Advanced Developers Conference
Development for Professionals!

C++17

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de

C++17 auf einen Blick

Kernsprache

- Fold Expressions
- constexpr if
- `if` und `switch` mit Initialisieren
- Strukturierte Bindung
- Inline Variablen
- Automatische Typableitung von Klassen-Templates und Nicht-Typ Template-Parametern
- Kopieren vermeiden
- `auto_ptr` und Trigraphen entfernt

Bibliothek

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

Was es (leider) nicht in C++17 schaffte

- Asynchrone, plattformabhängige Bibliothek zur Netzwerk- und I/O Programmierung
  - Basierend auf Boost.Asio Bibliothek von Christopher Kohlhoff

- Transactional Memory
  - Transaktionen aus der Datenbank Theorie auf Software angewandt
  - Transaktion folgen dem ACID
  - C++ soll es als `synchronized` Blöcke und `atomic` Blöcke geben

Was es (leider) nicht in C++17 schaffte

- Erweiterte Futures
  - C++11 Futures um Komposition erweitert
    - `fut.then(future)`: `future` wird nach `fut` ausgeführt
    - `fut.when_any(futures)`: `fut` wird ausgeführt, falls einer der `futures` fertig ist
    - `fut.when_all(futures)`: `fut` wird ausgeführt, falls alle der `futures` fertig sind

- Concepts
  - Typsystem für Templates
  - Typ-Parameter müssen Anforderungen genügen (`Equal`)
  - sind dort einsatzbar, wo `auto` verwendet wird
  - können selbst definiert werden

Was es (leider) nicht in C++17 schaffte

- Ranges Bibliothek
  - wird von Eric Niebler implementiert
  - setzt Concepts ein
  - erlaubt Algorithmen direkt auf Container zu agieren

    `std::sort(vec.begin(), vec.end())` ≠ `std::sort(vec)`
  - erweitert C++ um funktionale Feature
    - Bedarfsauswertung
    - Funktionskomposition
    - Range Comprehension

C++17 – Die Kernsprache

Kernsprache

- Fold Expressions
- constexpr if
- `if` und `switch` mit Initialisieren
- Strukturierte Bindung
- Inline Variablen
- Automatische Typableitung von Klassen-Templates und Nicht-Typ Template Parametern
- Kopieren vermeiden
- `auto_ptr` und Trigraphen entfernt

Typableitung von Template-Parametern

C++17 unterstützt automatische Typableitung von Nicht-Typ Template-Parametern.

- Nicht-Typ Template-Parameter sind
  - `nullptr`
  - Ganzzahlen
  - lvalue Referenzen
  - Zeiger
  - Aufzählungstypen

Typableitung von Template-Parametern

Für Klassen-Templates

```
template <auto N>
class MyClass{
  ..
};

template <int N>
class MyClass<N> {
  ..
};

MyClass<'x'> myClass2;      // Primary template for char
MyClass<2017>  myClass1;    // Partial specialisation for int
```

Typableitung von Template-Parametern

Für Variadic Templates

```
template <auto... ns>
class VariadicTemplate{
  ...
};

template <auto n1, decltype(n1)... ns>
class TypedVariadicTemplate{
  ...
};

VariadicTemplate<1, true, 'c', nullptr> varTemp;  // arbitrary type
TypedVariadicTemplate<1, 2, 3, 4>  typedVarTemp;  // fixed type
```

Kopieren vermeiden

RVO

```
Type func(){
  return Type{};  // no copy (1a)
}

Type type = func(); // no copy (2)
```

NRVO

```
Type func(){
  Type type;
  return type;  // copy allowed (1b)
}

Type type = func();  // no copy (2)
```

(N)RVO steht für (**N**amed) **R**eturn **V**alue **O**ptimization.

RVO (1a)

- Der Compiler darf nicht mehr kopieren.

NRVO (1b)

- Der Compiler darf kopieren.

