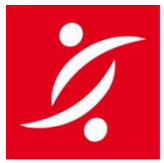

integrata
cegos

Moderne Softwareentwicklung mit C++11 und C++14

Seminarinhalte

- Kapitel 01 - Neuerungen im Sprachkern
  - rvalue Referenzen, move Semantik, automatische Typbestimmung, Initialisierer-Listen, einheitliche Initialisierung, streng typisierte Aufzählungen, bereichsbasierte for-Schleife, Lambda Funktionen, nullptr
- Kapitel 02 - Neuerungen in der Templateprogrammierung
  - Automatische Typdeklaration, externe Template Instanziierung, Variadic Templates, Referenz-Wrapper, move-Funktion, forward Funktion
- Kapitel 03 - Neuerungen in der Standardbibliothek
  - unique_ptr Klasse, shared_ptr Klasse, weak_ptr Klasse, bind Funktion, function – Klasse, Hash-basierte Container, Zeitpunkte und Zeiträume, Reguläre Ausdrücke
- Kapitel 04 - Unterstützung von Multithreading durch die Standardbibliothek
  - Threads, unique_lock Klasse, Mutexes, Mehrfache Sperren, Futures, async-Funktion, Atomare Operationen, Threadlokale Daten
- Kapitel 05 - Neue Eigenschaften der Sprachversion C++14
  - Auto return types, Generic Lambdas, das [deprecated] Attribut, binäre Literale, Digit Separators, Sized Deallocation, Ausblick auf C++17

Inhalt

integrata cegos

1

NEUERUNGEN IM SPRACHKERN

Übersicht Kapitel 01

integrata cegos

- nullptr
- Initialisierer-Listen
- Einheitliche Initialisierung
- Bereichsbasierte for-Schleife
- Automatische Typbestimmung
- rvalue Referenzen, Move Semantik
- Typed Enums
- Lambda Expressions

Keyword nullptr

integrata cegos

- Anstelle von `0` oder `NULL` sollte `nullptr` verwendet werden, um Nullpointer zu kennzeichnen
  - `Container* c = nullptr;`
  - `Container  c = nullptr; // Compilefehler`
- Damit werden Aufrufe überladener Funktionen eindeutig:
  - `void f(Container* p) { cout << "Pointer" << endl; }`
  - `void f(int p)  { cout << "Integer" << endl; }`
- Aufrufe mit Argument „Nullpointer“, bzw. 0
  - `f(nullptr); // Pointer`
  - `f(0);       // Integer`
  - `f(NULL);    // Integer`
- Hinweis: nullptr ist ein Literal vom Typ std::nullptr_t
  - Konversion std::nullptr_t -> int nicht möglich (s.o.)
  - Konversion std::nullptr_t -> bool möglich
    - `if (nullptr) {…} entspricht if (false)`

Einheitliche Objektinitialisierung mit { }

- mit { } : Bessere Unterscheidung zw. Initialisierung und Zuweisung
  - `int i{ 2 };`
  - `std::vector<int> v{ 1,2,3 };`
- Vorsicht bei std::vector:
  - `std::vector<int> v1{ 10,20 }; // Elemente 10 und 20`
  - `std::vector<int> v2( 10,20 ); // 10 mal Element 20`

Einheitliche Objektinitialisierung mit { }

integrata cegos

- Einsatz von std::initializer_list im Konstruktor eigener Klassen:
  - `Container() { cout << "Default ctor" << endl; }`
  - `Container(int i) { cout << "one int ctor" << endl; };`
  - `Container(std::initializer_list<int> lst) : elem{ new int[lst.size()] }, size{ lst.size() }`
- Konstruktoraufrufe für Klasse `Container`
  - `Container c1{ 1, 2, 3 }; // initializer_list ctor`
  - `Container c2;  // default ctor`
  - `Container c3{};  // default ctor`
  - `Container c4();  // Dekl. der Funktion "c4", Rückgabetyp Container`
  - `Container c5(1); // one int ctor`
  - `Container c6{ 1 }; // initializer_list ctor -> hat Vorrang vor one int ctor`
  - `double d;`
  - `Container c7{ d }; // Compile Fehler: invalid narrowing conversion`
  - `Container c8 (d); // one int ctor: cast double->int`

Bereichsbasierte for-Schleifen

integrata cegos

- Range-based for loops : foreach Schleife in C++

```
vector<string> strings{ "Alpha", "Beta", "Gamma" };
```

- Vor C++11:

```
for (vector<string>::iterator it = strings.begin();
     it < strings.end(); it++) {
     cout << *it << " ";
}
```

- Seit C++11:

```
for (string s : strings) {
     cout << s << " ";
}
```

- Funktioniert mit jeglichen Objektsequenzen, also auch mit Arrays

```
int numbers[3] = { 0, 1, 2 };
for (int i : numbers) {
     cout << i << " ";
}
```

Automatische Typableitung mit auto

integrata cegos

- Schlüsselwort `auto` :
  - veranlasst den Compiler, den Typ einer Variablen aus ihrem Initialisierungsausdruck abzuleiten
    - `auto a = 123;`
    - `auto a; // error: cannot deduce 'auto' type(initializer required)`
- Vorteile
  - erzwingt Initialisierung von Variablen
  - erleichtert Refactoring, denn eine Typänderung an einer Stelle wird so automatisch durch den gesamten Code propagiert
  - erspart Schreibarbeit (insbesondere in Templates)
- Nachteile
  - Code wird u.U. weniger aussagekräftig, da explizite Typinformation fehlt

Automatische Typableitung mit auto

- Schleife über Collection ohne auto (in template)

```
for (typename C::const_iterator it = col.begin();
     it < col.end(); ++it) {
      typename iterator_traits<C::const_iterator>::value_type
      value = *it;
      cout << value << endl;
}
```

