# Technische Informatik 2

## C/C++ Programmierung
## 1. Einführung

Prof. Dr. Ivo Wolf

Institut für Medizinische Informatik

hochschule mannheim



## Literatur zu C / C++

- B. Kernighan, D. Ritchie: Programmieren in C; Hanser Verlag 1990, 32,90 €
  - Das Standardwerk für C. Kein Lehrbuch, aber wenn man C mal kann, das einzig „wahre“, weil dort alle Zweifelsfälle bzgl. ANSI-C geklärt sind.
- Bjarne Stroustrup: Die C++ Programmiersprache; Addison-Wesley, 4. Auflage, 2009
  - Das Standardwerk für C++
- Ulrich Breymann: Der C++-Programmierer; Carl Hanser Verlag, 2. Auflage, 2011 dx.doi.org/10.3139/9783446428416
- Dietrich May: Grundkurs Software-Entwicklung mit C++; Springer, 2. Auflage, 2006 dx.doi.org/10.1007/978-3-8348-9022-1
  - sehr ausführlich, eher für einzelne (Teil-)Kapitel

# Literatur zu C / C++

Im Internet:

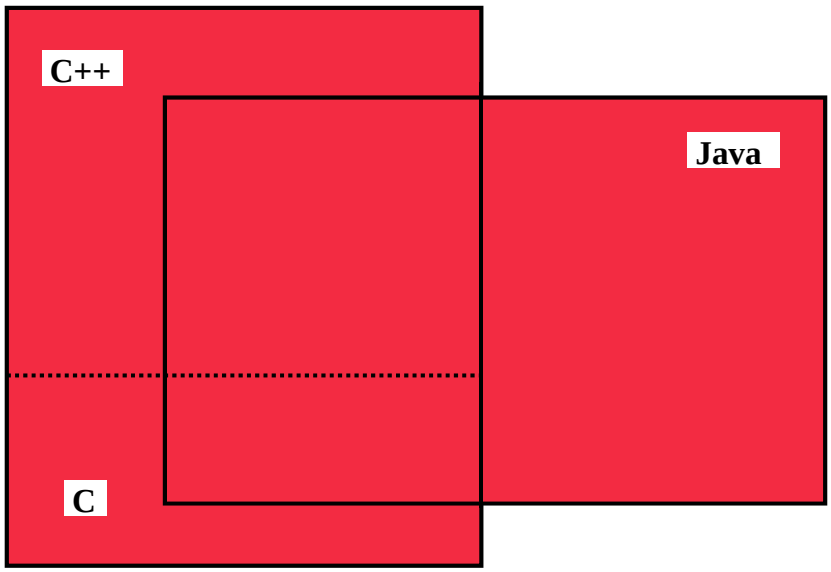
- de.wikibooks.org/wiki/C-Programmierung
- de.wikibooks.org/wiki/C++-Programmierung
- www.cplusplus.com
- www.parashift.com/c++-faq-lite/
- www.cppreference.com
- www.cpp-tutor.de
- en.wikipedia.org/wiki/C++



# 1. Einführung

# 1. Hello World

# C, C++ und Java - Schnittmengen






# Was fehlt in C/C++ (gegenüber Java)?

In C gibt es keine Klassen/Objekte

In C++ gibt es Klassen, aber nicht:

- Interfaces
- Reflections
- Garbage Collection

Nur mit Zusatzbibliotheken gibt es:

- Graphik (inkl.GUI)
- Netzwerk
- Hilfen zur Dokumentation

# Hello World – Beispielprogramm in Java (verändert)

```
Beispielprogramm in Java, leicht verändert:
import static java.lang.System.*;
// Ausgabe von "Hello World!"
public class hello {
   public static void main (String args[]) {
     out.println("Hello world!");
   }
}
```



# Hello World – Beispielprogramm in Java und C

```
Beispielprogramm in Java:
import static java.lang.System.*;
// Ausgabe von "Hello World!"
public class hello {
   public static void main (String args[]) {
     out.println("Hello world!");
   }
}

Beispielprogramm in C:
#include  <stdio.h>
/* Ausgabe von "Hello world" */
int main(int argc, char* argv[]) {
   printf("Hello world!");
   return 0;
}
```

Java: alles in Klassen!

C: kennt *keine* Klassen!

… ist nicht wirklich das Gleiche, nur in etwa ...

*globale* Funktion!

(Erfolgs-)Meldung an das Betriebssystem

# Hello World – Beispielprogramm in Java und C++

Beispielprogramm in Java:

```
import static java.lang.System.*;
// Ausgabe von "Hello World!"
public class hello {
   public static void main (String args[]) {
      out.println("Hello world!");
   }
}
```

Beispielprogramm in C++:

```
/* wie in C. Besser aber in modernerem Stil:*/
#include  <iostream>
using namespace std;
// Ausgabe von "Hello World!"
int main(int argc, char* argv[]) {
   cout << "Hello world!";
   return 0;
}
```

In C++ gibt es Klassen, aber hier braucht man nicht unbedingt eine …



# Hello World – Beispielprogramm in Java und C++

Beispielprogramm in Java:

```
public class hello {
   public static void main (String args[]) {
      System.out.println("Hello world!");
   }
}
```

Variante des Beispielprogramm in C++ (mit Klasse):

```
#include  <iostream>
class hello {
 public:
   static void kuenstlichesMain() {
      std::cout << "Hello world!";
   }
};

int main(int argc, char* argv[]) {
  hello::kuenstlichesMain();
  return 0;
}
```

Weiterhin geht es in der *globalen* Funktion main los!

# Lokale Variablen in C

- In C MÜSSEN alle lokalen Variablen am Anfang des Blocks (vor der ersten Anweisung) definiert werden:

```
#include  <stdio.h>
int main(int argc, char* argv[]) {
   int a=2;
   int b=a+1; // so ist es richtig.
   printf("Hello world!");
   int c=a+1; // Fehler in C (in C++ ok)
   return 0;
}
```



# Sprachumfang gering: Schlüsselwörter in C

```
auto     default  float    long      sizeof   union

break    do       for      register  static   unsigned

case     double   goto     return    struct   void

char     else     if       short     switch   volatile

const    enum     int      signed    typedef  while

continue extern

In C99 zusätzlich:
inline   restrict _Bool    _Complex  _Imaginary
```

… alles andere gehört strenggenommen nicht zur Sprache, sondern ist schon eine Erweiterung (die mit #include eingebunden wird) …

# Ausgabe: printf = print formatted

## Syntax:

```
#include <stdio.h> /*nicht vergessen …*/
int printf (Formatstring[, Argumentliste]);
```

optional

Formatstring → String(konstante) mit Formatelementen.
Argumentliste → [Argument [, Argument …]].

Die „eckigen Klammern“ stehen für: alles drin ist optional

## Semantik:

Ausgabe auf Standardausgabe. Dabei werden die im Formatstring enthaltenen **Formatelemente** durch die Werte aus der Argumentliste ersetzt.
Ein Formatelement bestimmt, wie ein Element aus der Argumentliste interpretiert und ausgegeben wird. Die Zuordnung Formatelement - Argument erfolgt von links nach rechts.

Rückgabewert: Anzahl der ausgegebenen Zeichen. Im Fehlerfall EOF.



