C++

HS2021

Marco Agostini

Computer Science
University of Applied Sciences of Eastern Switzerland
September 2021

Inhaltsverzeichnis

1 Introduction

1.1 Why C++?

- Work al almost all platforms from a micro controller to the main frame
- Multi-paradigm language with zero-cost abstraction
- High-level abstraction facilities
- The concepts from C++ can mostly be applied to any other programming language

1.2 Undefined Behaviour

The undefined behaviour is defined in the C++ standard (funny, isn't it?). C++ has no garbage collector. If in C++ something is written wrong and the compiler doesn't detect it: undefined behaviour can occur.

1.3 C++ Compilation Process

***.cpp files for source code**

- Also called Ïmplementation File"
- Function implementations (can be in .h files as well)
- Source of compilation - aka Translation Unit"

***.h files for interfaces and templates**

- Called "Header File"
- Declarations and definitions to be used in other implementation files.
- Textual inclusion through a pre-processor (C++20 will incorporate a Modulemechanism)
- #include "header.h"

3 Phases of Compilation

- **Preprocessor** – Textual replacement of preprocessor directives (#include)
- **Compiler** – Translation of C++ code into machine code (source file to object file)
- **Linker** - Combination of object files and libraries into libraries and executables

1.4 Declarations and Definitions

All things with a name that you use in a C++ program must be declared before you can do so!

Defining Functions

$$< return - type >< function - name > (< parameters >)\{/ * body * /\}$$

Tells the compiler that there is a function named $< function-name >$ that takes the parameters $< parameters >$ and returns a value of type $< return - type >$. The Signature of a function is just the combination of name and the parameter types.

One Definition Rule
While a program element can be declared several times without problem there can be only one definition of it.
(ODR = One Definition Rule)

Include Guard
Include guards ensure that a header file is only included once. Multiple inclusions could violate the One Definition Rule when the header contains definitions.

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_
#include <iosfwd>
struct Greeter {
};
#endif /* SAYHELLO_H_ */
```

2 Values and Streams

2.1 Variable Definitions

Defining a variable consists of specifying its <type>, its <variable-name> and its <initial value>. Empty braces mean default initialisation. Using = for initialisation we can have the compiler determine its type (do not combine with braces!).

$$< type >< variable - name >< initial - value >;$$

Constants
Adding the const keyword in front of the name makes the variable a single-assignment variable, aka a constant. A const must be initialised and is immutable.

When should const be used?

- A lot of code needs names for values, but often does not intend to change it
- It helps to avoid reusing the same variable for different purposes (code smell)
- It creates safer code, because a const variable cannot be inadvertently changed
- It makes reasoning about code easier
- Constness is checked by the compiler
- It improves optimization and parallelization (shared mutable state is dangerous)

Where to place Variable definition?
Do not practice to define all (potentially) needed variables up front (that style is long obsolete!). Every mutable global variable you define is a design error!

A Note on Naming
The C++ convention is to begin variable names with a lower case letter. Spell out what the variable is for and do not abbreviate!

Types for Variables
Are part of the language and don't need an include.

- short, int, long, long long – each also available as unsigned version
- bool, char, unsigned char, signed char - are treated as integral numbers as well
- float, double, long double

2.2 Values and Expressions

C++ provides automatic type conversion if values of different types are combined in an expression. Dividing integers by zero is undefined behavior.

Literal Example	Type	Value
'a' '\n' '\x0a'	char char char	Letter a, value: 97 <NL> character, value: 10 <NL> character, value: 10
1 42L 5LL int{} (not really a literal)	int long (größer) long long int	1 42 5 0 (default value)
1u 42ul 5ull	unsigned int unsigned long unsigned long long	1 42 5
020 0x1f 0XFULL	int int unsigned long long	16 (octal 20) 31 (hex 1F) 15 (hex F)
0.f .33 1e9 42.E-12L .3l	float double double long double long double	0 0.33 1000000000 (10^9) 0.00000000042 ($42*10^{-12}$) 0.3
"hello" (n+1) "\012\n\\"	char const [6] char const [4]	Array of 6 chars: h e l l o <NUL> Array of 4 chars: <NL> <NL> \ <NUL>

```
(5 + 10 * 3 - 6 / 2) // precedence as in normal mathematics = 32
auto x = 3; / 3 // Fractions results of int operations always rundet down! 1
auto y = x%2 ? 1 : 0; // int to boolean conversion 0=false, all other are
    true. = 1
```

2.3 Strings

std::string is C++'s type for representing sequences of char (which is often only 8 bit). This Strings are mutable in C++ in contrast to Java. Literals like äbäre not of type std::string they consist of const chars in a null terminated array.

To have a std::string we need to append an s. This requires using namespace std::literals;.