- (Bereichsbasierte) Schleife über Collection mit auto (in template)

```
for (auto value : col) {
     cout << value << endl;
}
```

rvalue Referenzen

- Herkömmliche Referenz
  - Syntax: T&
  - Semantik: konstanter Pointer auf Objekt vom Typ T
    - Zusatz: Adresse von Objekt kann ermittelt werden
  - `string text = "ABC";`
  - `string& ref = text; // Adresse von text kann ermittelt werden`

- rvalue Referenz
  - Syntax: T&&
  - Semantik: konstanter Pointer auf Objekt vom Typ T
    - Zusatz: Adresse von Objekt kann nicht ermittelt werden (temporäres Objekt)
  - `string&& textRValueRef2 = "xxx"; // geht`
  - `string&& textRValueRef1 = text; // geht nicht`
    - "error: an rvalue reference cannot be bound to an lvalue"

Einsatz von rvalue References : Move Semantik

integrata cegos

- Sei folgende Container Klasse gegeben (Auszug):

```
class Container {
private: int* elem = nullptr; int next = 0; size_t size;
public:
  // copy Konstruktor mit herkömmlicher Referenz
  Container(const Container& a) :
  elem{new int[a.size]},size{a.size},next{a.next}
  {
    for (size_t i = 0; i < size; i++) {
      elem[i] = a.elem[i]; // kopiere a.elem nach this.elem
    }
  }
  // move Konstruktor mit rvalue Reference
  Container(Container&& a) :
  elem{a.elem},size{a.size},next{a.next}
  { } // kein Kopieren nötig, this.elem = a.elem
```

Einsatz von rvalue References : Move Semantik

integrata cegos

- Fallstudie : Konstruktion eines Objektes aus Funktionsrückgabewert

```
Container createContainer()
{
  Container res{ 9, 9, 9, 9, 9 };
  return res;
}
Container c3{ 0, 0, 0, 0, 0 };
```

- Ohne move Konstruktor in Klasse `Container`
  - `// erst copy Konstruktor für temporäres Objekt, dann Zuweisung`
  - `c3 = createContainer(); // temporäres Objekt ist nicht mehr vorhanden`
- Mit move Konstruktor in Klasse `Container`
  - `// erst move Konstruktor für temporäres Objekt, dann Zuweisung`
  - `c3 = createContainer(); // temporäres Objekt ist nicht mehr vorhanden`

-> move Konstruktor erspart das Kopieren eines ohnehin temporären Objekts

Einsatz von rvalue References : Move Semantik

integrata cegos

- In welchen Szenarien wird ein Objekt kopiert oder gemoved?
  - als Zuweisungsziel
  - als Konstruktor-Argument (Initialisierungsobjekt)
  - als Funktionsargument
  - als Rückgabewert einer Funktion
  - als Exception
- Der Einsatz von Move-Konstruktoren, bzw. Move-Zuweisung verbessert u.U. die Performance in o.g. Situationen
- Vgl. auch Beispielcode zum Thema rvalue References, Move Semantik

Typed enums

integrata cegos

- Vor C++11 : herkömmliche enums
  - `enum playerstate   { playing, stopped };`
  - `enum recorderstate { recording, stopped };`
  - Datentyp der enum values : int, also ist folgendes möglich:
    - `playerstate s1 = playerstate::playing;`
    - `if (s1 == playing) {}; // unqualifizierter Zugriff`
    - `int s2 = recorderstate::stopped; // Zuweisung an int Variable`
- Ab C++11 : class enums (typed enums)
  - Gleichnamige enum values möglich
    - `enum class playerstate   {playing,stopped};`
    - `enum class recorderstate {recording, stopped };`
  - Strengere Typüberprüfung
    - `// nur qualifizierter Zugriff möglich`
    - `if (s1 == playerstate::playing) {};`
    - `// Zuweisung nur an enum Typ möglich`
    - `recorderstate s2 = recorderstate::stopped;`

Lambda Expressions I

- Lambda expression = anonymes Funktionsobjekt
- Vor C++11 realisiert mit Hilfe einer Funktionsklasse (aka functor)
- Funktionsklasse: Klasse mit überladenem function call operator ()
- Beispiel:

```
class Find {
private:
  const int valueToFind;
public:
  Find(const int v) : valueToFind(v) {};
  // (function) call operator
  bool operator()(const int element) const {
    return element == valueToFind;
  }
}
```

Lambda Expressions II

integrata cegos

- Die Funktion Container::find mit Funktionsparameter predicate
  - Parameter predicate ist eine Funktion der Form bool f(int)

```
template <typename P>
int find(P predicate)
{
  for (size_t i = 0; i < size; i++) {
    if (predicate(elem[i])) {
      return elem[i];
    }
  }
  throw std::exception("not found");
}
```

- Aufruf von find mit Functor vom Typ Find (Parameter für Suchalgorithmus)

```
Container c1{ 1, 2, 3 };
Find predicate{ 2 };
c1.find(predicate);
```

Lambda Expressions III

- Alternative ab C++11: Aufruf von Container::find mit Lambda Expression

```
int valueToFind = 2;
c1.find([valueToFind](int element){ return element==valueToFind; });
```

- allgemeine Form einer C++ Lambda Expression : `[](){}`
- `[]` -> capture list: enthält die Variablen aus der Umgebung, die für die Lambda Funktion sichtbar sind
- `()` -> parameter list: enthält die Parameter, mit denen die Lambda Funktion (hier von Container::find()) aufgerufen wird
- `{}` -> Funktionsrumpf: wird beim Aufruf ausgeführt
- Das aus einer Lambda Expression erzeugte Objekt zur Übergabe an einen Algorithmus heißt *closure object*

