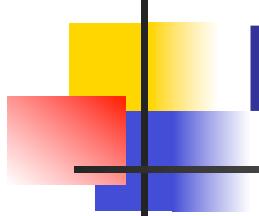


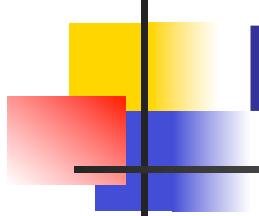
Die erste C++ Funktion

Funktionsdefinitionen- und
Aufrufe, Gültigkeitsbereich, Felder
als Funktionsargumente,
Bibliotheken, Standardfunktionen



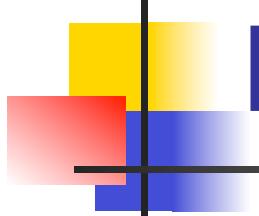
Funktionen

- kapseln häufig gebrauchte Funktionalität (z.B. Potenzberechnung) und machen sie einfach verfügbar



Funktionen

- kapseln häufig gebrauchte Funktionalität (z.B. Potenzberechnung) und machen sie einfach verfügbar
- strukturieren das Programm:
Unterteilung in kleine Teilaufgaben, jede davon durch eine Funktion realisiert



Funktionen

- kapseln häufig gebrauchte Funktionalität (z.B. Potenzberechnung) und machen sie einfach verfügbar
- strukturieren das Programm:
Unterteilung in kleine Teilaufgaben,
jede davon durch eine Funktion
realisiert

*Prozedurales Programmieren
(später mehr dazu)*

Funktion zur Potenzberechnung

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) result *= b;
    return result;
}
```

Funktion zur Potenzberechnung

```
// PRE: e >= 0 || b != 0.0 ← Vorbedingung
// POST: return value is b^e ← Nachbedingung
double pow (double b, int e)
{
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) result *= b;
    return result;
}
```

Programm zur Potenzberechnung

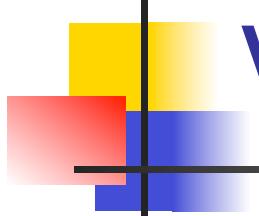
```
// Prog: callpow.cpp
// Define and call a function for computing powers.

#include <iostream>

// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) result *= b;
    return result;
}

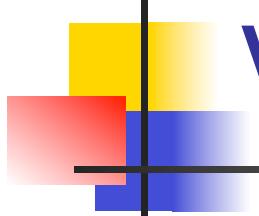
int main()
{
    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
    std::cout << pow( 5.0,  1) << "\n"; // outputs 5
    std::cout << pow( 3.0,  4) << "\n"; // outputs 81
    std::cout << pow(-2.0,  9) << "\n"; // outputs -512

    return 0;
}
```



Vor- und Nachbedingungen

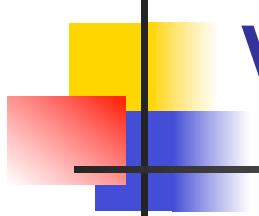
- beschreiben (möglichst vollständig) was die Funktion “macht”
- dokumentieren die Funktion für Benutzer (wir selbst oder andere)
- machen Programme lesbarer: wir müssen nicht verstehen, *wie* die Funktion es macht
- werden vom Compiler ignoriert



Vorbedingungen

Vorbedingung (*precondition*):

- o was muss bei Funktionsaufruf gelten?
- o spezifiziert *Definitionsreich* der Funktion



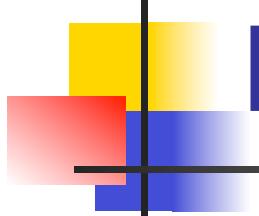
Vorbedingungen

Vorbedingung (*precondition*):

- o was muss bei Funktionsaufruf gelten?
- o spezifiziert *Definitionsbereich* der Funktion

0^e ist für $e < 0$ undefined:

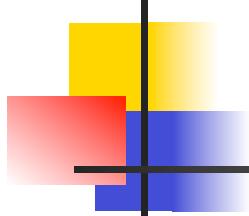
```
// PRE: e >= 0 || b != 0.0
```



Nachbedingungen

Nachbedingung (*postcondition*):

- was gilt nach dem Funktionsaufruf ?
- spezifiziert *Wert* und *Effekt* des Funktionsaufrufs



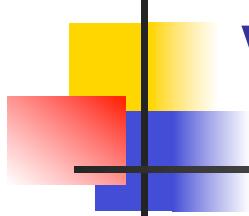
Nachbedingungen

Nachbedingung (*postcondition*):

- was gilt nach dem Funktionsaufruf ?
- spezifiziert *Wert* und *Effekt* des Funktionsaufrufs

Hier: nur Wert, kein Effekt:

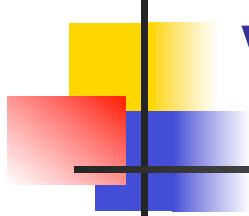
```
// POST: return value is b^e
```



Vor- und Nachbedingungen

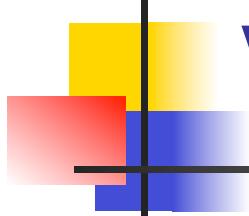
- o sind korrekt, wenn immer gilt:

Wenn die Vorbedingung beim Funktionsaufruf gilt, dann gilt auch die Nachbedingung nach dem Funktionsaufruf.



Vor- und Nachbedingungen

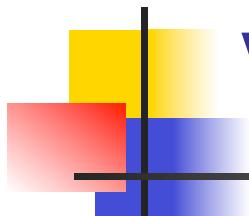
- sind korrekt, wenn immer gilt:
Wenn die Vorbedingung beim Funktionsaufruf gilt, dann gilt auch die Nachbedingung nach dem Funktionsaufruf.
- Gilt Vorbedingung beim Funktionsaufruf nicht, so machen wir keine Aussage!



Vor- und Nachbedingungen

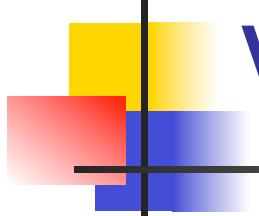
- sind korrekt, wenn immer gilt:
Wenn die Vorbedingung beim Funktionsaufruf gilt, dann gilt auch die Nachbedingung nach dem Funktionsaufruf.
- Gilt Vorbedingung beim Funktionsaufruf nicht, so machen wir keine Aussage!

C++-Standard-Jargon: “undefined behavior”



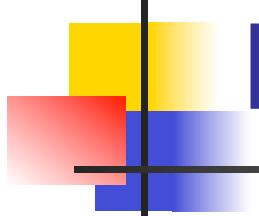
Vor- und Nachbedingungen

- Vorbedingung sollte so *schwach* wie möglich sein (möglichst grosser Definitionsbereich)



Vor- und Nachbedingungen

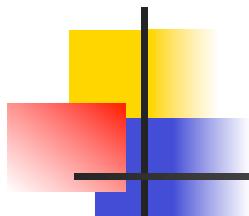
- Vorbedingung sollte so *schwach* wie möglich sein (möglichst grosser Definitionsbereich)
- Nachbedingung sollte so *stark* wie möglich sein (möglichst detaillierte Aussage)



Arithmetische Vor-und Nachbedingungen

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

sind formal inkorrekt:



Arithmetische Vor-und Nachbedingungen

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

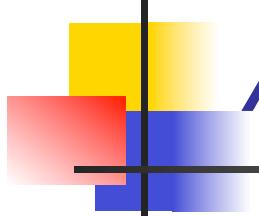
sind formal inkorrekt:

- o Überlauf, falls e oder b zu gross sind
- o b^e vielleicht nicht als **double** Wert darstellbar (Löcher im Wertebereich)

Arithmetische Vor-und Nachbedingungen

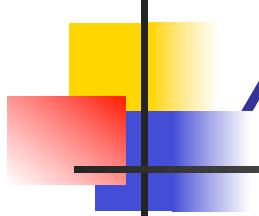
```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

Die exakten Vor- und Nachbedingungen sind plattformabhängig und meist sehr hässlich. Wir abstrahieren und geben die **mathematischen Bedingungen** an.



