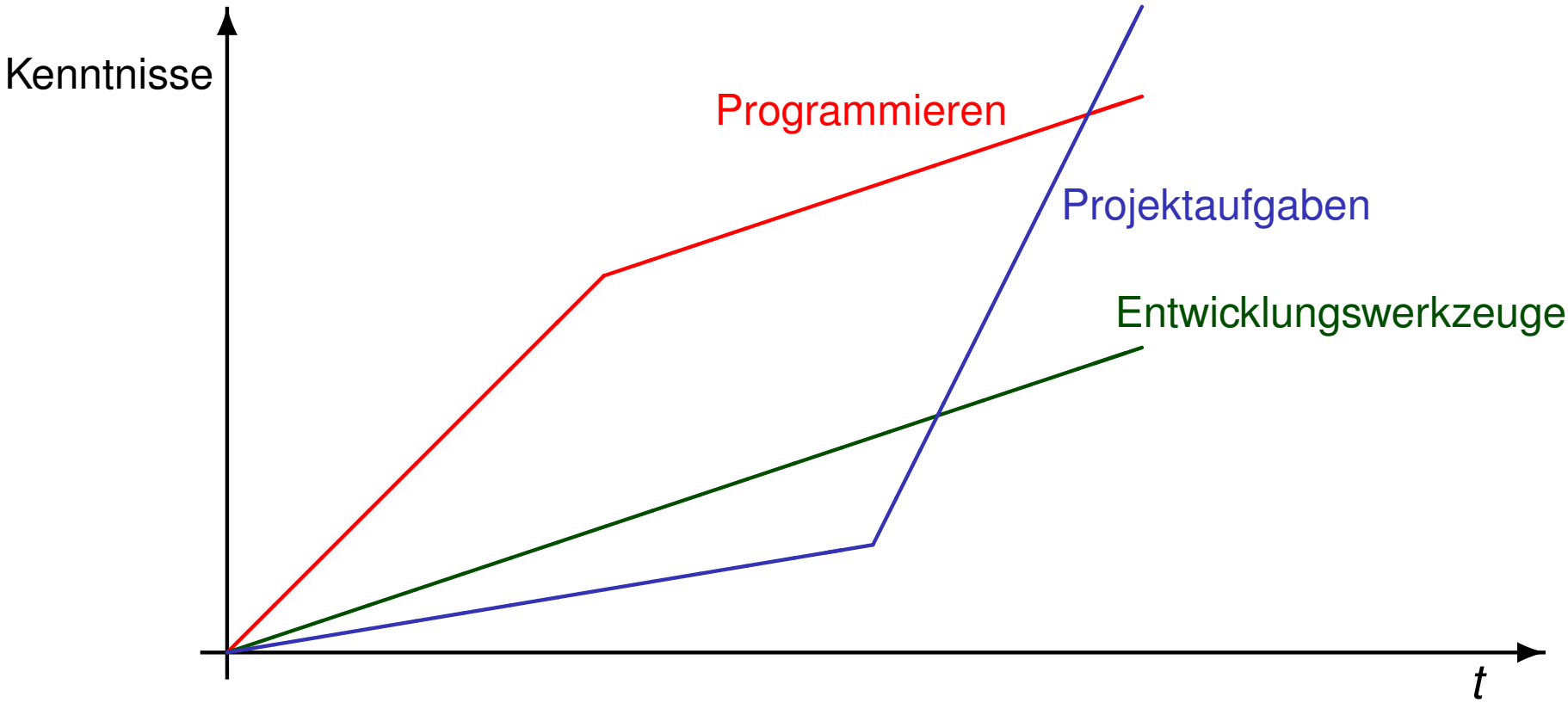
# Software-Entwicklung für eingebettete Systeme

- Hier wird programmiert.
- Richtig programmiert.

# Software-Entwicklung für eingebettete Systeme

- Hier wird programmiert.
- Richtig programmiert.

