Programmieren in C++

Einführung

Umfrage

- Was erwarten Sie von diesem Modul?
- Was möchten Sie in diesem Modul lernen?
- Was möchten Sie über C++ wissen?

Inhalt

- Übersicht über das Modul
- Literatur
- C++ erweitert C
- Unterschiede zwischen C++ und Java
- C/C++ Quellcode
- Build-Prozess
- Programm-Einstiegspunkt
- Einfache Datentypen
- Sichtbarkeit und Namensräume
- Speicherklassen
- Präprozessor

Übersicht

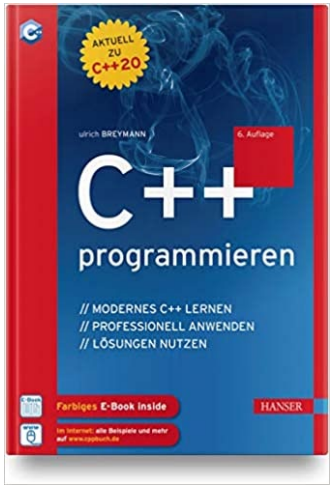
- Leitidee
  - Ausrichtung auf modernes C++ (C++11 bis und mit C++20)
  - Unterschiede zu und Gemeinsamkeiten mit Java aufzeigen
  - praktischer Einsatz von C++ erleben
- Ablauf
  - seminaristischer Unterricht
  - 3 Übungen mit Testatpflicht
  - 2 Klausuren: KW 45, KW 03
- Erforderliche Vorkenntnisse
  - Java-Kenntnisse
  - (C Grundlagen)

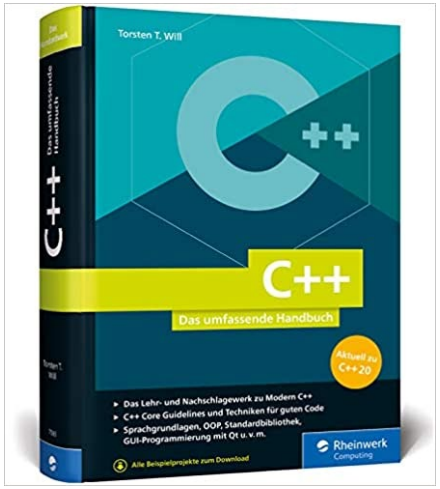
Drehbuch

KW	Datum	Kontaktstudium (KS)	Selbststudium (SS)	KS	SS	Tot
38	17.09.2024 19.09.2024	Einführung von C zu C++	Arbeitsblatt 0 (Visual Studio Tutorial)	3	1	4
39	24.09.2024 26.09.2024	C++ Grundlagen Referenzen	Arbeitsblatt 1	3	1	4
40	01.10.2024 03.10.2024	Zeiger, Arrays, Zeigerarithmetik	Arbeitsblatt 2	3	1	4
41	08.10.2024 10.10.2024	strukturierte Datentypen, Klassen	Arbeitsblatt 3	3	4	7
42	15.10.2024 17.10.2024	Performance-Betrachtungen Move-Semantik	Arbeitsblatt 4	3	4	7
43	22.10.2024 24.10.2024	Operatoren überladen	Arbeitsblatt 5 Abgabe Übung 1	3	4	7
44	29.10.2024 31.10.2024	Vererbung, RTTI Mehrfachvererbung	Arbeitsblatt 6	3	4	7
45	05.11.2024 07.11.2024	Prüfung 1	Prüfungsvorbereitung	2	7	9
46	12.11.2024 14.11.2024	Prüfungsnachbesprechung		3	1	4
47	19.11.2024 21.11.2024	Templates und Variadic Templates	Arbeitsblatt 7 Abgabe Übung 2	3	1	4
48	26.11.2024 28.11.2024	Projektwoche		0	0	0
49	03.12.2024 05.12.2024	Metaprogramming, Concepts	Arbeitsblatt 8	3	1	4
50	10.12.2024 12.12.2024	Streams, Stream-Manipulatoren	Arbeitsblatt 9	3	4	7
51	17.12.2024 19.12.2024	Standardbibliothek: Container, Iteratoren und Algorithmen	Arbeitsblatt 10	3	4	7
02	07.01.2025 09.01.2025	Einsatz der Standardbibliothek und funktionale Programmierung	Arbeitsblatt 11	3	4	7
03	14.01.2025 16.01.2025	Prüfung 2	Abgabe Übung 3 Prüfungsvorbereitung	2	6	8
		Total		43	47	90

