C++

HS2021

Marco Agostinis

marco.agostini@ost.ch

Computer Science
University of Applied Sciences of Eastern Switzerland
September 2021

Contents

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 Features

- C++ doesnt't has no methods only functions. A function does not have to be a member of an object. If a function belongs to an object it's a member function.
- Please do not write your own loops in C++ try to use the STL (Standard Template Library).
- C++ is compatible with standard C.
- There is no Garbage Collector!
- With a library we can publish functionalities to another program.

1.3 Terminology

Value 42
Type int, char, bool, long, float
Variable int const i{42}
Expression (2+4)*3

Statement `while ( true );`
Declaration int foo();
Definition int j;
Function void bar ( ) { }

1.4 Undefined Behaviour

The undefined behaviour is defined in the C++ standard (funny, isn't it?). Because of the fact, that C++ doesn't have a garbage collection, if in C++ something is written wrong and the compiler doesn't detect it: undefined behaviour can occur.

1.5 C++ Compilation Process

C++ has the advantage of direct compilation into machine code. This eliminates the overhead for a virtual machine in comparison to Java.

***.cpp files for source code**

- Also called "Implementation 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 "Module" mechanism)
- `#include "header.h"`

3 Phases of Compilation

- **Preprocessor** – Textual replacement of preprocessor directives, results in (*.i) files. (`#include`)
- **Compiler** – Translation of C++ code into machine code (source file (*.i) to object file (*.o))
- **Linker** - Combination of object files (*.o) and libraries into libraries and executables (*.exe).

1.6 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.

```
1 #ifndef SAYHELLO_H_
2 #define SAYHELLO_H_
3 #include <iosfwd>
4 struct Greeter {
5 };
6 #endif /* SAYHELLO_H_ */
```

2 Variables

- Variables always start with a lower case character
- Local variables must always contain a default value (Curly brackets or =).
- A global variable must never be mutable! (Hard to test and can cause problems when multithreading is used)
- Variables are as default value types and therefore declared on the stack.

2.1 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 into an expression. Dividing integers by zero is undefined behaviour.

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

2.3 Const

- Const should be used as often as possible, because it optimises the code.
- Const is comparable with the final from Java, although it has a higher guarantee that the variable is not changed.
- Const variables must be initiaized!
- To set const vars at compile time the keyword "constexpr" must be used.

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 .31	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>

Figure 1: C++ Variable Types

2.4 Auto

The keyword auto can be used to deduct the type of a variable automatically at the declaration.

```
1 auto const yearOfBirth = 2049; // int
2 auto const name = "Rick Deckard" // std::string
```

2.5 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 "ab" are 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;.

```
1  void printName(std::string name) {
2   using namespace std::literals;
3   std::cout << "my name is: "s << name;
4 }
```

String Capabilites

You can iterate over the contents of a string.

```
1 void toUpper(std::string & value) {
2   for (char & c : value) {
3     c = toupper(c);
4   }
5 }
```

3 Streams

- In the header files the inclusion of `#include <iosfwd>` forward declaration header. This is sufficient for function declarations.
- In a source file for "std::cin" and "std::cout" the `#include <iostream>` should be used. This containts all the definitions needed for "std::cin" etc. If just one of the two stream objects is need use either `#include <ostream>` or `#include <istream>`. The last two dont include the "std::cin" and "std::cout".
- In the main function "std::cin" and "std::cout" is used with the corresponding shift operators "«", "»".
- "std::istream" objects do return false if we are in an invalid stream state.
- "std::endl" flushes a buffered out stream. Better use "\n".

3.1 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. 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);
  }
  return "error";
}
```

3.2 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.

3.3 Manipulators

For the formatting of the output a vide variety of manipulator can be used.

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

Figure 2: Stream States in C++

4 Iterators

There are always two iterators used (begin() und end()). There is also the possibility to traverse a list from front to back (rbegin() and rend()). If the members are only read the const version (cbegin() and cend()) can be used.

4.1 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'; }
}
```

Element 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 << ", ";
}
```

4.2 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) {
```

```
14   return std::count(s.begin(), s.end(), " ");
15 }
16
17 // Summing up all values in a vector
18 std::vector<int> v{5, 4, 3, 2, 1};
19 std::cout << std::accumulate(std::begin(v), std::end(v), 0)<< " = sum\n";
20
21 // Number of elements in range
22 void printDistanceAndLength(std::string s) {
23   std::cout << "distance: "<< std::distance(s.begin(), s.end()) <<'\n';
24   std::cout << "in a string of length: "<< s.size()<<'\n';
25 }
26
27 // Printing all values of a vector
28 void printAll(std::vector<int> v) {
29   std::for_each(std::crbegin(v), std::crend(v), print);
30 }
31
32 // For each with a Lambda
33 void printAll(std::vector<int> v, std::ostream & out) {
34   std::for_each(std::crbegin(v), std::crend(v), [&out](auto x) {
35     out << "print: "<< x << '\n';
36   });
37 }
```

4.3 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().

