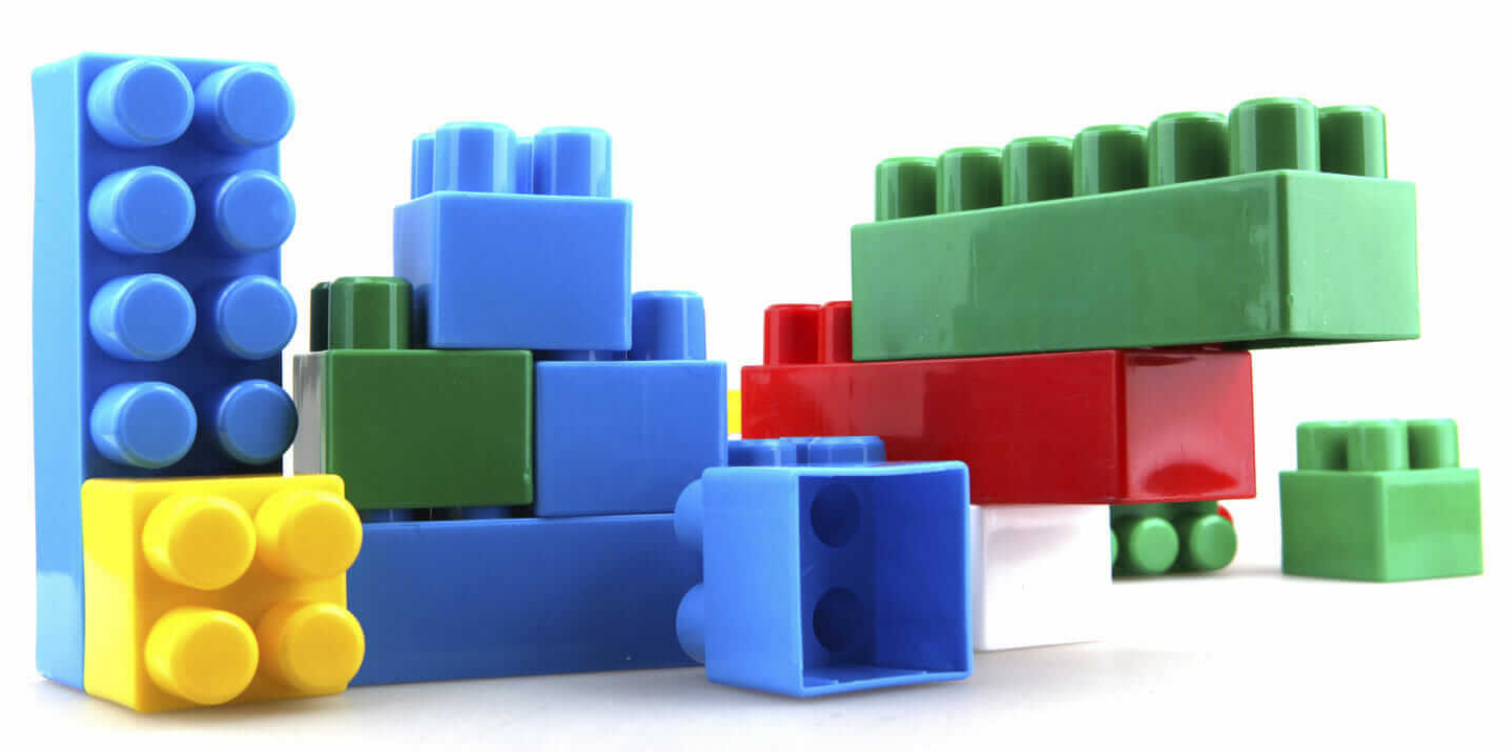
Hardwarenahe Programmierung I

U. Kaiser, R. Kaiser, M. Stöttinger, S. Reith

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

9. Modularisierung



https://redshift.autodesk.com/the-benefits-of-modular-construction/

Modularisierung in der Küche

Hochschule RheinMain

- Die Herstellung von Apfelkuchen ist die eigentliche Aufgabe. Das ist das **Hauptprogramm**.
- Die Herstellung von Hefeteig ist eine Teilaufgabe im Rahmen der Herstellung eines Apfelkuchens. Das ist eine **Funktion** oder ein **Unterprogramm**.
- Das Starten der Aktivität „Hefeteig erstellen“ aus der Zubereitungsvorschrift von Apfelkuchen bezeichnen wir als einen Aufruf des Unterprogramms aus dem Hauptprogramm. Wir sprechen von einem **Unterprogrammaufruf** oder einem **Funktionsaufruf**.
- Zwischen Haupt- und Unterprogramm müssen beim Aufruf ganz bestimmte Informationen fließen, z. B. darüber, wie viel Hefeteig zu erstellen ist und ob dem Teig Zucker zugesetzt werden soll.

Apfelkuchnrezept

600 g	Hefeteig
1,5 kg	Äpfel
...	

Zubereitung

Bereiten Sie den Hefeteig nach Rezept zu und rollen diesen dann auf einer bemehlten Arbeitsfläche quadratisch aus. Geben Sie den Teig dann auf ein mit Backpapier ausgelegtes Blech und ziehen den Rand an jeder Seite hoch. Der Teig kann dann mit einem Küchentuch abgedeckt noch ein wenig stehen bleiben. In der Zwischenzeit die Äpfel schälen, entkernen und in schmale Spalten schneiden. ...

Über den Austausch dieser Informationen muss zwischen Haupt- und Unterprogramm eine präzise Vereinbarung bestehen. Das Hauptprogramm muss wissen, welche Informationen das Unterprogramm benötigt und welche Ergebnisse es produziert. Eine solche Vereinbarung nennen wir eine **Schnittstelle**.

- Eine im Rahmen der Schnittstelle vereinbarte Einzelinformation, wie z.B „Zuckerzugabe in Gramm“, nennen wir einen **Parameter**. Alle Parameter zusammen beschreiben die Schnittstelle. Ein Parameter, durch den Informationen vom Hauptprogramm zum Unterprogramm fließen, bezeichnen wir als **Eingabeparameter**. Einen Parameter, durch den Informationen vom Unterprogramm zum Hauptprogramm zurückfließen, bezeichnen wir als **Rückgabeparameter**.
- Konkrete, durch die Parameter der Schnittstelle fließende Daten (z. B. 100 Gramm Zuckerzugabe) bezeichnen wir als **Parameterwerte**. Entsprechend der Flussrichtung bezeichnen wir die Parameterwerte auch als **Eingabewerte** oder **Rückgabewerte**.

Funktionsschnittstelle

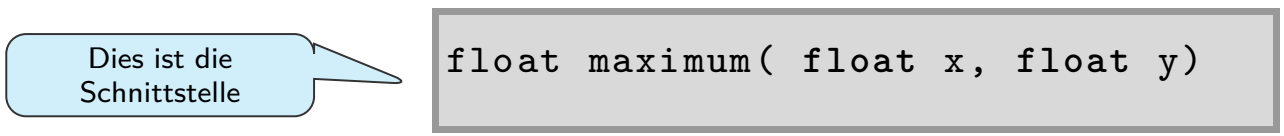
- An der **Schnittstelle** einer Funktion werden alle Informationen festgelegt, die zwischen Haupt- und Unterprogramm fließen.
- Wenn wir eine Funktion erstellen wollen, die das Maximum von zwei Gleitkommazahlen bestimmen soll, so fließen an der Schnittstelle die folgenden Informationen:
  - In die Funktion hinein fließen zwei `float`-Zahlen (Eingabeparameter), von denen die größere zu bestimmen ist.
  - Aus der Funktion heraus fließt eine `float`-Zahl (Rückgabeparameter), nämlich die größere der beiden hinein geflossenen Zahlen.

```
float maximum( float x, float y)
```

- Wie die Funktion ihre Aufgabe erledigt, interessiert an der Schnittstelle nicht. Es geht nur um die Informationen, die sie benötigt , um ihre Aufgabe zu erledigen und die Informationen, die sie nach Erledigung der Aufgabe zurückgibt.

Funktionsschnittstelle

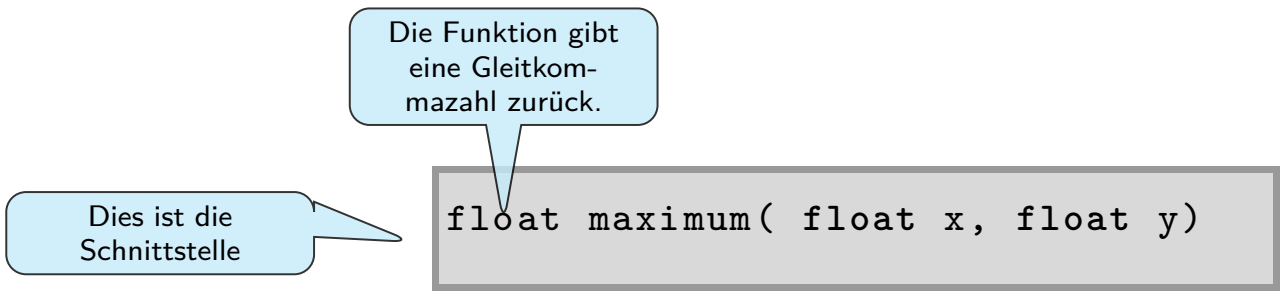
- An der **Schnittstelle** einer Funktion werden alle Informationen festgelegt, die zwischen Haupt- und Unterprogramm fließen.
- Wenn wir eine Funktion erstellen wollen, die das Maximum von zwei Gleitkommazahlen bestimmen soll, so fließen an der Schnittstelle die folgenden Informationen:
  - In die Funktion hinein fließen zwei `float`-Zahlen (Eingabeparameter), von denen die größere zu bestimmen ist.
  - Aus der Funktion heraus fließt eine `float`-Zahl (Rückgabeparameter), nämlich die größere der beiden hinein geflossenen Zahlen.



- Wie die Funktion ihre Aufgabe erledigt, interessiert an der Schnittstelle nicht. Es geht nur um die Informationen, die sie benötigt , um ihre Aufgabe zu erledigen und die Informationen, die sie nach Erledigung der Aufgabe zurückgibt.

Funktionsschnittstelle

- An der **Schnittstelle** einer Funktion werden alle Informationen festgelegt, die zwischen Haupt- und Unterprogramm fließen.
- Wenn wir eine Funktion erstellen wollen, die das Maximum von zwei Gleitkommazahlen bestimmen soll, so fließen an der Schnittstelle die folgenden Informationen:
  - In die Funktion hinein fließen zwei `float`-Zahlen (Eingabeparameter), von denen die größere zu bestimmen ist.
  - Aus der Funktion heraus fließt eine `float`-Zahl (Rückgabeparameter), nämlich die größere der beiden hinein geflossenen Zahlen.

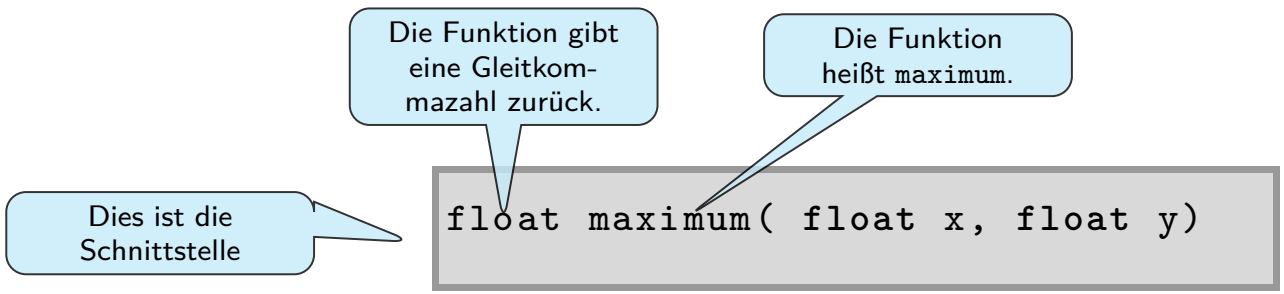


- Wie die Funktion ihre Aufgabe erledigt, interessiert an der Schnittstelle nicht. Es geht nur um die Informationen, die sie benötigt , um ihre Aufgabe zu erledigen und die Informationen, die sie nach Erledigung der Aufgabe zurückgibt.

Funktionsschnittstelle

Hochschule RheinMain

- An der **Schnittstelle** einer Funktion werden alle Informationen festgelegt, die zwischen Haupt- und Unterprogramm fließen.
- Wenn wir eine Funktion erstellen wollen, die das Maximum von zwei Gleitkommazahlen bestimmen soll, so fließen an der Schnittstelle die folgenden Informationen:
  - In die Funktion hinein fließen zwei `float`-Zahlen (Eingabeparameter), von denen die größere zu bestimmen ist.
  - Aus der Funktion heraus fließt eine `float`-Zahl (Rückgabeparameter), nämlich die größere der beiden hinein geflossenen Zahlen.

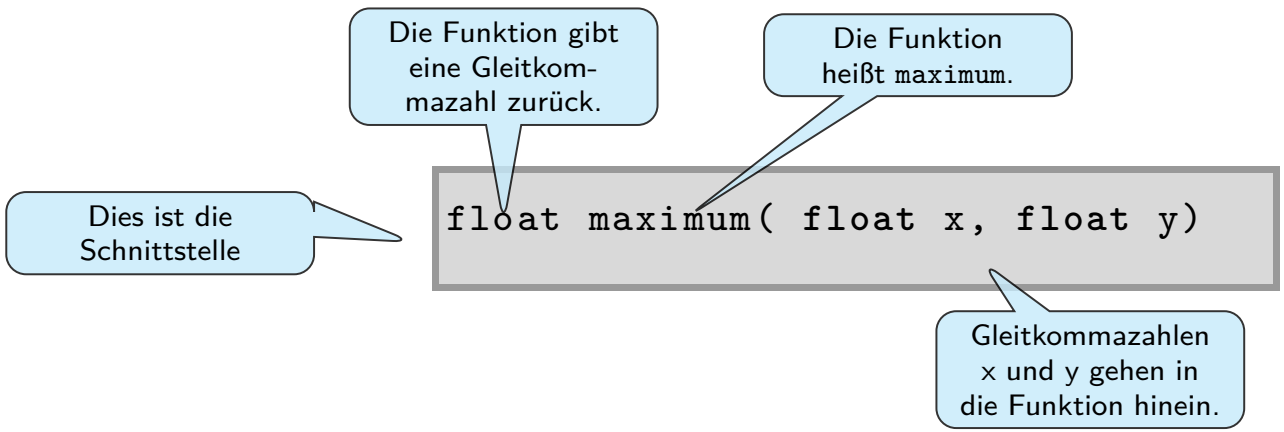


- Wie die Funktion ihre Aufgabe erledigt, interessiert an der Schnittstelle nicht. Es geht nur um die Informationen, die sie benötigt , um ihre Aufgabe zu erledigen und die Informationen, die sie nach Erledigung der Aufgabe zurückgibt.

Funktionsschnittstelle

Hochschule RheinMain

- An der **Schnittstelle** einer Funktion werden alle Informationen festgelegt, die zwischen Haupt- und Unterprogramm fließen.
- Wenn wir eine Funktion erstellen wollen, die das Maximum von zwei Gleitkommazahlen bestimmen soll, so fließen an der Schnittstelle die folgenden Informationen:
  - In die Funktion hinein fließen zwei `float`-Zahlen (Eingabeparameter), von denen die größere zu bestimmen ist.
  - Aus der Funktion heraus fließt eine `float`-Zahl (Rückgabeparameter), nämlich die größere der beiden hinein geflossenen Zahlen.

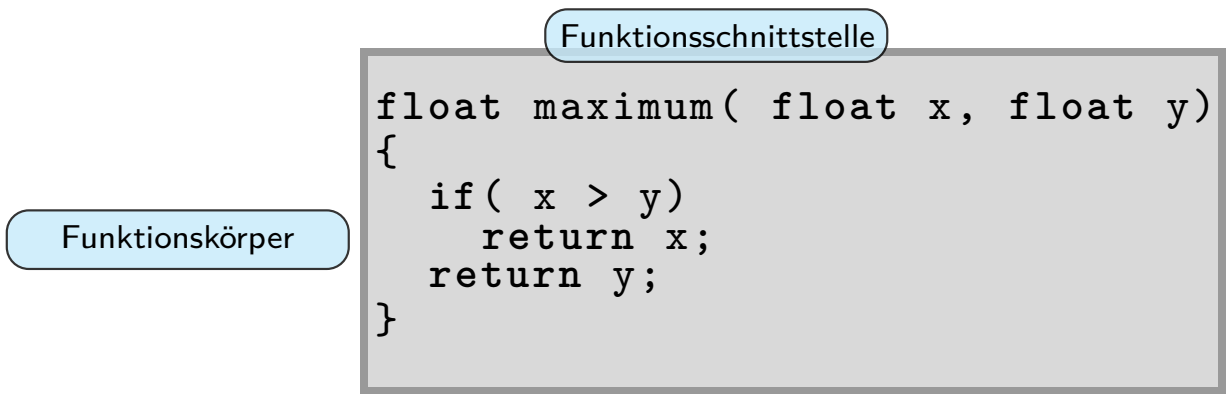


- Wie die Funktion ihre Aufgabe erledigt, interessiert an der Schnittstelle nicht. Es geht nur um die Informationen, die sie benötigt , um ihre Aufgabe zu erledigen und die Informationen, die sie nach Erledigung der Aufgabe zurückgibt.

Funktionskörper

Hochschule RheinMain

- Im **Funktionskörper** wird festgelegt, wie die Funktion ihre Aufgabe erledigt. Der Funktionskörper enthält den Quellcode, der den Ablauf der Funktion festlegt:

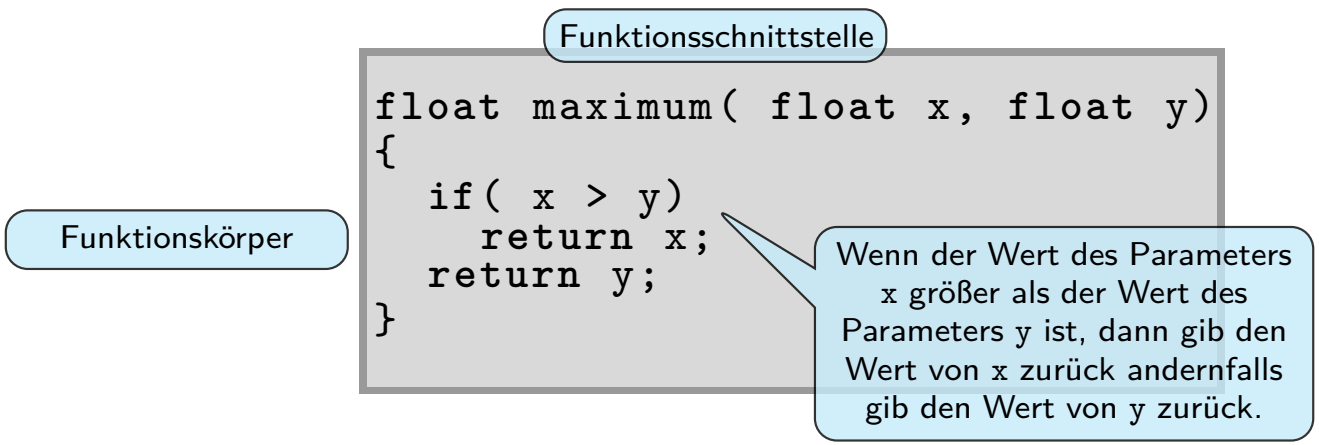


- Eine `return`-Anweisung beendet die Funktion und gibt den Wert des hinter `return` stehenden Ausdrucks an das rufende Programm zurück.
- Der Ergebnistyp des Ausdrucks (hier `float`) muss zum Rückgabetyp der Funktionsschnittstelle passen.

Funktionskörper

Hochschule RheinMain

- Im **Funktionskörper** wird festgelegt, wie die Funktion ihre Aufgabe erledigt. Der Funktionskörper enthält den Quellcode, der den Ablauf der Funktion festlegt:



- Eine `return`-Anweisung beendet die Funktion und gibt den Wert des hinter `return` stehenden Ausdrucks an das rufende Programm zurück.
- Der Ergebnistyp des Ausdrucks (hier `float`) muss zum Rückgabetyp der Funktionsschnittstelle passen.

Funktionsaufruf

- Funktionen können aus dem Hauptprogramm oder aus anderen Funktionen gerufen werden. Die aufrufende Funktion übergibt dazu die erforderlichen Parameter und erhält den von der Funktion ermittelten Rückgabewert:

Funktion

```
float maximum( float x, float y)
{
    if( x > y)
        return x;
    return y;
}
```

Hauptprogramm

```
void main()
{
    float a = 1, b = 2.3, c;

    c = maximum( a, b);
    c = maximum( 12.3, a*b + 1);
    c = b + maximum( 1, 2);
    c = maximum( 1, maximum( a, b) + 1);
}
```

- Anzahl und Typen der beim Aufruf übergebenen Parameter und die Verwendung des Rückgabetyps müssen mit der Schnittstellenvereinbarung übereinstimmen.