`auto_ptr` entfernt

`std::auto_ptr`

- wurden mit C++11 auf *deprecated* gesetzt
- werden mit C++17 entfernt
- verschieben beim kopieren heimlich ihre Ressource
- werden durch `std::unique_ptr` ersetzt

```
std::auto_ptr<int> ap1(new int(2011));
std::auto_ptr<int> ap2= ap1;                    // OK

std::unique_ptr<int> up1(new int(2011));
std::unique_ptr<int> up2= up1;                  // ERROR
std::unique_ptr<int> up3= std::move(up1);       // OK
```

Trigraphen entfernt

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

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

Welche Ausgabe besitzt das Programm?

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)

Was ich noch sagen wollte

`auto` in Kombination mit {}-Initialisierung

- Bisher

```
auto initA{1};          // std::initializer_list<int>
auto initB= {2};        // std::initializer_list<int>
auto initC{1, 2};       // std::initializer_list<int>
auto initD= {1, 2};     // std::initializer_list<int>
```

- C++17

```
auto initA{1};          // int
auto initB= {2};        // std::initializer_list<int>
auto initC{1, 2};       // error, no single element
auto initD= {1, 2};     // std::initializer_list<int>
```

C++17 – Die Bibliothek

Bibliothek

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

std::byte

Ein Byte ist für den Speicherzugriff konzipiert und ist keine Ganzzahl oder kein Buchstabe.

```
template <class IntType>
  constexpr byte operator<<(byte b, IntType shift);
template <class IntType>
  constexpr byte operator>>(byte b, IntType shift);
constexpr byte operator|(byte l, byte r);
constexpr byte operator&(byte l, byte r);
constexpr byte operator~(byte b);
constexpr byte operator^(byte l, byte r);

std::to_integer(std::byte b)
std::byte{integer}
```

string_view

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

`string_view` gibt es in vier Variationen:

```
std::string_view    std::basic_string_view<char>
std::wstring_view   std::basic_string_view<wchar_t>
std::u16string_view std::basic_string_view<char16_t>
std::u32string_view std::basic_string_view<char32_t>
```

`string_view`

- lässt sich billig kopieren.
- besitzt nur einen Zeiger auf die Zeichensequenz und seine Länge.

string_view

string_view

- 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.

string_view

```
string str = "   Lot of space";

string_view strView = str;
strView.remove_prefix(min(strView.find_first_not_of(" "), strView.size()));
cout << str << endl           //   Lot of space
     << strView << endl;      // Lot of space

char arr[] = {'L','o','t',' ','o','f',' ',
              's','p','a','c','e','\0', '\0', '\0'};

string_view strView2(arr, sizeof arr);
auto trimPos = strView2.find('\0');

if(trimPos != strView2.npos) strView2.remove_suffix(strView2.size() - trimPos)
cout << arr << ", size=" << sizeof arr << endl                  // Lot of space, size=17
     << strView2 << ", size=" << strView2.size() << endl; // Lot of space, size=14
```

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);
```

string_view

Substrings in konstanter Zeit erzeugen.

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

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

string_view

Performanzvergleich: erzeugen von 10 Millionen Substrings verschiedener Längen (`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

  `std::execution::par_unseq`

  - Parallel und vektorisiert ➡ SIMD

Parallele Algorithmen der STL

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

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

Nicht vektorisiert

```
movslq  -8(%rbp), %rax
movl    vec(,%rax,4), %ecx
addl    $5, %ecx
movslq  -8(%rbp), %rax
movl    %ecx, res(,%rax,4)
```

Vektorisiert

```
movdqa  .LCPI0_0(%rip), %xmm0    # xmm0 = [5,5,5,5]
movdqa  vec(%rip), %xmm1
paddd   %xmm0, %xmm1
movdqa  %xmm1, res(%rip)
paddd   vec+16(%rip), %xmm0
movdqa  %xmm0, res+16(%rip)
xorl    %eax, %eax
```

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

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

Die neuen Algorithmen

- sind für die parallele Ausführung geeignet.
- benötigen für die parallele Ausführung eine assoziative, binäre Operation.