```
 void printName(std::string name) {
  using namespace std::literals;
  std::cout << "my name is: "s << name;
}
```

String Capabilites
You can iterate over the contents of a string.

```
void toUpper(std::string & value) {
  for (char & c : value) {
    c = toupper(c);
  }
}
```

2.4 Input and Output Streams

Functions taking a stream object must take it as a reference, because they provide a side-effect to the stream (i.e., output characters).

Simple I/O Stream objects provide C++'s I/O mechanism with the help of the pre-defined globals: std::cin std::cout. Should only be used in the main function! Streams have a state that denotes if I/O was successful or not.

- Only .good() streams actually do I/O
- You need to .clear() the state in case of an error
- Reading a std::string can not go wrong, unless the stream is already !good().

Reading a std::strting Value

```
#include <iostream>
#include <string>
  std::string inputName(std::istream & in) {
    std::string name{};
    in >> name;
    return name;
}
```

Reading an int Value

```
int inputAge(std::istream& in) {
  int age{-1};
  if (in >> age) { // Boolean conversion
  return age;
  }
  return -1;
}
```

Chaining Input Operations

- Multiple subsequent reads are possible
- If a previous read already failed, subsequent reads fail as well

```
std::string readSymbols(std::istream& in) {
  char symbol{};
  int count{-1};
  if (in >> symbol >> count) {
    return std::string(count, symbol);
  }
```

```
7     return "error";
8 }
```

Stream States
Formatted input on stream is must check for is.fail() and is.bad(). If failed, is.clear() the stream and consume invalid input characters before continue.

State Bit Set	Query	Entered
<none>	is.good()	initial is.clear()
failbit	is.fail()	formatted input failed
eofbit	is.eof()	trying to read at end of input
badbit	is.bad()	unrecoverable I/O error

Headers
As a general advise the most matching include should be used.

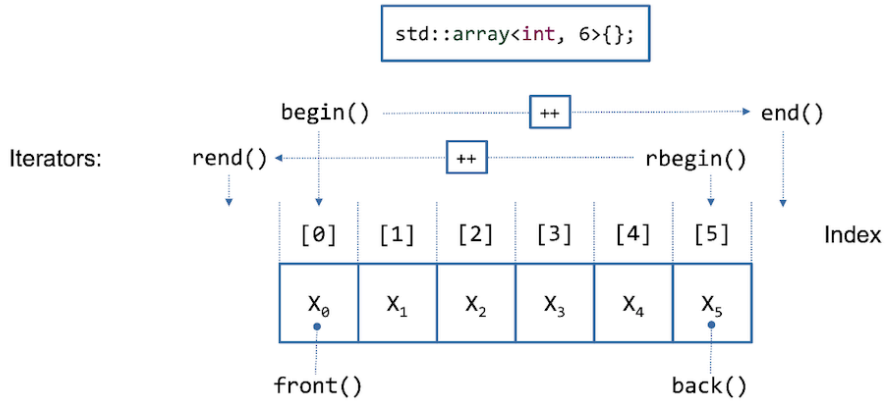
- *iosfwd* contains only the declarations for std::ostream and std::istream. This is sufficient for function declarations.
- *istream* and ostream contain the implementation of the corresponding stream, operators.
- *iostream* contains all of the above and additionally std::cout, std::cin, std::cerr. This is only required in the source file containing the main() function, because only there the global standard IO variables shall be used

3 Sequences and Iterators

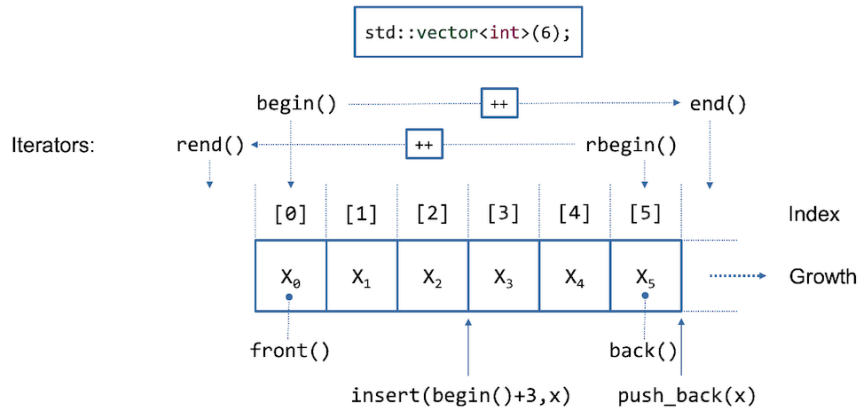
3.1 Introduction to std::array<T, N> and std::vector<T>

Array
C++'s std::array<T, N> is a fixed-size Container. T is a template type parameter (= placeholder for type). N is a positive integer, template non-type parameter (= placeholder for a value). Elements can be accessed with a subscript operator [] or at(). . The size is bound to the array object and can be queried using .size();. Avoid plain C-Array whenever possible: int arr[]1, 2, 3, 4, 5;

- at() throws an exception on invalid index access
- [] has undefined behavior on invalid index access Behavior
- The size of an array must be known at compile-time and cannot be changed. Otherwise it contains N default-constructed elements: std::array<int, 5> emptyArray;



Vector
C++'s std::vector<T> is a Container = contains its elements of type T (no need to allocate them). java.util.ArrayList<T> is a collection = keeps references to T objects (must be "new"ed). T is a template type parameter (= placeholder for type). std::vector can be initialized with a list of elements. Otherwise it is empty: std::vector<double> vd;.



Append Elements to an std::vector<T>

- *v.push_back*(< *value* >);
- *v.insert*(< *iterator − position* >, < *value* >);

Filling a Vector with Values