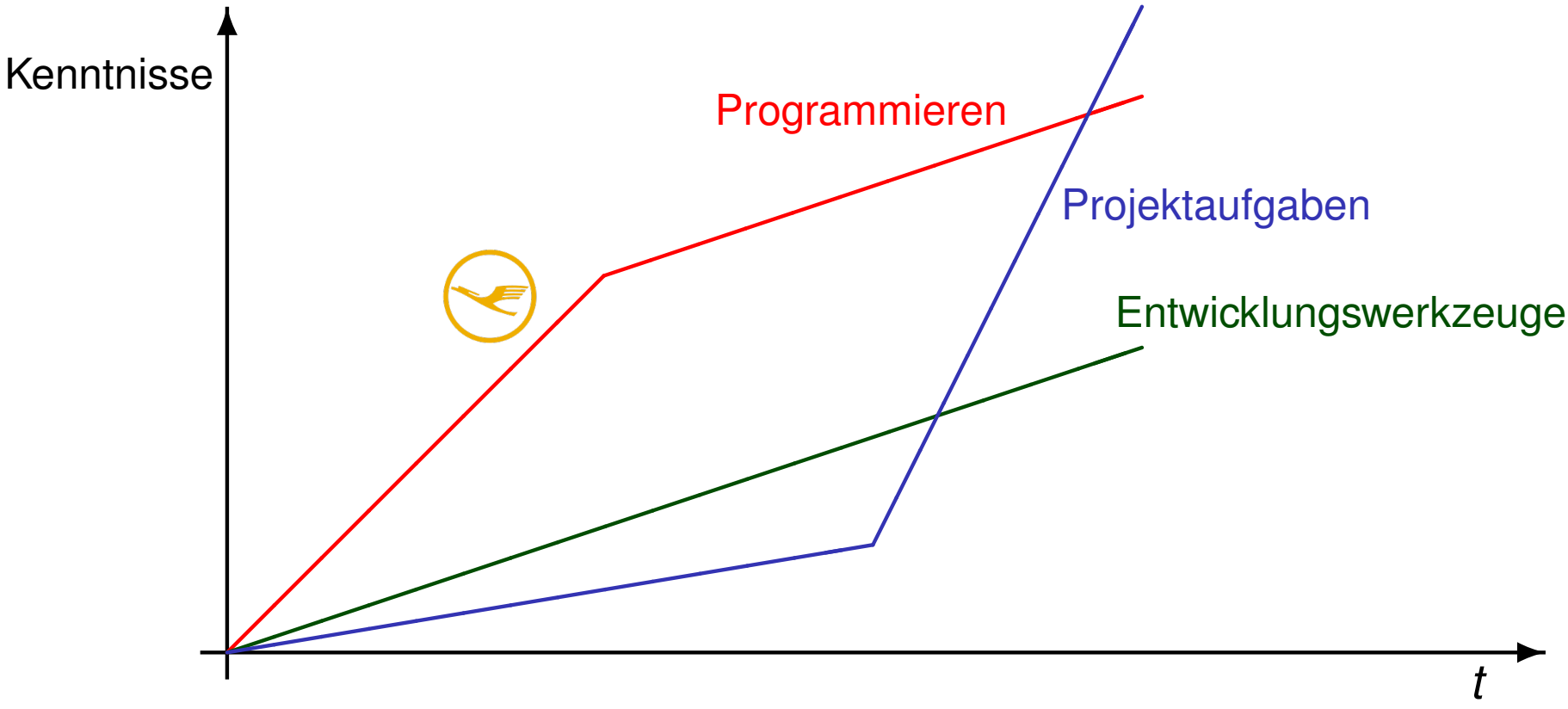
## Fahrplan



# Software-Entwicklung für eingebettete Systeme

- Hier wird programmiert.
- Richtig programmiert.

