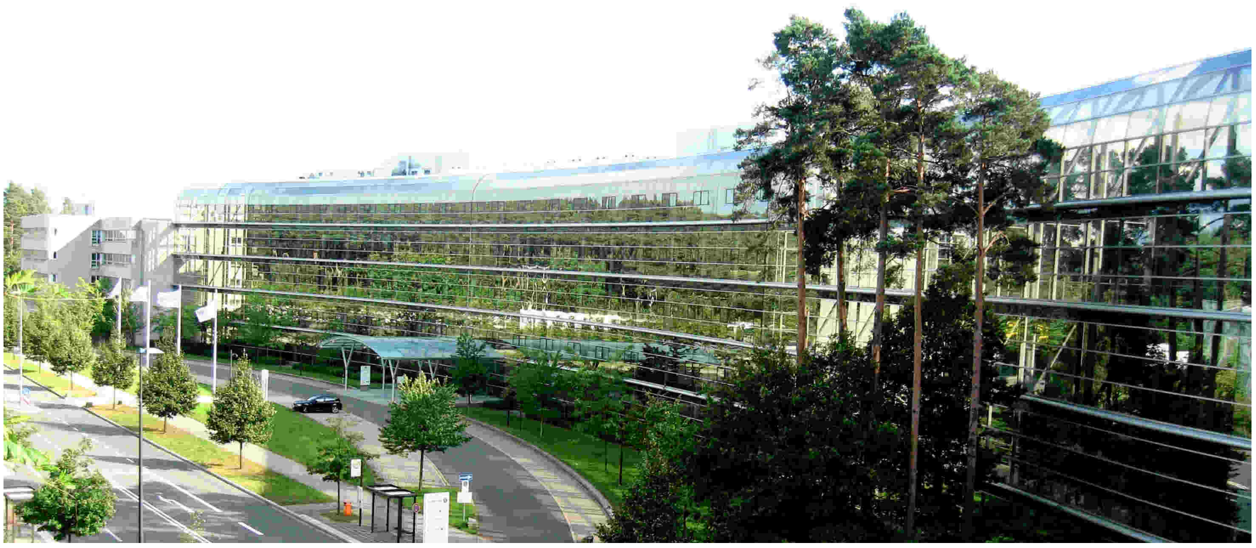
C++ - Seminar

for C-Programmers

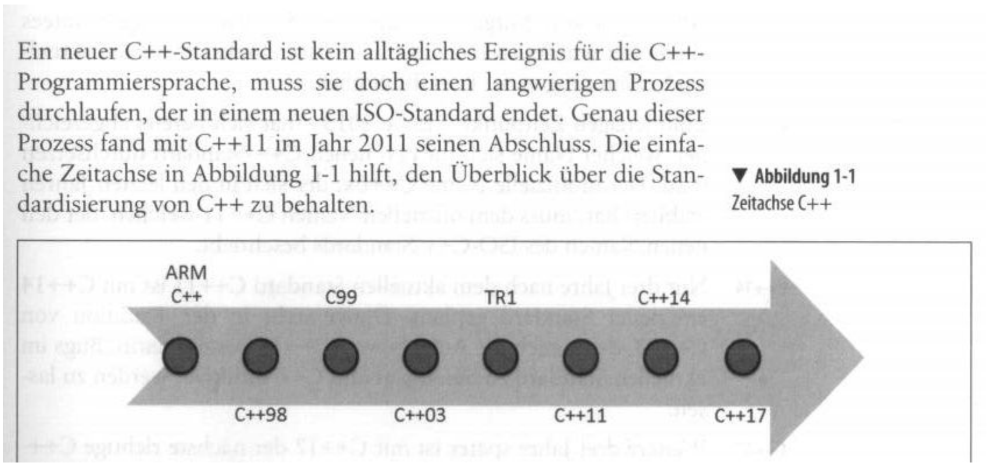
Dr.sc.nat. Michael J.M. Wagner, New Elements*

Revision 265



*michael@wagnertech.de

Contents


Ein neuer C++-Standard ist kein alltägliches Ereignis für die C++-Programmiersprache, muss sie doch einen langwierigen Prozess durchlaufen, der in einem neuen ISO-Standard endet. Genau dieser Prozess fand mit C++11 im Jahr 2011 seinen Abschluss. Die einfache Zeitachse in Abbildung 1-1 hilft, den Überblick über die Standardisierung von C++ zu behalten.

▼ **Abbildung 1-1**
Zeitachse C++

Figure 1: C++ history[2]

1 Introduction to C++[1]

The ancestors of C++ are Smalltalk and C. Smalltalk is the first object-oriented computer language. C is *the* language for system programming. Therefore C++ is an object-oriented programming language for system programming.

C++ started in the late 70-ies. The object-oriented hype grew up and therefore everything had to be object-oriented. In 1979 the first title was "C with classes". In 1983 the template mechanism was added and the language was called C++.

In the object-oriented hype it was thought, adding object-orientation to a computer language is enough for building reusable and maintainable software. New useful classes are developed and shared. But what happened: Useful classes were developed a few times solving the same problem. So in the 90-ies we had many implementation for classes e.g. handling text strings. Each compiler publisher shipped his own string class. And these string classes were not compatible. Therefore the process of standardization was started. The steps of standardization are shown in figure 1:

- *Annotated Reference Manual* with the drafts of the beginning standardization
- 1998: Definition of ANSI-C++

  This standard contains the important library classes. Most of them generic template classes. Therefore this standard often is called STL (*standard template library*)
- 2003: Improvements to C++98

  Most important: The `auto_ptr` class, the first helper class for a safe handling of heap memory.
- 2011: C++11

  In 2011 there was no object-oriented hype any longer. With functional programming a new paradigm entered the C++ programming language. What is functional programming? Functional programming avoids loops. If an algorithm has to be executed on a bunch of

[1]Wolf: Kap. 1
[2]Grimm: S. 3

data, you define the algorithm and tell the compiler to do this algorithm on that data. This approach enables the compiler to parallelize the execution. For a simple handling of algorithms the *lambda function* was included into C++. A lambda function is a adhoc definition of a function.