Literatur

- C++ programmieren: C++ lernen – professionell anwenden – Lösungen nutzen. Ulrich Breymann. Hanser, 2020.
- C++: Das umfassende Handbuch zu Modern C++. Aktuell zu C++20. Torsten T. Will. Rheinwerk Computing, 2020.
- C++ Schnelleinstieg: Programmieren lernen in 14 Tagen. Einfach und ohne Vorkenntnisse. mitp, 2021.





Links

- Language and Standard Library
  - cppreference.com
  - cplusplus.com
  - Microsoft Docs: C++ Language and Standard Libraries
- Features and Style
  - Overview Modern C++ Features
  - Cpp Core Guidelines
  - Google C++ Style Guide
- Online-Programming and Tools
  - godbolt.org
  - cppinsights.io
  - wandbox.org
  - quick-bench.com
  - codewars.com

C++ erweitert C

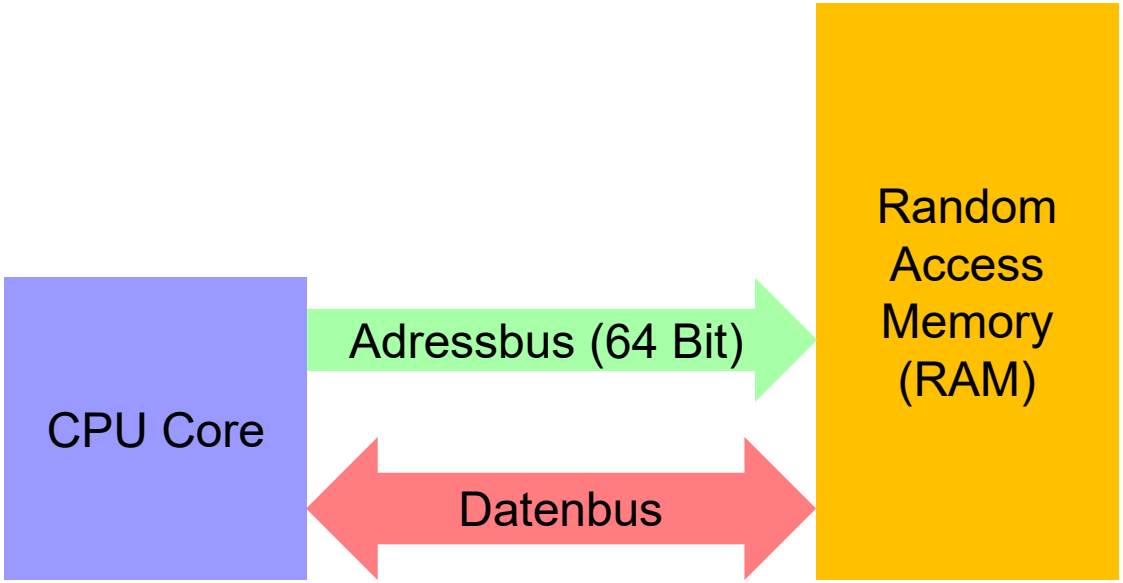
- C
  - kompiliert zu Binärdatei (.obj)
  - Präprozessor
  - primitive Datentypen
  - (schwach) typisiert
  - prozedural
  - Zeiger (Pointer)
  - Records (struct, union)
  - Funktionszeiger
  - C Standardbibliothek
- zusätzlich in C++
  - stark typisiert
  - objektorientiert, funktional
  - Referenzen
  - Klassen, Vererbung, Polymorphie
  - functional, Lambda-Ausdrücke
  - C++ Standardbibliothek
    - oft benötigte Klassen/Funktionen
    - generische Container
    - (parallele) Algorithmen
  - Namensräume
  - Generics (Templates)
  - Exceptions