4.4 Types

There are five different types of iterators in C++.

```
1 struct input_iterator_tag { };
2 struct output_iterator_tag { };
3 struct forward_iterator_tag : public inputer_iterator_tag { };
4 struct bidirectional_iterator_tag : public forward_iterator_tag {};
5 struct random_access_iterator_tag : public bidirectional_iterator_tag { };
```

4.4.1 Input Iterator

- The element can be read only once and after that the iterator has to be incremeneted.
- Used for `std::istream_iterator` and `std::istreambuf_iterator`

4.4.2 Forward Iterator

- Element can be read in and changed (Except element or container is const).
- Only allows forward iteration
- Sequenz can be iterated over multiple times

4.4.3 Bidirectional Iterator

- Element can be read in and changed (Except element or container is const).
- Allows forward and backwards iteration
- Sequenz can be iterated over multiple times
- The random access iterator behaves as the bidirectional iterator with the addition that the can access elements over the index

4.4.4 Output Iterator

- Current element can be changed once, after that the iterator has to be incremented.
- There is no end for this iterator (example console prints)
- Used for `std::ostream_iterator`
- Writes the result without knowing the result.

5 Functions

- Functions are always written in lower Camel Case
- A function must be declared always in a header file before the function is used
- A good function has a maximum of five parameters and does exactly one thing
- The call of the function parameters is not defined.
- The main function does implicit return a "0".
- Auto should not be used as a return type, exceptions are: inline, template or constexpr functions in header files.
- Void should not be used as a function parameter
- NEVER return a ref to a local variable since it produces a dangling Reference, because the value lives in the stack frame.

5.1 Default Arguments

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

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

5.2 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.

```
1 void incr(int & var);
2 int incr(int & var); // doesn't compile because of same signature
3 void incr(int & var, unsigned delta);
```

5.3 Reference / Value Arguments

Parameter Declarations

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

reference:
• Argument on call-site is accessed
	`void f(std::string const & s) {`
	`  //no modification`
	`  //efficient for large objects`
	`}`
		`void f(std::string & s) {`
		`  //modification possible`
		`  //side-effect also at call-site`
		`}`

copy:
• Function has its own copy of the parameter
	`void f(std::string const s) {`
	`  //no modification`
	`  //used for maximum constness`
	`}`
		`void f(std::string s) {`
		`  //modification possible`
		`  //side-effect only locally`
		`}`

- Value Parameter - Default void f(type par);
- Reference Parameter - side-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();

Functions as Parameters

Functions are "first class" objects in C++. You can pass them as augment and you can keep them in reference variables.

5.4 Variadic Arguments

Variadic functions take a variable number of arguments. This example is even a template function with variadic arguments.

```
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
  std::cout << first;
  if (sizeof...(Types)) {
    std::cout << ", ";
  }
  printAll(rest...);
}
```

5.5 Lambdas

- Can be written into variables `auto l = [](); l();`
- The smallest lambda is `[](){}` the first two brackets are the function object and the round brackets the call.

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');
```

5.5.1 Captures

Captured variables are imutable default. To change them they have to be declared as `mutable`. Captures are need to use variables outside of the lambda.

- `[=]` - default implicit capture variables used in body by value
- `[&]` - default capture variable used in body by reference
- `[var = value]` - introduce new capture variable with value
- `[=,& out]` - capture all by copy, out by reference
- `[&, = x]` - capture all by reference, but x by copy/value

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

5.6 Functor

Functors are types which provide an operation. Functors have an overloaded call operator. Lambdas internally work with functors. The `operator()` function can theoretically be overload as often as needed.

```
struct Accumulator {
  int count{0};
  int accumulated_value{0};
  void operator()(int value) {
    count++;
    accumulated_value += value;
  }
  int average() const {
    return accumulated_value / count;
```

```
10   }
11   int sum() const;
12 };
13
14 int average(std::vector<int> values) {
15   Accumulator acc{};
16   for(auto v : values) { acc(v); }
17   return acc.average();
18 }
19 int main(int argc, char **argv) {
20   std::vector<int> values { 1, 2, 6, 4, 5, 3 };
21   std::cout << average(values);
22 }
```

6 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).

6.1 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

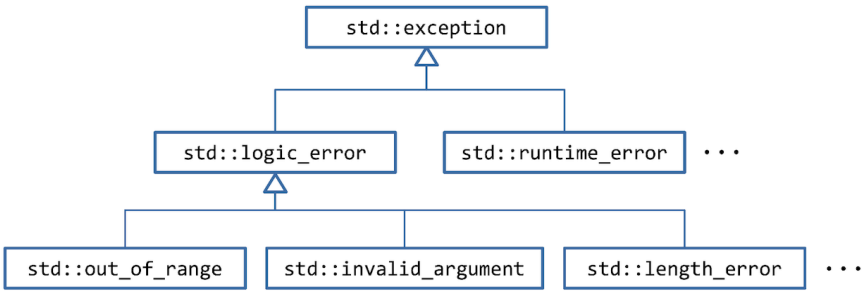
- Requires reference parameter
- (Bad!) Alternative: global variable (POSIX: errno)
- E.g: std::istreams's states (good(), fail()) is changed as a side-effect of input

6.2 Catching Exceptions

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

```
#include <stdexcept> // contains some sublcasses
try {
throw std::logic_error("message");
} 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 "reason" of type std::string. It provides the what() member function to obtain the "reason"



6.3 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.