# Ausgabe: printf = print formatted

## Formatelemente

Beispiel:

```
int a=1; float b=42.17;
printf ("Ein int:%i. Ein float:%f.",a,b);
// Ein int:1. Ein float:42.17.
```

## Syntax:

Formatelement → "%" [Flag] [ Feldbreite] ["."Genauigkeit] Typ

- i.w. Vorzeichen
- Länge des Ausgabefelds
- Mindestanzahl Ziffern / Zeichen
- legt fest, wie das zugeordnete Argument interpretiert und umgewandelt wird

# Ausgabe: printf = print formatted

| Typ | Argumenttyp | Ausgabe |
|---|---|---|
| d,i | int | dezimal |
| u | unsigned int | dezimal |
| o | unsigned int | oktal |
| x | unsigned int | hexadezimal (mit Kleinbuchstaben) |
| X | unsigned int | hexadezimal (mit Großbuchstaben) |
| f | float / double | Gleitkommazahl |
| e, E | float / double | Exponentialdarstellung |
| g,G | float / double | Gleitkommazahl /Exponentialdar-stellung (die kürzere von beiden) |
| c | char / int | einzelnes Zeichen |
| s | String | String bis '\0' (max. Genauigkeit viele Zeichen) |
| n | int * | Speichert im entsprechenden Argument die Anzahl der bis dahin ausgegebenen Zeichen |
| p | Zeiger | Ausgabe der im entspr. Argument enthaltenen Adresse (hexadez.) |
| % | keiner | Ausgabe des Zeichens % |

- Typ und tatsächlicher Typ des Arguments müssen übereinstimmen!
  **Achtung**: dies wird i.A. nicht überprüft! (Manche Compiler liefern Warnung)



# Ausgabe: printf = print formatted

```
// verwendete Variablen:
int a = 63; unsigned b = 40000; long n = 100;
  double x=123.4567;
char *s = "ein String";

printf ("%d %i %o %x %X %c \n", a, a, a, a, a, a);
//       63 63 77 3f 3F ?

printf ("%d %u %X \n", b, b, b);
//      -25536 40000 9C40

printf ("%f %E %g \n", x, x, x);
//       123.456700 1.234567E+02 123.457

printf ("<%5d>--<%05d> \n", a, a);
//       <   63>--<00063>

printf ("<%-5d>--<%+-5d>--<% d>--<% d>\n", a, a, a, -a);
//       <63   >--<+63  >--< 63>--<-63>

printf ("<%.2f>--<%10.2f>--<%+10.5f>--<%-10.3f>\n", x, x, x, x);
//      <123.46>--<    123.46>--<+123.45670>--<123.456   >
```

# printf: Flags*

**Flags** ⟶ ["+"] [ " "] ["-"] ["0"].

**Wirkung:**

| | |
|---|---|
| + | Positive Zahlen werden mit Vorzeichen ausgegeben. |
| " " | Positive Zahlen werden mit führendem Blank ausgegeben. |
| - | Im Feld wird linksbündig ausgegeben. |
| 0 | Das Feld wird vor der Zahl mit 0 aufgefüllt. |

**Feldbreite** ⟶ positive Ganzzahl | "*".

**Wirkung:**

Legt die Länge des Ausgabefelds fest. I.d.R. wird rechtsbündig ausgegeben. Verbleibender Platz wird mit Blanks aufgefüllt.
Ist die auszugebende Zeichenfolge länger als das Feld, so wird sie vollständig ausgegeben (die Feldbreite wird vergrößert).
Bei Feldbreite = "*" wird die Feldbreite durch ein zusätzliches Argument bestimmt, das vor dem auszugebenden Argument stehen und den Typ int haben muss.



# printf: Angabe der Genauigkeit*

Genauigkeit → positive Ganzzahl.
Wirkung:

- bei Gleitpunktzahlen (Voreinstellung: 6)
  - Typ f, e: Anzahl der Ziffern, die hinter dem Dezimalpunkt ausgegeben werden.
  - Typ g: Anzahl der signifikanten Stellen, d.h. die Gesamtanzahl von Ziffern; falls nötig wird gerundet.
- bei Ganzzahlen (Voreinstellung: 1)
  - Mindestanzahl von auszugebenden Ziffern.
  - Bei weniger vorhandenen Ziffern: es werden Nullen vorangestellt.
- bei Strings
  - Höchstanzahl der Zeichen, die ausgegeben werden.

## Ausgabe: spezielle Formatelemente

- \n Zeilenumbruch
- \t Tabulator
- \\ Backslash (\)
- \" Doppelte Anführungszeichen (")



## 1. Einführung

# 2. Der Kompiliervorgang

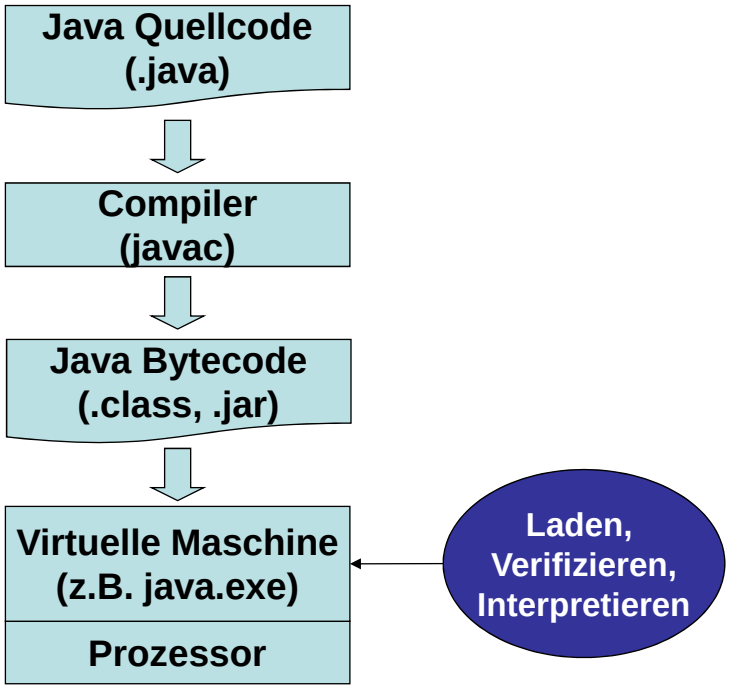
# Aus der GDI-Vorlesung … Der Java-Kompiliervorgang

- Kompilieren
  **javac Hello.java**
  (erzeugt Hello.class)