Unterschiede zwischen C++ und Java

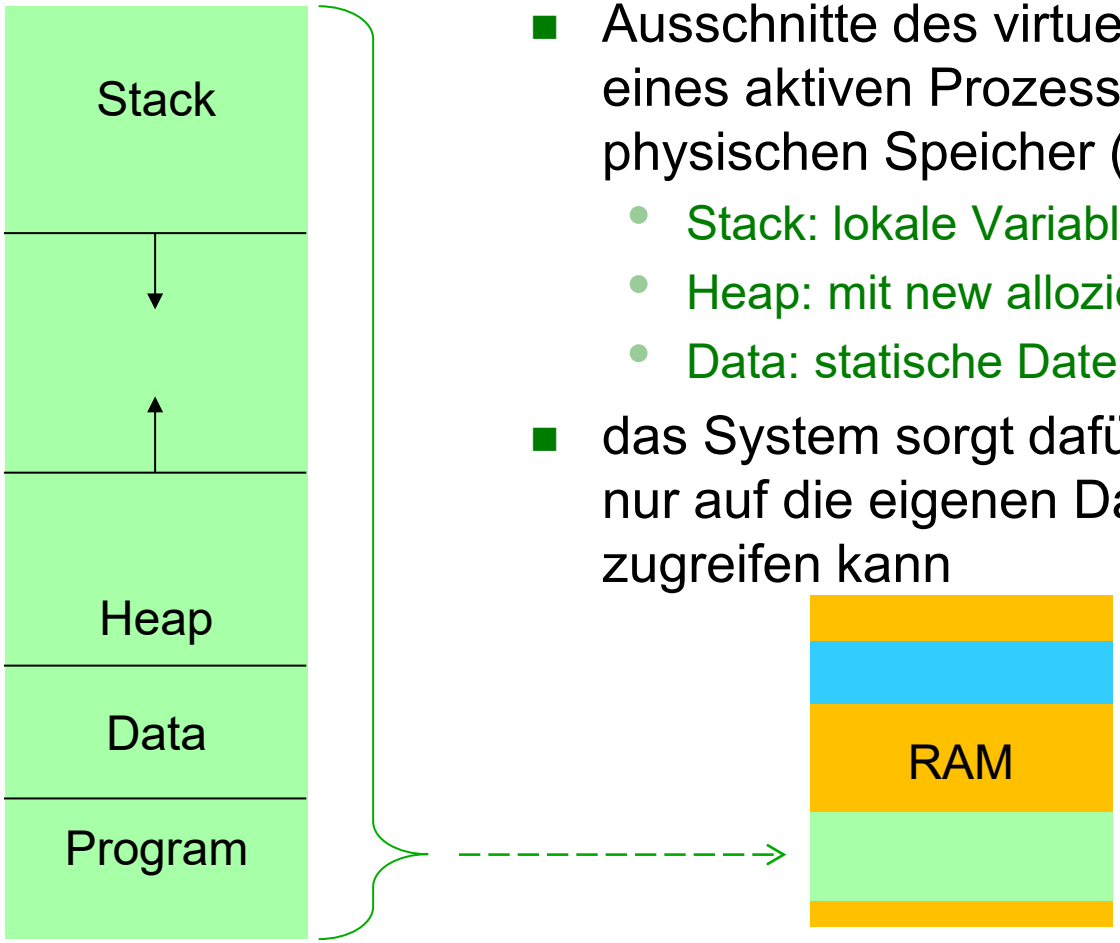
- plattformabhängiger Maschinencode anstatt Bytecode für die VM
  - hohe Performanz, keine VM
  - Optimierung zur Kompilationszeit und nicht zur Laufzeit wie in Java
- C++-Programme können auf unterliegendes System zugreifen
  - sehr hohe Flexibilität (fast alles ist machbar!)
  - System Calls (Betriebssystemfunktionsaufrufe) können verwendet werden (sollte vermieden werden, besser Standardbibliothek verwenden)
  - C/C++ Laufzeitumgebung (wird jedem C/C++-Programm angehängt)
- Flexibleres Speichermanagement
  - Speicheradressen sind sichtbar und können manipuliert werden
  - kein integrierter Garbage Collector, dafür Smart-Pointers mit Reference-Counting, d.h. nicht mehr benötigte Objekte und Arrays werden meistens automatisch gelöscht

