
C++17
Rainer Grimm
Training, Coaching und Technologieberatung
www.ModernesCpp.de

Historie von C++

C++98 (1998)

- Templates
- STL mit Containern und Algorithmen
- Strings
- I/O Streams

C++11 (2011)

- Move Semantik
- Vereinheitlichte Initialisierung
- `auto` und `decltype`
- Lambda-Funktionen
- `constexpr`
- Multithreading und das Speichermodell
- Reguläre Ausdrücke
- Smart Pointer
- Hashtabellen
- `std::array`

C++14 (2014)

- Reader-Writer Locks
- Verallgemeinerte Lambda-Funktionen

C++17 (2017)

- Fold expressions
- constexpr if
- Structured binding
- `std::string_view`
- Parallele Algorithmen der STL
- Dateisystem Bibliothek
- `std::any`, `std::optional` und `std::variant`

C++20 (2020)

- Asynchrone Netzwerkbibliothek
- Transactional Memory
- Erweitere Futures
- Concepts
- Range Bibliothek

C++17 – Die Kernsprache

Kernsprache

- Fold Expressions
- constexpr if
- `if` und `switch` mit Initialisieren
- Strukturierte Bindung
- Automatische Typableitung von Klassen-Templates
- Trigraphen entfernt

Fold Expressions

Reduziert ein Parameter Pack über einem binären Operator.

```
template<typename ... Args>
bool all(Args ... args){
  return ( ... && args);    // return ((true && true)&& true)&& false;

}

bool b = all(true, true, true, false);
```

Zwei Variationen

- die Fold Expression besitzt ein Startwert
- das Parameter Pack wird von links oder rechts parametrisiert

Fold Expressions

C++17 unterstützt 32 Operatoren in fold expressions:

```
+   -   *   /   %   ^   &   |   =   <   >   <<   >>   +=   -=
*=   /=   %=   ^=   &=   |=   <<=   >>=   ==   !=   <=   >=
&&   ||   ,   .*   ->*
```

Operatoren mit Default-Werten

Operator	Symbol	Default-Wert
Logisches AND	`&&`	`true`
Logisches OR	`||`	`false`
Komma Operator	`,`	`void()`

constexpr if

- constexpr if: `if constexpr predicate`
  - erlaubt es, Sourcecode bedingt zu kompilieren
  - benötigt einen Prädikat, das zur Compilezeit evaluierbar ist
  - prüft auch den nicht kompilierten Sourcecode auf Syntax

```
template <typename T>
auto getAnswer(T t){
  static_assert(std::is_arithmetic_v<T>); // arithmetic
  if constexpr (std::is_integral_v<T>)    // integral
    return 42;
  else                                    // floating point
    return 42.0;
}
```

! `getAnswer(5)` besitzt verschiedene Rückgabetypen

`if` und `switch` mit Initialisieren

Klassisches C++

```
std::map<int, std::string> myMap;

auto res = myMap.insert(value);
if (res.second){
  useResult(res.first);
  // ...}
else{
  // ...
}
```

C++17

```
std::map<int, std::string> myMap;

if (auto res = myMap.insert(value); res.second){
  useResult(res.first);
  // ...}
else{
  // ...
} // res is automatically destroyed
```

Entsprechend zur `for`-Schleifen, sind die Initialisierer nur im Bereich des Blockes gültig

Strukturierte Bindung

`std::tuple` oder `struct's` können direkt an Variablen gebunden werden.

```
std::tuple<T1, T2, T3, T4, T5> getValues(){ ... }
auto [a, b, c, d, e] = getValues(); // types are: T1, T2, T3, T4, T5

struct Values{
  std::string f{"test"};
  int g{5.5};
};
Values val;
auto [f, g] = v;
```

➡ Die Variable `a` bis `g` werden automatisch erzeugt

Strukturierte Bindung

Initialisierer und strukturierte Bindung arbeiten Hand in Hand

```
std::map<int,std::string> myMap;
if (auto [iter, succeeded] = myMap.insert(value); succeeded){
  useIter(iter);
  // ...
}
else{
// ...
} iter and succeeded are automatically be destroyed
```

Automatische Typableitung von Klassen-Templates

Funktions-Template können ihre Typen automatisch ableiten

```
template <typename T>
void funcTemp(const T& t){ }

int main(){
  funcTemp(5.5);          // not funcTemp<double>(5.5);
  funcTemp(5);            // not funcTemp<int>(5);
}
```

! Klassen-Templates konnten das nicht.