Assertions

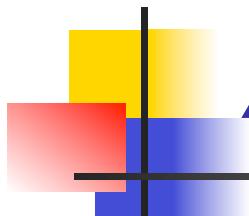
- Vorbedingungen sind nur Kommentare, wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?



Assertions

- Vorbedingungen sind nur Kommentare, wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    // der Rest ist wie vorher...
    ...
}
```



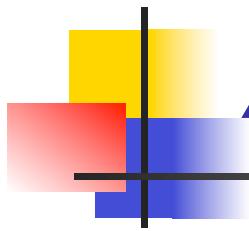
Assertions

- Vorbedingungen sind nur Kommentare, wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    // der Rest ist wie vorher...
    ...
}
```

assert (expression)

Ausdruck, dessen Wert nach `bool` konvertierbar ist



Assertions

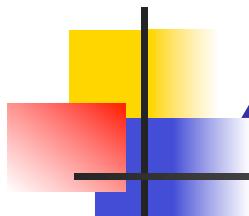
- Vorbedingungen sind nur Kommentare, wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    // der Rest ist wie vorher...
    ...
}
```

Falls *expression* Wert *true* hat: kein Effekt.

assert (*expression*)

Ausdruck, dessen Wert nach `bool` konvertierbar ist



Assertions

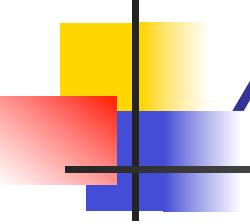
- Vorbedingungen sind nur Kommentare, wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    // der Rest ist wie vorher...
    ...
}
```

Falls *expression* Wert *false* hat: Programm wird mit entsprechender Fehlermeldung abgebrochen.

assert (expression)

Ausdruck, dessen Wert nach `bool` konvertierbar ist



Assertions

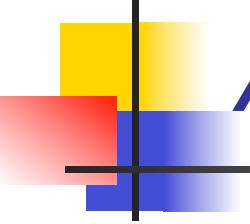
Wenn das fertige Programm einmal alle Assertions “überlebt”, so können wir den Compiler anweisen, sie “hinauszukompilieren” (kein Laufzeitverlust!).

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    // der Rest ist wie vorher...
    ...
}
```

Falls *expression* Wert *false* hat: Programm wird mit entsprechender Fehlermeldung abgebrochen.

assert (*expression*)

Ausdruck, dessen Wert nach `bool` konvertierbar ist



Assertions

Assertions sind ein wichtiges Werkzeug zur Fehlervermeidung während der Programmierung, wenn man sie *konsequent* und *oft* einsetzt.

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    // der Rest ist wie vorher...
    ...
}
```

Falls *expression* Wert *false* hat: Programm wird mit entsprechender Fehlermeldung abgebrochen.

assert (expression)

Ausdruck, dessen Wert nach `bool` konvertierbar ist

Assertions

```
#include <cassert>
```

Assertions sind ein wichtiges Werkzeug zur Fehlervermeidung während der Programmierung, wenn man sie *konsequent* und *oft* einsetzt.

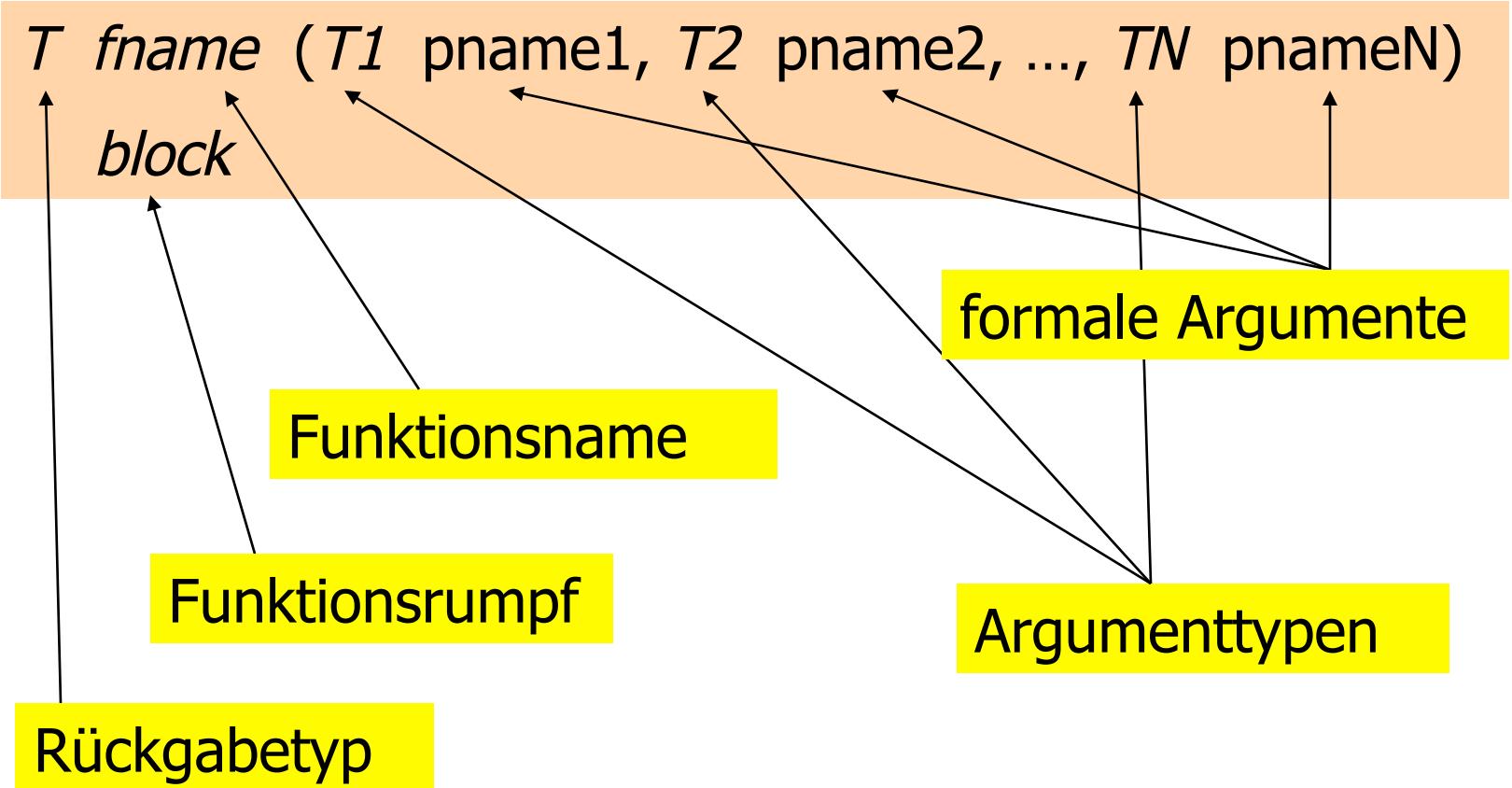
```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    // der Rest ist wie vorher...
    ...
}
```

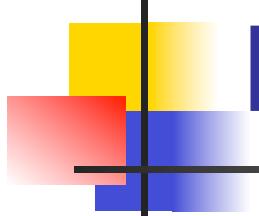
Falls *expression* Wert *false* hat: Programm wird mit entsprechender Fehlermeldung abgebrochen.

assert (expression)

Ausdruck, dessen Wert nach `bool` konvertierbar ist

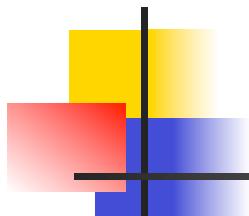
Funktionsdefinitionen





Funktionsdefinitionen

- dürfen nicht *lokal* (in Blocks, anderen Funktionen oder Kontrollanweisungen) auftreten
- können im Programm ohne Trennsymbole aufeinander folgen

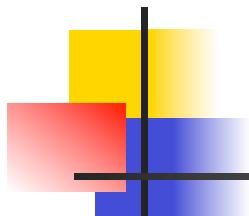


Funktionsdefinitionen

- können im Programm ohne Trennsymbole aufeinander folgen

```
int foo (int x)
{
    return x;
}

int bar (int x)
{
    return x + 1;
}
```



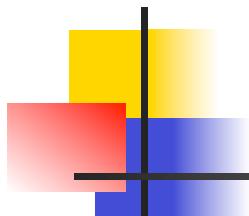
Funktionsdefinitionen

- können im Programm ohne Trennsymbole aufeinander folgen

```
int foo (int x)
{
    return x;
}

int bar (int x)
{
    return x + 1;
}
```

Diese Namen haben sich für inhaltsleere Funktionen eingebürgert, die lediglich C++ Aspekte demonstrieren.



