

Lecture 15: A Closer Examination

Bart Iver van Blokland

Projects

- Deadline to deliver project on blackboard: 11th of April
- Deadline to demonstrate project: 30th of April
 - NOTE! Only a few working days after Easter
 - Your project will **only** be graded when demonstrated
 - There are TA's available on both insperaøving days
 - Demonstrating before Easter is possible

Insperaøving

- You may bring **handwritten** notes on a **stamped** A5 sheet of paper
 - These are available in the course rooms
 - Same goes for the final exam

Fanfare Lecture – 29.04.2025

- Want to do anything?
Show your work yourself?
Got any other ideas for fun stuff?
 - Send me an email: bart.van.blokland@ntnu.no
- Focus: show everything that is cool, hand out prizes for best projects
(I'll do my best to make it festive)

Next week: bonus lecture – parallel computing

- Bonus lecture:
 - Introduction to parallel computing (using multiple execution threads)
 - Not part of the syllabus

Last week

Last week

x0	x1	x2	...	x8	x9	...	x30	x31	sp

- Processors execute instructions on registers
 - Each register (usually) holds one 64-bit value
- Instructions performing computations read their operands from and write their results to registers
- Data must be explicitly read from and written to memory with instructions before it can be processed
- Many variables are never written to memory by the compiler, and remain entirely in a register

```
// Add 42 to value in register x0  
// store the result in x1  
add x1, x0, #42
```

```
// Load from memory  
ldr x5 [x9]  
// Write to memory  
str x5 [x9]
```

sum(5, 6) →

```
1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }
```

```
1 sum(long, long):
2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret
```

<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp
5	6	??	??	1337



Since our parameters are basic number data types, they are stored in registers by convention

sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

```

<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp	
5	6	??	??	1337	← stack pointer

Stack memory



Address 1337

Stack variables
from other functions

sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

```

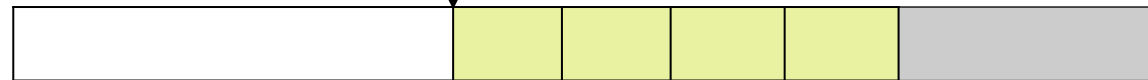
<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp
5	6	??	??	1305

← stack
pointer

Stack memory



Address $1337 - 32 = 1305$

In one instruction
we made space for
4 8-byte values!

sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

1 sum(long, long):

```

2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

```

<https://godbolt.org/z/sqEqh9K1s>

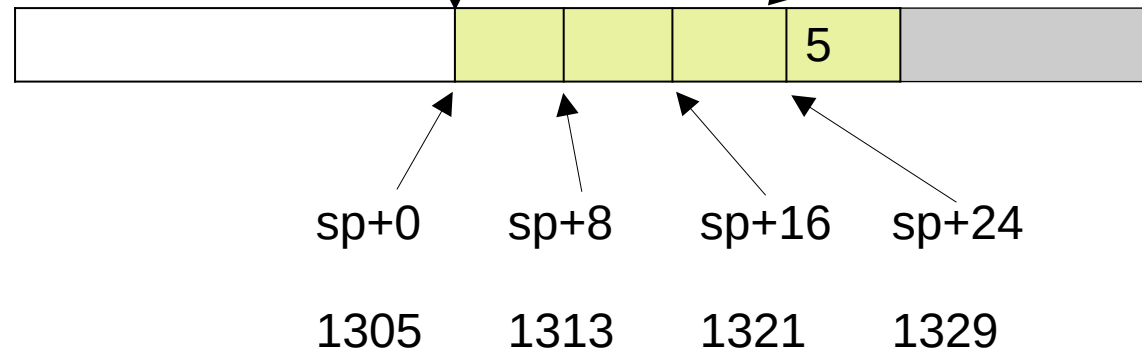
Registers

x0	x1	x8	x9	sp
5	6	??	??	1305

sum(5, 6) →

← stack
pointer

Stack memory



```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     sub     sp, sp, #32
3     str     x0, [sp, #24]
4     str     x1, [sp, #16]
5     ldr     x8, [sp, #24]
6     ldr     x9, [sp, #16]
7     add     x8, x8, x9
8     str     x8, [sp, #8]
9     ldr     x0, [sp, #8]
10    add     sp, sp, #32
11    ret