7 Classes and Operators

Are defined in header files and not in *.cpp files! The implementation can then be done in a suitable file.

- Class members are implicitly inline.
- A class does one thing well and is named after that.
- A class consists of member functions with only a few lines.
- Has a class invariant: provides guarantee about its state (values of the member variables).
- Don't make member variables const as it prevents copy assignment. Don't add members to communicate between member function calls.
- Member functions should when possible be const, as long as they don't change the this object

```
class <GoodClassName> {
  <member variables>
  <constructor>
  <member function>
};
```

Class type in a header file.

```
#ifndef DATE_H_
#define DATE_H_
class Date {

  int year, month, day;

public:
  Date() = default;
  Date(int year, int month, int day) : year{year}, month{month}, day{day} {/*...*/}
  static bool isLeapYear(int year) {/*...*/}

private:
  bool isValidDate() const {/*...*/}
};

#endif /* DATE_H_ */
```

Implementation of the class.

```
#include "Date.h"
Date::Date(int year, int month, int day) : year{year}, month{month}, day{day} {
    if (!isValidDate()) {
      throw std::out_of_range{"invalid date"};
    }
  }

Date::Date() : Date{1980, 1, 1} { } // Default constructor

Date::Date(Date const & other) : Date{other.year, other.month, other.day} { } // copy
    constructor

bool Date::isLeapYear(int year) {
  /* ... */
}
```

7.1 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

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

7.2 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.

```
1  class Date {
2  public:
3      Date(int year, int month, int day);
4      Date(); // Default-Constructor
5      Date() = default; // explizit Default-Constructor
6      Date(Date const &); // Copy-Constructor
7      Date(Date &&); // Move-Constructor
8      explicit Date(std::string const &); // Typeconversation-Constructor
9      Date(std::initializer_list<Element> elements); // Initializer List-Constructor
10     ~Date(); // Destructor
11     Date(Date const &) = delete; // delete implicit Copy-Constructor
12 };
```

7.3 Defaulted Constructor

In order no to state the default constructor explicitly in the cpp file it can be defined in the header file of the class. This is also possible for the move and the copy constructor.

```
1 #ifndef DATE_H_
2 #define DATE_H_
3 class Date {
4   int year{9999}, month{12}, day{31};
5   //...
6   Date() = default;
7   Date(int year, int month, int day);
8 };
9 #endif /* DATE_H_ */
```

7.4 Inline Functions

Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small (All the functions defined inside the class are implicitly inline).

7.5 Friend Functions

A friend function can be given a special grant to access private and protected members.

7.6 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.

```
1  class Base {
2  private:
3    int onlyInBase;
4  protected:
5    int baseAndInSubclasses;
6  public:
7    int everyoneCanFiddleWithMe
8  };
9  class Sub : public Base {
10   //Can see baseAndInSubclasses and
11   //everyoneCanFiddleWithMe
12 };
```

8 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.

8.1 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)));
}
```

8.2 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"
#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)));
  }
};
```

8.3 Greater / smaller than / same Overload

```
// word.h
class Word {
private:
  std::string word;

public:
```

```
7   Word() = default;
8   Word(std::string word);
9
10  std::ostream & print(std::ostream & os) const;
11  std::istream & read(std::istream & is);
12
13  bool operator<(Word const & rhs) const;
14 };
15
16 inline std::ostream & operator<<(std::ostream & os, Word const & word) {
17   return word.print(os);
18 }
19
20 inline std::istream & operator>>(std::istream & is, Word & word) {
21   return word.read(is);
22 }
23
24 inline bool operator>(Word const & lhs, Word const & rhs) {
25   return rhs < lhs;
26 }
27
28 inline bool operator>=(Word const & lhs, Word const & rhs) {
29   return !(lhs < rhs);
30 }
31
32 inline bool operator<=(Word const & lhs, Word const & rhs) {
33   return !(rhs < lhs);
34 }
35
36 inline bool operator==(Word const & lhs, Word const & rhs) {
37   return !(lhs < rhs) && !(rhs < lhs);
38 }
39
40 inline bool operator!=(Word const & lhs, Word const & rhs) {
41   return !(lhs == rhs);
42 }
43
44 // word.cpp
45 bool Word::operator<(Word const & rhs) const {
46   return std::lexicographical_compare(word.begin(), word.end(), rhs.word.begin() ,rhs
       .word.end(), [](char lhs, char rhs) {
47     return std::tolower(lhs) < std::tolower(rhs);
48   });
49 }
```

9 Enums

- Enums can be used for types that hold a few values.
- Every enum field can be converted into an int, starting with the 0.
- The names of an Enum cant be given out as default. For this a lookuptable is needed.
- With the class keyword (Scoped Enum) the type of the enum is not visible outside the namespace. The normal unscoped Enum is visible outside the namespace.

```
1  enum [class] <name> {
2    <enumerators>
3  };
4
5  enum class day_of_week {
6    Mon, Tue, Wed, Thu, Fri, Sat, Sun // 0 - 6
7
8    day_of_week operator++ (day_of_week & aDay) {
9      int day = (aDay + 1) % (Sun + 1); // Convertion to int
10     aDay = static_cast<day_of_week>(day);
11     return aDay;
12   }
13 };
```

10 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 namespace` shouldn't be used.

```
1  namespace demo {
2  void foo(); //1
3  namespace subdemo {
4  void foo() {/*2*/}
5  } // subdemo
6  } // demo
7
8  namespace demo {
9  void bar() {
10   foo(); //1
11   subdemo::foo(); //2
12 }
13 }
14
15 void demo::foo() { /*1*/ } // definition
16
17 int main() {
18   using demo::subdemo::foo;
19   foo(); //2
20   demo::foo(); //1
21   demo::bar();
22 }
```

10.1 Using Declaration

- 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 "pollution" of your namespace.