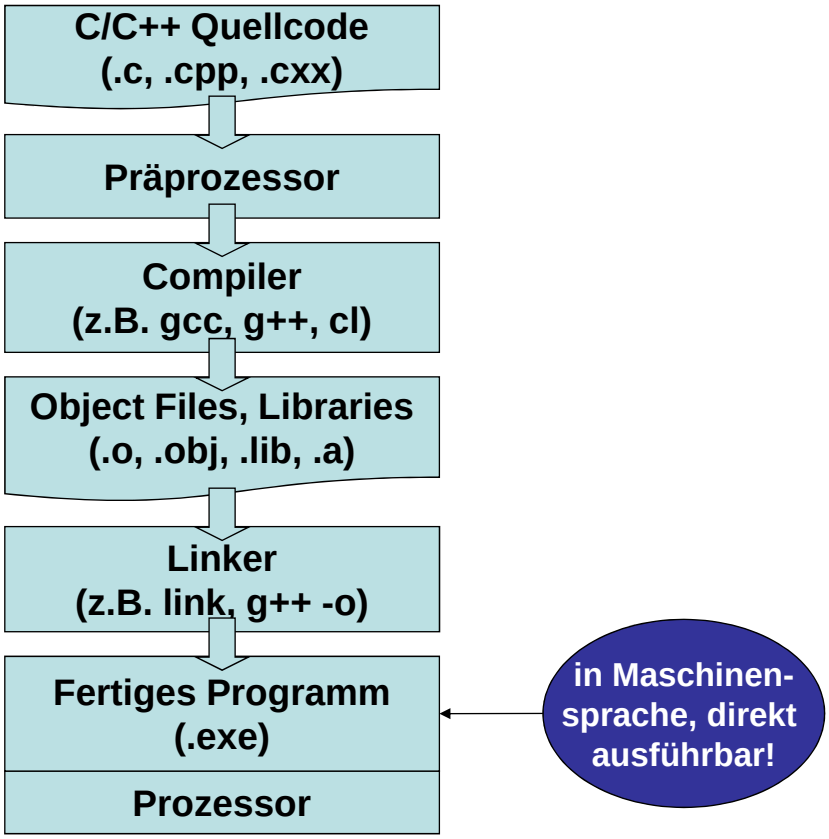
- Ausführen
  **java Hello**
  (startet Java-VM mit Hello.class)



# Aus der GDI-Vorlesung … Der Java-Kompiliervorgang



# Der C/C++ Kompiliervorgang im Überblick






# Der C/C++ Kompiliervorgang

- Kompilieren (einschließlich Präprozessor)
  **gcc -c Hello.c**
  (erzeugt Hello.o bzw. .obj)

- Linken:
  **gcc -o Hello Hello.o** [ggf. weitere .o Dateien]
  (erzeugt ausführbare Datei Hello bzw. Hello.exe)

- Ausführen
  **Hello**
  bzw. unter Unix/Linux meist: **./Hello**
  **(**Der Punkt steht für das aktuelle Verzeichnis.)

