14. Pointers, Algorithms, Iterators and Containers II

Iterations with Pointers, Arrays: Indices vs. Pointers, Arrays and Functions, Pointers and const, Algorithms, Container and Iteration, Vector-Iteration, Typedef, Sets, the Concept of Iterators

Recall: Pointers running over the Array

```
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

```
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
int* begin = a; // Pointer to the first element
int* end = a+n; // Pointer past the end
for (int* p = begin; p != end; ++p)
    *p = 0;
```

Arrays and Indices

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

Computational costs

```
Adresse von a[0] = a + 0·s
Adresse von a[n-1] = a + (n −1) ·s
```

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

The expression \( a[i] \) is equivalent to \( *(a + i) \).

\[ a[i] \]

\[ \rightarrow \]

\[ a + i \cdot s \]

### Arrays and Pointers

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

Computational cost

\[ \Rightarrow \text{one addition per element} \]

### 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 ...)*

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

define a[10];
define make_null_vector (a); // only sets a[0], a[1], a[2]

define int∗ b;
define 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
- $[\text{begin}, \text{end})$ designates the elements of the part of the array
- *valid range* means: there are array elements “available” here.
- $[\text{begin}, \text{end})$ is empty if begin == end

```cpp
void fill (int* begin, int* end, int value) {
    for (int* p = begin; p != end; ++p)
        *p = value;
}
```

```cpp
int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
    std::cout << a[i] << " ";
```
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.
  
  ```
  int* ptr = 5; // error: invalid conversion from int to int*
  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 \( T* \))
  ```
  int* iptr = 0;
  ```
- cannot be dereferenced (checked during runtime)
- to avoid undefined behavior
  ```
  int* iptr; // iptr points into ‘nirvana’
  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
  
  \[ p1 - p2 \] 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 and Pointers

Where does the `const`-modifier belong to?

*const T* is equivalent to *T const* and can be written like this

- `const int a;` ⟷ `int const a;`
- `const int* a;` ⟷ `int const* a;`
- `int* const a;` ⟷ `const int* a;`
- `int const* const a;` ⟷ `const int* const a;`

Read the declaration from right to left

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

---

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!

// 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;
}

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

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] << " "; // !!!
```
**Excursion: Templates**

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

```cpp
template <typename T>
void fill(T* begin, T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}
```

Example fill with templates

```cpp
int a[5];
fill(a, a+5, 1); // 1 1 1 1 1
char c[3];
fill(c, c+3, '!'); // !!!
```

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
  ```

**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.
**Also in memory: Vector ≠ Array**

```cpp
bool a[8] = {true, true, true, true, true, true, true, true};
```

8 Byte (Speicherzelle = 1 Byte = 8 Bit)

std::vector<bool> v (8, true);

Vector Iterators

For each vector there are two iterator types defined
- **std::vector<int>::const_iterator**
  - for non-mutating access
  - in analogy with const int* for arrays
- **std::vector<int>::iterator**
  - for mutating access
  - in analogy with int* for arrays

A vector-iterator is no pointer, but it behaves like a pointer:
- it points to a vector element and can be dereferenced (*it)
- it knows arithmetics and comparisons (++it, it+2, it < end,...)

**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

```
typedef Typ Name;
```

**Examples**
```
typedef std::vector<int> int_vec;
typedef int_vec::const_iterator Cvit;
```

Vector Iterators work like Pointers

```
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 << " ";
```

```
typedef std::vector<int>::iterator Vit;

// manually set all elements to 1
for (Vit it = v.begin(); it != v.end(); ++it)
    *it = 1;

// output all elements again, using random access
for (int i=0; i<5; ++i)
    std::cout << v[i] << " ";
```

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

```
std::set<int> s = {1, 2};
```
Sets: Example Application

- Determine if a given text contains a question mark and output all pairwise different characters!

Letter Salad (1)

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())`

Letter Salad (2)

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`

Sets and Indices?

- 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()`!
The Concept of Iterators

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),...

Every container algorithm is generic, that means:

- The container is passed as an iterator-range
- The algorithm works for all containers that fulfill 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.

In order to use the standard library, we have to know that:

- A static array `a` is a 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
Why Pointers and Iterators?

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

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

- returns an iterator to the smallest element
- To read the smallest element, we need to dereference:

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

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 indispensible. More about this later in this course.

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!

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

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

Infinite Recursion

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

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

Recursive Functions: Termination

As with loops we need
- progress towards termination

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

"n is getting smaller for each call."

Recursive Functions: Evaluation

Example: \( \text{fac}(4) \)

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

For example:

```
std::cout << fac(4)
n = 4 4 · 3! = 24
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++**

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

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

**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++

**Laufzeit**

`fib(50)` takes “forever” because it computes

\[ F_{48} \text{ two times, } F_{47} \text{ three times, } F_{46} \text{ five times, } F_{45} \text{ eight times, } F_{44} \text{ 13 times, } F_{43} \text{ 21 times } \ldots F_1 \text{ ca. } 10^9 \text{ times} (!) \]

```cpp
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 \( b \)!

```
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} = F_{i-1} + F_{i-2} -> F_i
        a = b;
        b += a_old;  // F_{i-1} += F_{i-2} -> F_i
    }
    return b;
}
```

Very fast, also for \( \text{fib}(50) \)

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.