```
std::vecor<int> v{};
v.resize(10);
std::fill(std::begin(v), std::end(v), 2);

std::vector<int> v(10);
```

```
std::fill(std::begin(v), std::end(v), 2);

std::vector v(10, 2);

// Filling increased values with iota
std::vector<int> v(100); std:iota(std::begin(v), std::end(v), 1);
```

Finding and counting elements of a vector
std::find() and std::find_if() return an iterator to the first element that matches the value or condition.

```
auto zero_it = std::find(std::begin(v), std::end(v), 0); if (zero_it == std
    ::end(v)) {
std::cout << "no zero found \n"; }
```

3.2 Iteration

Its possible to index a vector like an array but there is no bounds check. Accessing an element outside the valid range is Undefined Behavior.

Bad Style Iteration!

```
for (size_t i = 0; i < v.size(); ++i) {  //Index is "unsigned" 0-1=MAX_INT
  std::cout << "v[" << i << "] = " << v[i] << '\n'; }
}
```

Elemt Iteration (Range-Based for)

- Advantage: No index error possible
- Works with all containers, even value lists 1, 2, 3

	const: • element cannot be changed	**non-const:** • element can be changed

reference:
• element in vector is accessed
	`for (auto const & cref : v) {`
	`  std::cout << cref << '\n';`
	`}`
		`for (auto & ref : v) {`
		`  ref *= 2;`
		`}`

copy:
• loop has own copy of the element
	`for (auto const ccopy : v) {`
	`  std::cout << ccopy << '\n';`
	`}`
		`for (auto copy : v) {`
		`  copy *= 2;`
		`  std::cout << copy << '\n';`
		`}`

Iteration with Iterators

```
for (auto it = std::begin(v); it != std::end(v); ++it) {
  std::cout << (*it)++ << ", ";
}
// Guarantee to just have read-only access with std::cbegin() and std::cend
    ()
for (auto it = std::cbegin(v); it != std::cend(v); ++it) {
  std::cout << *it << ", ";
}
```

3.3 Using Iterators with Algorithms

Each algorithm takes iterator arguments. The algorithm does what its name tells us.