Funktionsaufruf

Hochschule RheinMain

- Funktionen können aus dem Hauptprogramm oder aus anderen Funktionen gerufen werden. Die aufrufende Funktion übergibt dazu die erforderlichen Parameter und erhält den von der Funktion ermittelten Rückgabewert:

Funktion

```
float maximum( float x, float y)
{
    if( x > y)
        return x;
    return y;
}
```

Hauptprogramm

Funktionsaufruf

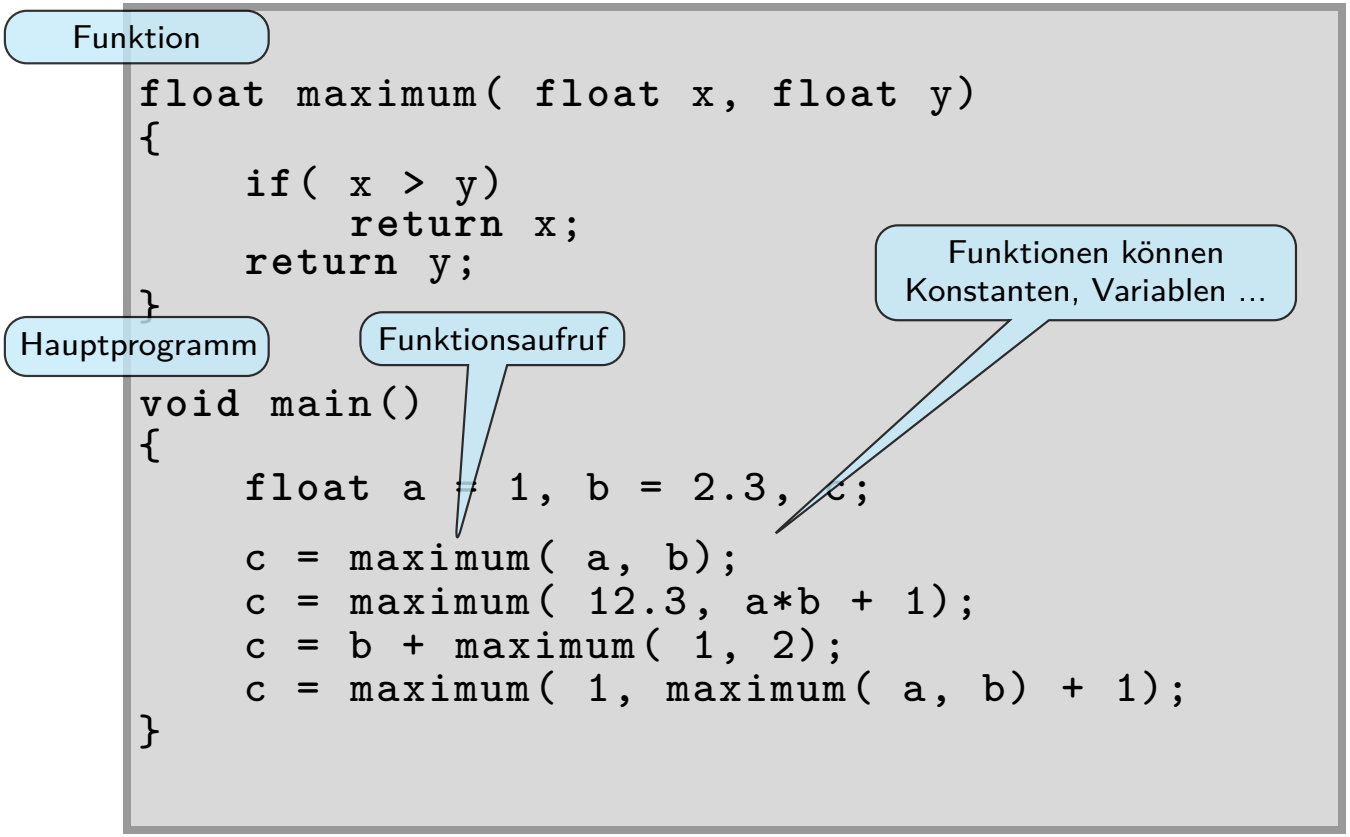
```
void main()
{
    float a = 1, b = 2.3, c;

    c = maximum( a, b);
    c = maximum( 12.3, a*b + 1);
    c = b + maximum( 1, 2);
    c = maximum( 1, maximum( a, b) + 1);
}
```

- Anzahl und Typen der beim Aufruf übergebenen Parameter und die Verwendung des Rückgabetyps müssen mit der Schnittstellenvereinbarung übereinstimmen.

Funktionsaufruf

Hochschule RheinMain

- Funktionen können aus dem Hauptprogramm oder aus anderen Funktionen gerufen werden. Die aufrufende Funktion übergibt dazu die erforderlichen Parameter und erhält den von der Funktion ermittelten Rückgabewert:



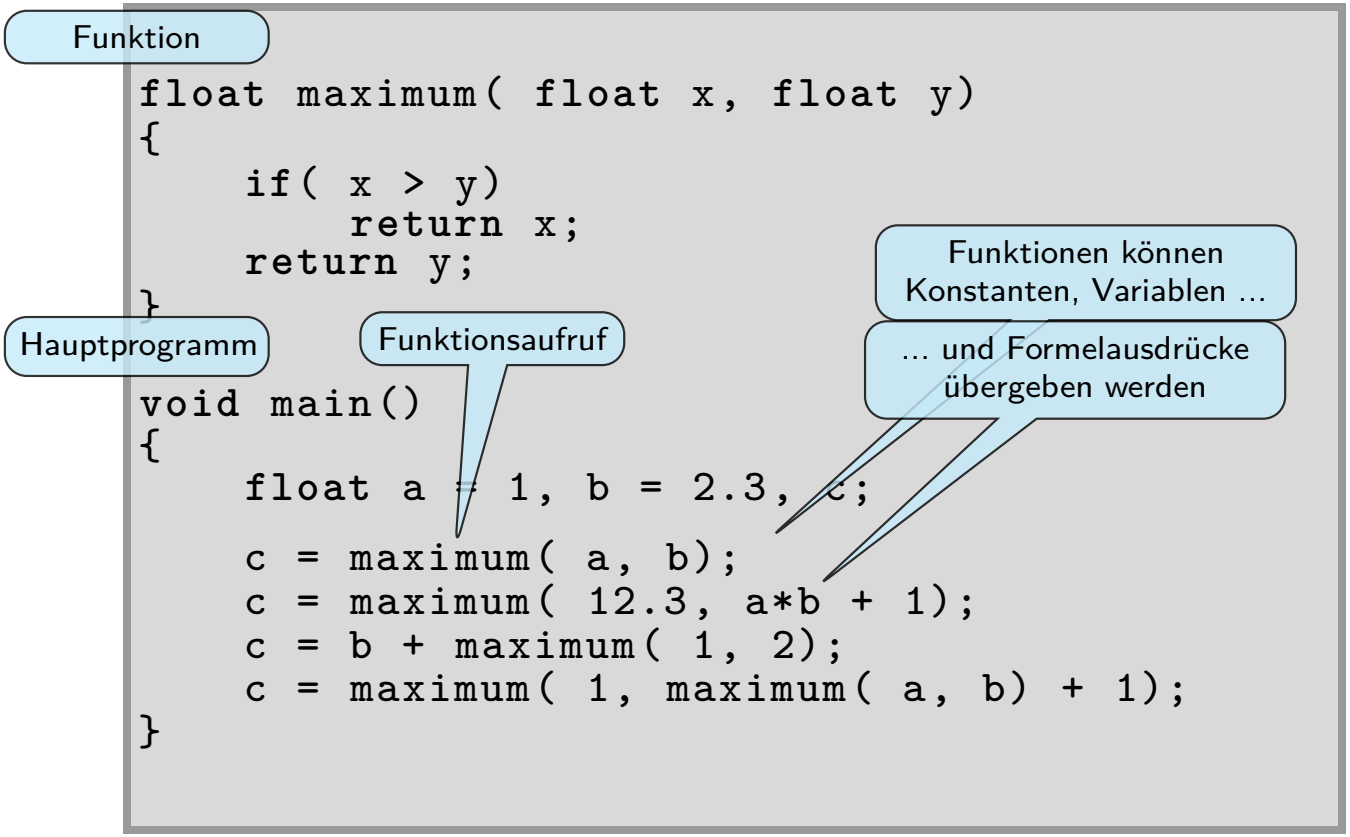

```
float maximum( float x, float y)
{
    if( x > y)
        return x;
    return y;
}

void main()
{
    float a = 1, b = 2.3, c;

    c = maximum( a, b);
    c = maximum( 12.3, a*b + 1);
    c = b + maximum( 1, 2);
    c = maximum( 1, maximum( a, b) + 1);
}
```

- Anzahl und Typen der beim Aufruf übergebenen Parameter und die Verwendung des Rückgabetyps müssen mit der Schnittstellenvereinbarung übereinstimmen.

Funktionsaufruf

Hochschule RheinMain

- Funktionen können aus dem Hauptprogramm oder aus anderen Funktionen gerufen werden. Die aufrufende Funktion übergibt dazu die erforderlichen Parameter und erhält den von der Funktion ermittelten Rückgabewert:



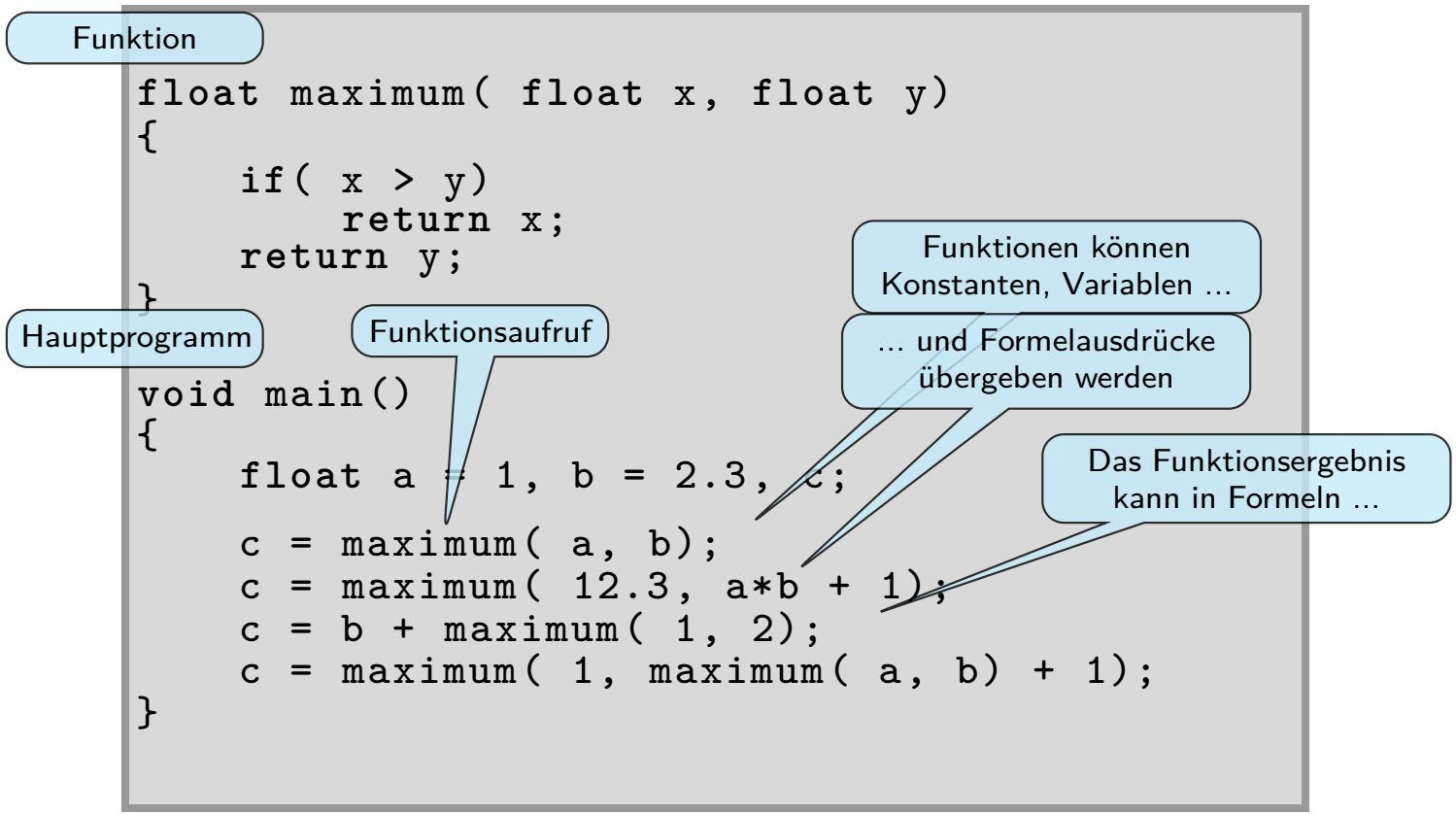

```
float maximum( float x, float y)
{
    if( x > y)
        return x;
    return y;
}

void main()
{
    float a = 1, b = 2.3, c;

    c = maximum( a, b);
    c = maximum( 12.3, a*b + 1);
    c = b + maximum( 1, 2);
    c = maximum( 1, maximum( a, b) + 1);
}
```

- Anzahl und Typen der beim Aufruf übergebenen Parameter und die Verwendung des Rückgabetyps müssen mit der Schnittstellenvereinbarung übereinstimmen.

Funktionsaufruf

Hochschule RheinMain

- Funktionen können aus dem Hauptprogramm oder aus anderen Funktionen gerufen werden. Die aufrufende Funktion übergibt dazu die erforderlichen Parameter und erhält den von der Funktion ermittelten Rückgabewert:

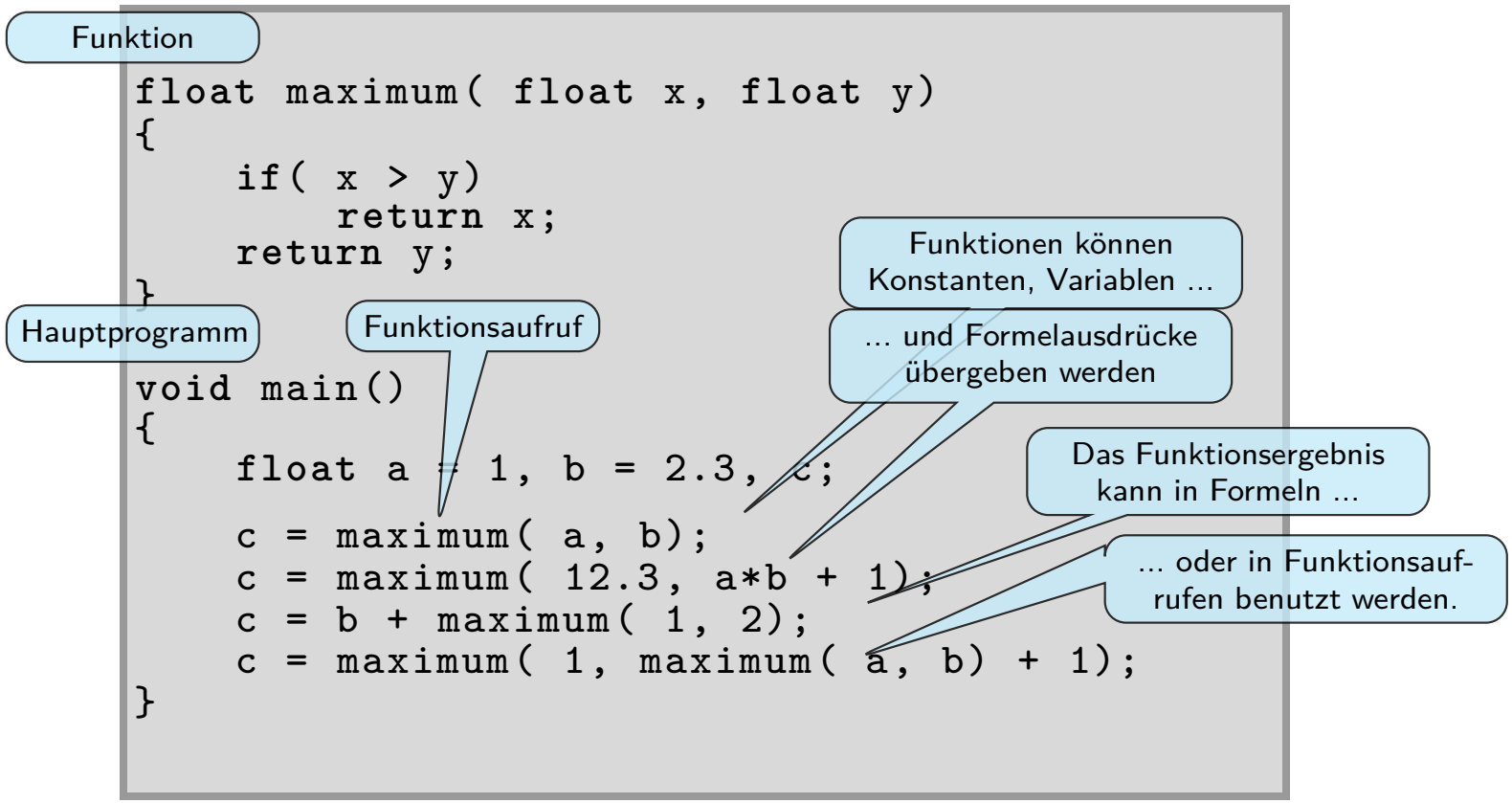


- Anzahl und Typen der beim Aufruf übergebenen Parameter und die Verwendung des Rückgabetyps müssen mit der Schnittstellenvereinbarung übereinstimmen.

Funktionsaufruf

Hochschule RheinMain

- Funktionen können aus dem Hauptprogramm oder aus anderen Funktionen gerufen werden. Die aufrufende Funktion übergibt dazu die erforderlichen Parameter und erhält den von der Funktion ermittelten Rückgabewert:

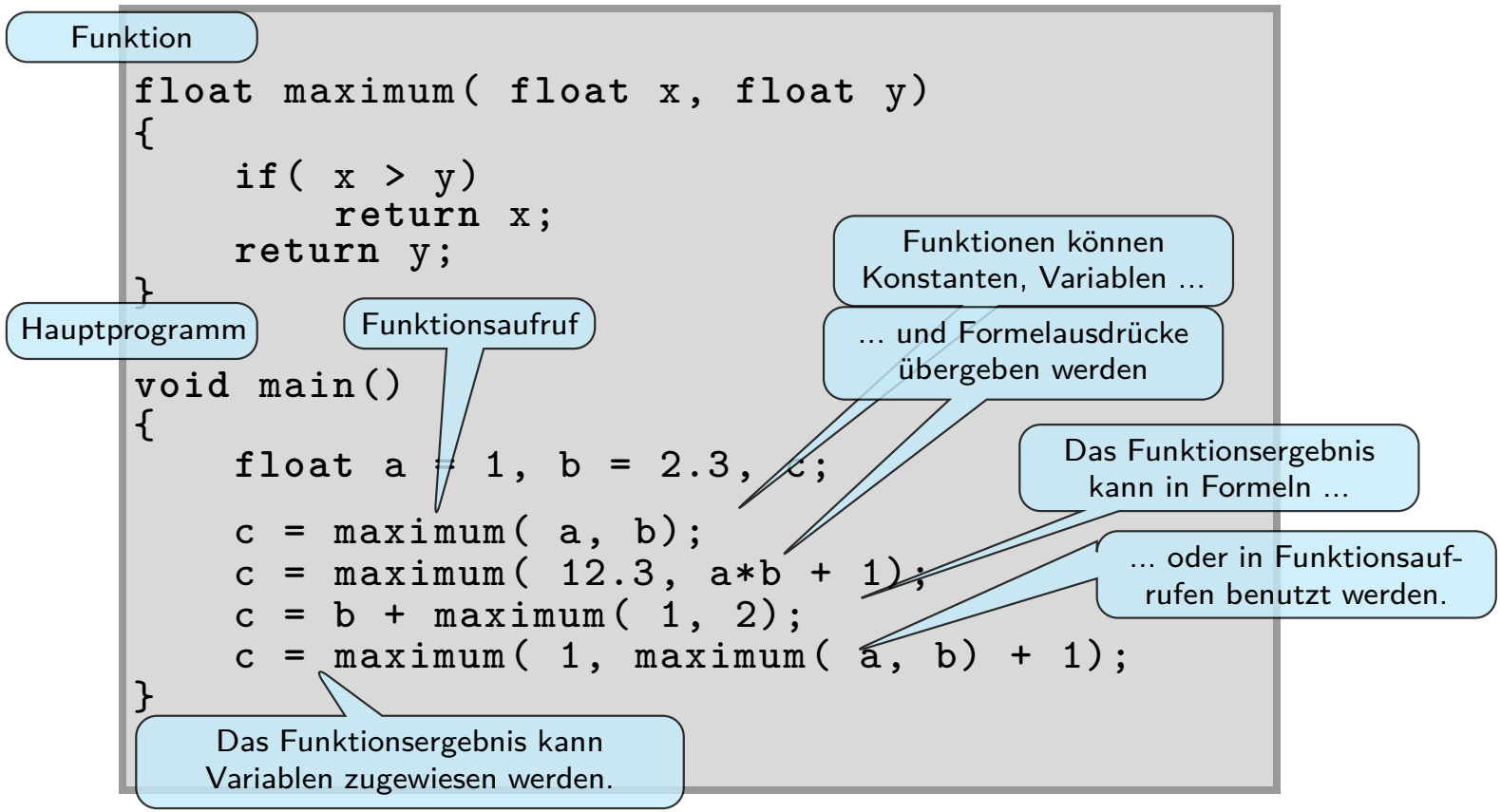


- Anzahl und Typen der beim Aufruf übergebenen Parameter und die Verwendung des Rückgabetyps müssen mit der Schnittstellenvereinbarung übereinstimmen.

Funktionsaufruf

Hochschule RheinMain

- Funktionen können aus dem Hauptprogramm oder aus anderen Funktionen gerufen werden. Die aufrufende Funktion übergibt dazu die erforderlichen Parameter und erhält den von der Funktion ermittelten Rückgabewert:



- Anzahl und Typen der beim Aufruf übergebenen Parameter und die Verwendung des Rückgabetyps müssen mit der Schnittstellenvereinbarung übereinstimmen.

Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.

```
int vergleich( float x, int y)
{
  if( x == y)
    return 1;
  return 0;
}

int ausgabe()
{
  printf( "Hallo Welt");
  return 1;
}

void test()
{
  int v;

  v = vergleich(1.0, 17);

  if( v == 0)
    return;
  ausgabe();
}
```

Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.

Verschiedene Parameter- und Rückgabetypen.

```
int vergleich( float x, int y)
{
  if( x == y)
    return 1;
  return 0;
}

int ausgabe()
{
  printf( "Hallo Welt");
  return 1;
}

void test()
{
  int v;

  v = vergleich(1.0, 17);

  if( v == 0)
    return;
  ausgabe();
}
```

Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.

Verschiedene Parameter- und Rückgabetypen.

```
int vergleich( float x, int y)
{
  if( x == y)
    return 1;
  return 0;
}

int ausgabe()
{
  printf( "Hallo Welt");
  return 1;
}

void test()
{
  int v;

  v = vergleich(1.0, 17);

  if( v == 0)
    return;
  ausgabe();
}
```

Es muss immer einen Returnwert geben.

Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.