```

<https://godbolt.org/z/sqEqh9K1s>

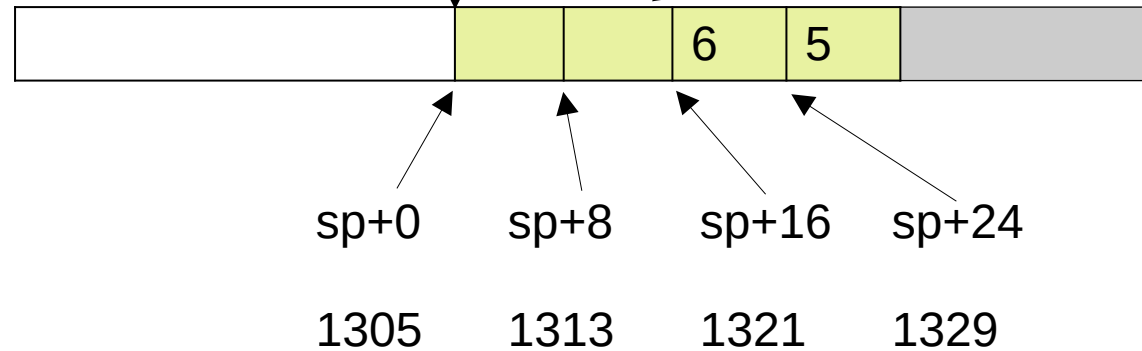
Registers

x0	x1	x8	x9	sp
5	6	??	??	1305

sum(5, 6) →

← stack
pointer

Stack memory



```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

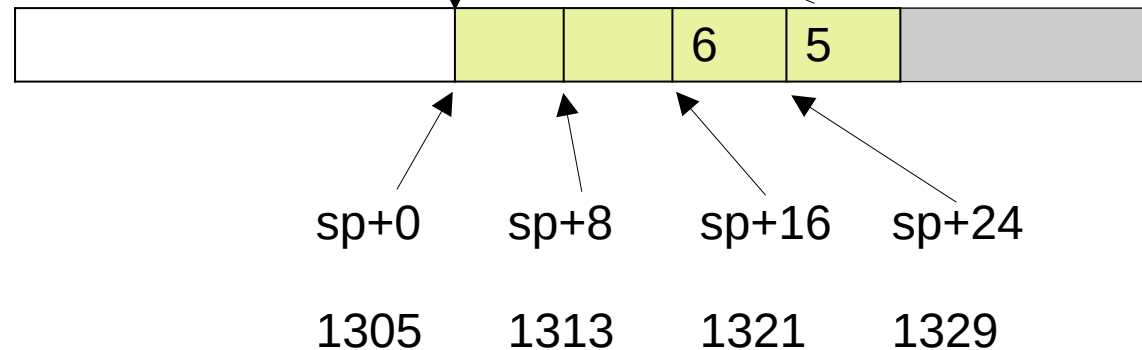
```

<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp	
5	6	5	??	1305	← stack pointer

Stack memory



sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

1 sum(long, long):

```

2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

```

<https://godbolt.org/z/sqEqh9K1s>

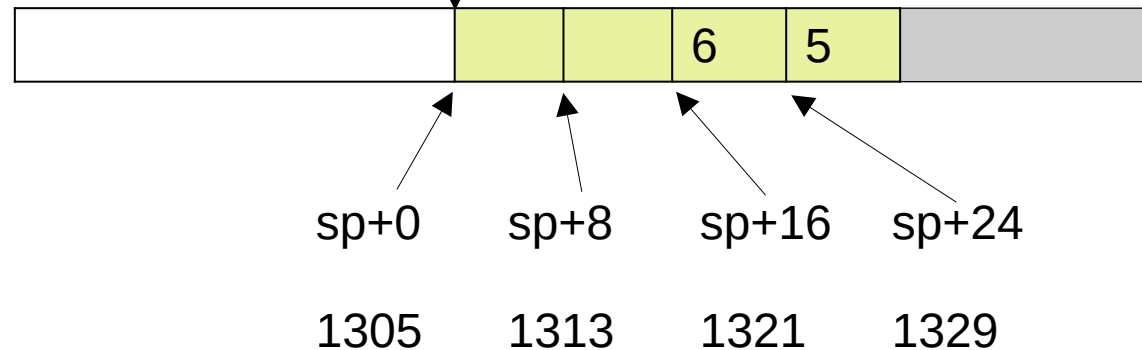
Registers

x0	x1	x8	x9	sp
5	6	5	6	1305

sum(5, 6) →

← stack
pointer

Stack memory



```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

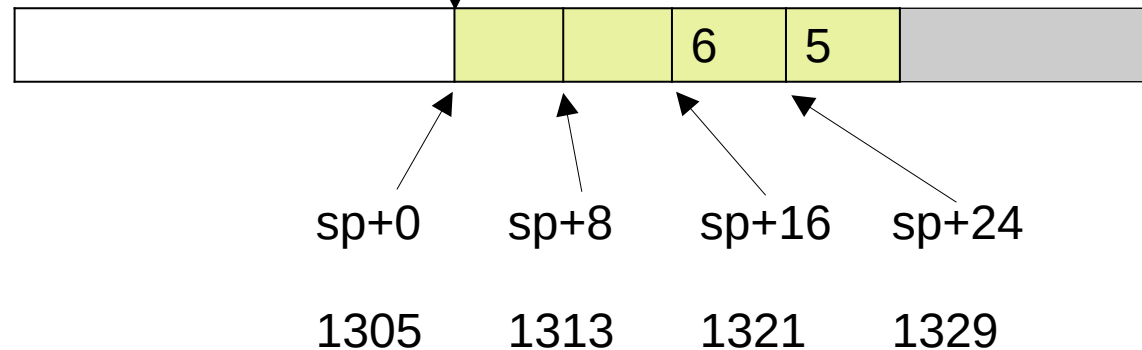
```

