ETH zürich

# Session 12
# Dynamic
# Data Structures II

**Informatik I**

Lan Zhang

22.05.2025

# Schedule

1. Exercise Feedback
2. Theory Recap
   - Pointer Arithmetics
   - Inheritance, Subtyping, and Polymorphism
3. In-Class Code Examples

# Pointer on Arrays

```
int* a = new int[5]{0, 8, 7, 2, -1};          // a points to first element
int* b = a;                                   // pointer assignment
++b;                                          // shift to the right
int my_int = *b;                              // read target
b += 2;                                       // shift by 2 elements
*b = 18;                                      // overwrite target
int* past = a+5;                              // past points behind last element
std::cout << (b < past) << "\n";              // compare pointers
delete[] a;                                   // delete memory block
```

https://lec.inf.ethz.ch/ifmp/2024/guides/tracing/dynamic.html

# Applying Pointers

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint valid ranges (e != o)
void f (const int* const b, const int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
// POST: The range [b, e) is copied in reverse order into the range [o, o+(e-b))
```

# In-Class Code Example

```
void our_vector::push_back(int new_element) {

   int* const new_elements = new int[this->count + 1];

   copy_range(this->elements, this->elements + this->count, new_elements);
   delete[] this->elements;

   new_elements[this->count] = new_element;
   this->count++;
   this->elements = new_elements;

 }
```

# Object-Oriented Programming

- Paradigm that organizes code into reusable, self-contained objects
- Improves code modularity

Polymorphism

- Methods with the same name but different behavior based on the object's type at runtime

Inheritance & Subtyping

- enables code reuse by allowing derived classes to inherit functionality from base classes
- allows a specific subtype to be used wherever a general type is expected

# Inheritance

```
class BinaryExp : public Exp {
  protected:   // Protected for derived classes to access!
      Exp* left;
      Exp* right;
  public:
      BinaryExp(Exp* l, Exp* r) : left(l), right(r) {}
      // No eval() here → force derived classes to implement it!
};

class Addition : public BinaryExp {
public:
    using BinaryExp::BinaryExp; // Inherit constructor
    double eval() const { return left->eval() + right->eval(); }
};
```