Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. Dr. h.c. J. Becker
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

Übung02: Informationstechnik (IT)

Harald Bucher

Institut für Technik der Informationsverarbeitung (ITIV)

Teil 2: Funktionen, Zeiger & Header

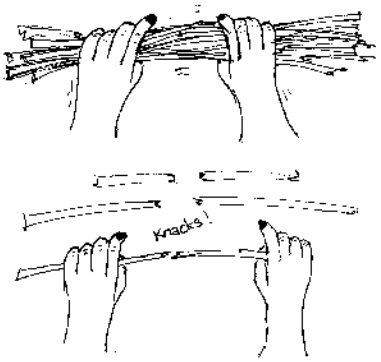
Inhalt: Übung02

1. Besprechung Übungsaufgaben
2. Funktionen
3. Zeiger & Referenzen
4. Header-Dateien

Funktionen

- Problemstellung
  - Wie kann ich ein Programm in mehrere Teile gliedern?
  - Wie kann ich den gleichen Programmablauf an verschiedenen Stellen mit unterschiedlichen Daten einfach wiederholen?

- Lösung: Funktionen
  - Teilung der Problemstellung in kleinere einfacher zu lösende Probleme, welche jeweils in einer Funktion behandelt werden
  - Nach dem Prinzip: Teile und Herrsche

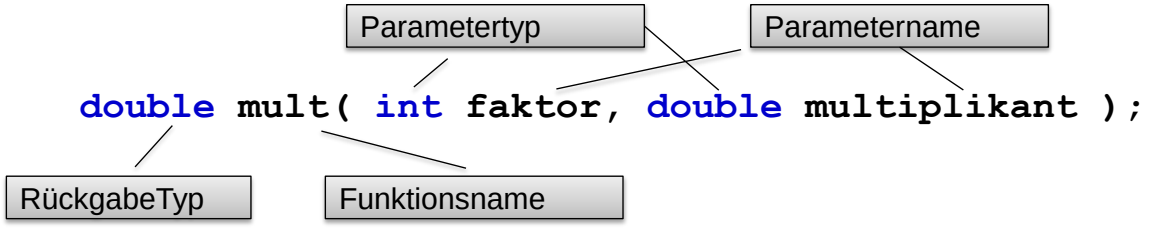


Deklaration von Funktionen

- Muss wie eine Variable deklariert (bekanntgemacht) werden
- Prototyp besteht aus Namen, Parametertypen & Ergebnistyp

- Syntax:
```
rueckgabeTyp funktionsName( parameterTyp1
    parameter1, parameterTyp2 parameter2, ... );
```
  - Typen können beliebige Variablentypen sein
  - Übergabeparameter sind optional
  - Soll nichts zurückgegeben werden: `void` verwenden

- Beispiel eines Prototyps:



Definition einer Funktion

- Beschreibung der Funktionalität einer Funktion
- Syntax:

```
rueckgabeTyp funktionsName( argumentTyp1 argument1, ... ) {
  Anweisungen;
  ...
  return rueckgabeWert;
}
```

Wie bei der Deklaration der Funktion allerdings ohne Semikolon

Anweisungen innerhalb der Funktion stehen in geschweiften Klammern

Funktion wird bei `return` direkt verlassen
Optional bei void-Funktionen

- Beispiel:

```
double mult( int faktor, double multiplikant ) {
  double ergebnis = faktor * multiplikant;
  return ergebnis;
}
```

Der Inhalt von `ergebnis` ersetzt nun sozusagen den Funktionsaufruf

Aufruf einer Funktion

- Aufruf einer Funktion
  - Können direkt als einzelne Anweisung aufgerufen werden
    Beispiel: `gebeAus( "Vogonische Kampfpoesie" );`
  - Können innerhalb anderer Funktionen als Werte verwendet werden
- Dies passiert bei Aufruf einer Funktion

```
...
double erg; double wert = 1.2;
erg = mult( 3, wert );
...
```

Werte werden übergeben

```
double mult( int faktor, double multiplikant ) {
  double ergebnis = faktor * multiplikant;
  return ergebnis;
}
```

Ergebnis wird zurückgegeben

Zwischenübung01: Funktionen

Schreiben Sie die Prototypen der folgenden Funktionen!

a) Die Funktion **`sum()`** liefert die Summe von drei **`double`**-Werten, die als Argumente übergeben werden.

b) Die Funktion **`isLeapYear()`** erhält eine Jahreszahl als Argument und gibt **`true`** zurück, falls das Jahr ein Schaltjahr ist, andernfalls **`false`**.

