Einführung in die Programmiersprache C++

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Letzte Vorlesung

1.Einführung in C

- 1.1 Historisches
- **1.2 Struktur eines C-Programms**
  - **1.2.1 Hello World**
  - **1.2.2 Quellen / Objektcode / Linken**
  - **1.2.3 Der C-Präprozessor**
- 1.3 Sprachelemente
- 1.4 Zeiger
- 1.5 Benutzerdefinierte Datentypen
- 1.6 Weitere Sprachelemente

C - Hello World

- Erzeuge eine Datei `hello.c` z.B. mit `emacs`

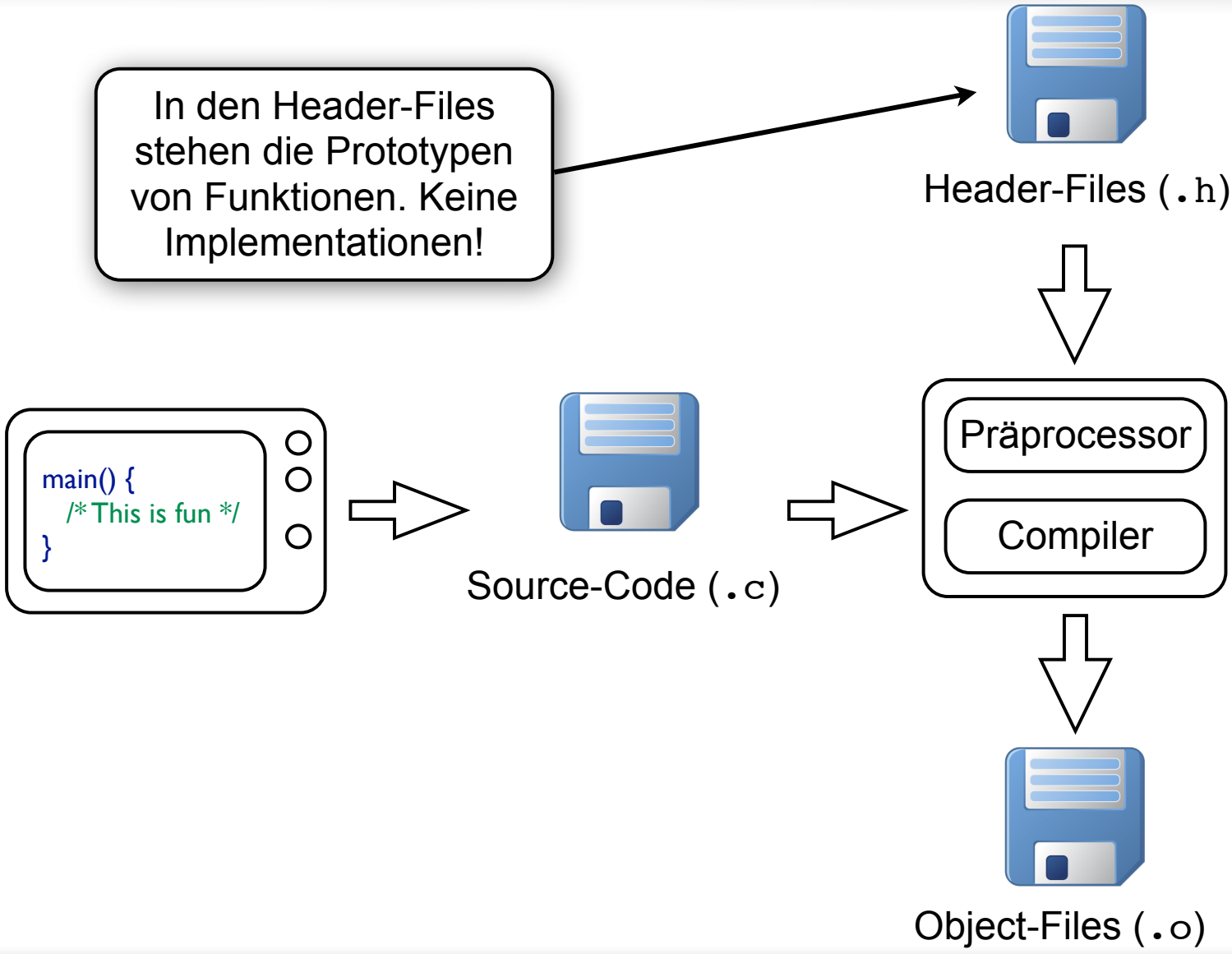
```c
#include <stdio.h>
int main(void)
{
    printf("hello, world!\n");
    return 0;
}
```