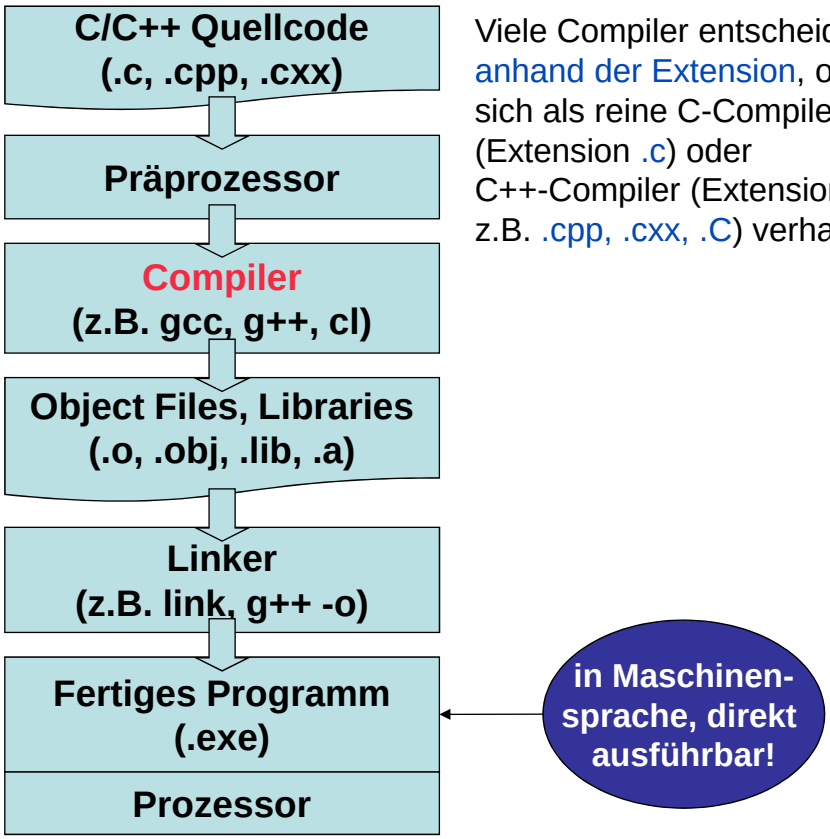
# 1. Einführung

## 2. Der Kompiliervorgang

- Compiler
- Linker
- Präprozessor
- Makefiles* und CMake*

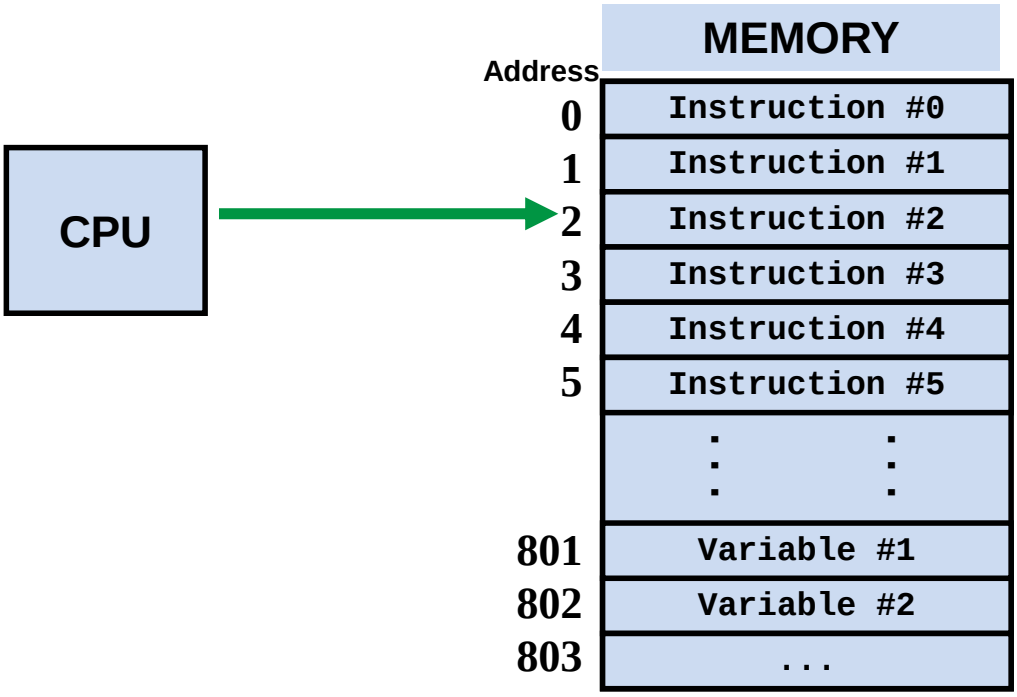


# Der C/C++ Kompiliervorgang im Überblick



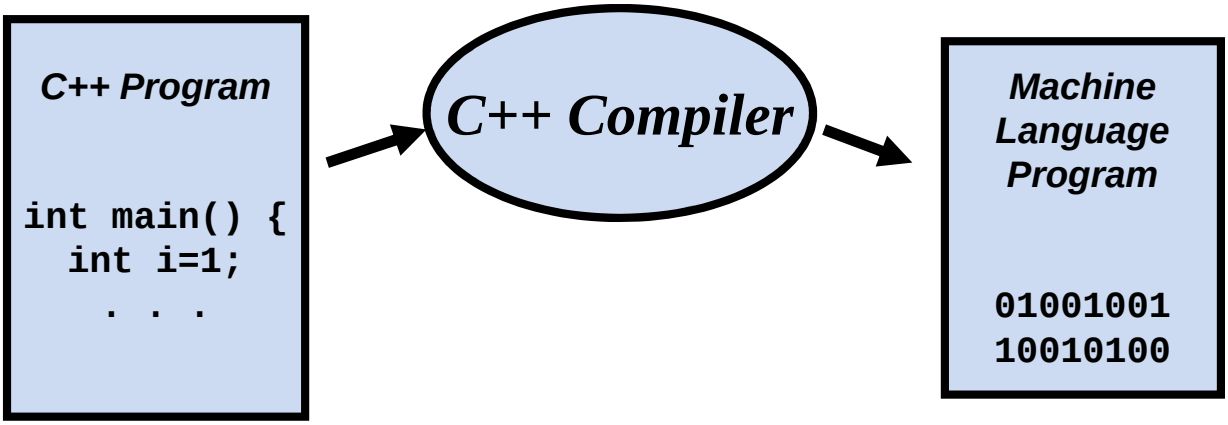

Viele Compiler entscheiden anhand der Extension, ob sie sich als reine C-Compiler (Extension .c) oder C++-Compiler (Extensionen z.B. .cpp, .cxx, .C) verhalten.

# Another Picture






# Compiler



# Assembler und Maschinen Sprache: 1-zu-1 Übersetzung

| *Assembly Language* | *Machine Language* |
|---|---|
| `ST 1,[801]` | `00100101 11010011` |
| `ST 0,[802]` | `00100100 11010100` |
| `TOP: BEQ [802],10,BOT` | `10001010 01001001 11110000` |
| `INCR [802]` | `01000100 01010100` |
| `MUL [801],2,[803]` | `01001000 10100111 10100011` |
| `ST [803],[801]` | `11100101 10101011 00000010` |
| `JMP TOP` | `00101001` |
| `BOT: LD A,[801]` | `11010101` |
| `CALL PRINT` | `11010100 10101000` |
| | `10010001 01000100` |



# As a C/C++ program

```
set memory[801] to hold 00000001  <-------  x=1;
set memory[802] to hold 00000000  <-------  i=0;
if memory[802] = 10 jump to instruction #7 <---  while (i!=10) {
increment memory[802]  <-------------------------  i++;
set memory[803] to 2 times memory[801]  <-------  x=x*2;
put memory[803] in to memory[801]  <-------  }
jump to instruction #3  <-------
print memory[801]  <-------------------------  printf("%d",x);
```

# Immer Variablen initialisieren!

- C/C++ Compiler erzeugen nicht automatisch Code zur Initialisierung von Variablen!
- Also:
  Variablen müssen immer selbst initialisiert werden!

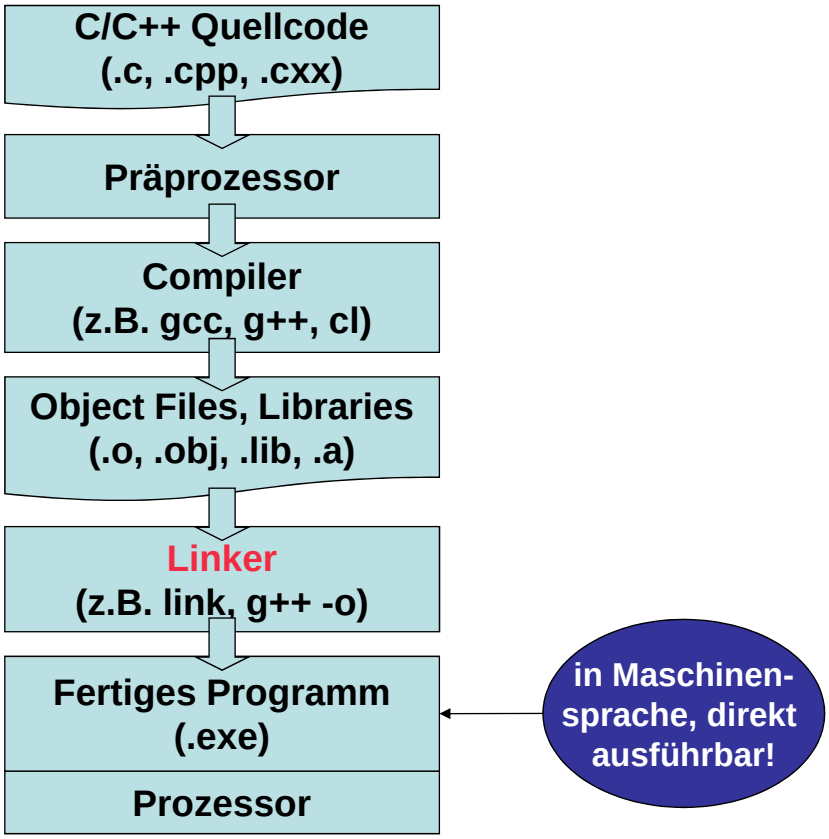


# 1. Einführung

# 2. Der Kompiliervorgang

- **Compiler**
- **Linker**
- **Präprozessor**
- **Makefiles* und CMake***

# Der C/C++ Kompiliervorgang im Überblick






# Zurück zu unserem Beispielprogramm

```
#include  <stdio.h>
/* Ausgabe von „Hello world“ */
int main(int argc, char* argv[]) {
   printf("Hello world!");
   return 0;
  }
```

- Was steht nun über printf in stdio.h?

  … nicht viel, nur die Signatur!

  ```
  int printf(const char * _Format, ...);
  ```

- Wo ist der Code?
  - irgendwo anders …
  - könnte in einem anderen .c-File sein, das auch kompiliert wird (oder wurde)
  - oder direkt in Maschinensprache geschrieben sein.

# Der Linker

Aufgabe des Linkers (engl. link = verbinden):

- Verbinden von mehreren kompilierten Dateien
- Die Verweise auf Funktionen/Klassen in anderen Dateien werden durch konkrete Werte aufgelöst
  - z.B. aus „rufe printf auf“ wird dann CALL 451 (451 soll die Adresse sein)
- „Gelinkt“ werden
  - Object-Files oder
  - Libraries = mehrere Object-Files in einer Datei (entspricht .jar in Java)
- Es gibt statische und dynamische Libraries:
  - statisch: Code landet in der Programmdatei
  - dynamisch: eigene Datei, die erst beim Programmstart endgültig verbunden wird



# Einbinden von Libraries

- Standard C Library:
  - wird standardmäßig hinzugelinkt
  - typischer Name: `libc, msvcrt`
  - enthält etwa 200 Funktionen, u.a. `printf` ...
- Andere Bibliotheken: Linker muss wissen, was er hinzulinken soll.
  - Eintrag in Dialogbox in Entwicklungsumgebung
  - oder Kommandozeilen-Parameter
  - Bei Arduino-IDE automatisch:
    - Bibliotheken liegen zusammen mit ihren Header-Dateien in einem Verzeichnis.
    - Wird die Header-Datei per `#include` eingebunden, wird die Bibliothek gelinkt.

# Einbinden von Libraries – Kommandozeile

- Typische Kommandozeilen-Option: `-llibrary`

  `gcc … -lmylib`

  – Also ohne Leerzeichen nach dem `-l`

  – gcc: lib *nach* den Dateien angeben, von denen die lib verwendet wird!

  – Linux/Unix: Name der Datei ist libmylib.a (statisches einbinden) oder libmylib.so (dynamisches einbinden)

  ➔ das „lib“ wird nach `-l` weggelassen!

