
- Then a potential return value is copied into the RV (EAX) register.
- The function will return to the address stored in the SPC.
  - Also the IP and the SP will now "get back" its content before calling *A()*.
- Cleaning the stack from the arguments depends on the calling convention:
  - With __cdecl: the caller needs to pop them from the stack and to reset the SP.
  - With __stdcall: the callee needs to pop them from the stack and to reset the SP.
  - (We can use compiler specific keywords or settings to declare calling conventions.)

- 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.



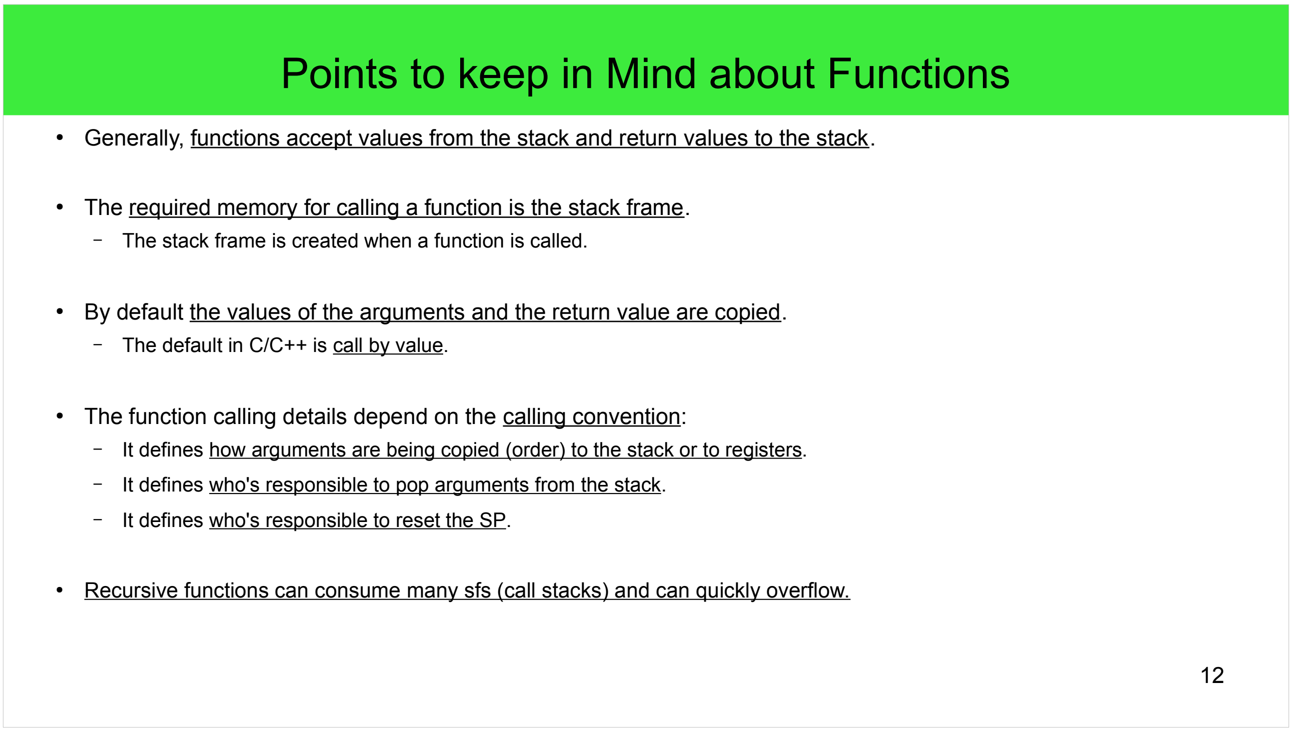
Stack Overflows – Simple Example

```
void Foo() {
        int i;
        int array[4];
        for (i = 0; i <= 4; ++i) {
            array[i] = 0;
        }
}
```

?	?	?	?	SPC	
0	0	0	0	*i*	*array*[4]
0	0	0	0	*array*[3]	
0	0	0	0	*array*[2]	
0	0	0	0	*array*[1]	
0	0	0	0	*array*[0]	

- Because we run over the boundaries of *array* we modify other parts of the stack.
  - So *array*[4] is *(*array* + 4) and *i*'s content resides there and *i* will be set to 0 again.
  - When *i* is 0 the for loop starts again...
- This kind of buffer overflow is kind of harmless, it just ends in an infinite loop.
  - But it does damage the stack!

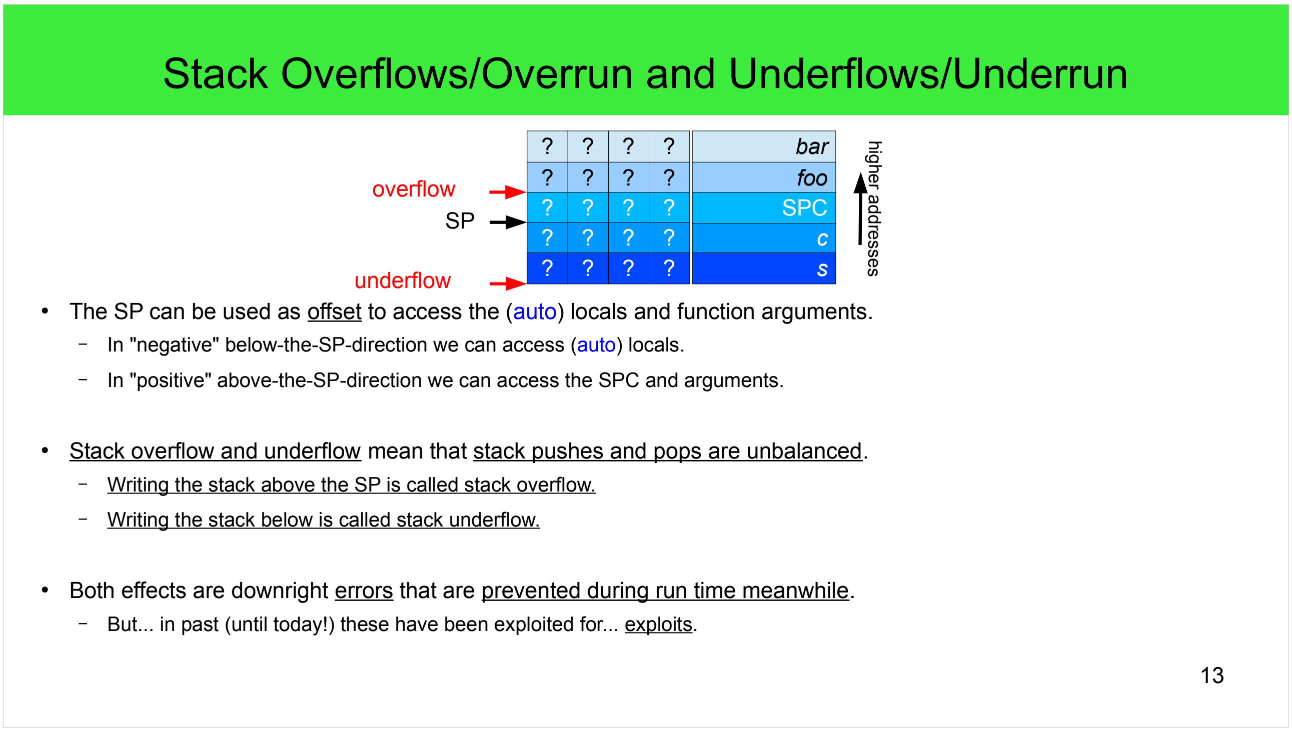
- Can anybody spot the error in *Foo()*?
- The g++ (Xcode) must be flagged with -fno-stack-protector to witness this effect.