Verschiedene Parameter- und Rückgabetypen.

```
int vergleich( float x, int y)
{
  if( x == y)
    return 1;
  return 0;
}

int ausgabe()
{
  printf( "Hallo Welt");
  return 1;
}

void test()
{
  int v;

  v = vergleich(1.0, 17);

  if( v == 0)
    return;
  ausgabe();
}
```

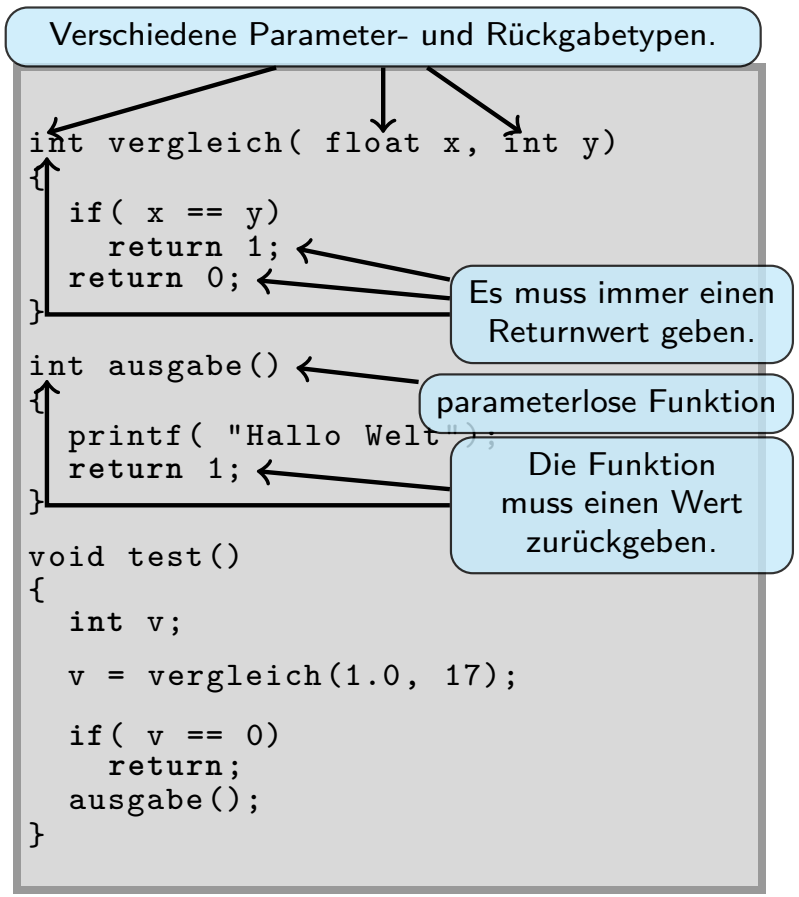
Es muss immer einen Returnwert geben.

parameterlose Funktion

Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

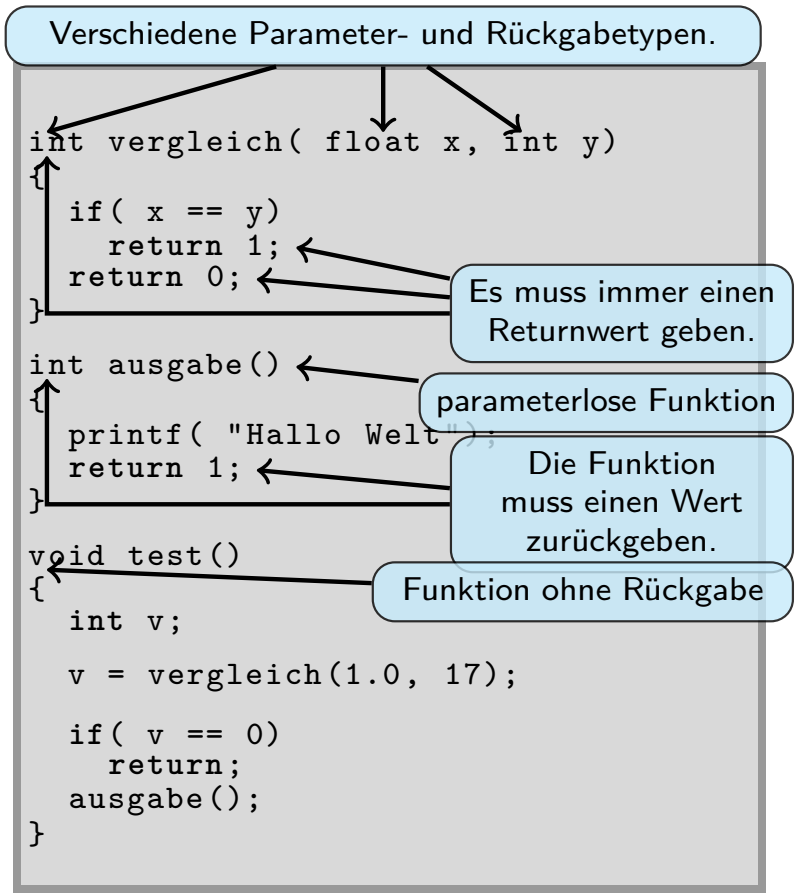
- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.



Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

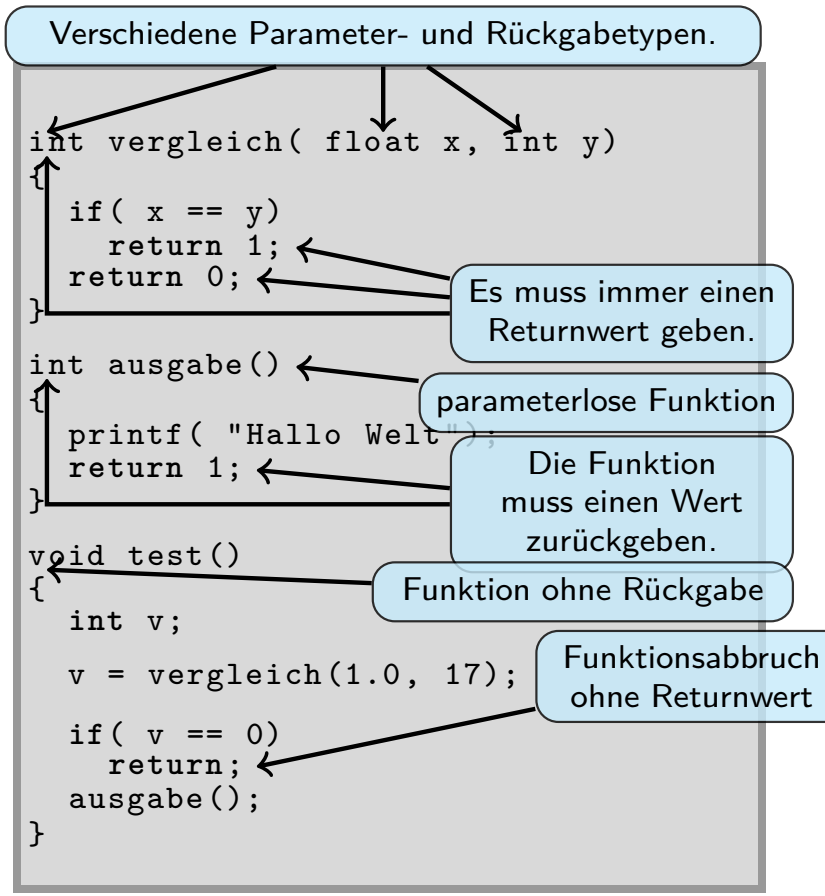
- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.



Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

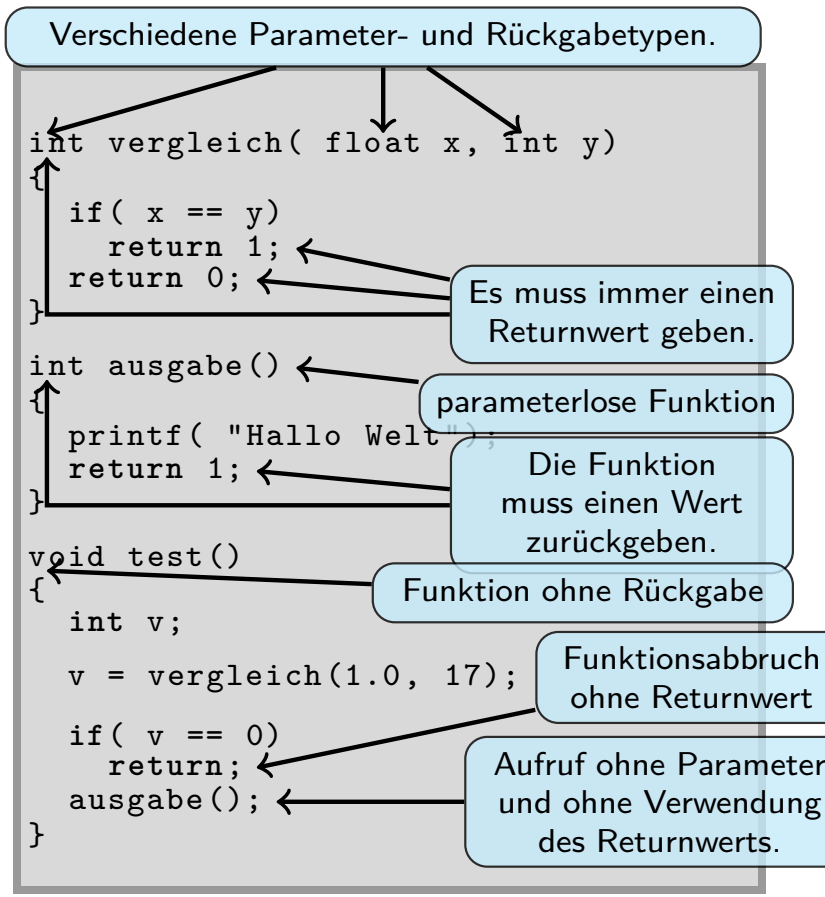
- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.



Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

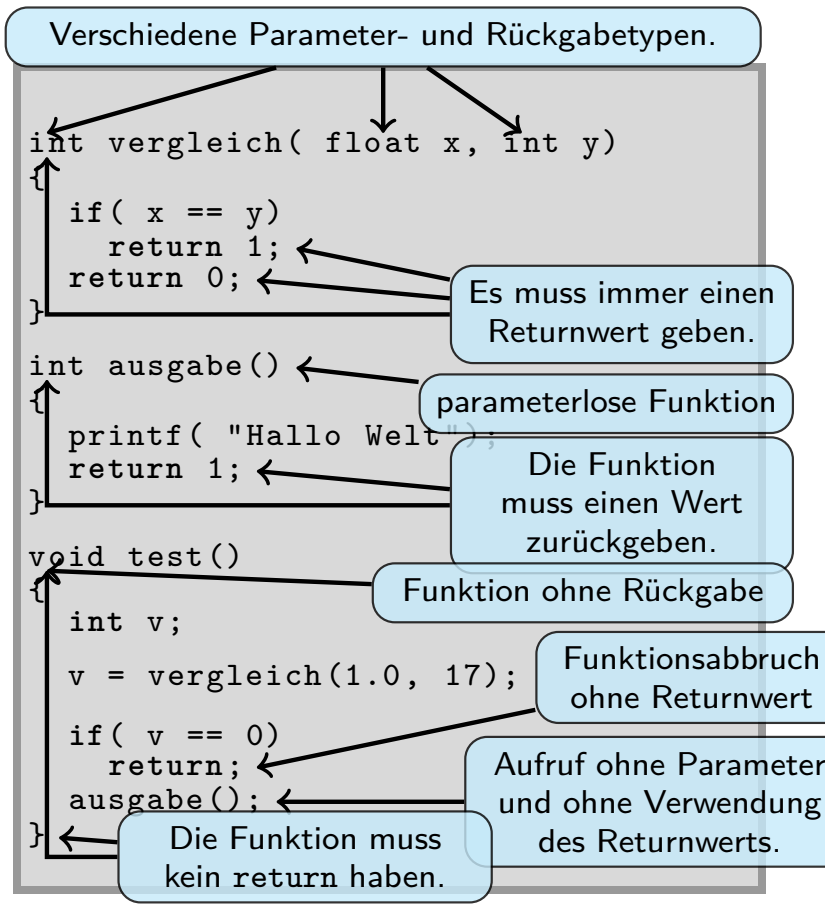
- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.



Unterschiedliche Parameter und Returnwerte

- Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabetyp `void`:

- Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passender Wert zurückgegeben werden.
- Wert zurückgegeben werden.Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.
- An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.
- Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.



Das Blackbox-Prinzip

- Eine Funktion ist eine „doppelte Blackbox“, in die von außen niemand hineinsehen, aus der aber auch von innen niemand heraussehen kann. Von innen und außen sieht man nur die Schnittstelle.

```
void main()
{
  float a = 1, b = 2.3, c;

  c = maximum( a, b);
}
```

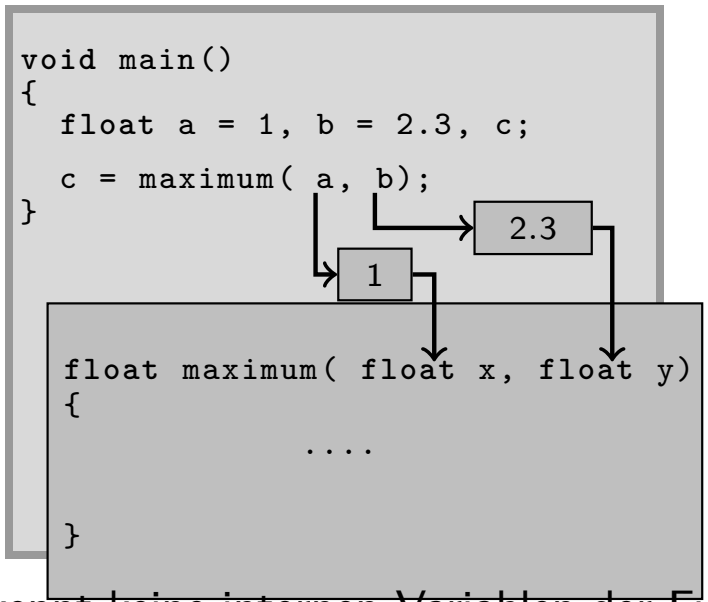
```
float maximum( float x, float y)
{
            ....

}
```

- Das aufrufende Programm kennt keine internen Variablen der Funktion. Umgekehrt kennt auch die Funktion keine Variablen des Hauptprogramms. Eine zufällige Gleichheit von Variablennamen ändert daran nichts. Weder das Hauptprogramm kann auf Variablen des Unterprogramms noch das Unterprogramm kann auf Variablen des Hauptprogramms zugreifen.
- An der Schnittstelle werden **Kopien** der Parameterwerte übergeben. Änderungen dieser Werte haben keine Auswirkungen auf das rufende Programm.

Das Blackbox-Prinzip

Hochschule RheinMain

- Eine Funktion ist eine „doppelte Blackbox“, in die von außen niemand hineinsehen, aus der aber auch von innen niemand heraussehen kann. Von innen und außen sieht man nur die Schnittstelle.

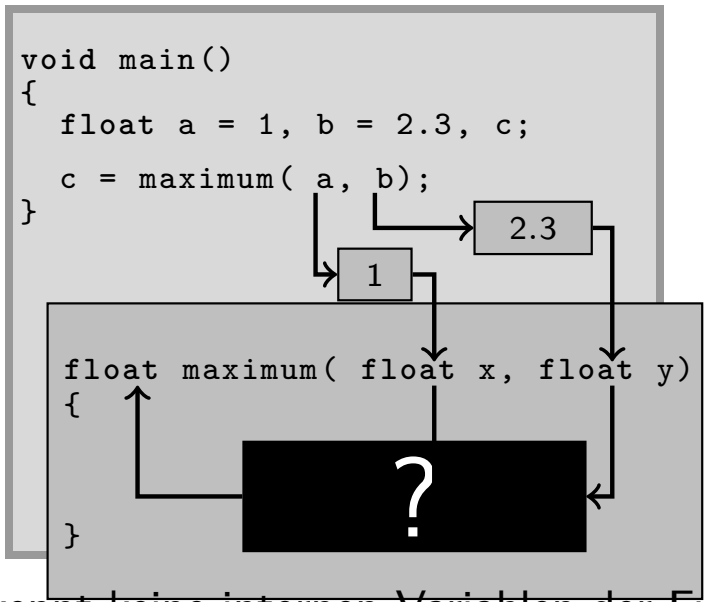


- Das aufrufende Programm kennt keine internen Variablen der Funktion. Umgekehrt kennt auch die Funktion keine Variablen des Hauptprogramms. Eine zufällige Gleichheit von Variablennamen ändert daran nichts. Weder das Hauptprogramm kann auf Variablen des Unterprogramms noch das Unterprogramm kann auf Variablen des Hauptprogramms zugreifen.
- An der Schnittstelle werden **Kopien** der Parameterwerte übergeben. Änderungen dieser Werte haben keine Auswirkungen auf das rufende Programm.

Das Blackbox-Prinzip

Hochschule RheinMain

- Eine Funktion ist eine „doppelte Blackbox“, in die von außen niemand hineinsehen, aus der aber auch von innen niemand heraussehen kann. Von innen und außen sieht man nur die Schnittstelle.

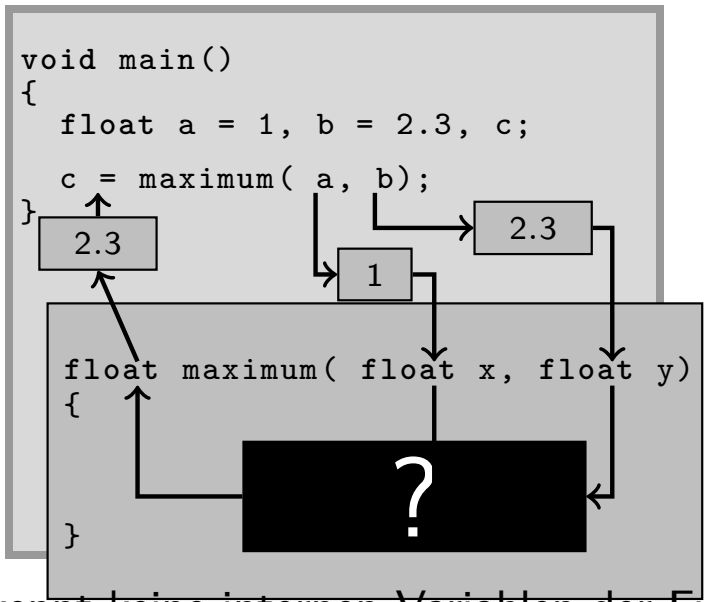


- Das aufrufende Programm kennt keine internen Variablen der Funktion. Umgekehrt kennt auch die Funktion keine Variablen des Hauptprogramms. Eine zufällige Gleichheit von Variablennamen ändert daran nichts. Weder das Hauptprogramm kann auf Variablen des Unterprogramms noch das Unterprogramm kann auf Variablen des Hauptprogramms zugreifen.
- An der Schnittstelle werden **Kopien** der Parameterwerte übergeben. Änderungen dieser Werte haben keine Auswirkungen auf das rufende Programm.

Das Blackbox-Prinzip

Hochschule RheinMain

- Eine Funktion ist eine „doppelte Blackbox“, in die von außen niemand hineinsehen, aus der aber auch von innen niemand heraussehen kann. Von innen und außen sieht man nur die Schnittstelle.



- Das aufrufende Programm kennt keine internen Variablen der Funktion. Umgekehrt kennt auch die Funktion keine Variablen des Hauptprogramms. Eine zufällige Gleichheit von Variablennamen ändert daran nichts. Weder das Hauptprogramm kann auf Variablen des Unterprogramms noch das Unterprogramm kann auf Variablen des Hauptprogramms zugreifen.
- An der Schnittstelle werden **Kopien** der Parameterwerte übergeben. Änderungen dieser Werte haben keine Auswirkungen auf das rufende Programm.

Technischer Ablauf: Funktionsaufruf

Nötige Schritte

- Parameter**kopien** für aufgerufene Funktion hinterlegen
- Programmsteuerung an die Prozedur übergeben
- Arbeitsspeicher für Prozedur bereitstellen (lokale Variablen)
- Prozedur ausführen
- Ergebnis an den Aufrufer übergeben
- Arbeitsspeicher der Prozedur wieder freigeben
- zum Aufrufer zurückkehren

Beispiel: Funktionsaufruf

```
int funktion(int p1, int p2, ..)
{
  int lokal1, lokal2, ...;
  int ergebnis;
  .......
  return(ergebnis);
}

main()
{
  int a, b, c, ...;
  int resultat;
  ......
  resultat = funktion(a, b, ...);
  ......
}
```

Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

1. Argumentkopien auf den Stack
2. Call → Rücksprungadresse auf den Stack

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```

EPC

Technisch: Funktionsaufruf

Hochschule RheinMain

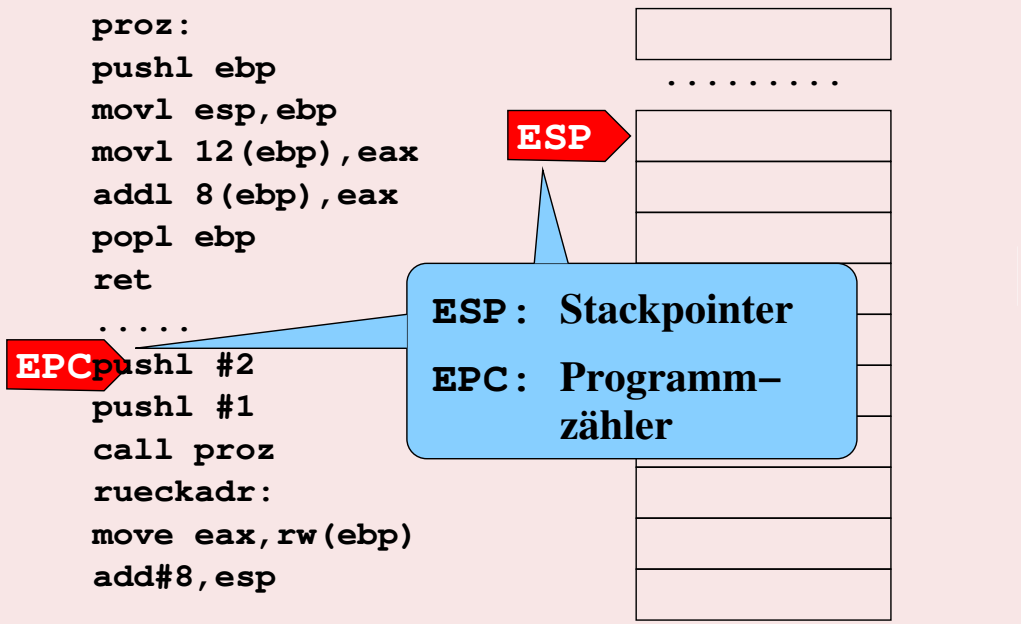
Beispiel Intel x86 (32-bit)

1. Argumentkopien auf den Stack
2. Call → Rücksprungadresse auf den Stack

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

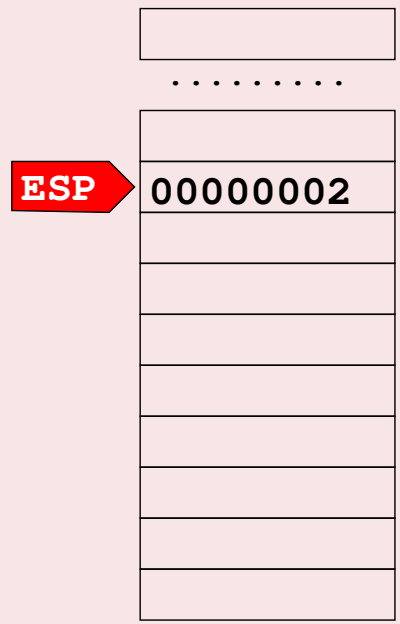
1. Argumentkopien auf den Stack
2. Call → Rücksprungadresse auf den Stack

