K03

C++ Einführung

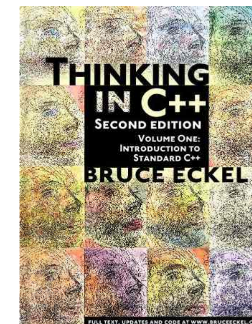
1. Allgemeines
2. Entwicklungsprozess
3. Gültigkeitsbereiche und Namespaces
4. Prozedur-/Funktionsaufrufe, Inlining, Default-Argumente
5. Elementare Typen und Strukturen
6. Vergleich Java/C++

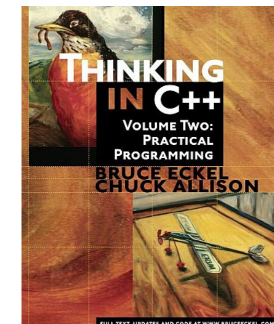
C++ Literatur

- B. Stroustrup: *Die C++ Programmiersprache*. 4. Auflage Addison-Wesley, 2010. ISBN: 3827328233
- B. Stroustrup: Einführung in die Programmierung mit C++ Pearson, 2010. ISBN: 978-3-8689-4005-3

Frei verfügbar:

- B. Eckel: *Thinking in C++, Second Edition*. Volume I/II. Prentice Hall. 2000/2003





Abschreckend – Neugierig machend?

*“C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do *nothing* but keep the C++ programmers out, that in itself would be a huge reason to use C.”**

* Linus Torvalds über C++ im Vergleich zu C, im Kontext von Git:
http://thread.gmane.org/gmane.comp.version-control.git/57643/focus=57918

Warum C++?

- Effiziente systemnahe (maschinennahe) Programmierung
  - Betriebssysteme
  - Eingebettete Systeme
- Darüber hinaus in unterschiedlichsten Anwendungsgebieten benutzt, z.B.:
  - Graphik, 2D & 3D Visualisierung
  - Numerische Berechnungen
  - Server-Anwendungen
  - Spieleprogrammierung
- Hoher Verbreitungsgrad
  - Es existieren sehr viele (freie) Bibliotheken
- C++ kann auch mit anderen Programmiersprachen gekoppelt werden, z.B. Visual Basic, C, Python, MATLAB, Java, Haskell, Go …

Disclaimer: Fokus hier auf "klassischem" C++. C++11/17 nachrangig.

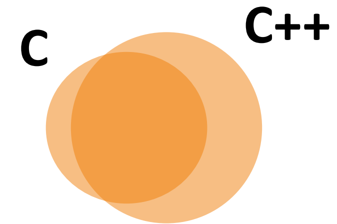
Wann C++ eher nicht?

- Um als Anfänger „mal schnell" die *x*-te Software zur Verwaltung der privaten Musiksammlung zu entwickeln

  → Eine der komplexesten Programmiersprachen.
- Kleine Projekte
- Skripte
- Schnelle Prototypenentwicklung
- Erhöhte Sicherheitsanforderungen – Stichwort *Sandboxing*
  - Bedingt durch die Möglichkeit direkt auf den Speicher zugreifen zu können, ohne das dies überwacht wird*, ist es leicht möglich den Speicher zu korrumpieren oder kompromittierbaren Code zu schreiben, der es erlaubt Schadfunktion einzuschleusen.

* Was aber andererseits heutige Betriebssysteme zumindest im Userspace einschränken durch isolierte Speicherbereiche pro Prozess.

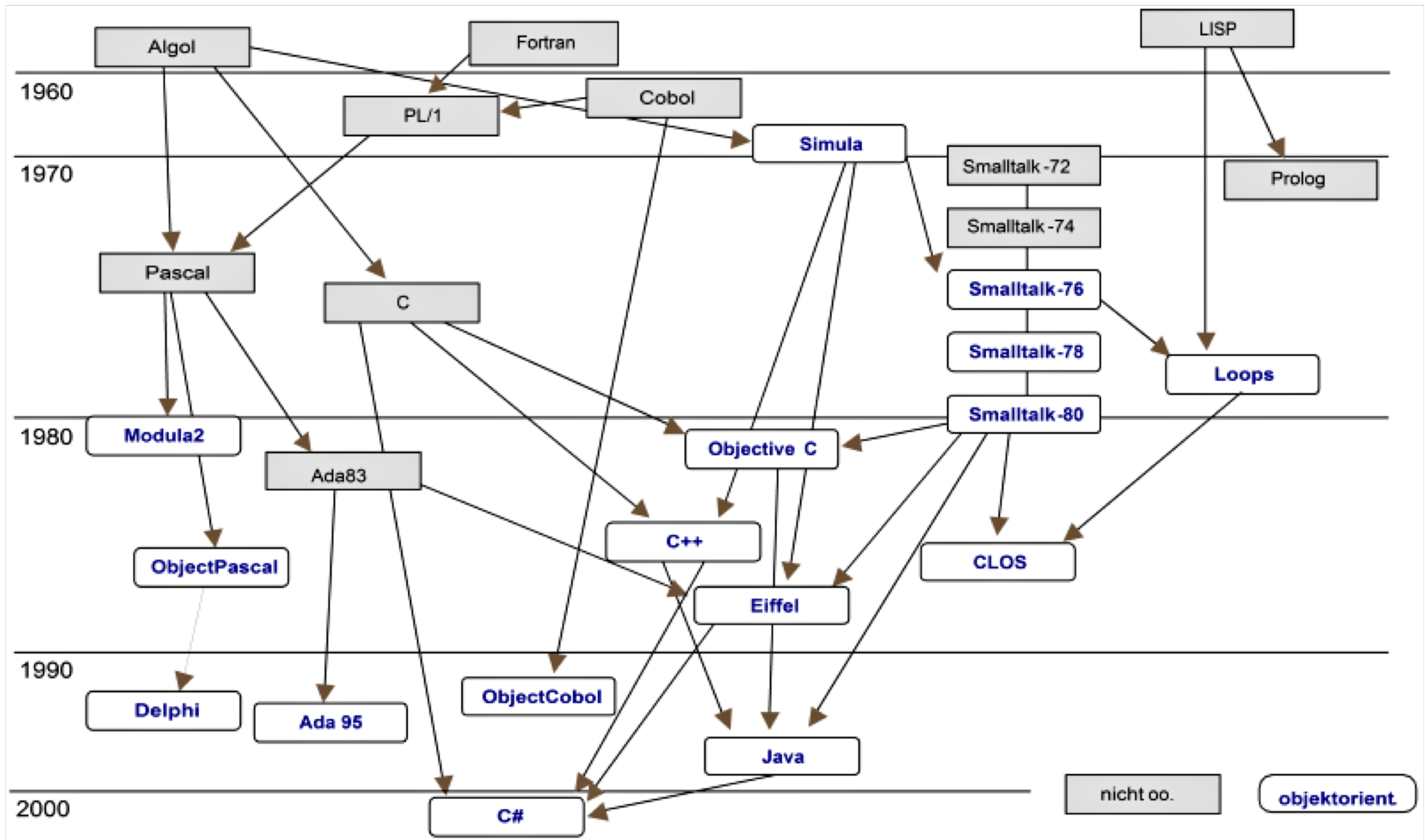
C++ zusammengefasst



- C++ **kein Superset** von C, obwohl beide vieles gemein haben
- Streng typisiert
- Dynamisches als auch statisches Binden
- Modular – Strukturierung des Quellcodes durch:
  - Separate Quellcode-Dateien, Namensbereiche, Klassen, Funktionen/Methoden
- Imperativ, systemnah
- Objektorientiert
  - Kapselung, Polymorphie, Mehrfachvererbung, Unterscheidg. Objektidentität/-gleichheit
- Bietet Möglichkeiten der funktionalen Programmierung
  - Methoden/Funktionen höherer Ordnung durch Funktionszeiger
- Überschreiben von Methoden und Überladen von Operatoren
- Ausnahmen (Exceptions) zur Fehlerbehandlung
- Templates zur generischen Programmierung
- Metaprogrammierung durch Makros (z.B. bedingtes Kompilieren)
- Designziel: zero-overhead abstractions*

* Stroustrup: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

Stammbaum der OO-Sprachen



Quelle: http://upload.wikimedia.org/wikipedia/commons/d/db/Historie.png

Schlüsselwörter in C++ und C

Schlüsselwörter in C **und** C++				
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

Legende: **Schwarz** Syntax, Semantik ähnlich in Java
Braun in der Vorlesung (**nicht**) behandelt

Schlüsselwörter **(nur C++)**				
asm	bool	catch	class	const_cast
delete	dynamic_cast	explicit	false	friend
inline	mutable	namespace	new	operator
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t

Das klassische erste Programm

hello_world.cpp

```
// it does what it does
#include <iostream>
int main()
{
  std::cout << "hello world!\n";
  return 0;
}
```

iostream: benötigt man, um das **cout** (console output) Objekt für die Ausgabe auf der Konsole benutzen zu können.

std::cout ist ein Standard-Stream-Output zur Ausgabe

```
> g++ hello_world.cpp -o hello_world
> ./hello_world
hello world!
```

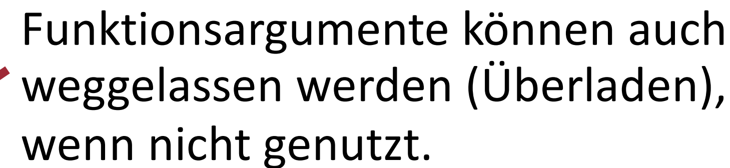
g++ ist der GNU C++ Compiler, der eine C++ Datei kompiliert und ein Executable erzeugt.