Points to keep in Mind about Functions

- Generally, functions accept values from the stack and return values to the stack.
- The required memory for calling a function is the stack frame.
  - The stack frame is created when a function is called.
- By default the values of the arguments and the return value are copied.
  - The default in C/C++ is call by value.
- The function calling details depend on the calling convention:
  - It defines how arguments are being copied (order) to the stack or to registers.
  - It defines who's responsible to pop arguments from the stack.
  - It defines who's responsible to reset the SP.
- Recursive functions can consume many sfs (call stacks) and can quickly overflow.

- Some compilers (and languages like F#) are able to enable tail recursion. Tail recursion means, that if the last expression of a function is the recursive call, the call can be done w/o using the stack to store auto variables (incl. parameters).

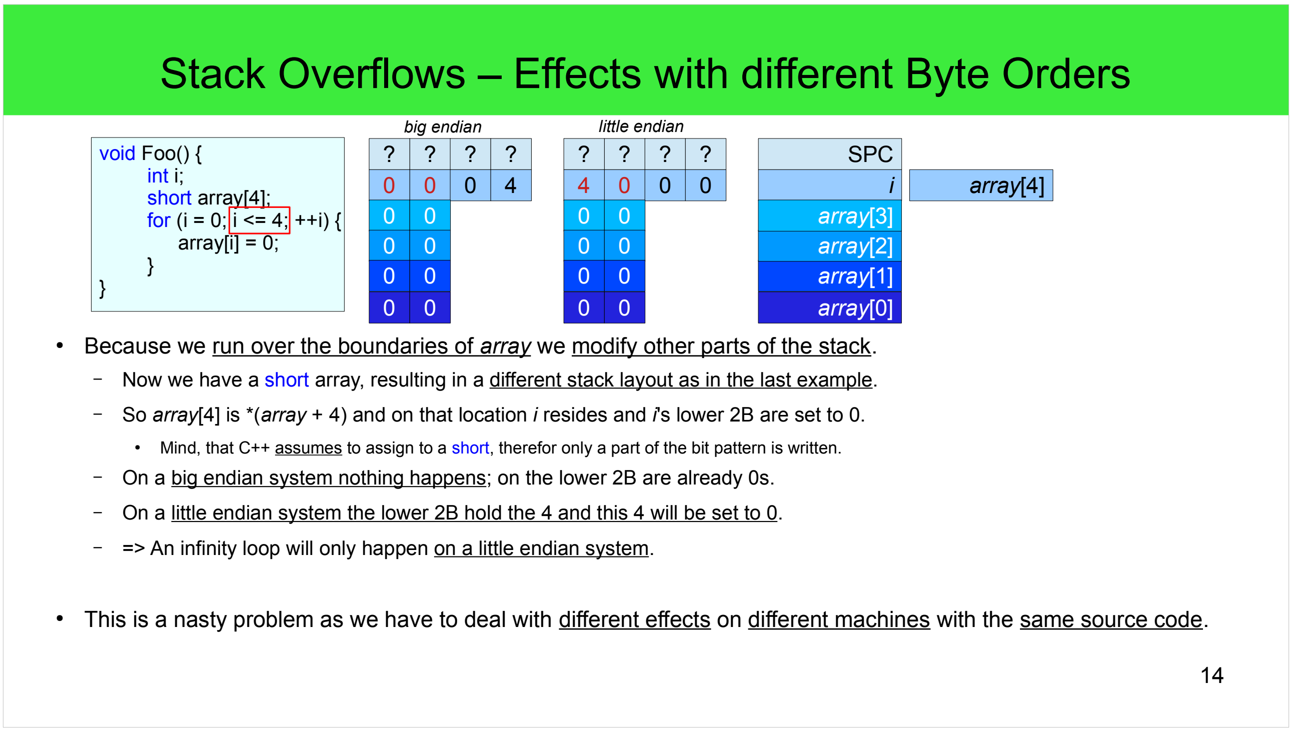


Stack Overflows/Overrun and Underflows/Underrun

? ? ? ? bar
overflow
? ? ? ? foo
SP
? ? ? ? SPC
? ? ? ? c
underflow
? ? ? ? s
higher addresses

- The SP can be used as offset to access the (auto) locals and function arguments.
  - In "negative" below-the-SP-direction we can access (auto) locals.
  - In "positive" above-the-SP-direction we can access the SPC and arguments.
- Stack overflow and underflow mean that stack pushes and pops are unbalanced.
  - Writing the stack above the SP is called stack overflow.
  - Writing the stack below is called stack underflow.
- Both effects are downright errors that are prevented during run time meanwhile.
  - But... in past (until today!) these have been exploited for... exploits.