```
// Counting blanks in a string
size_t count_blanks(std::string s) {
  size_t count{0};
  for (size_t = 0; i < s.size(); ++i) {
```

```
    if (s[i] == ' ') {
      ++count;
    }
  }
  return count;
 }

// Counting blanks in a string with algorithms
size_t count_blanks(std::string s) {
  return std::count(s.begin(), s.end(), ' ');
}

// Summing up all values in a vector
std::vector<int> v{5, 4, 3, 2, 1};
std::cout << std::accumulate(std::begin(v), std::end(v), 0)<< " = sum\n";

// Number of elements in range
void printDistanceAndLength(std::string s) {
  std::cout << "distance: "<< std::distance(s.begin(), s.end()) <<'\n';
  std::cout << "in a string of length: "<< s.size()<<'\n';
}

// Printing all values of a vector
void printAll(std::vector<int> v) {
  std::for_each(std::crbegin(v), std::crend(v), print);
}

// For each with a Lambda
void printAll(std::vector<int> v, std::ostream & out) {
  std::for_each(std::crbegin(v), std::crend(v), [&out](auto x) {
    out << "print: "<< x << '\n';
  });
}
```

3.4 Iterators for I/O

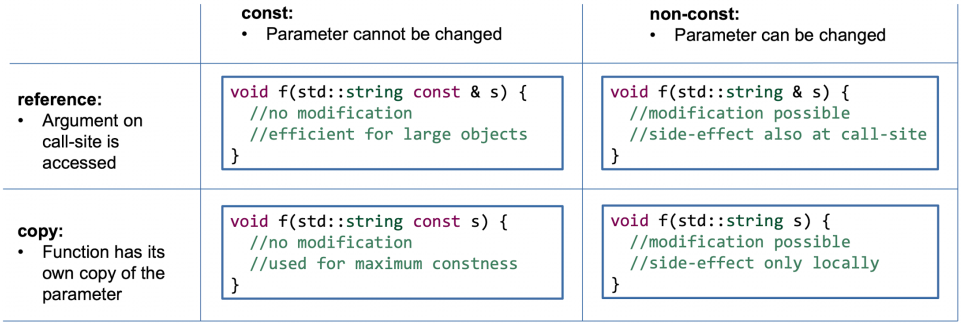
Iterators connect streams and algorithms. Streams (std::istream and std::ostream) cannot be used with algorithms directly.

- $std :: ostream\_iterator < T >$ outputs values of type T to the given std::ostream
  - No end() marker needed for output, it ends when the input range ends.
- $std :: istream_iterator < T >$ reads values of type T from the given std::istream
  - End iterator is the default constructed $std :: istream_iterator < T >$
  - It ends when the stream is no longer good().

4 Functions and Exceptions

4.1 Function Parameter Declarations

Parameter Declarations



- Value Parameter - Default void f(type par);
- Reference Parameter - site-effect void f(type & par);
- Const-Reference Parameter - optimisation void f(type const & par);
- Const Value Parameter - Prevent changing the para void f(type const par);

Function Return Type

- By (Const) Value - default type f(); or type const f();
- By Reference - Only return a reference parameter (or a call member variable from a member function) type & f(); or type const & f();

4.2 Function Overloading and Default Arguments

Function Overloading
The same function name can be used for different functions if parameter number or types differ. Function can not be overloaded just by their return type! If only the parameter type is different there might be ambiguities. The resolution fo overloads happens at compile-time = Ad hoc polymorphism.

Default Arguments
A function declaration can provide default arguments for its parameters *from the right.*

```
void incr(int & var, unsigned delta = 1);
// Default arguments can be omitted when calling the function
int counter {0};
incr(counter); // uses default for delta
```

Functions as Parameters
Functions are first classöbjects in C++. You can pass them as augment and you can keep them in reference variables.

Caution! Call Sequence!
Statements are sequenced by ; (semicolon). Within a single expression, such as a function call, sequence of evaluation is undefined!

4.3 Lambda Function

Defining Inline functions. Auto const for function variable for Lambda. [] introduces a Lambda function. Can contain captures: [=] or [&] to access variables from scope.