<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp	
5	6	11	6	1305	← stack pointer

Stack memory



sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

1 sum(long, long):

```

2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

```

<https://godbolt.org/z/sqEqh9K1s>

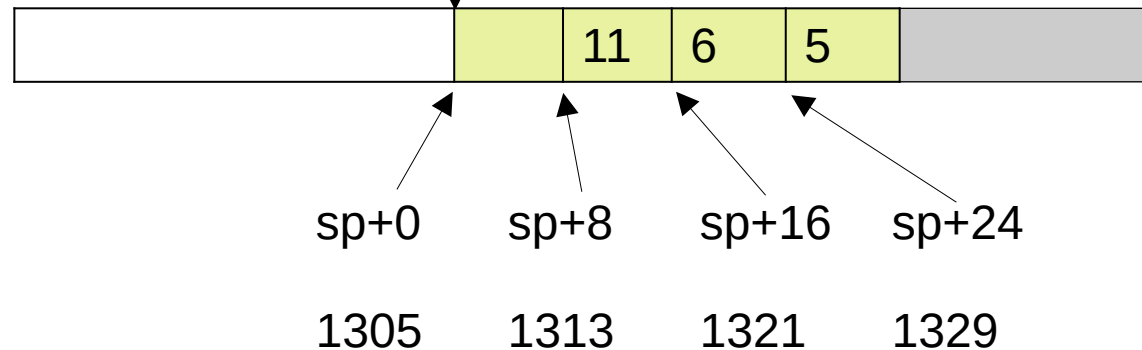
Registers

x0	x1	x8	x9	sp
5	6	11	6	1305

sum(5, 6) →

← stack
pointer

Stack memory



```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

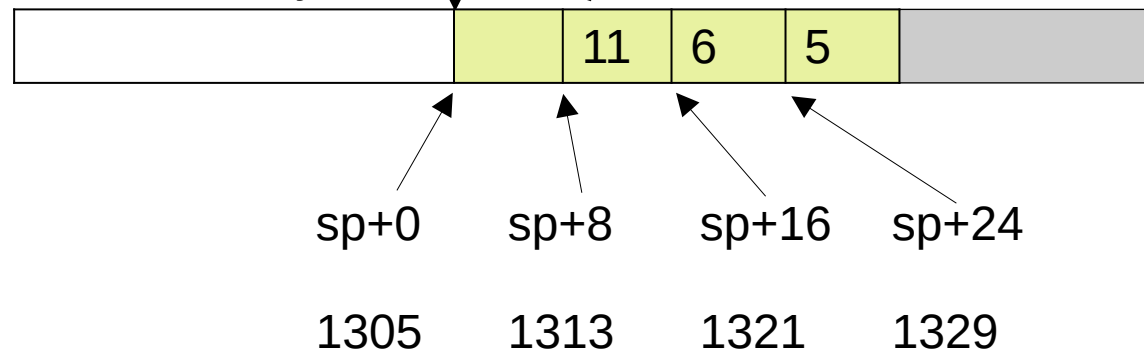
```

<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp	
11	6	11	6	1305	← stack pointer

Stack memory



sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

1 sum(long, long):

```

2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

```

<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp
11	6	11	6	1337

← stack
pointer

Stack memory



Moving the stack pointer back to its original position effectively deallocates all function variables in one instruction!

sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

```

<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp	← stack pointer
11	6	11	6	1337	

Stack memory



The processor now continues execution where the previous function left off.

The stack pointer is now where that function expects it to be.

sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     sub    sp, sp, #32
3     str    x0, [sp, #24]
4     str    x1, [sp, #16]
5     ldr    x8, [sp, #24]
6     ldr    x9, [sp, #16]
7     add    x8, x8, x9
8     str    x8, [sp, #8]
9     ldr    x0, [sp, #8]
10    add    sp, sp, #32
11    ret

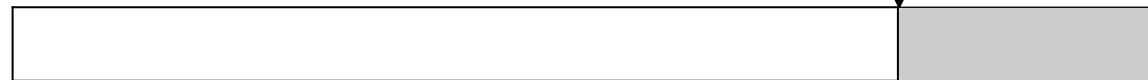
```

<https://godbolt.org/z/sqEqh9K1s>

Registers

x0	x1	x8	x9	sp
5	6	??	??	1337

Stack memory



sum(5, 6) →

```

1 long sum(long a, long b) {
2     long sumAB = a + b;
3     return sumAB;
4 }

```

```

1 sum(long, long):
2     add     x0, x1, x0
3     ret

```

Enabling release mode makes the function quite a bit shorter, but the result is still exactly the same!

<https://godbolt.org/z/sqEqh9K1s>

Lecture 15: A Closer Examination

Bart Iver van Blokland

(for real this time!)