Das Programm **hello_world** wird ausgeführt

Zumeist wird ein Makefile benutzt, um den Übersetzungsvorgang zu automatisieren.

Die spezielle **`main`**-Funktion

- Funktion zum Starten des Programms – der Einstiegspunkt
  - Sagt dem Betriebssystem, wo der Einsprung in das Programm stattfindet – nur eine **`main`**-Funktion pro Programm möglich.



test.cpp

```
int main(int argc, char** argv)
{
  int x = 0;
  return x;
}
```

Standard Ein- und Ausgabe

- **cout** wird zur Ausgabe auf die Konsole benutzt:

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

int main() {
     int i = 5;
     cout << "the value of i is " << i << endl;
}
```

- **cin** wird zur Eingabe benutzt:

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

int main() {
     int i;
     cout << "Enter a value for i ";
     cin >> i;
}
```

```
1  // small example
2  /* A first program in C++ */
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << "Welcome ";
8     std::cout << "to C++!\n";
9     return 0; //program ended successfully
10 } // end method main
```

```
Welcome to C++!
```

Kommentare

Mit **/*** und ***/** bzw. durch **//**.

Präprozessor-Direktiven

Zeilen mit **#** sind Präprozessor-Direktiven, also Befehle für den C++ Präprozessor.

#include <iostream> teilt dem Präprozessor mit, dass der Inhalt der Datei **<iostream>** mit in die C++ Datei eingebunden werden soll.

return ist eine Möglichkeit, eine Funktion zu beenden.
return 0 teilt dem Vater-Prozess mit, dass das Programm erfolgreich beendet worden ist.

- Exkurs: Programm kann auch durch **`exit(`*int*`)`** jederzeit beendet werden
  - Bekommt man durch **`#include <stdlib.h>`**
  - Signatur (Prototyp) ist **`void exit(int);`**
  - Parameter ist Rückgabewert für Vater-Prozess (**`0`** = normal exit)
  - Achtung: ruft keine Destruktoren auf; dazu später mehr

Fortsetzung Beispiel

- `std::cout`
  - Standard-Ausgabe Stream-Object
  - “Verbunden” mit der Konsole

  `std::` ist der Namesbereich "namespace" von `cout`

  `std::` kann durch `using std` weggelassen werden
- `<<`
  - Stream-Ausgabe Operator
  - Wert auf der rechten Seite des Operator wird dem Output-Stream übergeben (welcher mit der Konsole verbunden ist)
  - `std::cout << "Welcome to C++!\n";`
- `>>`
  - Stream-Eingabe Operator
  - Zeichen des Input-Stream werden der rechten Seite des Operators zugewiesen
- `\`
  - „Escape character“
  - Besondere Zeichen zur Ausgabe

K03

C++ Einführung

1. Allgemeines
2. **Entwicklungsprozess**
3. Gültigkeitsbereiche und Namespaces
4. Prozedur-/Funktionsaufrufe, Inlining, Default-Argumente
5. Elementare Typen und Strukturen
6. Vergleich Java/C++

Entwicklung mit mehreren Dateien

- In C++ ist es üblich, den Quelltext in mehreren Dateien zu halten
  - Gruppierung logisch zusammengehörender Funktionen/Klassen/Strukturen
  - Jede Datei bildet dadurch ein Modul
  - Dateiname und Inhalt der Datei sind für den Compiler zusammenhangslos (im Gegensatz zu z.B. Java); es ist aber üblich dass der Name den Inhalt in sinnvoller Weise repräsentiert.

main.cpp

```
int add(int a, int b);

int main(void) {
   int result, a=2, b=3;
   std::cout << add(a,b);
   return 0;
}
```

utils.cpp

```
int add(int a, int b) {
   return a + b;
}

int sub(int a, int b) {
   return a - b;
}
```

Organisation des Quellcodes (i)

```
#include <iostream>
#include "Body.h"

int main()
{
   Body a;
   std::cout<<a.volume()<<std::endl;
   return 0;
}
```

Deklaration in

Was ist Body ?

Was ist `Body::volume()` ?

test.cpp: Hauptprogramm, das die Klasse Body benutzt.

Organisation des Quellcodes (ii)

```
#include <iostream>
#include "Body.h"

int main()
{
   Body a;
   std::cout<<a.volume()<<std::endl;
   return 0;
}
```

test.cpp: Hauptprogramm, das die Klasse Body benutzt.

```
class Body
{
public:
   int volume() const;
...
};
```

Body.h: Header Datei in der die Klassen/Funktionen **deklariert** werden.

```
#include "Body.h"

int Body::volume() const
{
   return int k = 1;
}
...
```

Body.cpp: Implementierung (=**Definition**) der Klasse/Funktionen.

Deklaration versus Definition

Deklarationen und Definitionen sind zwei unterschiedliche Konzepte!

- Alle Bezeichner (Symbole) müssen dem Compiler bekannt gemacht werden, d.h. sie müssen vor der ersten Verwendung *deklariert* werden.

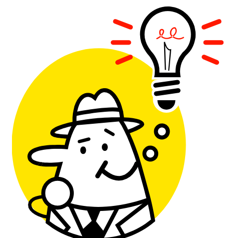
Deklaration

- Bekanntgabe an Compiler über Existenz z.B. einer Funktion, einer Klasse usw.
- Bei Funktion liefert sie die Signatur: Rückgabetyp und Liste der Argumente.
  - Deklaration einer Funktion wird auch als Funktionsprototyp bezeichnet.

Definition

- Variable: Hier wird vom Compiler Code erzeugt, der für sie Speicher reserviert.
- Klasse/Funktion: deren Implementierung.

Beachte: Jede Definition einer Variablen ist auch eine Deklaration.
Deklaration und Verwendung ohne Definition führt zu Linker-Fehler.
Mehrfachdeklaration sind möglich; Mehrfachdefinition jedoch nicht!



Funktionsdeklaration

Aufbau der Deklaration – Funktionsprototyp

$T_0 \quad F \; (\; T_1 \; [P_1], \; T_2 \; [P_2], \; \dots \;) \; ;$

T_0 = Typ des Funktions-Rückgabewertes

F = Name (Bezeichner) der Funktion

P_i = Formale Parameter (Variablen innerhalb der Funktion)

T_i = Typen der formalen Parameter

Beispiele:

```
int  foo(int x);
int  foo1 ( int, float );
void bar();
float evalSquare (float a0, float a1, float a2, float a3);
```

Funktionsdefinition

Syntax der Definition

```
T0   F ( T1 P1, T2 P2, … )
{
     . . .
}
```

- Innerhalb **{ }** stehen die Anweisungen und/oder Ausdrücke (*function body*)
- Formale Parameter werden wie andere Variablen innerhalb des Bodys verwendet
- **return x** beendet Funktion und gibt Funktionswert vom Typ T_0 zurück
- Falls T_0 = **void**, verwendet man **return** ohne Argument (nicht nötig, falls **return** unmittelbar vor **}**)

Beispiel:

```
bool sign(int x)
{
    if ( x >= 0)
        return true;
    else
        return false;
}
```

Beispiel

```
#include <iostream>
#include "Body.h"

 // declaration and definition
int global = 0;

int main()
{
   Body a; // definition
   int i=1, j; //defini (i), declar (j)
   int l = func1(i,j); // compiler err
   l = func2(i,j);
   std::cout<<a.volume()<<std::endl;
   return 0;
}
```