System-Architektur

- physischer Speicher (RAM) ist viel kleiner als virtueller Adressraum eines Prozesses
- jedes Byte des Speichers kann adressiert werden
- die Speicheradresse gibt an, von wo Daten gelesen oder wohin Daten gespeichert werden sollen



Memory Mapping



- Ausschnitte des virtuellen Adressraumes eines aktiven Prozesses werden auf den physischen Speicher (RAM) abgebildet
  - Stack: lokale Variablen, Parameter
  - Heap: mit new alloziert
  - Data: statische Daten, fixe Zeichenketten
- das System sorgt dafür, dass ein Prozess nur auf die eigenen Daten im Speicher zugreifen kann

Weitere Unterschiede zu Java

- Flexiblerer Polymorphismus
  - Methoden können bei Bedarf polymorph (virtual) sein
  - Operatoren dürfen überladen werden
  - Mehrfachvererbung ist auch für Klassen erlaubt
- Effizienz vor Sicherheit
  - keine Laufzeit-Checks bei Arrayzugriffen (Release-Version)
  - Arrays und Objekte können auch auf dem Stack angelegt werden
- Unterscheidung zwischen Referenzen und Zeigern
  - Referenzen müssen immer auf eine vorhandene Variable, Objekt oder Array verweisen
  - Zeiger dürfen irgendwohin zeigen, auch in ungültige Speicherbereiche
- keine strikt geschachtelten Namensräume
  - flexiblerer Umgang mit Namensräumen
  - undefinierter = globaler Namensraum
- Trennung zwischen Schnittstelle und Implementierung
  - Schnittstelldateien können flexibel eingesetzt werden
  - für Interfaces gibt es kein spezielles Schlüsselwort

C/C++-Dateien und deren Bedeutung

- *.c
  - C-Quellcode
- *.cpp
  - C++-Quellcode (Kompilationseinheit)
  - Implementierung von Funktionen und Methoden einer Klasse
- *.h
  - C/C++-Header-Dateien
  - wird nicht direkt kompiliert, sondern in eine oder mehrere cpp-Dateien inkludiert (importiert)
  - enthält mehrfach benötigte Definitionen (Klassen, Konstanten)
  - deklariert Klassen, Funktionen und Variablen
  - Verwendung der Standardbibliotheken
    - C-Bibliothek: #include <cstdint>
    - C++-Bibliothek: #include <iostream>
- *.hpp
  - wird hauptsächlich für header-only-Implementierungen verwendet

Schnittstelle und Implementierung

```
// mymath.h
int sum(int a, int b);
```
Deklaration

