Programmieren in C++

C++ Grundlagen

Inhalt

- Geschichte von C++
- Modernes C++ im Überblick
- Compiler Support
- Klassen und Objekte
- Typinferenz (Typdeduktion)
- Konstanten
- Einheitliche Initialisierung
- Initialisierungslisten
- Referenzen in C++
- Parameterübergabe und Rückgabetypen
- Strukturiertes Binden
- Typkonvertierung

C++: Die Anfänge

- 1979
  - „C with Classes“ wird von Bjarne Stroustrup in den AT&T Bell Laboratories entwickelt
  - „C with Classes“ erweitert C um ein Klassenkonzept, welches sich an Simula-67 anlehnt
- 1983
  - Umbenennung in C++
  - viele Erweiterungen
    - virtuelle Funktionen, Überladen von Funktionsnamen und Operatoren, Referenzen, Konstanten, änderbare Freispeicherverwaltung und eine verbesserte Typüberprüfung
- 1985
  - 1. C++ Referenzversion (noch nicht standardisiert)
- 1989
  - 2. Version (noch nicht standardisiert)
  - Erweiterungen
    - Mehrfachvererbung, abstrakte Klassen, statische Elementfunktionen, konstante Elementfunktionen und die Erweiterung des Schutzmodells um protected

C++: Der Weg zum ISO/IEC Standard

- 1990-1998
  - weitere Erweiterungen
    - Templates, Ausnahmen, Namensräume, neuartige Typumwandlungen und boolesche Typen
  - Erweiterung der C-Standardbibliothek um Datenströme und um die Standard Template Library (STL)
  - Standardisierungsverfahren
- 1998
  - Normung der endgültigen Fassung der Sprache C++ (ISO/IEC 14882:1998)
- 2003
  - Nachbesserung der Norm von 1998 (ISO/IEC 14882:2003)
- 2005
  - Technical Report 1 (TR1): Erweiterungen, die 2003 nicht standardisiert wurden
- 2011
  - Einführung des neuen C++-Standards → C++11

Modernes C++

- C++11
  - move semantic
  - smart pointers
  - threading & asynchronism
  - lambda expressions
  - initializer lists
  - range based for loops
  - variadic templates
  - type deduction
  - compile time assertions
- C++14
  - generic lambda expressions
  - user defined literals
  - return type deduction
  - make_unique
- C++17
  - parallel algorithms
  - fold expressions
  - class template deduction
  - file system library
  - variant, optional, …
  - string_view

C++20

- Concepts sind eine Erweiterung von Templates
  - festlegen von semantischen Kategorien für die Menge der zulässigen Datentypen
  - einfachere und ausdruckreichere Nutzung von Templates
- Ranges-Bibliothek
  - Algorithmen der Standard Library direkt auf Container anwenden
  - Algorithmen in Form einer Pipeline verknüpfen
  - Ausführung auf unendlichen Streams
- Module
  - Alternative zu Header-Dateien
  - Trennung von Header- und Sourcecode-Dateien auflösen
  - Präprozessoranweisungen eliminieren
  - Kompilieren beschleunigen
- Coroutinen
  - asynchrone Programmierung
  - kooperatives Multitasking zur eleganten Umsetzung von unendlichen Datenströmen, Event-Schleifen und Pipelines

Ausblick auf C++23

- Erweiterungen der Ranges Bibliothek
  - views::chunk, views::chunk_by
  - views::zip, views::zip_transform
  - views::join_width
  - ranges::to, ranges::iota
  - ranges::shift_right, shift_left
  - pipe support for user-defined range adapters
  - std::generator
- Attribute für Lambdas
- Monadic Interface für std::optional
- std::expected
- Deduktion von this
- std::inout_ptr, std::out_ptr
- constexpr std::unique_ptr
- std::to_underlying
- std::is_scoped_enum
- std::move_only_function
- std::byteswap
- std::stacktrace
- std::size_t literals
- std::unreachable
- std::mdspan
- multidimensional index operator
- Entfernung unnötiger leerer Parameterlisten in Lambda-Ausdrücken