```
auto const g = [](char c) -> {
  return std::toupper(c)M
};
g('a');
```

Lampda Caputre Examples

- Capturing a local variable by value
  - Local copy loves as long as the lambda lives
  - Local copy is immutable, unless lambda is declared as mutable
- Capturing a local variable by reference
  - Allows modification of the captured variable
  - Side-effect is visible in the surrounding scope, but referenced variable must live at least as long as the lambda lives.

```
// Capturing by value
int x = 5;
auto l = [x]() mutable {
  std::cout << ++x;
};
// Capuring by reference
auto const l = [&x]() {
  std::cout << ++x;
};
```

4.4 Failing Functions

What should we do, if a function cannot fulfil its purpose?

1. Ignore the error and provide potentially undefined behaviour
2. Return a standard result to cover the error
3. Return an error code or error value
4. Provide an error status as a side-effect
5. Throw an Exception

Ignore the Error

- Relies on the caller to satisfy all preconditions.
- Viable only if not dependent on other resources.
- Most efficient implementation.
- Simple for the implementer but hard for the caller.

Cover the Error with a Standard Result

- Reliefs the caller from the need to care if it can continue with the default value
- Can hide underlying problems.
- Often better if caller can specify its own default value.

Error Value

- Only feasible if result domains is smaller than return type
- POSIX defines -1 to mark failure of system calls
- Burden on the caller to check the result

Cover the Error with a Standard Result

- Requires reference parameter
- (Bad!) Alternative: global variable (POSIX: errno)
- E.g: std::istreams's states (good(), fail()) is changed as a side-effect of input

4.5 Exceptions

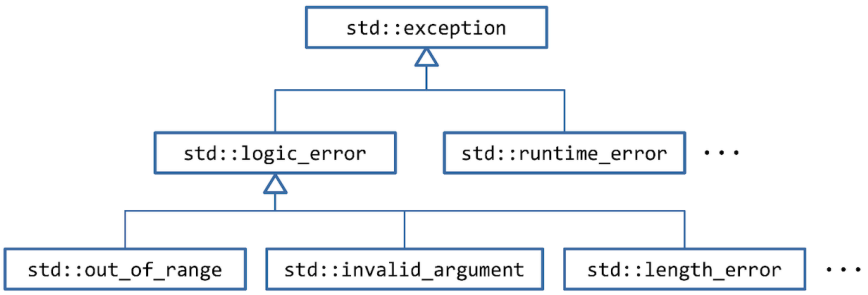
An exception can throw any copyable type. No means to specify what could be thrown. No check if you catch an exception that might be thrown at call-site. No meta-information is available as part of the exception. Exception thrown while exception is propagated results in a program abort (not while caught).

Catching Exceptions

Principle: Throw by value, catch by const reference. This avoids unnecessary copying and allows dynamic polymorphism for class types.

```
try {
throwingCall();
} catch (type const & e) {
  //Handle type exception
} catch (type2 const & e) {
  //Handle type2 exception
} catch (...) {
  //Handle other exception types
}
```

The Standard Library has some pre-defined exception types that you can also use in <stdexcept>. All have a constructor parameter for the reasonöf type std::string. It provides the what() member function to obtain the reason"



Keyword noexcept
Functions can be declared to explicitly not throw an exception with the noexcept keyword. The compiler does not need to check it. If an exception is thrown (directly or indirectly) from a noexcept function the program will terminate.

5 Classes and Operators

5.1 Classes

Are usually defined in header files and not in *.cpp files! Does one thing and is well named after that. Consists of member functions with only a few lines. Has a class invariant: provides guarantee about its state (values of the member variables). Constructors establish that invariant. Is easy to use without complicated protocol sequence requirements.

A class is usually defined in a header file. At the end of a class definition a semicolon is required.

Class Head
Keywords for defining a class: class or struct. Default visibility for members of the class are: private for class and public for struct.

Access Specifier

- private: visible only inside the class (and friends); for hidden data members
- protected: also visible in subclasses
- public: visible from everywhere; for the interface of the class

Member Variables
Don't make member variables const as it prevents copy assignment. Don't add members to communicate between member function calls.

Static Member Functions and Variables
No *static* in *.cpp file only in *.h file!

Constructors
Function with name of the class and no return type.

- Default Constructor -No parameters. Implicitly available if there are no other explicit constructors. Has to initialize member variables with default values.
- Copy Constructor - Has one <own-type> const & parameter. Implicitly available (unless there is an explicit move constructor or assignment operator). Copies all member variables.
- Move Constructor - Has one <own-type> && parameter. Implicitly available (unless there is an explicit copy constructor or assignment operator). Moves all members
- Typeconversation Constructor - Has one <other-type> const & parameter. Converts the input type if possible. Declare explicit to avoid unexpected conversions.
- Initializer List Constructor - Has one std::initializer_list parameter. Does not need to be explicit, implicit conversion is usually desired. Initializer List constructors are preferred if a variable is initialized with { }
- Destructor - Named like the default constructor but with a ∼. Must release all resources. Implicitly available. Must not throw an exception. Called automatically at the end of the block for local instances.