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```

EPC



Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

1. Argumentkopien auf den Stack
2. Call → Rücksprungadresse auf den Stack

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

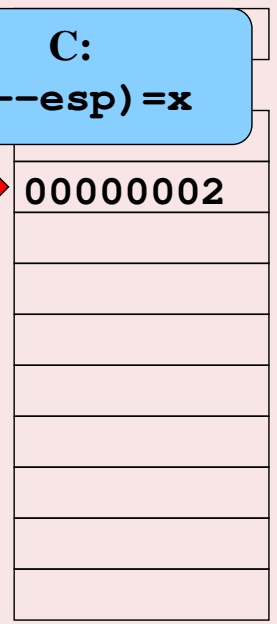
```
proz:
pushl ebp
movl esp,
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```

EPC

Assembler: push x

C: *(--esp)=x

ESP



Technisch: Funktionsaufruf

Hochschule RheinMain

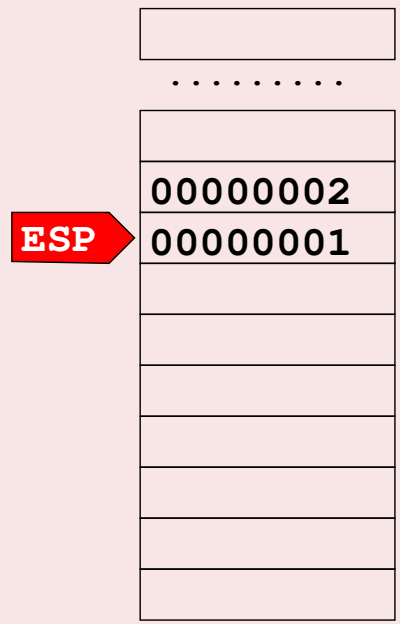
Beispiel Intel x86 (32-bit)

1. Argumentkopien auf den Stack
2. Call → Rücksprungadresse auf den Stack

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
EPCcall proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

1. Argumentkopien auf den Stack
2. Call → Rücksprungadresse auf den Stack

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl
movl
movl
addl
popl
ret
.....
push  #2
pushl #1
EPCcall proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```

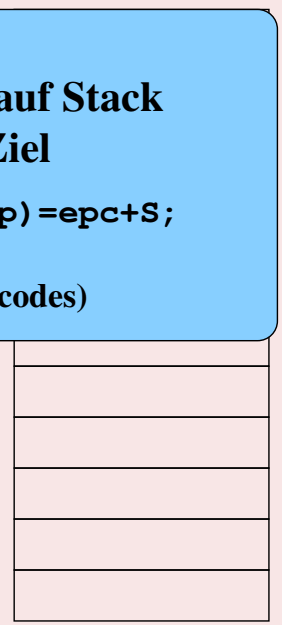
Call bewirkt:

1. Rücksprungadresse auf Stack

2. Programmzähler = Ziel

```
call proz      *(--esp)=epc+S;
epc=proz;
```

(S=Größe des Call–Opcodes)



Technisch: Funktionsaufruf

Hochschule RheinMain

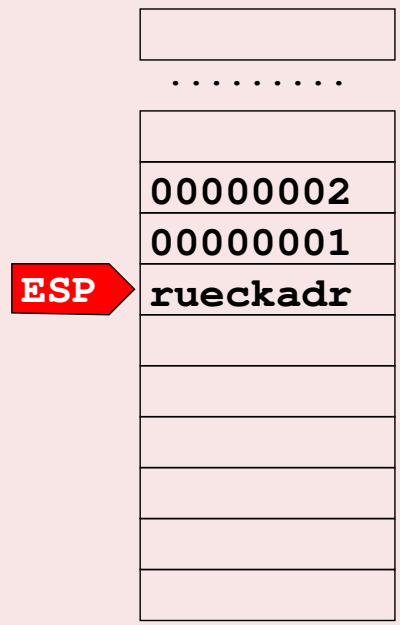
Beispiel Intel x86 (32-bit)

3. „Framepointer" (EBP) retten
4. Neuen Framepointer laden

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
EPC pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

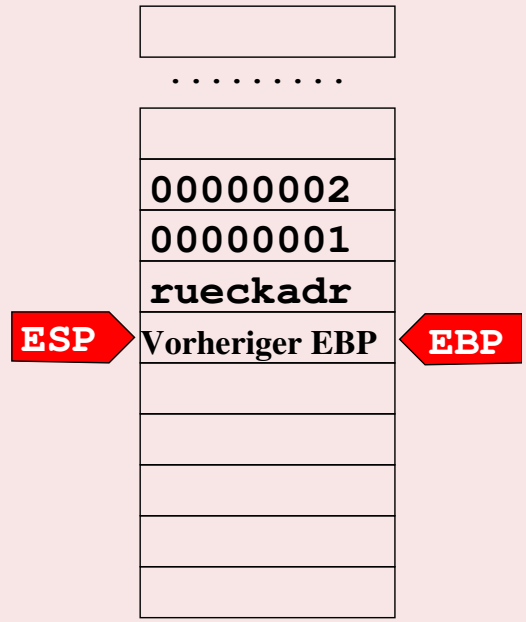
Beispiel Intel x86 (32-bit)

3. „Framepointer" (EBP) retten
4. Neuen Framepointer laden

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

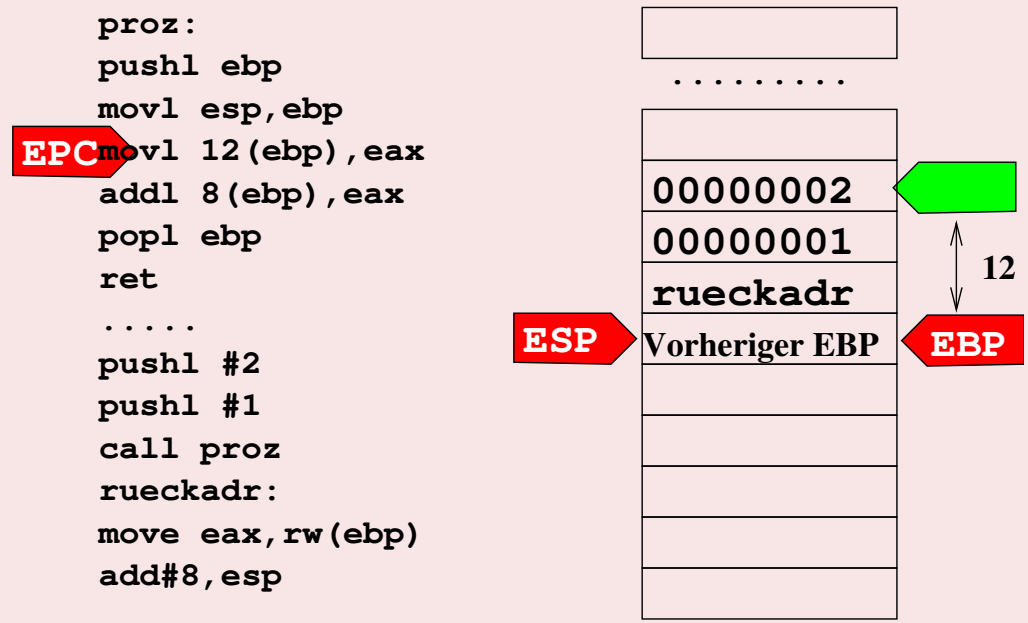
Beispiel Intel x86 (32-bit)

5. Parameter addierennor
6. Returnwert in Register (hier: EAX)

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

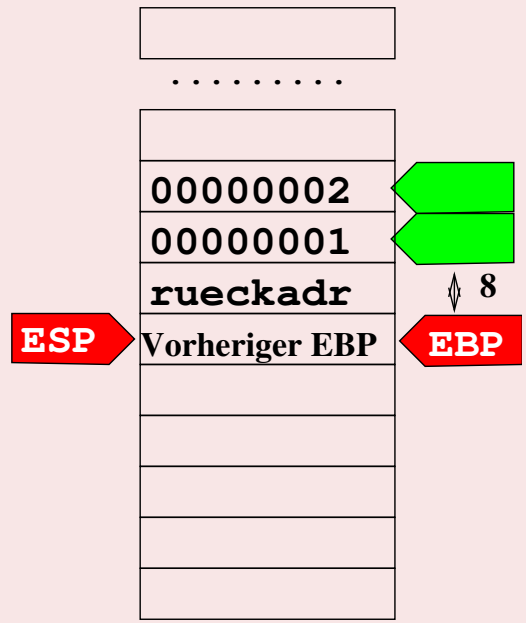
5. Parameter addierennor
6. Returnwert in Register (hier: EAX)

Beispiel: Funktionsaufruf

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

.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```

EPC



Technisch: Funktionsaufruf

Hochschule RheinMain

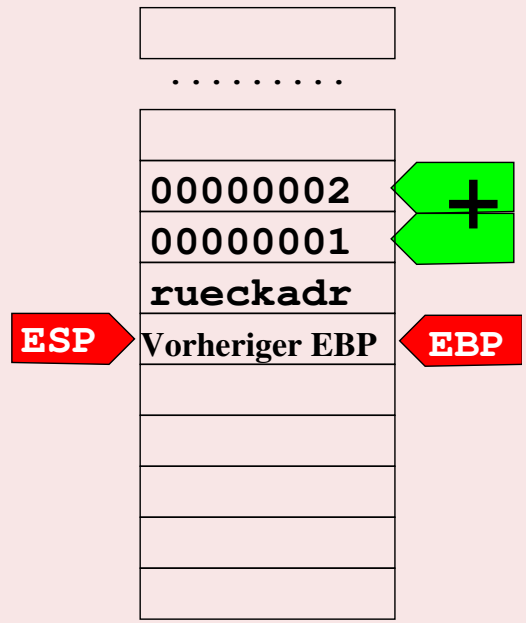
Beispiel Intel x86 (32-bit)

5. Parameter addierennor
6. Returnwert in Register (hier: EAX)

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
EPC addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

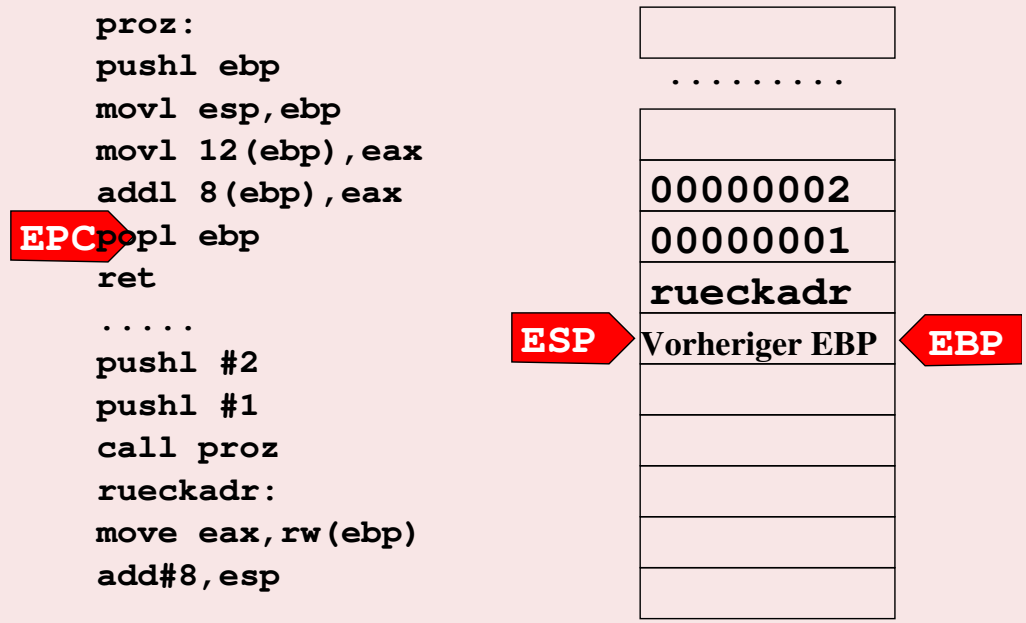
Beispiel Intel x86 (32-bit)

7. Alten Framepointer wiederherstellen
8. Rückkehr zum Aufrufer

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

7. Alten Framepointer wiederherstellen
8. Rückkehr zum Aufrufer

Beispiel: Funktionsaufruf

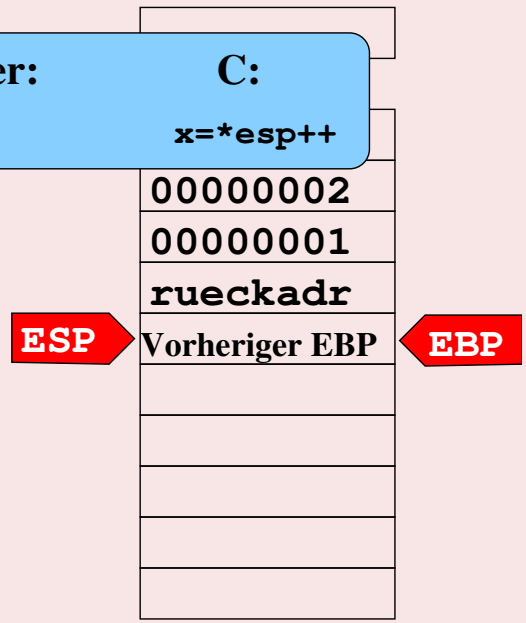
```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,eb
movl 12(ebp
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```

EPC

Assembler: pop x

C: x=*esp++



Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

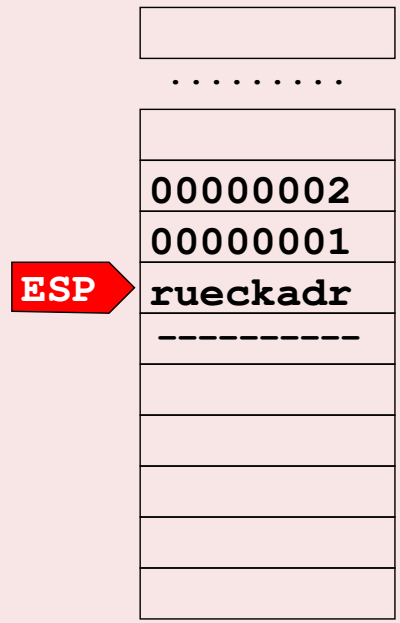
7. Alten Framepointer wiederherstellen
8. Rückkehr zum Aufrufer

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```

EPC



Technisch: Funktionsaufruf

Hochschule RheinMain

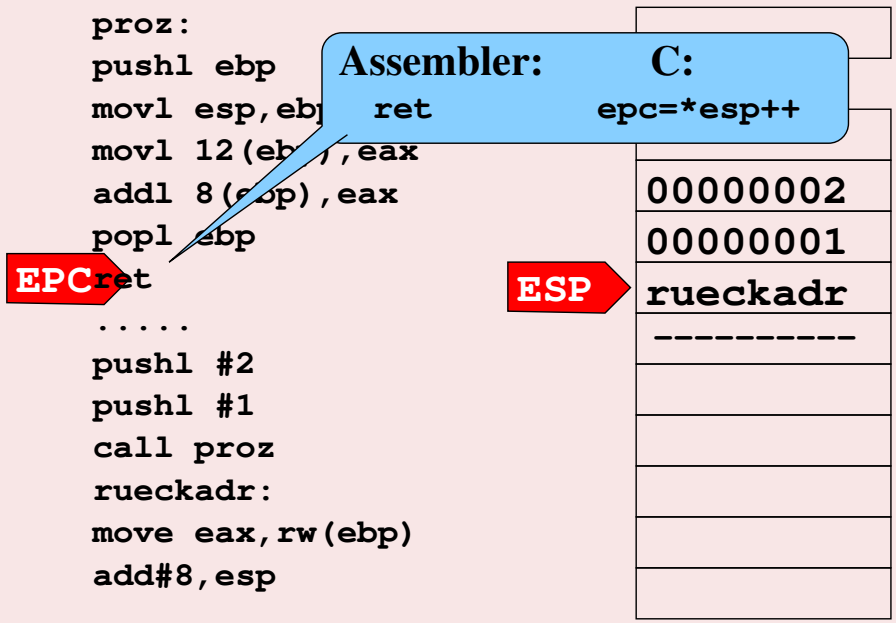
Beispiel Intel x86 (32-bit)

7. Alten Framepointer wiederherstellen
8. Rückkehr zum Aufrufer

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

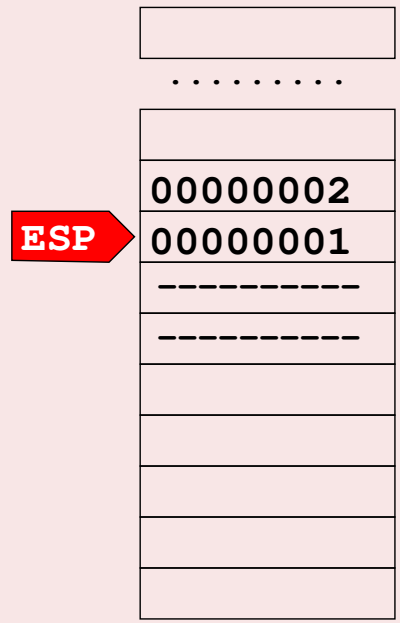
9. Ergebnis Speichern
10. Stack abräumen

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
movl eax,rw(ebp)
add#8,esp
```

EPC



Technisch: Funktionsaufruf

Hochschule RheinMain

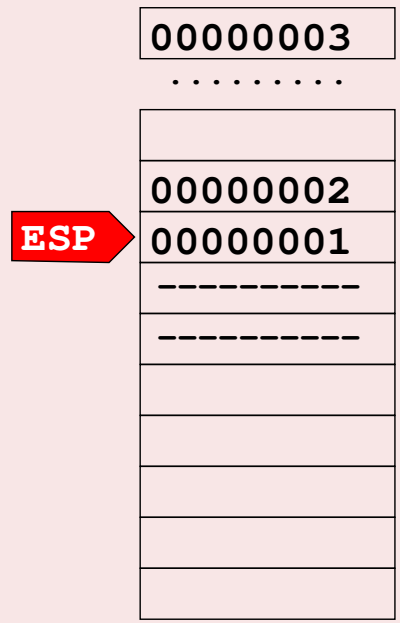
Beispiel Intel x86 (32-bit)

9. Ergebnis Speichern
10. Stack abräumen

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
EPCadd#8,esp
```



Technisch: Funktionsaufruf

Hochschule RheinMain

Beispiel Intel x86 (32-bit)

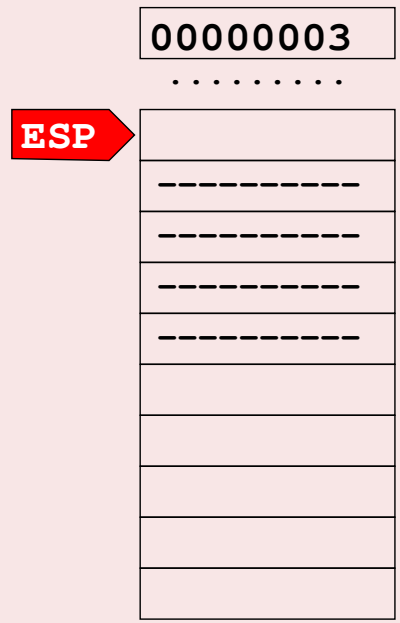
9. Ergebnis Speichern
10. Stack abräumen

Beispiel: Funktionsaufruf

```
int proz(int a, int b)
{
  return(a+b);
}
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
.....
```

EPC



Arrays als Parameter

- Wird ein Array an eine Funktion übergeben, so erhält die Funktion Zugriff auf die Originaldaten des Hauptprogramms. Es entsteht keine Kopie des gesamten Arrays sondern nur eine Kopie der Zugriffsinformation (Dazu mehr im Abschnitt über Zeiger und Adressen).

```
void main()
{
  int daten[10];
  init( 10, daten);
  ausgeben( 10, daten);
  umkehren( 10, daten);
  ausgeben( 10, daten);
}
```

.

```
void init( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    dat[i] = 2*i;
}
```

```
void umkehren( int anz, int dat[])
{
  int v, h, t;
  for(v = 0,h = anz-1; v < h; v++,h--)
  {
    t = dat[v];
    dat[v] = dat[h];
    dat[h] = t;
  }
}
```

```
void ausgeben( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    printf( "%d ", dat[i]);
  printf( "\n");
}
```

Arrays als Parameter

- Wird ein Array an eine Funktion übergeben, so erhält die Funktion Zugriff auf die Originaldaten des Hauptprogramms. Es entsteht keine Kopie des gesamten Arrays sondern nur eine Kopie der Zugriffsinformation (Dazu mehr im Abschnitt über Zeiger und Adressen).

Das Array wird im Hauptprogramm angelegt.

```
void main()
{
  int daten[10];
  init( 10, daten);
  ausgeben( 10, daten);
  umkehren( 10, daten);
  ausgeben( 10, daten);
}
```

.

```
void init( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    dat[i] = 2*i;
}
```

```
void umkehren( int anz, int dat[])
{
  int v, h, t;
  for(v = 0,h = anz-1; v < h; v++,h--)
  {
    t = dat[v];
    dat[v] = dat[h];
    dat[h] = t;
  }
}
```

```
void ausgeben( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    printf( "%d ", dat[i]);
  printf( "\n");
}
```

Arrays als Parameter

- Wird ein Array an eine Funktion übergeben, so erhält die Funktion Zugriff auf die Originaldaten des Hauptprogramms. Es entsteht keine Kopie des gesamten Arrays sondern nur eine Kopie der Zugriffsinformation (Dazu mehr im Abschnitt über Zeiger und Adressen).

Das Array wird im Hauptprogramm angelegt.

```
void main()
{
  int daten[10];
  init( 10, daten);
  ausgeben( 10, daten);
  umkehren( 10, daten);
  ausgeben( 10, daten);
}
```

.Die Unterprogramme erhalten die Anzahl der Elemente und den Zugriff auf die Originaldaten. Die Originaldaten werden initialisiert, umgekehrt und ausgegeben.

```
void init( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    dat[i] = 2*i;
}
```

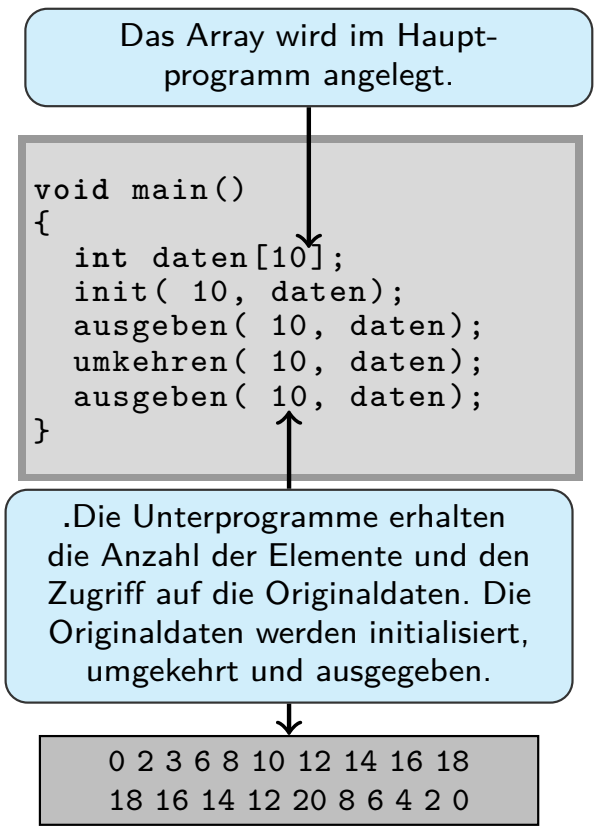
```
void umkehren( int anz, int dat[])
{
  int v, h, t;
  for(v = 0,h = anz-1; v < h; v++,h--)
  {
    t = dat[v];
    dat[v] = dat[h];
    dat[h] = t;
  }
}
```

```
void ausgeben( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    printf( "%d ", dat[i]);
  printf( "\n");
}
```

Arrays als Parameter

Hochschule RheinMain

- Wird ein Array an eine Funktion übergeben, so erhält die Funktion Zugriff auf die Originaldaten des Hauptprogramms. Es entsteht keine Kopie des gesamten Arrays sondern nur eine Kopie der Zugriffsinformation (Dazu mehr im Abschnitt über Zeiger und Adressen).




```
void init( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    dat[i] = 2*i;
}
```

```
void umkehren( int anz, int dat[])
{
  int v, h, t;
  for(v = 0,h = anz-1; v < h; v++,h--)
  {
    t = dat[v];
    dat[v] = dat[h];
    dat[h] = t;
  }
}
```

```
void ausgeben( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    printf( "%d ", dat[i]);
  printf( "\n");
}
```

Arrays als Parameter

Hochschule RheinMain

- Wird ein Array an eine Funktion übergeben, so erhält die Funktion Zugriff auf die Originaldaten des Hauptprogramms. Es entsteht keine Kopie des gesamten Arrays sondern nur eine Kopie der Zugriffsinformation (Dazu mehr im Abschnitt über Zeiger und Adressen).