Visual C++ (Visual Studio 2022)

- Übersicht über die Unterstützung des Standards in VS 2022 17.x
  - Übersichtsseite
  - Übersicht über viele andere Compiler

- C++20 Core Language Features
  - fast vollständig unterstützt
- C++20 Library Features
  - fast vollständig unterstützt
- C++23 Core Language Features
  - wenig unterstützt
- C++23 Library Features
  - fast vollständig unterstützt

Entwicklung von Klassen

- Klassische Aufteilung in
  - öffentliche Schnittstelle: h-Datei
  - Implementierung der Methoden: cpp-Datei
- Schnittstellendatei
  - definiert die Klasse (Attribute, Konstruktoren, Methoden, Operatoren, …)
  - kann inline programmierte Prozeduren enthalten
  - kann andere benötigte Schnittstellen inkludieren (#include)
- Implementierungsdatei (mehrere pro Klasse möglich)
  - inkludiert zugehörige Schnittstelle
  - kann weitere benötigte Schnittstellen inkludieren
  - implementiert die in der Schnittstelle beschriebenen Prozeduren
- header-only Software-Bibliotheken (Open Source)
  - der ganze Code wird in hpp- bzw. h-Dateien entwickelt und als Quellcode ausgeliefert

Klassendeklarationen

- Öffentliche Klasse

```
struct Point {
  int m_x, m_y;  // öffentliche Attribute (Members, Instanzvariablen)
  double dist(Point p) const;// in C kann ein struct nur Attribute enthalten
};
```

- Klasse

```
class Person {
  std::string m_name;     // private Attribute (Members, Instanzvariablen)
  int m_age;
public:           // ab hier wird die Sichtbarkeit auf public geändert
  Person(const char name[], int age); // Konstruktor
  std::string getName() const;
  void setAge(int age);
};
```

Klassenimplementierung und -nutzung

- Klasse Point

```
int Point::dist(Point p) const {
    const int dx = p.m_x – m_x;
    const int dy = p.m_y – m_y;
    return hypot(dx, dy);
}
```

- Lokale Instanzen erstellen

```
Point pnt1;     // liegt auf dem Stack
Point pnt2;     // liegt auf dem Stack
```

- Zugriff auf Instanzvariable

```
pnt.m_x = 3;
```

- Aufruf einer Instanzmethode

```
double d = pnt1.dist(pnt2);
```

- Klasse Person

```
Person::Person(const char name[], int age)
    : m_name(name), m_age(age)
{}
std::string Person::getName() const {
    return m_name;
}
```

- Lokale Instanz erstellen

```
Person pers("Peter", 21); // lauf dem Stack
```

- Aufruf von Instanzmethoden

```
pers.setAge(22);
// Stringobjekt wird kopiert (tiefe Kopie)
string s = pers.getName();
```

Automatische Typinferenz

- Schlüsselwort auto
  - bei Variablendefinitionen, wo aus dem Initialisierungswert der Variable der Typ der Variable für den Compiler automatisch ersichtlich ist, kann das Schlüsselwort auto anstatt des konkreten Typs hingeschrieben werden
  - Beispiele

```
auto x = 7;
double f();
auto g = f();
```

- Schlüsselwort decltype
  - decltype(x) ist eine Funktion, welche den Deklarationstyp des Ausdruckes x zurückgibt
  - Beispiele

```
decltype(8) y = 8;
decltype(g) h = 5.5;
```

Schlüsselwort constexpr (1)