  Beispiel: `-lm` für die Mathe-Lib `libm.a`



# Kompilieren und Linken auf der Kommandozeile

- Kompilieren (einschließlich Präprozessor)
  **gcc -c Hello.c Func.c**
  – erzeugt Hello.o und Func.o (bzw. .obj)
- Linken:
  **gcc -o MeinHello Hello.o Func.o -lmeineLib**
  – erzeugt ausführbare Datei MeinHello (bzw. Hello.exe); verbindet Maschinen-Code der Funktionen/Klassen aus Hello.o, Func.o und der Bibliothek libmeineLib
- Ausführen
  **MeinHello**
  bzw. unter Unix/Linux meist: **./MeinHello**
  **(**Der Punkt steht für das aktuelle Verzeichnis.)

Vorgegebene Kommando-Zeilen sollten Sie um weitere Dateien (Quellcode-Datei, Bibliotheken) ergänzen können.

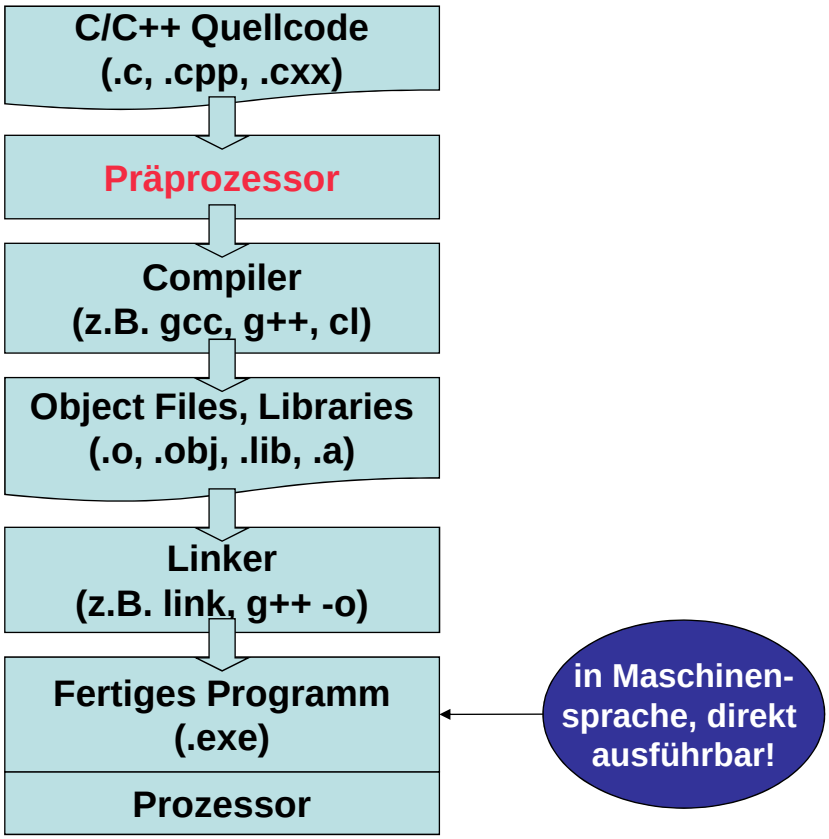
# 1. Einführung

## 2. Der Kompiliervorgang

- Compiler
- Linker
- Präprozessor
- Makefiles* und CMake*



# Der C/C++ Kompiliervorgang im Überblick



# Der Präprozessor

- … ist dem Compiler vorgeschaltet („Prä“-).
- … erkennt und verarbeitet spezielle Anweisungen im Quelltext (sog. Präprozessor-Anweisungen)
- … verändert den Quelltext, bevor er vom Compiler übersetzt wird.

**Zweck:**

- Bessere Lesbarkeit: kompakterer Quelltext
- Bessere Portierbarkeit: Verstecken von maschinen-abhängigen Details
- Geheimhaltung: nur Signaturen bekanntgeben (reicht dem Compiler aus)
- Schneller: nur Signaturen parsen statt kompletter Implementierung



# Präprozessor-Anweisungen: Allgemeine Syntax

Präprozessor-Anweisungen:

- beginnen mit #
- unmittelbar (ohne Leerzeichen) nach #: gewünschte Präprozessor-Anweisung
- vor # nur Leerzeichen/Tabs erlaubt
- normalerweise einzeilig
- sind mehrere Zeilen erforderlich: \ als letztes Zeichen in der Zeile

# Arten von Präprozessor-Anweisungen

1. Kopieren von externen Dateien in die zu übersetzende Datei (`#include`)
2. Textuelles Ersetzen von symbolischen Konstanten und Makros durch zuvor definierte Werte (`#define`)
3. Bedingte Übersetzung (`#ifdef` ...)



# Präprozessor-Anweisungen: 1. Einbinden von Dateien: #include

**Syntax:** **#include <**dateiname**>** **oder** **#include "**dateiname**"**

Datei wird NUR in speziellen Verzeichnissen gesucht: sog. „include-Verzeichnisse“

Datei wird zuerst im aktuellem Verzeichnis gesucht; falls dort nicht gefunden: Suche ebenfalls in den „include-Verzeichnissen“

**Semantik:** An der Stelle, an der die include-Anweisung steht, wird die angegebene externe Datei in die zu übersetzende Datei kopiert.

**Verwendung:** Informationen, die für mehrere Programme von Interesse sind an zentraler Stelle verfügbar machen.

**wichtigste Anwendung:** Einbindung von Headerdateien (z.B. Nutzung der mitgelieferten Laufzeitbibliotheken).

# Präprozessor-Anweisungen: 1. Einbinden von Dateien: #include

Quelldatei:

```
Zeile 1
#include "b.h"
Zeile 2
#include "c.h"
Zeile 3
```

```
Zeile 1
Zeile b1
Zeile b2
Zeile 2
Zeile c1
Zeile c2
Zeile 3
```

Datei b.h:

```
Zeile b1
Zeile b2
```

Datei c.h:

```
Zeile c1
Zeile c2
```

Eingabe für Compiler nach Präprozessorlauf



# #include von Header-Dateien: Header enthalten nur „Signaturen“

Der Compiler *muss* zur Verwendung einer Funktion oder Klasse (nur/mindestens) die Signatur kennen!

- Beispiel: damit

```
double y = sqrt(17);
```

funktioniert, muss irgendwo vorher mindestens die Signatur von `sqrt` stehen, also:

```
double sqrt(double x);
```

  - Sie können das selbst hinschreiben …
  - … normalerweise bindet man aber den entsprechenden Header ein, in dem genau das steht (und vieles mehr) :

```
#include <math.h>
```

# Header-Dateien: Klassen meist aufgeteilt in .h und .cpp

**Vector.h**

```
class Vector
{
public:
    Vector();
    float getLength();
private:
    float x,y;
};
```

**Vector.cpp**

```
#include "Vector.h"
#include <math.h>

Vector::Vector()
{
  x=0; y=0;
}
float Vector::getLength()
{
  return sqrt(x*x+y*y);
}
```



# Präprozessor-Anweisungen: 2. Konstantendefinition: #define

Konstantendefinition: Einfachste Form der #define-Anweisung.