```
// File: date.h
#ifndef DATE_H_
#define DATE_H_
class Date { // HEAD
  int year, month, day; // MEMBER VARIABLES
public:
  Date(int year, int month, int day) // CONSTRUCTOR
  : year{year}, month{month}, day{day} {/*...*/} // MEMBER INITIALISER LIST
  static bool isLeapYear(int year) {/*...*/}
private:
  bool isValidDate() const {/*...*/} };
#endif /* DATE_H_ */
```

```
class Date {
public:
    Date(int year, int month, int day);
    Date(); // Default-Constructor
    Date() = default; // explizit Default-Constructor
    Date(Date const &); // Copy-Constructor
    Date(Date &&); // Move-Constructor
    explicit Date(std::string const &); // Typeconversation-Constructor
```

```
    Date(std::initializer_list<Element> elements); // Initializer List-
    Constructor
    ~Date(); // Destructor
    Date(Date const &) = delete; // delete implicit Copy-Constructor
};
```

5.2 Inheritance

Base classes are specified after the name: $class < name >:< base1 >, ..., < baseN >$. Multiple inheritance is possible and inheritance can specify visibility. If no visibility is specified the default of the inheriting class is used.

```
class Base {
private:
  int onlyInBase;
protected:
  int baseAndInSubclasses;
public:
  int everyoneCanFiddleWithMe
};
class Sub : public Base {
  //Can see baseAndInSubclasses and
  //everyoneCanFiddleWithMe
};
```

5.3 Operator Overloading

Custom operators can be overloaded for user-defined types. Declared like a function, with a special name: <returntype> operator op(<parameters>);. Unary operators -> one parameters and binary operators -> two parameters.

Free Operator
Free operator< uses two parameters of Date each *const* & return type *bool*. Is inline when defined in header. The only problem we have is that we don't have access to private members.

```
// File Any.cpp
#include "Date.h" Any.cpp
#include <iostream>
void foo() {
  std::cout << Date::myBirthday;
  Date d{};
  std::cin >> d;
  std::cout << "is d older? " << (d < Date::myBirthday);
}

// File Date.h
class Date {
  int year, month, day; // private :-(
};
inline bool operator<(Date const & lhs, Date const & rhs) {
  return lhs.year < rhs.year ||  // Does not WOKR!
  (lhs.year == rhs.year && (lhs.month < rhs.month ||
    (lhs.month == rhs.month && lhs.day == rhs.day)));
}
```

Member Operator
Member operator< uses one parameter of type *Date*, which is *const&*, return type *bool* and Right-hand side of operation. Implicit this object: *const* due to qualifier, left-hand side of operation.