Zwischenübung01: Funktionen Lsg.

Schreiben Sie die Prototypen der folgenden Funktionen!

a) Die Funktion **`sum()`** liefert die Summe von drei **`double`**-Werten, die als Argumente übergeben werden.

```
double sum( double a, double b, double c );
```

b) Die Funktion **`isLeapYear()`** erhält eine Jahreszahl als Argument und gibt **`true`** zurück, falls das Jahr ein Schaltjahr ist, andernfalls **`false`**.

```
bool isLeapYear( int n );
```

Adressvariablen: Deklaration

- Auch Zeiger oder engl. Pointer genannt

- Speichern die Adressen von Speicherzellen
- Deklaration durch Anhängen von `*` an den Variablentyp

- Beispiel:
  - Normale Variable: `int wertvariable;`
  - Adressvariable: `int* adressvariable;`

- Es gibt zu jedem Variablentyp (auch für die Selbsterstellten) einen entsprechenden Adressvariablentyp

- Belegt unabhängig vom Typ immer den gleichen Speicherplatz
  - Je nach PC und Betriebssystem (im Allgemeinen 4 Byte – 32 Bit)

Adressvariablen: Zuweisung

- Speichern von Adressen in Adressvariablen
  - Zuweisung von Adressen mit Hilfe von `&` Operator
  - Merksatz: `&` bedeutet „Adresse von“
- `&` vor Variable, ist die Speicheradresse der Variablen

- Beispiel:

```
int wertVariable = 1000;
int* adressVariable;
adressVariable = &wertVariable;
```

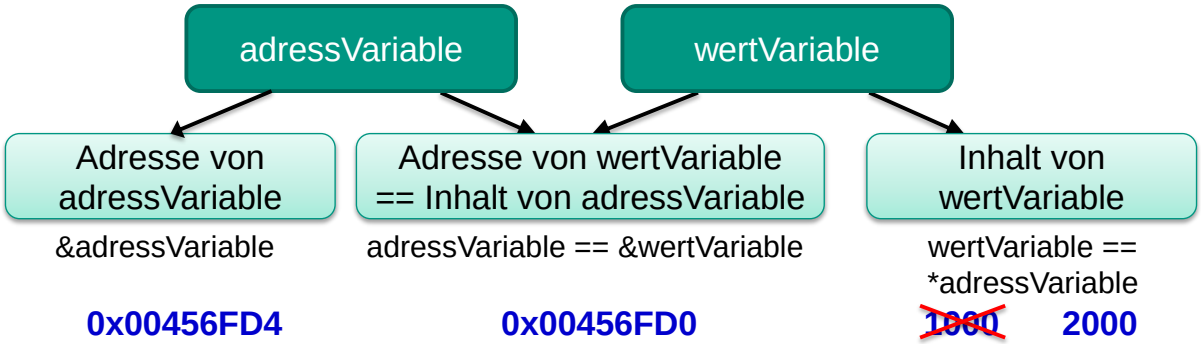
Variable	Inhalt der Variablen	Adresse (hex)
`wertVariable`	`1000`	`0x00456FD0`
`adressVariable`	`0x00456FD0`	`0x00456FD4`

Adressvariablen: Zugriff

- Um die Variable zu erhalten, auf welche ein Zeiger verweist, verwendet man den * Operator (Dereferenzierung)
- * vor Adressvariable, wird zum Inhalt, welcher an dieser Adresse gespeichert ist

- Beispiel:

```
int wertVariable = 1000;
int* adressVariable = &wertVariable;
*adressVariable = *adressVariable * 2;
```



Referenzen

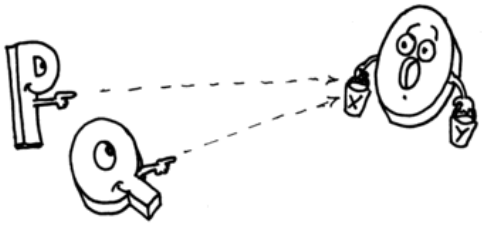
- Problemstellung:
  - Wie kann man auf eine Variable mit mehreren Namen zugreifen?