Lambda Expressions IV

integrata cegos

- Die capture list näher betrachtet:
- `[](){};`
  - `-> Keine captured variables stehen zur Verfügung`
- `[&](){};`
  - `-> Alle captured Variables stehen als Referenzen zur Verfügung`
- `[=](){};`
  - `-> Alle captured Variables stehen als Kopien zur Verfügung`
- `[&x](){};`
  - `-> Die captured Variable x steht als Referenz zur Verfügung`
- `[x](){}; oder ab C++14: [x = x](){};`
  - `-> Die captured Variable x steht als Kopie zur Verfügung`
- `[x,&y](){}; oder ab C++14: [x = x](){};`
  - `-> Die captured Variables x und y stehen als Kopie, bzw. Referenz zur Verfügung`

Lambda Expressions V

- Zusatz: Unterschied function pointer <-> lambda expression
  - `// Definition der Suchfunktion`
  - `bool find2Function(int element)  { return element == 2;}`
  - `// Definition und Verwendung des function pointers`
  - `bool(*findFunction)(int) = &find2Function;`
  - `c1.find(findFunction)`
- im Gegensatz zu Lambda und Function Object kein Zugriff auf die Umgebungsvariablen möglich
- d.h. für jedes Suchprädikat müsste ein eigene Funktion erstellt werden
- hier: find3Function, find4Function etc ...
- daher für diesen Zweck ungeeignet

Ab C++14: Generic Lambdas

integrata cegos

- Anstelle der Template Variante

```
template<typename T>
struct Lambda
{
  T valueToFind;
  Lambda(T v) : valueToFind{ v } {} ;
  auto operator()(T element) const
      { return element == valueToFind; }
};
auto lambda = Lambda<int>{ 2 };
```

- kann jetzt geschrieben werden:

```
auto lambda = [valueToFind](auto e){ return e == valueToFind; };
```

integrata cegos

2

NEUERUNGEN IN DER TEMPLATEPROGRAMMIERUNG

Übersicht Kapitel 02 Templateprogrammierung

integrata cegos

- Automatische Typdeklaration
- Externe Template Instanziierung
- Variadic Templates
- Referenz-Wrapper
- move-Funktion
- forward Funktion

Automatische Typdeklaration mit decltype

integrata cegos

- neu in C++11: declytpe = "declared type" eines Namens oder Ausdrucks
  - Vorgänger : typeof(expression) in GCC Erweiterungen oder Boost (war nie Teil des Standards)

```
template <typename T1, typename T2>
auto Add(T1 t1, T2 t2) -> decltype(t1 + t2) {
  return t1 + t2;
}
```

- auto in Kombination mit decltype:

  -> vom Compiler abgeleiteter Rückgabetyp der Funktion Add
- Typischer Anwendungsfall: der Rückgabetyp einer Funktion ist abhängig vom Typ ihrer Parameter
- Die Syntax "auto <name>(<param-list>) -> decltype(expr|name)" veranlasst den Compiler, bei der Template-Instanziierung, anstelle von "auto" den Typnamen des Ausdrucks (hier: t1+t2) oder Variablennamens einzusetzen

Ab C++14: Rückgabetyp auto

integrata cegos

- C++11 "trailing return type" syntax ist ab C++14 nicht mehr nötig
  - aus

```
template <typename T1, typename T2>
auto Add(T1 t1, T2 t2) -> decltype(t1 + t2) {
  return t1 + t2;
}
```

  - wird

```
template <typename T1, typename T2>
auto Add(T1 t1, T2 t2) {
  return t1 + t2;
}
```

  - Der Compiler leitet den Rückgabetyp aus der Funktionsimplementierung ab, d.h. aus dem Ausdruck oder Variablennamen nach `return`

Externe/Explizite Template Instanziierung I

integrata cegos

- Einmalige Instanziierung eines Templates innerhalb eines Projekts
  - Datei Container.h

```
template <typename T>
class Container
{
```

  - Datei IntContainer.h

```
#include "Container.h"
template class Container<int>;
```

    - hier wird das Class Template Container genau einmal als Container<int> instanziiert, also vom Compiler eine Klasse Container mit T = int generiert.
    - Jede Datei, die IntContainer.h inkludiert, kann damit auf dieselbe Instanziierung zugreifen.
    - Die wiederholte Instanziierung für jede Compilation Unit entfällt damit.

Externe / Explizite Template Instanziierung II

integrata cegos

- Verwendung einer expliziten Template Instanziierung
  - Datei Main.cpp

```
#include "IntContainer.h"
void process(Container<int> c) {
     c.printElements();
}
```

  - Hier wird "IntContainer.h" anstelle von "Container.h" inkludiert, um ein bereits passend instanziiertes Class Template zu erhalten

Variadic Templates I

- Ein Variadic Template ist ein Template mit beliebig langer Typliste
  - Vgl. Ellipsis: Funktion mit beliebig langer Parameterliste
  - z.B. printf aus stdio.h :
    - `int printf ( const char * format, ... );`
  - Beispiel Variadic Template : Template function f für beliebig viele Typparameter

```
template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
      // Verwendung der Typparameter T und Tail siehe nächste Folie
}
```

Variadic Templates II

- Template Function f

```
template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
     g(head); // beliebige Aktion für ersten Parameter
     f(tail...); // rekursiver Aufruf für den Rest
}
```

- Leere Funktion f für die leere Parameterliste -> Abbruch der Rekursion

```
void f() {};
```

- Beliebige Template Function g

```
template <typename T>
void g(T x)
{
 cout << x << " ";
}
```

