ETH zürich

Exercise Session 02 – Containers, Templates

Data Structures and Algorithms

These slides are based on those of the lecture, but were adapted and extended by the teaching assistant Adel Gavranović

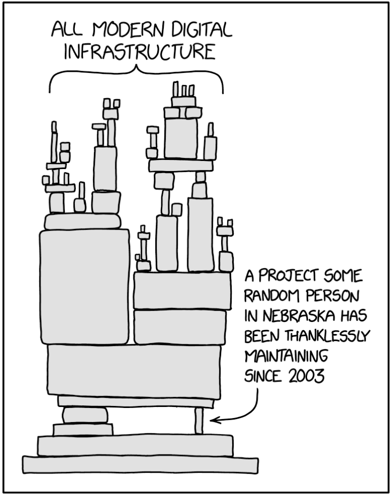
Today's Schedule

Intro
Follow-up
Feedback for **code** expert
Learning Objectives
C++ Container Library
Templates Recap
Repetition theory: Induction
Subarray Sum Problem
Code Example
Programming Exercise
Tips for **code** expert
Outro

n.ethz.ch/~agavranovic

- Exercise Session Material
- Adel's Webpage
- Mail to Adel

Comic of the Week



▸ xkcd

1. Intro

Intro

- Welcome Back!
- There was a miscommunication regarding exercise sessions in first week – Sorry for that!

2. Follow-up

Follow-up from last exercise session

- There's a **code** expert sandbox[1] now! (To try out code outside of exercises)
- That one confusing Runtime-Slide

[1] Can be found under "code examples" at the top

Slide from last session "A good strategy?"

If today I can solve a problem of size n (in some fixed time), then with a 10 or 100 times faster machine I can solve ...[2]

Complexity of Algorithm	(speed $\times 10$)	(speed $\times 100$)
$\log_2 n$	$n \rightarrow n^{10}$	$n \rightarrow n^{100}$
n	$n \rightarrow 10 \cdot n$	$n \rightarrow 100 \cdot n$
n^2	$n \rightarrow 3.16 \cdot n$	$n \rightarrow 10 \cdot n$
2^n	$n \rightarrow n + 3.32$	$n \rightarrow n + 6.64$

[2]To see this, you set $f(n') = c \cdot f(n)$ ($c = 10$ or $c = 100$) and solve for n'

Main Takeaway

- Faster computers won't be able to compensate for inefficient algorithms, since the increase in problem size that a significantly faster computer allows is uselessly small
  - e.g. from $n = 4$ to $n' \approx 7$ (per unit of time) in case of an algorithm of complexity $\mathcal{O}(2^n)$ if the new computer runs 10-times faster than the old
- Seriously, just write efficient code

3. Feedback for **code** expert

General things regarding **code** expert

- Nothing yet since the deadline for the current is tonight 23:59

Questions regarding **code** expert from your side?

4. Learning Objectives

Learning Objectives

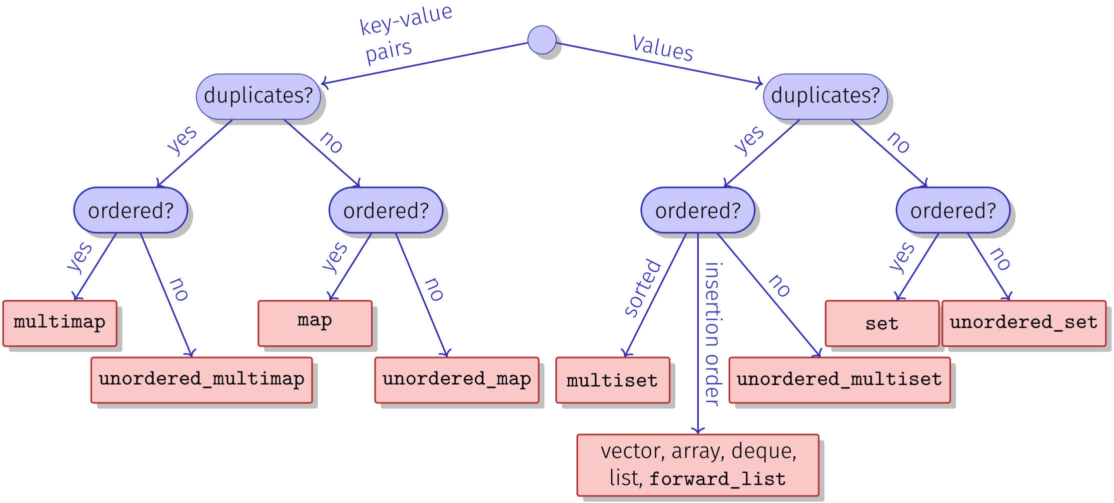
- ☐ Understand what Container are
- ☐ Understand what benefits Containers bring
- ☐ Understand what Templates are
- ☐ Understand what benefits Templates bring
- ☐ Understand how to do Induction Proofs in this course
- ☐ Be prepared to solve the next **code** expert exercises

5. C++ Container Library

What are containers *abstractly*?