Automatische Typableitung von Klassen-Templates

Klassen-Templates können ihre Typen mit C++17 automatisch ableiten.

```
Template <typename T>
struct classTemp{
  classTemp(const T& t){ }
};

int main(){
  classTemp(5.5);        // Classic classTemp<double>(5.5);
  classTemp(5);          // Classic classTemp<int>(5);
}
```

Trigraphen entfernt

Tripgraphen sind Kombinationen aus drei Buchstaben, die für ein Zeichen stehen.

Welche Ausgabe besitzt das Programm?

```
int main()??<
  ??( ??) ??< ??> ();
??>
```

Trigraphen entfernt

Reguläre Syntax

```
int main(){
  []{}();
}
```

Trigraph	Ersetztes Zeichen
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

Seit Ende der 1980er Jahre besteht keine Notwendigkeit mehr zur Verwendung von Trigraphen in C, da auf Tastaturen heutzutage alle Sonderzeichen vorhanden sind ... .
(Wikipedia)

C++17 – Die Bibliothek

Bibliothek

- `std::string_view`
- Parallele Algorithmen der STL
- Die Dateisystem Bibliothek
- `std::any`
- `std::optional`
- `std::variant`

`std::string_view`

Ein `std::string_view` ist eine Referenz auf einen String, die den String nicht besitzt.

`string_view`

- lässt sich billig kopieren
- benötigt keinen Speicher
- besitzt nur einen Zeiger auf die Zeichensequenz und seine Länge
- besitzt fast nur lesende Operationen.
- erhält die zwei neuen Methoden `remove_prefix` und `remove_suffix`.

`string_view` besitzt ein ähnliches Interface wie ein `string` um den einfachen Umstieg zu unterstützen.

std::string_view

Keine Speicherallokation mit `string_view`.

```
void* operator new(std::size_t count){
    std::cout << "   " << count << " bytes" << std::endl;
    return malloc(count);
}

void getString(const std::string& str){}
void getStringView(std::string_view strView){}

std::string large = "0123456789-123456789-123456789-123456789";     // 41 bytes
std::string substr = large.substr(10);                              // 31 bytes

std::string_view largeStringView{large.c_str(), large.size()};
largeStringView.remove_prefix(10);

getString(large);
getString("0123456789-123456789-123456789-123456789");              // 41 bytes
const char message []= "0123456789-123456789-123456789-123456789"; // 41 bytes
getString(message);

getStringView(large);
getStringView("0123456789-123456789-123456789-123456789");
getStringView(message);
```

std::string_view

Substrings in konstanter Zeit erzeugen.

```
string grimmsTales = strStream.str();
auto start = chrono::steady_clock::now();
for (auto i = 0; i  < access; ++i ) {
  grimmsTales.substr(randValues[i], count);
}
chrono::duration<double> durString= chrono::steady_clock::now() - start;
cout << durString.count();

string_view grimmsTalesView{grimmsTales.c_str(), size};
start = chrono::steady_clock::now();
for (auto i = 0; i  < access; ++i ) {
  grimmsTalesView.substr(randValues[i], count);
}
chrono::duration<double> durStringView= chrono::steady_clock::now() - start;
cout << durStringView.count();

cout << durString.count()/durStringView.count();
```

std::string_view

Performanzvergleich:

- erzeuge 10 Millionen Teilstrings der Länge `count`

count	std::string	std::string_view	std::string/ std::string_view
10	0.108 sec.	0.011 sec.	9.724
30	0.438 sec.	0.010 sec.	45.412
100	0.550 sec.	0.010 sec.	55.217
1000	1.541 sec.	0.009 sec.	156.351
10000	7.727 sec.	0.009 sec.	833.442
100000	70.435 sec.	0.009 sec.	7533.44

Parallele Algorithmen der STL

Die Ausführungsstrategie eines STL Algorithmus kann ausgewählt werden.