Today

- **Pointers and Memory**
- Operator overloading
- Exam info + tips

Pointers and Memory

- **Pointers and Memory Addresses**
- Pointers versus References
- References: pass-by-value vs pass-by-reference
- Memory: stack versus heap
- Memory: allocation and deallocation
- Memory: smart pointers
- Memory: arrays

Memory Addresses

- Each byte in memory has its own number, known as its *address*
 - Each consecutive byte has a consecutive address
- Modern systems tend to use 64-bit addresses, because 32-bit ones can only address up to 4GB

Address	Value
...	...
20,285,638	00000000
20,285,639	00000000
20,285,640	00000000
20,285,641	00000101
20,285,642	00000000
20,285,643	00000000
20,285,644	00000000
20,285,645	00000000
20,285,646	00000001
20,285,647	00110101
20,285,648	10001000
20,285,649	11000110
...	...

Pointers

- Pointer: a variable containing a memory address
- C++ also requires that you specify the data type of the value found at the referenced address
- The address of any variable can be obtained using the & operator:

```
int main() {  
    int value = 5;  
    int* ptr = &value;  
}
```


Pointers

- For example:

```
int main() {  
    int value = 5;  
    int* ptr = &value;  
}
```

ptr will have the value
20,285,638

Address	Value	Contents
...
20,285,638	00000000	int value
20,285,639	00000000	
20,285,640	00000000	
20,285,641	00000101	
20,285,642	00000000	int* ptr
20,285,643	00000000	
20,285,644	00000000	
20,285,645	00000000	
20,285,646	00000001	
20,285,647	00110101	
20,285,648	10001000	
20,285,649	11000110	
...

We don't expect you to be able to explain what is shown here on the exam. But understanding this does help :)

Pointers

- Syntax examples (declarations):

<code>float radius;</code>	Variable of type float
<code>float* radiusPtr;</code>	Pointer to float
<code>float *radiusPtr;</code>	Pointer to float
<code>std::vector<int>* radii;</code>	Pointer to a vector of int
<code>Item* nextItem;</code>	Pointer to type Item
<code>Color** image;</code>	Pointer to pointer to type Color

Pointers

- When we need pointers:
 - Class instances when using virtual methods
 - Older libraries often require them
 - Avoiding copying of data
 - Using heap allocated objects
 - Interoperability with C
 - Output parameters (shown later)

Pointer dereferencing

- Dereferencing a pointer: looking at or using the value that is stored at the address referenced by a pointer
- Done using the * operator:

```
int main() {  
    std::string value = "Hey";  
    std::string* ptr = &value;  
  
    // Using the referenced value  
    std::string copyOfValue = *ptr;  
    std::string added = *ptr + " there";  
  
    // Updating the referenced value  
    *ptr = "Greetings";  
}
```

Pointer dereferencing

- For objects: can use either the * or -> operator

```
int main() {  
    std::vector<int> numbers = {6, 2, 8, 4};  
    std::vector<int>* numbersPtr = &numbers;  
  
    // Both variants are equivalent:  
    int size1 = numbersPtr->size();  
    int size2 = (*numbersPtr).size();  
}
```

Common use case: output parameters

SDL_GetWindowSize

Get the size of a window's client area.

Syntax

```
void SDL_GetWindowSize(SDL_Window * window, int *w,  
                        int *h);
```

Function Parameters

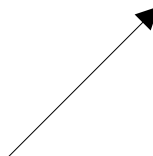
window	the window to query the width and height from
w	a pointer filled in with the width of the window, in screen coordinates, may be NULL
h	a pointer filled in with the height of the window, in screen coordinates, may be NULL

Common use case: output parameters

- Output parameter: a function parameter of a pointer type that is not used to pass a value into the function, but rather to return one from it
- One use: output more than one value from a function

```
TDT4102::Point TDT4102::AnimationWindow::getWindowDimensions() {  
    TDT4102::Point dimensions;  
    SDL_GetRendererOutputSize(rendererHandle, &dimensions.x,  
                                &dimensions.y);  
    return dimensions;  
}
```

The function writes both
output values directly into
our Point variable



&dimensions.x,
&dimensions.y);

Pointers and Memory

- Pointers and Memory Addresses
- **Pointers versus References**
- References: pass-by-value vs pass-by-reference
- Memory: stack versus heap
- Memory: allocation and deallocation
- Memory: arrays

Reference

- Effectively a pointer that is dereferenced automatically
(also implemented as such under the hood)
- Syntax:

```
std::string message = "Hello";  
std::string& referenceToMessage = message;  
  
// Updates the value of message:  
referenceToMessage = "Goodbye";  
  
// Reads the value of message:  
std::cout << referenceToMessage << std::endl;
```

Pointers vs References

	Pointers	References
References another value	Yes	Yes
Guaranteed to reference a value	No	Yes, unless you do something stupid
Can be set (initialised) to nullptr	Yes	No
Can be changed to nullptr	If not const	No
Can modify the address being referenced	If not const	No
Possible to create a vector or array containing these	Yes	No
Need to dereference explicitly	Yes	No

Best practice: use references when you have a choice!