10.2 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)

10.3 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.

10.4 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)

11 Container and Collections

Container Contains objects with value (vector, string, set, map)

Collection Contains objects by reference

- Container can be copied easily using the constructor (deep copy). `std::vector<int> vv{};`
- Container support the "clear()" function which empty the container.
- There are three types of containers: Sequence Containers, Associative Containers and Hashed Containers.
- With containers member function should be preferred over STL.
- Two containers with the same type can be compared `c1 == c2`

Containers can be: default-constructed, copy-constructed from another container of the same type, equality compared, emptied with clear().

- Sequence Containers (vector, deque, list, array)
  - Elements are accessible in order as they were inserted/created.
  - Find in linear time through the algorithm find.
- Associative Containers (set, multiset, map, multimap)
  - Elements are accessible in sorted order
  - find as member function in logarithmic time
- Hashed Containers (unorered_map, unordered_multimap, unordered_set, unorderd_multiset)
  - 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

Figure 3: Member Function of a Container

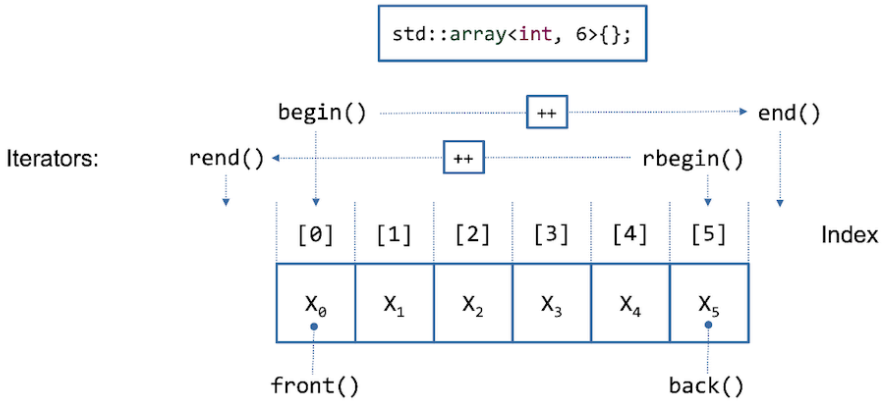
11.1 Common Container Constructors

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

11.2 Array

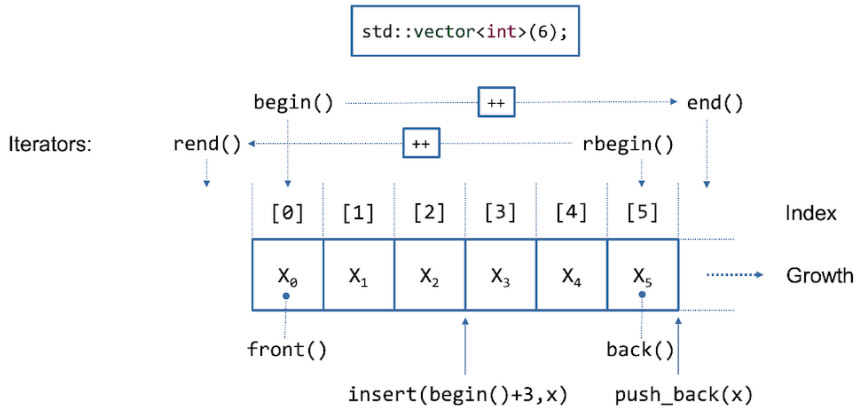
C++'s std::array<T, N> is a fixed-size Container. T is a template type parameter (= placeholder for type). N is a positive integer, template non-type parameter (= placeholder for a value). Elements can be accessed with a subscript operator [] or at(). The size is bound to the array object and can be 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;



11.3 Vector

- C++'s std::vector<T> is a Container = contains its elements of type T (no need to allocate them).
- The elements are allocated on the heap.
- If a vector is passed to a function we can prevent a copy when we pass it as const.



