Recap: `vector<T>`

- Can be initialised with arbitrary size $n$.
- Supports various operations:
  - `e = v[i];` // Get element
  - `v[i] = e;` // Set element
  - `l = v.size();` // Get size
  - `v.push_front(e);` // Prepend element
  - `v.push_back(e);` // Append element
  ...

- A vector is a *dynamic data structure*, whose size may change at runtime.

Our Own Vector!

- Today, we’ll implement our own vector: `vec`
- Step 1: `vec<int>` (today)
- Step 2: `vec<T>` (later, only superficially)

Vectors in Memory

Already known: A vector has a *contiguous* memory layout

Question: How to *allocate* a chunk of memory of *arbitrary* size during runtime, i.e. *dynamically*?
**new for Arrays**

- **Effect**: new contiguous chunk of memory $n$ elements of type $T$ is allocated
- This chunk of memory is called an *array* (of length $n$)

**new**

- **Type**: A pointer $T*$ (more soon)
- **Value**: the starting address of the memory chunk

**new (Without Arrays)**

- **Effect**: memory for a new object of type $T$ is allocated . . .
- . . . and initialized by means of the matching constructor
- **Value**: address of the new $T$ object, **Type**: Pointer $T*$
- Also true here: object “lives” until deleted explicitly (usefulness will become clearer later)
**Pointer Types**

\[ T^* \]  
Pointer type for base type \( T \)

An expression of type \( T^* \) is called *pointer (to \( T \))*

```
int* p; // Pointer to an int
std::string* q; // Pointer to a std::string
```

**T*  
Pointer type for base type \( T \)

A \( T^* \) must actually point to a \( T \)

```
int* p = ...;
std::string* q = p; // compiler error!
```

**Value** of a pointer to \( T \) is the *address* of an object of type \( T \)

```
int* p = ...;
std::cout << p; // e.g. 0x7ffd89d5f7cc
```

**Address Operator**

*Question:* How to obtain an object’s address?

- Directly, when creating a new object via new
- For existing objects: via the *address operator* `&`

- **Value** of the expression: the *address* of object (l-value) `expr`
- **Type** of the expression: A pointer \( T^* \) (of type \( T \) )
**Address Operator**

```c
int i = 5; // i initialised with 5
int* p = &i; // p initialised with address of i
```

Next question: How to “follow” a pointer?

**Dereference Operator**

```c
int i = 5; int* p = &i; // p = address of i int j = *p; // j = 5
```

**Address and Dereference Operator**

Answer: by using the *dereference operator* 

- **Value** of the expression: the value of the object located at the address denoted by `expr`
- **Type** of the expression: `T`
**Mnemonic Trick**

The declaration

```c
T* p; // p is of the type “pointer to T”
```

can be read as

```c
T *p; // *p is of type T
```

Although this is legal, we do not write it like this!

---

**Null-Pointer**

- Special pointer value that signals that no object is pointed to
- Represented by the literal `nullptr` (convertible to `T*`)

```c
int* p = nullptr;
```

- Cannot be dereferenced (runtime error)
- Exists to avoid undefined behaviour

```c
int* p; // p could point to anything
int* q = nullptr; // q explicitly points nowhere
```

---

**Pointer Arithmetic: Pointer plus `int`**

```c
T* p = new T[n]; // p points to first array element
```

How to point to rear elements? → **Pointer arithmetic**:

- `p` yields the value of the first array element, `*p` its value
- `*(p + i)` yields the value of the `i`th array element, for `0 ≤ i < n`
- `*p` is equivalent to `*(p + 0)`

---

**Null-Pointer**

```c
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to 1st element
int* p3 = p0 + 3; // p3 points to 4th element
*(p3 + 2) = 600; // set value of 6th element to 600
std::cout << *(p0 + 5); // output 6th element's value (i.e. 600)
```
Pointer Arithmetic: Pointer minus \textit{int}

- If \( \text{ptr} \) is a pointer to the element with index \( k \) in an array \( a \) with length \( n \)
- and the value of \( \text{expr} \) is an integer \( i \), \( 0 \leq k - i \leq n \),

then the expression

\[
\text{ptr} - \text{expr}
\]

provides a pointer to an element of \( a \) with index \( k - i \).

\[
\text{a} \]
\[
\downarrow
\]
\[
\text{ptr-expr}
\]
\[
\downarrow
\]
\[
\text{ptr}
\]

\[\text{a}[n]\]

\[\text{ptr}[k-i]\]

\[\text{ptr-expr}\]

\[605\]

Pointer Subtraction

- If \( p1 \) and \( p2 \) point to elements of the same array \( a \) with length \( n \)
- and \( 0 \leq k_1, k_2 \leq n \) are the indices corresponding to \( p1 \) and \( p2 \),

then

\[
p1 - p2 \text{ has value } k_1 - k_2
\]

Only valid if \( p1 \) and \( p2 \) point into the same array.