- Essentially, a container is some sort of organized collection of things
- Each Container has its benefits and drawbacks
- Each Container has its use cases
- Don't bother learning them by heart,...
  - ...since you will be familiar with many of them by the end of this course because you will study some of them very closely
- Each Container comes with its own cool helper-functions!
  - e.g. `.push_back()` for our beloved `std::vector`

C++ Containers



Sequence-Container

`vector`	`array`	`deque`	`list`	`forward_list`
contiguous dynamic memory	contiguous static memory	Non-contig. dynamic memory	Non-contig. dynamic memory	Non-contig. dynamic memory
random access	random access	random access		
fast push/pop back		fast push/pop front/back	fast push/pop front/back	fast push/pop front
bidirectional iteration	bidirectional iteration	bidirectional iteration	bidirectional iteration	forward iteration

dynamic: size can change during runtime, **static**: size fixed at compile-time, **random acccess**: direct, immediate access to any element by its *index* (e.g. `vec[42]`), **bidirectional**: backward and forwards iterable

Sets and Multisets

- `std::set<E>` contains unique elements
- `std::multiset<E>` allows duplicate elements
  - Iteration yields all elements in decreasing order (in non-deterministic order if `unordered_multiset`)
  - `std::multiset<E>::count(elem)` returns the number of occurences of a given element

Example of `std::multiset`

```
Content: Xanten Xenon Xenon Xenon Xerografie Xerophil Xylose
count("Xenon") = 3
count("Xylose") = 1
```

Maps and Multimaps

- `std::map<K,V>` contains pairs (key, value), where a key maps to at most one value
- `std::multimap<K,V>` allows duplicate pairs
  - Iteration yields all pairs in descending key order (in non-deterministic order, if `unordered_multimap`)
  - `std::multimap<K,V>::count(key)` returns the number of occurrences of a given key
  - `std::multimap<K,V>::equal_range(key)` returns all values (in non-det. order) for a given key

Example of `std::multimap<K,V>`

```
Content: {2, er} {2, du} {2, es} {3, Axt} {3, sie} {4, Igel}
count(2) = 3
Values for key 2: er du es
```

6. Templates Recap

Motivation

Goal: generic binary tree without duplicating code

```
class Node { ... }; // Node of a binary search tree
auto n1 = Node<int>(5);
auto n2 = Node<std::string>("Zürich");
n1.insert(1);
n2.contains(2); // Compiler error
```

Idea:

- Make classes and functions parametric in types (= template parameters) ...
- ... just as they are already parametric in values (= function parameters)

Types as Template Parameters

1. In the concrete implementation of a class replace the type that should become generic (e.g. `int`) by a representative element, e.g. `T`.
2. Put in front of the class the construct `template<typename T>` Replace `T` by the representative name).

The construct `template<typename T>` can be understood as **"for all types `T`"**.

Class template

```
template <typename K>
class Node {
  K key;
  Node* left, right;
public:
  Node(K k, Node* l, Node* r): key(k), left(l), right(r) {}

  bool contains(K search_key) const {
    return search_key == key
      || left != nullptr && left->contains(search_key)
      || right != nullptr && right->contains(search_key)
  }
  ...
};
```

Function Template: Analogous Approach

1. To make a concrete implementation generic, replace the specific type (e.g. int) with a name, e.g. `T`,
2. Put in front of the function the construct `template<typename T>` (Replace `T` by the chosen name)

Examples

- For free functions

```
template <typename T>
void swap(T& x, T& y) {
  T temp = x;
  x = y;
  y = temp;
}
```

```
template <typename Iter>
void is_sorted(Iter begin, Iter end){
  ...
}
```

- For operators

```
template <typename T>
ostream& operator<<(ostream& out, const Node<T> root) {
  ...
}
```

Semantics (Code-Generation)

For each template instance, the compiler creates a corresponding instantiated class (or function) → static code generation

```
Node<int> n1 = ...;
Node<std::string> n2 = ...;
Node<Student> n3 = ...;
```

n1

```
class Node_int {
  int key;
  ...
  bool contains(int k) {...}
  int max() {...}
};
```

n2

```
class Node_string {
  std::string key;
  ...
};
```

n3

```
class Node_Student {
  Student key;
  ...
};
```

Semantics (Code-Generation)

For each template instance, the compiler creates a corresponding instantiated class (or function) $\rightarrow$ static code generation

Question: what does this imply for seperate compilation?