**Beispiele:**

```
#define EINS 1
```

Alle Vorkommen von EINS im Bezeichnerkontext eines Programms werden durch 1 ersetzt.

```
#define forever for (;;)    /* Endlosschleife */
```

vernünftige Konstanten:

```
#define TRUE 1
#define FALSE 0
#define PI 3.1415926535
#define AND &&
#define OR ||
#define NOT !
```

# Präprozessor-Anweisungen: 2. Makrodefinition mit #define

**Syntax:** **#define** name(Parameterliste) Anweisungen…

optional optional

**Semantik:** name = Zeichenkette, die vor der Übersetzung des Programms in andere Zeichenketten (aus-)getauscht wird.

**textuelle Ersetzung !**

Austausch nur, wenn Zeichenkette alleinstehender „Token“ ist, nicht in zusammengesetzten Wörtern und Strings.

Sichtbarkeit ab Definitionsende bis Dateiende.

**#undef ...** Definition (ab diesem Punkt) außer Kraft setzen

**Begriffe:**

**späteres Austauschen** Makrosubstitution

**definierte Zeichenkette** Makro



# Präprozessor-Anweisungen: 2. Makrodefinition mit #define

```
#define KLEINER_NULL(a) a < 0
```

**kein Semikolon! (denn sonst wird das Semikolon mit eingesetzt)**

```
if (KLEINER_NULL(i) || KLEINER_NULL(x))
```

Wird durch den Präprozessor durch Folgendes ersetzt:

```
if ( i < 0 || x < 0)
```

# Präprozessor-Anweisungen: Makros
## Parameter in Klammern setzen!!

```
#define SCHALTJAHR(a)\
(a)%4 == 0 && (a)%100 !=0 || (a)%400 ==0
```

**Wichtig: Klammerung des Parameters**

```
if (SCHALTJAHR(2001)) . . .
if (SCHALTJAHR(2002)) . . .
if (SCHALTJAHR(2003)) . . .
if (SCHALTJAHR(2004)) . . .
if (SCHALTJAHR(2001+x)) . . .
```

```
(2001+x)%4 == 0 && (2001+x)%100 !=0 || (2001+x)%400 == 0
```

Ohne Klammerung wäre das Ergebnis:

```
2001+x %4 == 0 && 2001+x % 100 !=0 || 2001+x %400 == 0
```

also würden zuerst die Modulo-Operationen berechnet (Punkt vor Strich!):

```
2001+(x %4) == 0 && 2001+(x % 100) !=0 || 2001+(x %400) == 0
```

(Die orangen Klammern dienen nur zur Verdeutlichung der Rechenreihenfolge!)



# Makros vs. Funktionen

- Argumente von Makros können beliebige Datentypen annehmen.
- Argumente können auch Typbezeichner sein.
- Makros sind effizienter (vermeiden Laufzeit-Overhead).
- Der Quelltext wird umfangreicher, da jeder Makroaufruf eine textuelle Expansion bedingt.
- Rekursion ist nicht möglich.
- Komplizierte Makros → fehleranfällig.
- I.d.R. sollten Makros nur für sehr kurze Funktionen eingesetzt werden.

# Präprozessor-Anweisungen: 3. Bedingte Übersetzung: #ifdef

Anwendungen von `#ifdef`:

- Verhindern, dass Header-Dateien mehrfach eingebunden werden: Header Guards
- Debug-Ausgaben während der Entwicklung
- Programme mit plattformabhängigen (maschinenabhängigen) Programmteilen (Betriebssystem, Hardware, …)
  - Man möchte trotzdem 2 Versionen des Quelltextes vermeiden, da sonst die Gefahr besteht, dass bei Änderungen nicht synchron verfahren wird.

➔ Bedingte Übersetzung



# Präprozessor-Anweisungen: Bedingte Übersetzung: #ifdef

Bedingte Übersetzung:

- Hierbei werden bestimmte Teile des Quellcode mittels Präprozessor-Direktiven von der Compilation ausgeschlossen.
- Benötigte Direktiven:
  - #ifdef MYCONST
    (mit einer Präprozessor-Konstante MYCONST)
  - Alternativ: #if defined(MYCONST)
  - #ifndef für nicht-definiert
  - Vergleich mit Wert: #if MYCONST == 17
  - #elif Ausdruck (z.B. #elif defined(MYCONST) )
  - #else
  - #endif

# Präprozessor-Anweisungen: 3. Bedingte Übersetzung: #ifdef

Verwendung:

```
#define UNIX  /* nur diese Stelle muss geändert werden */

#ifdef W2K
        Programmtext1 mit Windows2000 spezifischen Anweisungen
#elif defined(UNIX)
        Programmtext2 mit Unix spezifischen Anweisungen
#endif
```

Noch besser:

- Präprozessor-Definitionen können dem Compiler (genauer: dem Präprozessor) auch auf der Kommandozeile übergeben werden.

→ gar keine Änderung am Quelltext notwendig!

→ nur der Compiler-Aufruf muss je nach Plattform anders sein
Beispiel für gcc: `gcc -Dmeindefine …`
(also ohne Leerzeichen hinter dem `-D`)



# Präprozessor-Anweisungen: 3. Bedingte Übersetzung: #ifdef

Problem in der Praxis: Beim Testen streut man eigene Debug-Anweisungen in den Programmtext ein (print Anweisungen).

Diese nur zum Testen des Programm notwendigen Ausgabeanweisungen kann man mittels der #ifdef-Direktive ein- bzw. ausschalten.

```
#define DEBUG  /* nur diese Stelle muss geändert werden */
......
#ifdef DEBUG
   printf (....);
   printf (....);
#endif
```

# Präprozessor-Anweisungen: 3. Bedingte Übersetzung: #ifdef

Variante: Macro zu Debug-Zwecken

```
#define DEBUG  /* nur diese Stelle muss geändert werden */
......
#ifdef DEBUG
  #define DEBUGMACRO(text) printf(text)
#else
  #define DEBUGMACRO(text)
#endif
```



# Header-Guards mit #ifdef

- Header-Guards: Verhindern doppeltes Einbinden von Headern

Vector.h

```
#ifndef __VECTOR_HEADER__
#define __VECTOR_HEADER__

class Vector
{
public:
    Vector();
    float getLength();
private:
    float x,y;
};
#endif /*__VECTOR_HEADER__*/
```

Vector.cpp

```
#include "Vector.h"
#include "VectorHelper.h"
/* inkludiert evtl. ebenfalls
Vector.h*/
#include <math.h>
Vector::Vector()
{
  x=0; y=0;
}
float Vector::getLength()
{
  return sqrt(x*x+y*y);
}
```

# 1. Einführung

## 2. Der Kompiliervorgang

- Compiler
- Linker
- Präprozessor
- Makefiles und CMake



## Steuerung der Kompilierung

- Kleine Änderung an einer Datei …
  - ➔ alles neu kompilieren und linken?
  - ➔ dauert viel zu lange!
- Nur das Kompilieren und Linken was nötig ist!
- Wer weiß, was nötig ist?
  - Entweder manuell vorgeben oder
  - Automatische Analyse des Quelltextes
- Beispiel: ein Header wurde geändert ➔ alle Dateien, die diesen einbinden, neu übersetzen

# Makefiles

## Makefiles:

- Dienen der Steuerung, was kompiliert und gelinkt werden muss
- Diese Aufgabe wird teilweise auch von Entwicklungsumgebungen übernommen
- Viele Varianten
- Oft auch generiert von Programmen, die den Quellcode analysieren
- Dateiname des Makefile: „Makefile“



# Klassisches Makefile

```
prog: prog.c prog.h
      gcc -o prog prog.c
```

## Bedeutet:

- Es geht um die Erstellung von `prog`
- Falls sich `prog.c` oder `prog.h` geändert haben (neuer sind als `prog`), muss die nachfolgende Zeile ausgeführt werden (kompiliert und gelinkt werden).