- Ausführungsstrategie

  `std::execution::seq`
  - Sequentiell in einem Thread

  `std::execution::par`
  - Parallel in mehreren Threads

  `std::execution::par_unseq`
  - Parallel und vektorisisiert in mehreren Threads ➡ SIMD

Parallele Algorithmen der STL

```
using namespace std;
vector<int> vec ={1, 2, 3, 4, 5 ... }


sort(vec.begin(), vec.end());     // sequential as ever

sort(execution::seq, vec.begin(), vec.end());       // sequential
sort(execution::par, vec.begin(), vec.end());       // parallel
sort(execution::par_unseq, vec.begin(), vec.end()); // par + vec
```

Parallele Algorithmen der STL

```
const int SIZE= 8;
int vec[]={1, 2, 3, 4, 5, 6, 7, 8};
int res[]={0,};

int main(){
  for (int i= 0; i < SIZE; ++i){
    res[i]= vec[i] + 5;
  }
}
```

Nicht vektorisiert

```
main:
  movl $0, %eax
.L2:
  movl vec(%rax), %ecx
  leal 5(%rcx), %edx
  movl %edx, res(%rax)
  addq $4, %rax
  cmpq $32, %rax
  jne .L2
  movl $0, %eax
  ret
```

Vektorisiert

```
main:
  movdqa .LC0(%rip), %xmm0
  xorl %eax, %eax
  movdqa vec(%rip), %xmm1
  paddd %xmm0, %xmm1
  paddd vec+16(%rip), %xmm0
  movaps %xmm1, res(%rip)
  movaps %xmm0, res+16(%rip)
  ret
```

Parallele Algorithmen der STL

adjacent_difference, adjacent_find, all_of any_of, copy, copy_if, copy_n, count, count_if, equal, **exclusive_scan**, fill, fill_n, find, find_end, find_first_of, find_if, find_if_not, **for_each**, **for_each_n**, generate, generate_n, includes, **inclusive_scan**, inner_product, inplace_merge, is_heap, is_heap_until, is_partitioned, is_sorted, is_sorted_until, lexicographical_compare, max_element, merge, min_element, minmax_element, mismatch, move, none_of, nth_element, partial_sort, partial_sort_copy, partition, partition_copy, **reduce**, remove, remove_copy, remove_copy_if, remove_if, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, rotate, rotate_copy, search, search_n, set_difference, set_intersection, set_symmetric_difference, set_union, sort, stable_partition, stable_sort, swap_ranges, transform, **transform_exclusive_scan**, **transform_inclusive_scan**, **transform_reduce**, uninitialized_copy, uninitialized_copy_n, uninitialized_fill, uninitialized_fill_n, unique, unique_copy

Parallele Algorithmen der STL

`std::transform_reduce`

- Haskells Funktion `map` heißt in C++ `std::transform`
- `std::transform_reduce` ➡ `std::map_reduce`

```
std::vector<std::string> str = {"Only", "for", "testing", "purpose"};

std::size_t res = std::transform_reduce(std::execution::par,
                  str.begin(), str.end(), 0,
                  [](std::string s){ return s.length(); },
                  [](std::size_t a, std::size_t b){ return a + b; });

std::cout << res << std::endl // 21
```

Die Dateisystem Bibliothek

Das Dateisystem

- basiert auf `boost::filesystem`.
- besteht aus den Abstraktionen Datei, Dateiname und Pfad.

Dateien können

- Verzeichnisse, harte Links, symbolische Links oder auch reguläre Dateien sein.

Pfade können

- absolute oder relative Pfade sein.

Die Komponenten des Dateisystems sind optional.