- 2014: C++14: Improvements to C++11

  The lambda function obtains a few additional features. In combination with the `auto` key word generic lambda functions can be written.

- 2017: C++17: Improvements to C++11

  Standardization of parallel algorithms

Objective of C++11:[3]

- For beginners: Simpler to learn
- For professionals: A better programming language for system programming
- Multi paradigm language
  - procedural, structured (C)
  - object oriented, generic (C++98)
  - functional: Avoidance of loops, functions as parameters. Enables the compiler to implicit parallization.

This Handout

This handout contains links to online resources and can only be used with these resources.

First C++ program

First C++ program[4]

Exercise:

Create a new *project* in your development tool and run a simple "Hello World" program.

[3]Grimm: p. 7.
[4]http://www.cplusplus.com/doc/tutorial/program_structure/

2 Elements of the C++ Programming Language

2.1 Language Expansion Compared to C

C++ is based on C. Therefore the complete C language syntax con be used in C++. With C++98 there are the following expansions:

- Streams for input/output (Sec. 3)
- Line comments: `//`
- References (Kap. 2.3)
- Default-Arguments (Kap. 2.4)
- Function overloading (Kap. 2.4)
- Language elements for object orientation (Sec. 5)
- Name spaces (Kap. 2.5)
- Operator overloading (Kap. 5.2)
- Templates (Kap. 6)
- Exceptions (Kap. 4)

With C++11 the language kernel is supplemented by:

- Uniform initialization
- Unicode support
- New semantics for keyword `auto`
- Range for: `for (elem : collection)`
- `enum class`
- Type alias with `using`
- Lambda function
- Move semantics
- Variadic templates
- `constexpr`

For the definition of constanst there are now three possibilities:

- `#define` directive: From C, replacement at compile time, not type save
- `const`: C++98, „real" variable, type save, can be overriden
- `constexpr`: C++11, replacement at compile time, type save

2.2 Arrays and Strings[5]

In C dealing with arrays and strings is not comfortable. With C++98 there are classes in the standard library that ease the handling of arrays and strings:

- Arrays: `std::vector<TYP>` [`006/listing001.cpp`, W:121]

  Element access by brackets without range check. To obtain a range check use `my_array.at(<position>)`.

- Strings: `std::string` [`006/listing005.cpp`, W:132]

For strings there are only few functions. In principle we have to distinguish two notations:

- Procedural notation: The string is a parameter of the function call: `f(string_var)`
- Object oriented notation: The name of the string is put with a dot*in front of* the function call: `string_var.empty()` (returns a boolean value, whether the string is empty)

The `std::string` implementation with C++98 case during the object oriented hype. Therefore nearly all functions for strings are in object oriented notation. Properties of strings[6]

Exercise:

With the following exercises a library management system is built up step by step. Here the first steps:

- Create a new project *Bucherei*.
- Modify in `Bucherei.cpp` the example `006/listing005.cpp` in that way, that 6 values are requested from the user (*signature, author, title, type, page number, duration*) and these 6 values are stored in variables. *signature, author, title* are strings, *type* is a single character, *page number* and *duration* are integers.

  Remark: In C++ variable names of simple variables are written in lowercase. The names may contain an underscore.
- An input check is not done.
- Output the variables in a formatted way.

2.3 References and Pointer[7]

Pointer and references are variables the reference/point to another variable. There are big differences in the syntax and in the usage. The most important difference is that a pointer can point to *nothing* (`NULL - nullptr`), a reference never.

When accessing the value pointers have to be dereferenced (`* ->`), a reference is used like the variable itself. A reference is like an alias.

[5]Wolf: Kap. 6
[6]http://www.cplusplus.com/reference/string/string/
[7]Wolf: Kap. 7

Pointer: [007/listing006.cpp, W:147]

References: [007/listing001.cpp, W:138f]

2.4 Functions[8]

Passing Parameters

Functions may have parameters. Per default there is a *call by value* call, i.e. the called function obtains a copy of the value. Changes of the value inside the callee have no impact to the caller. [008/listing003.cpp, W:154f]

Is a function called with a reference to a variable, these has two consequences:

- The variable is not copied into the callee.
- Changes inside the callee have impact to the caller.

Function definition with a reference[9]:

```
void add(int& val1, int val2) {
  // This function adds val2 to val1
  val1 += val2;
}
```

Is the first point regarded as a advantage, because the variable has a big size, but there is no interrest in an impact to the caller, a constant reference can be used.

Function definition with constant reference[10]:

```
void add(const int& val1, const int& val2) {
  // This function uses references to the variables
  // of the callee. It is not allowd to alter the values.
  std::cout << "The sum is: " << val1+val2 << '\n';
}
```

Functions with default argument: `void addieren(int val1 = 12, ...` [008/listing004.cpp]

Functions with return value: [008/listing005.cpp]

Exercice:

Divide the library example into functions and put them in different files:

- File `InOut.cpp/.h` with functions

```
void readMedium(std::string& signatur, std::string& autor,
  std::string& titel, char& typ, int& seitenzahl, int& spieldauer);
void writeMedium(const std::string& signatur,
  const std::string& autor, const std::string& titel,
  char typ, int seitenzahl, int spieldauer);
```

[8]Wolf: Kap. 8
[9]s. auch Wolf: Kap. 8.2.1
[10]s. auch Wolf: Kap. 8.4

- File with main routine:

  Define the six variables, supply values by calling `readMedium()`, and output them by `writeMedium()`.