Das Array wird im Hauptprogramm angelegt.

```
void main()
{
  int daten[10];
  init( 10, daten);
  ausgeben( 10, daten);
  umkehren( 10, daten);
  ausgeben( 10, daten);
}
```

.Die Unterprogramme erhalten die Anzahl der Elemente und den Zugriff auf die Originaldaten. Die Originaldaten werden initialisiert, umgekehrt und ausgegeben.

```
0 2 3 6 8 10 12 14 16 18
18 16 14 12 20 8 6 4 2 0
```

```
void init( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    dat[i] = 2*i;
}
```

```
void umkehren( int anz, int dat[])
{
  int v, h, t;
  for(v = 0,h = anz-1; v < h; v++,h--)
  {
    t = dat[v];
    dat[v] = dat[h];
    dat[h] = t;
  }
}
```

```
void ausgeben( int anz, int dat[])
{
  int i;
  for( i = 0; i < anz; i++)
    printf( "%d ", dat[i]);
  printf( "\n");
}
```

Die Größe des Arrays (hier 10) wird als Parameter übergeben, weil das Unterprogramm die Größe des Arrays nicht selbst ermitteln kann.

Strings als Funktionsparameter

- Strings sind Arrays und werden daher an der Funktionsschnittstelle wie Arrays behandelt:

```
void main()
{
  int l, v;
  l = stringlaenge( "qwert");
  printf( "Laenge: %d\n", l);
  v = stringvergleich( "qwert", "qwerz");
  if( v == 1)
      printf( "gleich\n");
  else
    printf( "ungleich\n");
}
```

```
Laenge: 5
ungleich
```

```
int stringlaenge( char s[])
{
  int i;
  for( i = 0; s[i] != 0; i++)
    ;
  return i;
}
```

```
int stringvergleich( char s1[], char s2[])
{
  int i;
  for( i = 0; (s1[i]!=0)&&(s1[i]==s2[i]); i++)
    ;
  return s1[i] == s2[i];
}
```

- Wegen des Terminatorzeichens kann das Unterprogramm das Ende des Strings ermitteln. Es muss daher keine Längeninformation mitgegeben werden. Verfügt das rufende Programm über die Länge des Strings, so kann diese mitgegeben werden, damit die Länge im Unterprogramm nicht erneut berechnet werden muß.
- Wird der String im Unterprogramm verändert, sollte die Puffergröße als Parameter mitgegeben werden, um Pufferüberschreitungen (Buffer Overflow) vermeiden zu können.

Strings als Funktionsparameter

- Strings können (wie Arrays) in Funktionen verändert werden. Zum Beispiel können Zeichen an einen String angehängt werden:

```
void append( char s[], char c)
{
  int i;
  for( i = 0; s[i] != 0; i++)
    ;
  s[i] = c;
  s[i+1] = 0;
}
```

```
void main()
{
  char txt[20];
  char b;
  txt[0] = 0;
  for( b = 'a'; b <= 'k'; b++)
  {
    append( txt, b);
    printf( "%s\n", txt);
  }
}
```

```
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
```

- Vorsicht bei Veränderungen von Strings im Unterprogramm.
- Programmierende sind dafür verantwortlich, dass keine Arraygrenzen überschritten werden und der String stets korrekt terminiert ist.
- Wird im obigen Beispiel append() mehr als 19 mal gerufen, stürzt das Programm ab. Fehlende Sicherungen gegen Pufferüberschreitungen sind eine Hauptfehlerquelle in C-Programmen.

Strings als Funktionsparameter

Hochschule RheinMain

- Strings können (wie Arrays) in Funktionen verändert werden. Zum Beispiel können Zeichen an einen String angehängt werden:

```
void append( char s[], char c)
{
  int i;
  for( i = 0; s[i] != 0; i++)
    ;
  s[i] = c;
  s[i+1] = 0;
}
```

```
void main()
{
  char txt[20];
  char b;
  txt[0] = 0;
  for( b = 'a'; b <= 'k'; b++)
  {
    append( txt, b);
    printf( "%s\n", txt);
  }
}
```

Leerer String

```
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
```

- Vorsicht bei Veränderungen von Strings im Unterprogramm.
- Programmierende sind dafür verantwortlich, dass keine Arraygrenzen überschritten werden und der String stets korrekt terminiert ist.
- Wird im obigen Beispiel append() mehr als 19 mal gerufen, stürzt das Programm ab. Fehlende Sicherungen gegen Pufferüberschreitungen sind eine Hauptfehlerquelle in C-Programmen.

Strings als Funktionsparameter

Hochschule RheinMain

- Strings können (wie Arrays) in Funktionen verändert werden. Zum Beispiel können Zeichen an einen String angehängt werden:

```
void append( char s[], char c)
{
  int i;
  for( i = 0; s[i] != 0; i++)
    ;
  s[i] = c;
  s[i+1] = 0;
}
```

```
void main()
{
  char txt[20];
  char b;
  txt[0] = 0;
  for( b = 'a'; b <= 'k'; b++)
  {
    append( txt, b);
    printf( "%s\n", txt);
  }
}
```

Leerer String

Die Zeichen a-k werden angehängt

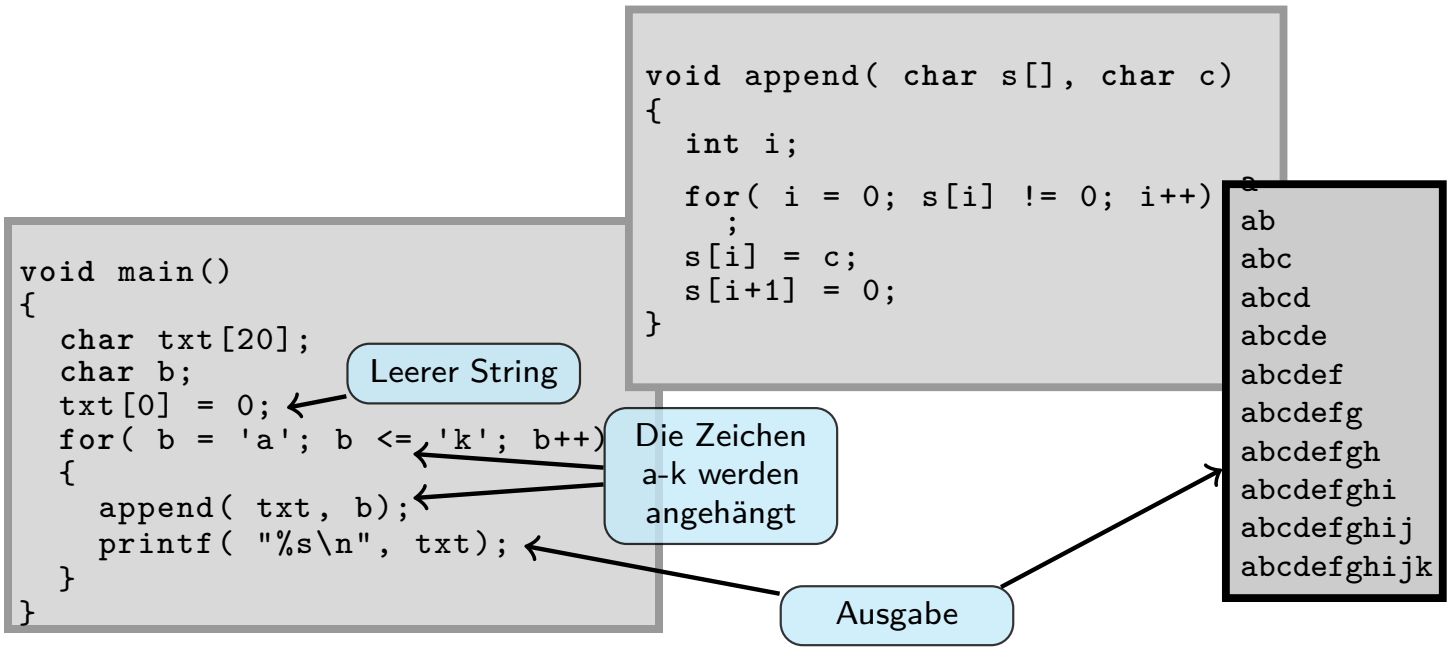
```
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
```

- Vorsicht bei Veränderungen von Strings im Unterprogramm.
- Programmierende sind dafür verantwortlich, dass keine Arraygrenzen überschritten werden und der String stets korrekt terminiert ist.
- Wird im obigen Beispiel append() mehr als 19 mal gerufen, stürzt das Programm ab. Fehlende Sicherungen gegen Pufferüberschreitungen sind eine Hauptfehlerquelle in C-Programmen.

Strings als Funktionsparameter

Hochschule RheinMain

- Strings können (wie Arrays) in Funktionen verändert werden. Zum Beispiel können Zeichen an einen String angehängt werden:

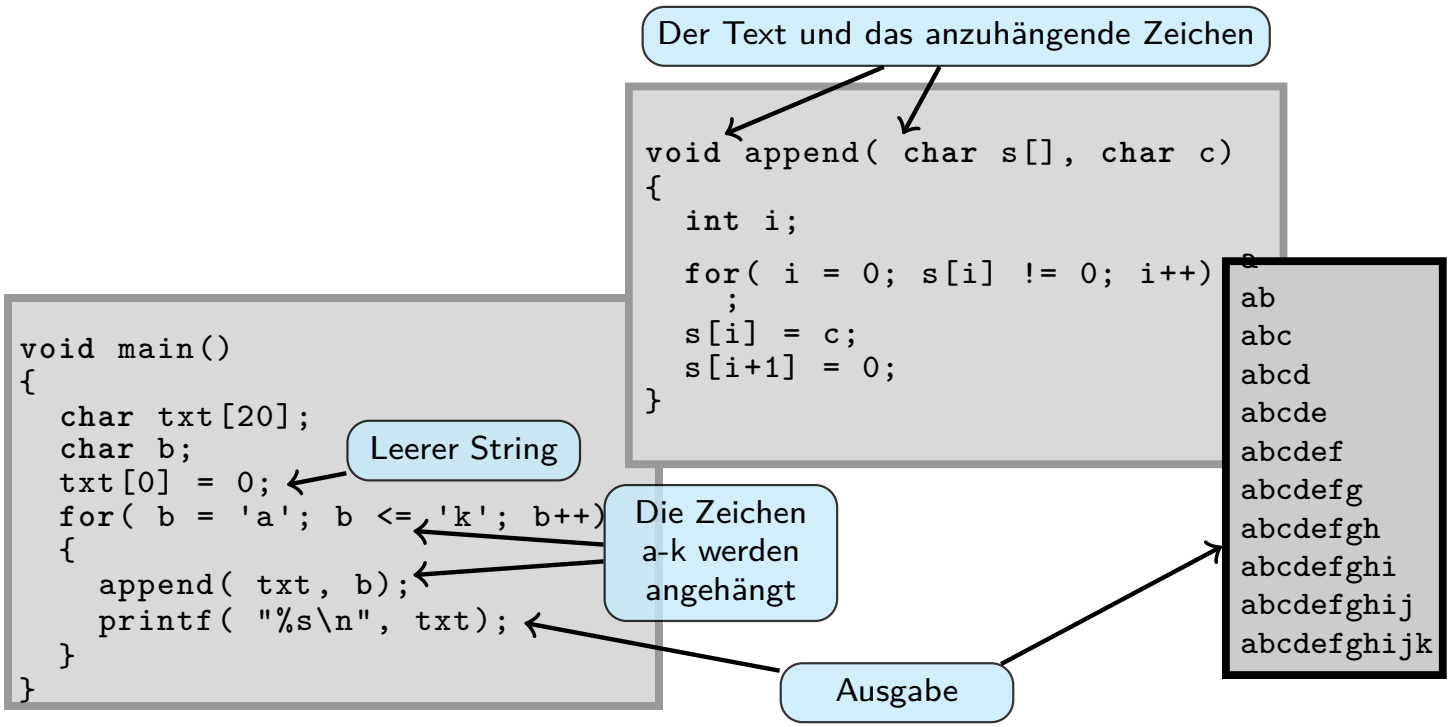


- Vorsicht bei Veränderungen von Strings im Unterprogramm.
- Programmierende sind dafür verantwortlich, dass keine Arraygrenzen überschritten werden und der String stets korrekt terminiert ist.
- Wird im obigen Beispiel append() mehr als 19 mal gerufen, stürzt das Programm ab. Fehlende Sicherungen gegen Pufferüberschreitungen sind eine Hauptfehlerquelle in C-Programmen.

Strings als Funktionsparameter

Hochschule RheinMain

- Strings können (wie Arrays) in Funktionen verändert werden. Zum Beispiel können Zeichen an einen String angehängt werden:

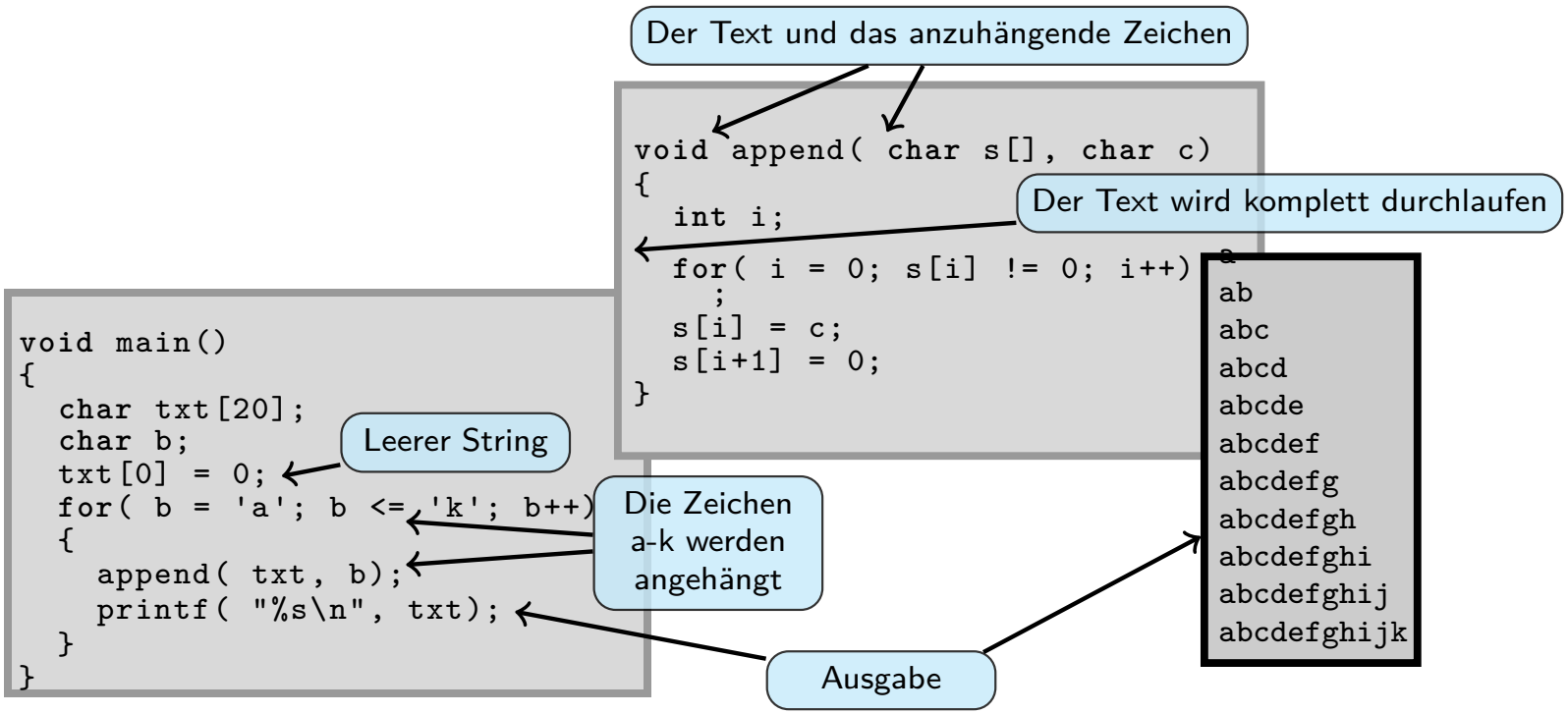


- Vorsicht bei Veränderungen von Strings im Unterprogramm.
- Programmierende sind dafür verantwortlich, dass keine Arraygrenzen überschritten werden und der String stets korrekt terminiert ist.
- Wird im obigen Beispiel `append()` mehr als 19 mal gerufen, stürzt das Programm ab. Fehlende Sicherungen gegen Pufferüberschreitungen sind eine Hauptfehlerquelle in C-Programmen.

Strings als Funktionsparameter

Hochschule RheinMain

- Strings können (wie Arrays) in Funktionen verändert werden. Zum Beispiel können Zeichen an einen String angehängt werden:

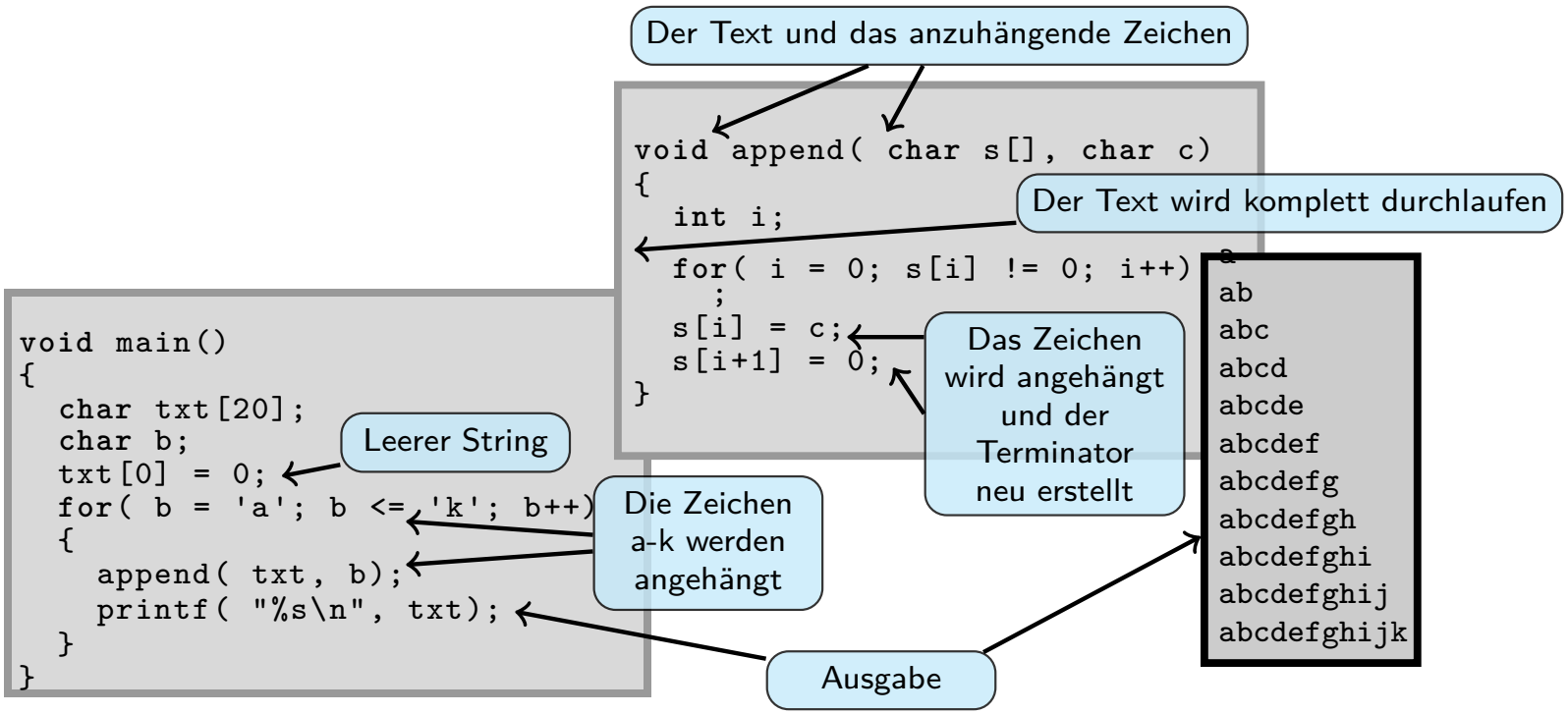


- Vorsicht bei Veränderungen von Strings im Unterprogramm.
- Programmierende sind dafür verantwortlich, dass keine Arraygrenzen überschritten werden und der String stets korrekt terminiert ist.
- Wird im obigen Beispiel `append()` mehr als 19 mal gerufen, stürzt das Programm ab. Fehlende Sicherungen gegen Pufferüberschreitungen sind eine Hauptfehlerquelle in C-Programmen.

Strings als Funktionsparameter

Hochschule RheinMain

- Strings können (wie Arrays) in Funktionen verändert werden. Zum Beispiel können Zeichen an einen String angehängt werden:



- Vorsicht bei Veränderungen von Strings im Unterprogramm.
- Programmierende sind dafür verantwortlich, dass keine Arraygrenzen überschritten werden und der String stets korrekt terminiert ist.
- Wird im obigen Beispiel `append()` mehr als 19 mal gerufen, stürzt das Programm ab. Fehlende Sicherungen gegen Pufferüberschreitungen sind eine Hauptfehlerquelle in C-Programmen.

Stringmanipulationen 2

- Um Pufferüberschreitungen abfangen zu können, sollte man Funktionen, die Strings verändern, die Puffergröße übergeben:

```
int append( int size, char s[], char c)
{
  int i;

  for( i = 0; s[i] != 0; i++)
    ;
  if( i >= size - 1)
    return 0;
  s[i] = c;
  s[i+1] = 0;
  return 1;
}
```

```
void main()
{
  char txt[20];
  char b;
  txt[0] = 0;
  for( b = 'a'; b <= 'k'; b++)
  {
    if( append( 20, txt, b))
      printf( "%s\n", txt);
  }
}
```

- Nach wie vor kann append() nur 19 mal mit Erfolg gerufen werden, aber das Programm stürzt nicht mehr ab und meldet zurück, wenn der String nicht mehr vergrößert werden kann.

Stringmanipulationen 2

Hochschule RheinMain

- Um Pufferüberschreitungen abfangen zu können, sollte man Funktionen, die Strings verändern, die Puffergröße übergeben:

```
int append( int size, char s[], char c)
{
  int i;

  for( i = 0; s[i] != 0; i++)
    ;
  if( i >= size - 1)
    return 0;
  s[i] = c;
  s[i+1] = 0;
  return 1;
}
```

```
void main()
{
  char txt[20];
  char b;
  txt[0] = 0;
  for( b = 'a'; b <= 'k'; b++)
  {
    if( append( 20,txt, b))
      printf( "%s\n", txt);
  }
}
```

Das Hauptprogramm übergibt beim Aufruf die Puffergröße.