- Übersetze sie mit einem C-Compiler (z.B. `gcc`)

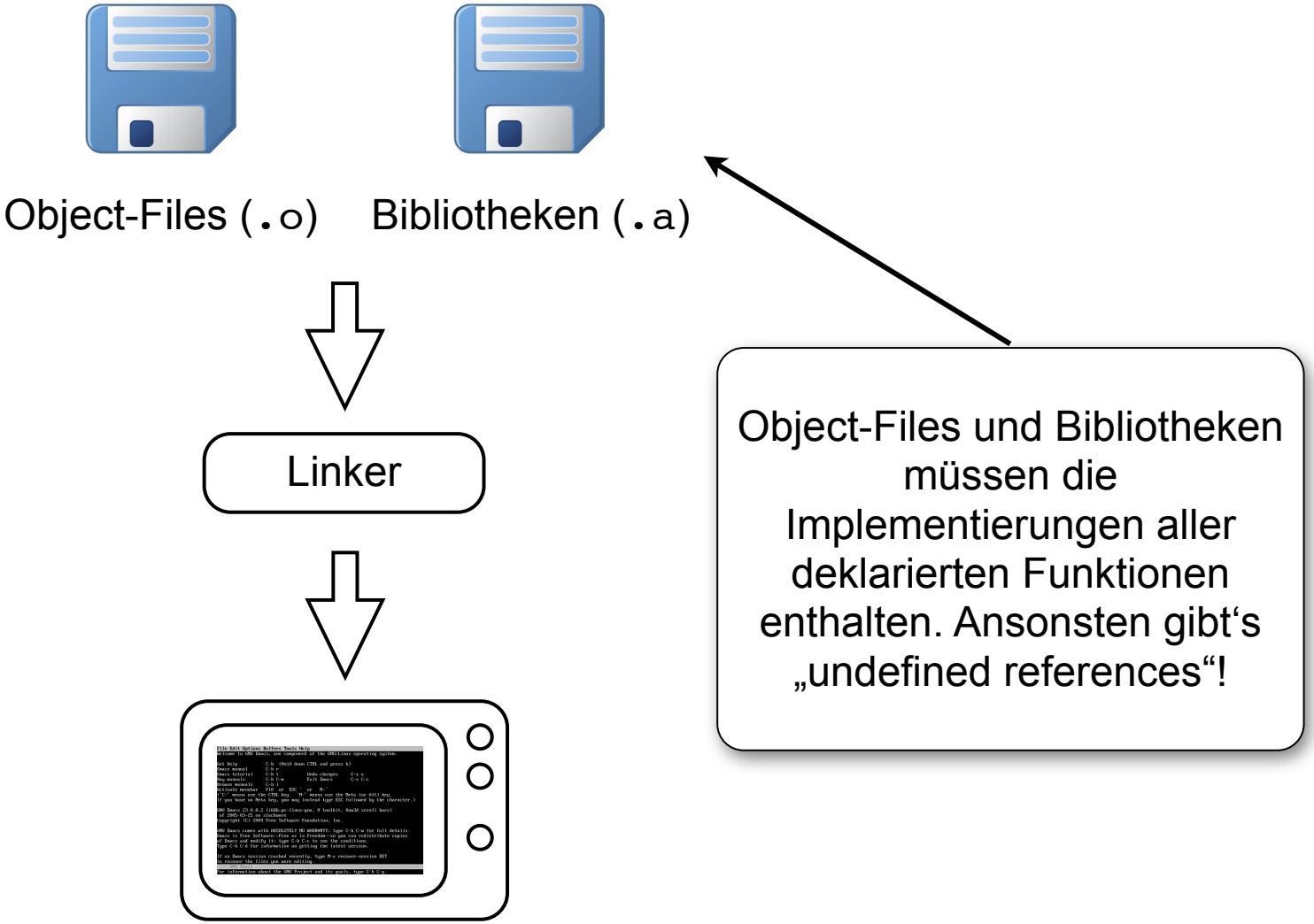
```
% gcc hello.c -o hello%
hello
hello, world!
%
```