Function Overloading

In C++ there can be functions having the same name, but different parameters:

```
int calculate ( int ivar );
int calculate ( int ivar1 , int ivar2 );
double calculate ( double dvar ) ;
double calculate ( double dvar1 , double dvar2 );
```

If the compiler does not find a fitting function, it trys implicit type conversion.

Exercise 8.6[11]

Supplement to 8.6.5:

- Write the function for area calculation in three variants:
  - having two integer input parameter and an integer return value
  - having two `double` input parameter and a `double` return value
  - having two `double` input parameter and a `double` output parameter

  These three functions have the same name.

2.5 Namespaces[12]

Pursuing the procedural programming paradigma, it is necessary to build namespaces. These enables to use the same function names in different contexts. In the 90'ies only object oriented language constructs were used (s. sec. 5). Simple procedural functions inside class definitions are called `static` elements:

```
class MyClass {
public:
  static int foo() {
    return 47;
  }
}

int main() {
  int result = MyClass::foo();
}
```

[11]Wolf: S. 178ff.
[12]Wolf: Kap. 9.2

Lateron a concept for namespaces was added.

The use of namespaces requires the following:

- Declaration of functions/classes within a `namespace` block.

  ```
  namespace VIP_Area {
    void function();
    void function2();
    int ival;
    float fval;
  }
  ```

- Definition of the functions in the `cpp` file can be performed in two ways:

  - inside a `namespace` block:

    ```
    namespace VIP_Area {
      void function() {
        // body of function
      }
    }
    ```

  - with `::` prefix at the function name:

    ```
    void VIP_Area::function() {
      // body of function
    }
    ```

- If a function of a namespace is used, it has to be specified explicitly:

  ```
  VIP_Area::function(); // call of function()
  ```

- A namespace can also be imported completely. This should only be done in source files. In header files there may be side effects.

  ```
  using namespace VIP_Area;
  function(); // call of function()
  ```

- Alternativly there can also only single elements of a namespace be imported:

  ```
  namespace VIP_Area {
  void function();
  void function2();
  // ...
  }
  // import of function() - without parantheses!
  using VIP_Area::funktion;
  ...
  function(); // Ok
  function2(); // Error! function2() is not imported.
  ```

Exercise:

Define a namespace for the functions `readMedium()` and `writeMedium()`:

- In `InOut.h` the declations are put inside a name space definition (`namespace InOut {`).

- In `InOut.cpp` the implementation must be prepended by a `InOut::`.
- When calling a function in the main routine also a `InOut::` has to be prepended.
- In the main routine import the namespace `std` completly and add an output at the end of the program.

3 File Input/Output

For reading and writing files there are the classes `std::ifstream` and `std::ofstream`. These classes support the following functionality:

- Open a file

  ```
  #include <fstream>
  std::ofstream file01("testfile001.dat");
  std::ifstream file02("testfile002.dat");
  if ( ! file02 ) {
    std::cerr << "File does not exist!" << endl;
  }
  std::ifstream file03;
  file03.open("testfile003.dat");
  if (file03.fail()) {
    std::cerr << "File does not exist!" << endl;
  }
  // testfile004.dat is appended
  std::ofstream file04("testfile004.dat", std::ios::app);
  ```

- Close a file: `data01.close();`
  In general it is not necessary to close files, because files are closed automatically at the end of a block.
- Read/write line by line

  ```
  #include <fstream>
  using namespace std;
  ifstream rStream("datei.txt");
  ofstream wStream("out.txt");
  string line;
  while (getline(rStream, line)) {
    wStream << line << endl;
  }
  ```

- For block-wise reading/writing there are the stream methods `read` and `write`.[13]

[13]Wolf: Sec. 17.3.6

Exercise:

Supplement the library application:

- Create the files `MediaManagement.h` and `MediaManagement.cpp`.
- Write a function `addMedium` with the following signature:
  ```
  int addMedium(const std::string& signature, const std::string& author,
    const std::string& title, char type, int pages, int duration)
  ```
  The declaration is put to the header, the definition of the function in the source file.
- Don't forget to include the headerfiles in the source files.
- `addMedium` appends the data comma-separated to the file `media.csv`.
- Call this function from the main program with values from `readMedium()` (if already implemented).

When reading a comma-separated file line by line, each line has to be split into its parts. These parts have to be stored in an appropriate variable. Up to now C++ has no comfortable means to do that. In the files `util.h/.cpp` some helper routines are provided:

```
vector<string> tokenize(const string& line, char c);
```

This function converts `line` into a `stringstream` and calls the next function to do the work. A `stringstream` is an in memory *stream* that can be used for reading as well as for writing.

```
vector<string> tokenize(stringstream& ss, char token);
```

This function reads again and again from the `stringstream` to the next occurrence of `token` or tho the next line feed and puts the parts read in to a `vector` (array) of strings.

```
char toChar(const string& str);
```

Converts a string into a single character by writing the string into a `stringstream` and reading back the first character.

```
int toInt(const string& str);
```

Converts a string into an integer by writing the string into a `stringstream` and reading back all digits from the *stream* into an integer variable.

Exercise:

Put the files `util.h/cpp` into your project and compile it.

4 Error Handling

For error handling there are various approaches:

- Return value of type `int`
- Return value of a dedicated error type
- Exceptions
- Routines to evaluate an error state

Recommendation:

- Expected (often application) errors are handled by return values. In this case the error can directly by handled on the calling location.
- Unexpected (often technical or logical) errors are handled by exceptions. The program flow is interrupted and handled to an error handler, typically located "on top" of the calling hierarchy.

In general in C++ "everything" can by thrown as exception. With C++98 standard exceptions were defined, that are recommended for use.