- Konstanter Ausdruck
  - ein Ausdruck, dessen Wert bereits zur Kompilationszeit bestimmt wird
  - der Ausdruck darf nur aus Literalen und anderen constexpr Werten bestehen
- Beispiele

```
constexpr size_t Length = 500;
constexpr size_t L2 = Length*Length/4;
constexpr char Grades[] = {'A', 'B', 'C', 'D', 'E', 'F' };
double constexpr Pi = 3.141596;
// nicht erlaubt, weil der Sinus nicht zur Kompilationszeit berechnet werden kann
constexpr double SinPi = sin(Pi);
// Verwendung von konstanten Ausdrücken
char text[Length];
std::array<int, L2>;
```

Schlüsselwort constexpr (2)

- Konstante Funktionen
  - eine Funktion, welche prinzipiell zur Kompilationszeit ausgeführt werden kann und einen constexpr Wert zurückliefert
  - Iteration und Rekursion sind erlaubt
  - die Funktion kann aber auch zur Laufzeit ausgeführt werden
- Beispiele

```
constexpr int sum(int x) {
      int sum = 0;
      for (int i = 0; i <= x; i++) sum += i;
      return sum;
}
constexpr int getFive() { return 2 + 3; }
constexpr int fib(int n) { return (n <= 1) ? n : fib(n-1) + fib(n-2); }
// Ausführung zur Kompilationszeit
constexpr int s = sum(getFive());
constexpr int res = fib(20);
// Ausführung zur Laufzeit
int result = fib(25);
```

Schlüsselwort constexpr (3)

- Konstante Bedingungen
  - eine if-Bedingung, welche zur Kompilationszeit ausgewertet wird
  - Compiler kann nicht erreichbare Code-Blöcke wegoptimieren
- Beispiele

```
constexpr UseFirstPart = false;

if constexpr (UseFirstPart) {
     // do something
} else {
     // do something else
}
```

Nichtveränderbare Speicherzellen

- Schlüsselwort const
  - Unveränderbarkeit: nach Initialisierung nur noch lesender Zugriff
- Beispiele

```
const auto age = pers1.getAge();
vector<int> v = { 1, 2, 3, 4, 5, 6 };
const size_t size = v.size();
size_t strlen(const char s[]) { … }
void print(const string& s) { … }
```

- const darf auch nach dem Typ stehen

```
double const PI = computePi();
auto const PID2 =  PI/2;
```

Vereinheitlichte Initialisierung

```
struct Base { };
struct Derived : public Base {
    int m_member;
    Derived(int a1, int a2) : Derived{a1 + a2} {}
    Derived(int a) : Base{}, m_member{a} {}
};
struct Triple {
    int a, b, c;            // Members sind öffentlich, kein Konstruktor vorhanden
};
Derived obj1{1, 2};         // Alternative: obj1(1, 2)
Derived obj2 = {1, 2};      // = funktioniert nur weil der Konstruktor nicht explizit ist
auto *p = new Derived{1, 2};// Alternative: new Derived(1, 2)
vector<int> vec = {1,2,3,4};// vector bietet einen ctor mit Initialisierungsliste an
Triple t1{};                // kein Konstruktoraufruf, sondern Aggregat-Initialisierung
Triple t2 = {7, 8};         // bei zu wenig Werten werden die restlichen Members
                            // mit 0 initialisiert
```

Initialisierungslisten (1)

- Initialisierungslisten sind ein generischer Typ

```
#include <initializer_list>
struct Tuple {
   int value[];
   Tuple(const initializer_list<int>& v);              // ctor #1
   Tuple(int a, int b, int c);                         // ctor #2
   Tuple(const initializer_list<int>& v, size_t cap); // ctor #3
};
Tuple t1(4, 5, 6);                  // ctor #2 wird verwendet
Tuple t2{1, 2, 3};                  // ctor #1 wird verwendet
Tuple t3{2, 4, 6, 8};               // ctor #1 wird verwendet
Tuple t4{{2, 4, 6}, 3};             // ctor #3 wird verwendet
```

- Randbedingungen
  - wenn die Initialisierungsliste der einzige Parameter ist, kann wie oben gezeigt vorgegangen werden
  - wenn noch weitere Parameter vorhanden sind, dann müssen die geschweiften Klammern verschachtelt werden