Append Elements to an std::vector<T>

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

Filling a Vector with Values

```
1  std::vecor<int> v{};
2  v.resize(10);
3  std::fill(std::begin(v), std::end(v), 2);
4
5  std::vector<int> v(10);
6  std::fill(std::begin(v), std::end(v), 2);
7
8  std::vector v(10, 2);
9
10 // Filling increased values with iota
11 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.

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

11.4 Double-Linked List

- Very efficient inserting at any position.
- Lower efficiency in bulk operations.
- Only bi-directional iterators - no index access

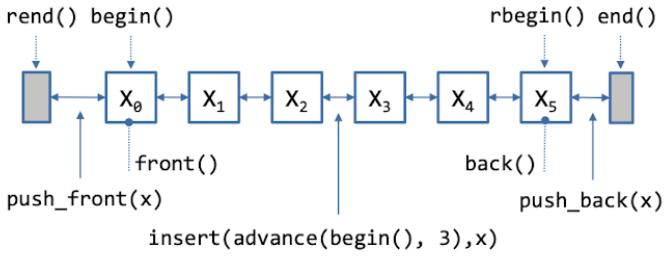


Figure 4: Double-Linked List

```
1 std::list<int> l = { 7, 5, 16, 8 };
2 l.push_front(25);
3 l.push_back(13);
4 // Insert an integer before 16 by searching
5 auto it = std::find(l.begin(), l.end(), 16);
6 if (it != l.end()) {
7   l.insert(it, 42);
8 }
```

11.5 Double-ended Queue, Deque

Are like a vector, but additionally elements can be added efficiently to the start of the container.

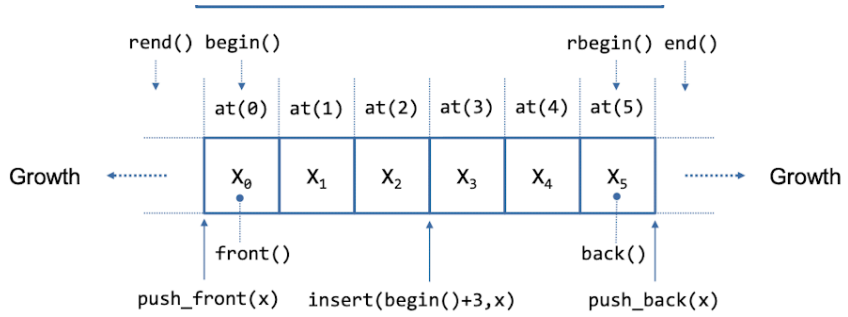


Figure 5: Deque

11.6 Queue, FIFO Adapter

In contrast to the Stack takes "pop()" the element from the begin of the Queue.

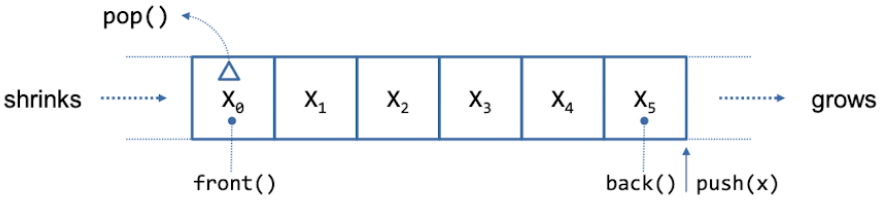


Figure 6: Queue

```
std::queue<int> q{};
q.push(42);
std::cout << q.front();
q.pop();
```

11.7 Stack, LIFO Adapter

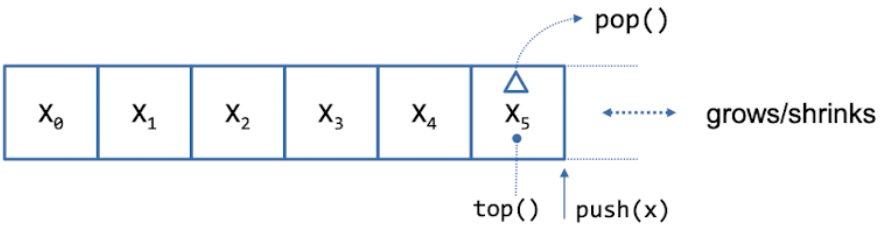


Figure 7: Stack

11.8 Set

The set does save all the elements in a tree. As a result there are no duplicates and all the elements are sorted automatically.

```
#include <set>
std::set<int> s {7,1,4,3,2,5,6};

#include <string>
#include <algorithm> -> transform
#include <iostream> -> cout
#include <iterator> -> ostream_iterator
#include <cctype> -> lowercase

// insert
std::string const input{"test string"};
std::set<char> myset { };
std::transform(input.begin(), input.end(), inserter(myset, myset.begin()), [](char c)
    {
  return tolower(c);
});

// print
std::ostream_iterator<char> out {std::cout}
std::copy(myset.begin(), myset.end(), out};
```

11.9 Multiset

In contrast to the set does the multiset allow duplicates.

```
#include <iostream>
#include <iterator>
#include <string>
#include <set>

using in=std::istream_iterator<std::string>;
using out=std::ostream_iterator<std::string>;
std::multiset<std::string> words{in{std::cin},in{}};
copy(cbegin(words), cend(words), out(std::cout, "\n"));
```

11.10 Map

In a map key-value pairs are stored, where the value can be in the set multiple times but the key is unique. Also the keys are stored in ascending order. The over can be overwritten with a 3rd template parameter.

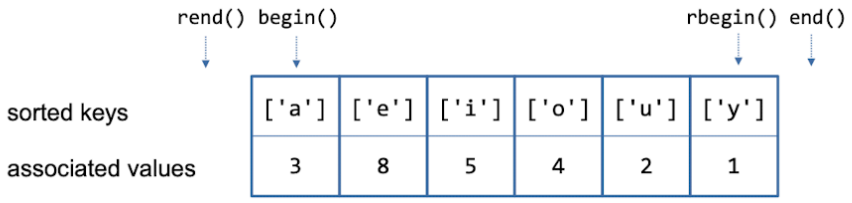


Figure 8: Map