Pointers and Memory

- Pointers and Memory Addresses
- Pointers versus References
- **References: pass-by-value vs pass-by-reference**
- Memory: stack versus heap
- Memory: allocation and deallocation
- Memory: arrays

Pass-by-reference vs pass-by-value

- Refers to the data types of function parameters
- How to tell:
 - Pass-by-reference: function parameter is a reference
 - Pass-by-value: function parameter is **not** a reference

```
void someFunction(std::string byValue, int& byReference) {  
  
}
```

Pass-by-reference vs pass-by-value

- What does it mean?
 - Pass-by-value: a copy is being created of the parameter value being passed in
 - Pass-by-reference: a reference to the variable being passed in is created

```
void someFunction(std::string byValue, int& byReference) {  
    // byValue is now a copy of 'text',  
    // but can be modified independently  
    // Modifying byReference here changes the value of 'number'  
}  
  
int main() {  
    std::string text = "Hello";  
    int number = 10;  
    someFunction(text, number);  
}
```

Pointers and Memory

- Pointers and Memory Addresses
- Pointers versus References
- References: pass-by-value vs pass-by-reference
- **Memory: stack versus heap**
- Memory: allocation and deallocation
- Memory: arrays

Memory: Stack versus Heap

- Two allocation types: stack and heap
 - Stack:
 - Small amount of memory (usually 1MB) for storing function variables and parameters
 - Automatically allocated / deallocated
 - Heap:
 - Potentially large amount of memory used for large data arrays or persistent data structures
 - Manually allocated / deallocated

Memory: Stack versus Heap

- Stack:
 - How to tell: function variables and parameters
 - Allocated and deallocated automatically
 - A stack variable:
 - Can be referenced by a pointer
 - Cannot be contained in a `std::shared_ptr` or `std::weak_ptr`

Memory: Stack versus Heap

- Heap:
 - How to tell:
 - Allocated using **new**
 - Referenced by a pointer
 - Must be manually allocated and deallocated
 - Done using **new** and **delete**
 - Data stored on the heap:
 - Must be referenced by a pointer
 - Should be contained in a `std::shared_ptr` or `std::weak_ptr` or `std::unique_ptr`

Memory: Stack versus Heap

- Note: whether something is allocated on the stack or heap is not a binary distinction. For example:

```
int main() {  
    std::vector<long> {4, 6, 3, 2};  
    std::string message = "Useful error message goes here";  
  
    int* heapInteger = new int {1};  
    return 0;  
}
```

`std::vector` and `std::string`: metadata (e.g. size) is stored on the stack, contents on the heap.

`heapInteger`: pointer referencing it is stored on the stack, and the integer's value on the heap.

Pointers and Memory

- Pointers and Memory Addresses
- Pointers versus References
- References: pass-by-value vs pass-by-reference
- Memory: stack versus heap
- **Memory: allocation and deallocation**
- Memory: arrays

Memory Allocation

- Stack memory:
 - Allocated at the beginning of the containing scope
 - Deallocated at the end of the containing scope

```
{           // a is allocated here
    int a = 5; // a is initialised here
}           // a is deallocated here
```

- Heap memory:
 - Allocated when using **new** or **new[]**
 - Deallocated when using **delete** or **delete[]**
 - Can (and should) be handled by a smart pointer

Smart Pointers

- Wrappers for pointers, whose destructors call **delete** or **delete[]**
- Big reason why this is needed:
Memory is deleted when an exception is thrown

Smart Pointers

- Variant 1: `std::unique_ptr`:
 - Cannot be copied, only moved with `std::move()`
 - Guarantees only one copy of a pointer exists
 - Deletes the memory when the object is destroyed
- Variant 2: `std::shared_ptr`:
 - Allows you to make copies
 - Counts the number of copies in existence. When that number hits 0, the memory is deleted

std::unique_ptr

- The contained address can only be moved from one pointer to another (guarantees uniqueness):

// pointer to int is stored in ptr

```
std::unique_ptr<int> ptr { new int{5} };
```

// pointer to int is moved to other, ptr is empty

```
std::unique_ptr<int> other { std::move(ptr) };
```

// pointer to int is back in ptr, other is empty

```
ptr = std::move(other);
```

Smart pointers

- Using pointers is the same as regular pointers:

```
std::unique_ptr<int> ptr { new int{5} };
```

```
*ptr = 15;  
std::cout << *ptr << std::endl;
```


Pointers and Memory

- Pointers and Memory Addresses
- Pointers versus References
- References: pass-by-value vs pass-by-reference
- Memory: stack versus heap
- Memory: allocation and deallocation
- **Memory: arrays**

Memory: (C-style) Arrays

- Allocate with `new[]`, deallocate with `delete[]`
- Use when a library requires them
- Otherwise, use `std::vector` and `std::array` as much as possible
- Constructors and destructors are called for each element

Memory: Arrays

- C-style arrays use a pointer to the start of the array:

```
float* array = new float[3];
```

Here the address stored in array would be 77,285,638

- Array elements are always stored next to each other.
- Kicker: no length is stored! Only where you can find the first element.

Address	Value	Contents
...
77,285,638	01100010	array[0]
77,285,639	01110010	
77,285,640	01100001	
77,285,641	00100000	
77,285,642	01101010	array[1]
77,285,643	01101111	
77,285,644	01100010	
77,285,645	01100010	
77,285,646	01100001	array[2]
77,285,647	00100000	
77,285,648	00111010	
77,285,649	00101001	
...