- Juchu!!
- Der Compiler übersetzt das Programm in ein Object-File
- Der Linker führt Object-Files und benötigte Bibliotheken zu einem ausführbaren Programm zusammen
- Hier macht `gcc` beides in einem Schritt

Erstellen eines C-Programms - Kompilieren



Erstellen ein C-Programms - Linken



Der C Präprozessor

- Was macht folgende Zeile?

```
#include <stdio.h>
```

- Vor dem Übersetzen des Programms wird der Präprozessor aktiv und behandelt alle Zeilen, die mit ‚#' beginnen
- Hier: Einfügen von Funktionsdefinitionen aus einer Bibliothek
  - Nicht die Implementierung von Funktionen wird eingefügt
  - Erlaubt, z.B. die Funktion `printf()` zu benutzen
- Die Implementierungen werden vom Linker hinzugefügt
- `#include` importiert den Inhalt der gesamten Datei `<stdio.h>`
- Wie verhindert man mehrfache Deklarationen?

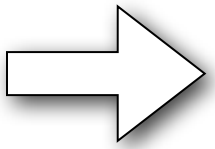
Der C-Präprozessor - `#define` (1)

- Mit Hilfe von `#define` lassen sich Textersetzungen definieren
- Meistens werden symbolische Konstanten definiert

```
#define MAX_LENGTH 100
```

- Der Präprozesor substituiert den String „`100`“ für alle auftretenden `MAX_LENGTH`
- Nur reine Textersetzung, keine Typprüfung!
- Beispiel:

```
#define MAX_LENGTH 100
/* later */
int i;
/* later */
if(i > MAX_LENGTH) {
  printf("Whoa there!\n");
}
```



```
                .
                .
                .
/* That code becomes */
if(i > 100) {
  printf("Whoa there!\n");
}
```

Der C-Präprozessor - `#define` (2)

- **Alle** auftretenden `MAX_LENGTH` werden mit 100 ersetzt
- Warum nicht direkt `100` schreiben?
  - Änderungen müssen nur an einer Stelle des Programms gemacht werden
  - Hard-Codierte Werte werden als *„magic numbers“* bezeichnet
    - Wiederholen sich häufig im Programm
    - Müssen an vielen Zeilen des Codes geändert werden
- Makros

```
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
```

- Wird überall ersetzt
- `#define` definert die Bedutung von Namen
- `#define ALL_CAPITAL_LETTERS` wird mit Großbuchstaben geschrieben (Konvention), d.h. nicht für Variablennamen

C-Präprocessor: #ifdef / #ifndef

- Benutzung von #ifdef ... #else ... #endif

```
#ifdef WINDOWS
#include <windows.h>
#else
#include <X11/X.h>
#endif
```

- `#ifndef` fügt Code ein, falls das Symbol nicht definiert ist
- Ermöglicht einfach den Code an spezielle Umgebungen anzupassen
- Kann benutzt werden, um Passagen an- oder auszuschalten

C - Präprozessor: Include-Guards

- Mehrfacheinbindung von Header-Files kann zu Problemen führen (Mehrfachdefinitionen)
- Schwierig zu unterdrücken, da Header-Files Header-Files einbinden können
- Folgender Mechanismus schafft Abhilfe:

```
/* Header-File "foo.h": */
#ifndef FOO_H
#define FOO_H

/* contents of file */

#endif /* FOO_H */
```

- Der Inhalt von `foo.h` wird nur einmal eingebunden!