Variadic Templates III

integrata cegos

- Verwendung der Variadic Template Function f

```
f(1, "ALPHA", 1.3, 2);
f("BETA",1.2,'x', 23.4);
```

Variadic Templates IV

integrata cegos

- Beispiele für Variadic Templates aus der STL
  - Zusätzlich zu

```
template<class _Ty1,class _Ty2>
struct pair
```

  - ab C++11 auch

```
template<class...>
class tuple;
```

  - Menge beliebiger Objekte
  - Anstelle von

```
template <class Arg1, class Result> struct unary_function
template <class Arg1,class Arg2,class Result> struct
binary_function
```

  - ab C++11:

```
template<class _Rx, class... _Types>
class function
```

    - Datentyp für Funktionen mit beliebiger Signatur -> vgl. „functions are objects“ in funktionalen Sprachen

std::reference_wrapper I

- std::reference_wrapper ist ein class template aus der STL
- Mit einem std::reference_wrapper kann aus einer Referenz ein kopierbares und zuweisbares Objekt erzeugt werden
- Damit können Referenzen in (STL) Containerklassen gespeichert werden
- Beispiel: siehe nächste Folien

std::reference_wrapper II

integrata cegos

- Beispiel std::vector<string> ohne reference_wrapper

```
// vector von Objekten
vector<string> namen;
string name {"Breitner"};
// push_back ruft Copy-Konstruktor von std:string
namen.push_back(name);
// ändere Namen in Vektor
namen[0].insert(0, "X");
cout << "Name in vector : " << namen[0] << endl;
//  Original Name bleibt unverändert
cout << "name : " << name << endl;
-> name : Breitner
```

reference_wrapper III

integrata cegos

- Beispiel std::vector mit reference_wrapper<string>

```
vector<reference_wrapper<string>> namenref;
// std::ref erzeugt Referenz aus Objekt
namenref.push_back(std::ref(name));
// ändere Namen in ref-Vektor
namenref[0].get().insert(0, "X");
cout << "Name in vector : " << namenref[0].get() << endl;
// Original Name ändert sich
cout << "name : " << name << endl;
-> name : XBreitner
```

Vorüberlegung: Wofür stehen *lvalue* und *rvalue* ?

integrata cegos

- lvalue: Objekt (=Speicherbereich) mit Namen
  - int i = 1;
  - Container c { 1, 2, 3};
  - void f(Container& param);
  - Kann auf der linken Seite einer Zuweisung stehen (Ausnahme const lvalue)
- rvalue: Objekt ohne Namen
  - f(7); // Konstante
  - f(Container {1, 2, 3}); // anonymes Objekt
  - f(g()); // Returnwert einer Funktion
  - throw My_Exception{message}; // Exception
  - Wird im aktuellen Scope nicht mehr gebraucht, sein Inhalt kann verschoben werden

Die Funktion std::move I

integrata cegos

- std::move wandelt ein benanntes Objekt in einen rvalue
  - Gegeben:
    - `void f(Container&& c) { // … }`
  - Kein std::move zum Aufruf von f nötig
    - `f(Container {1, 2, 3} ); // Objekt hat keinen Namen`
  - std::move zum Aufruf von f nötig
    - `Container c  {1, 2, 3};`
    - `f(std::move(c));  // Objekt hat einen Namen`
- std::move ist also eine Typkonvertierungsfunktion, funktional identisch mit einer cast-operation
  - std::move verschiebt keinen Speicherbereich
  - „it would have been better if move() had been called rval()“ (B. Stroustrup TC++PL 17.5.2)
- std::move signalisiert dem Compiler, dass das Objekt per Move-Semantik konstruiert oder zugewiesen werden *kann*
  - Voraussetzung dafür ist die Existenz von move Konstruktoren und/oder move Zuweisungsoperator

Die Funktion std::move II

- Beispiel:
- seit C++11 existieren 2 Überladungen von std::vector::push_back
  void push_back(const T& value);
  void push_back(T&& value);
- Seien folgende Objekte gegeben:

```
vector<Container> v1;
Container c1{ 1, 2, 3 };
```

- Folgender Aufruf wird an die erste Überladung von push_back gebunden

```
v1.push_back(c1); // c1 ist lvalue
```

  - Aufruf copy Konstruktor von Container
  - Im vector v1 steht eine Kopie von c1
- Um die zweite Überladung aufzurufen, ist std::move nötig

```
v1.push_back(std::move(c1)); // std::move wandelt c1 in rvalue
```

  - Aufruf move Konstruktor von Container
  - Im vector v1 steht der Inhalt von c1, c1 ist danach gültig aber undefiniert

Die Funktion std::forward I

- Die Template Function std::forward weist den Compiler an, ein rvalue Objekt zu erzeugen, *wenn sie mit einem rvalue Argument instanziiert worden ist*
- Wurde std::forward mit einem lvalue Argument instanziiert, erzeugt der Compiler ein lvalue Objekt
  - Verglichen mit std::move ist std::forward also eine bedingte Cast Funktion
  - Genauso wenig wie std::move Speicher verschiebt, leitet std::forward etwas weiter

Die Funktion std::forward II

integrata cegos

- Warum muss ein rvalue in ein rvalue *zurückverwandelt* werden ?
  - Weil ein rvalue Argument durch Bindung an einen Parameternamen einer Funktion automatisch wieder zu einem lvalue wird
- Beispiel
  - Template Funktion mit Rvalue Reference Parameter

```
template <typename T>
void storeInVector(T&& c) {
  using value_type = typename std::remove_reference_t<T>;
  std::vector<value_type> clist;
  clist.push_back(std::forward<T>(c));
}
```

  - Mögliche Aufrufe:

```
storeInVector(c1); // lvalue
storeInVector(Container{ 4, 5, 6 }); // rvalue
```