Die Dateisystem Bibliothek

Classes

`path`	represents a path (class)
`filesystem_error`	an exception thrown on file system errors (class)
`directory_entry`	a directory entry (class)
`directory_iterator`	an iterator to the contents of the directory (class)
`recursive_directory_iterator`	an iterator to the contents of a directory and its subdirectories (class)
`file_status`	represents file type and permissions (class)
`space_info`	information about free and available space on the filesystem (class)
`file_type`	the type of a file (enum)
`perms`	identifies file system permissions (enum)
`copy_options`	specifies semantics of copy operations (enum)
`directory_options`	options for iterating directory contents (enum)
`file_time_type`	represents file time values (typedef)

Die Dateisystem Bibliothek

Non-member functions

absolute system_complete	composes an absolute path converts a path to an absolute path replicating OS-specific behavior (function)
canonical	composes a canonical path (function)
copy	copies files or directories (function)
copy_file	copies file contents (function)
copy_symlink	copies a symbolic link (function)
create_directory create_directories	creates new directory (function)
create_hard_link	creates a hard link (function)
create_symlink create_directory_symlink	creates a symbolic link (function)
current_path	return current working directory (function)
exists	checks whether path refers to existing file system object (function)
equivalent	checks whether two paths refer to the same file system object (function)
file_size	returns the size of a file (function)
hard_link_count	returns the number of hard links referring to the specific file (function)
last_write_time	gets or sets the time of the last data modification (function)
permissions	modifies file access permissions (function)
read_symlink	obtains the target of a symbolic link (function)
remove remove_all	removes a file or empty directory removes a file or directory and all its contents, recursively (function)
rename	moves or renames a file or directory (function)
resize_file	changes the size of a regular file by truncation or zero-fill (function)
space	determines available free space on the file system (function)
status symlink_status	determines file attributes determines file attributes, checking the symlink target (function)
temp_directory_path	returns a directory suitable for temporary files (function)

Die Dateisystem Bibliothek

File types

`is_block_file`	checks whether the given path refers to block device (function)
`is_character_file`	checks whether the given path refers to a character device (function)
`is_directory`	checks whether the given path refers to a directory (function)
`is_empty`	checks whether the given path refers to an empty file or directory (function)
`is_fifo`	checks whether the given path refers to a named pipe (function)
`is_other`	checks whether the argument refers to an *other* file (function)
`is_regular_file`	checks whether the argument refers to a regular file (function)
`is_socket`	checks whether the argument refers to a named IPC socket (function)
`is_symlink`	checks whether the argument refers to a symbolic link (function)
`status_known`	checks whether file status is known (function)

Die Dateisystem Bibliothek: Überblick

```
int main(){
    cout << fs::current_path() << endl;                    // "/home/rainer"
    string dir= "sandbox/a/b";
    fs::create_directories(dir);
    ofstream("sandbox/file1.txt");
    fs::path symPath= fs::current_path() /=  "sandbox";
    symPath /= "syma";
    fs::create_symlink("a", "symPath");

    cout << fs::is_directory(dir) << endl;                 // true
    cout << fs::exists(symPath) << endl;                   // true
    cout << fs::is_symlink(symPath) << endl;               // true

    for(auto& p: fs::recursive_directory_iterator("sandbox"))
        cout << p << " ";          // "sandbox/syma sandbox/file1.txt sandbox/a sandbox/a"
    fs::remove_all("sandbox");
}
```