Memory: Arrays

- The `[]` operator reads the value at the address computed from the array's start pointer and the given index
- Since we only know where the array starts, we are responsible for ensuring we only request indices within the bounds of our array

Address	Value	Contents
...
77,285,638	01100010	array[0]
77,285,639	01110010	
77,285,640	01100001	
77,285,641	00100000	
77,285,642	01101010	array[1]
77,285,643	01101111	
77,285,644	01100010	
77,285,645	01100010	
77,285,646	01100001	array[2]
77,285,647	00100000	
77,285,648	00111010	
77,285,649	00101001	
...

Memory: (C-style) Arrays

- Another example: array of pointers

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

(argv points to the start of the array of pointers)

argv:

73,843,093	17,936,839	49,965,103	71,442,478	...
------------	------------	------------	------------	-----

Address	Value	Contents
...
73,843,093	00101110	'.'
73,843,094	00101111	'/'
73,843,095	01110000	'p'
73,843,096	01110010	'r'

Address	Value	Contents
...
17,936,839	00101101	'-'
17,936,840	00101101	'-'
17,936,841	01101000	'h'
17,936,842	01100101	'e'

Pointers and Memory

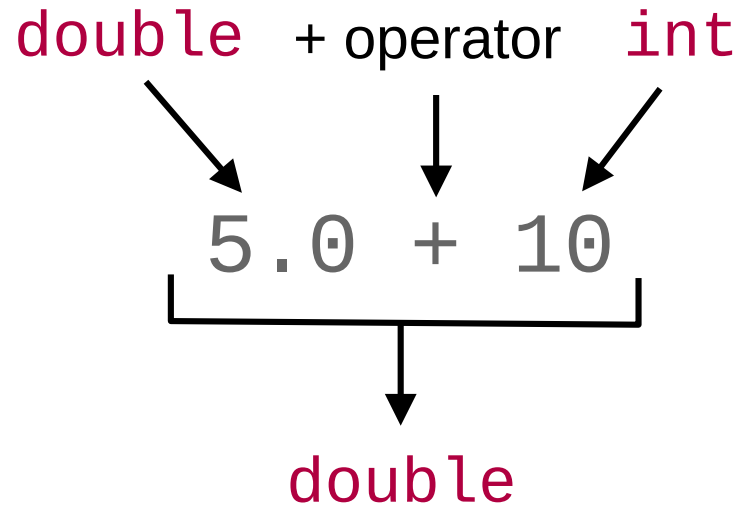
- Pointers and Memory Addresses
- Pointers versus References
- References: pass-by-value vs pass-by-reference
- Memory: stack versus heap
- Memory: allocation and deallocation
- Memory: arrays

Today

- Pointers and Memory
- **Operator overloading**
- Streams
- Exam info + tips

Operator overloading

You can think of an operator as a function that takes two parameters and returns another value



Operator overloading: syntax

value1 > value2

```
dataType operator>(dataType operand1, dataType operand2) {  
    // Implement the operator here  
}
```

The data type of the result of the operator
(for > that is usually **bool**)

Each dataType can be a different type!

Operator overloading

- Two variants:
 - Variant 1: standalone function
 - Must specify all parameters
 - Not all operators can be overloaded this way
 - Variant 2: method in class
 - Only right hand side parameter of the operator needs to be given as a parameter
 - All operators in the language can be overloaded in a class

Variant 1: Standalone function

```
type operator+ (type a, type b) {}  
type operator- (type a, type b) {}  
type operator* (type a, type b) {}  
type operator/ (type a, type b) {}  
type operator% (type a, type b) {}  
type operator^ (type a, type b) {}  
type operator& (type a, type b) {}  
type operator| (type a, type b) {}  
type operator, (type a, type b) {}  
type operator>> (type a, type b) {}  
type operator<< (type a, type b) {}  
type operator~ (type a) {}  
type operator! (type a) {}  
type operator++ (type a) {}  
type operator-- (type a) {}
```

```
type operator== (type a, type b) {}  
type operator!= (type a, type b) {}  
type operator&& (type a, type b) {}  
type operator|| (type a, type b) {}  
type operator< (type a, type b) {}  
type operator> (type a, type b) {}  
type operator<= (type a, type b) {}  
type operator>= (type a, type b) {}
```

For all of these, type can be replaced with **any** data type!

(exception: operators with data types that already exist, like `int + int`)

Variant 2: Class method

```
struct vec3 {  
    type operator[] (type index) {}  
    type operator() (type other) {}  
    type operator= (type other) {}  
    type operator+= (type other) {}  
    type operator-= (type other) {}  
    type operator*= (type other) {}  
    type operator/= (type other) {}  
    type operator%= (type other) {}  
    type operator^= (type other) {}  
    type operator&= (type other) {}  
    type operator|= (type other) {}  
    type operator>>= (type other) {}  
    type operator<<= (type other) {}  
};
```

These can **only** be declared as a member function!

The operators from the previous slide can be used too, except you leave out the first parameter for each of the operators listed there.

As before, type can be replaced with any other data type!