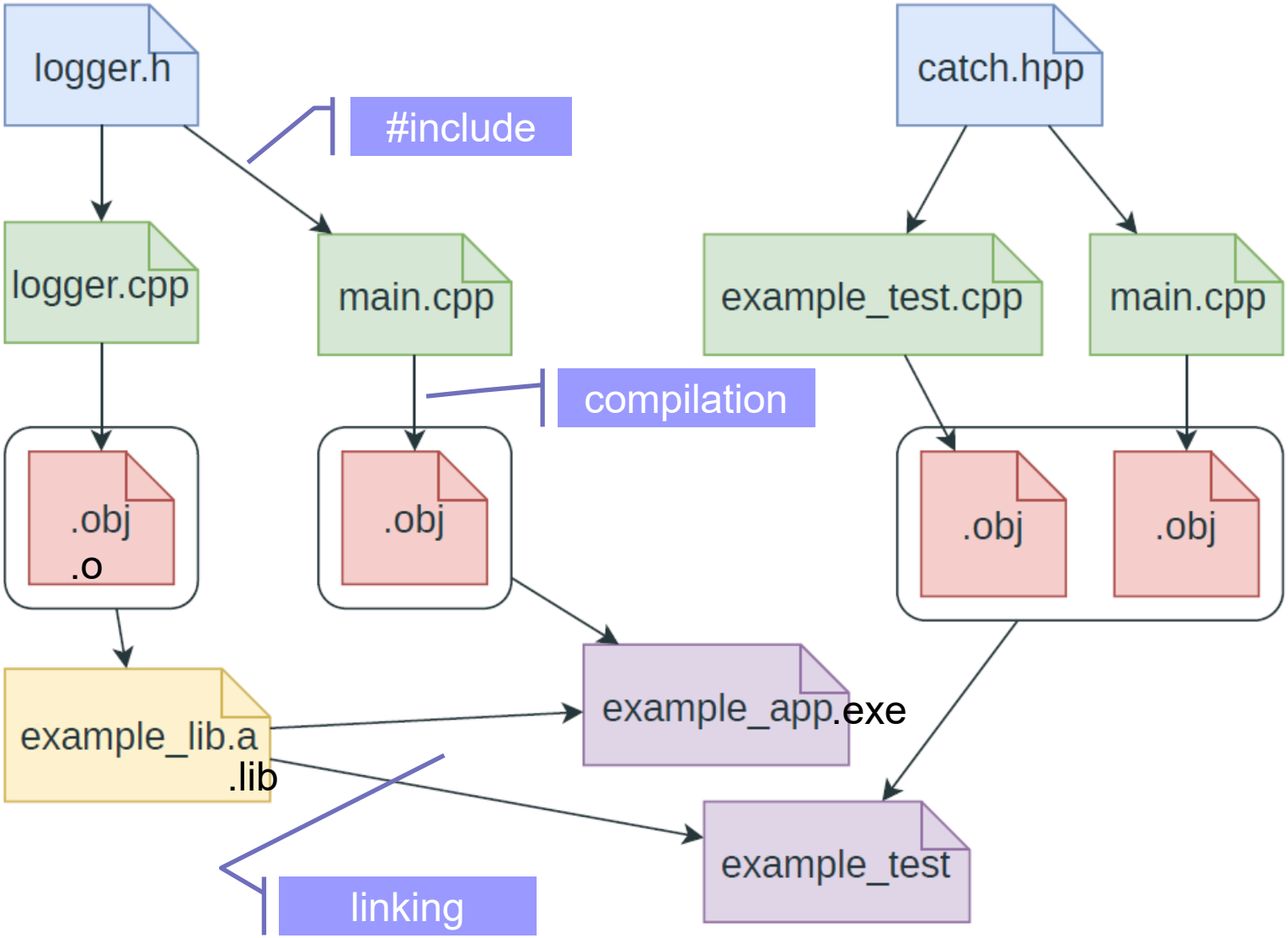
```
// mymath.cpp
int sum(int a, int b) {
   return a + b;
}
```
Implementierung

```
// main.cpp
#include "mymath.h"
int main() {
   int r = sum(5, 7);
}
```
Inkludierung der Schnittstelle

Verwendung

Programmerzeugung

- Präprozessor
  - Programmcode darf Makros enthalten
  - Makros werden unmittelbar vor der Kompilation evaluiert
  - Bsp. bedingte Kompilation
- Compiler (VC++ CL, GCC, clang, intel C++, …)
  - Syntaxüberprüfung des Quellcodes
  - Erzeugung von Objektdateien (Maschinencode mit nicht aufgelösten Verknüpfungen zu anderen Objektdateien, Endung *.obj)
- Linker (Binder)
  - Erzeugung von Bibliotheken oder ausführbaren Programmen aus einzelnen Objektdateien
    - statische Bibliotheken werden zur Kompilationszeit eingebunden (*.lib, lib*.a)
    - dynamische Bibliotheken werden zur Laufzeit geladen (*.dll, lib*.so)
    - ausführbare Programme (*.exe, in Linux keine spezielle Endung)
  - Verknüpfungen zwischen Objektdateien werden aufgelöst
  - Optimierungen (z.B. Entfernung nicht verwendeter Prozeduren/Funktionen) sind möglich


logger.h
#include
catch.hpp
logger.cpp
main.cpp
example_test.cpp
main.cpp
compilation
.obj
.o
.obj
.obj
.obj
example_app.exe
example_lib.a
.lib
example_test
linking