Funktionsdefinitionen

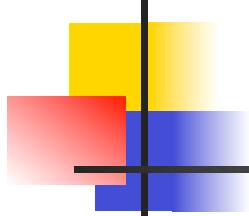
- können ohne Trennsymbole aufeinander folgen

```
int foo (int x)
{
    return x;
}

int bar (int x)
{
    return x + 1;
}
```

```
double pow (double b, int e)
{
    ...
}

int main ()
{
    ...
}
```



Funktionsaufrufe

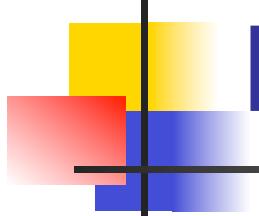
(z.B. **pow** (2.0, -2))

Aufrufargumente (Typen konvertierbar in T_1, \dots, T_N)

fname (expression₁, expression₂, ..., expression_N)



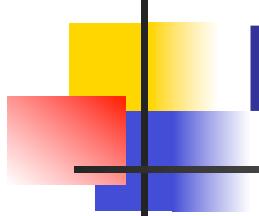
Ausdruck vom Typ T (Wert und Effekt
wie in der Nachbedingung von *fname*)



Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
- Funktionsaufruf selbst ist R-Wert

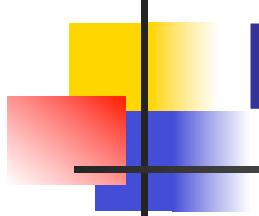


Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

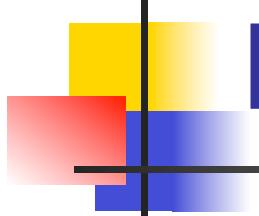
- Aufrufargumente sind R-Werte
- Funktionsaufruf selbst ist R-Wert

fname : R-Wert \times R-Wert $\times \dots \times$ R-Wert \longrightarrow R-Wert



Auswertung eines Funktionsaufrufs

- o Auswertung der Aufrufargumente
- o Initialisierung der formalen Argumente mit den resultierenden Werten
- o Ausführung des Funktionsrumpfes:
formale Argumente verhalten sich dabei wie lokale Variablen
- o Ausführung endet mit
return expression;



Auswertung eines Funktionsaufrufs

- o Auswertung der Aufrufargumente
- o Initialisierung der formalen Argumente mit den resultierenden Werten
- o Ausführung des Funktionsrumpfes:
formale Argumente verhalten sich dabei wie lokale Variablen
- o Ausführung endet mit
return expression;

Rückgabewert: Wert
des Funktionsaufrufs

Auswertung eines Funktionsaufrufs: Beispiel

`pow (2.0, -2)`

```
double pow (double b, int e)
{
    double result = 1.0;
    if (e < 0) {
        // be = (1/b)(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```

Auswertung eines Funktionsaufrufs: Beispiel

pow (2.0, -2)

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```

Auswertung eines Funktionsaufrufs: Beispiel

pow (2.0, -2)

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
    result *= b;
  return result;
}
```

Auswertung eines Funktionsaufrufs: Beispiel

pow (2.0, -2)

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e; // e = 2;
  }
  for (int i = 0; i < e; ++i)
    → result *= b; // 0.5
  return result;
}
```

i==0

Auswertung eines Funktionsaufrufs: Beispiel

`pow (2.0, -2)`

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
    → result *= b; // 0.25
  return result;
}
```

`i==1`

Auswertung eines Funktionsaufrufs: Beispiel

pow (2.0, -2)

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
    result *= b; // 0.25
  return result; // 0.25
}
```

Auswertung eines Funktionsaufrufs: Beispiel

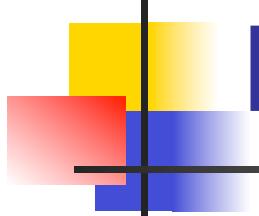
pow (2.0, -2)

Auswertung
komplett

0.25

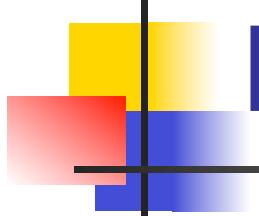
Rückgabe

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
    result *= b; // 0.25
  return result; // 0.25
}
```



Der Typ void

- fundamentaler Typ (Wertebereich leer)
- Verwendung als Rückgabetyp für Funktionen, die *nur* Effekt haben

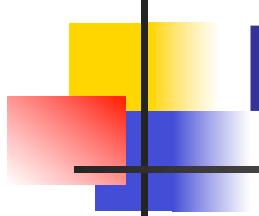


Der Typ void

- o fundamentaler Typ (Wertebereich leer)
- o Verwendung als Rückgabetyp für Funktionen, die *nur* Effekt haben

```
// POST: "(i, j)" has been written to standard output
void print_pair (const int i, const int j)
{
    std::cout << "(" << i << ", " << j << ") \n";
}

int main()
{
    print_pair(3,4); // outputs (3, 4)
}
```

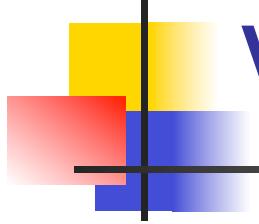


Der Typ void

- o fundamentaler Typ (Wertebereich leer)
- o Verwendung als Rückgabetyp für Funktionen, die *nur* Effekt haben

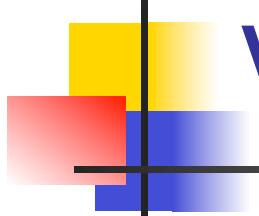
```
// POST: "(i, j)" has been written to standard output
void print_pair (const int i, const int j)           ← Void-Funktion
{
    std::cout << "(" << i << ", " << j << ") \n";
}

int main()
{
    print_pair(3,4); // outputs (3, 4)
}
```



Void-Funktionen

- brauchen kein `return`
- Ausführung endet, wenn Ende des Funktionsrumpfes erreicht wird

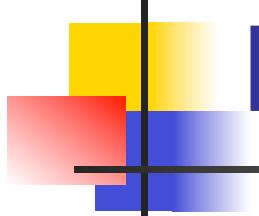


Void-Funktionen

- brauchen kein **return**
- Ausführung endet, wenn Ende des Funktionsrumpfes erreicht wird, oder...
- **return ;** erreicht wird, oder...
- **return *expression* ;** erreicht wird

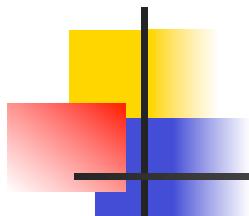


Ausdruck vom Typ **void** (z.B. Aufruf einer Funktion mit Rückgabetypr **void**)



Formale Funktionsargumente

- o Deklarative Region: Funktionsdefinition
- o sind ausserhalb der Funktionsdefinition *nicht* sichtbar
- o werden bei jedem Aufruf der Funktion neu angelegt (automatische Speicherdauer)
- o Änderungen ihrer Werte haben keinen Einfluss auf die Werte der Aufrufargumente (Aufrufargumente sind R-Werte)