- Nach wie vor kann append() nur 19 mal mit Erfolg gerufen werden, aber das Programm stürzt nicht mehr ab und meldet zurück, wenn der String nicht mehr vergrößert werden kann.

Stringmanipulationen 2

- Um Pufferüberschreitungen abfangen zu können, sollte man Funktionen, die Strings verändern, die Puffergröße übergeben:

Hier wird zusätzlich die Puffergröße übergeben.

```
int append( int size, char s[], char c)
{
  int i;

  for( i = 0; s[i] != 0; i++)
    ;
  if( i >= size - 1)
    return 0;
  s[i] = c;
  s[i+1] = 0;
  return 1;
}
```

```
void main()
{
  char txt[20];
  char b;
  txt[0] = 0;
  for( b = 'a'; b <= 'k'; b++)
  {
    if( append( 20,txt, b))
      printf( "%s\n", txt);
  }
}
```

Das Hauptprogramm übergibt beim Aufruf die Puffergröße.

- Nach wie vor kann append() nur 19 mal mit Erfolg gerufen werden, aber das Programm stürzt nicht mehr ab und meldet zurück, wenn der String nicht mehr vergrößert werden kann.

Stringmanipulationen 2

- Um Pufferüberschreitungen abfangen zu können, sollte man Funktionen, die Strings verändern, die Puffergröße übergeben:

Hier wird zusätzlich die Puffergröße übergeben.

```
int append( int size, char s[], char c)
{
  int i;

  for( i = 0; s[i] != 0; i++)
    ;
  if( i >= size - 1)
    return 0;
  s[i] = c;
  s[i+1] = 0;
  return 1;
}
```

Append kann nicht ausgeführt werden, da der Puffer zu klein ist.

```
void main()
{
  char txt[20];
  char b;
  txt[0] = 0;
  for( b = 'a'; b <= 'k'; b++)
  {
    if( append( 20,txt, b))
      printf( "%s\n", txt);
  }
}
```

Das Hauptprogramm übergibt beim Aufruf die Puffergröße.

- Nach wie vor kann append() nur 19 mal mit Erfolg gerufen werden, aber das Programm stürzt nicht mehr ab und meldet zurück, wenn der String nicht mehr vergrößert werden kann.

Bruchrechnung

- Im Folgenden werden Programmteile zum addieren und kürzen von Brüchen vorgestellt
- Hier zunächst eine Hilfsfunktion zur Berechnung des größten gemeinsamen Teilers zweier Zahlen:
- Den größten gemeinsamen Teiler (ggT) von zwei Zahlen erhält man, indem man solange die kleinere Zahl von der größeren abzieht bis beide Zahlen gleich sind.

```
int ggt( int a, int b)
{
  for( ; a != b; )
  {
    if( a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

Bruchrechnung

Hochschule RheinMain

- Im Folgenden werden Programmteile zum addieren und kürzen von Brüchen vorgestellt
- Hier zunächst eine Hilfsfunktion zur Berechnung des größten gemeinsamen Teilers zweier Zahlen:
- Den größten gemeinsamen Teiler (ggT) von zwei Zahlen erhält man, indem man solange die kleinere Zahl von der größeren abzieht bis beide Zahlen gleich sind.

Der ggT von a und b soll berechnet werden.

```
int ggt( int a, int b)
{
  for( ; a != b; )
  {
    if( a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

Bruchrechnung

Hochschule RheinMain

- Im Folgenden werden Programmteile zum addieren und kürzen von Brüchen vorgestellt
- Hier zunächst eine Hilfsfunktion zur Berechnung des größten gemeinsamen Teilers zweier Zahlen:
- Den größten gemeinsamen Teiler (ggT) von zwei Zahlen erhält man, indem man solange die kleinere Zahl von der größeren abzieht bis beide Zahlen gleich sind.

Der ggT von a und b soll berechnet werden.

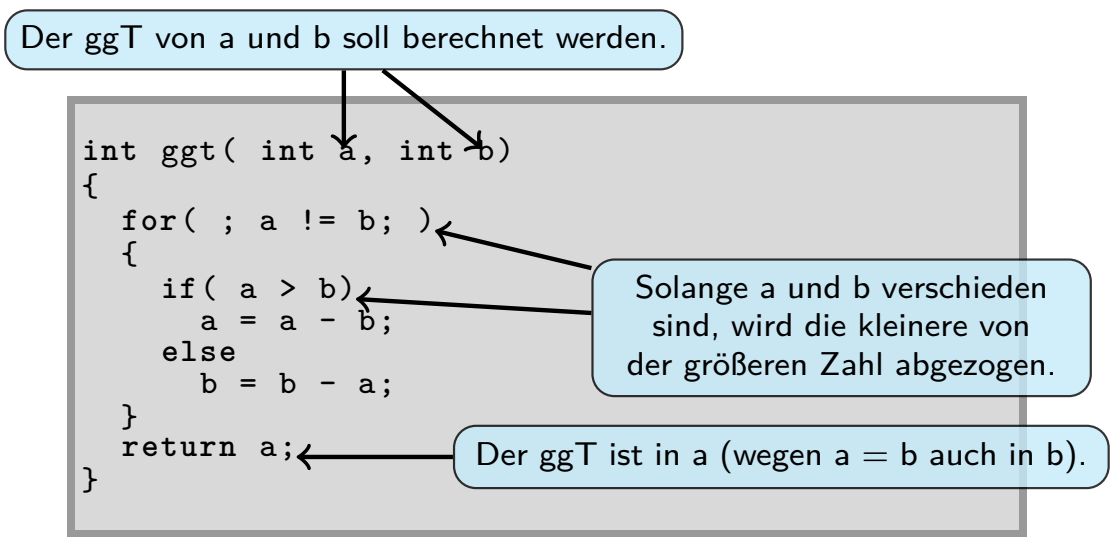
```
int ggt( int a, int b)
{
  for( ; a != b; )
  {
    if( a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

Solange a und b verschieden sind, wird die kleinere von der größeren Zahl abgezogen.

Bruchrechnung

Hochschule RheinMain

- Im Folgenden werden Programmteile zum addieren und kürzen von Brüchen vorgestellt
- Hier zunächst eine Hilfsfunktion zur Berechnung des größten gemeinsamen Teilers zweier Zahlen:
- Den größten gemeinsamen Teiler (ggT) von zwei Zahlen erhält man, indem man solange die kleinere Zahl von der größeren abzieht bis beide Zahlen gleich sind.



Bruchrechnung (2)

- Einen Bruch speichern wir mit Zähler und Nenner in einem Array mit zwei Elementen:

```
int bruch[2]; // bruch[0] ist der Zähler, bruch[1] ist der Nenner
```

- Brüche werden gekürzt, indem Zähler und Nenner durch ihren ggT geteilt werden:

```
void kuerzen( int b[])
{
  int gt;
  gt = ggt( b[0], b[1]);
  b[0] = b[0]/gt;
  b[1] = b[1]/gt;
}
```

- Brüche werden nach der Formel $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ addiert, wobei anschließend gekürzt werden sollte:

```
void addieren( int b1[], int b2[], int erg[])
{
  erg[0] = b1[0]*b2[1] + b2[0]*b1[1];
  erg[1] = b1[1]*b2[1];
  kuerzen( erg);
}
```

Bruchrechnung (2)

Hochschule RheinMain

- Einen Bruch speichern wir mit Zähler und Nenner in einem Array mit zwei Elementen:

```
int bruch[2]; // bruch[0] ist der Zähler, bruch[1] ist der Nenner
```

- Brüche werden gekürzt, indem Zähler und Nenner durch ihren ggT geteilt werden:

Der Bruch kommt als Array, Der Zähler hat den Index 0, der Nenner den Index 1.

```
void kuerzen( int b[])
{
  int gt;
  gt = ggt( b[0], b[1]);
  b[0] = b[0]/gt;
  b[1] = b[1]/gt;
}
```

- Brüche werden nach der Formel $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ addiert, wobei anschließend gekürzt werden sollte:

```
void addieren( int b1[], int b2[], int erg[])
{
  erg[0] = b1[0]*b2[1] + b2[0]*b1[1];
  erg[1] = b1[1]*b2[1];
  kuerzen( erg);
}
```

Bruchrechnung (2)

Hochschule RheinMain

- Einen Bruch speichern wir mit Zähler und Nenner in einem Array mit zwei Elementen:

```
int bruch[2]; // bruch[0] ist der Zähler, bruch[1] ist der Nenner
```

- Brüche werden gekürzt, indem Zähler und Nenner durch ihren ggT geteilt werden:

Der Bruch kommt als Array, Der Zähler hat den Index 0, der Nenner den Index 1.

```
void kuerzen( int b[])
{
  int gt;
  gt = ggt( b[0], b[1]);
  b[0] = b[0]/gt;
  b[1] = b[1]/gt;
}
```

Der ggT von Zähler und Nenner wird berechnet.

- Brüche werden nach der Formel $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ addiert, wobei anschließend gekürzt werden sollte:

```
void addieren( int b1[], int b2[], int erg[])
{
  erg[0] = b1[0]*b2[1] + b2[0]*b1[1];
  erg[1] = b1[1]*b2[1];
  kuerzen( erg);
}
```

Bruchrechnung (2)

Hochschule RheinMain

- Einen Bruch speichern wir mit Zähler und Nenner in einem Array mit zwei Elementen:

```
int bruch[2]; // bruch[0] ist der Zähler, bruch[1] ist der Nenner
```

- Brüche werden gekürzt, indem Zähler und Nenner durch ihren ggT geteilt werden:

Der Bruch kommt als Array, Der Zähler hat den Index 0, der Nenner den Index 1.

```
void kuerzen( int b[])
{
  int gt;
  gt = ggt( b[0], b[1]);
  b[0] = b[0]/gt;
  b[1] = b[1]/gt;
}
```

Der ggT von Zähler und Nenner wird berechnet.

Zähler und Nenner werden durch den ggT dividiert.

- Brüche werden nach der Formel $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ addiert, wobei anschließend gekürzt werden sollte:

```
void addieren( int b1[], int b2[], int erg[])
{
  erg[0] = b1[0]*b2[1] + b2[0]*b1[1];
  erg[1] = b1[1]*b2[1];
  kuerzen( erg);
}
```

Bruchrechnung (2)

Hochschule RheinMain

- Einen Bruch speichern wir mit Zähler und Nenner in einem Array mit zwei Elementen:

```
int bruch[2]; // bruch[0] ist der Zähler, bruch[1] ist der Nenner
```

- Brüche werden gekürzt, indem Zähler und Nenner durch ihren ggT geteilt werden:

Der Bruch kommt als Array, Der Zähler hat den Index 0, der Nenner den Index 1.

```
void kuerzen( int b[])
{
  int gt;
  gt = ggt( b[0], b[1]);
  b[0] = b[0]/gt;
  b[1] = b[1]/gt;
}
```

Der ggT von Zähler und Nenner wird berechnet.

Zähler und Nenner werden durch den ggT dividiert.

- Brüche werden nach der Formel $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ addiert, wobei anschließend gekürzt werden sollte:

Diese beiden Brüche sind zu addieren.

```
void addieren( int b1[], int b2[], int erg[])
{
  erg[0] = b1[0]*b2[1] + b2[0]*b1[1];
  erg[1] = b1[1]*b2[1];
  kuerzen( erg);
}
```

Bruchrechnung (2)

Hochschule RheinMain

- Einen Bruch speichern wir mit Zähler und Nenner in einem Array mit zwei Elementen:

```
int bruch[2]; // bruch[0] ist der Zähler, bruch[1] ist der Nenner
```

- Brüche werden gekürzt, indem Zähler und Nenner durch ihren ggT geteilt werden:

Der Bruch kommt als Array, Der Zähler hat den Index 0, der Nenner den Index 1.

```
void kuerzen( int b[])
{
  int gt;
  gt = ggt( b[0], b[1]);
  b[0] = b[0]/gt;
  b[1] = b[1]/gt;
}
```

Der ggT von Zähler und Nenner wird berechnet.

Zähler und Nenner werden durch den ggT dividiert.

- Brüche werden nach der Formel $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ addiert, wobei anschließend gekürzt werden sollte:

Diese beiden Brüche sind zu addieren.

```
void addieren( int b1[], int b2[], int erg[])
{
  erg[0] = b1[0]*b2[1] + b2[0]*b1[1];
  erg[1] = b1[1]*b2[1];
  kuerzen( erg);
}
```

$\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$

Bruchrechnung (2)

Hochschule RheinMain

- Einen Bruch speichern wir mit Zähler und Nenner in einem Array mit zwei Elementen:

```
int bruch[2]; // bruch[0] ist der Zähler, bruch[1] ist der Nenner
```

- Brüche werden gekürzt, indem Zähler und Nenner durch ihren ggT geteilt werden:

Der Bruch kommt als Array, Der Zähler hat den Index 0, der Nenner den Index 1.

```
void kuerzen( int b[])
{
  int gt;
  gt = ggt( b[0], b[1]);
  b[0] = b[0]/gt;
  b[1] = b[1]/gt;
}
```

Der ggT von Zähler und Nenner wird berechnet.

Zähler und Nenner werden durch den ggT dividiert.

- Brüche werden nach der Formel $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ addiert, wobei anschließend gekürzt werden sollte:

Diese beiden Brüche sind zu addieren.

```
void addieren( int b1[], int b2[], int erg[])
{
  erg[0] = b1[0]*b2[1] + b2[0]*b1[1];
  erg[1] = b1[1]*b2[1];
  kuerzen( erg);
}
```

$\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$

Das Ergebnis wird gekürzt

Bruchrechnung (3)

Hochschule RheinMain

- Mit den Funktionen `kuerzen` und `addieren` kann ein kleines Programm zur Bruchrechnung geschrieben werden:

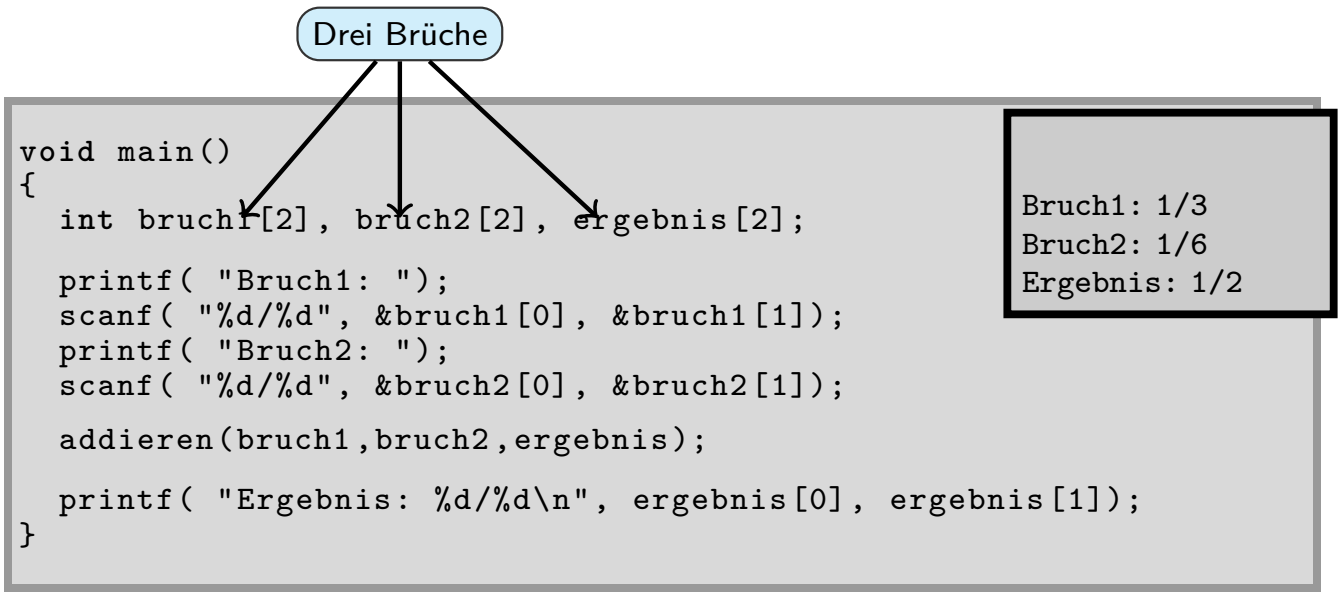
```
void main()
{
  int bruch1[2], bruch2[2], ergebnis[2];

  printf( "Bruch1: ");
  scanf( "%d/%d", &bruch1[0], &bruch1[1]);
  printf( "Bruch2: ");
  scanf( "%d/%d", &bruch2[0], &bruch2[1]);

  addieren(bruch1,bruch2,ergebnis);

  printf( "Ergebnis: %d/%d\n", ergebnis[0], ergebnis[1]);
}
```

```
Bruch1: 1/3
Bruch2: 1/6
Ergebnis: 1/2
```

Bruchrechnung (3)

Hochschule RheinMain

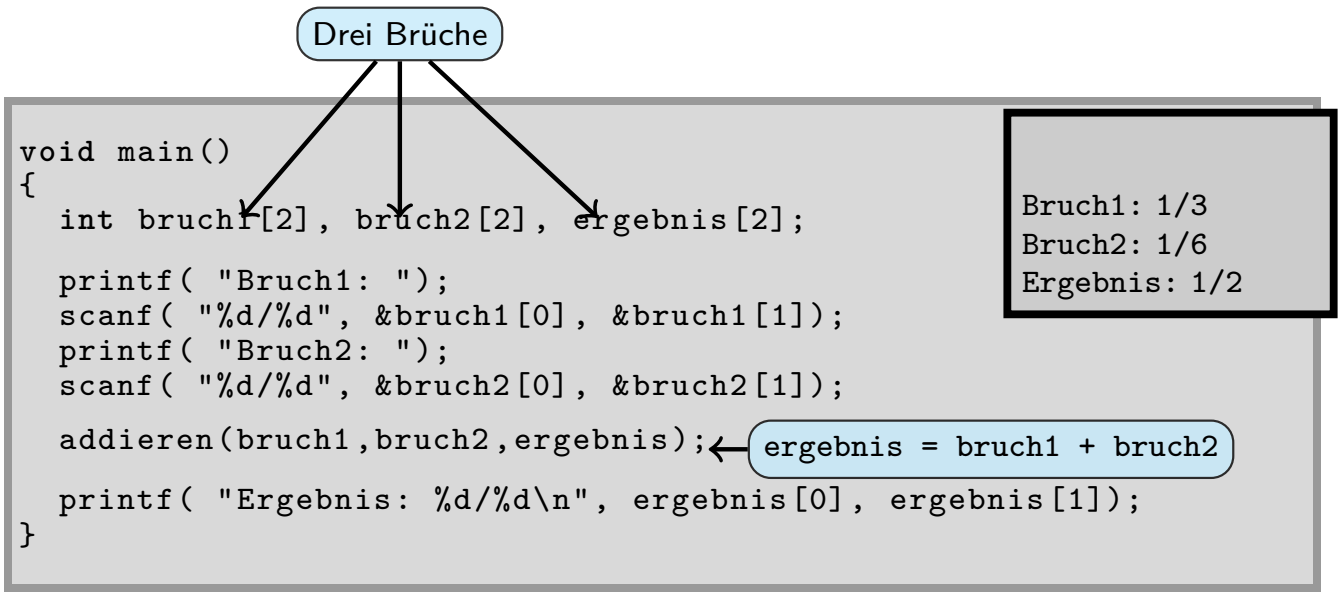
- Mit den Funktionen `kuerzen` und `addieren` kann ein kleines Programm zur Bruchrechnung geschrieben werden:



Bruchrechnung (3)

Hochschule RheinMain

- Mit den Funktionen `kuerzen` und `addieren` kann ein kleines Programm zur Bruchrechnung geschrieben werden:



Beispiel: Fakultätsfunktion

Hochschule RheinMain

- Am Beispiel der Fakultätsfunktion wollen wir eines der wichtigsten Programmierprinzipien kennenlernen. Dazu implementieren wir diese Funktion zunächst in naheliegender Weise.
- Die Fakultät $n! = 1 \cdot 2 \cdot 3 \ldots n$ kann **iterativ** in einer Schleife berechnet werden:

```
int fakultaet_iter( int n)
{
  int fak;
  for( fak = 1; n > 1; n--)
    fak = fak * n;
  return fak;
}

void main()
{
  int n;
  for( n = 0; n < 10; n++)
    printf( "%d! = %d\n", n, fakultaet_iter( n));
}
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 326880
```

Beispiel: Fakultätsfunktion

Hochschule RheinMain

- Am Beispiel der Fakultätsfunktion wollen wir eines der wichtigsten Programmierprinzipien kennenlernen. Dazu implementieren wir diese Funktion zunächst in naheliegender Weise.
- Die Fakultät $n! = 1 \cdot 2 \cdot 3 \ldots n$ kann **iterativ** in einer Schleife berechnet werden:

Der Parameter n wird immer mit fak multipliziert und dabei heruntergezählt.

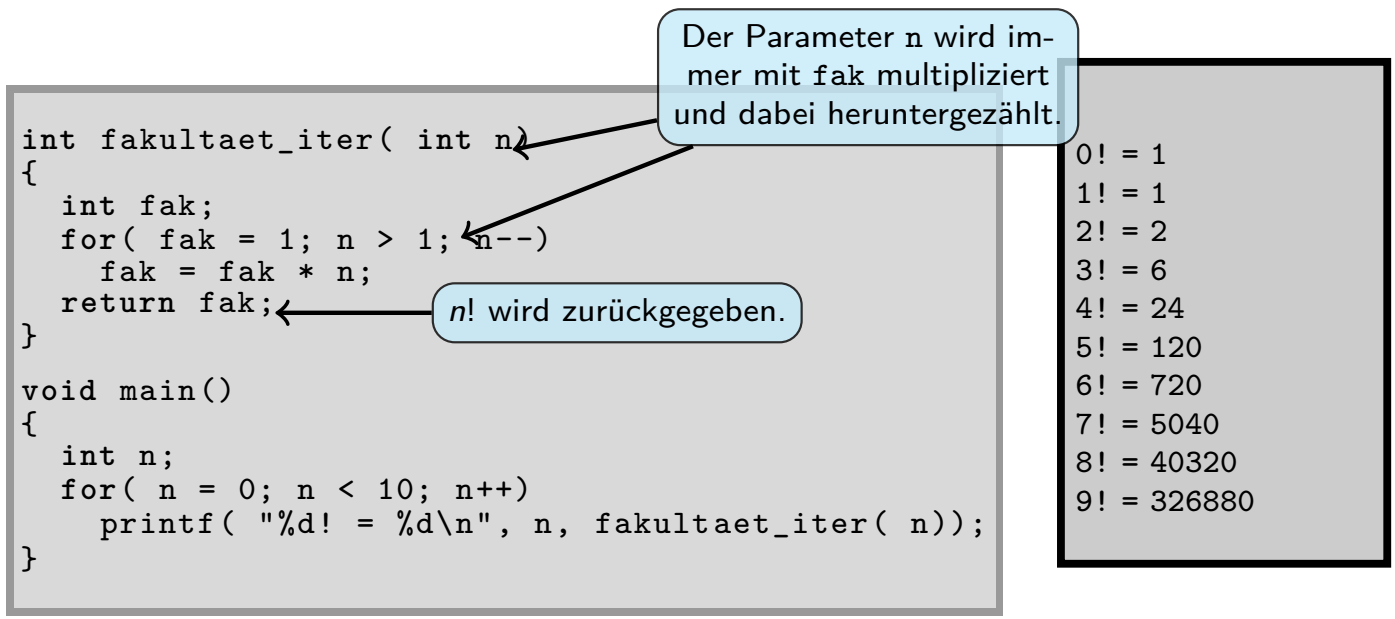
```
int fakultaet_iter( int n)
{
  int fak;
  for( fak = 1; n > 1; n--)
    fak = fak * n;
  return fak;
}

void main()
{
  int n;
  for( n = 0; n < 10; n++)
    printf( "%d! = %d\n", n, fakultaet_iter( n));
}
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 326880
```

Beispiel: Fakultätsfunktion

Hochschule RheinMain

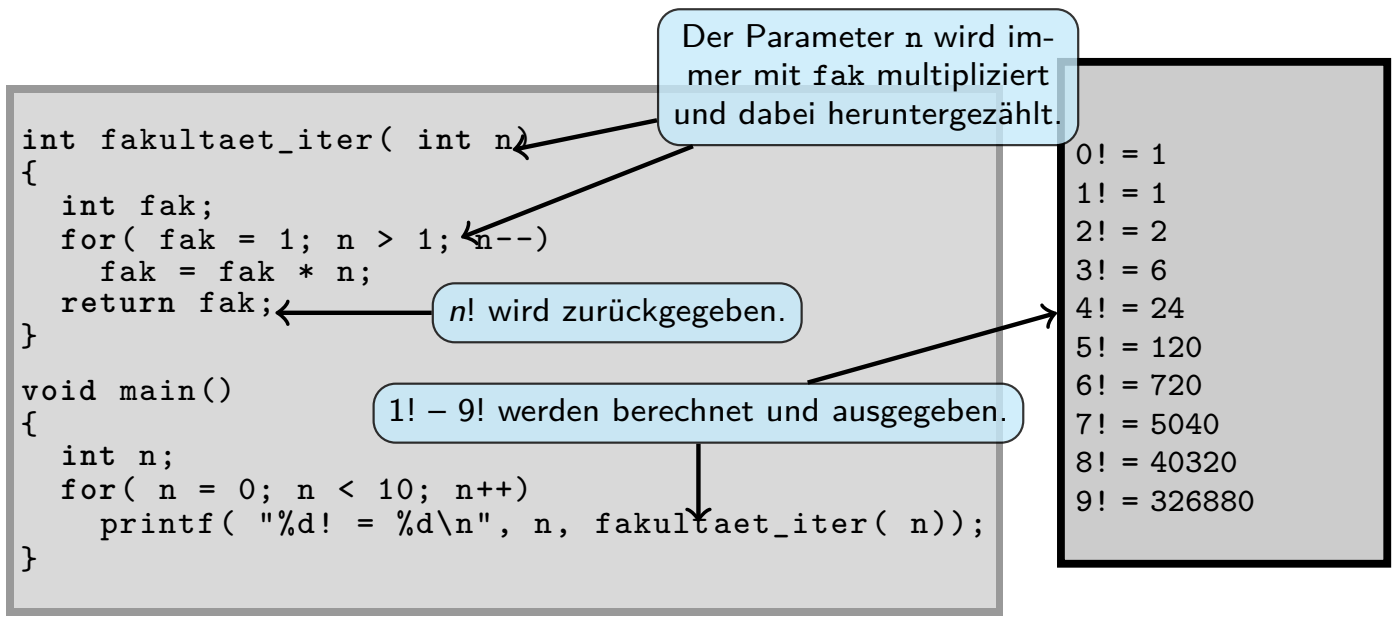
- Am Beispiel der Fakultätsfunktion wollen wir eines der wichtigsten Programmierprinzipien kennenlernen. Dazu implementieren wir diese Funktion zunächst in naheliegender Weise.
- Die Fakultät $n! = 1 \cdot 2 \cdot 3 \ldots n$ kann **iterativ** in einer Schleife berechnet werden:



Beispiel: Fakultätsfunktion

Hochschule RheinMain

- Am Beispiel der Fakultätsfunktion wollen wir eines der wichtigsten Programmierprinzipien kennenlernen. Dazu implementieren wir diese Funktion zunächst in naheliegender Weise.
- Die Fakultät $n! = 1 \cdot 2 \cdot 3 \ldots n$ kann **iterativ** in einer Schleife berechnet werden:



Rekursion

Hochschule RheinMain

- Funktionen können sich selbst unmittelbar oder mittelbar (über eine andere Funktion) aufrufen. Man nennt dies **Rekursion**. Rekursion ist eine einfache und elegante Lösungsstrategie, wenn man ein Problem auf ein oder mehrere strukturell gleiche, aber „kleinere“ Probleme zurückführen kann, bis man am Ende auf ein leicht zu lösendes Problem stößt.
- Rekursive Definition der Fakultätsfunktion:

$$n! = \begin{cases} 1, & \text{falls } n \leq= 1 \\ n \cdot (n-1)!, & \text{falls } n > 1 \end{cases}$$

- Die Definition führt unmittelbar zu einer rekursiven Implementierung der Fakultätsfunktion:

```
int fak(int n)
{
  if( n <= 1)
    return 1;
  return n * fak(n-1);
}
```

- An der Schnittstelle ist diese Funktion identisch mit der nicht rekursiven Variante und kann daher exakt gleich verwendet werden:

```
void main()
{
  int n;
  for( n = 0; n < 10; n++)
    printf( "%d! = %d\n", n, fak( n));
}
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
...
```

Rekursion

Hochschule RheinMain

- Funktionen können sich selbst unmittelbar oder mittelbar (über eine andere Funktion) aufrufen. Man nennt dies **Rekursion**. Rekursion ist eine einfache und elegante Lösungsstrategie, wenn man ein Problem auf ein oder mehrere strukturell gleiche, aber „kleinere“ Probleme zurückführen kann, bis man am Ende auf ein leicht zu lösendes Problem stößt.
- Rekursive Definition der Fakultätsfunktion:

$$n! = \begin{cases} 1, & \text{falls } n \leq= 1 \\ n \cdot (n-1)!, & \text{falls } n > 1 \end{cases}$$

In dieser Definition führt man die Berechnung von $n!$ auf die Berechnung der kleineren Fakultät $(n-1)!$ zurück, bis man am Ende auf ein leicht zu lösendes Problem (Berechnung von 1!) stößt.

- Die Definition führt unmittelbar zu einer rekursiven Implementierung der Fakultätsfunktion:

```
int fak(int n)
{
  if( n <= 1)
    return 1;
  return n * fak(n-1);
}
```

- An der Schnittstelle ist diese Funktion identisch mit der nicht rekursiven Variante und kann daher exakt gleich verwendet werden:

```
void main()
{
  int n;
  for( n = 0; n < 10; n++)
    printf( "%d! = %d\n", n, fak( n));
}
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
...
```

Rekursion

Hochschule RheinMain

- Funktionen können sich selbst unmittelbar oder mittelbar (über eine andere Funktion) aufrufen. Man nennt dies **Rekursion**. Rekursion ist eine einfache und elegante Lösungsstrategie, wenn man ein Problem auf ein oder mehrere strukturell gleiche, aber „kleinere“ Probleme zurückführen kann, bis man am Ende auf ein leicht zu lösendes Problem stößt.
- Rekursive Definition der Fakultätsfunktion:

$$n! = \begin{cases} 1, & \text{falls } n <= 1 \\ n \cdot (n-1)!, & \text{falls } n > 1 \end{cases}$$

In dieser Definition führt man die Berechnung von $n!$ auf die Berechnung der kleineren Fakultät $(n-1)!$ zurück, bis man am Ende auf ein leicht zu lösendes Problem (Berechnung von 1!) stößt.

- Die Definition führt unmittelbar zu einer rekursiven Implementierung der Fakultätsfunktion:

```
int fak(int n)
{
  if( n <= 1)
    return 1;
  return n * fak(n-1);
}
```

$n! = 1 \text{falls } n \leq 1$

- An der Schnittstelle ist diese Funktion identisch mit der nicht rekursiven Variante und kann daher exakt gleich verwendet werden:

```
void main()
{
  int n;
  for( n = 0; n < 10; n++)
    printf( "%d! = %d\n", n, fak( n));
}
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
...
```

Rekursion

Hochschule RheinMain

- Funktionen können sich selbst unmittelbar oder mittelbar (über eine andere Funktion) aufrufen. Man nennt dies **Rekursion**. Rekursion ist eine einfache und elegante Lösungsstrategie, wenn man ein Problem auf ein oder mehrere strukturell gleiche, aber „kleinere“ Probleme zurückführen kann, bis man am Ende auf ein leicht zu lösendes Problem stößt.
- Rekursive Definition der Fakultätsfunktion:

$$n! = \begin{cases} 1, & \text{falls } n <= 1 \\ n \cdot (n-1)!, & \text{falls } n > 1 \end{cases}$$

In dieser Definition führt man die Berechnung von $n!$ auf die Berechnung der kleineren Fakultät $(n-1)!$ zurück, bis man am Ende auf ein leicht zu lösendes Problem (Berechnung von 1!) stößt.

- Die Definition führt unmittelbar zu einer rekursiven Implementierung der Fakultätsfunktion:

```
int fak(int n)
{
  if( n <= 1)
    return 1;
  return n * fak(n-1);
}
```

$n! = 1$ falls $n \leq 1$

$n! = n \cdot (n-1)$ falls $n > 1$

- An der Schnittstelle ist diese Funktion identisch mit der nicht rekursiven Variante und kann daher exakt gleich verwendet werden:

```
void main()
{
  int n;
  for( n = 0; n < 10; n++)
    printf( "%d! = %d\n", n, fak( n));
}
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
...
```

Rekursion

Hochschule RheinMain

- Funktionen können sich selbst unmittelbar oder mittelbar (über eine andere Funktion) aufrufen. Man nennt dies **Rekursion**. Rekursion ist eine einfache und elegante Lösungsstrategie, wenn man ein Problem auf ein oder mehrere strukturell gleiche, aber „kleinere“ Probleme zurückführen kann, bis man am Ende auf ein leicht zu lösendes Problem stößt.
- Rekursive Definition der Fakultätsfunktion:

$$n! = \begin{cases} 1, & \text{falls } n <= 1 \\ n \cdot (n-1)!, & \text{falls } n > 1 \end{cases}$$

In dieser Definition führt man die Berechnung von $n!$ auf die Berechnung der kleineren Fakultät $(n-1)!$ zurück, bis man am Ende auf ein leicht zu lösendes Problem (Berechnung von 1!) stößt.

- Die Definition führt unmittelbar zu einer rekursiven Implementierung der Fakultätsfunktion:

```
int fak(int n)
{
  if( n <= 1)
    return 1;
  return n * fak(n-1);
}
```

$n! = 1$ falls $n \leq 1$

$n! = n \cdot (n-1)$ falls $n > 1$

Selbstaufruf (unmittelbare Rekursion)

- An der Schnittstelle ist diese Funktion identisch mit der nicht rekursiven Variante und kann daher exakt gleich verwendet werden:

```
void main()
{
  int n;
  for( n = 0; n < 10; n++)
    printf( "%d! = %d\n", n, fak( n));
}
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
...
```

Rekursion aus Maschinensicht

Hochschule RheinMain

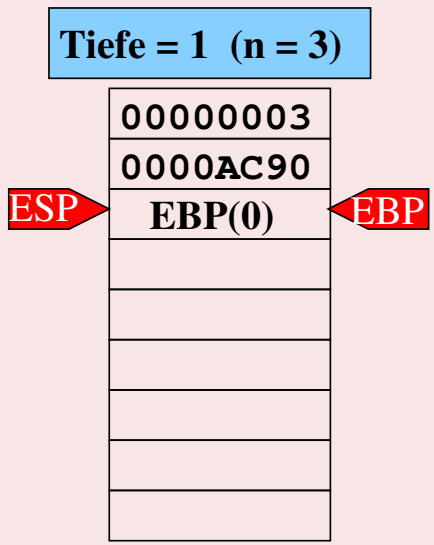
- Im Beispiel hier: Intel x86 (32-bit)

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit STOP markierten Zeile)

```
int fak(int n)
{
if(n <= 1)
  return 1;
 else
  return n * fak(n - 1);
}
.....
f3 = fak(3));
.....
```

```
0100 fak: pushl ebp
010      movl esp, ebp
         ..........
1208     call fak
120C     move eax,..
         ..........
135E     ret
         ..........
AC80     push #3
AC8E     call fak
AC90     move eax,..
```



Rekursion aus Maschinensicht

Hochschule RheinMain

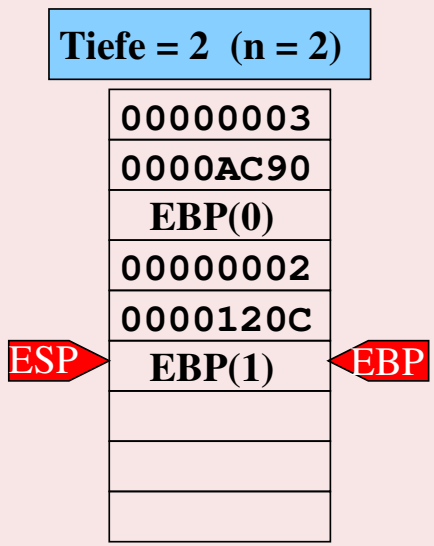
- Im Beispiel hier: Intel x86 (32-bit)

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit STOP markierten Zeile)

```
int fak(int n)
{
if(n <= 1)
  return 1;
 else
  return n * fak(n - 1);
}
.....
f3 = fak(3));
.....
```

```
0100 fak: pushl ebp
010  movl esp, ebp
     ..........
1208      call fak
120C      move eax,..
     ..........
135E      ret
     ..........
AC80      push #3
AC8E      call fak
AC90      move eax,..
```



Rekursion aus Maschinensicht

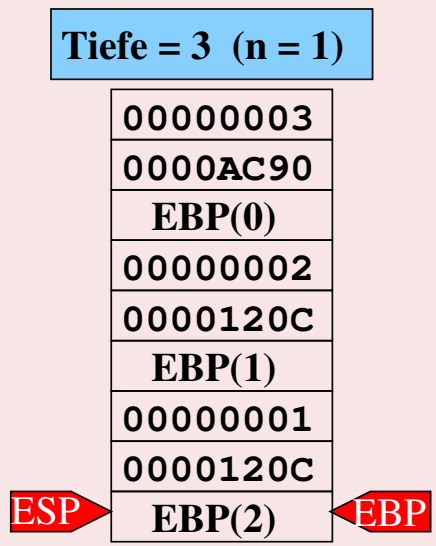
- Im Beispiel hier: Intel x86 (32-bit)

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit STOP markierten Zeile)

```
int fak(int n)
{
STOP if(n <= 1)
   return 1;
 else
   return n * fak(n - 1);
}
.....
f3 = fak(3));
.....
```

```
0100 fak: pushl ebp
010STOP     movl esp, ebp
          ..........
1208      call fak
120C      move eax,..
          ..........
135E      ret
          ..........
AC80      push #3
AC8E      call fak
AC90      move eax,..
```



Rekursion aus Maschinensicht

Hochschule RheinMain

- Im Beispiel hier: Intel x86 (32-bit)

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit STOP markierten Zeile)

```
int fak(int n)
{
 if(n <= 1)
 return 1;
 else
  return n * fak(n - 1);
}
.....
f3 = fak(3));
.....
```

```
0100 fak: pushl ebp
0104      movl esp, ebp
          ..........
1208      call fak
120C      move eax,..
          ..........
135       ret
          ..........
AC80      push #3
AC8E      call fak
AC90      move eax,..
```

Rekursion aus Maschinensicht

Hochschule RheinMain

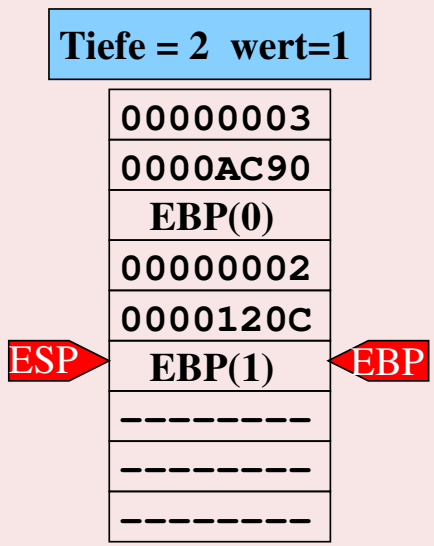
- Im Beispiel hier: Intel x86 (32-bit)

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit STOP markierten Zeile)

```
int fak(int n)
{
 if(n <= 1)
  return 1;
 else
  return n * fak(n - 1);
}
.....
f3 = fak(3));
.....
```

```
0100 fak: pushl ebp
0104      movl esp, ebp
          ..........
1208      call fak
120C      move eax,..
          ..........
135       ret
          ..........
AC80      push #3
AC8E      call fak
AC90      move eax,..
```



Rekursion aus Maschinensicht

Hochschule RheinMain

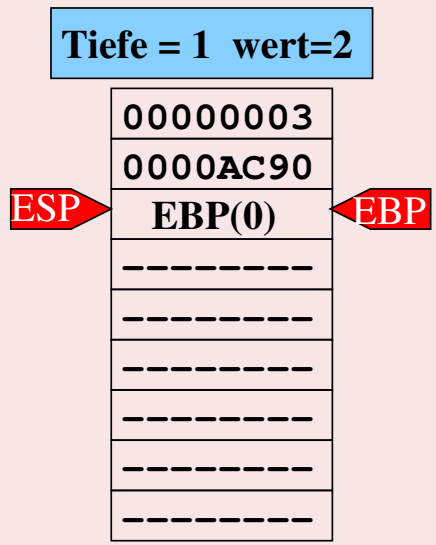
- Im Beispiel hier: Intel x86 (32-bit)

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit STOP markierten Zeile)

```
int fak(int n)
{
 if(n <= 1)
  return 1;
 else
STOP return n * fak(n - 1);
}
.....
f3 = fak(3));
.....
```

```
0100 fak: pushl ebp
0104      movl esp, ebp
          ..........
1208      call fak
120C      move eax,..
          ..........
135 STOP  ret
          ..........
AC80      push #3
AC8E      call fak
AC90      move eax,..
```



Rekursion aus Maschinensicht

Hochschule RheinMain

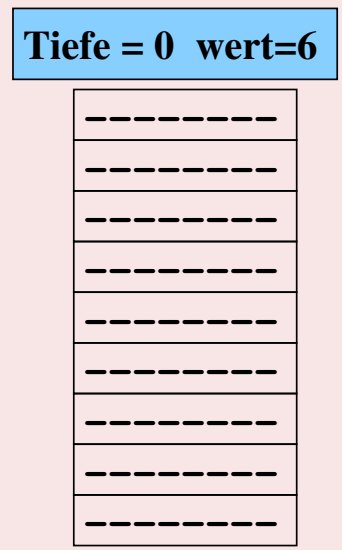
- Im Beispiel hier: Intel x86 (32-bit)

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit STOP markierten Zeile)

```
int fak(int n)
{
 if(n <= 1)
  return 1;
 else
  return n * fak(n - 1);
}

.....
f3 = fak(3));
STOP....
```

```
0100 fak: pushl ebp
0104      movl esp, ebp
          ..........
1208      call fak
120C      move eax,..
          ..........
135E      ret
          ..........
AC80      push #3
AC8E      call fak
AC9STOP   move eax,..
```



Lokale Variablen

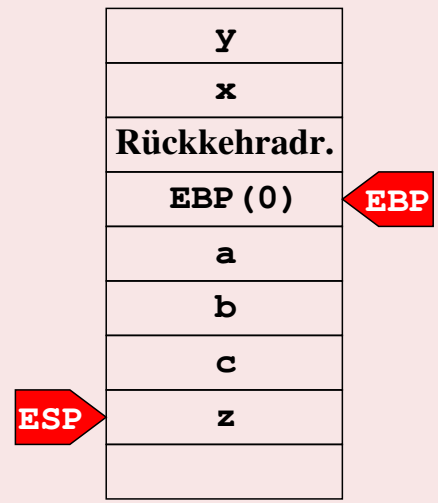
Hochschule RheinMain

- Im vorigen Beispiel gab es außer den Funktionsparametern keine lokalen Variablen
- Beobachtung: Stackpointer (ESP) und Framepointer (EBP) waren stets gleich (Wozu dann überhaupt ein Framepointer?)

Beispiel: Platz für lokale Variablen

```
fkt(int x, int y)
{
  int a, b, c;
  int z;
  a = x;
  b = y;
  c = x - y;
  z = 2 * (a + b) - c;
}
```

```
fkt: pushl ebp
     movl esp,ebp
     subl#16,esp
     movl 8(ebp),eax
     movl eax-,4(ebp)
     movl 12(ebp),eax
     movl eax-,8(ebp)
     movl 12(ebp),edx
     movl 8(ebp),eax
     subl edx,eax
     .....
     ret
```



→ Lokale Variablen liegen zwischen EBP und ESP.

- Jede Instanz der Funktion besitzt eigene lokale Variablen.
- Lokale Variablen liegen im Stack-Speicher (gemeinsam mit Rückkehradresse, geretteten Registerinhalten, etc.).

Lokale Variablen

Hochschule RheinMain

- Im vorigen Beispiel gab es außer den Funktionsparametern keine lokalen Variablen
- Beobachtung: Stackpointer (ESP) und Framepointer (EBP) waren stets gleich (Wozu dann überhaupt ein Framepointer?)

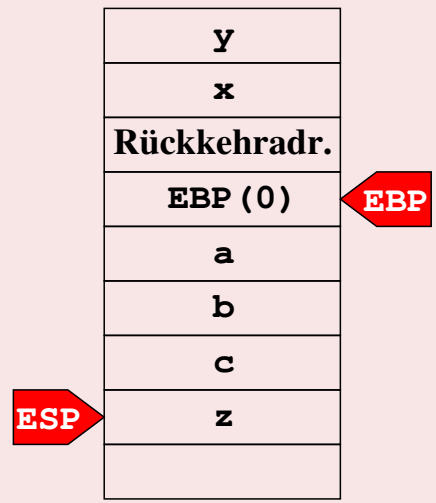
Beispiel: Platz für lokale Variablen

```
fkt(int x, int y)
{
  int a, b, c;
  int z;
  a = x;
  b = y;
  c = x - y;
  z = 2 * (a + b) - c;
}
```

4 int Variablen