test.cpp: Hauptprogramm, das die Klasse Body benutzt

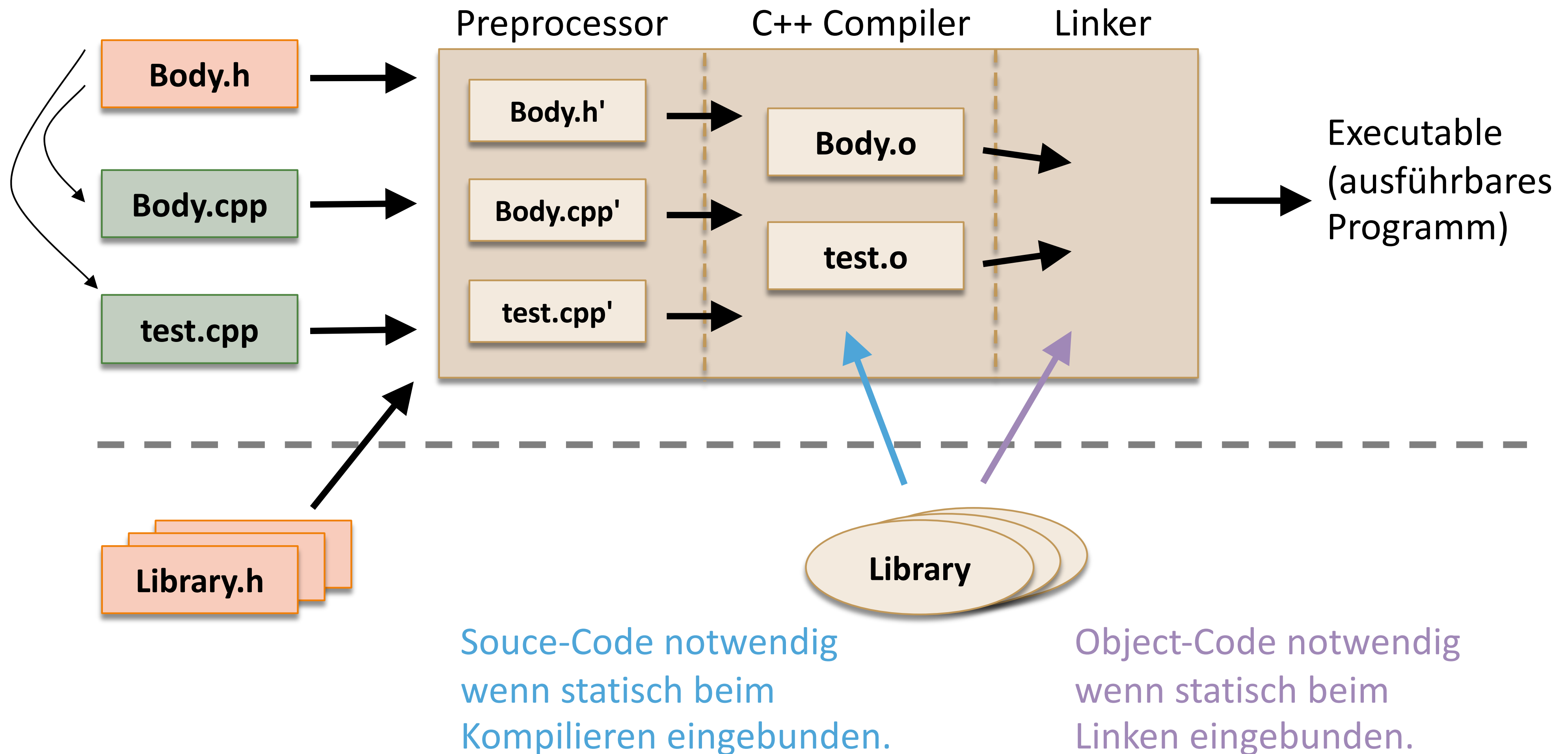
Body.h

```
int func1(int, int); //declaration
int func2(int k, int j); //declaration
class Body {
public:
   int volume();
};
```

Body.cpp

```
#include "Body.h"
//definition
int Body::volume()
{
    int MAX=10;   //declar + defini
   return MAX;
}
//definition
int func1(int k, int j) {
   return (k-j);
}
 //definition
int func2(int k, int j) {
   return (j+k);
}
```

Kompilieren und Linken - Stufen



Präprozessor

- Wertet Makros bzw. Direktiven aus:
  - Ein Makro/Direktive* ist eine Anweisung an einen Softwareprozessor; z.B. Textersetzung mittels **`#include`**-Anweisung durch Präprozessor.
    - Makros können Argumente haben; hier nicht behandelt.
  - Hauptanwendungsgebiet von Makros ist das bedingte Kompilieren:
    - Plattformunabhängigkeit: plattformspezifischer Code wird nur dann eingefügt, wenn tatsächlich für die betreffende Plattform kompiliert wird (ansonsten nicht).
- „Versteht“ selbst kein C++ (bzw. C).
- Der erzeugte C++ Quelltext wird an den Compiler übergeben.

Merke: Makros/Direktiven sind Anwendungsfälle der Meta-Programmierung – *„Programm im Programm“*

* Es existieren auch Direktiven, welche Anweisungen an den Compiler oder Linker sind.

Präprozessor-Anweisung – Beispiel

- Für die Arbeit mit mehreren Dateien wird meistes zu jeder Header-Datei die folgende Anweisung hinzugefügt:

```
#ifndef FILENAME_H
#define FILENAME_H

<header file body here>

#endif /* FILENAME_H */
```

- Damit wird gewährleistet, dass eine Header-Datei max. einmal beim Kompilieren eingebunden wird.

```
#ifndef BODY_H
#define BODY_H
int func1(int, int);//declaration
int func2(int k,int j);//declaration
const int MAX=10; //decla + defini.
class Body {
public:
    int volume();
};
#endif /* BODY_H */
```

Mehrfach-Einbindung von Headern

```
#ifndef _BASIS_H_
#def _BASIS_H_
// content of basis.h
int global=100;
int volume() {return global++; }
#endif
```

Header-Datei:
basis.h

Header-Datei: **statist.h**:

```
#include <iostream>
#include "basis.h"
```

Header-Datei: **graphen.h**:

```
#include <iostream>
#include "basis.h"
```

```
#include "statist.h"
#include "graphen.h"
int main()
{
   . . .
   return 0;
}
```

Quelldatei:
anwendung.cpp

Compiler

- Erwartet Quelltext (C++, C, usw.) ohne Präprozessoranweisungen
- Überprüft Syntax
- Generiert und optimiert Maschinencode
- Erzeugt Objekt-Datei(en) für den Linker
- Die Quelltext-Dateien, Zeilennummern und Bezeichner sind immer noch bekannt.
  - Je nach Compiler werde Letztere eventuell durch technische/interne Namen ersetzt (*engl. name mangling*), bedingt durch Regeln der Namensauflösung.

Objekt-Dateien

1. Objekt-Dateien enthalten Maschinencode für:
   - Funktionsdefinitionen,
   - globale Variablen, plus initiale Werte falls initialisiert.
2. Index der benutzten Symbole (Bezeichner).
   - **`nm`** (Linux) oder **`dumpbin`** (Windows) erzeugt diese Indexliste.

- Symbole sind noch keinen Adressen zugeordnet.
- Endung einer Objekt-Datei: **`.o`** (Linux) bzw. **`.obj`** (Windows)

Linker (Binder)

- Linker
  - Löst alle Abhängigkeiten der Objekt-Dateien auf.
  - Erzeugt das ausführbare Programm.
    - Dazu müssen u.a. Symbolen Adressen zugeordnet werden
  - Macht keine weitere Code-Optimierung.
  - Kennt keine Typen oder Variablen mehr.
  - Auflösung (technischer/interner) Namen ist abhängig vom Compiler. Es ist nicht garantiert, dass ein Linker Objektdateien von unterschiedlichen Compilern linken (binden) kann.

Statisches versus dynamisches Linken

- Statisches Linken: Alle Objektdateien werden in das ausführbare Programm gelinkt.
  - Der Linker sucht in allen Objekt-Dateien nach entsprechenden Symbolen und führt eine Zuweisung zwischen definierten Symbolen durch.
- Dynamisches Linken: Es gibt Symbole die erst zur Laufzeit aufgelöst und dynamisch gelinkt werden aus Bibliotheken.
  - `.dll` (Windows) und `.so` (Linux) Bibliotheken, die aus Objekt-Dateien bestehen und zur Laufzeit vom Betriebssystem eingebunden werden
  - Vorteile: Reduziert Grösse von ausführbaren Dateien; Fehler in Bibliothek erfordern nur erneutes Kompilieren der Bibliothek.
  - Nachteil: „**DLL Hell**" – Wenn neuere Version einer Bibliothek inkompatibel ist mit Programm das von älterer Version abhängt.

Bibliotheken (Libraries)

- Idee: bereits sorgfältig implementierte Software wiederverwenden.
- Library:
  - Menge von thematisch zusammenhängenden kompilierten Sourcen (Funktionen, Klassen, Typen, …), zusammengefasst in einer Datei.
  - Ist kein eigenständig lauffähiges Programm (enthält kein **`main`**).
- System-Library:
  - Library, die in bestimmten, vordefinierten Verzeichnissen installiert ist.
  - Kommt typischerweise mit dem Betriebssystem
Linux: **`/usr/lib`**
- Ein guter Programmierer kennt viele und die „richtigen“ Libraries ...

Bibliotheken (Libraries)

- Wie verwendet man System-Libraries?
  - In der C++ Datei: Header-Datei inkludieren
  - Beispiel: **`#include <math.h>`**

- Linken einer übersetzen Bibliothek (Objektdatei)
  - Konvention unter Linux:
    `-lm` linkt **`libm.so`**
  - Beispiel Math-Library linken:
    `g++ -o myprog file1.o file2.o –lm`

Makefile

- Der Prozess des Kompilierens wird durch ein Makefile automatisiert.
- Ein Makefile führt eine Liste der Abhängigkeiten der einzelnen Quelltext-Dateien und liefert Direktiven und Befehle zum automatischen Kompilieren des Projektes.

```
Body.o: Body.cpp, Body.h
   g++ -c -o Body.o Body.cpp
test_body.o: test_body.cpp, Body.h, iostream
   g++ -c -o test_body.o test_body.cpp
test: test.o, Body.o
   g++ -o test test.o Body.o
```


- Das Erstellen und das Verwalten von Makefiles kann sehr aufwändig sein. Es gibt Werkzeuge zum automatischen Erzeugen von Makefile:
  - Tmake, QMake, CMake, SCons

TMake

TMake ist frei verfügbar unter:

`http://tmake.sourceforge.net/`

“to create and maintain makefiles for software projects. It can be a painful task to manage makefiles manually, especially if you develop for more than one platform or use more than one compiler. tmake automates and streamlines this process and lets you spend your valuable time on writing code, not makefiles.”