```
std::for_each_n

std::exclusive_scan
std::inclusive_scan

std::transform_exclusive_scan
std::transform_inclusive_scan

std::parallel::reduce
std::parallel::transform_reduce
```

transform:

- wende eine Funktion auf jedes Element des Bereichs an

reduce:

- wende ein binäre Operation sukzessive auf jedes Element des Bereichs an und reduziere dabei den Bereich auf ein Ergebnis

scan:

- wende `reduce` auf den Bereich an und speichere alle Zwischenergebnisse

Alle Operationen starten von links.

Parallele Algorithmen der STL

std::inclusive_scan

```
std::vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::inclusive_scan(std::execution::par,
     intVec.begin(), intVec.end(), intVec.begin(),
     [](int fir, int sec){ return fir * sec; }, 1);

// 1 2 6 24 120 720 5040 40320 362880
```

std::parallel::reduce

```
std::string res = std::parallel::reduce(std::execution::par,
     strVec.begin() + 1, strVec.end(), strVec[0],
     [](auto fir, auto sec){ return fir + "::" + sec; });

// Only::for::testing::purpose
```

Parallele Algorithmen der STL

`std::parallel::transform_reduce`

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

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


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

std::cout << result << std::endl;      //   21
```

Parallele Algorithmen der STL

Gefahr von kritischen Wettläufen und Verklemmungen

```
int numComp= 0;
vector<int> vec={1,3,8,9,10};
sort(parallel::par, vec.begin(), vec.end(),
     [&numComp](int fir, int sec){ numComp++; return fir < sec; }
);
```

➡ Der Zugriff auf **numComp** muss atomar sein.

Parallele Algorithmen der STL

Statische Ausführungsstrategie

```
template <class ForwardIt>
void quicksort(ForwardIt first, ForwardIt last){
  if(first == last) return;
  auto pivot = *next(first, distance(first,last)/2);
  ForwardIt middle1 = partition(parallel::par, first, last,
                    [pivot](const auto& em){ return em < pivot; });
  ForwardIt middle2 = partition(parallel::par, middle1, last,
                    [pivot](const auto& em){ return !(pivot < em); });
  quicksort(first, middle1);
  quicksort(middle2, last);
}
```

~~Parallel Algorithmen der STL~~ => C++20

Dynamische Ausführungsstrategie

```
std::size_t threshold= ...; // some value

template <class ForwardIt>
void quicksort(ForwardIt first, ForwardIt last){
  if(first == last) return;
  std::size_t distance= distance(first, last);
  auto pivot = *next(first, distance/2);
  parallel::execution_policy exec_pol= parallel::par;
  if ( distance < threshold ) exec_pol= parallel_execution::seq;
  ForwardIt middle1 = std::partition(exec_pol, first, last,
                      [pivot](const auto& em){ return em < pivot; });
  ForwardIt middle2 = std::partition(exec_pol, middle1, last,
                     [pivot](const auto& em){ return !(pivot < em); });
  quicksort(first, middle1);
  quicksort(middle2, last);
}
```

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

```
std::cout << "Current path: " << fs::current_path() << std::endl;

std::string dir= "sandbox/a/b";
fs::create_directories(dir);

std::ofstream("sandbox/file1.txt");
fs::path symPath= fs::current_path() /=  "sandbox";
symPath /= "syma";
fs::create_symlink("a", symPath);

std::cout << "fs::is_directory(dir): " << fs::is_directory(dir) << std::endl;
std::cout << "fs::exists(symPath): "  << fs::exists(symPath) << std::endl;
std::cout << "fs::symlink(symPath): " << fs::is_symlink(symPath) << std::endl;


for(auto& p: fs::recursive_directory_iterator("sandbox"))
    std::cout << p << std::endl;
// fs::remove all("sandbox");
```

```
Current path: "/tmp/1469540273.75652"

fs::is_directory(dir): true
fs::exists(symPath): true
fs::symlink(symPath): true

"sandbox/syma"
"sandbox/file1.txt"
"sandbox/a"
"sandbox/a/b"
"sandbox/a/b/c"
```

Die Dateisystem Bibliothek

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

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

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

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

fs::remove(path);
```

```
Write time on server Wed Apr 12 19:47:14 2017
Write time on server Wed Apr 12 19:47:14 2017