  - Abhängig davon, ob c lvalue oder rvalue ist, wählt der Compiler für das Template
    - push_back(const T& value) oder
    - push_back(T&& value)

std::forward III

integrata cegos

- Wie würde sich folgende Template Function verhalten ?

```
template <typename T>
void storeInVector(T&& c) {
  using value_type = typename std::remove_reference_t<T>;
  std::vector<value_type> clist;
  clist.push_back(c); // fragwürdig: ohne std::forward(c) !
}
storeInVector(c1); // lvalue
storeInVector(Container{ 4, 5, 6 }); // rvalue
```

- Das Objekt c würde bei jeder Instanziierung vom Compiler als lvalue interpretiert werden
- Es würde immer `push_back( const T& value )` ausgewählt werden
- Es würde immer der Copy-Konstruktor von Container ausgewählt werden

std::forward IV

- Beobachtungen
  - Offenbar kann an den Parameter `c` in `f(T&& c)` auch ein lvalue gebunden werden
  - D.h. eine rvalue reference `T&&` kann in Template(!) Funktionen auch eine lvalue reference sein
    - Vorschläge für bessere Namen:
    - `T&&` ist „universal reference“ (Scott Meyers)
    - `T&&` ist „forwarding reference“ (Quelle unbekannt)
  - Um einen `T&&` Parameter c einer Template Funktion je nach Instanziierung des Templates als rvalue oder lvalue zu verwenden, muss in der Template Funktion `std::forward<T>(c)` verwendet werden
- Zusatz
  - Ein mit `auto&& c2 = c1;` definiertes Objekt ist im obigen Sinne ebenfalls eine „universal reference“

integrata cegos

3

NEUERUNGEN IN DER STANDARDBIBLIOTHEK STL

Übersicht Kapitel 03

- unique_ptr Klasse
- shared_ptr Klasse
- weak_ptr Klasse
- bind Funktion und function - Klasse
- Hash-basierte Container
- Zeitpunkte und Zeiträume
- Reguläre Ausdrücke

Heap Memory Management Templates

integrata cegos

- Heap Memory Management vor C++11:
  - „Naked new“
    - `Container* p = new Container{ 1, 2, 3 }; // Speicher anfordern`
    - `// mit pointer auf Speicher arbeiten`
    - `delete p; // Speicher wieder freigeben`
- Probleme
  - Freigabe wird am Ende des scopes leicht vergessen
  - Pointer wird in anderen scope kopiert -> Verantwortung für Freigabe unklar
  - Führt zu memory leaks
- Ähnliche Problematik auch bei anderen Ressourcen
  - Locks, Sockets, File Handles, Thread Handles
- Ab C++11 neue STL Class Templates
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr

std::unique_ptr

integrata cegos

- Das Class Template unique_ptr kapselt einen herkömmlichen Pointer
  - -> Compiler generiert eine Klasse, die verantwortlich für den rohen Pointer ist

```
unique_ptr<Container> up1{ new Container{ 1, 2, 3 } };
up1.get()->printElements(); // oder auch up1->printElements();
```

  - Wenn up1 nicht mehr gültig ist, wird automatisch ~Container() aufgerufen
  - Dies passiert
    - am Ende des Funktions-Scopes
    - bei einem return aus der Funktion
    - bei einer Exception
- Eine unique_ptr Klasse ist alleiniger Eigentümer des Pointers
  - Daher keine Copy-Semantik für unique_ptr

```
unique_ptr<Container> up2 = up1; // compile time error
```

  - sondern nur Move-Semantik

```
unique_ptr<Container> up2 = std::move(up1);
```

  - Vorsicht: up1.get() liefert jetzt nullptr

std::unique_pointer vs. std::auto_ptr

integrata cegos

- C++98 definierte das class template `auto_ptr`
- `auto_ptr` sorgt ebenfalls für automatischen Aufruf des Destruktors

Aber:

- `auto_ptr` ist seit C++11 deprecated, denn
  - `auto_ptr` unterstützt keine C++11 move-Semantik
  - die `auto_ptr` copy Zuweisung (operator =) ist tatsächlich eine move Zuweisung
  - `auto_ptr` kann deshalb nicht in STL Templates wie `std::vector` oder `std::sort` verwendet werden

-> `unique_ptr` ersetzt `auto_ptr` ab C++11

std::shared_ptr

integrata cegos

- Das Class Template `shared_ptr` kapselt ebenso einen herkömmlichen Pointer

```
shared_ptr<Container> sp1{ new Container{ 1, 2, 3 } };
```

- `shared_ptr` unterstützt copy Semantik
  - mehrere Eigentümer des rohen Pointers sind möglich

```
shared_ptr<Container> sp2 = sp1; // copy Semantik
void storePointer(shared_ptr<Container> sp) {..}
storePointer(sp2); // copy Semantik
```

- wenn kein `shared_ptr` mehr gültig ist, wird automatisch der Destruktor des gekapselten Objekts gerufen
- Die Lebensdauer des gekapselten Objekts ist also weniger gut zu überblicken wie die eines durch `unique_ptr` gekapselten Objekts

std::weak_ptr

integrata cegos

- Das Class Template `weak_ptr` erzeugt aus einem `shared_ptr` einen weiteren „smart pointer“ auf das gekapselte Objekt

```
shared_ptr<Container> sp1{ new Container{ 1, 2, 3 } };
weak_ptr<Container> wp(sp1);
```

- Das gekapselte Objekt wird aber nach wie vor beim Löschen des letzten `shared ptr` gelöscht

```
sp1 = nullptr; // löscht Objekt trotz weak_ptr wp
```