TMake - Beispiel

test_body.pro

```
HEADERS     =  Body.h
SOURCES     =  Body.cpp hello.cpp
TARGET      =  test_body
```

```
>: setenv TMAKEPATH /local/tmake/lib/linux-g++
>: setenv PATH $PATH:/local/tmake/bin
>: tmake test_body.pro -o Makefile
>: make
g++ -c -pipe -Wall -W -O2   -o Body.o Body.cpp
g++ -c -pipe -Wall -W -O2   -o hello.o hello.cpp
rm -f test_body
g++  -o test_body Body.o hello.o
>: make clean
rm -f Body.o hello.o     test_body
rm -f core *~
```

- TMake:
  - Generiert automatisch ein Makefile.
  - Die Anhängigkeiten zwischen den Dateien werden automatisch erkannt.

Dokumentation des C++ Codes

- Das Verwalten und Modifizieren von grossen C++ Paketen kann sehr komplex sein.
  - Gute Dokumentation ist deshalb sehr wichtig.

- Doxygen ist ein Werkzeug zur Dokumentation von C++ Quelltexten vergleichbar zu Javadoc.
  - Doxygen ist frei verfügbar unter `http://www.doxygen.org`

Doxygen

Beispiel: Dokumentation einer Funktion

```
/** Dokumentation of a function
 *   @param lower        lower bound of the range
 *   @param upper        upper bound of the range
 *   @return             A vector with the same size as this
 *                       vector and binary elements
 *   @warning            some detail ..
 *   @todo               ...
 *   @bug                ...
 */

FVector findInRange(float lower, float upper) const;
```

Doxygen analysiert die C++ Dateien und erzeugt html-Dateien mit der entsprechenden Dokumentation.

C++ Einführung

1. Allgemeines
2. Entwicklungsprozess
3. **Gültigkeitsbereiche und Namespaces**
4. Prozedur-/Funktionsaufrufe, Inlining, Default-Argumente
5. Elementare Typen und Strukturen
6. Vergleich Java/C++

Gültigkeitsbereiche in C++

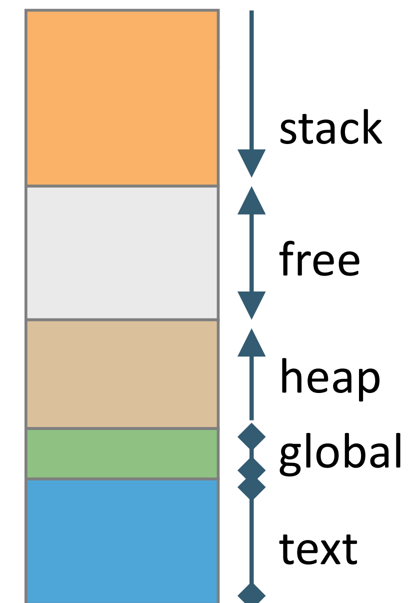
- C++ (und viele andere Programmiersprachen) benutzen lexikalischen Scope. Ein Scope wird u.A. durch einen **Block** gebildet.
- Ein Block ist ein Paar, bestehend aus `{´ und `}´.
- Blöcke können beliebig tief verschachtelt sein.
- Syntaxregeln definieren, an welchen Stellen im Programm Blöcke gebildet werden können.

- Bezeichner, die innerhalb eines Blocks deklariert sind, heissen **lokal**.
- Bezeichner, die ausserhalb aller Blöcke und Funktionen definiert sind, heissen **global**.

Globale Sichtbarkeit

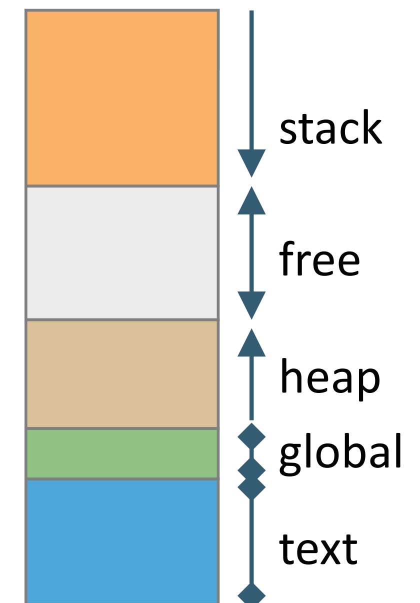
- Globale Bezeichner sind in einem Block sichtbar (z.B. in einer Funktion), wenn:
  - die Deklaration vor der Funktion oder dem Block liegt,
  - der Bezeichner anders heisst als die im Block,
  - bei einer Funktion alle Parameter der Funktion anders heissen als der Bezeichner,
  - alle lokalen Bezeichner (Variablen, Typen, etc..) anders benannt sind.

- Globale Variablen liegen im globalen Segment.



Lokale Sichtbarkeit

- Bezeichner, die in einem Block (geschachtelten Block) deklariert werden, sind sichtbar:
  - nur innerhalb dieses Blocks, ab der Stelle der Deklaration bis zum Ende des Blocks
  - von Blöcken, welche in diesen Block geschachtelt sind, falls sie nicht selbst einen Bezeichner mit gleichem Name deklarieren.
- Die Sichtbarkeit einer Funktion entspricht der eines Bezeichners ausserhalb aller Blöcke
  (Funktionen können in C++ nicht geschachtelt werden)
- Lokale Variablen liegen auf dem Stack.



Globale Variablen

- Globale Variablen können von jedem Ort aus erreicht werden, auch wenn sie durch lokale Bezeichner „verdeckt“ sind.
- Der zugehörige Operator `::` heißt Bereichsoperator *(engl. scope resolution operator)*

- Globale Variablen sollten möglichst vermieden werden!
  - Seiteneffekte sind möglich, wenn die Variablen in verschiedenen Funktionen verwendet werden.
  - Globale Variablen sind in der Regel schlecht zu warten.
  - Ihre Initialisierung – Reihenfolge – ist nicht eindeutig.

Sichtbarkeit: Beispiel

```
int x = 10;                 // Globale Variable

void f () {                 // Funktionsblock
  int y = x;                // Benutze globales x
  int x = y - 10;           // Lege lokales x an
  ::x = ::x - 9;            // Benutze globales x

  {                         // verschachtelter Block
    int y = x;              // Welchen Wert hat y?
    int x = 5;
    // Wie oft wird diese Schleife durchlaufen?
    for (int j = 0; j < ::x; ++j) { /* ... */ }
  }
  x = x * 100;              // Welches x wird hier verändert?
}
```

Problem bei der Verwendung globaler Variablen ...

- Wieso wird sich der Compiler (beim Übersetzen) mit **`gcc -Wall util.cpp test.cpp -o test`** darüber beschweren ?

```
/* util.cpp */
#include <iostream>
using namespace std;

int g_numCalls = 0;

void someFunc(void)
{
   cout << "someFunc:util "
        << g_numCalls++
        << endl;;
}
```

```
/* test.cpp */
#include <iostream>
using namespace std;

void someFunc(void);
int  g_numCalls = 10;

int main(void)
{
   cout << "someFunc:main "
          << g_numCalls
          << endl;
   someFunc();
   someFunc();
}
```

- Beim Linken tritt zweimal die gleiche globale Variable auf !

Speicherklasse von Objekten

- Die Deklaration eines Objektes legt neben Typ und Name auch seine Speicherklasse fest.
- Diese bestimmt die Lebensdauer.
- Die Speicherklasse eines Objektes ist festgelegt durch:
  1. Die Position der Deklaration innerhalb der Quelldatei
  2. Die optionale Speicherklassen-Spezifikation:
     - **`extern/static`** siehe später
     - **`auto`** (Default) beim Erreichen einer Definition wird das Objekt auf dem Stack neu erzeugt und beim Verlassen wieder zerstört (im Gegensatz zu **`static`**)
     - **`register`** Zur Beschleunigung der Programmausführung, Variable wird im Register gehalten

Der Modifier **static** (i)

- Eine wesentliche Eigenschaft statischer Objekte ist ihre statische (=permanente) Lebensdauer.
- Statische Objekte werden im Datensegment eines Programms gehalten, nicht auf dem Stack.

static hat hierbei drei verschiedene Bedeutungen:

1. Vor einer globalen Variable oder Funktion

```
static int  g_someValue = 0;
static void g_someFunction(void);
```

   - Weist den Linker an, diesen Bezeichner nicht zu exportieren.
   - Beschränkt die Sichtbarkeit des Bezeichners auf die Datei.
   - Der Linker wird ihn nicht verwenden, um Abhängigkeiten anderer Dateien aufzulösen.

Der Modifier **static** (ii)

2. Vor einer lokalen Variable (in einer Funktion)

```
void someFunc(void) {
  static int array[4000];
}
```

- Platziert die Variable nicht auf dem Stack
- Hat den Effekt, dass die Variable zwischen wiederholten Aufrufen ihren Wert beibehält

→ Funktion mit Gedächtnis
(ohne auf global sichtbare Variablen zurückgreifen zu müssen)

- Verwendung z.B. um zu vermeiden, dass grosse Objekte immer wieder auf dem Stack angelegt werden.