Die Dateisystem Bibliothek: Dateirechte

```
void printPerms(fs::perms perm){
  std::cout << ((perm & fs::perms::owner_read) != fs::perms::none ? "r" : "-")
            << ((perm & fs::perms::owner_write) != fs::perms::none ? "w" : "-")
            << ((perm & fs::perms::owner_exec) != fs::perms::none ? "x" : "-")
            << ((perm & fs::perms::group_read) != fs::perms::none ? "r" : "-")
            << ((perm & fs::perms::group_write) != fs::perms::none ? "w" : "-")
            << ((perm & fs::perms::group_exec) != fs::perms::none ? "x" : "-")
            << ((perm & fs::perms::others_read) != fs::perms::none ? "r" : "-")
            << ((perm & fs::perms::others_write) != fs::perms::none ? "w" : "-")
            << ((perm & fs::perms::others_exec) != fs::perms::none ? "x" : "-")
            << std::endl;
}
std::ofstream("rainer.txt");

std::cout << "Initial file permissions for a file: ";
printPerms(fs::status("rainer.txt").permissions());

fs::permissions("rainer.txt", fs::perms::add_perms |
                fs::perms::owner_all | fs::perms::group_all);
std::cout << "Adding all bits to owner and group:  ";
printPerms(fs::status("rainer.txt").permissions());

fs::permissions("rainer.txt", fs::perms::remove_perms |
  fs::perms::owner_write | fs::perms::group_write | fs::perms::others_write);
std::cout << "Removing the write bits for all:     ";
printPerms(fs::status("rainer.txt").permissions());
```

```
Initial file permissions for a file: rw-r--r--
Adding all bits to owner and group:  rwxrwxr--
Removing the write bits for all:     r-xr-xr--
```

Die Dateisystem Bibliothek: Größe

```
namespace fs = std::filesystem;

fs::space_info root = fs::space("/");
fs::space_info usr = fs::space("/usr");

cout << ".         Capacity       Free       Available\n"
     << "/    " << root.capacity << "    "
     << root.free << "    " << root.available << "\n"
     << "usr  "  << usr.capacity << "    "
     << usr.free <<  "    " << usr.available;
```

```
          Capacity       Free       Available
/    42140499968   18342744064   17054289920
usr  42140499968   18342744064   17054289920
```

Die Dateisystem Bibliothek: Zeitattribut

```
fs::path path = fs::current_path() / "rainer.txt";
ofstream(path.c_str());
auto ftime = fs::last_write_time(path);

time_t cftime = chrono::system_clock::to_time_t(ftime);
cout << "Write time on server "
     << asctime(localtime(&cftime));
cout << "Write time on server "
     << asctime(gmtime(&cftime)) << endl;

fs::last_write_time(path, ftime + 2h);
ftime = fs::last_write_time(path);

cftime = chrono::system_clock::to_time_t(ftime);
cout << "Local time on client "
          << asctime(localtime(&cftime));
```

```
Write time on server Tue Oct 10 06:28:04 2017
Write time on server Tue Oct 10 06:28:04 2017

Local time on client Tue Oct 10 08:28:04 2017
```

Die Dateisystem Bibliothek: Dateicharakteristiken

```
void printStatus(const fs::path& path_){
  cout << path_;
  if(!fs::exists(path_)) cout << " does not exist";
  else{
    if(fs::is_block_file(path_)) cout << " is a block file\n";
    if(fs::is_character_file(path_)) cout << " is a character device\n";
    if(fs::is_directory(path_)) cout << " is a directory\n";
    if(fs::is_fifo(path_)) cout << " is a named pipe\n";
    if(fs::is_regular_file(path_)) cout << " is a regular file\n";
    if(fs::is_socket(path_)) cout << " is a socket\n";
    if(fs::is_symlink(path_)) cout << "          is a symlink\n";
  }
}

fs::create_directory("rainer");
printStatus("rainer");

std::ofstream("rainer/regularFile.txt");
printStatus("rainer/regularFile.txt");

fs::create_directory("rainer/directory");
printStatus("rainer/directory");

mkfifo("rainer/namedPipe", 0644);
printStatus("rainer/namedPipe");

struct sockaddr_un addr;
addr.sun_family = AF_UNIX;
std::strcpy(addr.sun_path, "rainer/socket");
int fd = socket(PF_UNIX, SOCK_STREAM, 0);
bind(fd, (struct sockaddr*)&addr, sizeof addr);
printStatus("rainer/socket");

fs::create_symlink("rainer/regularFile.txt", "symlink");
printStatus("symlink");

printStatus("dummy.txt");
```

```
"rainer" is a directory
"rainer/regularFile.txt" is a regular file
"rainer/directory" is a directory
"rainer/namedPipe" is a named pipe
"rainer/socket" is a socket
"symlink" is a regular file
          is a symlink
"dummy.txt" does not exist
```