- Lösung: Referenzen
  - Kann direkt wie die Originalvariable verwendet werden
  - Bei Funktionen sehr nützlich

- Deklaration mit & „Ampersand“
- Beispiel:

```
int wert = 10.7;
int& verweis = wert;
```

- Keine Variable → belegt keinen Speicher
  - Muss initialisiert werden
  - Später nicht mehr veränderbar

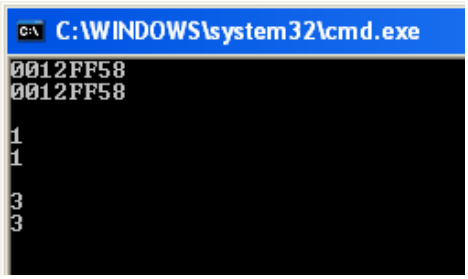


Arrays und Zeiger

- Arrayname = Zeiger auf das erste Arrayelement

- Beispiel:
```
int arr[3] = { 1, 2, 3 };
int* arrayZeiger = arr; // = &arr[0];
```
- `arr` und `arrayZeiger` sind Integer-Zeiger auf `arr[0]`, an dieser Stelle ist der Wert `1` gespeichert

- Beispiel:
```
cout << arrayZeiger << endl;
cout << arr << endl << endl;

cout << *arrayZeiger << endl;
cout << *arr << endl << endl;

cout << *(arrayZeiger + 2) << endl;
cout << *(arr + 2) << endl << endl;
```



Arrays und Zeiger – Rechnen mit Zeigern

- Zugriff auf das Array auch über Zeiger möglich (schreiben / lesen)
  - `arr[2]` ist gleichbedeutend mit `*( arr + 2 )`
  - Es werden auch hier keine Grenzen überprüft
  - Größe der Rechenschritte im Speicher wird entsprechend dem Typ des Zeigers automatisch angepasst

- Beispiel: `int arr[3] = { 1, 2, 3 };`

```
int* arrayZeiger = arr;
arrayZeiger = arrayZeiger + 2;
```

Variable	Inhalt der Variablen	Adresse (hex)
`arr[0]`	1	0x0012FF50
`arr[1]`	2	0x0012FF54
`arr[2]`	3	0x0012FF58
`arrayZeiger`	~~0x0012FF50~~ 0x0012FF58	0x0012FF40

Zeigerarithmetik und Read-Only Zeiger

- Arithmetische Operationen und Vergleiche sind mit Zeigern möglich
  - Operatoren `++, --, +=, -=, ...`
  - Auf Operatorenreihenfolge achten (oder Klammern setzen)
- Subtraktion von Zeigern zur Bestimmung des Index in Arrays
  - Addition von Zeigern nicht zulässig, da auch nicht sinnvoll
- Beispiel:

```
int index = arrayzeiger - arr;
```

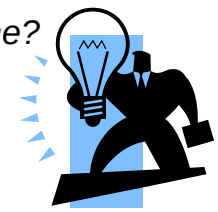
index = 2

- Ein Read-Only-Zeiger kann nur lesend auf Objekte zugreifen
  - Deklaration wie bei Variablen mit **`const`**
  - Zeiger selbst kann verändert werden (worauf er zeigt)
- Beispiel:

```
const int* cptrarr = arr;
```


Zwischenübung02: Zeiger & Referenzen

Worin unterscheiden sich die Ausgaben der folgenden Programme?