Gültigkeitsbereich formaler Funktionsargumente: Beispiel 1

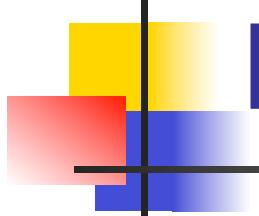
```
int main() {  
    double b = 2.0;  
    int e = -2;  
    std::cout << pow(b,e); // outputs 0.25  
    std::cout << b;        // outputs 2  
    std::cout << e;        // outputs -2  
  
    return 0;  
}
```

Gültigkeitsbereich formaler Funktionsargumente: Beispiel 1

```
int main() {  
    double b = 2.0;  
    int e = -2;  
    std::cout << pow(b,e); // outputs 0.25  
    std::cout << b;        // outputs 2  
    std::cout << e;        // outputs -2
```

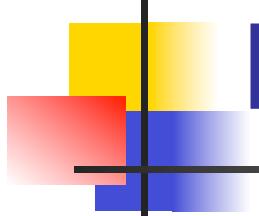
```
    return 0;
```

- } *nicht die b und e in der Definition von pow, sondern die hier definierten Variablen lokal zum Rumpf von main*



Gültigkeitsbereich einer Funktion

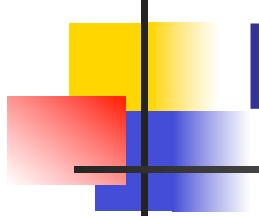
- ist der Teil des Programms, in dem die Funktion aufgerufen werden kann
- ist definiert als die Vereinigung der Gültigkeitsbereiche aller ihrer *Deklarationen*



Gültigkeitsbereich einer Funktion

- ist der Teil des Programms, in dem die Funktion aufgerufen werden kann
- ist definiert als die Vereinigung der Gültigkeitsbereiche aller ihrer *Deklarationen* (es kann mehrere geben)

Deklaration: wie Definition, aber ohne *block*



Gültigkeitsbereich einer Funktion

- ist der Teil des Programms, in dem die Funktion aufgerufen werden kann
- ist definiert als die Vereinigung der Gültigkeitsbereiche aller ihrer *Deklarationen* (es kann mehrere geben)

Deklaration: wie Definition, aber ohne *block*

Beispieldeklaration: **double pow (double b, int e)**

Gültigkeitsbereich einer Funktion: Beispiel

```
#include<iostream>

int main()
{
    std::cout << f(1); // Fehler: f undeclared
    return 0;
}

int f (const int i) // Gültigkeitsbereich von f
                  // ab hier
{
    return i;
}
```

Gültigkeitsbereich einer Funktion: Beispiel

```
#include<iostream>

int f (int i); // Gültigkeitsbereich von f ab hier

int main()
{
    std::cout << f(1); // im Gültigkeitsbereich, ok
    return 0;
}

int f (const int i)
{
    return i;
}
```

Gültigkeitsbereich einer Funktion: Beispiel

```
#include<iostream>

int f (int i); // Gültigkeitsbereich von f ab hier

int main()
{
    std::cout << f(1); // im Gültigkeitsbereich, ok
    return 0;
}

int f (const int i)
{
    return i;
}
```

Bei Deklaration:
const int äquivalent zu **int**

Gültigkeitsbereich einer Funktion: Beispiel

```
#include<iostream>

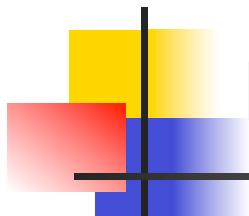
int f (int i); // Gültigkeitsbereich von f ab hier

int main()
{
    std::cout << f(1); // im Gültigkeitsbereich, ok
    return 0;
}

int f (const int i)
{
    return i;
}
```

Bei Deklaration:
`const int` äquivalent zu `int`

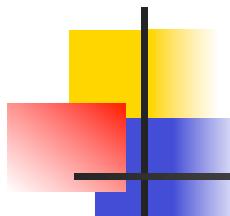
Grund: Das Verhalten der Funktion
“nach aussen” ist unabhängig davon



Separate Deklarationen sind manchmal notwendig: Problem

```
int f (...) // Gültigkeitsbereich von f ab hier
{
    g(...) // f ruft g auf, aber g ist undeclared
}

int g (...) // Gültigkeitsbereich von g ab hier
{
    f(...) // g ruft f auf, ok
}
```



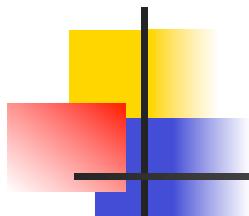
Separate Deklarationen sind manchmal notwendig: Problem

```
int g (...); // Gültigkeitsbereich von g ab hier
```

```
int f (...); // Gültigkeitsbereich von f ab hier
{
    g(...); // f ruft g auf, ok
}
```

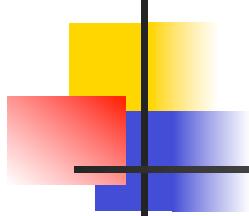
```
int g (...)

{
    f(...); // g ruft f auf, ok
}
```



Prozedurales Programmieren

- Funktionen erlauben die Zerlegung der Gesamtaufgabe in klar abgegrenzte Teilaufgaben



Prozedurales Programmieren

- Funktionen erlauben die Zerlegung der Gesamtaufgabe in klar abgegrenzte Teilaufgaben
- Bei Verwendung “sprechender” Funktionsnamen wird das Gesamtprogramm viel übersichtlicher und verständlicher

Prozedurales Programmieren: Berechnung perfekter Zahlen

Bisher (Musterlösung)

```
// input
std::cout << "Find perfect numbers up to n =? ";
unsigned int n;
std::cin >> n;

// computation and output
std::cout << "The following numbers are perfect.\n";
for (unsigned int i = 1; i <= n ; ++i) {
    // check whether i is perfect
    unsigned int sum = 0;
    for (unsigned int d = 1; d < i; ++d)
        if (i % d == 0) sum += d;
    if (sum == i)
        std::cout << i << " ";
}
```

Prozedurales Programmieren: Berechnung perfekter Zahlen

Bisher (Musterlösung)

```
// input  
  
std::cout << "Find perfect numbers up to n =? "  
  
unsigned int n;  
  
std::cin >> n;  
  
  
// computation and output  
  
std::cout << "The following numbers are perfect.\n";  
  
for (unsigned int i = 1; i <= n ; ++i) {  
    // check whether i is perfect  
  
    unsigned int sum = 0;  
  
    for (unsigned int d = 1; d < i; ++d)  
        if (i % d == 0) sum += d;  
  
    if (sum == i)  
        std::cout << i << " ";  
}
```

Was ist nochmal eine perfekte Zahl???