- What is an exploit?
  - The point with exploits is, that the program (the executable code) is still intact, but foreign data is injected. – This is similar to biological viruses, which use intact cells as factories to reproduce its own DNA, that is injected as foreign data.
- A stack overflow leads to overwriting already used stack memory, a stack underflow means that stack content that is not used by "us" is read.
- It should be said that for the following examples to compile and run many stack protections needed to be deactivated on the compiler level. If the protections remained activated, the compiler would add stack guard elements into the code and we would get run time errors, before the stack violation could get effective and dangerous.



Stack Overflows – Effects with different Byte Orders

```
void Foo() {
        int i;
        short array[4];
        for (i = 0; i <= 4; ++i) {
            array[i] = 0;
        }
}
```

big endian

?	?	?	?
0	0	0	4
0	0		
0	0		
0	0		
0	0		

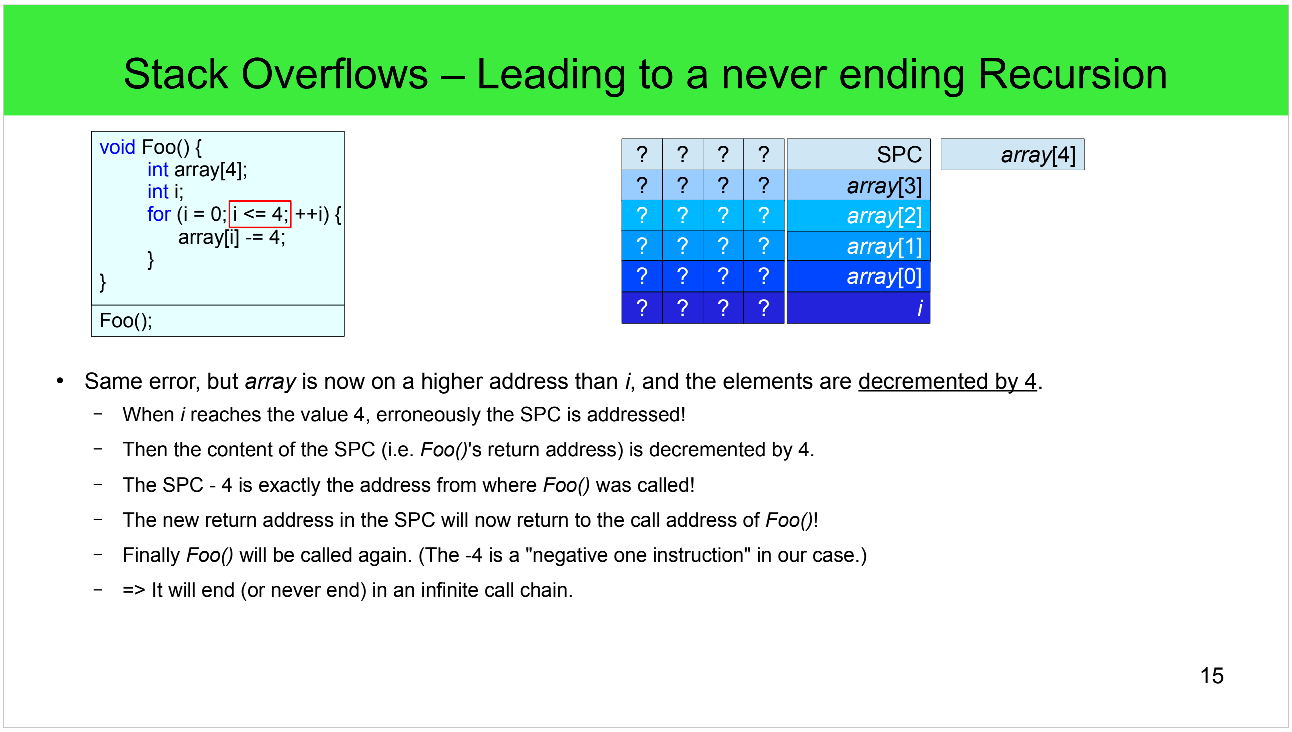
little endian

?	?	?	?
4	0	0	0
0	0		
0	0		
0	0		
0	0		

SPC	
i	*array*[4]
array[3]	
array[2]	
array[1]	
array[0]	

- Because we run over the boundaries of *array* we modify other parts of the stack.
  - Now we have a short array, resulting in a different stack layout as in the last example.
  - So *array*[4] is *(*array* + 4) and on that location *i* resides and *i*'s lower 2B are set to 0.
    - Mind, that C++ assumes to assign to a short, therefor only a part of the bit pattern is written.
  - On a big endian system nothing happens; on the lower 2B are already 0s.
  - On a little endian system the lower 2B hold the 4 and this 4 will be set to 0.
  - => An infinity loop will only happen on a little endian system.
- This is a nasty problem as we have to deal with different effects on different machines with the same source code.

- This is an example of how problems can emerge silently.
- The g++ (Xcode) must be flagged with -fno-stack-protector to witness this effect.



