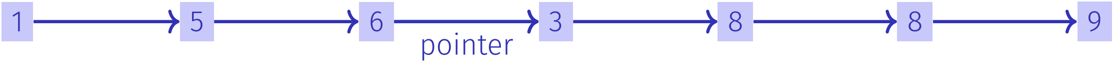
21. Dynamic Data Structures II

Linked Lists, Vectors as Linked Lists

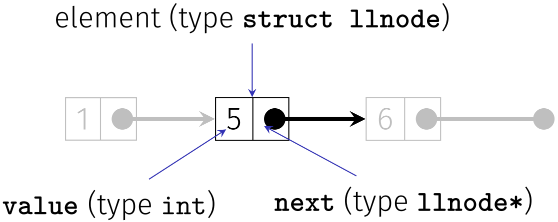
Different Memory Layout: Linked List

- **No** contiguous area of memory and **no** random access
- Each element points to its successor
- Insertion and deletion of **arbitrary** elements is simple



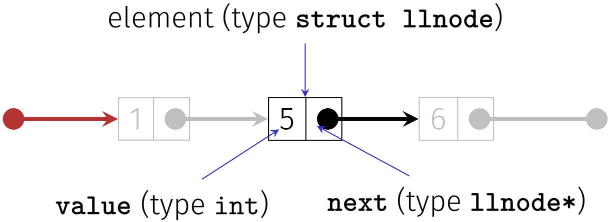
⇒ Our vector can be implemented as a linked list

Linked List: Zoom




```
struct llnode {
  int value;
  llnode* next;

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

Vector = Pointer to the First Element




```
class llvec {
  llnode* head;
public: // Public interface identical to avec's
  llvec(unsigned int size);
  unsigned int size() const;
  ...
};
```

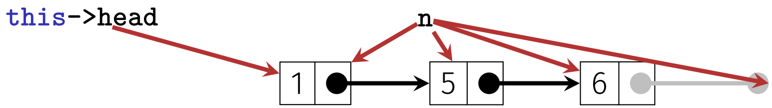
Function `llvec::print()`

```
struct llnode {
  int value;
  llnode* next;
  ...
};
```

```
void llvec::print(std::ostream& sink) const {
  for (llnode* n = this->head;      // Pointer to first element
       n != nullptr;                // Abort if end reached
       n = n->next)                 // Advance pointer element-wise
  {
    sink << n->value << ' ';        // Output current element
  }
}
```

Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
  for (llnode* n = this->head;
       n != nullptr;
       n = n->next)
  {
    sink << n->value << ' ';  // 1 5 6
  }
}
```



Function `llvec::operator[]`

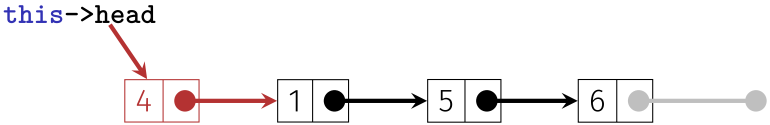
Accessing ith Element is implemented similarly to **`print()`**:

```
int& llvec::operator[](unsigned int i) {
  llnode* n = this->head;      // Pointer to first element

  for (; 0 < i; --i)           // Step to ith element
    n = n->next;

  return n->value;             // Return ith element
}
```

Function `llvec::push_front()`

Advantage **`llvec`**: Prepending elements is very easy:

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



Attention: If the new **`llnode`** weren't allocated *dynamically*, then it would be deleted (= memory deallocated) as soon as **`push_front`** terminates

Function `llvec::llvec()`

Constructor can be implemented using **`push_front()`**:

```
llvec::llvec(unsigned int size) {
  this->head = nullptr;

  for (; 0 < size; --size)
    this->push_front(0);
}
```

`head` initially points to nowhere

Prepend `0` `size` times

Use case:

```
llvec v = llvec(3);
std::cout << v; // 0 0 0
```

Function `llvec::push_back()`

Simple, but inefficient: traverse linked list to its end and append new element

```
void llvec::push_back(int e) {
  llnode* n = this->head;

  for (; n->next != nullptr; n = n->next);

  n->next =
    new llnode{e, nullptr};
}
```