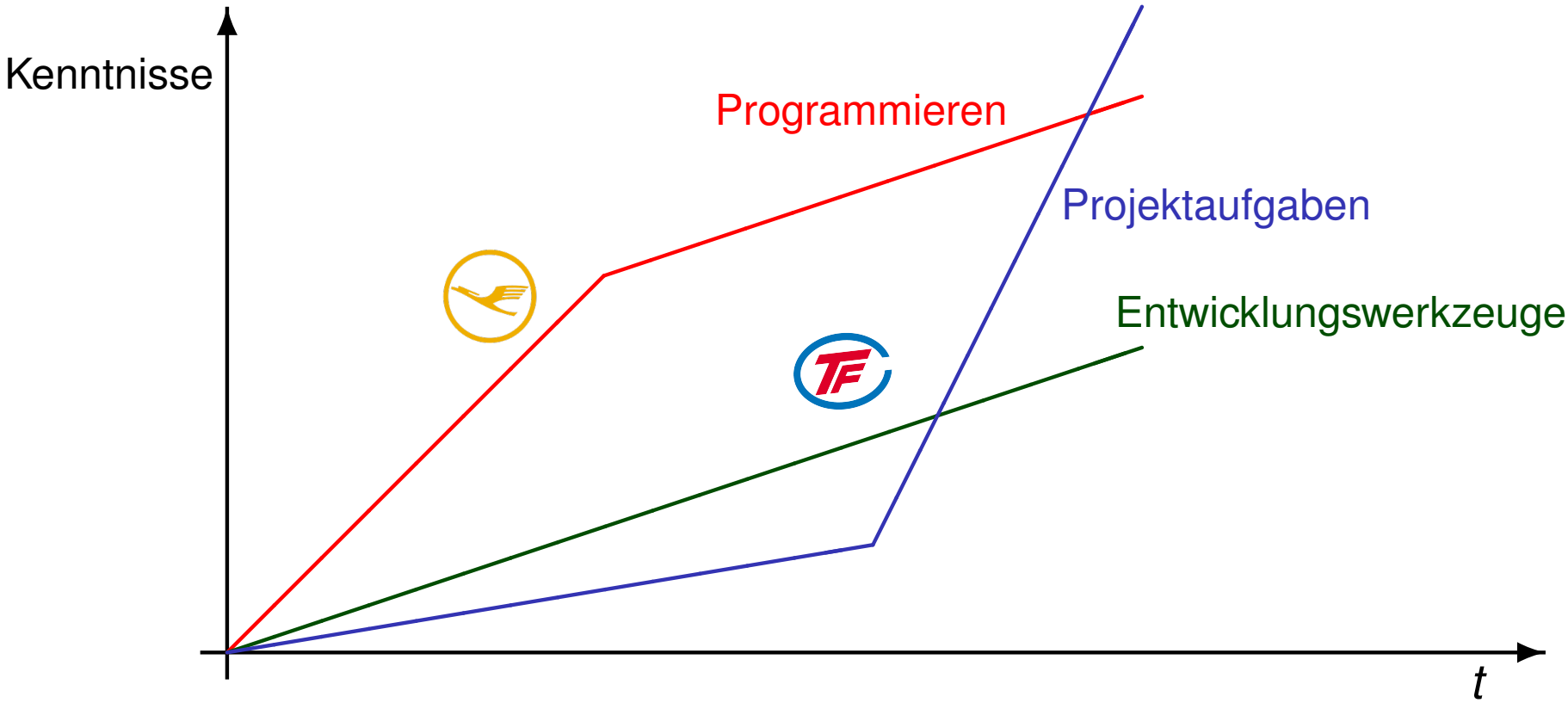
## Fahrplan



# Software-Entwicklung für eingebettete Systeme

- Hier wird programmiert.
- Richtig programmiert.

## Fahrplan



# 1 Programmieren in C

## 1.1 Ausdrücke und Anweisungen

```
#include <stdio.h>

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

# 1 Programmieren in C

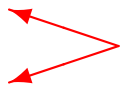
## 1.1 Ausdrücke und Anweisungen

```
#include <stdio.h>

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

# 1 Programmieren in C

## 1.1 Ausdrücke und Anweisungen

```
#include <stdio.h>

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




Ausdrücke:

# 1 Programmieren in C

## 1.1 Ausdrücke und Anweisungen

```
#include <stdio.h>

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

Anweisungen

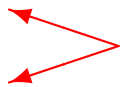
Ausdrücke:

| | |
|---|---|
| "Hello,␣world!\n" | String-Konstante |

# 1 Programmieren in C

## 1.1 Ausdrücke und Anweisungen

```
#include <stdio.h>

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



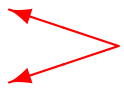

Ausdrücke:

| | |
|---|---|
| `"Hello,␣world!\n"` | String-Konstante |
| `0` | Integer-Konstante |

# 1 Programmieren in C

## 1.1 Ausdrücke und Anweisungen