Exceptions[14]

Throwing a runtime exception in user code:
`throw runtime_error("anything happened");`

To catch all exceptions in a cascaded way, at first the most specialized are caught, at last the rest by `catch (...)`:

```
#include <stdexcept>
try {
  // here are calls that might produce exceptions
}
catch (std::runtime_error& e) {
  cout << "Runtime Error: " << e.what() << std::endl;
}
catch (std::exception& e) {
  cout << "Problem: " << e.what() << std::endl;
}
catch (...) {
  cout << "Unknown exception." << std::endl;
}
```

Exercise:

Write a function in `MediaManagement`
`bool isSignatureInFile(const std::string& signatur)`, that

- checks, if the file `media.csv` can be opened for reading. If the file is not present, the signature is allowed to be put into a new file. Therefore do a `return false;`

[14]http://www.cplusplus.com/doc/tutorial/exceptions/

- Read the file line by line.
- Check, if each line has 6 tokens (`tokens.size()`). If not, throw an exception.
- Split off the signature and compare it with the given parameter.
- If the signature is already present, return `true`.
- Can the signature not be found until end of file, return `false`.

Complete the function `addMedium` with return values. Create in the header file corresponding integer constants:

```
constexpr int RC_OK = 0;
constexpr int RC_DUPLICATE = 1;
```

- Call the function `isSignatureInFile`. React on a duplicate error by returning the corresponding error code.
- Evaluate the return code in the main program and catch the exceptions.

5 Object Orientation

The mere procedural programming paradigm gets to its limits in big projects. One problem are name conflicts in the global name space. Beside the concept of name spaces class definitions were used to structure the code. Another problem is the responsibility to data in data structures that are passed through the whole system.

These experiences have led to object orientation:

- Data structures are useful for managing data that belongs together.
- In big projects it becomes unclear which data element is dealt with by which part of the program and who is allowed to perform changes in data.
- It is not clear from which moment on which data field is valid.

This has led to the idea of controlling the access to data structures (reading and writing). Direct access is prohibited. There are functions which perform the data access for all users. A data structure together with its access routines is called *class*.

Instantiating a class means giving memory to the data structure. Only an instantiated structure (= object instance) can be used.

In contrast to structures, in classes the data elements cannot be accessed directly. All access is done by public *methods*.

On the other hand classes are (mis-)used as containers to embrace functions that are linked logically. This kind of a class is a functional module. Functions that do not deal with the internal data structure are called `static`.

Real existing classes can be found anywhere in the spectrum between "functional module" and "data manager".

5.1 Classes

Definition of classes[15]

Typically the first part (class definition, `class NAME { ... };` is put into a header file named `NAME.h`, the definition of the methods (here: `void Rectangle::set_values ...` is put to `NAME.cpp`. A caller has to include `NAME.h`.

Class methods that access internal data can emphasis this by using the `this->`-pointer:

```
void Rectangle::set_values (int x, int y) {
  this->width = x;
  this->height = y;
}
```

Read only access[16] → *Const member functions*

Instancing an instance on the stack:

```
int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Instantiation on heap[17] → *Pointers to classes*

To overcome the *create - throw exception - memory leak* - problem smart pointers can be used. Since 2003 there is the `auto_ptr`, replaced by the `unique_ptr` with C++11.

```
#include <iostream>
#include <memory>

int main () {
  std::unique_ptr<int> apointer (new int);

  *apointer=10;

  return 0;
} // object is deleted automatically with the end of the block
```

Properties of `unique_ptr`[18]

[15]http://www.cplusplus.com/doc/tutorial/classes/
[16]http://www.cplusplus.com/doc/tutorial/templates/
[17]http://www.cplusplus.com/doc/tutorial/classes/
[18]http://www.cplusplus.com/reference/memory/unique_ptr/

Constructors and Destructors

Constructors[19] → *Constructors, Overloading constructors, Member initialization in constructors*

Empty function bodies (`{ }`) are simply implemented *inline* in the header file.

Special members[20]

The big five:

- Copy constructor: `Class(const Class&)`
- Copy assignment: `Class& operator=(const Class&)`
- Destructor: `~Class()`
- Move constructor (C++11): `Class(Class&&)`
- Move assignment (C++11): `Class& operator=(Class&&)`

The *rule of five*: Implement all of them or none of them (the better choice, *rule of zero*).

- Use of containers of the standard library
- Use of smart pointers

Supplement the library application:

- Create a class `Medium` with its data fields. In object-oriented languages classes are put in files with the same name (`Medium.cpp/.h`).
- Write a default constructor `Medium()` supplying default values.
- Write a constructor, that takes the data elements as parameters
- Write a *getter* for each data element (e.g. `string getSignature() const;`)
- Supplement the module *MediaManagement* by a function `int addMedium(const Medium& medium)`, that reuses the existing method.
- Supplement the main function by the instantiation of a *medium* on the stack and call the new function.

[19] http://www.cplusplus.com/doc/tutorial/classes/
[20] http://www.cplusplus.com/doc/tutorial/classes2/

5.2 Operators

In C++ existing operators can be overloaded, that means it can be filled with meaning for the corresponding object class. The stream operator is herein of special interest for forming a good looking representation.

Overloading operators[21]

Non-member operator overloads are often defined as Friend functions[22]

Input/output operators overloading[23]

Exercise:

Supplement the class `Medium` by input/output operators and an assignment operator.

- The output operator is producing a line as written to `media.csv`.
- The input operator fills the private data of an instance. For that reason the input string is converted into a `stringstream`. Then each element is taken from the stream by `getline(STREAM, STRING, SEP);`.
- Two instances are equal, if their signatures are equal.

Use these operators.

- Supplement the class `Medium` by a constructor that takes a csv line as parameter and uses the input operator.
- Supplement MediaManagement by a function `bool isSignatureInFile(const Medium& medium)`, that creates an instance of `Medium` for each line and checks for duplicates by comparing the instances (==).
- Supplement MediaManagement by a function `int addMedium(const Medium& medium)`, that
  - uses the function `isSignatureInFile`,
  - appends the given instance of `Medium` by using the output operator.
- Test the new implenentation.

5.3 Inheritance

Motivation: Implement common stuff only once.

Inheritance between classes[24] → *Inheritance between classes, What is inherited from the base class?*

[21]http://www.cplusplus.com/doc/tutorial/templates/
[22]http://www.cplusplus.com/doc/tutorial/inheritance/
[23]https://www.tutorialspoint.com/cplusplus/input_output_operators_overloading.htm
[24]http://www.cplusplus.com/doc/tutorial/inheritance/

Overwriting functions: [`014/vererbung001/kunde.h`, W:348]

An inherited function can be directly accessed inside the child class:

```
void print() const {
   std::cout << "Kundennummer:" << get_kundennummer() << std::endl;
   Person::print();
   // statt:
   // std::cout << "Vorname      :  " << get_vorname() << std::endl;
   // std::cout << "Nachname     :  " << get_nachname() << std::endl;
}
```

Instantiating a derived class the constructor of the base class is also called. By default the standard constructor. By explicit call in the initializer list a specific base construtor can be selected (line 15).

Since C++11 constructors can be derived to the derived class:

```
class Base {
public:
   Base(int ival) { cout << ival << endl; }
   Base(string s) { cout << s << endl; }
};

class Child : public Base {
public:
   // Inheritance of constructors
   using Base::Base;
   Child(float fval) { cout << fval << endl; }
};

// Examples of use
Child s1(222.222f);  // Child::Child(float)  + Base::Base()
Child s2(456);       // Base::Base(int)
Child s3("Hi Dad");  // Base::Base(string)
```

The derived class is type compatible to its base class (*is a*), but not vice versa:
.[listings/014/vererbung003/main.cpp]

Exercise:

Supplement the library application by the files `MediumHira.h/.cpp`:

- Derive the classes `Book` and `CD` from `MediumBase`.
- Distribute the attributes on the classes in a meaningful way.
- `Book` and `CD` obtain a constructor taking 4 parameter.
- Supplement each class by a method `string format() const`, that returns the data as string in that format as use in the file. You can use either the `std::to_string` or the output stream operator together with a `stringstream`.
- For class `MediumBase` a dummy implementation is used.
- Test the `format` function for all three classes in the main program.

Polymorphism

Motivation for this section:

Exercise:

- Supplement the library application by a function
  `int addMedium(const MediumBase& medium),`
  that uses the new `format()` method to write in the file. The duplicate check can be abandoned.
- Use this new method in the main program that way, that first a book and then a CD is passed to the method.

It can be seen, that only dummy entries are in the file, event there were correct book and CD instances in the main program. At this point a technique is needed, that decides at runtime what is the actual object class and calls the appropriate implementation. This is called *polymorphism*.

Polymorphic overwriting is done by the modifier `virtual` in the base class and `override` in the derived class: [listings/014/virtual02/main.cpp]

Exercise:

Supplement the library application:

- Implement the `format()` method polymorphic.
- Run the application.

Abstract Classes and Methods

The dummy implementation in `MediumBase` does not make much sense, but it must be existent for formal compiler reasons. The pure existence can be documented by an abstract method declaration:

```
virtual string format() = 0;
```

This demands the derived classes to implement this method. As a consequence the compiler forbids an instantiation of the base class.

A class owning at least one abstract method is called an abstract class.

Exercise:

Change `MediumBase` into an abstract class.

Exercise for unique_ptr:

- Supplement the class definition of MediumBase by a static method static MediumBase* createMedium(const string& csv_line), that creates an instance of Buch/CD on heap. Within this method the instance is kept in a unique_ptr.
- Call this function in the main program. In the main program the instance is kept in an unique_ptr and pass the instance to addMedium.

6 Templates

Templates provide the means of generic programming. That is programming without fixing the actual object class the function is dealing with. Function and class templates can be distinguished.

6.1 Function Template

Function Templates[25]

If there are types that require a spacial treatment, templates can be specialized: .[listings/015/Templates004/main.cpp]

Templates with several types: [015/Templates005/main.cpp]

Explicit instantiation of a template with a distinct type:
Example: [015/Templates006/main.cpp]

Exercise:

In util there is defined a function toInt. The conversion into float, bool or char would be the same way.

- Write a function template T toVal(const std::string& token) that does the converions in the described way.
- Test the implementation (e.g. use it instead of toInt and toChar. Mention the explicit instantiation needed here.

6.2 Class Templates

Class Templates[26]

[25]http://www.cplusplus.com/doc/oldtutorial/templates/
[26]http://www.cplusplus.com/doc/oldtutorial/templates/

For templates code is built and compiled when it is instantiated for a fixed type. For types that are used often templates can be instantiated on stock.

```
template<class T>
class Y // template definition
{
   void mf() { }
};
template class Y<char*>;      // explicit instantiation
```

To avoid compiling a template on different occurences, it had to be declared as:

```
extern template class Y<char*>;
```

This technique has advantages with big templates. Superfuous code is discarded at linking time (e.g. multiple compilation of a template instantiated for the same type).[27]

6.3 Templates of the Standard Library

Standard containers[28]→`vector, map`

Map[29] → `operator[], at(), size(), count()`

Iterators:

- `begin()` creates an iterator pointing to the first element.
- `end()` returns an iterator pointing after the end.
- The `++` operator pushes the iterator to the next element.

Iterator on a map:

```
std::map <std::string,std::string> phonebook;
// fill phonebook with data
std::map <std::string,std::string>::iterator mapIt;
for ( mapIt=phonebook.begin(); mapIt!=phonebook.end(); mapIt++)
        std::cout << mapIt->first << ": " << mapIt->second << std::endl;
```

Range-based For-Loop and Type Deduction

Known from many modern computer languages (foreach). It is usable for

- C-Array
- Containers (e.g. `vector, map, initializer_list`)
- Usage: `for (type var : array)`

[27]https://stackoverflow.com/questions/8130602/using-extern-template-c11 (2.1.2020)
[28]http://www.cplusplus.com/reference/stl/
[29]http://www.cplusplus.com/reference/map/map/

The compiler knows in many cases which type to use.

- Typing by `auto`
  `auto i = 5;`
  `for (auto var : array)`
- Typing by `decltype`

```
int b;
decltype(b) a;
```

`auto` always stands for the value type (no reference). `decltype` stands for reference types also. With C++14 there is a construct `decltype(auto)` that can be used like `auto` and can also represent reference types.[30]

```
auto i           = 5;    // int
int& iref        = i;    // int&
auto c           = iref; // int
auto& d          = i;    // int&
decltype(iref) e;        // int&
decltype(auto) f = iref; // int&
```

With C++11 it became simple to iterate over containers:

```
for ( auto& mapIt: phonebook)
        std::cout << mapIt.first << ": " << mapIt.second << std::endl;
```

Using iterators and without the use of `auto` the same code is written as:

```
std::map <std::string,std::string>::iterator mapIt;
for ( mapIt=phonebook.begin(); mapIt!=phonebook.end(); mapIt++)
        std::cout << mapIt->first << ": " << mapIt->second << std::endl;
```

Exercise:

Supplement the library application. For not reading the file each time, media data shall be stored in a `std::map` Therefore MediaManagement obtains internal data, becomes a class.

- Create a class `MediaManagementClass` with internal data
  `std::map<std::string, Medium> medium_map;`.
- Write a method `load()`, that reads the file and fills the map. Key is the signature, value an instance of `Medium`. This function is called in the constructor.

  Take the implementation of `bool isSignatureInFile(const Medium& m)` as model.
- Write a method `bool checkDuplicate(const string& signature)`, that checks if a given key exists in the `medium_map`.
- Write a method `void show()`, that iterates on the map and outputs the stored media. Consider the element of a map is a key-value pair. The key is accessed by `.first`, the value by `.second`.

[30]https://en.wikipedia.org/wiki/C%2B%2B14 (27.8.2018)

- Write a method `int addMedium(const Medium& medium)`, that calls `checkDuplicate` and appends the medium to the file. Then the map is updated by a call of `load`.
- Use this code in the main program.

The `Medium` instances are directly copied into the *map*. Therefore this construct is not working for instances from our class hierarchie. In that case the instances has to be on heap and the map manages pointers.

- Add a map `map<string,unique_ptr<MediumBase> >` to `MediaManagementClass` and fill it with the `load` method. To create this instance the fabric function `createMedium` can be used.
- Supplement the method `show()` with an output of this map.

Algorithms of the Standard Librabry

Functions[31] $\rightarrow$ `for_each(), find(), find_if(), copy(), count(), count_if(), sort()` Many of this function need a callback function as parameter. This can either be a "normal" function, either a function object. A function object (functor) is a class that implements the `()` operator:

```
class MyFunctor {
private:
        int internal;
public:
        MyFunctor(int i) :internal(i) {}
        bool operator()(int i) { return i<internal; }
}
```

Exercice:

Supplement the class `MedienverwaltungClass` by functions that use the algirithms of the standard library. As preparation the function `load()` is not only building the media map, but also a `vector` that holds all the media instances read from the file.

The exercices for `count_if`, `for_each`, `sort` and `find_if` can either be solved by the use of functors or by lambda functions.

- `count_if`: A function `countBooks()` that outputs the number of books.
- `for_each`: A function `show1()` that outputs the elements of the media map.

[31] http://www.cplusplus.com/reference/algorithm/

- `copy`: A function `show2()` that output the elements of the media vector. output stream iterator[32]
- `sort`: A function `show_sort_author()` that outputs all media sorted by `author`:
  - Write a function
    `bool comp_media(const Medium& m1, const Medium& m2)`
    that returns `true` if `author` of `m2` is bigger.
  - Sort the media vector by the `sort` function.
  - The output is done by calling `show2()`.
- `count`: A function `checkDuplicate(const Medium&)` that checks the media vector.
- `find_if`: A function `checkDuplicate1(const string&)` that browses the media map.

  The predicate function is a functor that takes the signature as parameter in the constructor. When executed it takes a `pair<string,Medium>` as paramter. The key of the parameter (`.first`) must be compared with the signature.

7 Tour de C++ 11[33]

Remark: If any construct does not work with your compiler, use Wandbox[34].

7.1 Language Kernel

Variadic Templates

Templates with a variable parameter count

```
template <typename ... Args>
function (Args ... args)
```

- `args` is a „parameter pack".
- `args...` is the unpacked parameter list.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <typeinfo>

template<typename T=int>
void mischmasch(T val=0L) {
  std::cout.width(3);
```

[32]http://www.cplusplus.com/reference/iterator/ostream_iterator/algorithm/ (3.5.2019)
[33]Grimm: Teil I
[34]https://wandbox.org/

```
  std::cout << typeid(val).name() << " ("
       << sizeof(val) << ") : " << val << std::endl;
}

template<typename First, typename ... Rest>
void mischmasch(First first, Rest ... rest) {
  mischmasch(first);
  mischmasch(rest ...);
}

int main( void ) {
 mischmasch();  // -> int=0L
 mischmasch(100);
 mischmasch(123, -2999, 123.123, "Hallo", 'c');
 mischmasch("Test", 2e12, 'x', 55555);
 return 0;
}
```

Exercise:

Write a function `bool split(istream,char,...)` that reads values from a comma separated *input stream* and converts them into the given types of the parameter list. Test the code + by providing a line from `medien.csv`:
`split(istream,char,string,string,string,char,int,int)`

The function is called this way:

```
string signatur, autor, titel;
char typ;
int seitenzahl, spieldauer;
stringstream ss ("A01,Heinz Autor,Der Titel,B,125,0");
bool ok = split(ss,',',signatur,autor,titel,typ,seitenzahl,spieldauer);
```

Lambda Functions

- Functions without name (anonymous functions)
- Lambdas can be used where function pointer of functors (executable objects implementing the `()` operator) are used.