```
1  std::map<char, size_t> vowels
2  {{'a',0},{'e',0},{'i',0},{'o',0},{'u',0},{'y',0}};
3
4  // Increment Value of Key
5  ++vowels['a'];
6
7  // Beim Iterieren ist jedes Element ein pair<char, size>
8  for(auto const & p:vowels) {
9    std::cout << p.first << " = "<< p.second << "\n";
10 }
```

11.11 Multimap

Allows to have multiple keys.

12 STL Algorithms

The "algorithm.h" are the algorithms defined for general purpose. and in the "numeric.h" are 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.

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

12.1 Examples

12.2 Pitfalls

13 Function Templates

- Can be compared as a Generic in Java. The keyword "template" is used to declare a template.
- The template parameter list contains one or more templates parameters.
- C++ uses duck-typing. So every type can be used as argument as long as it supports the used operations.
- Function templates are normally defined and implemented in a header file.
- Tempalate functions are implicitly inline
- We can write generics with templates.

The compiler resolves the function template and figures out the template arguments.

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

```
1  // file min.h
2  template <typename T>
3  T min(T left, T right) {
4    return left < right ? left : right;
5  }
6  // file smaller.cpp
7  #include "min.h"
8  #include <iostream>
9  int main() {
10   int first;
11   int second;
12   if (std::cin >> first >> second) {
13     auto const smaller = min(first, second); std::cout << "Smaller of " << first << "
       and " << second << " is: " << smaller << '\n';
14   }
15 }
```

Template Argument Deduction

13.1 Variadic Templates

- For function templates with an arbitrary number of parameters
- Needs at least one pack parameter
- Pack Expansion: For each argument in that pack an instance of the pattern is created
- In an instance of the pattern the parameter pack name is replaced by an argument of the pack
- Needs a base case for the recursion (after the last parameter is done, it would call the function without a parameter, which is invalid) → Base case must be written before the template function.

```cpp
#include <iostream>
#include <string>

// Base Case
void printAll(std::ostream & out) {
}

template<typename First, typename...Types>
void printAll(std::ostream & out, First const & first, Types const &...rest) {
  out << first;
  if (sizeof...(Types)) {
    out << ", ";
  }
  printAll(out, rest...);
}

int main() {
  int i{42}; double d{1.25}; std::string book{"Lucid C++"};
  printAll(std::cout, i, d, book);
}
```

14 Class Templates

- In addition to functions also class types can habe template parameters
- Since C++17, similar to function templates, the compiler might deduce the template arguments
- Class templates deliver types with compile time parameters
- Member function which never are used are also never compiled
- Compile-time polymorphism
- Class templates can be specialized

Rules

- Define class templates completely in header files!
- Member functions of class templates
  - Either in class template directly
  - Or as inline function templates in the same header file
- When using language elements depending directly or indirectly on a template parameter, you must specify typename when it is naming a type.
- static member variables of a template class can be defined in header without violating ODR, even if included in several compilation units.

```
template <TemplateParameters>
class TemplateName { /*...*/ };

template <typename T>
class Stack { /*...*/ };
```

14.1 Template Argument Deduction (C++17)

Similar to function templates, the compiler might deduce the template arguments. This is a compile-time polmorphism.

```
std::vector newValues{1, 2, 3}; // The compiler can deduce the type
std::vector<int> emptyValues{};
```

14.2 Type Alias & Dependent Names

- It is common for template definitions to define type aliases in order to ease their use.
- Within the template definition you might use names that are directly or indirectly depending on the template parameter.
- Dependent Name: Compiler geht standardmässig davon aus, dass es sich im eine Variable, oder eine Funktion handelt. Wenn es ein Typ ist (wie size_type), muss das keyword typename verwendet werden.

Example

```
template <typename T> // Class template with one typename par
class Sack {
  using SackType = std::vector<T>;
  using size_type = typename SackType::size_type; // dependent name
  SackType theSack{};

public:
  bool empty() const {
    return theSack.empty();
  }
  size_type size() const {
    return theSack.size();
  }
  void putInto(T const & item) {
```

```
15      theSack.push_back(item);
16    }
17    T getOut(); // member forward declaration
18  };
```

Define the function outside of the template class definition.

```
1  template <typename T>
2  inline T Sack<T>::getOut() { // implementation outside of class
3    if (empty()) {
4      throw std::logic_error{"Empty Sack"};
5    }
6    auto index = static_cast<size_type>(rand() % size());
7    T retval{theSack.at(index)};
8    theSack.erase(theSack.begin() + index);
9    return retval;
10 }
```

14.3 Inheritance

Rule: Always use this->variable (or className::) to refer to inherited members in a template class.

15 Dynamic Heap Memory Management

Dont do this yourself! Always rely on library classes for managing it.

15.1 When ist Heap Memory used?

- Stack memory is scarce
- It might be needed for creating object structures.
- Also needed for polymorphic factory functions to class hierarchies.
- Resource Acquisition Initialization (RAII) Idiom

15.2 Legacy Heap Memory

Dont use this!
C++ allows allocating objects on the heap directly. If done manually, you are responsible for deallocation and risk undefined behaviour.