Stack Overflows – Leading to a never ending Recursion

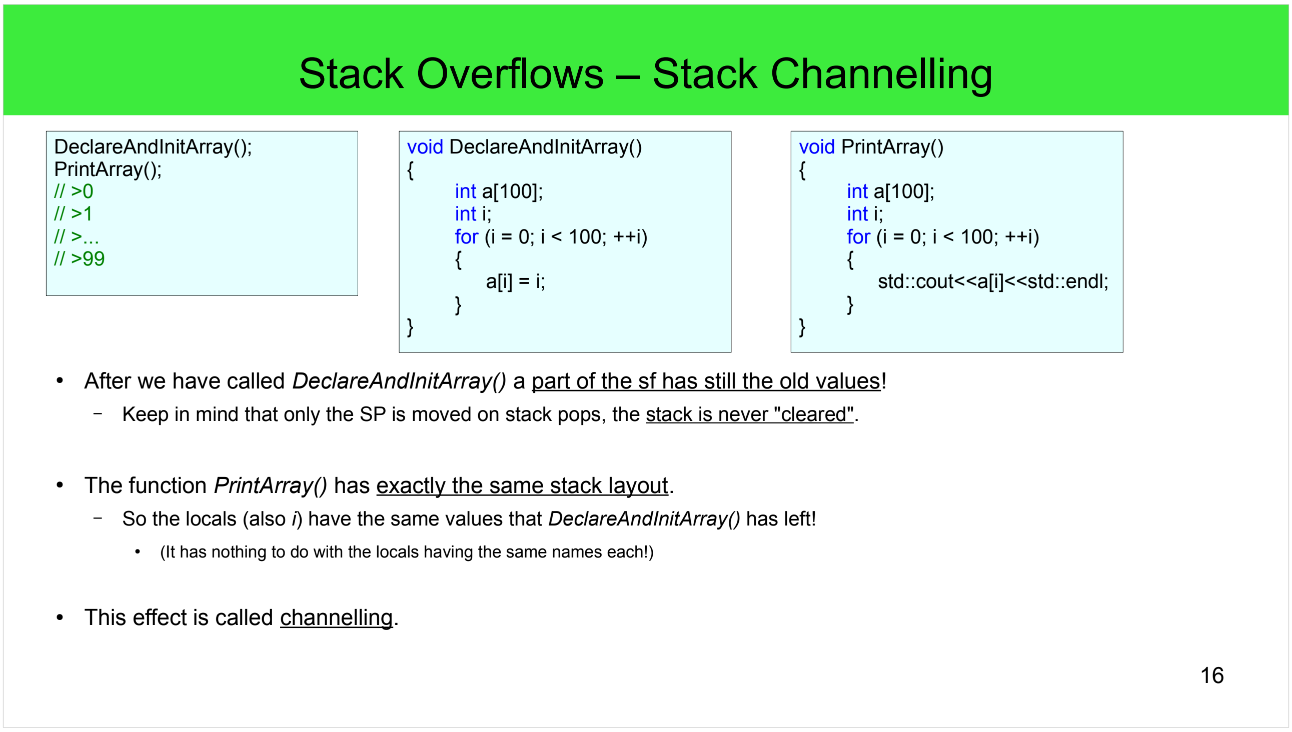
```
void Foo() {
        int array[4];
        int i;
        for (i = 0; i <= 4; ++i) {
            array[i] -= 4;
        }
}
```

```
Foo();
```

?	?	?	?	SPC	*array*[4]
?	?	?	?	*array*[3]	
?	?	?	?	*array*[2]	
?	?	?	?	*array*[1]	
?	?	?	?	*array*[0]	
?	?	?	?	*i*	

- Same error, but *array* is now on a higher address than *i*, and the elements are decremented by 4.
  - When *i* reaches the value 4, erroneously the SPC is addressed!
  - Then the content of the SPC (i.e. *Foo()*'s return address) is decremented by 4.
  - The SPC - 4 is exactly the address from where *Foo()* was called!
  - The new return address in the SPC will now return to the call address of *Foo()*!
  - Finally *Foo()* will be called again. (The -4 is a "negative one instruction" in our case.)
  - => It will end (or never end) in an infinite call chain.

- The effect shown above is present in our memory model. Whether this effect emerges is highly dependent on our platform (e.g. calling convention). Some runtimes can spot the error on the stack (e.g. gcc/macOS). – Nevertheless it is an error!
- What makes a buffer overrun:
  - This is the hard part: Inject a piece of malicious code into the memory.
  - The program executes the function *f()* as normal path of execution.
  - Between the arguments and the locals of *f()*, the return address is stored. Let's call the locals in common "buffer on the stack". If we write over the upper address-boundary of this buffer, we can overwrite the return address. The new address will then point to the location of malicious code, we had injected before.
  - When *f()* returns, program execution will continue with executing the malicious code we had injected before.
- (The interception of signals, i.e. that of a cable or wireless signals, is also a kind of hacking but minimally invasive.(



Stack Overflows – Stack Channelling

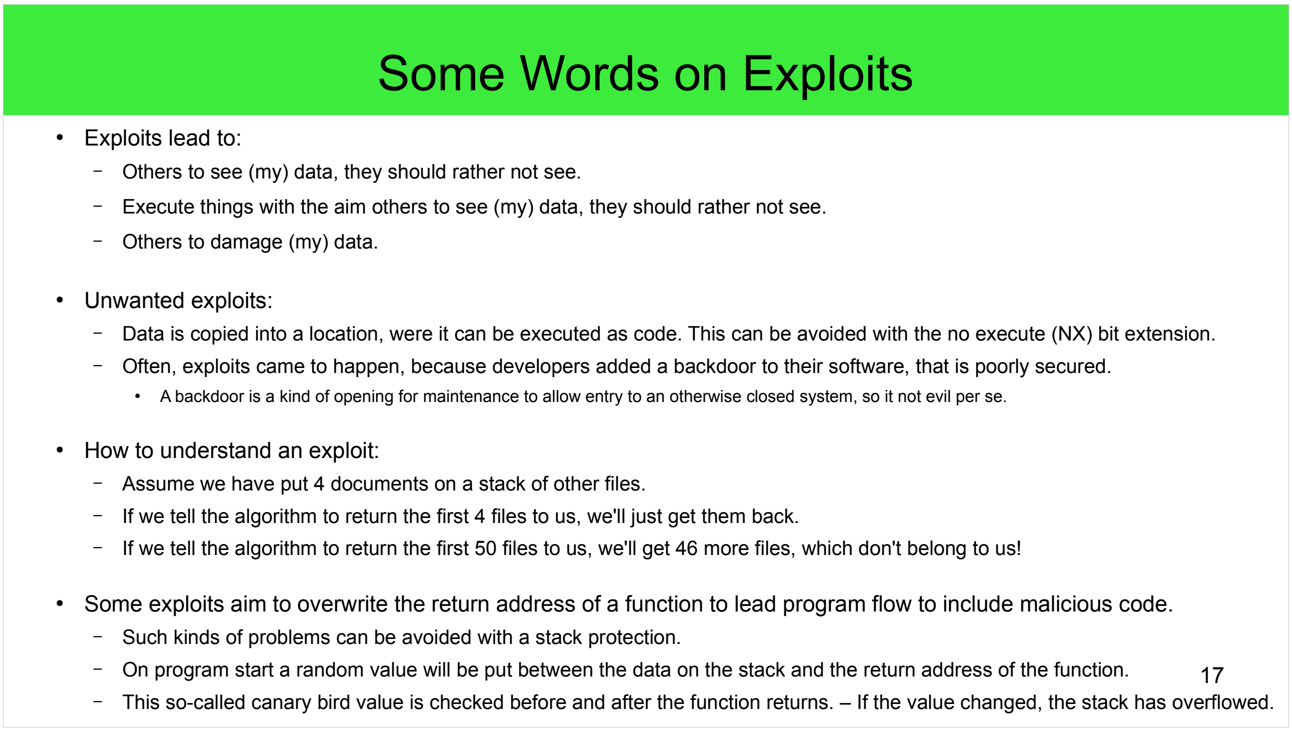
```
DeclareAndInitArray();
PrintArray();
// >0
// >1
// >...
// >99
```

```
void DeclareAndInitArray()
{
        int a[100];
        int i;
        for (i = 0; i < 100; ++i)
        {
                a[i] = i;
        }
}
```

```
void PrintArray()
{
        int a[100];
        int i;
        for (i = 0; i < 100; ++i)
        {
                std::cout<<a[i]<<std::endl;
        }
}
```