`std::optional`

`std::optional` ist ein Datentyp, der einen oder keinen Wert besitzen kann.

std::optional

- ist durch Haskell Maybe Monade inspiriert.
- wird für Berechnungen verwendet, die nur eventuell einen Wert zurückgibt. ➡ Abfrage einer Datenbank oder einer Hashtabelle

Spezielle Werte wurden verwendet, um Nicht-Ergebnisse darzustellen.

➡ Null-Zeiger, leere Strings oder besondere Zahlen

❗ Nicht-Ergebnisse sind ein Missbrauch des Typsystems.

std::optional

```
optional<int> getFirst(const vector<int>& vec){
  if ( !vec.empty() ) return optional<int>(vec[0] );
  else return optional<int>();
}
int main(){
    vector<int> myVec{1,2,3};
    vector<int> myEmptyVec;
    auto myInt= getFirst(myVec);

    if (myInt){
        cout << *myInt << endl;                                  // 1
        cout << myInt.value() << endl;                           // 1
        cout << myInt.value_or(2017) << endl;                    // 1
    }
    optional<int> myEmptyInt= getFirst(myEmptyVec);

    if (!myEmptyInt) cout << myEmptyInt.value_or(2017) << endl;  // 2017
}
```

`std::any`

`std::any` ist ein type-sicherer Container, der genau einen beliebigen Wert eines beliebigen Typs annehmen kann.

`std::any`

- Die Werte der Typen müssen copy-konstruierbar sein.
- Die freie Funktion `std::any_cast` erlaubt den typ-sicheren Zugriff auf den Wert.

`std::any_cast` wirft eine `std::bad_any_cast` Ausnahme, falls der nachgefragte Typ nicht passt.

std::any

```
struct MyClass{};

using namespace std;

int main(){
    vector<any> anyVec(true, 2017, string("test"), 3.14, MyClass());
    cout << any_cast<bool>(anyVec[0]);                    // true
    int myInt= any_cast<int>(anyVec[1]);
    cout << myInt << endl;                                // 2017
    cout << anyVec[0].type().name();                      // b
    cout << anyVec[1].type().name();                      // i

}
```

`std::variant`

`std::variant` ist eine typ-sichere Union.

`std::variant`

- besitzt einen Wert eines ihrer Typen.
- kann keine Referenzen, Arrays oder den Typ `void` besitzen.
- kann einen Typ mehrfach besitzen.

Eine Default-initialisierte `std::variant`

- verwendet ihre erste Variante.

➡ Die erste Variante benötigt einen Default-Konstruktor.

std::variant

```
int main(){
  variant<int, float> v, w;
  v = 12;                       // v contains int

  int i = get<int>(v);
  w = get<int>(v);
  w = get<0>(v);                // same effect as the previous line
  w = v;                        // same effect as the previous line
  //  get<double>(v);           // error: no double in [int, float]
  //  get<3>(v);                // error: valid index values are 0 and 1
  try{
    get<float>(w);              // w contains int, not float: will throw
  }
  catch (bad_variant_access&) {}

  variant<string> v("abc");     // converting constructor works when unambiguous
  v = "def";                    // converting assignment works when unambiguous
}
```

C++17 auf einen Blick

Kernsprache

- Fold Expressions
- constexpr if
- `if` und `switch` mit Initialisieren
- Strukturierte Bindung
- Automatische Typableitung von Klassen-Templates
- Trigraphen entfernt

Bibliothek

- `std::string_view`
- Parallele Algorithmen der STL
- Die Dateisystem Bibliothek
- `std::any`
- `std::optional`
- `std::variant`

Groß oder Klein

Blogs

[www.grimm-jaud.de [De]](http://www.grimm-jaud.de)

[www.ModernesCpp.com [En]](http://www.ModernesCpp.com)

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de