Geschachtelte
Schleifen

Prozedurales Programmieren: Berechnung perfekter Zahlen

Neu mit Funktionen

```
// POST: return value is the sum of all divisors of i
//        that are smaller than i

unsigned int sum_of_proper_divisors (const unsigned int i)
{
    unsigned int sum = 0;
    for (unsigned int d = 1; d < i; ++d)
        if (i % d == 0) sum += d;
    return sum;
}

// POST: return value is true if and only if i is a
//        perfect number

bool is_perfect (const unsigned int i)
{
    return sum_of_proper_divisors (i) == i;
}
```

Prozedurales Programmieren: Berechnung perfekter Zahlen

Neu mit Funktionen

```
int main()
{
    // input
    std::cout << "Find perfect numbers up to n =? ";
    unsigned int n;
    std::cin >> n;

    // computation and output
    std::cout << "The following numbers are perfect.\n";
    for (unsigned int i = 1; i <= n ; ++i)
        if (is_perfect (i)) std::cout << i << " ";
    std::cout << "\n";

    return 0;
}
```

Prozedurales Programmieren: Berechnung perfekter Zahlen

Neu mit Funktionen

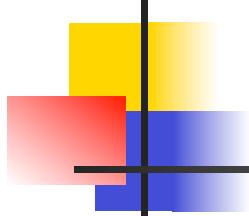
```
int main()
{
    // input
    std::cout << "Find perfect numbers up to n =? ";
    unsigned int n;
    std::cin >> n;

    // computation and output
    std::cout << "The following numbers are perfect.\n";
    for (unsigned int i = 1; i <= n ; ++i)
        if (is_perfect (i)) std::cout << i << " ";
    std::cout << "\n";

    return 0;
}
```

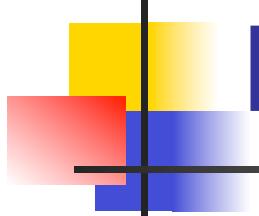
Das Programm ist selbsterklärend

keine geschachtelten
Schleifen mehr



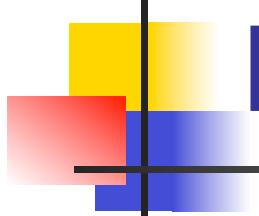
Prozedurales Programmieren

- Bisher konnten wir nur ohne Funktionen leben, weil die Programme meistens einfach und kurz waren



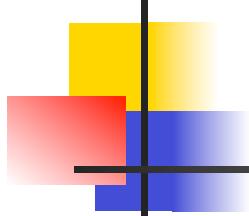
Prozedurales Programmieren

- Bisher konnten wir nur ohne Funktionen leben, weil die Programme meistens einfach und kurz waren
- Bei komplizierteren Aufgaben schreibt man ohne Funktionen leicht *Spaghetti-Code*



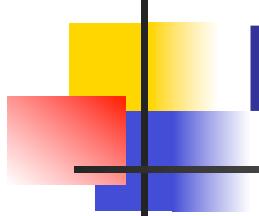
Prozedurales Programmieren

- Bisher konnten wir nur ohne Funktionen leben, weil die Programme meistens einfach und kurz waren
- Bei komplizierteren Aufgaben schreibt man ohne Funktionen leicht *Spaghetti-Code*, so wie in der...
- ...Programmiersprache BASIC (1963+)



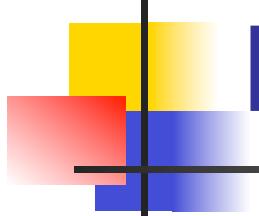
Felder als Funktionsargumente

- Zu Erinnerung: wir können Felder nicht mit anderen Feldern initialisieren



Felder als Funktionsargumente

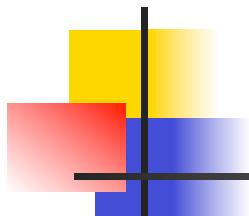
- Zu Erinnerung: wir können Felder nicht mit anderen Feldern initialisieren
- Genau das bräuchten wir aber, wenn wir Felder als Funktionsargumente benutzen wollten!



Felder als Funktionsargumente

- Zu Erinnerung: wir können Felder nicht mit anderen Feldern initialisieren
- Genau das bräuchten wir aber, wenn wir Felder als Funktionsargumente benutzen wollten!
- Lösung: wir übergeben das Feld mittels **zwei Zeigern!**

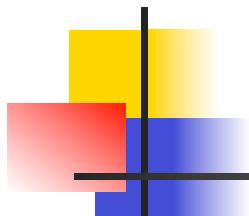




Felder als Funktionsargumente

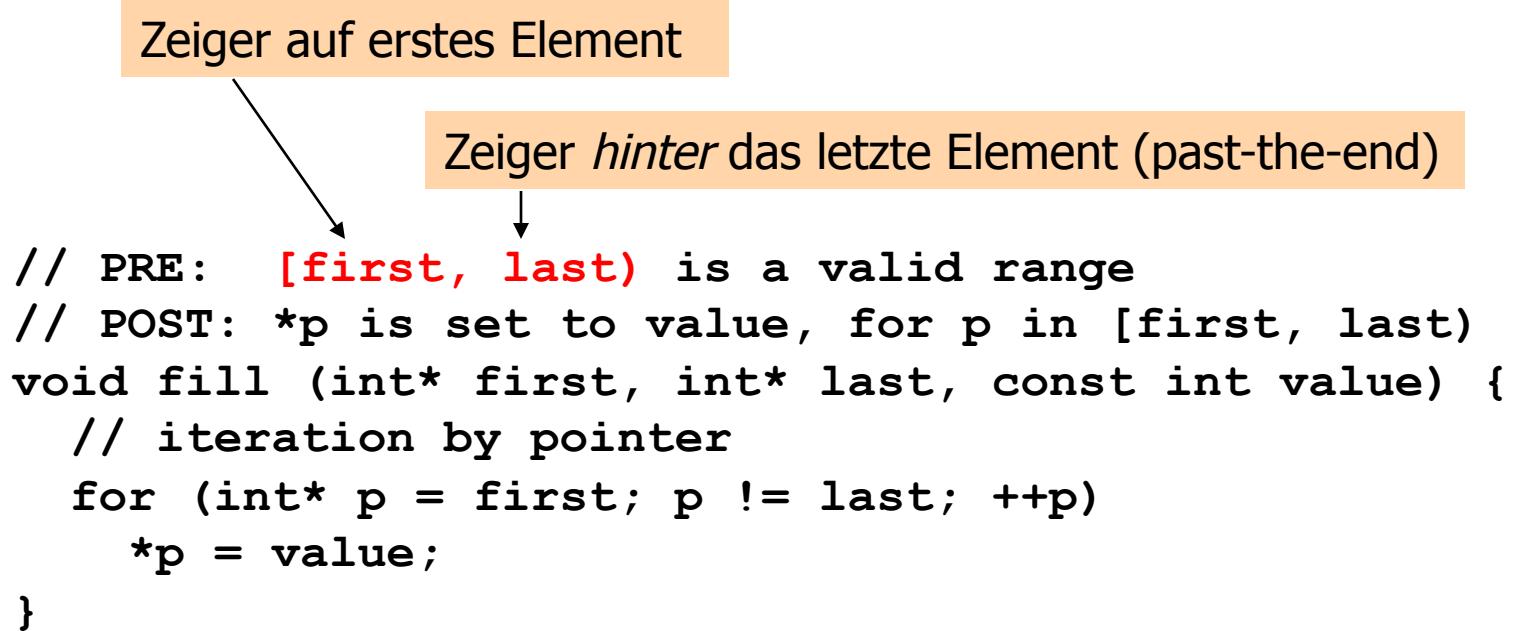
- Beispiel: Füllen aller Feldelemente mit einem festen Wert

```
// PRE: [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, const int value) {
    // iteration by pointer
    for (int* p = first; p != last; ++p)
        *p = value;
}
```



Felder als Funktionsargumente

- Beispiel: Füllen aller Feldelemente mit einem festen Wert



```
// PRE: [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, const int value) {
    // iteration by pointer
    for (int* p = first; p != last; ++p)
        *p = value;
}
```

Zeiger auf erstes Element

Zeiger *hinter* das letzte Element (past-the-end)

Felder als Funktionsargumente

- Beispiel: Füllen aller Feldelemente mit einem festen Wert

The diagram illustrates the parameters of the `fill` function. A box labeled "Zeiger auf erstes Element" has an arrow pointing to the `first` parameter in the code. Another box labeled "Zeiger *hinter* das letzte Element (past-the-end)" has an arrow pointing to the `last` parameter. A yellow callout box contains the text "Hier ‘leben’ wirklich Elemente eines Feldes". A curved arrow from this callout points back to the `last` parameter.

```
// PRE: [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, const int value) {
    // iteration by pointer
    for (int* p = first; p != last; ++p)
        *p = value;
}
```