Syntax:

```
[*1](*2){*3}
*1: Binding to the local context
  []: No Binding
  [=]: All values are copied (snapchot).
  [&]: All variables are referenced.
*2: runtime parameter
*3: implementation
```

Exercise:

- Compile and analyze the example `anhang_a/lambda.cpp`
- Supplement this example by usage of a function pointer and a functor.

Binding at the context

If it is intended to take variables of the current context into a lambda function these variables have to be specified in squared braces. With a prepended `&` the variable is taken as reference, otherwise it is copied. If all variables should be bound as reference, there is a single `&` between the braces, a `=` copies the variables (default):
`[a,&b]`: Puts a copy of `a` and a reference to `b` into the context of the lambda function.
`[=,&b]`: All variables are copied, but `b` is referenced in the context of the lambda function.

With C++14 variables can explicitly be filled out of the context, even the `move` function can be used.

```
auto lambda = [value = std::move(ptr)] {return *value;};
```

Exercise:

Modify `lambda.cpp` that way that the divisor is variable. The divisor is given by

- a variable out of teh current context.
- by assigning a number inside the context definition.

Generic Lambdas (C++14)[35]

With C++11 all type parameters had to be declared explicitly. With C++14 the parameters can be left generic:

```
auto lambda = [](auto x, auto y) {return x + y;};
```

This code is equivalent to an executable class template (functor):

```
struct unnamed_lambda
{
  template<typename T, typename U>
    auto operator()(T x, U y) const {return x + y;}
};
auto lambda = unnamed_lambda{};
```

[34]Grimm: S. 14
[35]https://en.wikipedia.org/wiki/C%2B%2B14 (27.8.2018)

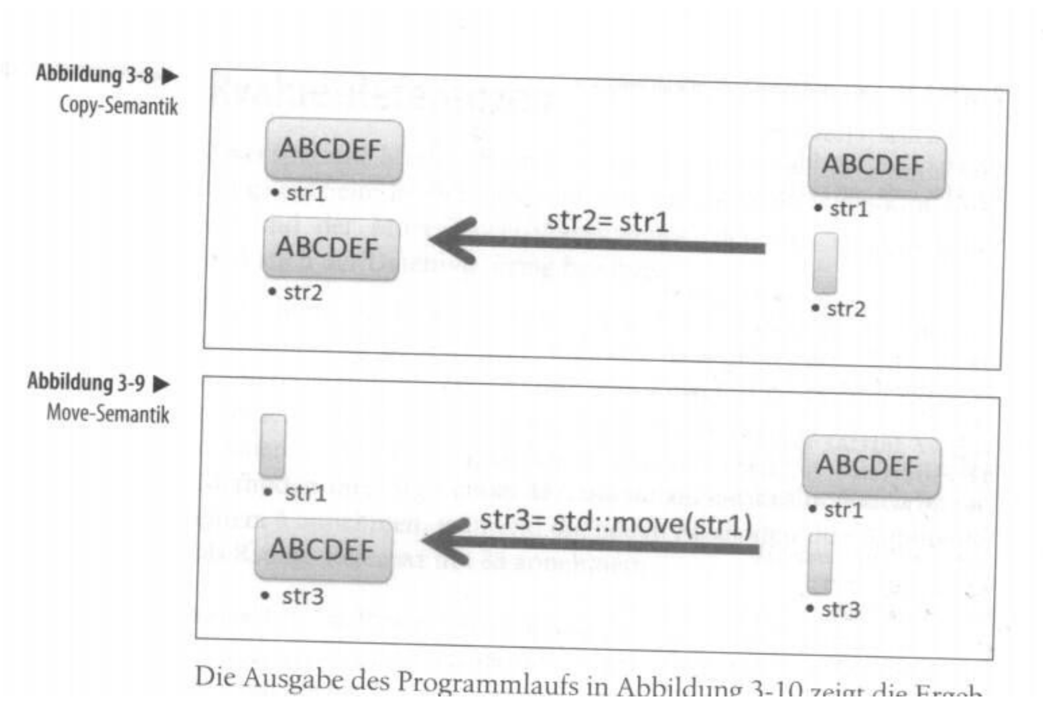


Figure 2: Move vs. Copy[36]

Move Semantics

Problem: C++ copies much too much:

- Objects are copied into containers
- Objects are copied, but the source is no longer used (e.g. return statement).
- C++ behaves different to Java, C#, Visual C++: In these languages only references are copied.

Move utilizes a performant transfer of objects: Fig. 2.

- When is *move* used?
  explicit: `a = std::move(b);`
  implicit: On the right side is a *rvalue*
- What is a *rvalue*?
  - An expression that can only stand on the right side of the equation sign.
  - An expression without name: `a = MyObject();`

Definition syntax of move constructor and move assignment:

```
MyClass(MyClass&&) { ... }
MyClass& operator=(MyClass&&) { ... }
```

- *Rule of the big five*: If one of the big five is implemented, all have to be implemented.

[36]Grimm: S. 32

- *Rule of zero*: It is better to avoid an implementation of the big five[37].