# Klassisches Makefile mit Regeln*

```
%.o: %.c
     gcc -c $<
```

Bedeutet:

- Falls eine `.c`-Datei neuer ist als die `.o`-Datei gleichen Namens (abgesehen vom Suffix), muss die nachfolgende Zeile ausgeführt werden (kompiliert werden).
- $< bedeutet: die erste Abhängigkeit
- $+ bedeutet: alle Abhängigkeiten
- Die erste Zeile kann auch lauten: `.c.o`
- Mehr zu klassischen Makefiles unter: http://www.ijon.de/comp/tutorials/makefile.html



# Ein typisches klassisches Makefile*

```
CC = /usr/bin/gcc
CFLAGS = -Wall -g -D_REENTRANT
LDFLAGS = -lm -lpthread

OBJ = datei1.o datei2.o datei3.o datei4.o

prog: $(OBJ)
     $(CC) $(CFLAGS) -o prog $(OBJ) $(LDFLAGS)

%.o: %.c
     $(CC) $(CFLAGS) -c $<
```

# Makefile-Generatoren

- Bei klassischen Makefiles muss man die Abhängigkeiten selbst eingeben:

```
prog: prog.c prog.h
    gcc -o prog prog.c
```

- Makefile-Generatoren
  - analysieren den Quellcode und extrahieren die Abhängigkeiten
  - erzeugen Makefiles
  - die ggf. sogar den Generator aufrufen, um die Abhängigkeiten zu aktualisieren
- Beispiele: CMake, qmake, automake



# 1. Einführung

# 2. Der Kompiliervorgang

- **Compiler**
- **Linker**
- **Präprozessor**
- **Makefiles und CMake**

# CMake (www.cmake.org)

- Plattformunabhängige Makefiles
- Generiert Projektfiles für Visual Studio, Eclipse, Qt Creator (kann cmake nutzen)…
- Verwendet u.a. von KDE
- Anbindung an Test-Umgebung (ctest, CDash)



| Site | Build Name | Update | | Configure | | | Build | | | Test | | | | Build Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Files | Min | Error | Warn | Min | Error | Warn | Min | NotRun | Fail | Pass | Min | |
| v20n17.pbm.ihost.com | AIX00C518-xlC | 75 | 0.8 | 0 | 0 | 2.8 | 0 | 0 | 85.1 | 0 | 0 | 612 | 22.8 | 2009-03-06T02:17:22 EST |
| heart | HP-UXia64-aCC | 9 | 0.3 | 0 | 0 | 1.7 | 0 | 0 | 28.2 | 0 | 0 | 44 | 0.4 | 2009-03-06T03:18:58 EST |
| DASH6.kitware | Linux-g++ | 12 | 0.2 | 0 | 0 | 0.1 | 0 | 0 | 87.2 | 0 | 0 | 700 | 41.7 | 2009-03-05T23:06:40 EST |
| blackrose.sandia.gov | Linux-g++-debug | 9 | 1.8 | 0 | 0 | 0.4 | 0 | 0 | 18.2 | 0 | 0 | 1072 | 22.9 | 2009-03-05T23:05:13 EST |
| DASH6.kitware | Linux-g++-static | 0 | 0.1 | 0 | 0 | 0.1 | 0 | 0 | 153.2 | 0 | 2 +2 | 698 +2 | 55.2 | 2009-03-06T02:07:27 EST |
| gargon.hooperlab | Linux-gcc-4.3.2-x86_64-DevelMesa | 9 | 0.4 | 0 | 0 | 0 | 0 | 0 | 30.4 | 0 | 0 | 721 | 9.6 | 2009-03-06T00:20:34 EST |



# CMake

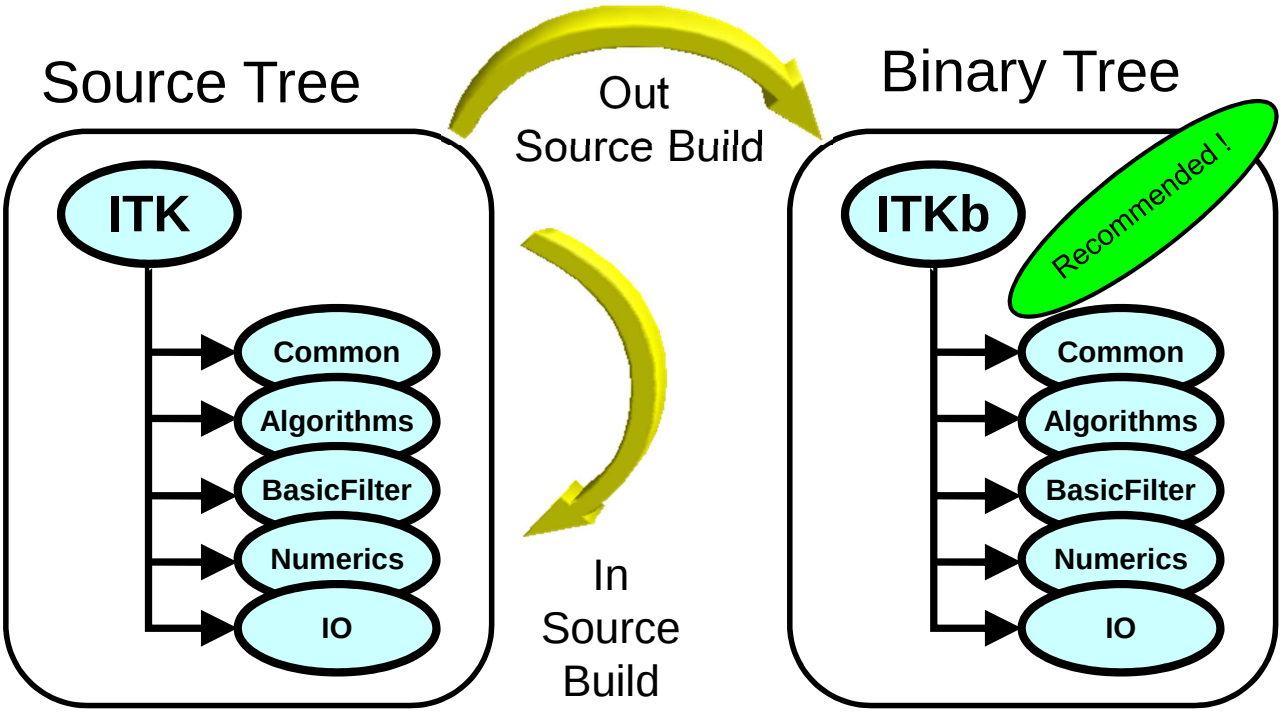
**C++ Compiler:**
**gcc 4.X**
**Visual C++**
**Eclipse**
**Qt Creator**
**…**

**Plattform-unabhängiges Make-Tool:**

**CMake**

Erzeugt Projekt bzw. Makefiles für die eingesetzte Plattform/den verwendeten Compiler/die verwendete Entwicklungsumgebung.