Felder als Funktionsargumente

- Beispiel: Füllen aller Feldelemente mit einem festen Wert

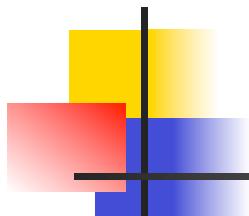
```
// PRE: [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, const int value) {
    // iteration by pointer
    for (int* p = first; p != last; ++p)
        *p = value;
}
```

Zeiger auf erstes Element

Zeiger *hinter* das letzte Element (past-the-end)

Hier "leben" wirklich
Elemente eines Feldes

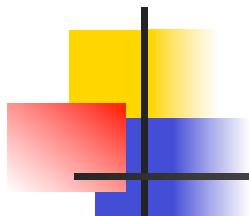
Durchlaufe den Bereich und setze jedes Element auf **value**



Felder als Funktionsargumente

- Anwendungsbeispiel:

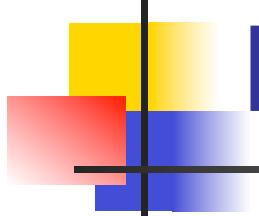
```
int main()
{
    int a[5];
    fill (a, a+5, 1); // a == {1, 1, 1, 1, 1}
    return 0;
}
```



Felder als Funktionsargumente

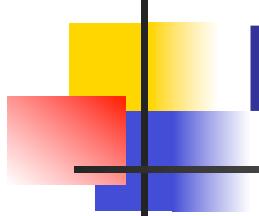
- Anwendungsbeispiel (unter Verwendung der Standardbibliothek):

```
#include <algorithm>
int main()
{
    int a[5];
    std::fill (a, a+5, 1); // a == {1, 1, 1, 1, 1}
    return 0;
}
```



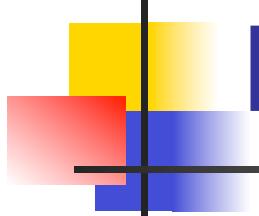
Mutierende Funktionen

- sind Funktionen, die Werte von Programmobjekten ändern.



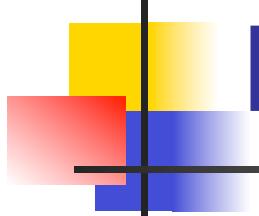
Mutierende Funktionen

- sind Funktionen, die Werte von Programmobjekten ändern.
- Beispiel: `fill`, `std::fill` (ändern Werte der Elemente eines Feldes)



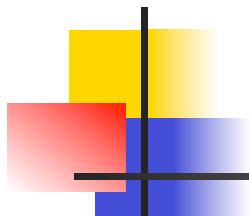
Mutierende Funktionen

- sind Funktionen, die Werte von Programmobjekten ändern.
- Beispiel: `fill`, `std::fill` (ändern Werte der Elemente eines Feldes)
- `pow` ist eine nichtmutierende Funktion.



Modularisierung

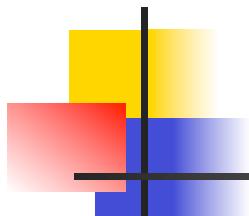
- Funktionen wie `pow`, `fill`, ... sind in vielen Programmen nützlich.
- Es ist nicht sehr praktisch, die Funktionsdefinition in jedem solchen Programm zu wiederholen.
- Ziel: Auslagern der Funktion



Modularisierung: Auslagerung

Separate Datei `pow.cpp`:

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) result *= b;
    return result;
}
```



Modularisierung: Auslagerung

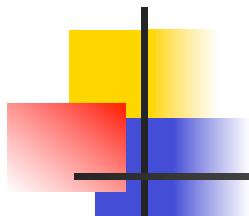
Aufrufendes Programm `callpow2.cpp`:

```
// Prog: callpow2.cpp
// Call a function for computing powers.

#include <iostream>
#include "pow.cpp"

int main()
{
    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
    std::cout << pow( 5.0,  1) << "\n"; // outputs 5
    std::cout << pow( 3.0,  4) << "\n"; // outputs 81
    std::cout << pow(-2.0,  9) << "\n"; // outputs -512

    return 0;
}
```



Modularisierung: Auslagerung

Aufrufendes Programm `callpow2.cpp`:

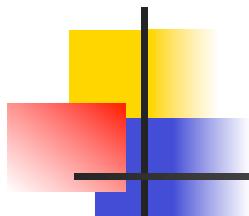
```
// Prog: callpow2.cpp
// Call a function for computing powers.

#include <iostream>
#include "pow.cpp"
int main()
{
    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
    std::cout << pow( 5.0,  1) << "\n"; // outputs 5
    std::cout << pow( 3.0,  4) << "\n"; // outputs 81
    std::cout << pow(-2.0,  9) << "\n"; // outputs -512

    return 0;
}
```

Dateiangabe relativ zum Arbeitsverzeichnis

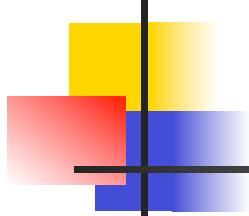




Modularisierung: getrennte Übersetzung

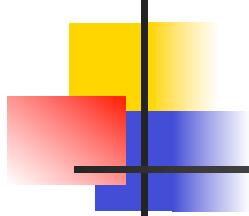
Problem der Auslagerung:

- `#include` kopiert den Inhalt der inkludierten Datei (`pow.cpp`) in das inkludierende Programm (`callpow2.cpp`)
- Compiler muss die Funktionsdefinition für jedes aufrufende Programm neu übersetzen.
- bei vielen und grossen Funktionen kann das jeweils sehr lange dauern.



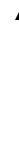
Modularisierung: getrennte Übersetzung

- `pow.cpp` kann getrennt übersetzt werden (z.B. `g++ -c pow.cpp`)
- Resultat ist kein ausführbares Programm (`main` fehlt), sondern **Objekt-Code** `pow.o`

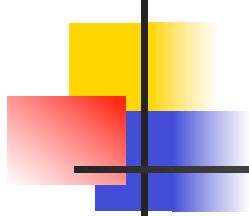


Modularisierung: getrennte Übersetzung

- `pow.cpp` kann getrennt übersetzt werden (z.B. `g++ -c pow.cpp`)
- Resultat ist kein ausführbares Programm (`main` fehlt), sondern **Objekt-Code `pow.o`**

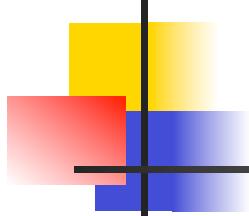


Maschinensprache-Befehle für Funktionsrumpf von `pow`



Modularisierung: getrennte Übersetzung

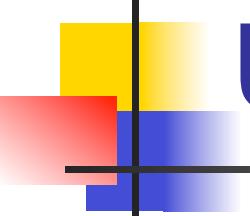
- Auch das aufrufende Programm kann getrennt übersetzt werden, ohne Kenntnis von **pow.cpp** oder **pow.o** !



Modularisierung: getrennte Übersetzung

- Auch das aufrufende Programm kann getrennt übersetzt werden, ohne Kenntnis von `pow.cpp` oder `pow.o` !
- Compiler muss nur die *Deklaration* von `pow` kennen; diese schreiben wir in eine *Header*-Datei `pow.h`:

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e);
```



Modularisierung: getrennte Übersetzung

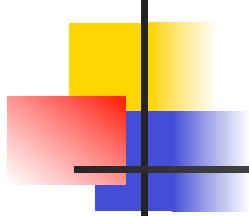
Aufrufendes Programm `callpow3.cpp`:

```
// Prog: callpow3.cpp
// Call a function for computing powers.

#include <iostream>
#include "pow.h"

int main()
{
    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
    std::cout << pow( 5.0,  1) << "\n"; // outputs 5
    std::cout << pow( 3.0,  4) << "\n"; // outputs 81
    std::cout << pow(-2.0,  9) << "\n"; // outputs -512

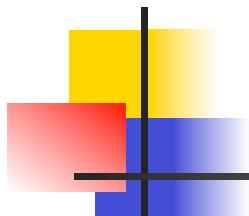
    return 0;
}
```



Modularisierung: getrennte Übersetzung

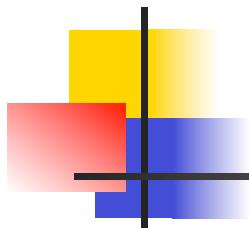
- `callpow3.cpp` kann getrennt übersetzt werden (`g++ -c callpow3.cpp`)
- Resultat ist kein ausführbares Programm (`pow` fehlt), sondern **Objekt-Code** `callpow3.o`

Anstatt der Maschinensprache-Befehle für Funktionsrumpf von `pow` enthält `callpow3.o` einen *Platzhalter* für die Speicheradresse, unter der diese zu finden sein werden.