Operator overloading: notes

- Just like any other function, parameters to overloaded operators can be taken in as any data type, including (const) references
- And the same goes for return types. They can be anything you want
 - But you probably should return sensible types, like a bool from a < operator

Operator overloading: examples in STL

- `std::filesystem::path`:

```
std::filesystem::path path = "directory";  
path /= "directory2" / "file.txt";
```

- `std::string`

```
std::string message1 = "Hello";  
std::string message2 = "There";  
message1 += " " + message2;
```

- `std::ostream`

```
std::cout << "Did you ever hear the tragedy of Darth Plageuis the
```

Today

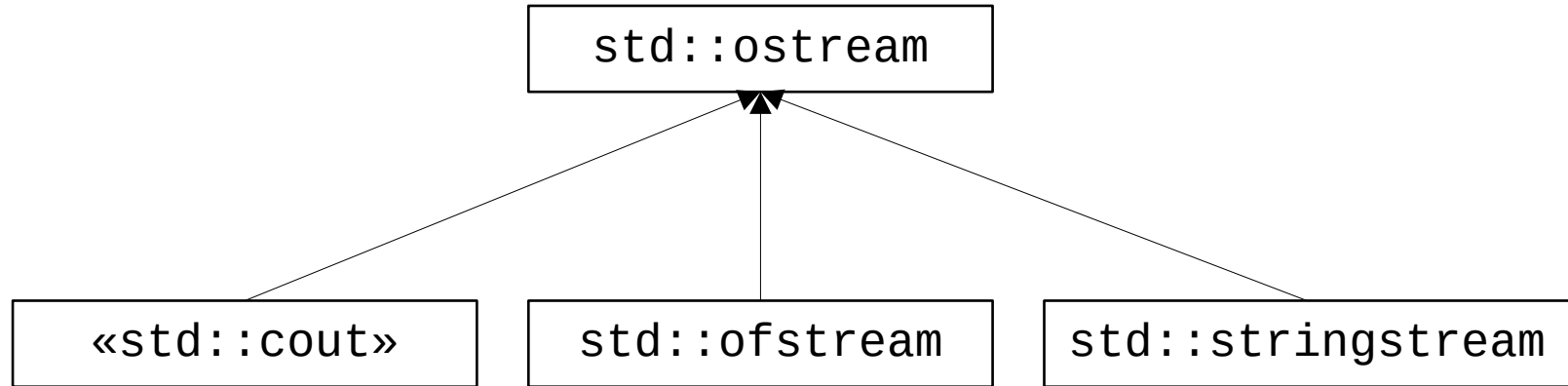
- Pointers and Memory
- Operator overloading
- **Streams**
- Exam info + tips

Stream model

- Intent: read and write data from and to places
- Sources:
 - `std::cin` (the terminal)
 - Files
 - `std::stringstream`
- Destinations:
 - `std::cout` (the terminal)
 - Files
 - `std::stringstream`

Stream model

- Implementation: relies on inheritance and operator overloading!



Stream model

- Implementation: relies on inheritance and operator overloading!
- Only basic types are supported by `std::ostream` itself

```
class ostream {  
    std::ostream& operator<<(bool __n);  
    std::ostream& operator<<(short __n);  
    std::ostream& operator<<(unsigned short __n);  
    std::ostream& operator<<(int __n);  
    std::ostream& operator<<(unsigned int __n);  
    std::ostream& operator<<(long __n);  
    std::ostream& operator<<(unsigned long __n);  
    std::ostream& operator<<(long long __n);  
    std::ostream& operator<<(unsigned long long __n);  
    std::ostream& operator<<(float __f);  
    std::ostream& operator<<(double __f);  
    std::ostream& operator<<(long double __f);  
    std::ostream& operator<<(const void* __p);  
};
```

Stream model

```
std::cout << -5 << std::endl;
```

```
class ostream {  
    std::ostream& operator<<(bool __n);  
    std::ostream& operator<<(short __n);  
    std::ostream& operator<<(unsigned short __n);  
    std::ostream& operator<<(int __n);  
    std::ostream& operator<<(unsigned int __n);  
    std::ostream& operator<<(long __n);  
    std::ostream& operator<<(unsigned long __n);  
    std::ostream& operator<<(long long __n);  
    std::ostream& operator<<(unsigned long long __n);  
    std::ostream& operator<<(float __f);  
    std::ostream& operator<<(double __f);  
    std::ostream& operator<<(long double __f);  
    std::ostream& operator<<(const void* __p);  
};
```

Stream model

```
std::cout << 33.6f << std::endl;
```

```
class ostream {  
    std::ostream& operator<<(bool __n);  
    std::ostream& operator<<(short __n);  
    std::ostream& operator<<(unsigned short __n);  
    std::ostream& operator<<(int __n);  
    std::ostream& operator<<(unsigned int __n);  
    std::ostream& operator<<(long __n);  
    std::ostream& operator<<(unsigned long __n);  
    std::ostream& operator<<(long long __n);  
    std::ostream& operator<<(unsigned long long __n);  
    std::ostream& operator<<(float __f);  
    std::ostream& operator<<(double __f);  
    std::ostream& operator<<(long double __f);  
    std::ostream& operator<<(const void* __p);  
};
```

