Recall: Pointers running over the Array

Beispiel

```cpp
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
    std::cout << *p << ' ';  // 3 4 6 1 2
```

- An array can be converted into a pointer to its first element.
- Pointers “know” arithmetics and comparisons.
- Pointers can be dereferenced.
- Pointers can be used to operate on arrays.

Arrays: Indices vs. Pointer

```cpp
int a[n];
// Task: set all elements to 0
// Solution with indices is more readable
for (int i = 0; i < n; ++i)
    a[i] = 0;
```

- Solution with pointers is faster and more generic
- Pointer to the first element
- Pointer past the end
- for (int* p = begin; p != end; ++p)
  *p = 0;

Arrays and Indices

```cpp
// Set all elements to value
for (int i = 0; i < n; ++i)
    a[i] = value;
```

Computational costs

\[
\text{Adresse von } a[0] = a + 0 \cdot s \\
\text{adresse of } a[n-1] = a + (n - 1) \cdot s
\]

⇒ One addition and one multiplication per element
**The Truth about Random Access**

The expression

\[ a[i] \]

is equivalent to

\[ *(a + i) \]

\[ a + i \cdot s \]

---

**Arrays and Pointers**

// set all elements to value
for (int* p = begin; p != end; ++p)
  *p = value;

Computational cost

\[ \begin{array}{c}
\uparrow \\
\text{begin} \\
\text{end}
\end{array} \]

⇒ one *addition* per element

---

**Reading a book ... with indices ... with pointers**

**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
- turn the page
- ....

---

**Array Arguments: Call by (const) reference**

```cpp
void print_vector (const int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    std::cout << v[i] << " ";
  }
}

void make_null_vector (int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
```

---
Array Arguments: *Call by value (not really ...)*

```c
void make_null_vector (int v[3]) {
    for (int i = 0; i<3 ; ++i) {
        v[i] = 0;
    }
}
...
int a[10];
make_null_vector (a); // only sets a[0], a[1], a[2]

int* b;
make_null_vector (b); // no array at b, crash!
```

Array Arguments: *Call by value does not exist*

- Formal argument types $T[n]$ or $T[]$ (array over $T$) are equivalent to $T*$ (pointer to $T$)
- For passing an array the pointer to its first element is passed
- Length information is lost
- Function cannot work on a part of an array (example: search for an element in the second half of an array)

Arrays in Functions

- Convention of the standard library: pass an array (or a part of it) using two pointers
  - `begin`: pointer to the first element
  - `end`: pointer *behind* the last element
  - `[begin, end)` designates the elements of the part of the array
  - `valid` range means: there are array elements “available” here.
  - `[begin, end)` is empty if `begin == end`

```c
// PRE: [begin, end) is a valid range
// POST: every element within [begin, end) will be set to value
void fill (int* begin, int* end, int value) {
    for (int* p = begin; p != end; ++p)
        *p = value;
}
...
int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
    std::cout << a[i] << " "; // 1 1 1 1 1
```

Arrays in Functions: fill

- `fill` expects pointers to the first element of a range
- `fill` passes the address (of the first element) of `a`
Pointers are not Integers!

- Addresses can be interpreted as house numbers of the memory, that is, integers.
- But integer and pointer arithmetics behave differently. \( \text{ptr + 1} \) is not the next house number but the \( s \)-next, where \( s \) is the memory requirement of an object of the type behind the pointer \( \text{ptr} \).
- Integers and pointers are not compatible

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

Null-Pointer

- special pointer value that signals that no object is pointed to represented by the integer number 0 (convertible to \( T* \))
- cannot be dereferenced (checked during runtime)
- to avoid undefined behavior

\[ \text{int* iptr = 0;} \]
\[ \text{int j = *iptr; // illegal address in *} \]

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”

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.) like in chapter 2
Mutating Functions

- Pointers can (like references) be used for functions with effect

Beispiel

```c
int a[5];
fill(a, a+5, 1); // modifies a
```

Such functions are called *mutating*

Const Correctness

- 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 and nonempty range
// POST: the smallest value in [begin, end) is returned
int min (const int* begin, const int* end)
{
    assert (begin != end);
    int m = *begin; // current minimum candidate
    for (const int* p = ++begin; p != end; ++p)
        if (*p < m) m = *p;
    return m;
}
```

Mark with `const`: value of objects cannot be modified through such `const`-pointers.

**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 (*begin++ != *(--end)) return false;
    return true;
}
```

`begin` `end`
Algorithms

For many problems there are prebuilt solutions in the standard library

Example: filling an array

```cpp
#include <algorithm> // needed for std::fill
...
int a[5];
std::fill (a, a+5, 1);
for (int i=0; i<5; ++i)
  std::cout << a[i] << " "; // 1 1 1 1 1
```

Algorithms

Advantages of using the standard library
- simple programs
- less sources of errors
- good, efficient code
- code independent from the data type
- there are also algorithms for more complicated problems such as the efficient sorting of an array

Excursion: Templates

Templates permit the provision of a type as argument
- The compiler finds the matching type from the call arguments

Example fill with templates

```cpp
template <typename T>
void fill (T * begin, T * end, T value) {
  for (T * p = begin; p != end; ++p)
    *p = value;
}
int a[5];
fill (a, a+5, 1); // 1 1 1 1 1
char c[3];
fill (c, c+3, '!'); // !!!
std::fill is also implemented as template!
```

Algorithms

The same prebuilt algorithms work for many different data types.