Der Modifier `static` (iii)

3. Vor einer Klassenvariable (oder Methode)

```
class Whatever {
    private:
        static Whatever* instance;
    public:
        static Whatever* getInstance (void);
}
```

später

- Definiert globale Variable/Funktion für eine Klasse
- Alle Instanzen einer Klasse teilen sich ihre statische Variablen
- Zugriff von aussen (falls `public`): `Class::Member`
- Statische Memberfunktionen können nur auf statische Membervariablen zugreifen

Problem bei der Verwendung von globalen Variablen …

Lösung für das Problem der Mehrfachverwendung globaler Variablen (Folie 45):

```
/* util.cpp */
#include <iostream>
using namespace std;

int g_numCalls = 0;

void someFunc(void)
{
   cout << "someFunc:util "
        << g_numCalls++
        << endl;;
}
```

```
gcc -Wall util.cpp test.cpp -o test
```

Ausgabe:

```
someFunc:main 10
someFunc:test 10
someFunc:test 11
```

```
/* test.cpp */
#include <iostream>
using namespace std;

static int g_numCalls = 10;
static void someFunc(void)
{
  cout<<"someFunc:test "
      << g_numCalls++
      << endl;
}
int main(void)
{
   cout << "someFunc:main "
        << g_numCalls
        << endl;
   someFunc();
   someFunc();
}
```

Problem bei der Verwendung von globalen Variablen ...

- **`static`** kann hier helfen, aber:
  - Die Variablen **`g_numCalls`** aus **`util.cpp`** und **`g_numCalls`** aus **`test.cpp`** stehen in keiner Beziehung zueinander (sind sozusagen 'privat' per Datei).

Problem bei der Verwendung von globalen Variablen …

Nochmals zum Problem der Mehrfachverwendung globaler Variablen (Folie 45):

```
/* util.cpp */
#include <iostream>
using namespace std;

int g_numCalls = 0;

void someFunc(void)
{
   cout << "someFunc:util "
        << g_numCalls++
        << endl;;
}
```

```
gcc -Wall -ansi util.cpp
       test.cpp -o test
```

```
/* test.cpp */
#include <iostream>
using namespace std;

static int g_numCalls = 10;
// declaration of someFunc
void someFunc(void);

int main(void)
{
   cout << "someFunc:main "
        << g_numCalls
        << endl;
   someFunc();
   someFunc();
}
```

Ausgabe:

```
someFunc:main 10
someFunc:util 0
someFunc:util 1
```

Problem bei der Verwendung von globalen Variablen …

Nochmals zum Problem der Mehrfachverwendung globaler Variablen (Folie 45):

```
/* util.cpp */
#include <iostream>
using namespace std;

int g_numCalls = 0;

void someFunc(void)
{
   cout << "someFunc:util "
        << g_numCalls++
        << endl;;
}
```

```
gcc -Wall -ansi util.cpp
        test.cpp -o test
```

Ausgabe:

```
someFunc:main 0
someFunc:util 0
someFunc:util 1
```

```
/* test.cpp */
#include <iostream>
using namespace std;

extern int g_numCalls;
void someFunc(void);

int main(void)
{
   cout << "someFunc:main "
        << g_numCalls
        << endl;
   someFunc();
   someFunc();
}
```

extern: Variable ist irgendwo global definiert, kann hier aber benutzt werden

Globale Variablen: Weiteres Beispiel

- Was könnte hier Probleme verursachen ?

```
/* debug.h */

int debug_level;
…
```

```
/* test.cpp */

#include "debug.h"
…
```

```
/* debug.cpp */

#include "debug.h"
…
```

```
/* otherfile.cpp */

#include "debug.h"
…
```

Übersetzung mit: `gcc -Wall *.cpp -o test`

- Fehler: der Linker wird drei mal die gleiche globale Variable **debug_level** sehen.

Weiteres Beispiel: Lösung mit **static**?

- **static** beseitigt zwar den Compilerfehler, verursacht aber ein neues Problem!

```
/* debug.h */

static int debug_level;
…
```

```
/* test.cpp */

#include "debug.h"
…
```

```
/* debug.cpp */

#include "debug.h"
…
```

```
/* otherfile.cpp */

#include "debug.h"
…
```

Übersetzung mit: `gcc -Wall *.cpp -o test`

- Jede Datei bekommt ihre eigene Version der Variablen **debug_level**!

Externe Bindung

- Forderung:
  - die Variable `int debug_level` soll einmal (im Header) deklariert werden
  - die Variable soll von verschiedenen Dateien aus benutzt werden können
  - es soll nicht mehrfach Platz für die Variable allokiert werden
  - Deklaration der Variablen als `extern`

- Deklaration:

```
extern int debug_level;
```

- Das bedeutet für den Compiler:

  "irgendwo existiert eine Variable `debug_level` *vom Typ* `int`*,
  aber der Speicher dafür liegt nicht hier; der Linker kümmert sich
  um die korrekte Einbindung!"*

- Die Variable muss dann natürlich einmal in einer der `.cpp`-Dateien definiert (angelegt und initialisiert) werden.

Weiteres Beispiel: Lösung mit **extern**

- Korrekte Variante: die Variable **debug_level** wird einmal in einem Header als **extern** deklariert, im entsprechenden C++-File definiert (im Speicher angelegt).

```
/* debug.h */


extern int debug_level;

…
```

```
/* test.cpp */

#include "debug.h"

…
```

```
/* debug.cpp */

#include "debug.h"
int debug_level = 3;

…
```

```
/* otherfile.cpp */

#include "debug.h"

…
```

Namensbereiche *engl. namespace*

- Beim Einbinden von Header-Dateien werden die darin deklarierten globalen Bezeichner in den aktuellen Programmkontext eingefügt.
- Das kann Namenskollisionen führen, wenn die Bezeichner bereits vergeben sind.
- Dies ist insbesondere dann ein Problem, wenn der Programmierer auf die Namensgebung keinen Einfluss hat.
  - z.B. bei Verwendung zweier kollidierender externer (*third-party*) Bibliotheken

Namensbereiche (ii)

- Lösung: **namespace**-Mechanismus
- Syntax:

```
namespace <namespace-Name>
{
  // Member-Deklaration bzw. Definition
}
```

- Namespace Members können Variablen, Typen, Funktion und andere Namensbereiche sein.

Zugriff auf Namensbereich

- Members innerhalb eines **`namespace`** werden wie gewohnt angesprochen.
- Von ausserhalb wird durch den qualifizierten Namen des **`namespace`** Members zugegriffen.

- Qualifizierter Name *(engl. fully qualified name)*

```
<namespace-Name>::<Member>
```

- Beispiel:

```
// std ist der Namensbereich der Standard-Bibliothek
std::cout << "some text " << std::endl;
```

Die **using**-Anweisung

- Es kann sehr lästig werden, häufig benutzte **namespace** Members bei jeder Verwendung komplett zu qualifizieren.
- Innerhalb eines Gültigkeitsbereiches kann die **using**-Anweisung den Zugriff lockern.
- Beispiel:

```
using <namespace-Name>::<Member>;
```

oder

```
using <namespace-Name>; // alle Member des Namensbereich
```

Namensbereiche: Beispiel

```
#include <iostream>
namespace MyStruct {
  struct Y {
     static int x;
  };
  int Y.x = 13;
}
namespace std {
   void newFunction (void) {}
   // in namespace von
      iostream einfügen.
}
int x = 10;
int main () {
   using std::cout;
   using std::endl;
```

Neuer Namensbereich **MyStruct**

Erweiterung des Namensbereiches **std**

```
   // Zugriff namespace-Struct
   MyStruct::Y var1;

   // Zugriff benutzen

   // Öffne neuen Gültigkeitsber.
   {
     double x = 3.1415;

     cout << x;     // -> Pi
     cout << ::x;   // -> 10
   }
   cout << MyStruct::Y.x << endl;
}
```

K03

C++ Einführung

1. Allgemeines
2. Entwicklungsprozess
3. Gültigkeitsbereiche und Namespaces
4. **Prozedur-/Funktionsaufrufe, Inlining, Default-Argumente**
5. Elementare Typen und Strukturen
6. Vergleich Java/C++

Was passiert bei Funktionsaufrufen?

Wiederholung: Siehe *Aufrufkonventionen* im Kapitel zu Assembler.

```
int min(int a, int b)
{
   if (a < b) return a;
   return b;
}

int main()
{
   int a = 42;
   int b = 137;
   // addr1
   std::cout << min(a, b) << endl;
   return 0;
}
```

1. Parameter auf dem Stack ablegen, inkl. Platz für Rückgabewert, falls vorhanden.
2. Rücksprungadresse auf dem Stack ablegen.

top	addr1
t-1	return
t-2	a
t-3	b

3. Sprung zur Funktion `min`.
4. Funktion ausführen.
5. Rücksprung hinter den Aufruf.
6. Parameter vom Stack nehmen.
7. Nächste Anweisung.

Inline-Funktionen in C++

```
inline int min(int a, int b)
{
   if(a < b) return a;
   return b;
   // (a < b ? a : b)
}

int main()
{
   int a = 42;
   int b = 137;

   std::cout << min(a,b) << endl;
   return 0;
}
```

Häufige Aufrufe von „kleinen“ Funktionen beeinträchtigt das Laufzeitverhalten:

- Sicherung der Rücksprungadresse
- Objekte kopieren/erzeugen
- Hin-/Rücksprung

Deshalb kann man Funktionen als **inline** definieren.