```
#include <stdio.h>

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




Ausdrücke:

| | |
|---|---|
| `"Hello,␣world!\n"` | String-Konstante |
| `0` | Integer-Konstante |
| `printf ("Hello,␣world!\n")` | Funktionsaufruf, Integer |

# 1 Programmieren in C

## 1.1 Ausdrücke und Anweisungen

Ausdrücke: Wert, Typ

- Konstante
- Funktionsaufrufe

Anweisung: mit Semikolon abschließen

- Ausdruck (Wert wird ignoriert)
- leere Anweisung
- **return**-Anweisung

Anweisungsblock: geschweifte Klammern

# 1 Programmieren in C

## 1.2 Seiteneffekte

- Konstante: kein Seiteneffekt
- Funktionsaufruf: kann Seiteneffekt haben

Beispiel: Funktionsaufruf printf ("Hello,␣world!\n")

- „Haupteffekt“:
  zurückgegebener Wert (Anzahl ausgegebener Zeichen)
- Seiteneffekt: Text ausgeben

Ausdruck als Anweisung:
Wert wird ignoriert, Seiteneffekt wichtig

# 1 Programmieren in C

## 1.3 Funktionen aufrufen

Anzahl der übergebenen Parameter
wird nicht immer überprüft

Bei Fehlern: „zufällige“ Werte

Mit Option `-Wall`: Warnung

# 1 Programmieren in C

## 1.4 Operatoren

Unäre Operatoren:
foo() −foo foo−− foo++ −−foo ++foo

Binäre Operatoren:
+ − ∗ / % & | ^ << >>

Ternäre Operatoren:
(später)

# 1 Programmieren in C

## 1.4 Operatoren

Unäre Operatoren:

foo() −foo foo−− foo++ −−foo ++foo


Binäre Operatoren:

+ − ∗ / % & | ^ << >>

Ternäre Operatoren:

(später)

# 1 Programmieren in C

## 1.5 Variable

```
#include <stdio.h>

int a, b = 3;

int main (void)
{
   printf ("a = %d, b = %d\n", a, b);
   static int a = 7;
   int b = 5;
   printf ("a = %d, b = %d\n", a, b);
   a = b = 12;
   printf ("a = %d, b = %d\n", a, b);
   return 0;
}
```

# 1 Programmieren in C

## 1.5 Variable

```
#include <stdio.h>

int a, b = 3;      ←—— b = 3 (explizit), a = 0 (implizit)

int main (void)
{
   printf ("a = %d, b = %d\n", a, b);
   static int a = 7;
   int b = 5;      ←—— Initialisierung bei jedem Aufruf
   printf ("a = %d, b = %d\n", a, b);
   a = b = 12;     ←—— setze beide auf 12
   printf ("a = %d, b = %d\n", a, b);
   return 0;
}
```

# 1 Programmieren in C

## 1.5 Variable

```
#include <stdio.h>

int a, b = 3;      ←—— b = 3 (explizit), a = 0 (implizit)

int main (void)
{
   printf ("a␣=␣%d,␣b␣=␣%d\n", a, b);
   static int a = 7;
   int b = 5;      ←—— Initialisierung bei jedem Aufruf
   printf ("a␣=␣%d,␣b␣=␣%d\n", a, b);
   a = b = 12;     ←—— setze beide auf 12
   printf ("a␣=␣%d,␣b␣=␣%d\n", a, b);
   return 0;
}
```

Sowohl die globalen **int** a, b = 3 als auch **static int** a = 7 behalten ihre Werte zwischen zwei Aufrufen der Funktion.

# 1 Programmieren in C

## 1.5 Variable

Zuweisungsoperatoren in C

= += −= ∗= /= %= &= |= ^= <<= >>=

# 1 Programmieren in C

## 1.5 Variable

Zuweisungsoperatoren in C
= += −= ∗= /= %= &= |= ^= <<= >>=
sind ganz normale binäre Operatoren

# 1 Programmieren in C

## 1.5 Variable

Zuweisungsoperatoren in C
= += −= ∗= /= %= &= |= ^= <<= >>=
sind ganz normale binäre Operatoren
mit Seiteneffekt.