Example: filling an array

```cpp
#include <algorithm> // needed for std::fill
...
char c[3];
std::fill (c, c+3, "!");
for (int i=0; i<3; ++i)
  std::cout << c[i]; // !!!
```
Containers and Traversal

- **Container**: Container (Array, Vector, ...) for elements
- **Traversal**: Going over all elements of a container
  - Initialization of all elements (fill)
  - Find the smallest element (min)
  - Check properties (is_palindrome)
  - ...
- There are a lot of different containers (sets, lists, ...)

Iteration Tools

- Arrays: indices (random access) or pointers (natural)
- Array algorithms (std::) use pointers
  ```cpp
  int a[5];
  std::fill (a, a+5, 1); // 1 1 1 1 1
  ```
- How do you traverse vectors and other containers?
  ```cpp
  std::vector<int> v (5, 0); // 0 0 0 0 0
  std::fill (?, ?, 1); // 1 1 1 1 1
  ```

Vectors: *too sexy for pointers*

- Our `fill` with templates does not work for vectors...
- ...and `std::fill` also does not work in the following way:
  ```cpp
  std::vector<int> v (5, 0);
  std::fill (v, v+5, 1); // Compiler error message!
  ```
- Vectors are snobby...
- they refuse to be converted to pointers,...
- ...and cannot be traversed using pointers either.
- They consider this far too primitive.
Vector-Iterators

**Iterator**: a “pointer” that fits to the container.

**Example**: fill a vector using `std::fill` – this works

```cpp
#include <vector>
#include <algorithm> // needed for std::fill
...
std::vector<int> v(5, 0);
std::fill(v.begin(), v.end(), 1);
for (int i=0; i<5; ++i)
  std::cout << v[i] << " "; // 1 1 1 1 1
```

Vector-Iterators: `begin()` and `end()`

- `v.begin()` points to the first element of `v`
- `v.end()` points past the last element of `v`
- We can traverse a vector using the iterator...

```cpp
for (std::vector<int>::const_iterator it = v.begin();
     it != v.end(); ++it)
  std::cout << *it << " ";
```

- ...or fill a vector.

```cpp
std::fill(v.begin(), v.end(), 1);
```

Type names in C++ can become looooooong

- **std::vector<int>::const_iterator**
- The declaration of a type alias helps with

```cpp
typedef Typ Name;
```

**existing type**

**Name that can now be used to access the type**

Examples

```cpp
typedef std::vector<int> int_vec;
typedef int_vec::const_iterator Cvit;
```
Vector Iterators work like Pointers

```cpp
typedef std::vector<int>::const_iterator Cvit;
std::vector<int> v(5, 0); // 0 0 0 0 0
// output all elements of a, using iteration
for (Cvit it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
```

Other Containers: Sets

- A set is an unordered collection of elements, where each element is contained only once.
- \{1, 2, 1\} = \{1, 2\} = \{2, 1\}
- C++: `std::set<T>` for a set with elements of type T

Sets: Example Application

- Determine if a given text contains a question mark and output all pairwise different characters!
Consider a text as a set of characters.

```cpp
#include<set>
...
typedef std::set<char>::const_iterator Csit;
...
std::string text = "What are the distinct characters in this string?";
std::set<char> s (text.begin(), text.end());
```

Set is initialized with String iterator range `[text.begin(), text.end())`

Determine if the text contains a question mark and output all characters

```cpp
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
  std::cout << "Good question!\n";

// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
  std::cout << ∗it;
```

Ausgabe:
Good question!
?Wacdeghinrst

Can you traverse a set using random access? No.

```cpp
for (int i=0; i<s.size(); ++i)
  std::cout << s[i];
```

error message: no subscript operator

Sets are unordered.
- There is no “ith element”.
- Iterator comparison `it != s.end()` works, but not `it < s.end()`!

C++ knows different iterator types
- Each container provides an associated iterator type.
- All iterators can dereference (∗it) and traverse (++it)
- Some can do more, e.g. random access (it[k], or, equivalently ∗(it + k)), traverse backwards (--it),..
The Concept of Iterators

Every container algorithm is generic, that means:
- The container is passed as an iterator-range
- The algorithm works for all containers that fulfil the requirements of the algorithm
- `std::find` only requires `*` and `++`, for instance
- The implementation details of a container are irrelevant.

Why Pointers and Iterators?