Zurück zum Hello World!

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

int main(void)
{
    printf("hello, world!\n");
    return 0;
}
```

- Das ist der Code, den der Compiler zu sehen bekommt!

Ausgabe mit `printf()`

- Ausgabe auf den Bildschirm

```
int a = 5;
double pi = 3.14159;
char s[] = "I am a string!";
printf("a = %d, pi = %f s = %s\n", a, pi, s);
```

- Substituiert die Werte für
- `\n` markiert einen
- Beispiele für Forma

Noch viel mehr Formatierungen werden in den `man`-pages erklärt!

	nteger (dezimal)
	unsigned integer
`%f`	floating point
`%x, %X`	hexadezimale Zahl
`%c`	char
`%s`	string, d.h. char array
`%p`	pointer
`%%`	Prozentzeichen

Gliederung

C - Funktionen / Datentypen

- C-Programme bestehen aus Funktionen
- Funktionen
  - Nehmen Argumente als Eingabe
  - Berechnen etwas
  - Geben ein Ergebnis zurück
- Einstiegspunkt für ein C-Programm: `main()`-Funktion
- Jegliche Daten in C haben einen Typ!
  - Beispiel
    - `int`
    - `char`
    - `float`
    - `double`
  - Variablen speichern die Daten
  - Variablen müssen vor der Verwendung deklariert werden!

C-Datentypen (1)

- Typdeklarationen

```
int i;          /* name = i    type = int   /*
char c;         /* name = c    type = char /*
float some_float = 3.14;
```

- Identifier: `i, c, d, some_float`
- Optionale Initialisierung
- Booleans sind `0`, oder nicht Null (meistens `1`)
- Strings: Felder vom Typ `char`

```
char some_string[9] = "woo hoo!";
char some_string[] = "woo hoo!";
char* same_again = "woo hoo!";
```

- Mehr zu Feldern und Strings später!
- Andere Typen: structs, pointer

C - Datentypen (2)

- `int`
  - gewöhnlich 32 Bit lang
  - hängt aber vom Computer ab (64-Bit Maschinen, 16-Bit Mikroprozessoren)
- `char`
  - 0 .. 256
- `float`
  - Approximation einer reellen Zahl mit einfacher Genauigkeit
- analog: `double` mit doppelter Genauigkeit
- Speicherbelegung der Datentypen ist im C-Standard nicht definiert und hängt von der Implementierung des Compilers und dem Zielsystem ab
- Die Größen kann man in `<limits.h>` und `<float.h>` finden

C-Datentypen (3)

- Zusätzlich gibt es Modifiers:
  - `unsigned`
  - `signed`
  - `short`
  - `long`
- Modifier modfizieren die Größe und den Wertebereich eines Datentyps
- `signed` / `unsigned` gehen mit `int` und `char`
- `short` geht mit `int`
- `long` kann neben `int` auch mit `double` verwendet werden
- es gibt `long long double`
- Es müssen folgende Ungleichungen gelten:
  - `char` <= `short int` <= `int` <= `long int` <= `long long int`
  - `float` <= `double` <= `long double` <= `long long double`

C-Datentypen (4) - ANSI

Datentyp	Verwendung	Wertebereich	Größe
`char`	Zeichen oder kleine natürliche Zahlen	-127 ... 127	1 Byte
`unsigned char`	kleine natürliche Zahl ohne Vorzeichen	0 ... 256	1 Byte
`int`	natürliche Zahl	-2^{31} ... $2^{31} - 1$	4 Byte
`unsigned int`	natürliche Zahl ohne Vorzeichen	0 ... 2^{32}	4 Byte
`float`	Dezimalzahl	$1.5e^{-45}$ ... $3.4e^{38}$	4 Byte
`double`	Dezimalzahl	$5.0e^{-324}$ ... $1.7e^{308}$	8 Byte
`long double`	Dezimalzahl	$1.9e^{-4951}$ .. $1.1e^{4932}$	10 Byte

Daten für normale 32- und 64-Bit Systeme. Genauigkeiten: `float` 7-8 Stellen, `double` 15-16 Stellen, `long double` 19 - 20 Stellen

C-Felder / Arrays (1)

- Möglichkeit, Daten des selben Typs in einem Objekt zusammen zu fassen
- Lineare Sequenz von Daten
  - Array von ints
  - Array von doubles
  - Array von chars (String)
- Eindimensionales Array von 3 Integers:

```
int arr[3];
int sum;
arr[0] = 1;
arr[1] = 22;
arr[2] = -35;
sum = arr[0] + arr[1] + arr[2];
```

- Achtung: Nicht initialisierte Arrays enthalten Datenmüll!

C-Felder / Arrays (2)

- Beispiele:

```
int my_array[10];                          /* not initialized */
int my_array[5]  = { 1, 2, 3, 4, 5 };      /* initialized     */
int my_array[]   = { 1, 2, 3, 4, 5 };      /* OK, initialized */
int my_array[4]  = { 1, 2, 3, 4, 5 };      /* warning         */
int my_array[10] = { 1, 2, 3, 4, 5 };      /* OK, partially
                                              initialized   */
```

- Bemerkung zur teilweisen Initialisierung
  - Restliche Werte sind mit 0 initialisiert
- Explizite Initialisierung

```
int i;
int my_array[10];
for (i = 0; i < 10; i++) {
    my_array[i] = 2 * i;
}
```

- In der Regel die beste Möglichkeit

C-Felder / Arrays (3)

- Achtung: C ist eine unsichere Programmiersprache

```
int my_array[10];
/* What happens here? */
printf("%d\n", my_array[0]);
/* What happens here? */
printf("%d\n", my_array[1000]);
```

- Kann zu einem “segmentation fault“ führen: Das Programm stürzt ab
- Es kann auch passieren, dass auf irgendeine nicht initialisierte Speicherzelle zugegriffen wird, dann wird “Müll“ ausgegeben
- Solche Zugriffe sind die Ursache für viele Sicherheitslücken in Betriebssystemen und deren Software

C-Felder / Arrays (4)

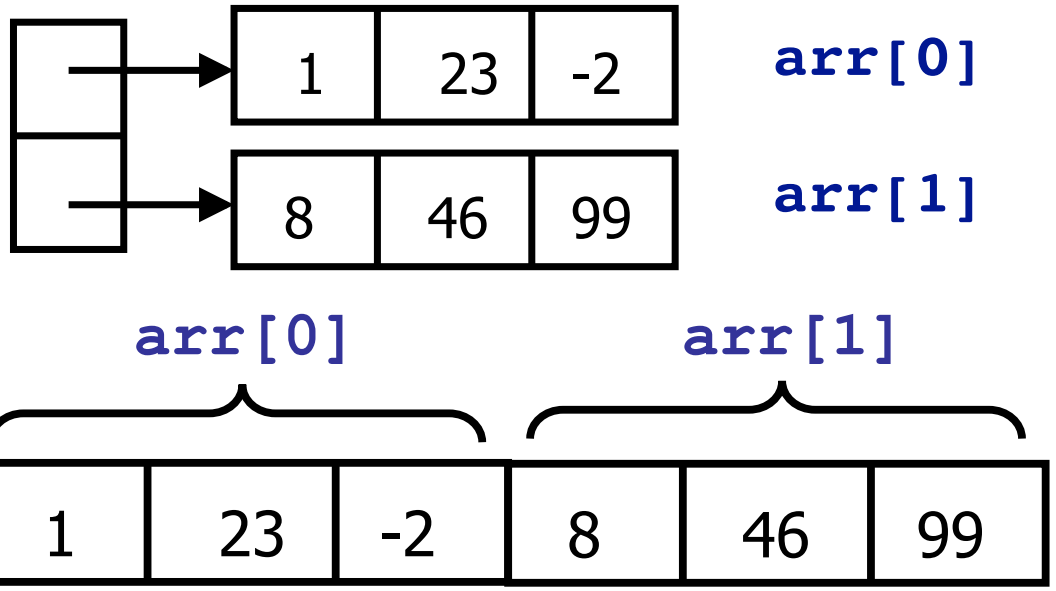
- zweidimensionale Arrays:

```
int arr[2][3];                          /* NOT arr[2, 3] */
int i, j;
int sum = 0;

arr[0][0] = 1;
arr[0][1] = 23;
arr[0][2] = -12;
arr[1][0] = 85;
arr[1][1] = 46;
arr[1][2] = 99;

for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
        sum += arr[i][j];
    }
}

printf("sum = %d\n", sum);
```

C-Felder / Arrays (5)

- Zweidimensionale Felder können komponentenweise in eindimensionale Felder aufgespalten werden

```
int arr[2][3];
/* initialize... */
/* arr[0] is array of 3 ints */
/* arr[1] is another array of 3 ints */
```



C-Felder / Arrays (6)

- Initialisierung zweidimensionaler Arrays:

```
int arr[2][3];                          /* not initialized */
int arr[2][3]
  = { { 1, 2, 3 }, { 4, 5, 6 } };       /* OK */
int arr[2][3]
  = { 1, 2, 3, 4, 5, 6 };               /* warning with -Wall */
int arr[2][]
  = { { 1, 2, 3 }, { 4, 5, 6 } };       /* invalid */
int arr[][]
  = { { 1, 2, 3 }, { 4, 5, 6 } };       /* invalid */
int arr[][3]
  = { { 1, 2, 3 }, { 4, 5, 6 } };       /* OK */
int arr[][3]
  = { 1, 2, 3, 4, 5, 6 };               /* warning with -Wall */
int arr[][3]
  = { { 1, 2, 3 }, { 4, 5 } };          /* OK; missing value = 0 */
```

- Regel: Alle Dimensionen, außer der, die am weitesten links steht, müssen spezifiziert werden, obwohl der Compiler selbst zählen kann.

C - Operatoren (1)

- Numerisch: `+ - * / %`
- Zuweisungen: =

```
int i = 10;
int j = 20;
i = 2 + i * j;
j = j % 2;
```

- Wie wird `i = 2 + i * j` ausgewertet?

a) `i = (2 + i) * j;`
b) `i = 2 + (i * j);`

- `*` bindet stärker als +
- Benutze `()` um eine andere Interpretation zu erzwingen
- Andere Zuweisungsoperatoren: `+=, -=, +=` ...

```
i += 2;
```

C - Operatoren (2)

- Inkrement und Dekrement: ++, --

```
i++;
++i;
```

- Vergleichsoperatoren
  - < <= => >
  - == (für Gleichheit)
  - != (für Ungleichheit)
- Logische Operatoren
  - Argumente sind Integers, die als Booleans benutzt werden
  - ! ist das („unäre nicht“)
  - && ist das logische ^ (und)
  - || ist das logische v (oder)

C - Operatoren (3)

- Beispiele:

```
int bool1, bool2, bool3, bool4;
bool1 = 0;                          /* false */
bool2 = !bool1;                     /* bool2 -> true */
bool3 = bool1 || bool2;             /* value? */
bool4 = bool1 && bool2;             /* value? */
```

- Der unäre Minus-Operator

```
int var1 = 10;
int var2;
var2 = -var1;
```

Vorsicht bei ++ und --

- ++ und -- können als Präfix und Postfix verwendet werden
- Sie haben aber unterschiedliche Bedeutung!

```
int a = 0;
a++;  /* OK */
++a;  /* OK */
```

- Oben machen beide Aufrufe das Gleiche, aber:

```
int a, b, c;
a = 10;
b = ++a;  /* What is b? */
          /* 11 */
a = 10;
c = a++;  /* What is c? */
          /* 10 */
```