Da **min()** als **inline** definiert ist, wird eine Kopie der Funktion vom Compiler an diese Stelle eingesetzt.

```
std::cout << (a < b ? a : b) <<endl;
```

Verwendung von Inline-Funktionen/- Methoden

Inline-Funktionen bzw. -Methoden werden verwendet, wenn …

- … diese kurz sind und
- … diese (sehr) häufig aufgerufen werden.

Vorteil:

Das Programm kann effizienter (schneller) werden.

Nachteil:

Das ausführbare Programm wird grösser (durch Codevervielfältigung).

Globale Funktionen in C++

- In C++ können Funktionen global (ausserhalb von Klassen) definiert werden. Solche Funktionen gehören nicht zu einer bestimmten Klasse – sie stellen in der Regel generell anwendbare Algorithmen zur Verfügung.

```
float square(float x);  // function declaration
...

float square(float x) { // function definition
   return x*x;
}

...

int main(int argc, char** argv) {
   float f = 5.0;
   std::cout << "The square of " << f << " is " << square(f);
}
```

Default-Argumente in C++

- In C++ können Funktionen Default-Argumente besitzen

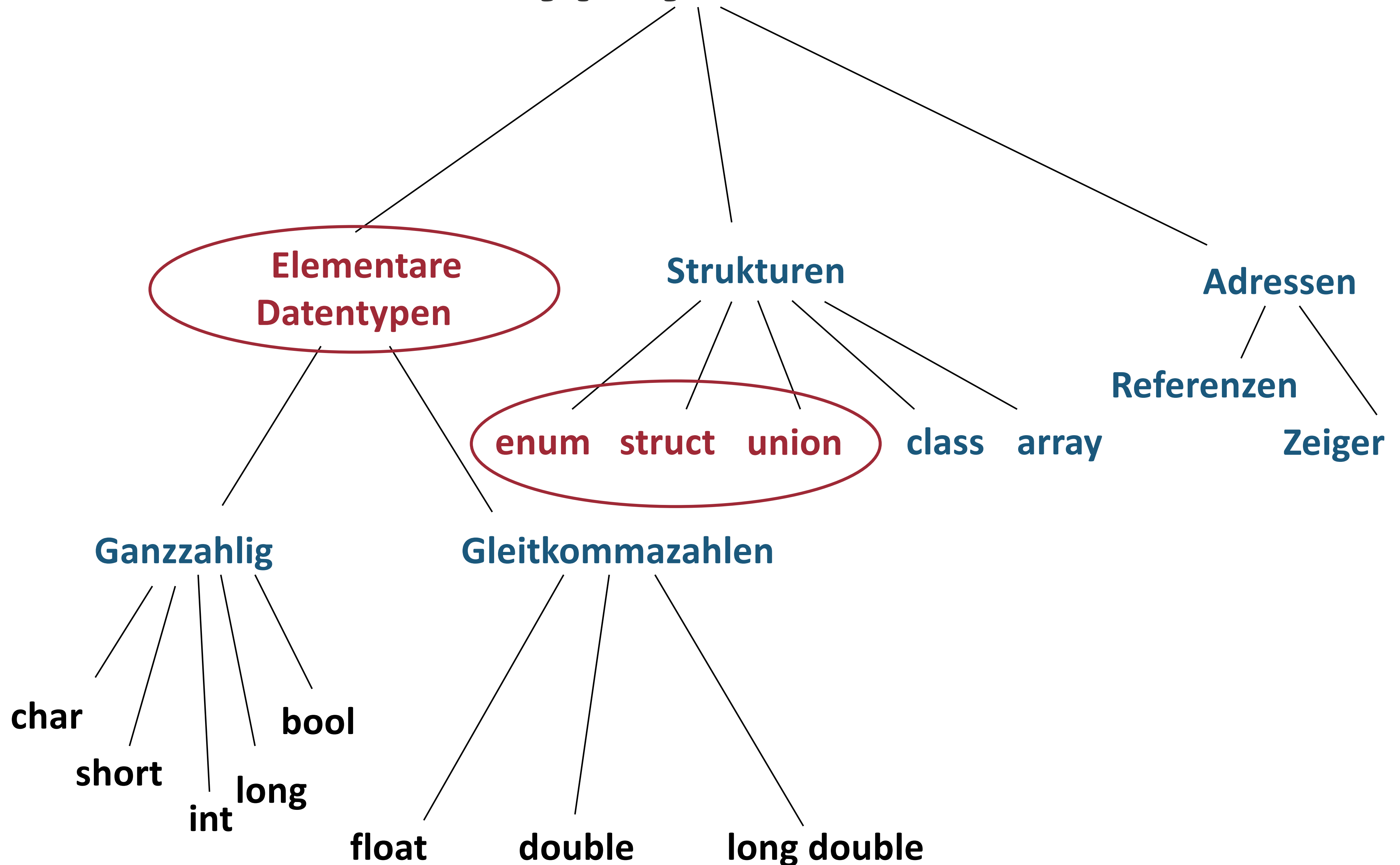
```
float pow(float base, int exp=2);   // function declaration

...

float pow(float base, int exp) {    // function definition
   if (exp == 0) return 1;
   for (int i=1; i < exp; ++i) base *= base;
   return base;
}

...

int main(int argc, char** argv) {
   pow(5);                          // return 25
   pow(5, 3);                       // return 125
}
```

K03

C++ Einführung

1. Allgemeines
2. Entwicklungsprozess
3. Gültigkeitsbereiche und Namespaces
4. Prozedur-/Funktionsaufrufe, Inlining, Default-Argumente
5. **Elementare Typen und Strukturen**
6. Vergleich Java/C++

Das **Typsystem** in C++



Elementare Datentypen in C++

Nur bedingt portabel!

Typ	typ. Speicherplatz - Werte mancher Typen abhängig v[illegible]	Wertebereich
`char`	1 Byte	-128 bis +127
`unsigned char`	1 Byte	0 bis +255 (ASCII)
`wchar_t`	2 byte	0 bis +65'535 (Unicode)
`int`	4 Byte (in der Regel)	-2'147'483'648 bis +2'147'483'647
`unsigned int`	4 Byte (in der Regel)	0 bis +4294967295
`short`	2 Byte	-32'768 bis +32'767
`unsigned short`	2 Byte	0 bis +65'535
`long`	8 byte	–9'223'372'036'854'775'808 bis +9'223'372'036'854'775'807
`unsigned long`	8 byte	0 bis +18'446'744'073'709'551'615

Mit Hilfe des Operators **`sizeof`** kann man den Speicherbedarf von Variablen bzw. allen Arten von Datentypen bestimmen, also nicht nur von elementaren.

Beispiele:

```
sizeof (char);     /* liefert 1 */
short myShort;
sizeof (myShort)   /* liefert 2 */
```

C++11 (und C11): <stdint.h> <cstdint>

Portable Typen:

Vorzeichenbehaftet	Vorzeichenlos	Beschreibung
`intmax_t`	`uintmax_t`	Maximal von der Zielplattform unterstützte Breite.

`int8_t`
`int16_t`
`int32_t`
`int64_t`
	`uint8_t`
	`uint16_t`
	`uint32_t`
	`uint64_t`
		Genau 8, 16, 32, 64 Bit breit.
		Je nach Zielplattform nicht definiert, wenn Typ mit dieser Breite nicht existiert.

`int_least8_t`
`int_least16_t`
`int_least32_t`
`int_least64_t`
	`uint_least8_t`
	`uint_least16_t`
	`uint_least32_t`
	`uint_least64_t`
		Mindestens 8, 16, 32, 64 Bit breit.
		Kompaktester Typ, der mind. *n* Bit breit ist; praktisch meist identisch mit **`int*_t`** bzw. **`uint*_t`**

`int_fast8_t`
`int_fast16_t`
`int_fast32_t`
`int_fast64_t`
	`uint_fast8_t`
	`uint_fast16_t`
	`uint_fast32_t`
	`uint_fast64_t`
		Mindestens 8, 16, 32, 64 Bit breit.
		Wenn breiterer Typ besser von Zielplattform unterstützt wird, dann wird dieser benutzt.[1)]

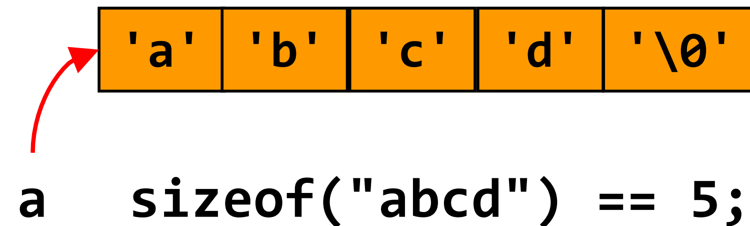
1) Beispiel: 64 Bit-Plattform: `uint_fast16 // angefordert` → `uint64_t // tatsächlich`

Klassische C-Strings

- C-Strings sind äquivalent zu Arrays of **`char`**
- Enden immer mit **`'\0'`** – man sagt sie sind *0-terminiert*.

```
char a[5] = "abcd";
```

```
a[0] == 'a'
a[4] == '\0'
```




```
sizeof("abcd") == 5;
```