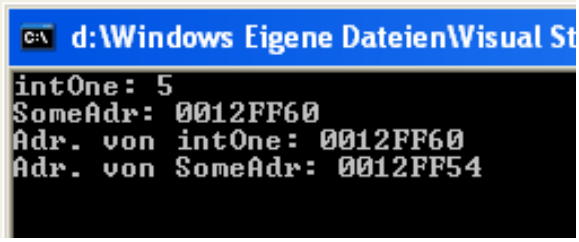
```
int main() {
   int intOne;
   int* SomeAdr = &intOne;

   intOne = 5;
   cout << "intOne: "
        << intOne << endl;
   cout << "SomeAdr: "
        << SomeAdr << endl;
   cout << "Adr. von intOne: "
        << &intOne << endl;
   cout << "Adr. von SomeAdr: "
        << &SomeAdr << endl;
   return 0;
}
```

```
int main() {
   int intOne;
   int& SomeRef = intOne;

   intOne = 5;
   cout << "intOne: "
        << intOne << endl;
   cout << "SomeRef: "
        << SomeRef << endl;
   cout << "Adr. von intOne: "
        << &intOne << endl;
   cout << "Adr. von SomeRef: "
        << &SomeRef << endl;
   return 0;
}
```

Zwischenübung02: Zeiger & Referenzen Lsg. (1)

```
int main() {
    int intOne;
    int* SomeAdr = &intOne;

    intOne = 5;
    cout << "intOne: "
         << intOne << endl;
    cout << "SomeAdr: "
         << SomeAdr << endl;
    cout << "Adr. von intOne: "
         << &intOne << endl;
    cout << "Adr. von SomeAdr: "
         << &SomeAdr << endl;
    return 0;
}
```

- Zeiger

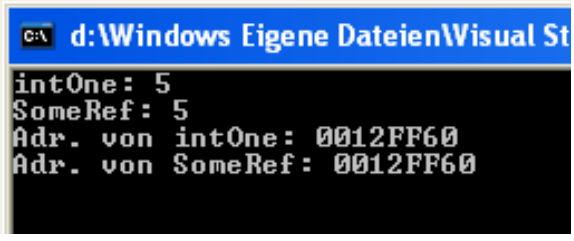


- Zeiger sind eigenständige Variablen
- Zeiger haben eine eigene Speicheradresse

Zwischenübung02: Zeiger & Referenzen Lsg. (2)

```
int main() {
    int intOne;
    int& SomeRef = intOne;

    intOne = 5;
    cout << "intOne: "
         << intOne << endl;
    cout << "SomeRef: "
         << SomeRef << endl;
    cout << "Adr. von intOne: "
         << &intOne << endl;
    cout << "Adr. von SomeRef: "
         << &SomeRef << endl;
    return 0;
}
```

- Referenzen



- Referenzen sind keine eigenständigen Variablen
- Referenzen haben keine eigene Speicheradresse

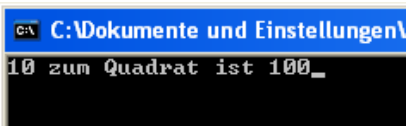
Call by Value: Ablauf

- Was passiert bei dieser Art der Variablenübergabe?

```
[...]
double quadrieren( double zahl ) {
  zahl = zahl * zahl;
  return zahl;
}


int main() {
  double basis = 10;
  double quadrat = quadrieren( basis );
  cout << basis << " zum Quadrat ist " << quadrat ;
  return 0;
}
```

verändert übergebene Variable **basis** nicht , da **zahl** eine Kopie ist

Ablauf:
1. Es wird eine Variable **zahl** angelegt
2. Es wird der Wert der Variablen **basis** in die Variable **zahl** kopiert
3. Funktion wird ausführt (Veränderung der Variablen **zahl**)
4. Rückgabe des Ergebnisses an die Stelle des Funktionsaufrufes



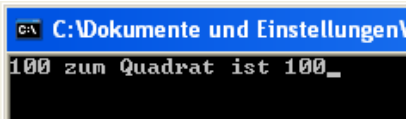
Call by Reference: Ablauf

- Was passiert bei dieser Art der Variablenübergabe?

```
[...]
double quadrieren( double& zahl ) {
  zahl = zahl * zahl;
  return zahl;
}

int main() {
  double basis = 10;
  double quadrat = quadrieren( basis );
  cout << basis << " zum Quadrat ist " << quadrat ;
  return 0;
}
```

!!! Einzige Veränderung !!!