Initialisierungslisten (2)

- Als Funktionsargument

  können by-value oder by-reference übergeben werden

```
void print(const initializer_list<int>& ls) {
    for (auto i: ls) cout << i << ' ';
    cout << endl;
}
```

- In Range-For-Schleifen

```
for (auto i :{ 2, 4, 6, 8 }) cout << i << ',';
```

Referenzen (C++)

- Referenzen
  - sind Aliasse für andere Variablen (sog. lvalue)
  - haben keine eigene Repräsentanz im Speicher
  - werden durch ein & gekennzeichnet
  - müssen immer initialisiert werden (Neuinitialisierung ist unmöglich)
  - vereinfachen die effiziente Parameterübergabe (keine Datenkopie)

- Beispiele

```
int k = 2;
int& rk = k;        // rk ist ein Alias für k
rk = 3;             // die Variable k kriegt den Wert 3

Point pnt1;
auto& rx = pnt1.m_x;      // rx is ein Alias für das
                          // Attribut m_x von pnt1
rx = 5;             // Attribut m_x von pnt1 erhält den Wert 5
```

Parameterübergabe

- In welcher Art können Objekte an Methoden übergeben werden?
  - By Value

```
void foo1(int x)            // by value: Daten (auch Zeiger) werden kopiert
```

  - By Reference

```
void foo3(const Person& p)// in: referenzierte Person wird nicht kopiert
void foo4(Person& p)      // in-out: referenzierte Person wird nicht kopiert
                          // kann aber in foo4 verändert werden
```

  - By Pointer

```
void foo5(Point* p)       // good-practice: nur für out-Parameter
                          // verwenden, da beim Aufruf der out-Parameter
                          // gut über den Adressoperator erkennbar ist
```

- Good Practice
  - Datentypen mit weniger oder gleichviel Speicher wie zwei Zeiger werden üblicherweise by value übergeben

Rückgabetypen

- By Value
  - Daten werden in Form eines temporären Objekts zurückkopiert

    ```
    double sqrt(double x)
    Point move(const Point& p, int dx) // Ansatz: Point is immutable
    ```

  - bei grossen Objekten effizientere in-out oder out Parameterübergabe nutzen
- By Reference
  - darf nur verwendet werden, wenn die Referenz auf das zurückgegebene Objekt eine längere Lebensdauer als die Ausführung der Funktion hat

    ```
    Point& Point::move(int dx, int dy) { … return *this; }// Point is mutable
    ```

  - falsche Verwendung (verwendet impliziten Zeiger auf zerstörtes Objekt)

    ```
    Point& createPoint(int x, int y) { Point p(x, y); return p; }
    ```

Strukturiertes Binden von Werten

- Elemente eines Arrays strukturiert binden

```
int arr[3] = { 2, 3, 4 };
auto [a, b, c] = arr;     // Variablen a, b und c erhalten
                          // Kopien der drei Arraywerte
```

- Elemente eines Paars strukturiert binden

```
pair<int, double> p = { 5, 3.14 };
auto [i, d] = p;   // Variable i ist vom Typ int, d vom Typ double

map<string, int> m;
for(const auto& [key, value] : map) {
    cout << key << ',' << value << endl;
}
```

Typkonvertierung im Überblick

- C
  - Syntax: (type)expression
- C++
  - static_cast: normale Typkonvertierung

```
int x = static_cast<int>(2.0);
```

  - dynamic_cast: down-cast in Klassenhierarchie

```
Base *b = new Derived(21);
Derived *d = dynamic_cast<Derived*>(b);
```

  - const_cast: const hinzufügen oder entfernen

```
const Point p;
const_cast<Point&>(p).setX(4);
```

  - reinterpret_cast: keine Compiler-Checks

```
float f = 3.14f;
int bitRepresentation = *reinterpret_cast<int*>(&f);
```