# Konfigurieren bei CMake

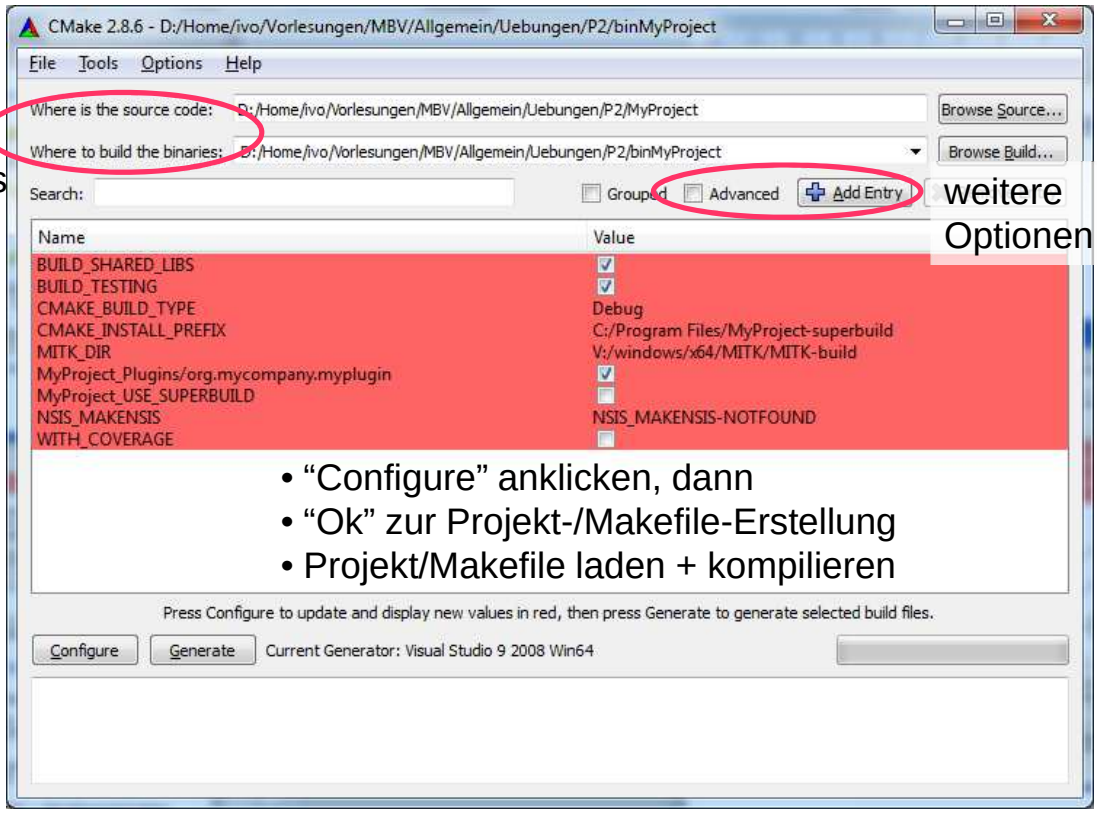





# Konfigurieren mit der CMake-Oberfläche (cmake-gui)

Source/
Binary-
Verzeichnis

Optionen




weitere
Optionen

- “Configure” anklicken, dann
- “Ok” zur Projekt-/Makefile-Erstellung
- Projekt/Makefile laden + kompilieren

## CMakeLists.txt

```
PROJECT( MeinProjektName )

INCLUDE_DIRECTORIES(
  ${PROJECT_SOURCE_DIR}
)

ADD_EXECUTABLE( MeinExecutableName
  meinSourceFile.cpp
)

TARGET_LINK_LIBRARIES ( MeinExecutableName
  eineWichtigeBibiliothek
)
```



## 1. Einführung

# 3. Strukturen

# Strukturen

- Strukturen sind:
  - Zusammenstellung von Variablen in einem neuen Typ
  - Wie ein Record einer Datenbank
  - (in C) Klassen ohne Methoden (Funktionen)
    (in C++ können sie auch Methoden enthalten)
- Gibt es in C und C++



# Strukturen – Beispiel

```
struct Data {
  int x;
  int y;
  double ave;
  double test[2];/* Array fester Größe*/
};
```

**“;” am Ende!**

# Benutzung von struct-Variablen

- Können auf dem Stack liegen:
  - dann kein “new” nötig!
  - Nutzung wie einfache Variablen möglich.

```
struct Data d;    // kein “new” nötig!
d.x = 7;          // direkt verwendbar!
d.test[0] = 2;
```

- Zuweisungen kopieren den Inhalt:

```
struct Data d2;
d2 = d;  // Kopiert (!) den Inhalt
```

  - Vorsicht: Wenn ein Member ein Zeiger ist, wird nur der Zeiger kopiert, nicht das, worauf der Zeiger zeigt!

- Member-Variablen werden nicht automatisch initialisiert!

- In C++ kann das Schlüsselwort `struct` bei der Instanziierung wegfallen.



# Einschub: typedef

typedef :

- Gibt einem Typ einen neuen Namen
- Beispiel:

```
typedef int GanzeZahl;
GanzeZahl a=17;
```

- In C für die Umbenennung von `structs` verwendet:

```
typedef struct Data DataType;
DataType d; //statt struct Data d;
```

# 1. Einführung

# 4. Unions



## Unions

Unions sind:

- wie Strukturen (`struct`), bei denen alle Member-Variablen an der selben Stelle gespeichert sind

➔ Zugriff auf denselben Speicherbereich auf verschiedene Weise möglich

```
union char2float
{
  char c[4];
  float f;
};
```

**“;” am Ende!**

## Unions – Beispiel

```
union char2float
{
  char c[4];
  float f;
};
```

“;” am Ende!

```
union char2float myvar;
myvar.f = 17.2;
myvar.c[0] = 2; // ändert 1. Byte von myvar.f
printf("%f", myvar.f); // nicht mehr 17.2 …
```

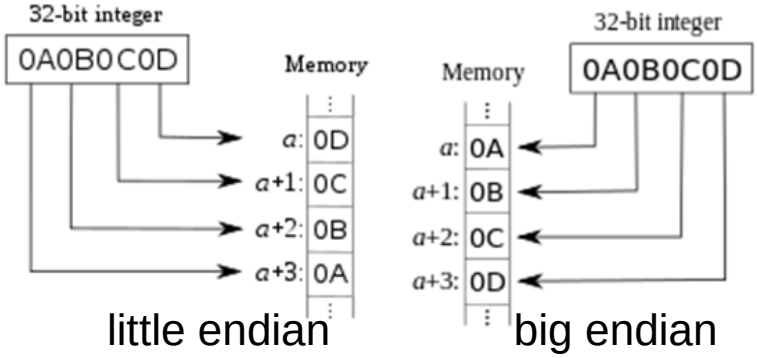


## 1. Einführung

# 5. Endianness

# Endianness

- Welches „Ende“ der Stellen einer Zahl wird zuerst genannt?
  - 103 – hundert-drei: größeres „Ende“ zuerst
    - → „big endian“
  - 42 – „zwei-und-vierzig“: kleineres „Ende“ zuerst
    - → „little endian“
- Beispiel 32-bit Integer:




little endian big endian



# Endianness – welche ist besser?

- Beide Varianten sind üblich:
  - Intel-Prozessoren sind little endian
  - Im Netzwerkbereich ist big endian weit verbreitet (z.B. IPv4, IPv6, TCP, UDP)
  - Oft sind die Gründe historisch (um Abwärts-Kompatibilität zu erreichen).
- Es gibt auch:
  - Bi-endian Prozessoren (umschaltbar)
  - Middle-endian