Sollen einzelne der implizit durch den Compiler generierten Operationen verboten werden, kann dies mit `delete`.

```
class Simple{
private:
  int *daten{nullptr}; // Roher Zeiger !!!
  Simple( int n ) : daten{ new int[n] } {}
  ~Simple() { delete[] daten; }
  Simple(const Simple&) = delete;  // keine Kopie
  Simple& operator=(const Simple&)=delete; // keine Zuweisung
  Simple(Simple&&) = delete;  // kein Verschieben
  Simple& operator=(Simple&&)=delete; // keine Verschiebzuweisung
};
```

7.2 Standard Library

New Functions

C++11 Algorithm[38] $\rightarrow$ `all_of`, `any_of`, `copy_if`, `is_sorted`, `minmax_element`

It is planned to standardisize the parallelisation of the algorithms with C++17.

Exercise:

Supplement `MediaManagementClass` by the method `checkDuplicateAlgo` that does the check by the use of `std::any_of`.

```
template <class InputIt, class UnaryPred>
  bool any_of (InputIt first, InputIt last, UnaryPred pred);
```

New Containers

- `tuple`: A set of values of different types (generalization of `pair`)
- `array`: `vector` of constant length
  Advantage: Better performance
- Further containers that may have a better performance in spacial cases.
  - `forward_list`
  - `unordered_map`
  - `unordered_set`
  - `unordered_multimap`
  - `unordered_multiset`

[37]Wolf: 291.
[38]http://www.cplusplus.com/reference/algorithm/

Exercise[39]:

Compare the access time of a `map` with an `unordered_map`.

Create two big assotiative containers of type `map<int,int>` and `unordered_map<int,int>` and access a tenth of the elements by the index operator with random indices. Increment the size of the containers.

The measurement of the access time is postponed to the introduction of random numbers and time library.

Smart Pointer

- `auto_ptr`: C++03 implementation: No difference between move and copy → does not fit to C++11
- `unique_ptr` replaces `auto_ptr`.
- `shared_ptr`: works with reference counter.
- `weak_ptr`: wrapper for `shared_ptr`: "A non activated `shared_ptr`". An activiation is done by `.lock()`. Can be useful for the handling of cyclic references (s. fig. 3).

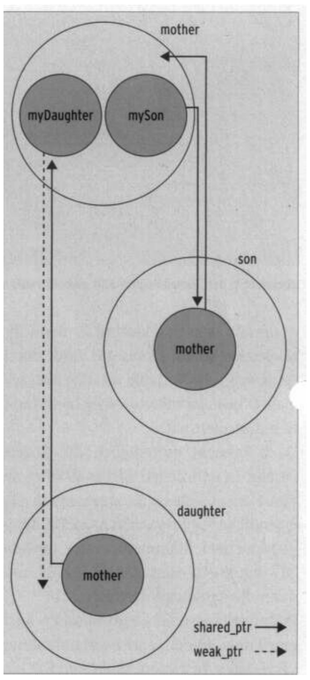


Figure 3: Cyclic Reference[40]

[39]Grimm: Aufg. 20-13 S. 447: `mapHashComparison.cpp`.
[40]Linux Magazin 04/2013, p. 106.

Exercise:

- Compile and analyze example 5-9 (`uniquePtrMove.cpp`).
  Access the pointers `unique1` and `unique2` before and after the move statement.
- Compile and analyze example `anhang_a/smart_ptr01.cpp` [W:442]
- Compile and analyze example `TourdeC++11/sharedWeakPtr.cpp`.
- Compile and analyze example *Zyklische Referenz*[41].

Regular Expressions

Principle setup:

- Define a regular expression
  `std::regex r(<muster>);`
- check if a string matches the regex exactly
  `std::regex_match (<str>,r);`
- Get the search result
  `std::smatch s;`
  `std::regex_search(<str>, s, r);`
  `s` contains the matches.
- Search and replace
  `std::regex_replace (<str>,r,<repl>);`
  With `$1, $2, ...` groups of the regular expression can be accessed.

Exercise:

- Compile and analyze the example `anhang_a/regexpr.cpp` [W:436f]
- Build a parser for comma separated strings that recognizes the types integer, byte, value[42].[43]

Remark: There is a REGEX-Tester[44]

Random Numbers

The generation of random numbers works along the following pattern:

- Creation of a random number stream

[41]Listing 4 from LM 03/2013
[42]Value: C++ literal like 22_km. Sample solution: Regex.cpp
[43]Sample solution: Regex.cpp
[44]http://regex101.com

- Mapping to a distribution function
- Retrival of random numbers

The example `DieStandardbibliothek/distribution.cpp`[45] shows how to generate random numbers:

- L. 25/28: Definition of the random number stream
- L. 33-39: Mapping on the desired distribution function
- L. 47-50: Retrieval of the random numbers

Exercise:

Supplement the `map/unordered_map` comparision exercise[46]:

- Assign every element of `map/unordered_map` its key value.
- Access (in a loop) `mapSize/10` elements by generating random keys and output them.

Time Library

The C++11 timer library knows the following classes:

- Time point `std::chrono::time_point`
- Duration `std::chrono::duration`
- Clock `std::chrono::system_clock`

A time measurement can be implemented by the following code:

```
auto start = std::chrono::system_clock::now();
// code for measurement
std::chrono::duration<double> dur=std::chrono::system_clock::now()-start;
std::cout << "time: " << dur.count() << " seconds" << std::endl;
```

Exercise:

Complete the `map/unordered_map` comparision exercise[47]:

Measure the access time. To get reasonable results do no screen output during the measurement.

[45] Grimm: Beispiel 19-30, S. 355.
[46] Part of `DieStandardbibliothek/Aufgaben/mapHashComparison.cpp`
[47] Part of `DieStandardbibliothek/Aufgaben/mapHashComparison.cpp`

8 References

Grimm, Rainer	C++11 für Programmierer, 2014
Wolf, Jürgen	Grundkurs C++, 3. Auflage