```
fkt: pushl ebp
movl esp,ebp
subl#16,esp
movl 8(ebp),eax
movl eax-,4(ebp)
movl 12(ebp),eax
movl eax-,8(ebp)
movl 12(ebp),edx
movl 8(ebp),eax
subl edx,eax
.....
ret
```

4 * sizeof(int) = 16



→ Lokale Variablen liegen zwischen EBP und ESP.

- Jede Instanz der Funktion besitzt eigene lokale Variablen.
- Lokale Variablen liegen im Stack-Speicher (gemeinsam mit Rückkehradresse, geretteten Registerinhalten, etc.).

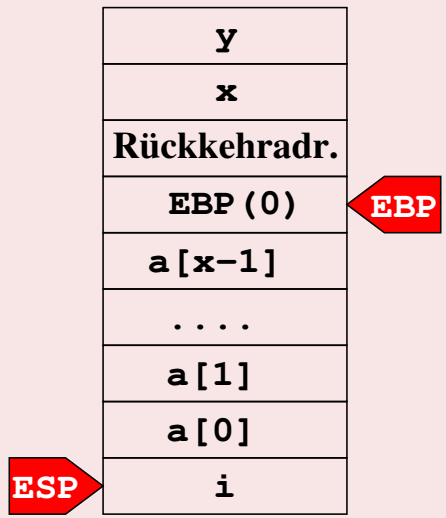
Dynamische lokale Variablen

Hochschule RheinMain

- Im vorigen Beispiel war die Gesamtgröße der lokalen Daten konstant, deshalb konnte zur Allokation ein fester Betrages zum Stackpointer addiert werden
- Seit dem C99 Standard gibt es jedoch auch lokale Felder *variabler* Größe:

Beispiel: variable Feldgröße

```
fkt(int x, int y)
{
  int a[x];
  int i;

  for(i = 0; i < x; i++)
    a[i] = ....;
  ....
}
```



→ Feldgröße kann von Instanz zu Instanz variieren

→ Adressabstand zwischen Framepointer und Variablen (hier z.B. `i`) ist *nicht* konstant.

Dynamische lokale Variablen

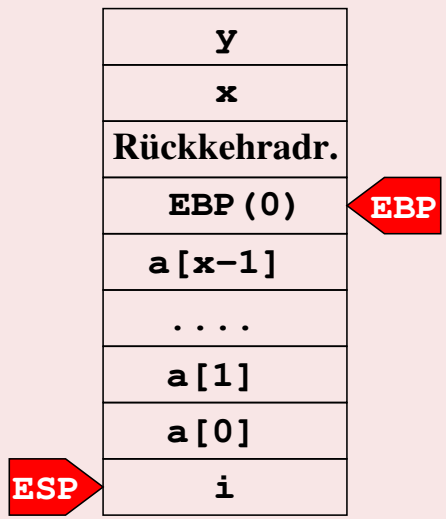
Hochschule RheinMain

- Im vorigen Beispiel war die Gesamtgröße der lokalen Daten konstant, deshalb konnte zur Allokation ein fester Betrages zum Stackpointer addiert werden
- Seit dem C99 Standard gibt es jedoch auch lokale Felder *variabler* Größe:

Beispiel: variable Feldgröße

Die Array-Größe wird durch den Wert der Variablen x festgelegt

```
fkt(int x, int y)
{
  int a[x];
  int i;

  for(i = 0; i < x; i++)
    a[i] = ....;
  ....
}
```



→ Feldgröße kann von Instanz zu Instanz variieren

→ Adressabstand zwischen Framepointer und Variablen (hier z.B. i) ist *nicht* konstant.

Pufferüberlauf-Attacke

Hochschule RheinMain

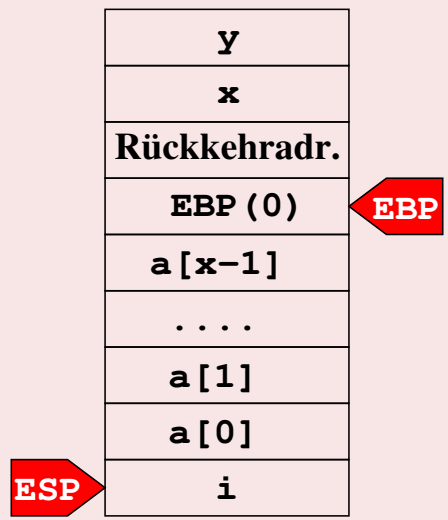
- Vorsicht: Indizes werden in C nicht zur Laufzeit geprüft
- Zum Beispiel kann über `a[x+1]` auf die Rückkehradresse der Funktion zugegriffen werden.

→ Mit `a[x+1] = Zieladresse` kann die Rückkehradresse verändert werden.

→ Statt zum Aufrufer zurückzukehrem, kann ein beliebiges Stück (womöglich in Form von Daten eingeschleusten) Schadcodes angesprungen werden.

Beispiel: Gefahr von Pufferüberlauf

```
fkt(int x, int y)
{
  int a[x];
  int i;

  for(i = 0; ; i++)
    a[i] = untrusted_data();
  ....
}
```



- Keine spezielle Eigenschaft dynamischer Arrays: Diese Angriffsmöglichkeit besteht in gleicher Weise auch bei Arrays fester Größe.

Pufferüberlauf-Attacke

Hochschule RheinMain

- Vorsicht: Indizes werden in C nicht zur Laufzeit geprüft
- Zum Beispiel kann über `a[x+1]` auf die Rückkehradresse der Funktion zugegriffen werden.

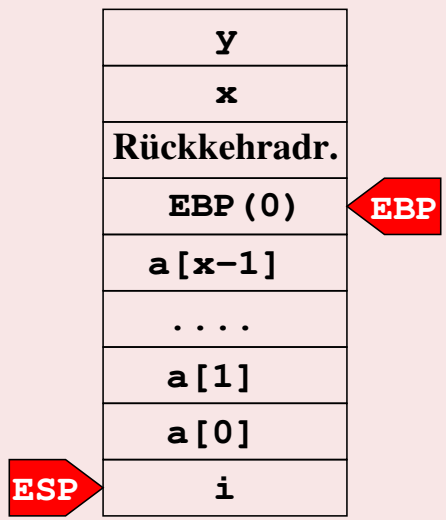
→ Mit `a[x+1] = Zieladresse` kann die Rückkehradresse verändert werden.

→ Statt zum Aufrufer zurückzukehrem, kann ein beliebiges Stück (womöglich in Form von Daten eingeschleusten) Schadcodes angesprungen werden.

Beispiel: Gefahr von Pufferüberlauf

```
fkt(int x, int y)
{
  int a[x];
  int i;

  for(i = 0; ; i++)
    a[i] = untrusted_data();
  ....
}
```

Fehler: Keine Begrenzung auf `i < x`



- Keine spezielle Eigenschaft dynamischer Arrays: Diese Angriffsmöglichkeit besteht in gleicher Weise auch bei Arrays fester Größe.

Globale Variablen

Hochschule RheinMain

- Variablen außerhalb von Funktionen sind global und können funktionsübergreifend genutzt werden:

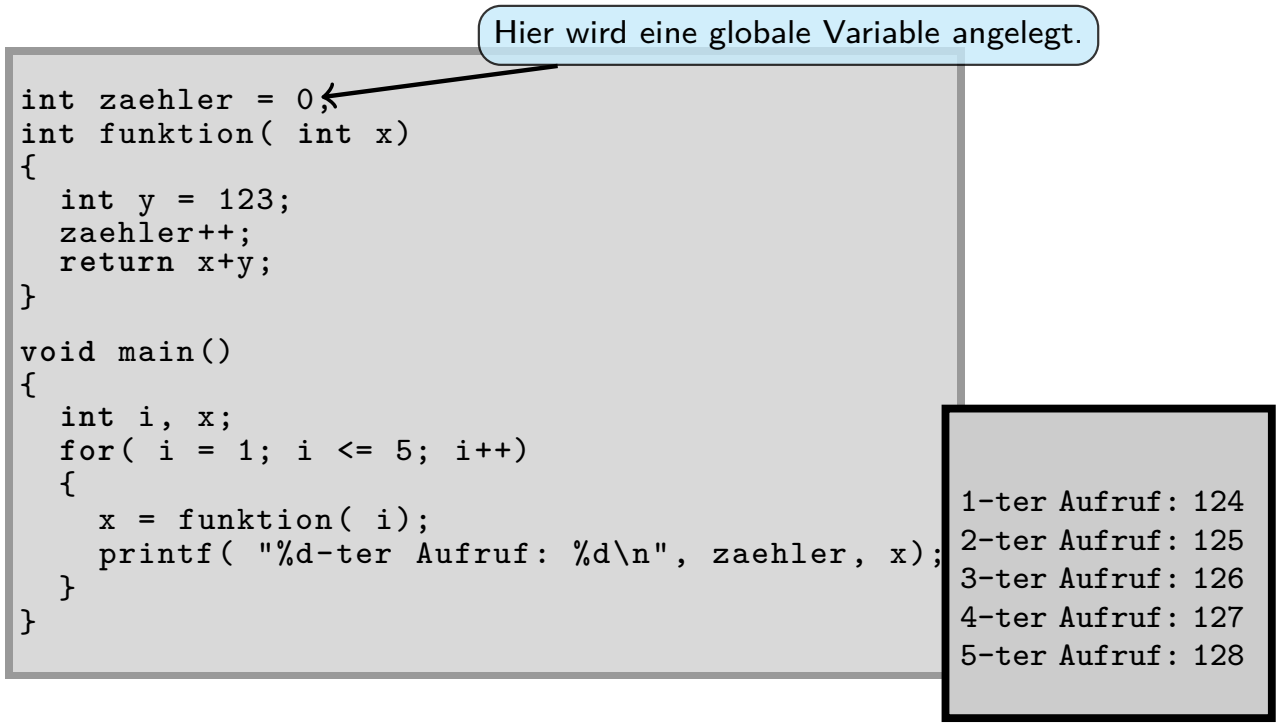
```
int zaehler = 0;
int funktion( int x)
{
  int y = 123;
  zaehler++;
  return x+y;
}

void main()
{
  int i, x;
  for( i = 1; i <= 5; i++)
  {
    x = funktion( i);
    printf( "%d-ter Aufruf: %d\n", zaehler, x);
  }
}
```

```
1-ter Aufruf: 124
2-ter Aufruf: 125
3-ter Aufruf: 126
4-ter Aufruf: 127
5-ter Aufruf: 128
```

- Globale Variablen sollten nur sparsam verwendet werden, da sie die Schnittstellen unterlaufen (Seiteneffekt). Verwenden Sie eine globale Variable nur für ein Datum, das programmweit überall mit der gleichen Bedeutung verwendet wird.
- Während lokale Variablen mit jedem Funktionsaufruf eine eigene Instanz erhalten, existieren globale (statische) Variablen nur ein mal. Rekursive (und allgemeiner: *reentrante*) Funktionen sollten deshalb i.d.R. keine solchen Variablen benutzen.

Globale Variablen

- Variablen außerhalb von Funktionen sind global und können funktionsübergreifend genutzt werden:

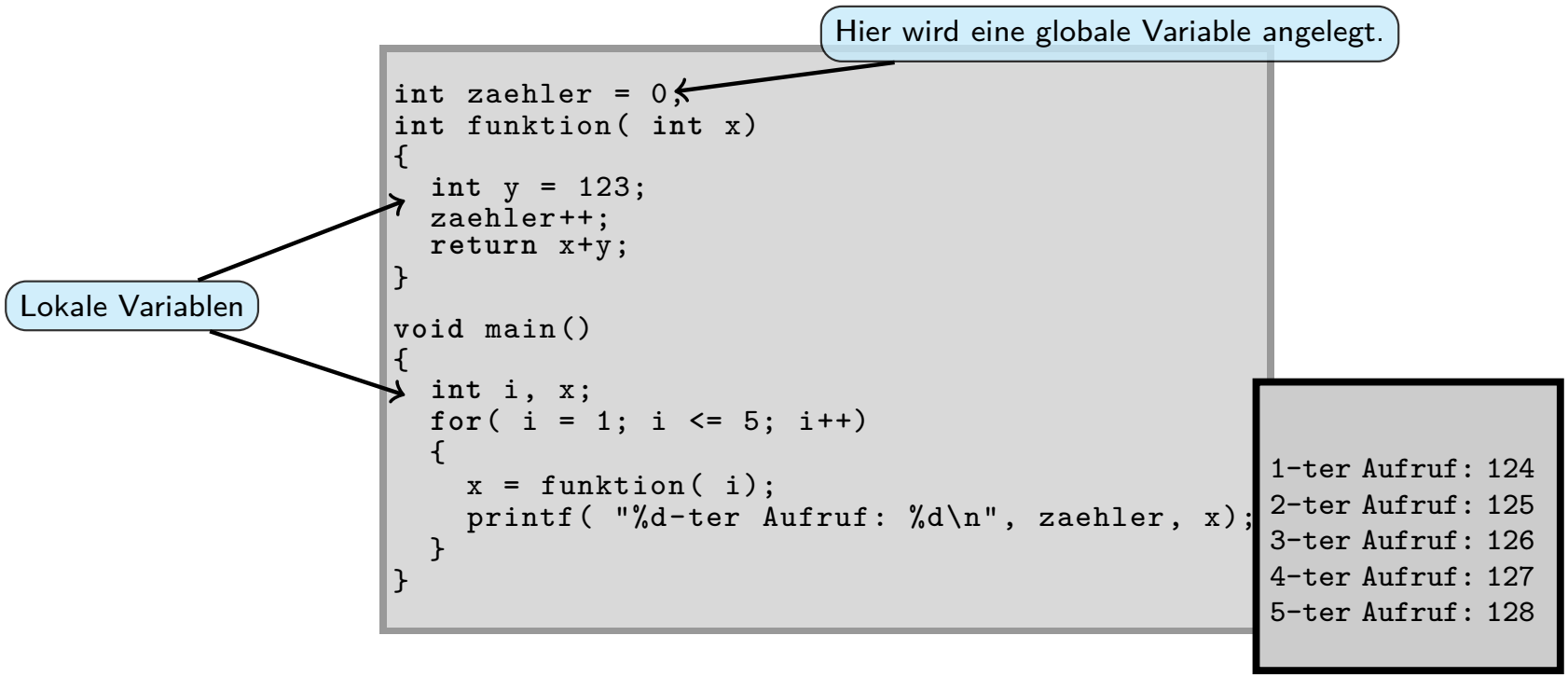


- Globale Variablen sollten nur sparsam verwendet werden, da sie die Schnittstellen unterlaufen (Seiteneffekt). Verwenden Sie eine globale Variable nur für ein Datum, das programmweit überall mit der gleichen Bedeutung verwendet wird.
- Während lokale Variablen mit jedem Funktionsaufruf eine eigene Instanz erhalten, existieren globale (statische) Variablen nur ein mal. Rekursive (und allgemeiner: *reentrante*) Funktionen sollten deshalb i.d.R. keine solchen Variablen benutzen.

Globale Variablen

Hochschule RheinMain

- Variablen außerhalb von Funktionen sind global und können funktionsübergreifend genutzt werden:

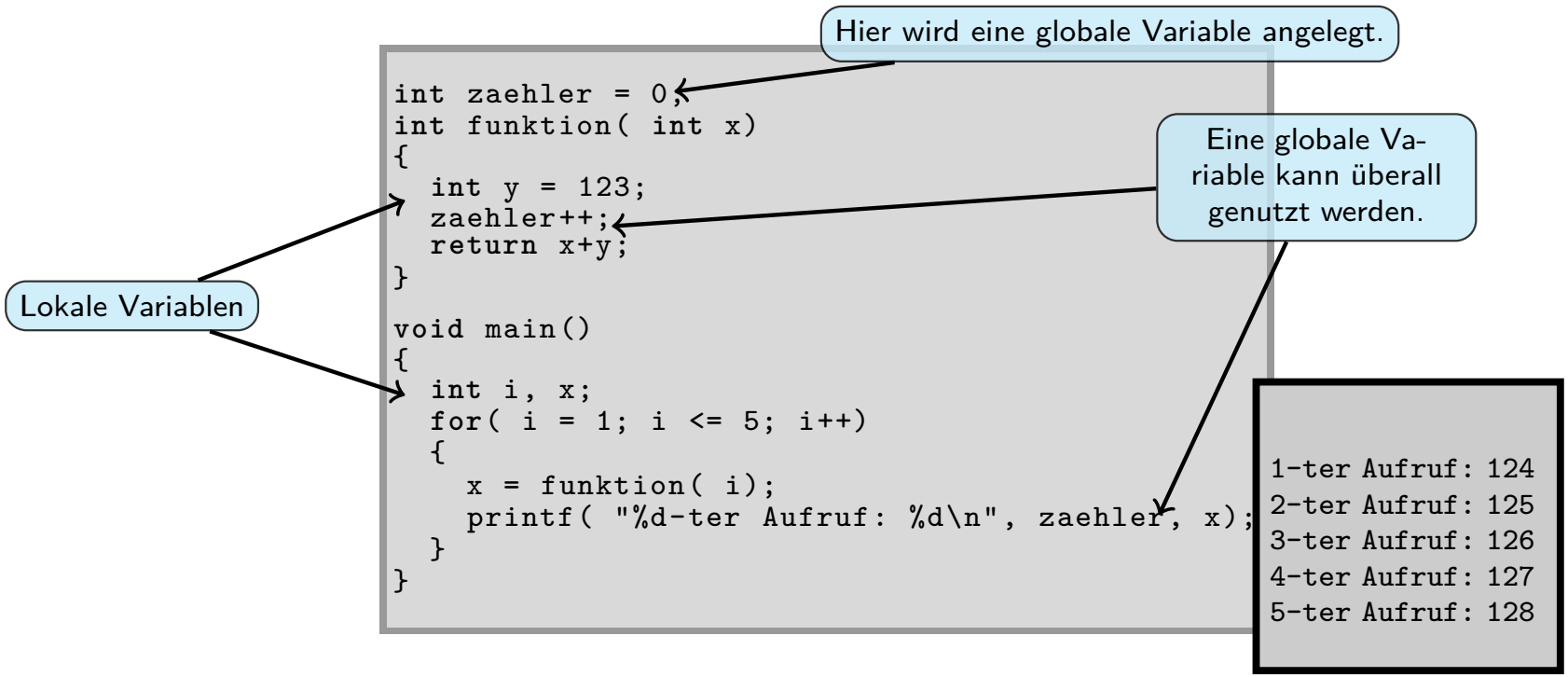


- Globale Variablen sollten nur sparsam verwendet werden, da sie die Schnittstellen unterlaufen (Seiteneffekt). Verwenden Sie eine globale Variable nur für ein Datum, das programmweit überall mit der gleichen Bedeutung verwendet wird.
- Während lokale Variablen mit jedem Funktionsaufruf eine eigene Instanz erhalten, existieren globale (statische) Variablen nur ein mal. Rekursive (und allgemeiner: *reentrante*) Funktionen sollten deshalb i.d.R. keine solchen Variablen benutzen.

Globale Variablen

Hochschule RheinMain

- Variablen außerhalb von Funktionen sind global und können funktionsübergreifend genutzt werden:

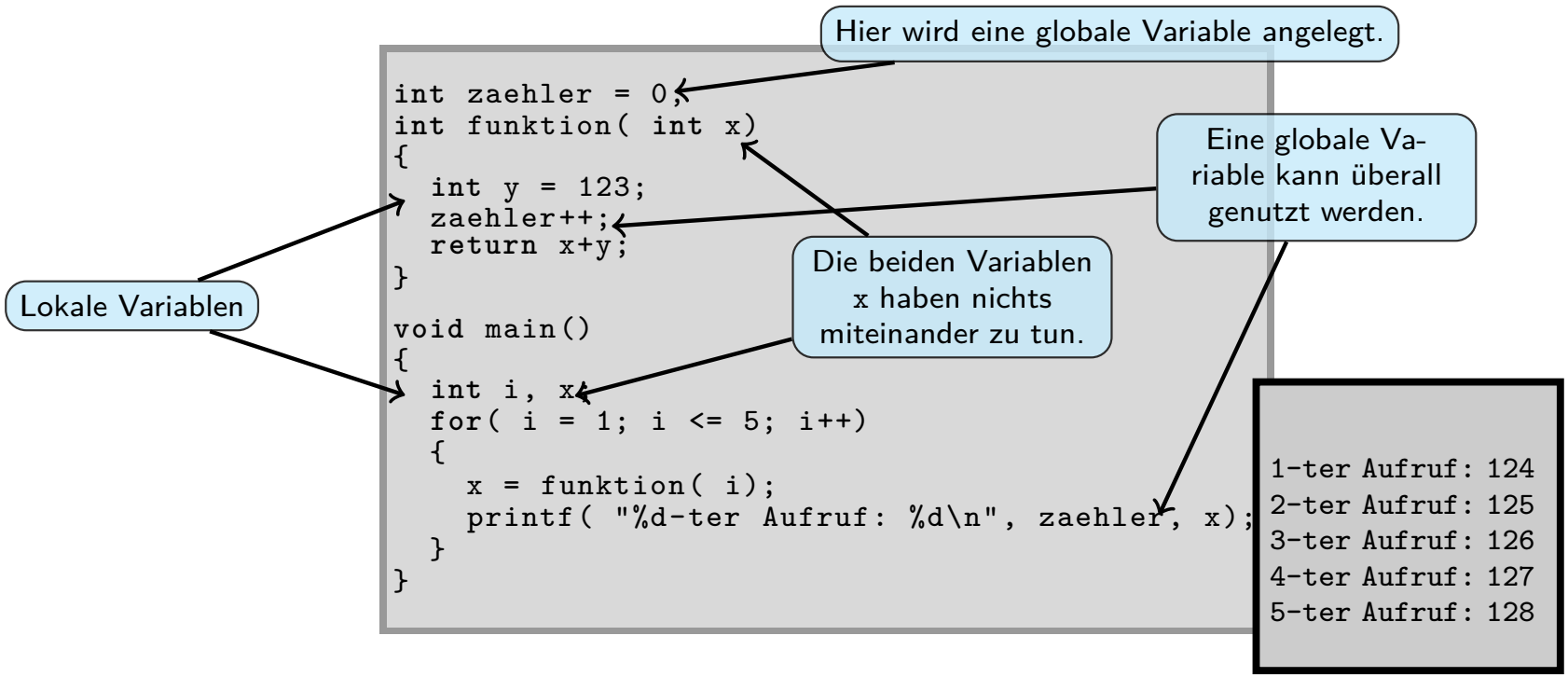


- Globale Variablen sollten nur sparsam verwendet werden, da sie die Schnittstellen unterlaufen (Seiteneffekt). Verwenden Sie eine globale Variable nur für ein Datum, das programmweit überall mit der gleichen Bedeutung verwendet wird.
- Während lokale Variablen mit jedem Funktionsaufruf eine eigene Instanz erhalten, existieren globale (statische) Variablen nur ein mal. Rekursive (und allgemeiner: *reentrante*) Funktionen sollten deshalb i.d.R. keine solchen Variablen benutzen.

Globale Variablen

Hochschule RheinMain

- Variablen außerhalb von Funktionen sind global und können funktionsübergreifend genutzt werden:



- Globale Variablen sollten nur sparsam verwendet werden, da sie die Schnittstellen unterlaufen (Seiteneffekt). Verwenden Sie eine globale Variable nur für ein Datum, das programmweit überall mit der gleichen Bedeutung verwendet wird.
- Während lokale Variablen mit jedem Funktionsaufruf eine eigene Instanz erhalten, existieren globale (statische) Variablen nur ein mal. Rekursive (und allgemeiner: *reentrante*) Funktionen sollten deshalb i.d.R. keine solchen Variablen benutzen.