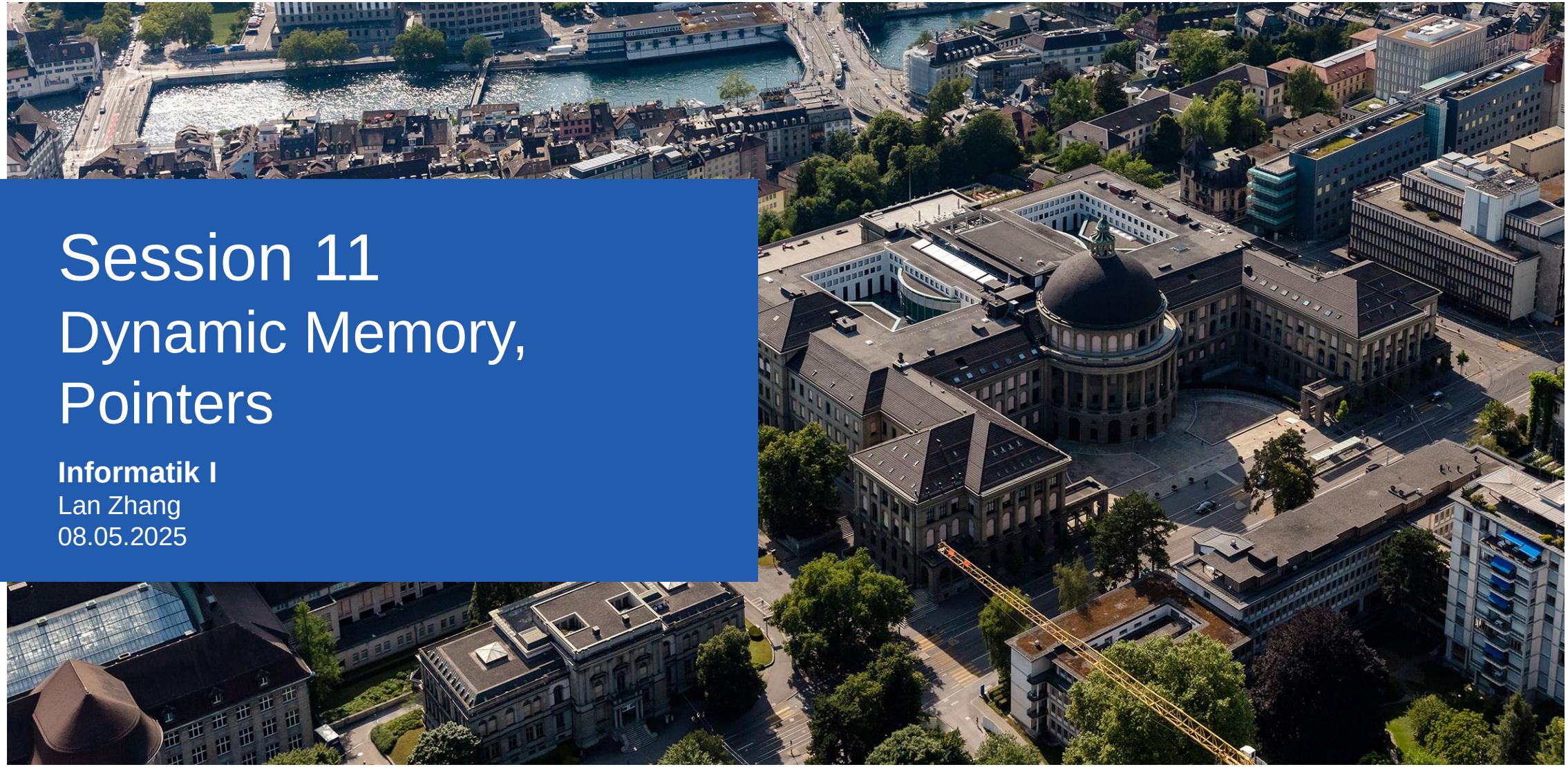
ETH zürich



Session 11
Dynamic Memory, Pointers

Informatik I

Lan Zhang

08.05.2025

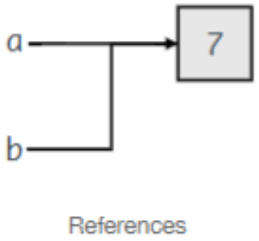
Schedule

1. Exercise Feedback
2. Theory Recap
   - Pointers
3. In-Class Code Examples
   - our_list::init
   - our_list::swap
   - our_list::extend

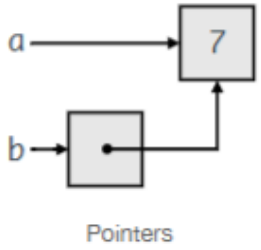
Pointers

- Pointers can change the address (of the referenced variable/object) they contain after initialization.
- Change target variable –> reference or address operator &
- Change value stored in target variable -> dereference or value operator *

Declaration: `type* ptr = address (&variable` or `otr_ptr` or `nullptr)`



References



Pointers

```
int a = 3;
int& b = a;
b = 7;
```

```
int* b = &a;  // b stores the address value of a (64-bit e.g. 0x32ef97)
*b = 7;  // *b accesses the value of a
```

Operators & and * in C++

& Operator:

1. Logical AND: `z = x && y;`
2. Declare a reference: `int& y = x;`
3. Address of a variable: `int *ptr_a = &a;`

* Operator:

1. Multiplication: `z = x * y;`
2. Declare a pointer: `int *ptr_a = &a;`
3. Dereference a pointer: `int a = *ptr_a;`

Pointer Syntax

The `this` Pointer

- Used in member functions (methods internal to a struct or class) to refer to the object instance.

```
struct Number {
  int number;
  void increment(int num){
    (*this).number = (*this).number + num;
  }
};
```

- (*this).number accesses the member variable number of the current object
- this->number is a shorthand for (*this).number

(ptr1->ptr2->ptr3->ptr4 instead of *(*((*prt1).ptr2).ptr3).ptr4...)

Pointer Syntax

- When accessing data members or member functions from within a class, the implicit argument (*`this`) is used automatically