- After we have called *DeclareAndInitArray()* a part of the sf has still the old values!
  - Keep in mind that only the SP is moved on stack pops, the stack is never "cleared".
- The function *PrintArray()* has exactly the same stack layout.
  - So the locals (also *i*) have the same values that *DeclareAndInitArray()* has left!
    - (It has nothing to do with the locals having the same names each!)
- This effect is called channelling.

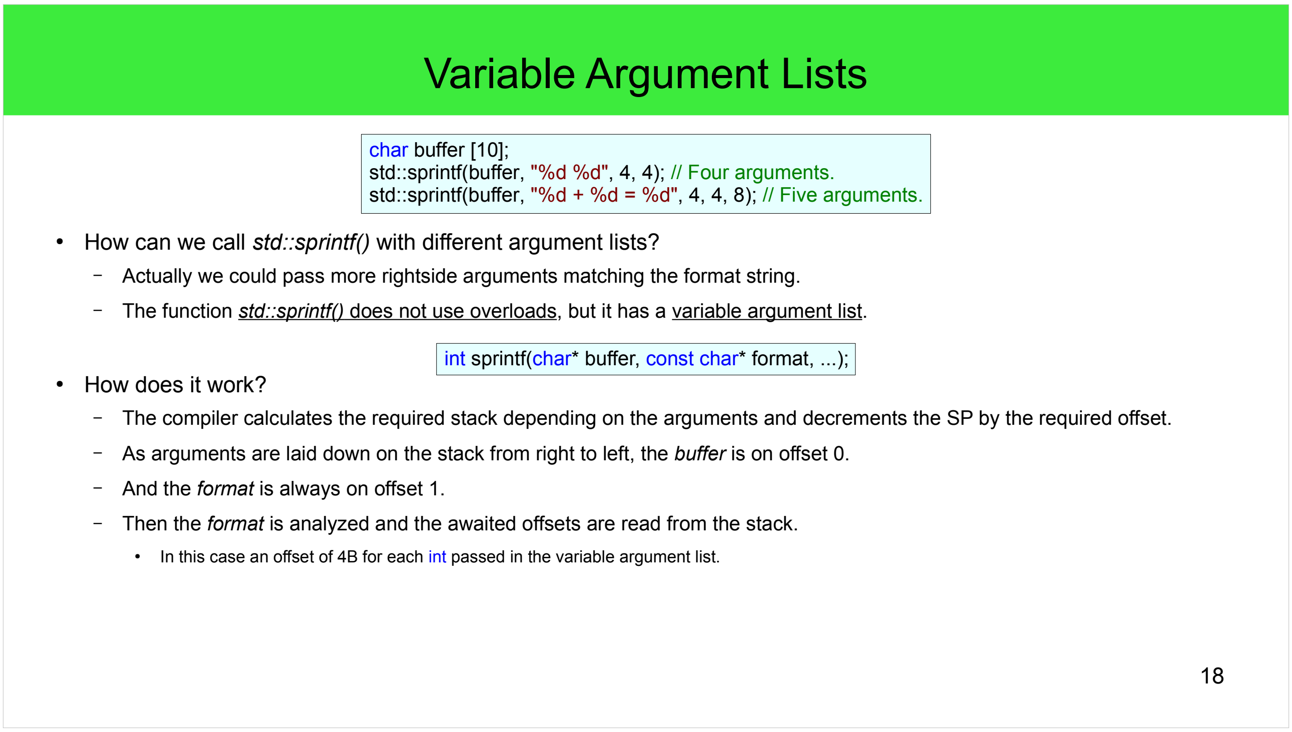
- Stack channelling is interesting for hardware near code as we find it in drivers.
- It should be said that all these manipulations on the stack can still lead to undefined behavior. This is because we are often about to write memory that is not owned by us, and also mind that the stack could be differently organized on different platforms (e.g. no decrementing stack).



Some Words on Exploits

- Exploits lead to:
  - Others to see (my) data, they should rather not see.
  - Execute things with the aim others to see (my) data, they should rather not see.
  - Others to damage (my) data.
- Unwanted exploits:
  - Data is copied into a location, were it can be executed as code. This can be avoided with the no execute (NX) bit extension.
  - Often, exploits came to happen, because developers added a backdoor to their software, that is poorly secured.
    - A backdoor is a kind of opening for maintenance to allow entry to an otherwise closed system, so it not evil per se.
- How to understand an exploit:
  - Assume we have put 4 documents on a stack of other files.
  - If we tell the algorithm to return the first 4 files to us, we'll just get them back.
  - If we tell the algorithm to return the first 50 files to us, we'll get 46 more files, which don't belong to us!
- Some exploits aim to overwrite the return address of a function to lead program flow to include malicious code.
  - Such kinds of problems can be avoided with a stack protection.
  - On program start a random value will be put between the data on the stack and the return address of the function.
  - This so-called canary bird value is checked before and after the function returns. – If the value changed, the stack has overflowed.

- A meaningful/very serious exploit was present in the OpenSSL library from 2011 to 2014: the Heartbleed-exploit.
- There is also the term "protected mode", which refers to a special operation mode of x86 CPUs. As the name suggests, this mode provides some security: it restricts processes to access only certain regions in memory. – However, maybe the main reason for its introduction was, that this mode allows a process to address more memory directly.



Variable Argument Lists

```
char buffer [10];
std::sprintf(buffer, "%d %d", 4, 4); // Four arguments.
std::sprintf(buffer, "%d + %d = %d", 4, 4, 8); // Five arguments.
```

- How can we call *std::sprintf()* with different argument lists?
  - Actually we could pass more rightside arguments matching the format string.
  - The function *std::sprintf()* does not use overloads, but it has a variable argument list.