- Should templates go into .h (declarations) or .cpp (definitions) files?
- Is it possible to ship the compiled implementation (binary file compiled from .cpp) alongside the header file?

Generalizing Code using Templates

```
class Vector {
public:
  Vector() {...}
  float& operator [](int i) { return data[i]; }
private:
  float data[3];
};

float scalar_product(Vector a, Vector b) {
    float result = 0;
    for (int i=0; i<3; ++i)
       result += a[i] * b[i];
    return result;
}
```

Generalizing Code using Templates

```
template <typename T>
class Vector {
public:
  Vector() {...}
  T& operator [](int i) { return data[i]; }
private:
  T data[3];
};

template <typename T>
T scalar_product(Vector<T> a, Vector<T> b) {
    T result = 0;
    for (int i=0; i<3; ++i)
       result += a[i] * b[i];
    return result;
}
```

Generalizing Code using Templates

```
template <unsigned N, typename T>
class Vector {
public:
  Vector() {...}
  T& operator [](int i) { return data[i]; }
private:
  T data[N];
};

template <unsigned N, typename T>
T scalar_product(Vector<N, T> a, Vector<N, T> b) {
    T result = 0;
    for (int i=0; i<N; ++i)
       result += a[i] * b[i];
    return result;
}
```

Type testing

- Templates: syntactic checks
- Instances: checks as usual

```
template <typename T>
T abs(T v) {
  return 0 <= v ? v : -v;
}
// main
abs(8); // OK
```

```
template <typename T>
T abs(T v) {
  return 0 <= v ? v : -v; // Error
}
// main
abs("hi"); // Error
```

```
template <typename T>
void swap(T& x, T& y) {
  ...
}
// main
double a = 1.0;
double b = 7;
swap(a, b); // OK
```

```
template <typename T>
void swap(T& x, T& y) {
  ...
}
// main
double a = 1.0;
string b = "seven";
swap(a, b); // Error
```

Other Languages

All languages try to foster code reuse but chose different solutions.

- C++, Rust:
  - static code generation
  - no runtime overhead
  - difficult to integrate into OOP
- C#, Scala (, Java)
  - type parameters are turned into runtime values
  - well-suited for OOP
  - minor runtime overhead
- Python, JavaScript:
  - dynamic typing (duck typing)
  - no syntactic overhead
  - potentially significant runtime overhead

6.1 auto vs templates

auto

- Placeholder type specifier
  - Must be uniquely determined by direct context: initialiser code, or returns
  - User could write type themself, but leave it to the compiler

```
std::vector<int> vec = ...;
auto it = vec.cbegin();
// placeholder for td::vector<int>::const_iterator
```

- Failing examples:

```
auto x; // x has no initializer
x = 0.0;
auto first_or_else(std::vector<int> data, unsigned int or_else) {
  if (data.size() == 0) return or_else;
  else return data[0];
}
```

Templates

- Parameters are unknown until instantiated

```
template <typename N>
char sign(N v) {
  if (0 <= v) return '+';
  else return '-';
}

template <typename T1, typename T2>
struct Pair {
  T1 fst;
  T2 snd;
};
```

- Instantiation may happen anywhere

```
Pair<int, double> p1 = Pair{1, 0.1};
auto p2 = Pair<std::string, bool>{"Brazil", true};
```

Combining templates and auto

`auto` inside template must be determined after instantiation

```
template <typename C>
void print(C container) {
  for (auto& e : container)
  std::cout << e << ' ';
}

std::vector<int> numbers = {1, 2, 3};
print(numbers); // now auto can be determined

std::vector<std::string> airports = {"LAX", "LDN", "ZHR"};
print(airports); // now auto can be determined
```

Combining templates and auto

`auto` inside template must be determined after instantiation

```
template <typename C>
void print(C container) {
  for (auto& e : container)
  std::cout << e << ' ';
}
```

Question: Is it possible to not use `auto` here?

Answer: Yes, for example by replacing auto with an additional template parameter `E`

From auto to templates

- Before C++20 auto function parameters are forbidden

  ```
  void print(auto x) {...} // Compiler error
  ```

  Question: Why do you think that is?
  Answer: Cannot determine type from context
- Since C++20 auto function parameters are allowed

  ```
  void print(auto x) {...} // ok
  ```

  Clearly, it is still not possible to determine what auto stands for.
  Question: What could be the meaning of auto in this case?
  Answer: It is a shorthand for a template parameter!

  ```
  template <typename T>
  void Print(T x){ ... }
  ```

7. Repetition theory: Induction

Induction: what is required?