```
// File Any.cpp
#include "Date.h"
```

```cpp
#include <iostream>
void foo() {
  std::cout << Date::myBirthday;
  Date d{};
  std::cin >> d;
  std::cout << "is d older? " << (d < Date::myBirthday);
}
// File Date.h
class Date {
  int year, month, day; // private :-)
  bool operator<(Date const & rhs) const {
  return year < rhs.year ||
  (year == rhs.year && (month < rhs.month ||
  (month == rhs.month && day == rhs.day)));
  }
};
```

6 Enums and Namespaces

6.1 Namespaces

- Namespaces are scopes for grouping and preventing name clashes
- Global namespaces has the :: prefix
- Nesting of namespaces is possible
- Nesting of scopes allows hiding of names
- Namespaces can only be defined outside of classes and functions
- The same same namespace can be opened and closed multiple times
- Qualified names are. used to access names in a namespace: demo::subdemo::foo()
- A name with a leading :: is called fully qualified name: ::std::cout.

Using Declarations

- Import a name from a namespace into the current scope
  - That name can be used without a namespace prefix
  - Useful if the name is used very often
- Alternative: using alias for types if name is long
- There are also using directives, which import ALL names of a namespace into the current scope.
  - Use them only in local scope to avoid pollutionöf your namespace.

Anonymous Namespaces

- Special case: omit name after namespace
- Implicit using directive for the chosen stream
- Hides modules internals
- Use them only in source files (*.cpp)

6.2 Name Resolution of Namespace Members

Types and (non-member) functions belonging to that type should be placed in a common namespace. The Advantage is *Argument Dependent Lookup! ADL:* When the compiler encounter an unqualified function or operator call with an argument of a user-defined type it looks into the namespaces in which that type is defined to resolve the functionoperator. E.g. it is not necessary to write std:: in front of for_each when std::vector::begin() is an argument of the function.

6.3 Enumerations

Enumerations are useful to represent types with only a few values. An enumeration creates a new type that can easily be converted to an integral type. The individual values (enumerators) are specified in the type. Unless specified explicitly, the values start with 0 and increase by 1.

6.4 Arithmetic Types

Disclaimer: You usually do not want to implement your own arithmetic types! We will cover the basics.

- Arithmetic types must be equality comparable
- Boost can be used to get != operator → boost::equality_comparable
- It might be convenient to have the output operator
- Result must be in a specific range (Modulo)

7 Standard Containers

There are three main types of standard containers in the C++ language.
Containers can be: default-constructed, copy-constructed from another container of the same type, equality compared, emptied with clear().

- Sequence Containers
  - Elements are accessible in order as they were inserted/created.
  - Find in linear time through the algorithm find.
- Associative Containers
  - Elements are accessible in sorted order
  - find as member function in logarithmic time
- Hashed Containers
  - Elements are accessible in unspecified order
  - find as member function in constant time

Member Function	Purpose
`begin()` `end()`	Get iterators for algorithms and iteration in general
`erase(iter)`	Removes the element at position the iterator `iter` points to
`insert(iter, value)`	Inserts value at the position the iterator `iter` points to
`size()` `empty()`	Check the size of the container

Abbildung 1: Member Function of a Container

7.1 Common Container Constructors

```
// Constructor with Initializer List
std::vector<int> v{1,2,3,5,6,11};
// Construction with a number of elements, five times a 42
std::list<int> l(5,42);
// Range with a pair of iterators
std::deque<int> q{begin(v), end(v)};
```

7.2 Sequence Container

std::vector<T>, std::deque<T>, std::list<T>, std::array<N, T>.
Defines order of elements as inserted/appended to the container. Lists are very good for splicing and in the middle insertions. Array and deque are very efficient unless bad usage.

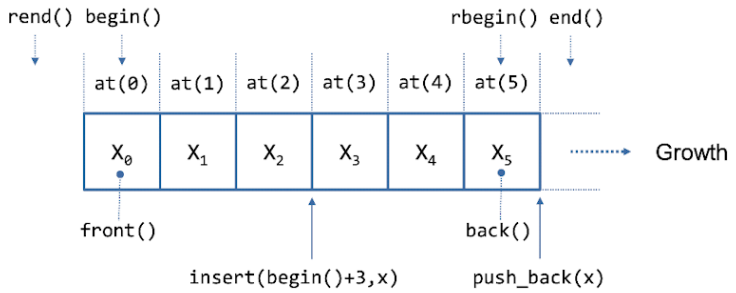


Abbildung 2:

7.3 Associated Containers

Can be searched by content and not by sequence.

	Key Only	Key-Value Pair
Key Unique	`std::set<T>`	`std::map<K, V>`
Multiple Equivalent Keys	`std::multiset<T>`	`std::multimap<K, V>`

Abbildung 3:

7.4 Hashed Containers

Introduced in C++11. Standard lacks feature for creating your own hash functions.

7.5 Iterators

Different containers support iterators of different capabilities. Categories are formed around increasing power".

Input Iterator
Supports reading the currentëlement (of type Element). Allows for one-pass input algorithms. Can be compared with == and !=. Can also be copied.