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

- How does it work?
  - The compiler calculates the required stack depending on the arguments and decrements the SP by the required offset.
  - As arguments are laid down on the stack from right to left, the *buffer* is on offset 0.
  - And the *format* is always on offset 1.
  - Then the *format* is analyzed and the awaited offsets are read from the stack.
    - In this case an offset of 4B for each int passed in the variable argument list.

- 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).

The Mystery of returning Cstring Literals

- We know that we can't return pointers to stack elements from a function.
  - The pointers are meaningless to the caller, as the memory is already stack-popped:

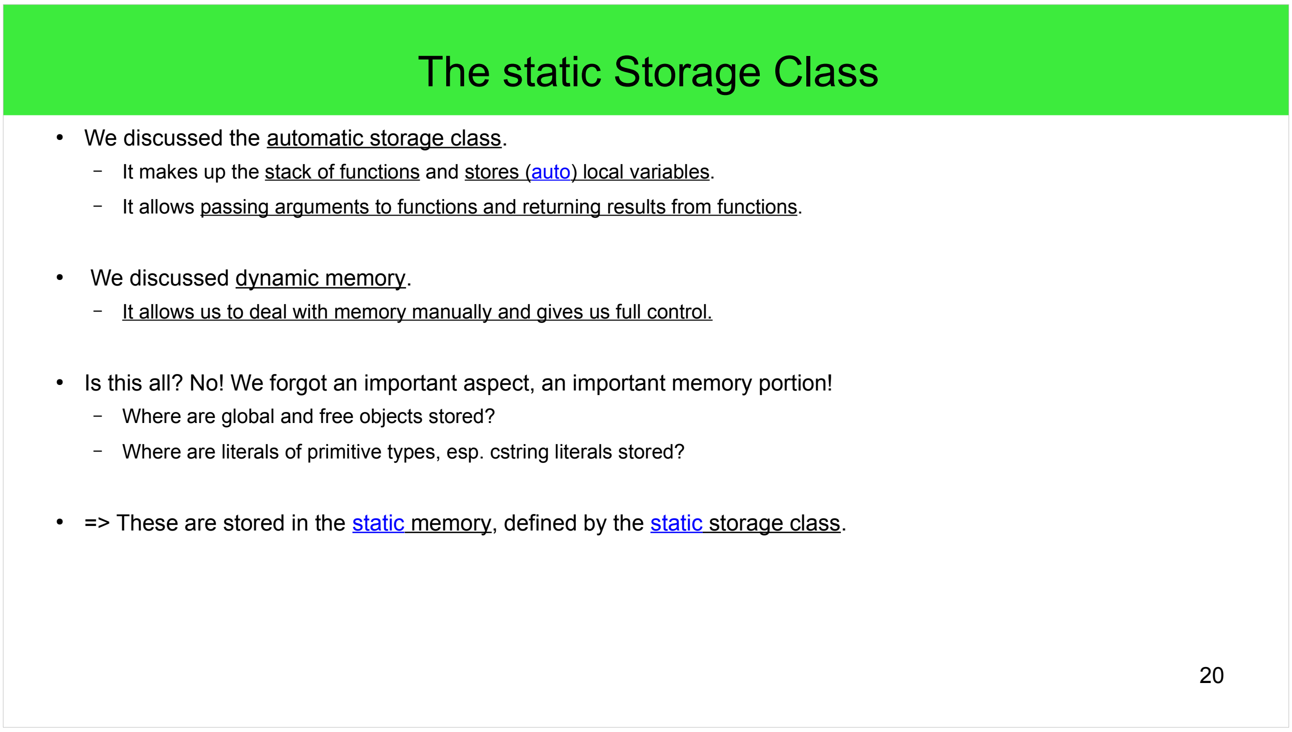
```
int* getValues() {              // Defining a function that returns a pointer to the locally
    int values[] = {1, 2, 3}; // defined array (created on stack).
    return values;            // This pointer points to the 1st item of values.
}
```

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

- But cstring literals can be legally returned! – How can that work?

```
const char* getString() { // Defining a function that returns a cstring literal.
    return "Hello there!";
}
```

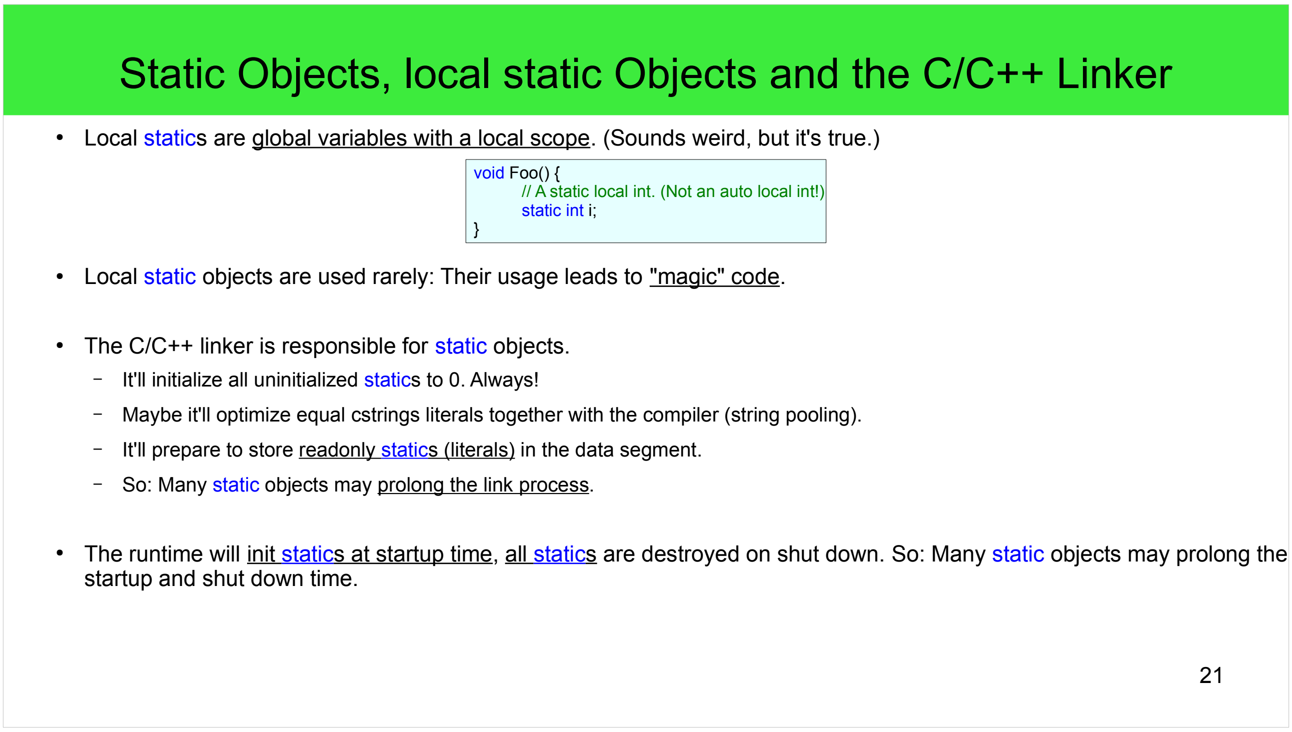
```
const char* aString = getString();
std::cout<<"The returned cstring is: "<<aString<<std::endl; // Ok!
// >The returned cstring is: Hello there!.
```