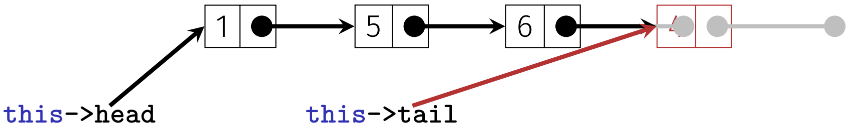
Start at first

… and go to the last element

Append new element to currently last

Function `llvec::push_back()`

- More efficient, but also slightly more complex:
    1. Second pointer, pointing to the last element: **`this->tail`**
    2. Using this pointer, it is possible to append to the end directly



- But: Several corner cases, e.g. vector still empty, must be accounted for

Function `llvec::size()`

Simple, but inefficient: *compute* size by counting

```
unsigned int llvec::size() const {
  unsigned int c = 0;

  for (llnode* n = this->head;
       n != nullptr;
       n = n->next)
    ++c;

  return c;
}
```

Count initially `0`

Count linked-list length

Return count

Function `llvec::size()`

More efficient, but also slightly more complex: *maintain* size as member variable

1. Add member variable **`unsigned int count`** to class **`llvec`**
2. **`this->count`** must now be updated *each* time an operation (such as **`push_front`**) affects the vector's size

Efficiency: Arrays vs. Linked Lists

- Memory: our **avec** requires roughly n ints (vector size n), our **llvec** roughly $3n$ ints (a pointer typically requires 8 byte)
- Runtime (with **avec** = **std::vector**, **llvec** = **std::list**):

22. Containers, Iterators and Algorithms

Containers, Sets, Iterators, const-Iterators, Algorithms, Templates

Vectors are Containers

- Viewed abstractly, a vector is
  1. A collection of elements
  2. Plus operations on this collection
- In C++, **`vector<`**T**`>`** and similar data structures are called *container*
- Called *collections* in some other languages, e.g. Java

Container properties

- Each container has certain *characteristic properties*
- For an array-based vector, these include:
  - Efficient index-based access (`v[i]`)
  - Efficient use of memory: Only the elements themselves require space (plus element count)
  - Inserting at/removing from arbitrary index is potentially inefficient
  - Looking for a specific element is potentially inefficient
  - Can contain the same element more than once
  - Elements are in insertion order (ordered but not sorted)

Containers in C++

- Nearly every application requires maintaining and manipulating arbitrarily many data records
- But with different requirements (e.g. only append elements, hardly ever remove, often search elements, ...)
- That's why C++'s standard library includes several containers with different properties, see `https://en.cppreference.com/w/cpp/container`
- Many more are available from 3rd-party libraries, e.g. `https://www.boost.org/doc/libs/1_68_0/doc/html/container.html`, `https://github.com/abseil/abseil-cpp`

Example Container: `std::unordered_set`$<T>$

- A *mathematical set* is an unordered, duplicate-free collection of elements:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set`$<T>$
- Properties:
  - Cannot contain the same element twice
  - Elements are not in any particular order
  - Does not provide index-based access (`s[i]` undefined)
  - Efficient "element contained?" check
  - Efficient insertion and removal of elements
- Side remark: implemented as a hash table

Use Case `std::unordered_set<T>`

Problem:

- given a sequence of pairs (*name, percentage*) of Code Expert submissions . . .

```
// Input: file submissions.txt
Friedrich 90
Schwerhoff 10
Lehner 20
Schwerhoff 11
```

- . . . determine the submitters that achieved at least 50%

```
// Output
Friedrich
```

Use Case `std::unordered_set<T>`

```
std::ifstream in("submissions.txt");
std::unordered_set<std::string> names;

std::string name;
unsigned int score;

while (in >> name >> score) {
  if (50 <= score)
    names.insert(name);
}

std::cout << "Unique submitters: "
          << names << '\n';
```

- Open **submissions.txt**
- Set of names, initially empty
- Pair *(name, score)*
- Input next pair
- Record name if score suffices
- Output recorded names

Example Container: **std::set**$<T>$

- Nearly equivalent to **std::unordered_set**$<T>$, but the elements are *ordered*

  $$\{1,2,1\} = \{1,2\} \neq \{2,1\}$$

- Element look-up, insertion and removal are still efficient (better than for **std::vector**$<T>$), but less efficient than for **std::unordered_set**$<T>$
- That's because maintaining the order does not come for free
- Side remark: implemented as a red-black tree

Use Case `std::set<T>`

```
std::ifstream in("submissions.txt");
std::set<std::string> names;

std::string name;
unsigned int score;

while (in >> name >> score) {
  if (50 <= score)
    names.insert(name);
}

std::cout << "Unique submitters: "
          << names << '\n';
```

`set` instead of `unsorted_set` . . .