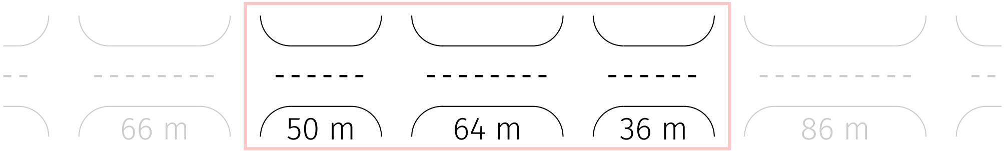
- Prove statements, for example $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Base clause:
  - The given (in)equality holds for one or more base cases.
  - e.g. $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$.
- Induction hypothesis: we assume that the statement holds for some n
- Induction step ($n \to n+1$):
  - From the validity of the statement for n (induction hypothesis) it follows the one for $n+1$.
  - e.g.: $\sum_{i=1}^{n+1} i = n+1+\sum_{i=1}^{n} i = n+1+\frac{n(n+1)}{2} = \frac{(n+2)(n+1)}{2}$.

8. Subarray Sum Problem

Naïve Solution, prefix sums, binary search, Sliding Window

Street section of a given length

Given: distances between all crossroads on a street



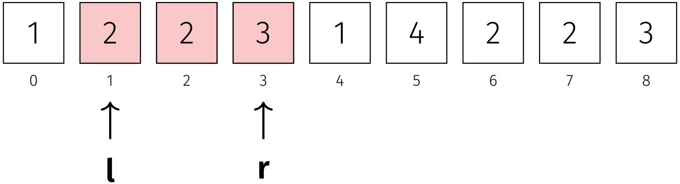
Wanted: street section of length 150 meters between crossroads

Subarray Sum Problem

Given: a sequence $a[0], \ldots, a[n-1]$ of non-negative integers

Wanted: a subsequence with sum k:
pair (l, r) with $0 \leq l \leq r \leq n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Example: $n = 9, k = 7$ **Solution:** $l = 1, r = 3$.



Strategies?

Given: a sequence $a[0], \ldots, a[n-1]$ of non-negative integers
Wanted: a subsequence with sum k:
pair (l, r) with $0 \leq l \leq r \leq n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Strategies

$\Theta(n^3)$	Three loops
$\Theta(n^2)$	Prefix Sums
$\Theta(n \log n)$	Binary Search
$\Theta(n)$	Sliding Window

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
↑	↑	↑	↑	↑	↑				
l,l r	l r	l r	l r	r	r				

window sum $=$ **256868587**

Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window
- if l reaches the start before r reaches the end
  $\Rightarrow$ sum too small $\Rightarrow$ r is increased until it reaches the end of the window

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
	↑	↑	↑	↑	↑				
	l	**l**	**l,r**	**r**	**r**				

Analysis

We consider the lexicographically smallest (left-most) window with sum k, called *target window*

- In each step of the algorithm either l or r is increased. The algorithm terminates after a maximum of $2n$ steps.
- Assume r reaches the end of the target window before l reaches the start of the target window, then l keeps increasing until it reaches the start of the window.
- Assume l reaches the start of the target window before r reaches the end of the target window, then r keeps increasing until it reaches the end of the window.

Exercise: window with sum closest to k

9. Code Example

10. Programming Exercise

Preparing remarks for the homework (Prefix Sum in 2D)

Sum in Subarray (naive algorithm)

Input: A sequence of n numbers $(a_0, a_1, \dots, a_{n-1})$ and a sub-interval $I = [x_0, x_1]$
Output: $\sum_{i=x_0}^{x_1} a_i$.

$\mathcal{S} \leftarrow 0$
for $i \in \{x_0, \dots, x_1\}$ **do**
  $\mathcal{S} \leftarrow \mathcal{S} + a_i$
return $\mathcal{S}$

Idea of the exercise

- Use the prefix sum to compute the sum of arbitrary sub-intervals with constant running time
- **Generalize** to two dimensions.

11. Tips for **code** expert

Tips for **code** expert Exercise 2

Task "Prefix Sum in 2D"

- Study the Prefix Sum in 1D[3] well and go from there
- Make sketches!

Task "Sliding Window"

- Sketches!

Task "Proofs by Induction"

- The binomial formula will be useful for the second one
- Please format it well or just scan a PDF and upload it

Task "Karatsuba Ofman"

- Just translate the math into code

[3] There's an inplementation in the code examples on **code** expert

12. Outro

General Questions?

See you next time

Have a nice week!