The static Storage Class

- We discussed the automatic storage class.
  - It makes up the stack of functions and stores (auto) local variables.
  - It allows passing arguments to functions and returning results from functions.
- We discussed dynamic memory.
  - It allows us to deal with memory manually and gives us full control.
- Is this all? No! We forgot an important aspect, an important memory portion!
  - Where are global and free objects stored?
  - Where are literals of primitive types, esp. cstring literals stored?
- => These are stored in the static memory, defined by the static storage class.

- Dynamic memory is not an explicit storage class in C/C++.



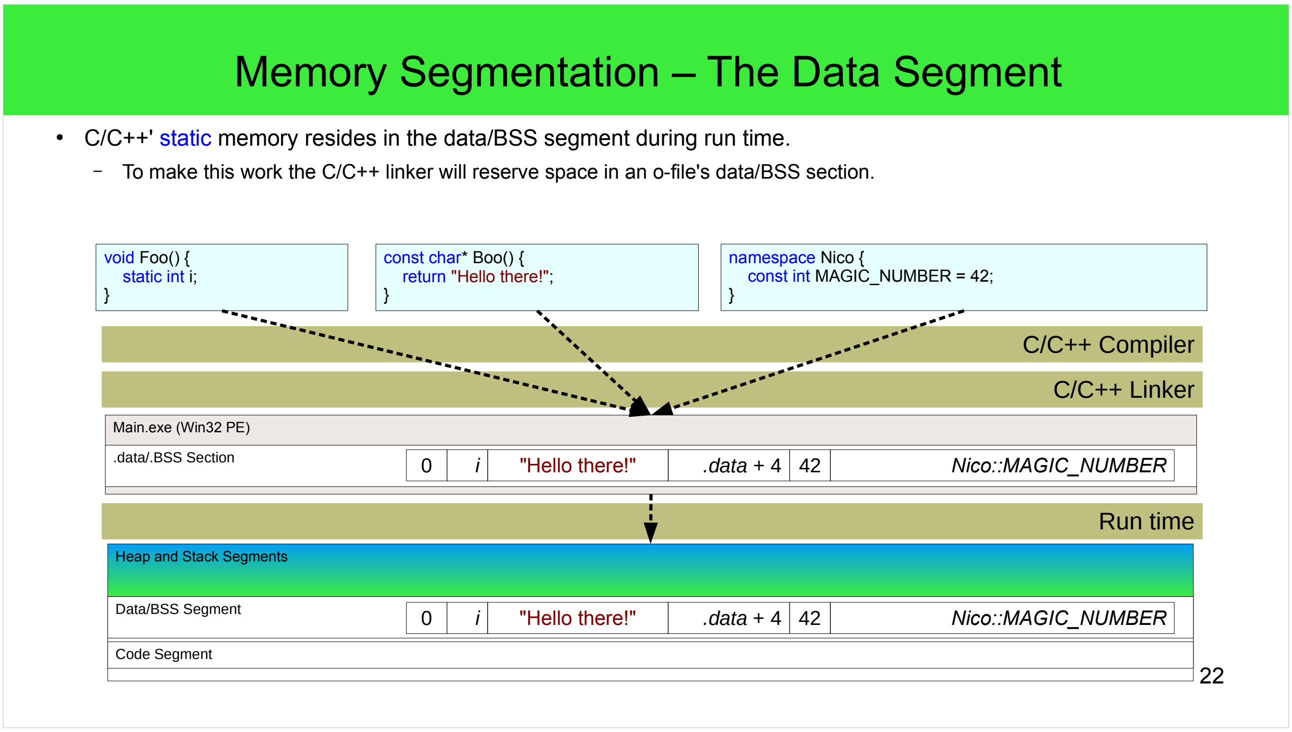
Static Objects, local static Objects and the C/C++ Linker

- Local statics are global variables with a local scope. (Sounds weird, but it's true.)

```
void Foo() {
        // A static local int. (Not an auto local int!)
        static int i;
}
```

- Local static objects are used rarely: Their usage leads to "magic" code.
- The C/C++ linker is responsible for static objects.
  - It'll initialize all uninitialized statics to 0. Always!
  - Maybe it'll optimize equal cstrings literals together with the compiler (string pooling).
  - It'll prepare to store readonly statics (literals) in the data segment.
  - So: Many static objects may prolong the link process.
- The runtime will init statics at startup time, all statics are destroyed on shut down. So: Many static objects may prolong the startup and shut down time.

- Why string pooling?
  - Because it can reduce the size of the resulting executable!
- The initialization/destruction strategy of non-local statics should be clear. Why?
  - Well, "globals" need to be initialized before the program runs and destroyed when the program ends.
- So: all statics have the lifetime of the program!
- The initialization order of non-local statics is undefined (it often depends on the link procedure), but some standard C++ objects like *std::cout* and *std::cin* are guaranteed to be initialized before any user defined non-local is initialized.