```
struct input_iterator_tag{};

Element operator*();
It & operator++();
It operator++(int);
bool operator==(It const &);
bool operator!=(It const &);
It & operator=(It const &);
It(It const &); //copy ctor
```

Forward Iterator
Can do whatever an input iterator can do plus: supports changing the current element. Still allows only for one-pass input algorithms.

```
struct forward_iterator_tag{};
Element & operator*();
It & operator++();
It operator++(int);
bool operator==(It const &);
bool operator!=(It const &);
It & operator=(It const &);
It(It const &); //copy ctor
```

Bidirectional Iterator
Can do whatever the forward iterator can do plus going backwards.

```
struct bidirectional_iterator_tag{};
It & operator--();
It operator--(int);
```

Random Access Iterator
Can do what the bidiretional iterator can do plus: Directly access element at index (offset to current position): distance can be positive or negative, Go n steps forward or backward, SSubtacttwo iterators to get the distance, Compare with relational operators (<, <=, >, >=). Allows also random access in algorithms.

Output Iterators
Can write value to current element, but only once (it = value). Modeled after std::ostream_iterator.

```
struct output_iterator_tag{};
Element & operator*();
It & operator++();
It operator++(int);
```

7.5.1 Iterator Functions

Has two functions std::distance(start, goal); std::advance(itr, n);.

```
int main() {
std::vector<int> primes{2, 3, 5, 7, 11, 13};
auto current = std::begin(primes); auto afterNext = std::next(current); std
    ::cout << "current: " << *current << " afterNext: " << *afterNext << '\n'
    ;
std::advance(current, 1); std::cout << "current: " << *current << "
    afterNext: " << *afterNext << '\n';
}
```

8 STL Algorithms

The älgorithm.häre the algorithms defined for general purpose. and in the numeric.häre the general numeric functions.

What is are the advantages of the STL algorithms?

- Correctness
  - It is much easier to use an algorithm correctly than implementing loops correctly.
- Readability
  - Applying the correct algorithm expresses your intention much better than a loop.
  - Someone else will appreciate it when the code is readable and easily understandable.
- Performance
  - Algorithms might perform better than handwritten loops

Iterator for Ranges

- First - Iterator pointing to the first element.
- Last - Iterator pointing to the last element.
- if First == Last the range is empty.

```
std::vector<int> values{54, 23, 17, 95, 85, 57, 12, 9};
std::xxx(begin(values), end(values), ...);
```

8.1 Functor

8.2 Examples

8.3 Pitfalls

9 Function Templates

Can be compared as a Generic in Java. The keyword templateïs used to declare a template. The template parameter list contains one or more templates parameters. The compiler resolves the function template and figures out the template arguments. C++ uses duck-typing. So every type can be used as argument as long as it supports the used operations.

```
template <Template-Parameter-List>
FunctionDefinition
```

```
// file min.h
template <typename T>
T min(T left, T right) {
  return left < right ? left : right;
}
// file smaller.cpp
#include "min.h" smaller.cpp
#include <iostream>
int main() {
  int first;
  int second;
  if (std::cin >> first >> second) {
    auto const smaller = min(first, second); std::cout << "Smaller of " <<
   first << " and " << second << " is: " << smaller << '\n';
  }
}
```

Template Definition

- Templates are usually defined in a header file
  - A compiler needs to see the whole template definition to create an instance
  - Function template definitions are implicitly inline
- The definition in a source file is possible, but then it can only be used in that translation unit.
- Type checking happens twice
  - When the template is defined: only basic checks are performed: syntax and resolution of names that are independent of the template parameters.
  - When the template is instantiated (used): The compiler checks whether the template arguments can be used as required by the template.

Template Argument Deduction