# 1 Programmieren in C

## 1.5 Variable

Zuweisungsoperatoren in C
= += −= ∗= /= %= &= |= ^= <<= >>=
sind ganz normale binäre Operatoren
mit Seiteneffekt.

| | |
|---|---|
| a + 7 | ergibt a + 7 (und hat keinen Seiteneffekt). |
| a = 7 | ergibt 7 (und weist a den Wert 7 zu). |
| a += 7 | ergibt a + 7 (und weist a diesen Wert zu). |

# 1 Programmieren in C

## 1.6 Funktionen schreiben

```c
#include <stdio.h>

int answer (void)
{
   return 42;
}

void foo (void)
{
   printf ("%d\n", answer ());
}

int main (void)
{
   foo ();
   return 0;
}
```

# 1 Programmieren in C

## 1.6 Funktionen schreiben

```c
#include <stdio.h>

int answer (void)
{
  return 42;
}

void foo (void)
{
  printf ("%d\n", answer ());
}

int main (void)
{
  foo ();
  return 0;
}
```

```c
#include <stdio.h>

answer ()
{
  return 42;
}

void foo ()
{
  printf ("%d\n", answer ());
}

main ()
{
  foo ();
  return 0;
}
```

# 1 Programmieren in C

## 1.6 Funktionen schreiben

```c
#include <stdio.h>

int answer (void)
{
  return 42;
}

void foo (void)
{
  printf ("%d\n", answer ());
}

int main (void)
{
  foo ();
  return 0;
}
```

```c
#include <stdio.h>

answer ()
{
  return 42;
}

void foo ()
{
  printf ("%d\n", answer ());
}

main ()
{
  foo ();
  return 0;
}
```

K&R-C
(ohne Prototypen)

# 1 Programmieren in C

## 1.6 Funktionen schreiben

ISO-C
(mit Prototypen)

```c
#include <stdio.h>

int answer (void)
{
  return 42;
}

void foo (void)
{
  printf ("%d\n", answer ());
}

int main (void)
{
  foo ();
  return 0;
}
```

K&R-C
(ohne Prototypen)

```c
#include <stdio.h>

answer ()
{
  return 42;
}

void foo ()
{
  printf ("%d\n", answer ());
}

main ()
{
  foo ();
  return 0;
}
```

# 1 Programmieren in C

## 1.6 Funktionen schreiben

```c
#include <stdio.h>

int answer (void)
{
  return 42;
}

void foo (void)
{
  printf ("%d\n", answer ());
}

int main (void)
{
  foo ();
  return 0;
}
```

ISO-C
(mit Prototypen)
**(Besser.)**

```c
#include <stdio.h>

answer ()
{
  return 42;
}

void foo ()
{
  printf ("%d\n", answer ());
}

main ()
{
  foo ();
  return 0;
}
```

K&R-C
(ohne Prototypen)

# 1 Programmieren in C

## 1.6 Funktionen schreiben

ISO-C

```
#include <stdio.h>

void add_verbose (int a, int b)
{
   printf ("%d␣+␣%d␣=␣%d\n",
               a, b, a + b);
}


int main (void)
{
   add_verbose (2, 3);
   return 0;
}
```

K&R-C

```
#include <stdio.h>

void add_verbose (a, b)
   int a, b;
{
   printf ("%d␣+␣%d␣=␣%d\n",
               a, b, a + b);
}

main ()
{
   add_verbose (2, 3);
   return 0;
}
```

# 1 Programmieren in C

## 1.6 Funktionen schreiben

K&R-C mit ISO-Prototypen

```
#include <stdio.h>

void add_verbose (int, int);
int main (void);

void add_verbose (a, b)
   int a, b;
{
   printf ("%d␣+␣%d␣=␣%d\n", a, b, a + b);
}

int main ()
{
   add_verbose (2, 3);
   return 0;
}
```

# 1 Programmieren in C

## 1.6 Funktionen schreiben

```
void add_verbose (int a, int b)
{
   printf ("%d␣+␣%d␣=␣%d\n", a, b, a + b);
}
```

Der Datentyp **void**

# 1 Programmieren in C

## 1.6 Funktionen schreiben

```
void add_verbose (int a, int b)
{
   printf ("%d␣+␣%d␣=␣%d\n", a, b, a + b);
}
```

Der Datentyp **void**

- steht für „nichts“.