- -> ein `weak_ptr` gehört nicht zu den Eigentümern des Objekts
- -> ein `weak_ptr` kann auf ein nicht mehr existentes Objekt zeigen
- Die Funktion lock versucht, aus einem weak_ptr wieder einen shared_ptr zu gewinnen

```
shared_ptr<Container> sp2 = wp.lock();
if (sp2 == nullptr) {cout << "weak pointer has expired " <<
endl;}
```

- Einsatz von weak_ptr z.B.
  - Realisierung von Object Caches (gecachte Objekte sind weak_ptr)
  - Observer Pattern (Observable hält weak_ptr auf Observer)

Die Funktions Adapter std::bind und std::function

integrata cegos

- Gegeben sei folgende Funktion

```
template <typename T>
bool isLess(T limit, T element) {
  return element<limit;
}
```

- Ein Funktions Adapter erzeugt aus einer Funktion ein aufrufbares Objekt (Functor)
- bind erzeugt aus einem Funktionsnamen und einer Argumentliste ein Funktionsobjekt

```
auto lessFunctor = bind(isLess<int>, 3, placeholders::_1);
```

- function erlaubt die Definition eines typisierten Funktionsobjekts

```
function<bool(int)> lessFunctor = bind(isLess<int>, 3,
placeholders::_1);
```

- Einsatz : callback Funktionen, Übergabe von Algorithmen an Datenstrukturen (Visitor Pattern)
  - Lambda Ausdruck ist u.U. die elegantere Alternative
  - `[limit](int number){return number < limit;}`

Hash-basierte STL Container I

integrata cegos

- Alle hash-basierten Container sind assoziative Container
  - sie speichern also nur keys oder key-value Paare
- Assoziative STL Container die schon vor C++11 existierten:
  - `set`, `mulitset`
  - `map`, `multimap`

    -> automatisch geordnet, i.d.R. als (balanced) tree implementiert

    -> Zeitkomplexität der Suche nach key: O(log(n))
- Vor C++11 existierten keine hash-basierten STL Container
  - nur nicht-standardisierte Templates: hash_(multi)set, hash_(multi)map
- Ab C++11 gibt es
  - `unordered_set`, `unordered_multiset`
  - `unordered_map`, `unordered_multimap`

    -> ungeordnet, als Hashtabelle implementiert

    -> Zeitkomplexität der Suche nach key: i.d.R. O(log(1)),
    also unabhängig von n

Hash-basierte STL Container II

- Beispiel `unordered_set<Item>`
  - User-defined type

```
struct Item {
	int nr;
	string name;
	bool operator==(const Item& a) const {
		return this->nr == a.nr && this->name == a.name ;
	}
};
```

  - Hash-Funktion für user-defined type

```
struct ItemHashFunction {
	size_t operator() (const Item& a) const {
		return hash<int>()(a.nr) ^ hash<string>()(a.name);
	}
};
```

  - Definition und Einsatz eines unordered_set

```
Item i1{ 1,"ALPHA" };
unordered_set<Item, ItemHashFunction> itemset;
itemset.insert(i1);
auto it = itemset.find(i2);
if (it != itemset.end()) { … }
```

Zeitpunkte und Zeiträume I

integrata cegos

- Neu in C++11

```
#include <chrono>
using namespace std::chrono
```

- Clock liefert time_point
  - system_clock (System Uhr, kann u.U. zurückgesetzt werden)

```
system_clock::time_point t0 = system_clock::now();
bzw.
auto                     t1 = system_clock::now();
```

  - steady_clock (kontinuierlich fortschreitende Zeit, konstante Zeit zwischen ticks)
  - high_resolution_clock (kürzeste Zeit zwischen ticks)
- Mit time_point kann gerechnet werden

```
(t1 - t0)
```

- duration bezeichnet einen Zeitraum zwischen 2 timepoints

```
duration_cast<milliseconds>(t1-t0).count() << " ms elapsed\n"
```

Zeitpunkte und Zeiträume II

integrata cegos

- Formatierte Ausgabe mit `to_time_t` und `strftime`

```
auto now = chrono::system_clock::now();
time_t ttp = chrono::system_clock::to_time_t(now);
char str[maxchar];
```

- neu in C++11 sind u.a. die formatting characters %F und %T
- %F = YYYY-MM-DD ISO 6801 %T = HH:MM:SS ISO 6801,

```
strftime(str, sizeof(str),"%F %T", localtime(&ttp));
```

- Alternative mit std::put_time (neu in C++11)

```
auto tm = *std::localtime(&ttp);
stringstream ss;
ss << std::put_time(&tm, "%F %T");
```

Reguläre Ausdrücke I

integrata cegos

- Neu in C++11

```
#include <regex>
```

- Regulärer Ausdruck: Ein String, der eine Menge von Strings spezifiziert

```
regex isodate{ "[0-9]{4}-[0-9]{2}-[0-9]{2}" };
```

- Funktion regex_match: testet String gegen regulären Ausdruck

```
if (regex_match("30-11-2015", isodate)) { … } // false
if (regex_match("30.11.2015", isodate)) { … } // false
if (regex_match("2015-30-11", isodate)) { … } // true
```

  - Variante mit character class \d (any decimal digit)

```
regex isodate2{R"(\d{4}-\d{2}-\d{2})"}; // R erlaubt \ in string
```

Reguläre Ausdrücke II

integrata cegos

- Funktion regex_search: durchsucht stream nach regulärem Ausdruck

```
string input = "ALPHA BETA GAMMA DELTA”;
```

- Suche in `input` nach allen Strings, die mit B oder G anfangen

```
smatch submatches;
regex pattern{ "[B|G]([^ ]*) " }; // ([^ ]*) ist
group/subpattern
while (regex_search(input, submatches, pattern)) {
      cout << " found: full match" <<  submatches[0]
           << " submatch "         <<  submatches[1] << endl;
      input = submatches.suffix().str();
}
```