- The pointer difference describes “how far away the elements are from each other”

\[605\]

\[605\]

\[606\]

Pointer Operators

<table>
<thead>
<tr>
<th>Description</th>
<th>Op</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscript</td>
<td>[]</td>
<td>2</td>
<td>17</td>
<td>left</td>
<td>R-value → L-value</td>
</tr>
<tr>
<td>Dereference</td>
<td>*</td>
<td>1</td>
<td>16</td>
<td>right</td>
<td>R-Wert → L-Wert</td>
</tr>
<tr>
<td>Address</td>
<td>&amp;</td>
<td>1</td>
<td>16</td>
<td>rechts</td>
<td>L-value → R-value</td>
</tr>
</tbody>
</table>

Precedences and associativities of \( +, - \), ++ (etc.) as in Chapter 2

\[\text{int* ptr = 5; // error: invalid conversion from int to int*}\]
\[\text{int a = ptr; // error: invalid conversion from int* to int}\]

Pointers are not Integers!

- Addresses can be interpreted as house numbers of the memory, that is, integers
- But integer and pointer arithmetic behave differently.

\[\text{ptr + 1 is not the next house number but the } s-\text{next, where } s \text{ is the memory requirement of an object of the type behind the pointer } \text{ptr.}\]

- Integers and pointers are not compatible
Sequential Pointer Iteration

```cpp
char* p = new char[3]{'x', 'y', 'z'};

for (char* it = p; it != p + 3; ++it)
{
    std::cout << *it << ' '; // x y z
}
```

Random Access to Arrays

```cpp
char* p = new char[3]{'x', 'y', 'z'};

for (int i = 0; i < 3; ++i)
    std::cout << p[i] << ' '; // x y z
```

But: this is less efficient than the previously shown sequential access via pointer iteration.
**Reading a book ... with random access ... with sequential access**

**Random Access**
- open book on page 1
- close book
- open book on pages 2-3
- close book
- open book on pages 4-5
- close book
- ...

**Sequential Access**
- open book on page 1
- turn the page
- turn the page
- turn the page
- turn the page
- ...

---

**Static Arrays**

- int* p = new int[expr] creates a dynamic array of size expr
- C++ has inherited static arrays from its predecessor language C:
  int a[expr]
- Static arrays have, among others, the disadvantage that their size expr must be a constant. I.e. expr can, e.g. be 5 or 4*3+2, but kein von der Tastatur eingelesener Wert n.
- A static array variable a can be used just like a pointer
- Rule of thumb: Vectors are better than dynamic arrays, which are better than static arrays

---

**Arrays in Functions**

**C++ convention**: arrays (or a segment of it) are passed using two pointers

- begin: Pointer to the first element
- end: Pointer past the last element
- [begin, end) Designates the elements of the segment of the array
- [begin, end) is empty if begin == end
- [begin, end) must be a valid range, i.e. a (pot. empty) array segment

---

**Arrays in (mutating) Functions**: fill

```c++
// PRE: [begin, end) is a valid range
// POST: Every element within [begin, end) was set to value
void fill(int* begin, int* end, int value) {
    for (int* p = begin; p != end; ++p)
        *p = value;
}
```

```c++
... int* p = new int[5];
fill(p, p+5, 1); // Array at p becomes {1, 1, 1, 1, 1}
```
Functions with/without Effect

- Pointers can (like references) be used for functions with effect. Example: `fill`
- But many functions don’t have an effect, they only read the data
- ⇒ Use of `const`
- So far, for example:

```c
const int zero = 0;
const int& nil = zero;
```

Positioning of Const

Where does the `const`-modifier belong to?

- `const T` is equivalent to `T const` (and can be written like this):
  ```c
  const int zero = ... ⇐⇒ int const zero = ...
  const int& nil = ... ⇐⇒ int const& nil = ...
  ```

Both keyword orders are used in praxis

Const and Pointers

Read the declaration from right to left

- `int const p;` p is a constant integer
- `int const* p;` p is a pointer to a constant integer
- `int* const p;` p is a constant pointer to an integer
- `int const* const p;` p is a constant pointer to a constant integer

Non-mutating Functions: print

There are also non-mutating functions that access elements of an array only in a read-only fashion

```c
// PRE: [begin, end) is a valid range
// POST: The values in [begin, end) were printed
void print(int const* const begin, const int* const end)
{    for (int const* p = begin; p != end; ++p)
std::cout << *p << ' ';
}
```

Pointer `p` may itself not be `const` since it is mutated (`++p`)
const is not absolute

- The value at an address can change even if a const-pointer stores this address.

beispiel

```c
int a[5];
const int* begin1 = a;
int* begin2 = a;
*begin1 = 1; // error *begin1 is const
*begin2 = 1; // ok, although *begin will be modified
```