. . . and the output is in alphabetical order

Printing Containers

- Recall: **`avec::print()`** and **`llvec::print()`**
- What about printing **`set`**, **`unordered_set`**, ...?
- Commonality: iterate over container elements and print them

Similar Functions

- Lots of other useful operations can be implemented by iterating over a container:
- **`contains(c, e)`**: true iff container **`c`** contains element **`e`**
- **`min/max(c)`**: Returns the smallest/largest element
- **`sort(c)`**: Sorts **`c`**'s elements
- **`replace(c, e1, e2)`**: Replaces each **`e1`** in **`c`** with **`e2`**
- **`sample(c, n)`**: Randomly chooses **`n`** elements from **`c`**
- ...

Recall: Iterating With Pointers

- Iteration over an *array*:
  - Point to start element: `p = this->arr`
  - Access current element: `*p`
  - Check if end reached: `p == this->arr + size`
  - Advance pointer: `p = p + 1`

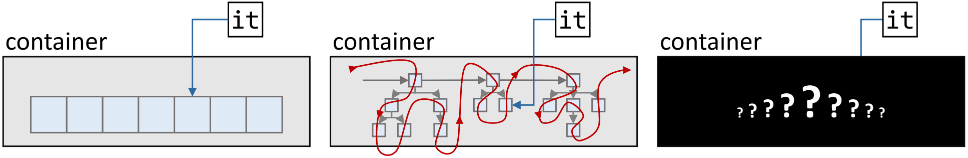
- Iteration over a *linked list*:
  - Point to start element: `p = this->head`
  - Access current element: `p->value`
  - Check if end reached: `p == nullptr`
  - Advance pointer: `p = p->next`

Iterators

- Iteration requires only the previously shown four operations
- But their implementation depends on the container
- $\Rightarrow$ Each C++container implements their own *Iterator*
- Given a container **`c`**:
  - **`it = c.begin()`**: Iterator pointing to the first element
  - **`it = c.end()`**: Iterator pointing *behind* the last element
  - **`*it`**: Access current element
  - **`++it`**: Advance iterator by one element
- Iterators are essentially pimped pointers

Iterators

- Iterators allow accessing different containers in a *uniform* way: **`*it`**, **`++it`**, etc.
- Users remain independent of the container implementation
- Iterator knows how to iterate over the elements of “its” container
- Users don’t need to and also shouldn’t know internal details
- ⇒



Example: Iterate over `std::vector`

```
std::vector<int> v = {1, 2, 3};

for (std::vector<int>::iterator it = v.begin();
     it != v.end();
     ++it) {

  *it = -*it;
}

std::cout << v; // -1 -2 -3
```

`it` is an iterator specific to `std::vector<int>`

`it` initially points to the first element

Abort if `it` reached the end

Advance `it` element-wise

Negate current element ($e \rightarrow -e$)

Example: Iterate over `std::vector`

Recall: type aliases can be used to shorten often-used type names

```
using ivit = std::vector<int>::iterator; // int-vector iterator

for (ivit it = v.begin();
     ...
```

Negate as a Function

As before: passing a *range (interval)* to work on

```
void neg(std::vector<int>::iterator begin;
         std::vector<int>::iterator end) {

  for (std::vector<int>::iterator it = begin;
       it != end;
       ++it) {

    *it = -*it;
  }
}
```

Negate elements in interval [**begin**, **end**)

Negate as a Function

As before: passing a *range (interval)* to work on

```
void neg(std::vector<int>::iterator begin;
         std::vector<int>::iterator end);
```

```
// in main():
std::vector<int> v = {1, 2, 3};
neg(v.begin(), v.begin() + (v.size() / 2));
```

Negate first half

Algorithms Library in C++

- The C++standard library includes lots of useful algorithms (functions) that work on iterator-defined intervals [*begin*, *end*)
- For example **`find`**, **`fill`** and **`sort`**; see also `https://en.cppreference.com/w/cpp/algorithm`
- Thanks to iterators, these $\geq$ 100 (!) algorithms can be applied to any* container: the 17 (!) C++standard container, our **`avec`** and **`llvec`** (discussed next), etc.
- Without this uniform access to container elements, we would have to duplicate *lots* of code

Not every algorithm can be applied to every container. It is, e.g. Not possible to sort a **`std::set`**.