Stream model

```
std::cout << 33.6 << std::endl;
```

```
class ostream {  
    std::ostream& operator<<(bool __n);  
    std::ostream& operator<<(short __n);  
    std::ostream& operator<<(unsigned short __n);  
    std::ostream& operator<<(int __n);  
    std::ostream& operator<<(unsigned int __n);  
    std::ostream& operator<<(long __n);  
    std::ostream& operator<<(unsigned long __n);  
    std::ostream& operator<<(long long __n);  
    std::ostream& operator<<(unsigned long long __n);  
    std::ostream& operator<<(float __f);  
    std::ostream& operator<<(double __f);  
    std::ostream& operator<<(long double __f);  
    std::ostream& operator<<(const void* __p);  
};
```

Stream model

- If anything is missing, we can create our own operator
- It can be used with all output streams because of inheritance!

```
std::ostream& operator<<(std::ostream& o, std::vector<int>& vec) {  
    o << "(";  
    for(int i = 0; i < vec.size(); i++) {  
        o << i;  
        if(i < vec.size() - 1) {  
            o << ", ";  
        }  
    }  
    o << ")";  
    return o;  
}
```

Stream model

- Possible exam question:
Why must the ostream object in the example below be a reference?

```
std::ostream& operator<<(std::ostream& o, std::vector<int>& vec) {  
    o << "(";  
    for(int i = 0; i < vec.size(); i++) {  
        o << i;  
        if(i < vec.size() - 1) {  
            o << ", ";  
        }  
    }  
    o << ")";  
    return o;  
}
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << "Guessed: " << a << ", answered: " << b << std::endl;
```


Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << "Guessed: " << a << ", answered: " << b << std::endl;
```

```
std::ostream& operator<<(std::ostream& o, std::string s) {  
    /* ... */  
    return o;  
}
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << a << ", answered: " << b << std::endl;
```

```
std::ostream& operator<<(std::ostream& o, int n) {  
    /* ... */  
    return o;  
}
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << ", answered: " << b << std::endl;
```

```
std::ostream& operator<<(std::ostream& o, std::string s) {  
    /* ... */  
    return o;  
}
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << b << std::endl;
```

```
std::ostream& operator<<(std::ostream& o, int n) {  
    /* ... */  
    return o;  
}
```

Stream model

- Returning the ostream object allows you to chain it:

```
std::cout << std::endl;
```

Input streams

- Input streams (which inherit from `std::istream`) allow the processing of input
- Uses the same overloaded operator mechanism as `std::ostream`

```
int x = 0;  
std::cin >> x;
```

// Calls:

```
class istream {  
    /* ... */  
    std::istream& operator>>(int& n) {}  
    /* ... */  
};
```

Input streams

- Input streams will separate input into «words»

```
Write your message:  Well          hello there
                    how
are
things                going?█
```

```
std::cout << "Write your message: ";
std::string word;
std::cin >> word; // Well
std::cin >> word; // hello
std::cin >> word; // there
std::cin >> word; // how
std::cin >> word; // are
std::cin >> word; // things
std::cin >> word; // going?
```

Input streams

- Input streams will separate input into «words»
 - But for numbers will only read until it encounters any letters

```
std::string word;  
int number;  
std::cout << "Write your message: ";  
std::cin >> word;  
std::cout << "word: " << word << std::endl;  
std::cin >> number;  
std::cout << "number: " << number << std::endl;  
std::cin >> word;  
std::cout << "word: " << word << std::endl;
```

```
Write your message: Hello 48 test  
word: Hello  
number: 48  
word: test
```

```
Write your message: Test 96word  
word: Test  
number: 96  
word: word
```


Today

- Pointers and Memory
- Operator Overloading
- Streams
- **Exam info + tips**

Exam: 02.06.2025

- Two sessions:
 - 9:00 – 13:00 + 15m for submission
 - 15:00 – 19:00 + 15m for submission

Check studentweb which group you are in!!

- Any exam information will be published on Blackboard

Exam: 02.06.2025

- Tips:
 - **READ THE QUESTION!!**
 - For open questions: be specific
 - Show us that you understand the concept
 - We give partial score for answers that are partially correct
 - For example, correctly describing how you would solve a task
- We ultimately look for what you do and don't understand

Exam: 02.06.2025

- Tips:

- **READ THE QUESTION!!**

- You do not need to understand all code that is handed out.
Focus on what each individual function needs to do
- Your code does not necessarily need to compile
 - For example, a misspelled variable name can still net you full score if the answer is otherwise correct
- Get familiar with cppreference.com
 - Search is unavailable on exam computers

Exam: 02.06.2025

- Tips:

- **READ THE QUESTION!!**

Next week

- Bonus lecture:
 - Introduction to parallel computing (using multiple execution threads)
 - Not part of the syllabus
 - Send me an email if you want me to talk about anything else

Fanfare Lecture: 29.04.2025

- Going through all the cool things done in projects
- I'll do my best to make it festive
- Want to show off your work yourself?
Other fun ideas? Send me an email!