Memory Segmentation – The Data Segment

- C/C++' static memory resides in the data/BSS segment during run time.
  - To make this work the C/C++ linker will reserve space in an o-file's data/BSS section.

```
void Foo() {
    static int i;
}
```

```
const char* Boo() {
    return "Hello there!";
}
```

```
namespace Nico {
    const int MAGIC_NUMBER = 42;
}
```

C/C++ Compiler

C/C++ Linker

Main.exe (Win32 PE)

.data/.BSS Section	0	*i*	"Hello there!"	*.data + 4*	42	*Nico::MAGIC_NUMBER*

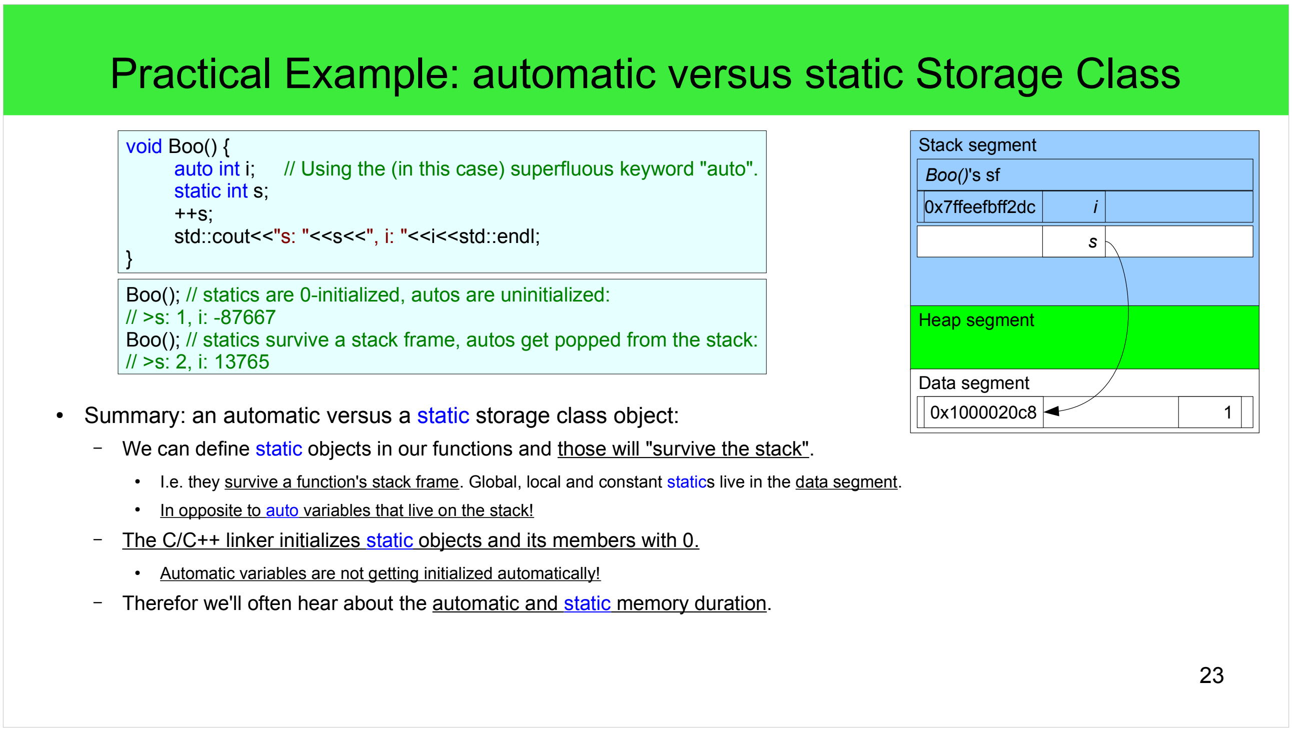
Run time

Heap and Stack Segments

Data/BSS Segment	0	*i*	"Hello there!"	*.data + 4*	42	*Nico::MAGIC_NUMBER*

Code Segment

- The .BSS section/segment (historical abbreviation for Block Started by Symbol) is a part of the .data section/segment that is dedicated to static/global objects that are not explicitly initialized by the programmer (like *i*).
- Keeping data and code in the same memory is an important aspect of the "von Neumann architecture".
  - It is also a typical problem of the "von Neumann architecture", that data and code share the same address space: data could overwrite code (usually accidentally) or data could be executed as code (usually intentionally as attack).
  - The presentation of this memory is a simplified version of real mode memory, where the memory separation into data and code segment was introduced. Basing on the real mode, the protected mode was developed: If code tried executing data in the data segment, the CPU would issue a hardware interrupt that would immediately stop program execution.
  - Modern OS' also use the protected mode, but with a flat memory model, where all segments reside in the same linear address range. So, the above mentioned segment based protection doesn't work. Instead of segments, OS' rely on pages. As pages can only be marked as being readonly or read/write, additional information was needed to mark code as being not executable. – The No eXecute (NX) bit was introduced by AMD (at AMD it is also called Enhanced Virus Protection (EVP), Intel calls it eXecute Disable (XD) bit and Microsoft calls it Data Execution Prevention (DEP)). – Trying to execute code in "NX-memory", will again issue a hardware interrupt. Other CPU manufactures (e.g. IBM/PowerPC) had similar technologies much earlier.



Practical Example: automatic versus static Storage Class

```
void Boo() {
        auto int i;     // Using the (in this case) superfluous keyword "auto".
        static int s;
        ++s;
        std::cout<<"s: "<<s<<", i: "<<i<<std::endl;
}
```

```
Boo(); // statics are 0-initialized, autos are uninitialized:
// >s: 1, i: -87667
Boo(); // statics survive a stack frame, autos get popped from the stack:
// >s: 2, i: 13765
```

- Summary: an automatic versus a static storage class object:
  - We can define static objects in our functions and those will "survive the stack".
    - I.e. they survive a function's stack frame. Global, local and constant statics live in the data segment.
    - In opposite to auto variables that live on the stack!
  - The C/C++ linker initializes static objects and its members with 0.
    - Automatic variables are not getting initialized automatically!
  - Therefor we'll often hear about the automatic and static memory duration.

- In C/C++ there exist following storage classes: auto, static, register, extern and mutable. Esp. the storage classes extern and mutable need more discussion in future lectures. The register keyword is generally ignored by today's compilers, it was esp. relevant for C in the late 1970ies.

Thank you!