```
1 // dont use new / delete
2 auto pr = new int{};
3 std::cout << *ptr << '\n';
4 delete ptr;
```

15.3 Modern Heap Management

In the modern C++ world we can use smart pointers, which are C++ templates, to make memory management easier. With these smart pointers we dont have to call "delete ptr;" by ourselfs. Still: always prefer storing the value locally as value-type variable (Stack-based or member).

- Delete Pointer - must never be called.
- Unique Pointer - for unshared Heap Memory (cant be copied).
- Shared Pointer - for shared Heap Memory (work as Java references, can be copied and moved).
- If the last "shared_ptr" handle gets destroyed, the allocated object gets deleted.
- Shared Pointer have the problem of cycles. For this reason there is a "weak_ptr" to break the cycles.

15.3.1 std::unique_ptr<T>

- defined in "<memory>"
- Used for unshared heap memory
  - Or for local stuff that must be on the heap
  - Can be returned from a factory function
- Only a single owner exists
- Not the best for class hierarchies
- Can not be copied

Use Cases

- As member variable
- As local variable

15.3.2 Shared Pointer

- Works more like a Java reference and allows multiple owners.
- The pointer is `std::shared_ptr` and associates objects of Type T using `std::makeshared<T>()`.

```
1 struct Article { Article(std::string title, std::string content);
2   //..
3 };
4 Article cppExam{"How to pass CPl?", "In order to pass the C++ exam, you have to..."};
5 std::shared_ptr<Article> abcPtr = std::make_shared<Article>("Alphabet", "
    ABCDEFGHIJKLMNOPQRSTUVXYZ");
```

Use Cases

- If you really need heap-allocated objects, because you create your own object networks
- If you need to support run-time polymorphic container contents or class members that can not be passed as reference, e.g., because of lifetime issues
- Factory functions returning std::shared_ptr for heap allocated objects.
- But first check if alternatives are viable:
  - (const) references as parameter types or class members
  - Plain member objects or containers with plain class instances

The usage is counted on the referenced object to keep track of how many reference currently point to this object on the heap.

15.3.3 std::weak_ptr<T>

- The "shared_ptr" cycles need to be broken
- "weak_ptr" does not allow direct access to the object
- A "weak_ptr" does not know weather the pointee is still alive
- with "lock()" to the object can be acquired if alive.

Figure 9:

16 Inheritance

Inheritance is always then used, when specific components want to be reused and extended. Inheritance can be bad because it creates a very strong dependency.

- Inheritance is default public (Classes). For structs the inheritance is private.
- The constructors are not inherited implicitly we have to specify that.
- The parent is always constructed first and after that the children.
- Assining or send parameters per value from an inherited class to the base class result in **Object Slicing**.

```
class MyClass : Base {}; // implicit private
struct MyStruct : Base {}; // implicit public

class MyClass : public Base {
  public:
     using Base::Base; // inherit constructor
};
```

16.1 Initialising Multiple Base Classes

Base constructors can be explicitly called in the member initializer list. You should put base class constructor class before the initialization of members. The compiler enforces this rule, even though you can put the list of initializers in wrong order.

```
class DerivedWithCtor : public Base1, public Base2 {
  int mvar;
public:
  // calls base1, base2, mvar
  DerivedWithCtor(int i, int j) : Base1{i}, Base2{}, mvar{j} {}
};
```

16.2 Dynamic Polymorphism

- Operator and function overloading and templates allow polymorphic behaviour at compile time
- Dynamic polymorphism needs object references or (smart) pointers to work
  - Syntax overhead
  - The base class must have a good abstratction
  - Copying carries the danger of slicing (partial copying)

16.2.1 Shadowing Member Functions

- if a function is reimplemented in a derived class, it shadows its counterpart in the base class
- However, if accessed through a declared bases object, the shadowing function is ignored

```
struct Base {
  // shadowed function
  void sayHello() const {
    "Im Base\n"
  }
}
struct Derived : Base {
  // shadowing function
  void sayHello() const {
    std::cout << "hi, im derived\n";
  }
};
void greet(Base const & base) {
  base.sayHello();
}
in main() {
  Derived derived{};
  greet(derived); // Hi, im Base (static call)
}
```

16.3 Virtual Member Functions

- To achieve dynamic polymorphism "virtual" member functions are required
- "Virtual" member functions are bound dynamically.
- The virtual keyword ist automatically inherited and does not have to be restated at childs.
- Its possible to state overriding functions with "override"
- To override a virtual function the signatures habe to be the same!