# 1 Programmieren in C

## 1.6 Funktionen schreiben

```
void add_verbose (int a, int b)
{
  printf ("%d␣+␣%d␣=␣%d\n", a, b, a + b);
}
```

Der Datentyp **void**

- steht für „nichts“.
- Rückgabewert kann ignoriert werden.

# 1 Programmieren in C

## 1.6 Funktionen schreiben

```
void add_verbose (int a, int b)
{
  printf ("%d␣+␣%d␣=␣%d\n", a, b, a + b);
}
```

Der Datentyp **void**

- steht für „nichts“.
- Rückgabewert muß ignoriert werden.

# 1 Programmieren in C

## 1.7 Zeiger

```
#include <stdio.h>

void calc_answer (int *a)
{
   *a = 42;
}

int main (void)
{
   int answer;
   calc_answer (&answer);
   printf ("The answer is %d.\n", answer);
   return 0;
}
```

# 1 Programmieren in C

## 1.7 Zeiger

```
#include <stdio.h>

void calc_answer (int *a)
{
   *a = 42;
}

int main (void)
{
   int answer;
   calc_answer (&answer);
   printf ("The answer is %d.\n", answer);
   return 0;
}
```

- *a ist eine **int**.

# 1 Programmieren in C

## 1.7 Zeiger

```
#include <stdio.h>

void calc_answer (int *a)
{
  *a = 42;
}

int main (void)
{
  int answer;
  calc_answer (&answer);
  printf ("The answer is %d.\n", answer);
  return 0;
}
```

- *a ist eine **int**.
- unärer Operator *:
  Pointer-Dererefenzierung

# 1 Programmieren in C

## 1.7 Zeiger

```
#include <stdio.h>

void calc_answer (int *a)
{
   *a = 42;
}

int main (void)
{
   int answer;
   calc_answer (&answer);
   printf ("The answer is %d.\n", answer);
   return 0;
}
```

- *a ist eine **int**.
- unärer Operator *:
  Pointer-Dererefenzierung

⟶ a ist ein Zeiger (Pointer) auf eine **int**.

# 1 Programmieren in C

## 1.7 Zeiger

```
#include <stdio.h>

void calc_answer (int *a)
{
   *a = 42;
}

int main (void)
{
   int answer;
   calc_answer (&answer);
   printf ("The answer is %d.\n", answer);
   return 0;
}
```

- *a ist eine **int**.
- unärer Operator *:
  Pointer-Dererefenzierung

⟶ a ist ein Zeiger (Pointer) auf eine **int**.

- unärer Operator &: Adresse

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable.

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>

int main (void)
{
   int prime[5] = { 2, 3, 5, 7, 11 };
   int *p = prime, i = 0;
   while (i < 5)
      printf ("%d\n", *(p + i++));
   return 0;
}
```

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>

int main (void)
{
   int prime[5] = { 2, 3, 5, 7, 11 };
   int *p = prime, i = 0;
   while (i < 5)
      printf ("%d\n", *(p + i++));
   return 0;
}
```

- prime ist eine Ansammlung von fünf ganzen Zahlen.

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>

int main (void)
{
   int prime[5] = { 2, 3, 5, 7, 11 };
   int *p = prime, i = 0;
   while (i < 5)
      printf ("%d\n", *(p + i++));
   return 0;
}
```

- prime ist ein Array von fünf ganzen Zahlen.

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>

int main (void)
{
   int prime[5] = { 2, 3, 5, 7, 11 };
   int *p = prime, i = 0;
   while (i < 5)
      printf ("%d\n", *(p + i++));
   return 0;
}
```



- prime ist ein Array von fünf ganzen Zahlen.
- prime ist ein Zeiger auf eine **int**.