An iterator for **llvec**

We need:

1. An **llvec**-specific iterator with at least the following functionality:
   - Access current element: **operator***
   - Advance iterator: **operator++**
   - End-reached check: **operator!=** (or **operator==**)
2. Member functions **begin()** and **end()** for **llvec** to get an iterator to the beginning and past the end, respectively

Iterator `llvec::iterator` (Step 1/2)

```
class llvec {
  ...
public:
  class iterator {
    ...
  };


  ...
}
```

- The iterator belongs to our vector, that's why **`iterator`** is a public *inner class* of **`llvec`**
- Instances of our iterator are of type **`llvec::iterator`**

Iterator `llvec::iterator` (Step 1/2)

```
class iterator {
  llnode* node;

public:
  iterator(llnode* n);
  iterator& operator++();
  int& operator*() const;
  bool operator!=(const iterator& other) const;
};
```

- `llnode* node;` — Pointer to current vector element
- `iterator(llnode* n);` — Create iterator to specific element
- `iterator& operator++();` — Advance iterator by one element
- `int& operator*() const;` — Access current element
- `bool operator!=(const iterator& other) const;` — Compare with other iterator

Iterator `llvec::iterator` (Step 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): node(n) {}
```

Let iterator point to **n** initially

```
// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
  assert(this->node != nullptr);

  this->node = this->node->next;

  return *this;
}
```

Advance iterator by one element

Return reference to advanced iterator

Iterator `llvec::iterator` (Step 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
  return this->node->value;
}


// Comparison: when are two iterators not equal?
bool llvec::iterator::operator!=(
    const llvec::iterator& other) const
{
  return this->node != other.node;
}
```

Access current element

`this` iterator different from `other` if they point to different element

An iterator for **`llvec`** (Repetition)

We need:

1. An **`llvec`**-specific iterator with at least the following functionality:
   - Access current element: **`operator*`**
   - Advance iterator: **`operator++`**
   - End-reached check: **`operator!=`** (or **`operator==`**)

2. Member functions **`begin()`** and **`end()`** for **`llvec`** to get an iterator to the beginning and past the end, respectively

Iterator `llvec::iterator` (Step 2/2)

```
class llvec {
  ...
public:
  class iterator {...};

  iterator begin();
  iterator end();

  ...
}
```

`llvec` needs member functions to issue iterators pointing *to the beginning* and *past the end*, respectively, of the vector

Iterator `llvec::iterator` (Step 2/2)

```
llvec::iterator llvec::begin() {
  return llvec::iterator(this->head);
}

llvec::iterator llvec::end() {
  return llvec::iterator(nullptr);
}
```

Iterator to first vector element

Iterator past last vector element

Const-Iterators

- In addition to **`iterator`**, every container should also provide a *const-iterator* **`const_iterator`**
- Const-iterators grant only read access to the underlying Container
- For example for **`llvec`**:

```
llvec::const_iterator llvec::cbegin() const;
llvec::const_iterator llvec::cend() const;

const int& llvec::const_iterator::operator*() const;
...
```

- Therefore not possible (compiler error): **`*(v.cbegin()) = 0`**

Const-Iterators

Const-Iterator *can* be used to allow only reading:

```
llvec v = ...;
for (llvec::const_iterator it = v.cbegin(); ...)
  std::cout << *it;
```

It would also possible to use the non-const **`iterator`** here

Const-Iterators

Const-Iterator *must* be used if the vector is const:

```
const llvec v = ...;
for (llvec::const_iterator it = v.cbegin(); ...)
  std::cout << *it;
```

It is not possible to use **`iterator`** here (compiler error)

Range-based `for` Loop

- Sequential iteration over an **`llvec`**, using an iterator (const-iterator possible, as are other containers):

```
llvec v(3); // v == {0, 0, 0}
for (llvec::iterator it = v.begin(); it != v.end(); ++it)
  std::cout << *it; // 000
```

- Can alternatively be written as follows:

```
for (int i : v) std::cout << i; // 000
```

  Is then translated to an iterator-based loop.