```
struct Base {
  virtual void sayHello() const {
    std::cout << "Hi, I'm Base\n";
  }
};

struct Derived : Base {
  void sayHello() const { // virtual is automatically inhertited
    std::cout << "Hi, I'm Derived\n";
  }
};
void greet(Base const & base) {
  base.sayHello();
}

int main() {
  Derived derived{};
  greet(derived); // Hi, I'm Derived (dynamic call)
}
```

16.4 Calling Virtual Member Functions

- Value Object
  - Class type determines function, regardless of virtual
- Reference
  - Virtual member of derived class called through base class reference
- Smart Pointer
  - Virtual member of derived class called through smart pointer to base class
- Dump Poiner (rarely used)
  - Virtual member of derived class called through base class pointer

```
void greet(Base base) {
  base.sayHello(); // Value: always calls base
}

void greet(Base & base) {
  base.sayHello(); // Reference: dynamic binding
}

void greet(std::unique_ptr<Base> base) {
  base.sayHello(); // dynamic binding
}

void greet(Base const * base) {
  base->sayHello(); // dynamic binding
}
```

16.5 Abstract Base Classes

- There are no Interfaces in C++
- A pure virtual member function makes a class abstract
- To mark a virtual member function as pure virtual it has zero assigned after its signature
- Abstract classes cannot be instantiated (like in Java)

```
struct abstractBase {
  virtual void doitnow() = 0;
}
```

16.6 Destructors

- Classes with virtual members require a virtual Destructor
- Otherwise when allocated on the heap with make_unique and assigned to a unique_ptr only the destructor of Base is called

```
struct Fuel {
  virtual void burn() = 0;
  virtual ~Fuel() { std::cout << "put into trash\n" }
};

struct Plutonium : Fuel {
  void burn() { std::cout << "split core\n"; }
  ~Plutonium() { std::cout << "store many years\n"; }
};

int main() {
  std::unique_ptr<Fuel> surprise = std::make_unique<Plutonium>(); // both called
}
```

16.7 Object Slicing

The object slicing problem can be solved if we set the copy operations as delted.

```
struct Base {
  Base & operator=(Base const & other) = deleted;
  Book(Book const & other) = deleted;
}
```

17 Initialisation

- Default Initialization
- Value Initializtaiton
- Direct Initialization
- Copy Initialization
- List Initialization
- Aggregate Initialization

17.1 Default Initialization

Is the simplest type of initialization. We simply don't provide an initializer. This depends on the kind of entity we want to declare. This does not work for references! Also does not rally work with a const object.

```
int global_variable; // implicitly static
void di_function() {
  static long local_static; // Default I
  long local_variable; // Default I
}
struct di_class {
  di_class() = default; // Default Initializer
  char member_variable; // not in ctor init list
};
```

Effects

- Static Variables are Zero initialized first, then their types default constructor is called.
- Non static integer and floating point variable are unitialized!
- Object of class types are constructed using their default constructor.
  - Member Variables not in a ctor-init-list are default initialized
- Arrays initialized all of their elements accordingly

```
struct blob {
  blob(int); // suppreses default constructor
};
blob static_instance; // error no matching function blob::blob();!
```

```
void di_function() {
  long local_variable; // no initialization
  std::string local_text; // gets default constructed, string default constructor
}
struct di_class {
  di_class() = default;
  char member_variable; // default initiliazed, since value type the content is
    rubbish
};
```

17.2 Value Initialization

Value Initialization is perfomed with empty `()`, `{}` but curly bracets are prefered, since it works with more classes. Invokes the default constructor for class types.

```
#include <string> #include <vector>
void vi_function() {
  int number { };
  std::vector<int> data { };
  std::string actually_a_function(); // Is a function and not a variable!!
}
```

17.3 Direct Initialization

Nearly the same as Value initialization but we directly define the value. Using `{}` only applies to non class types.

```
#include <string>
void diri_function() {
  int number{32}; // DI
  std::string text("CPL"); // DI
  word vexing (std::string()); // Most Vexing Parse
}
```

Most Vexing Parse There are two interpretations of this expression: Initialization with a value-initialiized string, Decleration of a function returnin g a word and taking an unnamed pointer to a function returning a string.

17.4 Copy Initialization

Initialization using `=`.
Pseudocode for behavior of copy initialization.

```
if(object has type class, rhs has the same type)
  if(rhs is temporary) object is constructed in place
  else( copy constructor is invoked )
else(Suitable conversion sequence is searched for)
```

```
#include <string>
std::string string_factory() { return ""; }

void ci_function() {
  std::string in_place = string_factory(); // object in place
  std::string copy = in_place; // copy constructor
  std::string converted = "CPl"; // converted
}
```

17.5 List Initialization

Uses the non empty `{}`. If there is a suitable constructor takin `std::initializer_list` is selected. Otherwise a suitable constructor is searched.

```
// Direct List Initialization
std::string direct { "CPI" };
// Copy List Initialization
std:: string copy = { "CPIA" };
```

Pittfall Since the `std::itializer_list` constructor is preferred, you might run into trouble.

```
int ouch() {
  std::vector<int> data {10, 42}; // creates with initializer list.
  return data[5]; // out of bound!
}
```

18 Aggregates

Is a simple class type.

- Can have other types as public class types
- Can have member variables and functions
- Must not have user-provided, inherited or explicit constructors
- must not have protected or private direct members

```
1  struct person {
2    std::string name;
3    int age{42};
4
5    bool operator<(person const & other) const {
6      return age < other.age;
7    }
8
9    void write(std::ostream & out) const {
10     out << name << ": " << age << "\n";
11   }
12 };
13
14 int main() {
15   person rudolf{"Rudolf", 32};
16   rudolf.write(std::cout);
17 }
```

18.1 Aggregate Initialization

Conecptional a special case of the list initialization. If the type is an aggregate, the members and base classes are initialized from the initializers in the list.

```
1 person rudolf{"Rudolf"};
```