- `found: Full match BETA submatch ETA`
- `found: Full match GAMMA submatch AMMA`

Reguläre Ausdrücke III

- regex_iterator: durchsucht stream nach regulärem Ausdruck

```
string list = "ALPHA BETA GAMMA";
regex wordpattern{ R"(\s*\w+)" } ;
```

- Suche bis zum Ende

```
for (sregex_iterator it(list.begin(), list.end(), wordpattern);
     it != sregex_iterator{};
     ++it) {
    cout << "found " << (*it)[0] << endl;
}
```

- found ALPHA
- found BETA
- found GAMMA

integrata cegos

4

UNTERSTÜTZUNG VON MULTITHREADING DURCH DIE STANDARDBIBLIOTHEK

Übersicht Kapitel 04

integrata cegos

- Futures und async
- Threads
- unique_lock Klasse
- Mutexes
- Atomare Operationen
- Condition variables
- Threadlokale Daten

std::future und std::async

integrata cegos

- Gegeben: eine Funktion f

```
int f(int number) { //compute result . . .  return result;}
```

- Ziel: Ausführung von f, möglichst ohne den aktuellen Thread zu blockieren

```
future<int> fut = async(f,0);
cout << "waiting for result ... " << endl;
```

  - Die Funktion async startet f mit Parameter 0 und assoziiert ein future Objekt mit f
  - Der aufrufende Thread führt sofort die nächste Zeile (`cout << "waiting ..`) aus

- Den Rückgabewert von f erhält man über das future Objekt

```
cout << "result = " << fut.get() << endl;
```

  - Dabei blockiert future::get() den aufrufenden Thread wenn f noch nicht beendet ist

- Scheinbar führt async die Funktion immer auf einem anderen thread aus
  - Tatsächlich kann die Funktion auch auf dem rufenden thread ausgeführt werden
    - Dann wird f erst bei `fut.get()`aufgerufen (lazy (deferred) evaluation). Die Entscheidung fällt die Implementierung abhängig vom Systemzustand
      - Anzahl CPUs (Kerne), Anzahl aktiver, evtl. gepoolter threads, etc …
    - Mit `async(std::launch::async,f,0)` kann neuer Thread erzwungen werden

Task based vs. Thread based concurrency

integrata cegos

- 2 Ansätze für Nebenläufigkeit in C++
- „Task based“, d. h. mit Hilfe von std::future und std::async() -siehe vorige Folie
  - `auto fut = async(f,0);`
  - `fut.get()`
  - Die Funktion f wird in diesem Zusammenhang „task“ genannt
  - Kommunikation zwischen rufendem und aufgerufenem Thread per call + return
    - Tatsächlich über ein vom Standard nicht näher spezifiziertes Heap Objekt
    - Dieses Objekt heißt *shared state*
  - Entscheidung über Abbildung task -> thread wird Implementierung überlassen
  - Nutzung evtl. vorhandener Threadpools inkl. Loadbalancing
- „Thread based“, d.h. mit Hilfe von std::thread - siehe nächste Folien
  - Low Level Ansatz – i.d.R. mehr Handarbeit nötig

Die Klasse std::thread

integrata cegos

- Bildet Funktionen portierbar auf OS-Threads innerhalb eines Adressraums ab
- Gegeben

```
void f () { // do something }
void g () { // do something }
```

- Funktion f und g auf Thread t1 und t2 starten

```
thread t1 { f };
thread t2 { g };
```

  - Jetzt werden drei Funktionen nebenläufig, u.U. auch echt parallel, abgearbeitet
- Im aktuellen Thread auf Ende der Threads t1 und t2 warten

```
t1.join(); // Wird Destruktor von t1 vor t1.join() aufgerufen:
t2.join(); // -> Programmabbruch
```

Threads stoppen / Exceptions in threads

integrata cegos

- Thread stoppen
  - Ein Thread stoppt, wenn seine Funktion beendet ist
  - Es existiert keine portable Variante, einen Thread ohne dessen Mithilfe zu stoppen
    - std::terminate() stoppt *alle* threads
- Nicht behandelte Exception in einem thread
  - std::terminate() wird aufgerufen
- Vgl. Task (mit std::future)
  - Die Task erzeugt eine Exception (in einem std::promise Objekt)
  - Die Exception kann beim Aufruf von future::get() behandelt werden

```
try {
  cout << "result = " << fut.get() << endl;
} catch (…) { . . . };
```

std:thread: Parameter und Rückgabewerte

integrata cegos

- Parameter an Thread-Funktion übergeben

```
void f (SomeType& byref) { … }
void g (SomeType  byval) { … }
```

  - By reference

```
thread t1 { f,  std::ref(arg) }; // evtl. race condition
```

  - By value

```
thread t2 { g , arg }; // sicher keine race condition wg. Kopie
```

- Rückgabewert einer Thread-Funktion verarbeiten

`SomeType f () { … } ->` Rückgabewert wird ignoriert

  - Also:
    - Rückgabewert via by reference Übergabe (s.o.)
    - Oder shared data

```
void f (SomeType byval, SomeType* shared) { } // evtl. race condition
```

    - Oder `std::future / std::promise` (siehe vorige Folien)

Exkurs: thread mit lambda expression

integrata cegos

- Funktioniert genauso wie mit Funktionen, bzw. Funktoren

```
thread t3{ [&z3](){ while (z3>0) z3--; } };
```