Local time on client Wed Apr 12 21:47:14 2017
```

Die Dateisystem Bibliothek

```
void printPerms(fs::perms perm){
  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" : "-")
       << endl;
}
...
printPerms(fs::status("rainer.txt").permissions());

fs::permissions("rainer.txt", fs::perms::add_perms |
    fs::perms::owner_all | fs::perms::group_all);
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);
printPerms(fs::status("rainer.txt").permissions());
```

```
rw-r--r--
rwxrwxr--
r-xr-xr--
```

Die Dateisystem Bibliothek

```
int main(){

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

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

}
```

```
.         Capacity       Free       Available
/     42140499968    12245942272    10957488128
usr   42140499968    12245942272    10957488128
```

std::optional

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

std::optional

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

Spezielle Werte wurden bisher verwendet, um Nicht-Ergebnisse zu 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 typ-sicherer Container, der genau einen beliebige Wert eines 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
}
```

std::variant

std::variant empfängt einen Visitor.

```
std::vector<std::variant<char, long, float, int, double, long long>>
            vecVariant = {5, '2', 5.4, 100ll, 2011l, 3.5f, 2017};

for (auto& v: vecVariant){
  std::visit([](auto&& arg){std::cout << arg << " ";}, v);
}

for (auto& v: vecVariant){
  std::visit([](auto&& arg){std::cout << typeid(arg).name() << " ";}, v);
}

std::common_type<char, long, float, int, double, long long>::type res{};
std::cout << "typeid(res).name(): "  << typeid(res).name() << std::endl;

for (auto& v: vecVariant){
  std::visit([&res](auto&& arg){res+= arg;}, v);
}
std::cout << "res: " << res << std::endl;

for (auto& v: vecVariant){
  std::visit([](auto&& arg){arg *= 2;}, v);
  std::visit([](auto&& arg){std::cout << arg << " ";}, v);
}
```

```
5 2 5.4 100 2011 3.5 2017

int char double __int64 long float int

typeid(res).name(): double

res: 4191.9

10 d 10.8 200 4022 7 4034
```

Was ich noch sagen wollte

- Das verbesserte Interface der assoziativen Container

```
std::map<int, std::string> ordMap, ordMap2;
```

  - Elemente zu einer `std::map` hinzufügen

```
ordMap.try_emplace(3, 3, 'c');
ordMap.insert_or_assign(5, std::string(3,'d'));
```

  - Zwei Maps mergen

```
ordMap.merge(ordMap2);
```

  - Elemente zwischen Maps verschieben

```
auto nodeHandle = ordMap2.extract(3);
ordMap.insert(std::move(nodeHandle));
```

  - Den Schlüssel eins Schlüssel/Wert Paares ändern

```
auto nodeHandle = ordMap.extract(3);
nodeHandle.key() = 5;
ordMap.insert(std::move(nodeHandle));
```

Was ich noch sagen wollte

- `std::optional`, `std::any` und `std::variant` können in-place erzeugt werden.

```
optional<string> opt1(std::in_place, "C++17");
optional<string> opt2(std::in_place,5, 'C')
optional<string> opt3(std::in_place, {'C', '+', '+', '1', '7'});
```

- Vereinheitlichter Zugriff auf Container
  - `std::size:` gibt die Größe eines Containers zurück.
  - `std::empty:` gibt zurück, ob ein Container leer ist.
  - `std::data:` gibt einen Zeiger auf die Daten des Containers zurück.

C++17 auf einen Blick

Kernsprache

- Fold Expressions
- constexpr if
- `if` und `switch` mit Initialisieren
- Strukturierte Bindung
- Inline Variablen
- Automatische Typableitung von Klassen-Templates und Nicht-Typ Template-Parametern
- Kopieren vermeiden
- `auto_ptr` und Trigraphen entfernt

Bibliothek

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

Groß oder Klein

Advanced Developers Conference
Development for Professionals!

Meine Blogs

www.grimm-jaud.de [De]
www.ModernesCpp.com [En]

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de