Would you not prefer the code

```cpp
for (int i=0; i<n; ++i)
a[i] = 0;
```

over the following code?

```cpp
for (int* ptr=a; ptr<a+n; ++ptr)
    *ptr = 0;
```

Maybe, but in order to use the generic `std::fill(a, a+n, 0);`, we have to work with pointers.

Why Pointers and Iterators?

Example: To search the smallest element of a container in the range `[begin, end)` use the function call

```cpp
std::min_element(begin, end)
```

- returns an `iterator` to the smallest element
- To read the smallest element, we need to dereference:
  ```cpp
  *std::min_element(begin, end)
  ```

In order to use the standard library, we have to know that:
- a static array `a` is at the same time a pointer to the first element of `a`
- `a+i` is a pointer to the element with index `i`

Using the standard library with different containers: Pointers ⇒ Iterators
That is Why: Pointers and Iterators

- Even for non-programmers and “dumb” users of the standard library: expressions of the form
  `*std::min_element(begin, end)`
  cannot be understood without knowing pointers and iterators.
- Behind the scenes of the standard library: working with dynamic memory based on pointers is indispensable. More about this later in this course.

14. Recursion 1

Mathematical Recursion, Termination, Call Stack, Examples, Recursion vs. Iteration

Mathematical Recursion

- Many mathematical functions can be naturally defined recursively.
- This means, the function appears in its own definition

$$n! = \begin{cases} 
1, & \text{if } n \leq 1 \\
n \cdot (n - 1)!, & \text{otherwise}
\end{cases}$$

Recursion in C++: In the same Way!

// POST: return value is n!
unsigned int fac (unsigned int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fac (n-1);
}

$$n! = \begin{cases} 
1, & \text{if } n \leq 1 \\
n \cdot (n - 1)!, & \text{otherwise}
\end{cases}$$

Infinite Recursion

- is as bad as an infinite loop...
- ...but even worse: it burns time and memory

```c
void f()
{
  f(); // f() -> f() -> ... stack overflow
}
```

Recursive Functions: Termination

As with loops we need
- progress towards termination

```c
fac(n):
  terminates immediately for \( n \leq 1 \), otherwise the function is called recursively with \(< n\).
```

"n is getting smaller for each call."

Recursive Functions: Evaluation

Example: `fac(4)`
```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialization of the formal argument: \( n = 4 \)
recursive call with argument \( n - 1 = 3 \)

The Call Stack

For each function call:
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack

```
n = 1   1! = 1
fac(1)  1

n = 2   2 · 1! = 2
fac(2)  2

n = 3   3 · 2! = 6
fac(3)  6

n = 4   4 · 3! = 24
fac(4)  24
```

std::cout << fac(4)
Euclidean Algorithm

- finds the greatest common divisor $\gcd(a, b)$ of two natural numbers $a$ and $b$
- is based on the following mathematical recursion (proof in the lecture notes):

$$\gcd(a, b) = \begin{cases} 
a, & \text{if } b = 0 \\
\gcd(b, a \mod b), & \text{otherwise}
\end{cases}$$

Euclidean Algorithm in C++

```c++
unsigned int gcd(unsigned int a, unsigned int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

Termination: $a \mod b < b$, thus $b$ gets smaller in each recursive call.

Fibonacci Numbers

$$F_n := \begin{cases} 
0, & \text{if } n = 0 \\
1, & \text{if } n = 1 \\
F_{n-1} + F_{n-2}, & \text{if } n > 1
\end{cases}$$

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$

Fibonacci Numbers in C++

```c++
unsigned int fib(unsigned int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2); // n > 1
}
```

Laufzeit

$\text{fib}(50)$ takes “forever” because it computes $F_{48}$ two times, $F_{47}$ three times, $F_{46}$ five times, $F_{45}$ eight times, $F_{44}$ 13 times, $F_{43}$ 21 times ... $F_1$ ca. $10^9$ times (!)

```
unsigned int fib (unsigned int n) 
{ 
    if (n == 0) return 0; 
    if (n == 1) return 1; 
    return fib (n-1) + fib (n-2); // n > 1 
}
```

Correctness and termination are clear.
Fast Fibonacci Numbers

Idea:
- Compute each Fibonacci number only once, in the order $F_0, F_1, F_2, \ldots, F_n$!
- Memorize the most recent two numbers (variables $a$ and $b$)!
- Compute the next number as a sum of $a$ and $b$!

Fast Fibonacci Numbers in $C++$

```c++
unsigned int fib (unsigned int n){
    if (n == 0) return 0;
    if (n <= 2) return 1;
    unsigned int a = 1; // F_1
    unsigned int b = 1; // F_2
    for (unsigned int i = 3; i <= n; ++i){
        unsigned int a_old = a; // F_i-2
        a = b; // F_i-1
        b += a_old; // F_i-1 += F_i-2 -> F_i
    }
    return b;
}
```

Recursion and Iteration

Recursion can always be simulated by
- Iteration (loops)
- explicit “call stack” (e.g. array)

Often recursive formulations are simpler, but sometimes also less efficient.