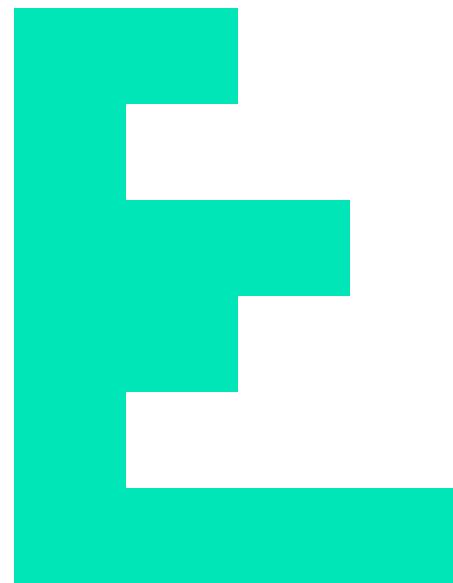
Modularisierung: Der Linker

- baut ausführbares Programm aus den relevanten Objekt-Codes (`pow.o` und `callpow3.o`) zusammen
- ersetzt dabei die Platzhalter für Funktionsrumpfadressen durch deren wirkliche Adressen im ausführbaren Programm

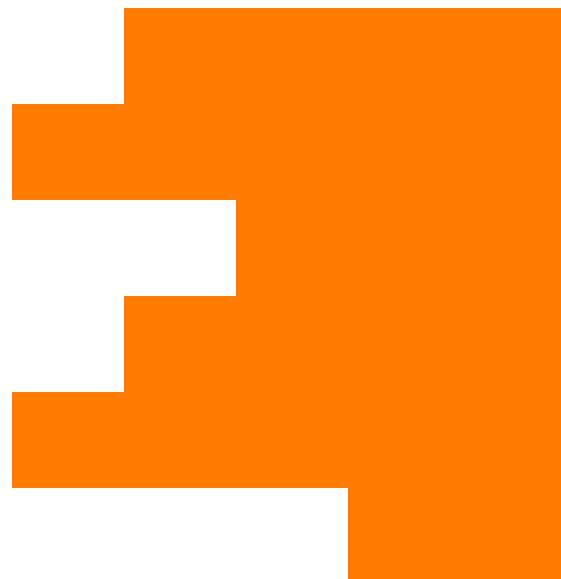


Modularisierung: Der Linker

pow . o

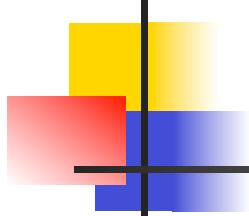


callpow3 . o



Maschinensprache für den
Funktionsrumpf von **pow**

Maschinensprache für die
Aufrufe von **pow**



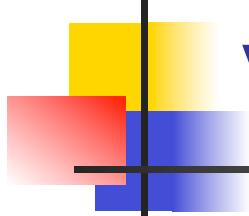
Modularisierung: Der Linker

`callpow3` (ausführbares Programm)



Maschinensprache für den
Funktionsrumpf von `pow`

Maschinensprache für die
Aufrufe von `pow`



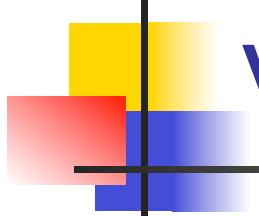
Modularisierung: Verfügbarkeit von Quellcode

- `pow.cpp` (Quellcode) wird nach dem Erzeugen von `pow.o` (Objekt-Code) nicht mehr gebraucht und könnte gelöscht werden
- Viele (meist kommerzielle) Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode
- Vor-und Nachbedingungen in Header-Dateien sind die *einzigen* verfügbaren Informationen

Modularisierung: Verfügbarkeit von Quellcode

“Open source” Software:

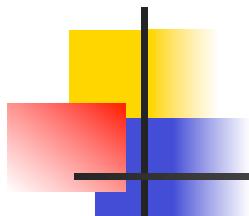
- Alle Quellen sind verfügbar
- nur das erlaubt die Weiterentwicklung durch Benutzer und andere interessierte Personen
- im akademischen Bereich ist “open source” Software weit verbreitet



Modularisierung: Verfügbarkeit von Quellcode

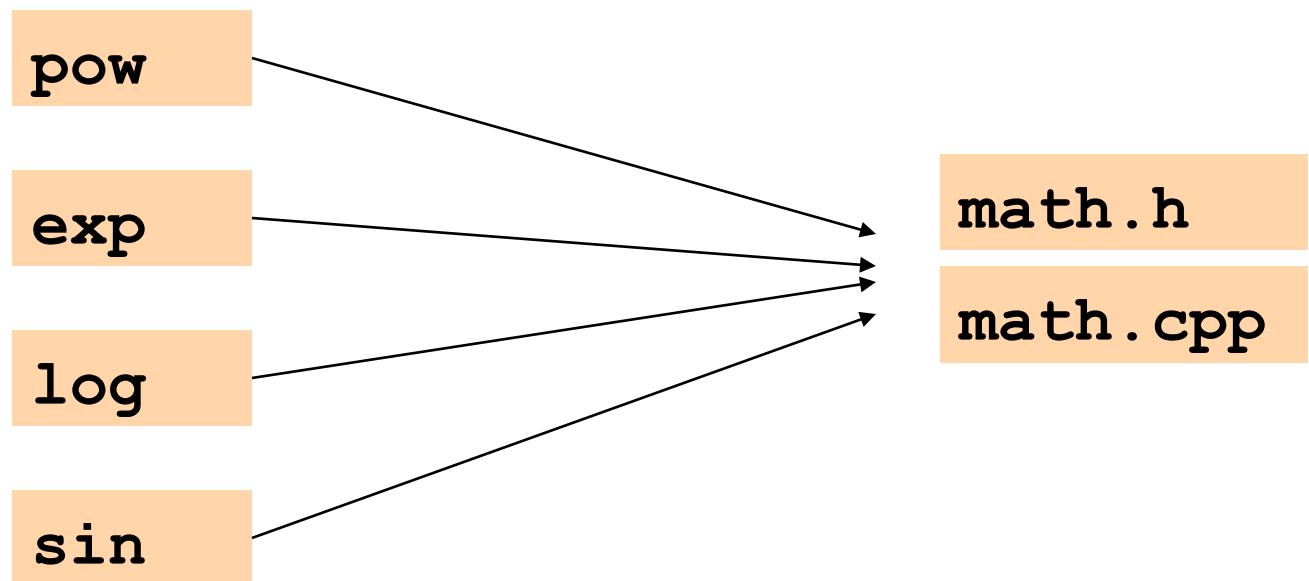
“Open source” Software:

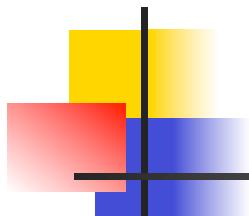
- Alle Quellen sind verfügbar
- nur das erlaubt die Weiterentwicklung durch Benutzer und andere interessierte Personen
- im kommerziellen Bereich ist “open source” auf dem Vormarsch (trotz Microsoft...)
- Bekannteste “open source” Software: Das Betriebssystem **Linux**