Compiler und Build-System

- Windows
  - empfohlene IDE: Visual Studio (VS)
  - Projektdateien
    - *.sln Solution enthält ein oder mehrere Projekte
    - *.vcxproj Projekt-Einstellungen
  - Dokumentation: «VisualStudioTutorial_2022.pdf»
- alle Plattformen
  - IDE nach Wahl
  - Einsatz von CMake
    - überprüfen, ob cmake installiert ist: cmake –version
    - Projektkonfiguration: CMakeLists.txt
    - CMake Cache erstellen: cmake .
    - Projekt bilden: cmake –build . (oder einfach: make)
  - Dokumentation: Documentation | CMake

One-Pass-Compiler

- Idee
  - die Kompilation einer cpp-Datei verläuft in einem einzigen Ablauf von oben nach unten
  - inkludierte Dateien werden an der Stelle von #include in den Text eingefügt
- Konsequenzen
  - bevor ein Bezeichner (Variable, Klasse usw.) verwendet werden darf, muss er deklariert bzw. definiert werden
    → Deklaration bzw. Definition eines Bezeichners muss vor seiner Benutzung kompiliert werden
  - zyklische Abhängigkeiten müssen durch Vordeklarationen aufgebrochen werden (Stichwort: Klassen- und Funktionsprototypen)

Einstiegspunkt eines Programms

- Main-Methode
  - ausführbare Programme (Executables) benötigen genau einen Einstiegspunkt
    → eine beliebige Objektdatei enthält genau eine main-Funktion
- Beispiel

```
#include <iostream>
using namespace std; // Verwendung des Namensraums std

int main(int argc, char* argv[]) {
   cout << "The program arguments are: " << endl;
   for (int i=0; i < argc; i++) {
      cout << i << ": " << argv[i] << endl;
   }
   return 0;
}
```

Einfache Datentypen

- Grundsatz
  - Speicherbedarf der einfachen Datentypen ist Compiler spezifisch
  - alle ganzzahligen Datentypen (inkl. char) gibt es
    - vorzeichenlos (unsigned) und
    - vorzeichenbehaftet (signed) in der Zweierkomplementdarstellung

- Typischer Speicherbedarf auf 32- und 64-Bit-Plattformen

bool	1 Byte	1 Byte	// kompatibel mit Integer: false ≡ 0
char	1 Byte	1 Byte	
wchar_t	2 Bytes	2 Bytes	// wide character z.B. für Unicode
byte	1 Bytes	1 Byte	
short	2 Bytes	2 Bytes	
int	4 Bytes	4 Bytes	
long	4 Bytes	4 Bytes	// wie long int
long long	8 Bytes	8 Bytes	// wie long long int
float	4 Bytes	4 Bytes	
double	8 Bytes	8 Bytes	// long double kann länger sein
size_t	4 Bytes	8 Bytes	// unsigned, Resultat des sizeof-Operators

Integer mit definiertem Speicherbedarf

- Header
  - #include <cstdint>
- Typen (nur definiert, wenn die Plattform sie anbieten kann)
  - uint8_t 1 Byte
  - int8_t 1 Byte
  - uint16_t 2 Bytes
  - int16_t 2 Bytes
  - uint32_t 4 Bytes
  - int32_t 4 Bytes
  - uint64_t 8 Bytes
  - int64_t 8 Bytes
  - uintptr_t Integer, welcher eine Adresse verlustlos speichern kann
32-Bit-System: 4 Bytes, 64-Bit-System: 8 Bytes

Eigene Typenbezeichner

- Konzept
  - eigene Typenbezeichner für primitive oder strukturierte Typen definieren
- Einsatzzweck
  - Kurzschreibweise einführen
- Schlüsselworte
  - typedef (C/C++)
  - using (C++)
- Beispiele
  - using int32_t = int; // in <cstdint>
  - using uint64_t = unsigned long long; // in <cstdint>
  - using MyType = long double;

Sichtbarkeit von Bezeichnern

- Grundsatz
  - Bezeichner (Variablen, Funktionen, Klassen usw.) müssen vor der Nutzung deklariert werden
  - der Ort der Deklaration bestimmt die Sichtbarkeit des Bezeichners
- Block { … }
  - Blöcke können verschachtelt sein
  - Bezeichner in einem Block auf gleicher Ebene müssen eindeutig sein
  - Bezeichner sind in inneren Blöcken sichtbar
  - ein Bezeichner aus einem äusseren Block kann in einem inneren Block neu deklariert werden und verdeckt dadurch den äusseren Bezeichner
- Beispiel

```
{
   int x = 5;
   {
      int x = 3;
      std::cout << x << std::endl;
   }
   std::cout << x << std::endl;
}
```