- Bei der Verwendung klassischer C-Strings sind einige Details zu berücksichtigen:
  - 0-Terminierung
  - Speicherallokation
  - separate Funktionen für Suche, Kopie, Teilstrings, etc ... in **`<string.h>`**

C++-Strings

- Die C++ Standardbibliothek beinhaltet Klasse **std::string**

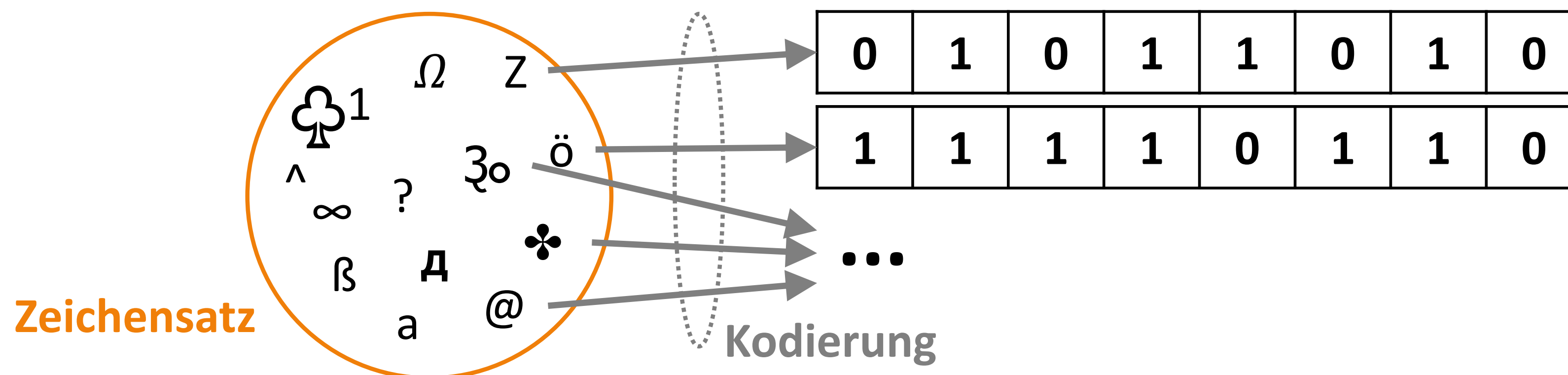
```
#include <string>      // Achtung: kein .h
using namespace std;   // Programmierer sind schreibfaul

string text1 = "A nice day"; // Die klassische Deklaration
string text2("A nice day");  // Äquivalent via Konstruktor
```

```
string copy = text1;          // Erzeugen einer Kopie
cout << "Text: " << copy;     // Ausgabe via C++ iostream
// Konvertierung in C-sytle string ist möglich, aber
// const verhindert, dass auf diesem Weg Daten verändert
// werden können
const char* text_p = copy.c_str();
text_p[0] = 'X';    // Fehler !
```

Strings: Zeichensatz & Kodierung

engl. charset & character encoding

- Ein Zeichensatz ist eine nichtleere, endliche Menge an Zeichen.
- Die Kodierung ist eine Abbildung der Zeichen eines Zeichensatzes auf üblicherweise Ganzahlen (zur Repräsentation dieser im Speicher).



- Internationale Standards wie *ASCII* (1963), *EBCDIC* (1964), *ISO 8859* (1986), *Unicode* (1991) definieren einerseits einen Zeichensatz und andererseits ein- oder mehrere verschiedene Kodierungen.
- Beide Begriffe werden häufig vermischt oder synonym verwandt, sind aber strikt genommen nicht dasselbe.

Strukturen – `struct`

- Strukturen fassen logisch zusammengehörende Daten zusammen, z.B.:

```
struct student {
        int id;
        char name[80];
};
```

Structure tag

Structure members

- Für den Entwickler
  - **`id`** und **`name`** gehören nun zusammen
  - **`struct student`** erzeugt ein gemeinsames Datenfeld

- Für den Compiler
  - **`id`** und **`name`** liegen „nebeneinander“ im Speicher
  - **`struct student`** ist ein komplexer Datentyp, der an Funktionen übergeben werden kann

Syntax & Verwendung von **struct**

Syntax einer C-Struktur

```
struct [name] {
   <type> field1;
   <type> field2;
   …
} [instance list];
```

Beispieldeklaration:

```
struct Foo {
      int field1;
      char field2;
};
```

Definition (und Deklaration)

```
struct Foo foo;
struct point {
   int a;
   double b;
} objct1, objct2;
struct point objct3 = {1, 3.12}
```

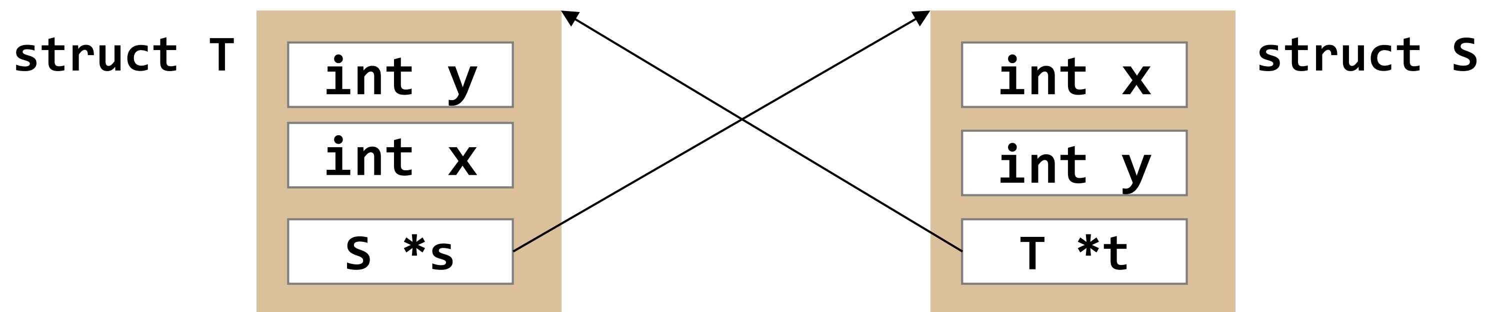
Datenzugriff:

```
foo.field1;
```

“Zugriff auf Feld **field1** der Instanz **foo** der **struct Foo**”

Forward-Declaration von **struct**

- Problem: wie deklariert man die folgenden beiden **structs**?



Lösung: Forward-Deklaration

Vorgehensweise funktioniert nur für. Typen deren Grösse schon bekannt ist (z.B. Zeiger und Methoden).

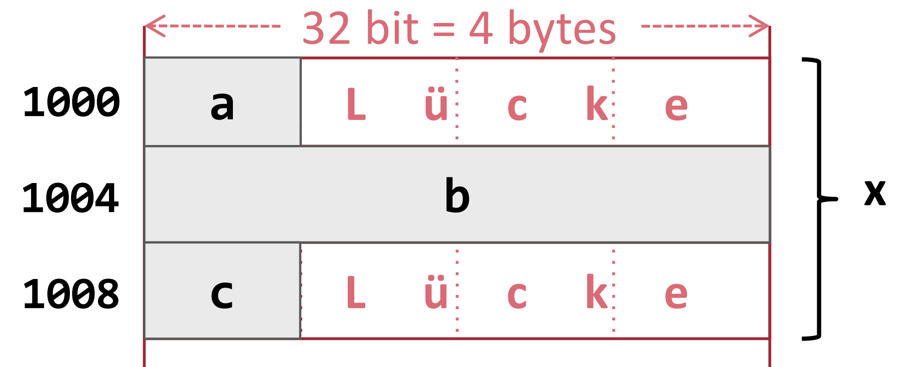
Analoge Vorgehensweise auch bei Klassen.

```
struct S;
struct T {
        …
        S *s;
        …
};
struct S {
        …
        T *t;
        …
};
```

Speicherausrichtung (engl. alignment)

- Compiler richtet Member einer Struktur im Speicher an vordefinierten Grenzen aus*:
  - z.B. am Wortanfang: wenn $\mathrm{sizeof}(t) \bmod w \mathrel{!=} 0$ (t ist Typ, w Wortbreite in Bytes) dann werden Lücken eingefügt (engl. padding).

```
struct foo {
   char a;
   int  b;
   char c;
} x;
```



    - Nachteil: Es geht Speicherplatz verloren.
    - Vorteil: schneller Speicherzugriff da optimal durch Hardware unterstützt.
  - Manche CPUs unterstützen nichtausgerichtete Anordnung, aber:
    - Langsamer wenn mehr als ein Speicherzugriff und/oder Cache miss.
    - Je nach Hardware kein atomares Lesen bei mehreren Speicherzugriffen.

* Details siehe z.B.: http://en.wikipedia.org/wiki/Data_structure_alignment#Typical_alignment_of_C_structs_on_x86

Aufzählungen – **enum**

- Eine Aufzählung ist ein selbst definierter ganzzahliger Typ mit Schlüsselwort **enum**

```
enum PrimaryColors {
        RED = 2,
        GREEN,
        BLUE
};
```

```
PrimaryColors color = GREEN;
switch(color)
{
        case RED:
        ...
        case GREEN:
}
```

- Default: Erster Wert (**RED**) wird mit **0** initialisiert
- Jeder weitere Konstante hat einen um eins erhöhten Wert zum Vorgänger
- Man kann aber auch den ersten Wert beliebig wählen (**RED = 2**)

Syntax von **enum**

Deklaration

```
enum name {
     OptionName [= int],
     OptionName [= int],
   …
} [instance list];
```

Definition

```
enum Color {
    RED = -3,
    GREEN = 4,
    BLUE = 12
} color, *color_ptr;
enum Color c;
void drawCircle
    (enum Color c);
```

Alle Werte können aber auch explizit vorgegeben werden

Zeigervariable: Variable, die im Speicher auf eine Speicherstelle verweist, die einen Wert vom Typ **Color** beinhaltet.