Modularisierung: Bibliotheken

- Gruppierung ähnlicher Funktionen zu Bibliotheken



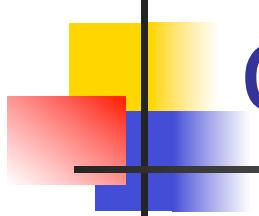


Modularisierung: Bibliotheken

- Eigener Namensraum vermeidet Konflikte mit Benutzer-Funktionalität

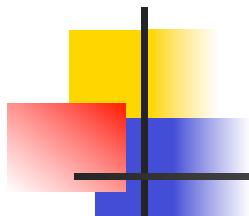
```
// math.h
// A small library of mathematical functions.

namespace ifm {
    // PRE: e >= 0 || b != 0.0
    // POST: return value is b^e
    double pow (double b, int e);
    ...
    double exp (double x);
    ...
}
```



Benutzen von Funktionen aus der Standardbibliothek

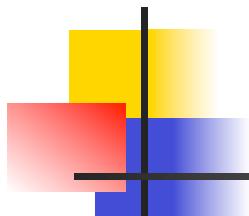
- vermeidet die Neuerfindung des Rades (z.B. gibt es `std::pow`, `std::fill`)
- führt auf einfache Weise zu interessanten und effizienten Programmen



Primzahltest mit `std::sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn
kein d in $\{2, \dots, n-1\}$ ein Teiler von n ist.

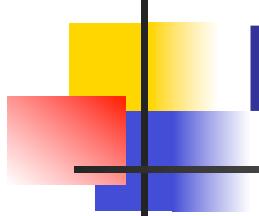
```
unsigned int d;  
for (d = 2; n % d != 0; ++d);
```



Primzahltest mit `std::sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn
kein d in $\{2, \dots, \sqrt{n}\}$ ein Teiler von n ist.

```
const unsigned int bound =  
    (unsigned int)(std::sqrt(n));  
unsigned int d;  
for (d = 2; d <= bound && n % d != 0; ++d);
```



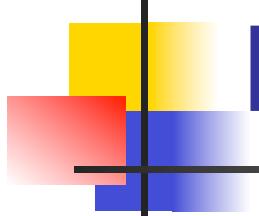
Primzahltest mit `std::sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn
kein d in $\{2, \dots, \sqrt{n}\}$ ein Teiler von n ist.

```
const unsigned int bound =  
    (unsigned int)(std::sqrt(n));  
unsigned int d;  
for (d = 2; d <= bound && n % d != 0; ++d);
```

Fall 1: Nach `for`-Anweisung gilt

$d \leq \text{bound}$ (das heisst $d < n$):



Primzahltest mit `std::sqrt`

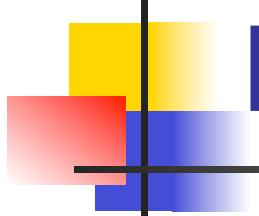
$n \geq 2$ ist Primzahl genau dann, wenn
kein d in $\{2, \dots, \sqrt{n}\}$ ein Teiler von n ist.

```
const unsigned int bound =  
    (unsigned int)(std::sqrt(n));  
unsigned int d;  
for (d = 2; d <= bound && n % d != 0; ++d);
```

Fall 1: Nach `for`-Anweisung gilt

$d \leq \text{bound}$ (das heisst $d < n$):

Dann: $n \% d == 0$, d ist echter Teiler, n keine Primzahl

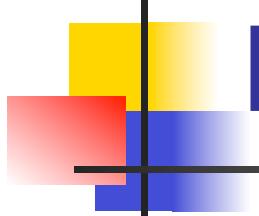


Primzahltest mit `std::sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn
kein d in $\{2, \dots, \sqrt{n}\}$ ein Teiler von n ist.

```
const unsigned int bound =  
    (unsigned int)(std::sqrt(n));  
unsigned int d;  
for (d = 2; d <= bound && n % d != 0; ++d);
```

Fall 2: Nach `for`-Anweisung gilt
 $d > bound$ (das heisst $d > \sqrt{n}$):



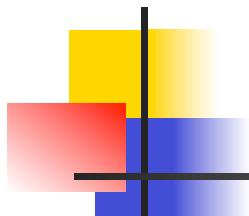
Primzahltest mit `std::sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn
kein d in $\{2, \dots, \sqrt{n}\}$ ein Teiler von n ist.

```
const unsigned int bound =  
    (unsigned int)(std::sqrt(n));  
unsigned int d;  
for (d = 2; d <= bound && n % d != 0; ++d);
```

Fall 2: Nach `for`-Anweisung gilt
 $d > bound$ (das heisst $d > \sqrt{n}$):

Dann: kein Teiler in $\{2, \dots, \sqrt{n}\}$, n ist Primzahl



Primzahltest mit std::sqrt

```
// Program: prime2.cpp
// Test if a given natural number is prime.

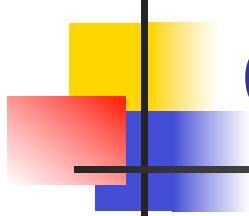
#include <iostream>
#include <cmath>

int main ()
{
    // Input
    unsigned int n;
    std::cout << "Test if n>1 is prime for n =? ";
    std::cin >> n;

    // Computation: test possible divisors d up to sqrt(n)
    const unsigned int bound = (unsigned int)(std::sqrt(n));
    unsigned int d;
    for (d = 2; d <= bound && n % d != 0; ++d);

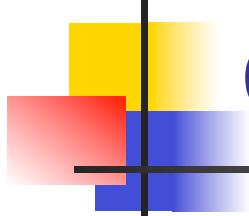
    // Output
    if (d <= bound)
        // d is a divisor of n in {2, ..., [sqrt(n)]}
        std::cout << n << " = " << d << " * " << n / d << ".\n";
    else
        // no proper divisor found
        std::cout << n << " is prime.\n";

    return 0;
}
```



Benutzen von Funktionen aus der Standardbibliothek

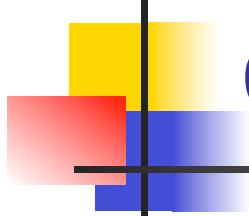
prime2.cpp könnte inkorrekt sein, falls
z.B. **std::sqrt(121) == 10.998**



Benutzen von Funktionen aus der Standardbibliothek

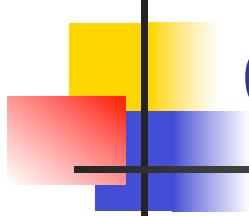
`prime2.cpp` könnte inkorrekt sein, falls
z.B. `std::sqrt(121) == 10.998`

- `bound == 10`
- `d == 11` am Ende von `for`
- Ausgabe: `121 is prime.`



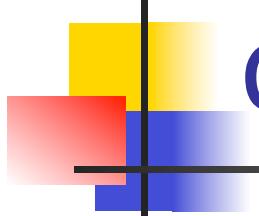
Benutzen von Funktionen aus der Standardbibliothek

- Für `std::sqrt` garantiert der IEEE Standard 754 noch, dass der exakte Wert auf den nächsten darstellbaren Wert gerundet wird (wie bei `+, -, *, /`)



Benutzen von Funktionen aus der Standardbibliothek

- Für `std::sqrt` garantiert der IEEE Standard 754 noch, dass der exakte Wert auf den nächsten darstellbaren Wert gerundet wird (wie bei +, -, *, /)
- Also: `std::sqrt(121) == 11`
- `prime2.cpp` ist korrekt unter dem IEEE Standard 754



Benutzen von Funktionen aus der Standardbibliothek

- Für andere mathematische Funktionen gibt der IEEE Standard 754 *keine* solchen Garantien, sie können also auch weniger genau sein!
- Gute Programme müssen darauf Rücksicht nehmen und, falls nötig, “Sicherheitsmassnahmen” einbauen.