- Mutating access is possible as well:

```
for (int& i : v) i += 3;
for (int i : v) std::cout << i; // 369
```

Type-generic Container

Type-specific containers

Type-generic container

`https://upload.wikimedia.org/wikipedia/commons/d/df/Container_01_KMJ.jpg` (CC BY-SA 3.0)

P.S.: Templates are not relevant for the exam

(Type-generic containers (templates in general) aren't relevant for the exam)

Type-generic Container

Class **cell**: a simple, single-element container for **int**

```
class cell {
  int element;

public:
  cell(int e);
  int& value();
};
```

Constructor stores **e** in container

container's element

```
cell::cell(int e)
  : element(e) {}

int& cell::value() {
  return this->element;
}
```

Access the element

Better: generic **cell<E>** for every element type **E** (analogous to **std::vector<E>**)

Type-generic Container with Templates

Templates enable *type-generic* functions and classes:

```
template<typename E>
class cell {
  E element;

public:
  cell(E e);
  E& value();
};
```

Let **E** an arbitrary type ...

...then **cell** manages an element of type **E**

- Types can be used as *parameters*
- Type parameters are valid in the “templated” scope

Type-generic Container with Templates

- Signatures and implementations must be "templated"
- For separately provided implementations, the class prefix must be written in generic form

```
template<typename E>
class cell {
  E element;

public:
  cell(E e);
  E& value();
};
```

```
template<typename E>
cell<E>::cell(E e)
    : element(e) {}

template<typename E>
E& cell<E>::value() {
  return this->element;
}
```

Type-generic Container with Templates

```
cell<int> c1(313);
cell<std::string> c2("terrific!")
```

- For *declarations*, e.g. **`cell<int>`**, type parameters must be provided explicitly . . .
- . . . but they are *inferred* by the compiler everywhere else, e.g. for **`c1(313)`**, i.e. when invoking the generic constructor **`cell(E e)`** (where type parameter **`E`** is instantiated by the compiler with **`int`**)

More Templates: Generic Output Operator

- Goal: A *generic* output operator **`<<`** for *iterable Containers*: **`llvec`**, **`avec`**, **`std::vector`**, **`std::set`**, ...
- I.e. **`std::cout << c << '\n'`** should work for any such container **`c`**

More Templates: Generic Output Operator

- Generic output operator with two type parameters

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

Intuition: operator works for every output stream **sink** of type **S** and every container **container** of type **C**

More Templates: Generic Output Operator

- Generic output operator with two type parameters

```cpp
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

- The compiler infers suitable types from the call arguments

```cpp
std::set<int> s = ...;
std::cout << s << '\n';
```

← S = std::ostream, C = std::set<int>

More Templates: Generic Output Operator

Implementation of `<<` *constrains* `S` and `C` (Compiler errors if not satisfied):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
  for (typename C::const_iterator it = container.begin();
       it != container.end();
       ++it) {

    sink << *it << ' ';
  }

  return sink;
}
```

`C` must appropriate iterators
– with appropriate functions

More Templates: Generic Output Operator

Implementation of `<<` *constrains* `S` and `C` (Compiler errors if not satisfied):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
  for (typename C::const_iterator it = container.begin();
        it != container.end();
        ++it) {

    sink << *it << ' ';
  }

  return sink;
}
```

`S` must support outputting elements (`*it`) and characters (`' '`)

Templates: Conclusion

- Templates realise *static code generation/static metaprogramming* in C++
- Template code is *copied* per type instantiation. When using **cell<int>** and **cell<std::string>**, the compiler creates two *instantiated copies* of **cell**'s code: conceptually, the two (no longer generic) classes **cell_int** and **cell_stdstring**.
- Templates reduce code duplication and facilitate code reuse
- Compiler errors that refer to templates are unfortunately often even more complex than C++ errors usually already are