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>

int main (void)
{
   int prime[5] = { 2, 3, 5, 7, 11 };
   int *p = prime, i = 0;
   while (i < 5)
      printf ("%d\n", *(p + i++));
   return 0;
}
```



- prime ist ein Array von fünf ganzen Zahlen.
- prime ist ein Zeiger auf eine **int**.
- p + i ist ein Zeiger auf den i-ten Nachbarn von *p.

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>

int main (void)
{
   int prime[5] = { 2, 3, 5, 7, 11 };
   int *p = prime, i = 0;
   while (i < 5)
      printf ("%d\n", *(p + i++));
   return 0;
}
```



- prime ist ein Array von fünf ganzen Zahlen.
- prime ist ein Zeiger auf eine **int**.
- p + i ist ein Zeiger auf den i-ten Nachbarn von *p.
- *(p + i) ist der i-te Nachbar von *p.

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>

int main (void)
{
   int prime[5] = { 2, 3, 5, 7, 11 };
   int *p = prime, i = 0;
   while (i < 5)
      printf ("%d\n", *(p + i++));
   return 0;
}
```



- prime ist ein Array von fünf ganzen Zahlen.
- prime ist ein Zeiger auf eine **int**.
- p + i ist ein Zeiger auf den i-ten Nachbarn von *p.
- *(p + i) ist der i-te Nachbar von *p.
- Andere Schreibweise: p[i] statt *(p + i)

# 1 Programmieren in C

## 1.8 Arrays

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

```
#include <stdio.h>

int main (void)
{
   int prime[5] = { 2, 3, 5, 7, 11 };
   int i = 0;
   while (i < 5)
      printf ("%d\n", prime[i++]);
   return 0;
}
```



- prime ist ein Array von fünf ganzen Zahlen.
- prime ist ein Zeiger auf eine **int**.
- p + i ist ein Zeiger auf den i-ten Nachbarn von $*$p.
- $*$(p + i) ist der i-te Nachbar von $*$p.
- Andere Schreibweise: p[i] statt $*$(p + i)

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
   char hello_world[] = "Hello,␣world!\n";
   int i = 0;
   while (hello_world[i] != 0)
      printf ("%d", hello_world[i++]);
   return 0;
}
```

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
   char hello_world[] = "Hello,␣world!\n";
   int i = 0;
   while (hello_world[i])
      printf ("%d", hello_world[i++]);
   return 0;
}
```

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
   char hello_world[] = "Hello,␣world!\n";
   char *p = hello_world;
   while (*p)
      printf ("%d", *p++);
   return 0;
}
```

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
   char hello_world[] = "Hello, world!\n";
   char *p = hello_world;
   while (*p)
      printf ("%c", *p++);
   return 0;
}
```

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
   char hello_world[] = "Hello, world!\n";
   char *p = hello_world;
   while (*p)
      printf ("%c", *p++);
   return 0;
}
```

- Ein **char** ist eine kleinere **int**.

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello, world!\n";
  char *p = hello_world;
  while (*p)
    printf ("%c", *p++);
  return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **char**s.

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello, world!\n";
  char *p = hello_world;
  while (*p)
    printf ("%c", *p++);
  return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **char**s, also ein Zeiger auf **char**s.

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello, world!\n";
  char *p = hello_world;
  while (*p)
    printf ("%c", *p++);
  return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **char**s,
  also ein Zeiger auf **char**s
  also ein Zeiger auf (kleinere) Integer.

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello, world!\n";
  char *p = hello_world;
  while (*p)
    printf ("%c", *p++);
  return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **char**s,
  also ein Zeiger auf **char**s
  also ein Zeiger auf (kleinere) Integer.
- Die Formatspezifikation
  entscheidet über die Ausgabe:

  %d dezimal
  %x hexadezimal
  %c Zeichen

# 1 Programmieren in C

## 1.8 Arrays

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello, world!\n";
  char *p = hello_world;
  while (*p)
    printf ("%c", *p++);
  return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **char**s,
  also ein Zeiger auf **char**s
  also ein Zeiger auf (kleinere) Integer.
- Die Formatspezifikation
  entscheidet über die Ausgabe:

  %d dezimal
  %x hexadezimal
  %c Zeichen
  %s String

# Aufgaben

**Aufgabe 1:** Sieb des Eratosthenes

möglichst viele Primzahlen berechnen

**Aufgabe 2:** Kalender

Jahreszahl eingeben,
Kalender mit Feiertagen ausgeben