- You do not necessarily have to specify it explicitly
- You must use `*this` explicitly if a reference to the implicit argument is to be returned

```
struct Number {
  int number;
  Number& increment(int num){
    number += num;
    return *this;
  }
};
```

Pointer Syntax

`const` pointer

- No write access to target: `const int* a_ptr = &a;`
- No write access to pointer: `int* const a_ptr = &a;`

Null pointer

- Does not point to any memory address, used to initialize an empty pointer
- The dereferencing of a `nullptr` leads to an error

In-Class Code Example

Linked List: Non-contiguous memory area, each element points to its successor. (No random access)

```
struct llnode{
   int value;
   llnode* next;
   llnode(int v, llnode* n): value(v), next(n) {} \\ Constructor
};
```

Vector as linked list

```
class llvec {
     llnode* head;
  public:
     llvec(int size);
     int size() const;
 };
```

In-Class Code Example

print

```
void llvec::print(std::ostream& out) const {
   for (llnode* n = this->head; n != nullptr; n = n->next) {
   out << n->value << ' ';
  }
}
```

operator[]

```
int& llvec::operator[](int i) {
   llnode* n= this->head;
   for (; 0 < i; --i) n = n->next;
   return n->value;
}
```

push_front

```
void llvec::pushfront(int e) {
 this->head = new llnode{e, this->head};
```

In-Class Code Example

`our_list::init`

- Initialize head to nullptr.
- Create a node for each new element and append to the list

`our_list::swap`

- Swap a given node with the next one by adjusting pointers.
- Note: Swap the actual nodes, not just their values.

`our_list::extend`

- Find the last node of the list.
- Add elements from the iterator to the linked list