Namensraum

- Globaler Namensraum
  - alle Deklarationen ausserhalb irgendeines Blocks gehören zum globalen Namensraum
  - Namen sind überall im Code gültig (Namenskonflikte können auftreten)
  - main-Funktion muss im globalen Namensraum liegen
- Namensraum
  - Gruppieren von Bezeichnern in eigenem Namensräumen
  - dürfen geschachtelt werden
  - Vermeidung von Namenskonflikten
  - Syntax: namespace *identifier* { *named_entities* }
  - Beispiel

```
namespace simpleMath {
    const float PI = 3.141593f;
}
namespace math {
    const double PI = 3.14159265358979323846264338327795;
}
```

Scope-Operator und using

- Grundsatz
  - innerhalb eines Namensraums (inkl. globaler) können Bezeichner ganz normal angesprochen werden
  - wird auf einen Bezeichner in einem anderen Namensraum verwiesen, so kommt der Scope-Operator :: oder using zum Einsatz
    - Bsp: math::Pi oder ::x
- Schlüsselwort using
  - führt einen Namen in der aktuellen Deklarationsregion (Block) ein
  - auf Scope-Operator kann verzichtet werden (vereinfacht die Schreibweise)
  - Beispiel

```
{
   using namespace std;   // führt alle Bezeichner aus std ein
   using math::PI;        // führt nur PI aus math ein
   cout << PI << endl;
}
```

Modulvariablen und Modulmethoden

- Sichtbarkeitsbereich
  - beschränkt auf die Objektdatei
  - jedoch über Methoden- und Klassengrenzen hinweg
- Automatische Nullinitialisierung
- Einsatzgebiet
  - bei nicht-objektorientierter Programmierung als Ersatz von Klassenvariablen und -methoden
  - Alternative zum Heap für grosse Arrays

```
#include <iostream>
#include <cmath>
using namespace std;

static double lastX;

static double foo(double x) {
   double r = lastX + x;
   lastX = x;
   return r;
}

int main() {
   int x = 5;
   cout << "foo(" << x << ") = "
        << foo(x);
}
```

Speicherklassen

- Statischer Speicher
  - globale Variablen, Variablen in einem Namensraum, Modulvariablen und Klassenvariablen
  - Variablen werden automatisch mit 0 initialisiert, falls nicht anderweitig definiert
  - Variablen bleiben während der ganzen Laufzeit des Programms im Speicher
- Automatischer Speicher (Stack)
  - Funktionsparameter, lokale Variablen
  - wird zur Laufzeit auf dem Stack angelegt und beim Verlassen des Blocks automatisch vom Stack entfernt
- Dynamischer Speicher (Heap)
  - Objekte werden zur Laufzeit mit new (C++) oder malloc (C) auf dem Heap alloziert
  - nicht mehr benötigte Objekte müssen mit delete (C++) oder free (C) freigegeben werden

Präprozessor

- #define XYZ, #define MyMacro(arg1, arg2)
  - definiert ein Symbol/Makro XYZ mit oder ohne Parameter
  - Präprozessor löst Symbol auf, d.h. ersetzt es durch Ersetzungstext
- #undef
  - löscht die Definition eines Symbols, d.h. das Symbol ist danach nicht mehr definiert
- #ifdef XYZ, #else und #endif
- #if defined(XYZ) && defined(QR) || defined(UV)
  - bedingte Kompilation: die Kompilation eines Quellcode-Blocks ist abhängig von der Definition eines oder mehrerer logisch verknüpfter Symbole
- Stringizer
  - #define MyMacro(arg) cout << #arg << endl

Präprozessor Beispiel

```
#include <iostream>
using namespace std;

inline int sqr(int x) {
      return x*x;
}
```

```
int main() {
   int k = 0, sum = 0;
   for (int i=1; i<=10; i++) {
   #ifdef _DEBUG
      const int t = sqr(k++);
      clog << t << endl;
      sum += t;
   #else
      sum += sqr(k++);
   #endif
   }
   cout << sum << endl;
}
```