- Kürzere Variante

```
thread t3{ [&]{ while (z3>0) z3--; } };
```

  - Zur Erinnerung
    - `[&]` Capture list -> alle lokalen Variablen stehen by reference zur Verfügung
    - `()` Parameter list -> hier leer, kann weggelassen werden
    - `{}` Funktionsrumpf -> wird vom Thread ausgeführt (hier: t3)

- Alternative Variante

```
thread t3{ [=]()mutable { while (z3 > 0) z3--; } };
```

  - `[=]` by-value capture list
  - `()` muss hier stehen, wg. mutable
  - `mutable` muss hier stehen, weil call operator () per default const ist

Mehrere threads, eine Datenstruktur: std::mutex und std::unique_lock

integrata cegos

- Greifen mehrere threads schreibend auf dieselbe Datenstruktur zu, droht race condition
  - Ausnahme: Datenstruktur ist immutable (vgl. String in Java/C#, funktionale Sprachen)
- Zugriff auf die Datenstruktur muss dann synchronisiert werden
  - Mutual exclusion object (mutex)

```
std::mutex m;
// unique_lock ruft m.lock()
// a) mutex gehört anderem thread: aktueller thread wartet
// b) mutex ist frei: aktueller thread wird Eigentümer
{
  std::unique_lock<mutex> lock{ m };
  // access shared data ...
}
// Destruktor von unique_lock ruft m.unlock(): mutex wieder frei
// wartende threads werden aufgeweckt
```

Alternative zu mutex: std::atomic

- atomic : schützt eine Variable vor konkurrierendem Zugriff
  - Schutz wie mit mutex, aber realisiert mit Hilfe von speziellen Maschinenbefehlen

```
int number;
std::atomic<int> atomicNumber;
number++; // read-modify-write interruptible
atomicNumber++; // read-modify-write atomic
```

- atomic verhindert zudem, dass der Compiler Codezeilen umsortiert

```
// keine mögliche Optimierung, da atomicNumber vom Typ atomic
ist
atomicNumber++;
number++;
```

Kennzeichnung speziellen Speichers: std::volatile

integrata cegos

- volatile: schützt vor Eliminierung von Zeilen zum Zweck der Optimierung

```
// Annahme memory ist spezieller Speicher, z.B. memory-mapped IO
volatile int memory;
memory = 10; // Compiler könnte Zeile eliminieren
memory = 20;
// keine mögliche Optimierung, da int vom Typ volatile ist:
memory = 20;
```

Threadkommunikation: std::condition_variable

integrata cegos

- std::condition_variable blockiert einen thread solange, bis eine beliebige Bedingung erfüllt ist, hält währenddessen aber nicht das schützende mutex

```
condition_variable condition;
queue<message> messages;
mutex m; // protects condition and messages
void produce() {
  while (true) {
        unique_lock<mutex> lock{ m };
        condition.wait(lock, []() { return messages.empty(); }); //
  release m
        // reacquire m and write to queue
        lock.unlock();
        condition.notify_one(); // unblock waiting consumer
void consume() {
  while (true) {
        unique_lock<mutex> lock{ m };
        condition.wait(lock, []() { return !messages.empty(); }); //
  release m
        // reacquire m and read from queue
        lock.unlock();
        condition.notify_one(); // unblock waiting producer
```

Threadlokale Daten

integrata cegos

- Speicherklasse `thread_local`
  - Variable existiert einmal *pro thread*

```
string thread_local username;
void logIn() {
    printf("thread %d logs in with user name %s \n",
    this_thread::get_id(), username.c_str());
}
void performAction() {
    printf("user %s performs action .... \n", username.c_str());
}
void logOut() {
    printf("user %s logs out on thread %d \n", username.c_str(),
    this_thread::get_id());
}
```

  - Destruktor wird zum Ende des threads aufgerufen
  - spart stack-space pro thread
  - keine Synchronisation via lock nötig, u.U. performanter

5

NEUE EIGENSCHAFTEN DER SPRACHVERSION C++14

VERSCHIEDENE ZUSÄTZLICHE NEUERUNGEN/ AUSBLICK C++17

- Rückgabetyp auto und Generic Lambdas
  -> bereits in Kap02 / Kap03 behandelt
- das [deprecated] Attribut
  - kennzeichnet überholte Programmteile

```
[[deprecated("f() has been deprecated. Consider using g() ")]]
void f() { };
```

- Binäre Literale

```
unsigned char b128 = 0b10000000;
```

- Digit Separators

```
long int onemillion = 1'000'000;
```

- Sized Deallocation
  - Objektgröße sz steht delete Operator zur Verfügung

```
struct Point { int x; int y; }
static void operator delete(void* ptr, std::size_t sz) { … }
Point* p = new Point();
delete p; // beim Aufruf wird Objektgröße automatisch übergeben
```

Ausblick C++17

integrata cegos

- Vorschläge für neue Features in Sprachversion C++17
  - Auswahl nach https://isocpp.org/files/papers/D4492.pdf
  - Concepts (Constraints für Template-Typparameter)
    ```
    template<Sortable Cont>
    void sort(Cont& container);
    ```
    -> Der Container Typ Cont muss sortierbar sein
    -> Sorgt für einfachere Fehlermeldungen bei der Template Instanziierung
  - Modules (für schnelleren Compile/Link-Vorgang + bessere Sourcecode Organisation)
  - Ranges (einfachere Bestimmung von Bereichen innerhalb einer Objektmenge)
    - `vector<int> v;`
    - `Statt sort(v.begin(),v.end()) -> sort( v );`
  - Asynchrone IO Operationen
  - Parallele Algorithmen
  - SIMD Vectors (gleiche Operation parallel auf mehreren Datenfeldern)
  - Software Transactional Memory (als Alternative zu lock-based concurrency)

Copyright und Impressum

Integrata Cegos GmbH
Zettachring 4
70567 Stuttgart