Unions – **union**

- Struktur bei der alle Member an der gleichen Adresse beginnen.
- Anwendungsfälle:
  - Speicherbereich den man mit Werten verschiedener Datentypen belegen will; z.B. Knoten in Baum der Werte vom Typ **int** oder **double** enthalten kann; d.h. Wertebereich ist: **int** ∪ **double**
  - Eher selten: Uminterpretation von Daten

```
union foo {
   int i;    // 4 bytes
   double d; // 8 bytes
} x;
```

```
x.i = 3;
int y = x.i; // y = 3
x.d = 3.5;
int z = x.i; // z = ?
```

niederwertige 32 bits von 3.5
interpretiert als 32-bit Integer

- Grösse eines Union im Speicher entspricht Grösse seines grössten Member; im Beispiel: **sizeof(x) = 8**

Zusammenfsg. syntaktischer Varianten

1. Deklaration benannte Struktur/Union/Enum:

```
struct
union   name { /* ... */ };
enum
```

2. Deklaration anonyme Struktur/Union/Enum und Definition Variable(n) dieses Typs:

```
struct
union   { /* ... */ } var_1, ..., var_n;
enum
```

3. Deklaration benannte Struktur/Union/Enum und Definition Variable(n) dieses Typs:

```
struct
union   name { /* ... */ } var_1, ..., var_n;
enum
```

4. Variablendefinition(en) einer zuvor deklarierten Struktur/Union/Enum:

```
struct
union   name var_1, ..., var_n;
enum
```

Schlüsselwort **typedef** (i)

- Typedef bietet die Möglichkeit, Typen einen neuen Namen zu geben

```
unsigned char mybyte;
```

```
typedef  unsigned char  BYTE;
BYTE mybyte;
```

- **BYTE** ist nun ein **Alias** für **unsigned char**
- Beide Definitionen von **mybyte** sind nun äquivalent für den Compiler
- Vorteile der **typdef** Definition:
  - Aussagekräftiger (Lesbarkeit)
  - Maschinenabhängige Typen können isoliert werden. Bei der Portierung muss nur der Alias „umgehangen" werden.

Schlüsselwort **typedef** (ii)

Anwendung auf Strukturen:

```
struct Student {
    int id;
    char name[80];
};


struct Student st;
```

```
typedef struct {
    int id;
    char name[80];
} STUDENT;


STUDENT st;
```

Schlüsselwort **typedef** (iii)

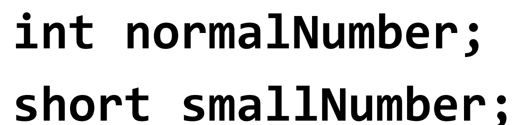
Vorteil

- Man kann schnell an einer Stelle im Code den Type einer bestimmten Variablen ändern.
- Wird häufig in Verbindung mit **struct** verwendet

```
struct point {
        int a;
        double b;
        };

struct point objct2;
```

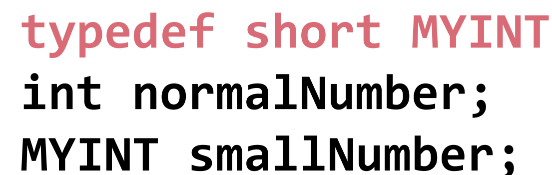
```
typedef struct point {
        int a;
        double b;
} POINT;

POINT objct2;
```

```
int normalNumber;
int smallNumber;
```



```
int normalNumber;
short smallNumber;
```

```
typedef int MYINT
int normalNumber;
MYINT smallNumber;
```



```
typedef short MYINT
int normalNumber;
MYINT smallNumber;
```

Schlüsselwort **typedef** (iv)

- Die Lesbarkeit eines Programms wird durch **typedef** verbessert, vor allem bei komplexen Datentypen, z.B. Funktionszeigern:

```
char* (*search)(float, int);
```

search ist ein Zeiger auf eine Funktion, die zwei Eingabe-Parameter vom Typ **float** und **int** besitzt und die einen Zeiger auf **char** liefert
(später mehr über Zeiger in C++)

```
typedef char* (*PTR_TO_FUNC) (float, int);
```

Besser (kürzer):

```
PTR_TO_FUNC search;
```

K03

C++ Einführung

1. Allgemeines
2. Entwicklungsprozess
3. Gültigkeitsbereiche und Namespaces
4. Prozedur-/Funktionsaufrufe, Inlining, Default-Argumente
5. Elementare Typen und Strukturen
6. **Vergleich Java/C++**

Elementare Datentypen in Java/C++

Elementare Datentypen in Java

- **`boolean`**
- **`byte`**
- **`char`**
- **`short`**
- **`int`**
- **`long`**
- **`float`**
- **`double`**

Elementare Datentypen in C++

- **`bool`**
- **`(unsigned) char`**
- **`wchar_t`**
- **`(unsigned) short`**
- **`(unsigned) int`**
- **`(unsigned) long`**
- **`float`**
- **`double`**
- **`long double`**

Vergleich Java/C++ – Zuweisungen

In Java ist das Verhalten von elementaren Datentypen und Objekten unterschiedlich

In C++ ist das Verhalten von elementaren Datentypen und Objekten identisch

Java			C++	
int a = 0;	// Creation of an integer	=	int a = 0;	// Creation of an integer
int b = a;	// b holds a copy of a	=	int b = a;	// b holds a copy of a
b = 42;	// only b is changed	=	b = 42;	// only b is changed
Body x;	// no object creation	≠	Body x;	// the object x is created
Body y = x;	// y holds a reference to x	≠	Body y = x;	// y holds a copy of x
y.setVol(42);	// y and x are changed	≠	y.setVol(42);	// only y is changed

Vergleich Java/C++ — Referenzen

In C++ gibt es die Möglichkeit, explizit Referenzen zu benutzen — damit verhalten sich dann Objekte wie Objekte in Java

Java

```
Body y = x;          // y holds a
                     // reference to x

y.setVol(42);        // y and x are
                     // changed
```

=

C++

```
Body& y = x;         // y holds a
                     // reference to x

y.setVol(42);        // y and x are
                     // changed
```

Vergl. Java/C++ — Funktionsaufrufe

Funktions-Deklaration in Java | Funktions-Deklaration in C++

Java

```
void f1( int a)         // a is a copy
{
  a = 42;
}

void f2( Body x)        // x is a reference
{
  x.setVol( 42);
}


// Call
f1(b);               // b is not modified

f2(y);               // y is modified
```

C++

```
void f1( int a)       // a is a copy
{
  a = 42;
}

void f2( Body x)      // x is a copy
{
  x.setVol( 42);
}


// Call
f1(b);        // b is not modified

f2(y);        // y is not modified
```

f1: Java = C++ ; f2: Java ≠ C++ ; f1(b): = ; f2(y): ≠

Vergl. Java/C++ — Call by Reference

Funktions-Deklaration in Java

Funktions-Deklaration in C++

Java

```
void f2( Body x) // x is a reference
{
  x.setVol( 42);
}


// Call

f2(y);                // y is modified
```

=

=

C++

```
void f2( Body& x) // x is a reference
{
  x.setVol( 42);
}


// Call

f2(y);                // y is modified
```

Vergleich Java/C++ — Referenz

Achtung: In C++ kann man eine Referenz auf ein Objekt nach der Erzeugung nicht mehr ändern

Java

```
Body a;
Body b;
Body c = a;     // c refers to a

c = b;          // now, c
                // refers to b
```

= / ≠

C++

```
Body a;
Body b;
Body& c = a;    // c refers to a

c = b;          // b is copied to c
                // (which is a)

Body& d;        // ERROR! d points
                // nowhere
```

In C++ ist eine Referenz ein anderer Name ("Aliasname") für ein bereits existierendes Objekt.

Vergleich Java/C++ – Ausführung

C++

- Compiler: Quelltext wird kompiliert (= übersetzt in Maschinencode):
  - \+ Optimale Leistung möglich.
  - – Wenn das Programm auf unterschiedlichen Betriebssystemen (Linux, Windows, Mac, ...) oder Architekturen (Sparc, ARM, X86) laufen soll, dann muss man auch unterschiedliche Varianten verwalten.

Java

- Interpreter/Just-in-time Compiler: Java Byte-Code ist Grundlage der Ausführung durch die Java Virtual Machine:
  - \+ Der Byte-Code kann (ohne weiteres) auf allen Architekturen ausgeführt werden, für die eine JVM verfügbar ist.
  - \+ Moderne JVMs haben Just-in-time Compiler: wird Byte-Code oft genug ausgeführt, dann wird er zur Laufzeit kompiliert und danach als Maschinencode ausgeführt → optimale Leistung nach einer gewissen Laufzeit.
  - – Overhead durch Speicherverwaltung und geprüfte Speicherzugriffe.