- const is a promise from the point of view of the const-pointer, not an absolute guarantee

Wow – Palindromes!

```c
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(--end)) return false;
    return true;
}
```

Arrays, `new`, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a $T$-array of size $n$
- $T* \ p = new T[n]$: pointer $p$ points to the first array element
- Pointer arithmetic enables accessing rear array elements
- Sequentially iterating over arrays via pointers is more efficient than random access
- `new T` allocates memory for (and initialises) a single $T$-object, and yields a pointer to it
- Pointers can point to something (not) const, and they can be (not) const themselves
- Memory allocated by `new` is not automatically released (more on this soon)
- Pointers and references are related, both “link” to objects in memory. See also additional the slides pointers.pdf

Array-based Vector

- Vectors … that somehow rings a bell 😐
- Now we know how to allocate memory chunks of arbitrary size …
- … we can implement a vector, based on such a chunk of memory
- avec – an array-based vector of int elements

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: vec
- Schritt 1: vec<int> (heute)
- Schritt 2: vec<T> (später, nur kurz angesehen)
Array-based Vector `avec`: Class Signature

class avec {
  // Private (internal) state:
  int* elements; // Pointer to first element
  unsigned int count; // Number of elements

public: // Public interface:
  avec(unsigned int size); // Constructor
  unsigned int size() const; // Size of vector
  int& operator[](int i); // Access an element
  void print(std::ostream& sink) const; // Output elems.
}

Constructor `avec::avec()`

```cpp
avec::avec(unsigned int size) : count(size) {
  elements = new int[size];
}
```

Side remark: vector is not initialised with a default value

Excursion: Accessing Member Variables

```cpp
avec::avec(unsigned int size): count(size) {
  this->elements = new int[size];
}
```

- `elements` is a member variable of our `avec` instance
- That instance can be accessed via the `pointer` `this`
- `elements` is a shorthand for `(*this).elements`
- Dereferencing a pointer `(*this)` followed by a member access `.elements` is such a common operation that it can be written more concisely as `this->elements`
- Mnemonic trick: “Follow the pointer to the member variable”

Function `avec::size()`

```cpp
int avec::size() const {
  return this->count;
}
```

Usage example:

```cpp
avec v = avec(7);
assert(v.size() == 7); // ok
```
**Function avec::operator[]**

```cpp
int& avec::operator[](int i) { return this->elements[i]; }
```

Element access with index check:

```cpp
int& avec::at(int i) const {
    assert(0 <= i && i < this->count);
    return this->elements[i];
}
```

**Usage example:**

```cpp
avec v = avec(7);
std::cout << v[6]; // Outputs a "random" value
v[6] = 0;
std::cout << v[6]; // Outputs 0
```

---

**Function avec::operator[]** is needed twice

```cpp
int& avec::operator[](int i) { return elements[i]; }
const int& avec::operator[](int i) const { return elements[i]; }
```

- The first member function is not const and returns a non-const reference
  ```cpp
  avec v = ...; // A non-const vector
  std::cout << v.get[0]; // Reading elements is allowed
  v.get[0] = 123; // Modifying elements is allowed
  ```

- It is called on non-const vectors

- The second member function is const and returns a const reference
  ```cpp
  const avec v = ...; // A const vector
  std::cout << v.get[0]; // Reading elements is allowed
  v.get[0] = 123; // Compiler error: modifications are not allowed
  ```

- It is called on const vectors

Also see the example attached to this PDF
Function `avec::print()`

Output elements using sequential access:

```cpp
void avec::print(std::ostream& sink) const {
    for (int* p = this->elements; p != this->elements + this->count; ++p) {
        sink << *p << ' ';
    }
}
```

Pointer to first element
Advance pointer element-wise
Abort iteration if past last element
Output current element

Finally: overload output operator:

```cpp
std::ostream& operator<<(std::ostream& sink, const avec& vec) {
    vec.print(sink);
    return sink;
}
```

Observations:
- Constant reference to `vec`, since unchanged
- But not to `sink`: Outputing elements equals change `sink` is returned to enable output chaining, e.g. `std::cout << v << '
`

Further Functions?

```cpp
class avec {
    ...
    void push_front(int e) // Prepend e to vector
    void push_back(int e) // Append e to vector
    void remove(unsigned int i) // Cut out ith element
    ...
}
```

Commonalities: such operations need to change the vector’s `size`

Resizing arrays

An allocated block of memory (e.g. `new int[3]`) cannot be resized later on

```
|   |   | 2 | 1 | 7 |
```

Possibility:
- Allocate more memory than initially necessary
- Fill from inside out, with pointers to first and last element
- But eventually, all slots will be in use
- Then unavoidable: Allocate larger memory block and copy data over

Deleting elements requires shifting (by copying) all preceding or following elements

Similar: inserting at arbitrary position