Ablauf:
1. Die übergebene Variable **basis** wird in **zahl** umbenannt
2. Funktion wird ausführt (Veränderung der Variablen **zahl / basis**)
3. Rückgabe des Ergebnisses an die Stelle des Funktionsaufrufes



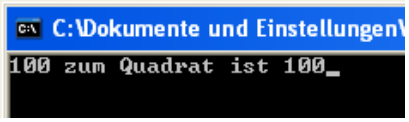
Call by Pointer: Ablauf

- Was passiert bei dieser Art der Variablenübergabe?

```
[...]
double quadrieren( double* zahl ) {
  *zahl = *zahl * *zahl;
  return *zahl;
}


int main() {
  double basis = 10;
  double quadrat = quadrieren( &basis );
  cout << basis << " zum Quadrat ist " << quadrat ;
  return 0;
}
```

Adressvariable

Ablauf:
1. Es wird eine AdressVariable **zahl** angelegt
2. Es wird die Adresse der Variablen **basis** in die Adressvariable **zahl** kopiert
3. Funktion wird ausführt (Zugriff auf die Variable **basis** über den Zeiger **zahl**)
4. Rückgabe des Ergebnisses an die Stelle des Funktionsaufrufes



Übergabe eines Arrays: Beispiel

- Was passiert bei der Übergabe eines Arrays an eine Funktion?

```
[...]
long summe( long arr[], int anzahl ) {
  long ergebnis = 0;
  for( int i = 0; i < anzahl; i++ ) {
    ergebnis = ergebnis + arr[i];
  }
  return ergebnis;
}

int main() {
  long daten[] = {546,465,99,86,598,655,86,6,9,974};
  cout << "Die Summe der Elemente ist " << summe( daten, 10 );
  return 0;
}
```

Äquivalent zu: **long* arr** → Zeigervariable

Funktion weiß nicht, wie groß das Array ist, dies muss übergeben werden oder festgelegt sein

Äquivalent zu: ***(arr + i)**

Übergabe einer Adresse

Zwischenübung03: Übergabeparameter

1. Was ist die Ausgabe des folgenden Programms?
2. Wie muss dass Programm verändert werden, damit das vertauschen funktioniert?

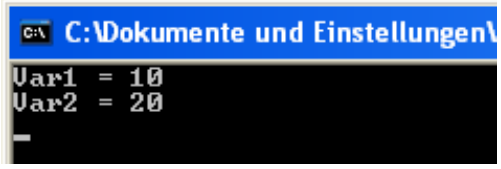
```
[...]
void vertauschen( int var1, int var2 ) {
  int temp = 0;
  temp = var1;
  var1 = var2;
  var2 = temp;
}

int main() {
  int var1 = 10;
  int var2 = 20;
  vertauschen( var1, var2 );
  cout << "Var1 = " << var1 << endl;
  cout << "Var2 = " << var2 << endl;
  return 0;
}
```

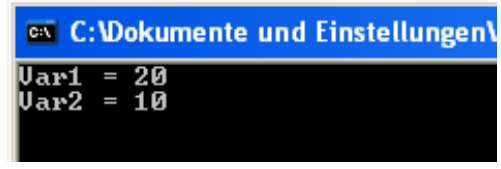

Zwischenübung03: Übergabeparameter Lsg.

1. Was ist die Ausgabe des folgenden Programms?
2. Wie muss dass Programm verändert werden, damit das vertauschen funktioniert?

```
[...]
void vertauschen( int &var1, int &var2 ) {
  int temp = 0;
  temp = var1;
  var1 = var2;
  var2 = temp;
}

int main() {
  int var1 = 10;
  int var2 = 20;
  vertauschen( var1, var2 );
  cout << "Var1 = " << var1 << endl;
  cout << "Var2 = " << var2 << endl;
  return 0;
}
```

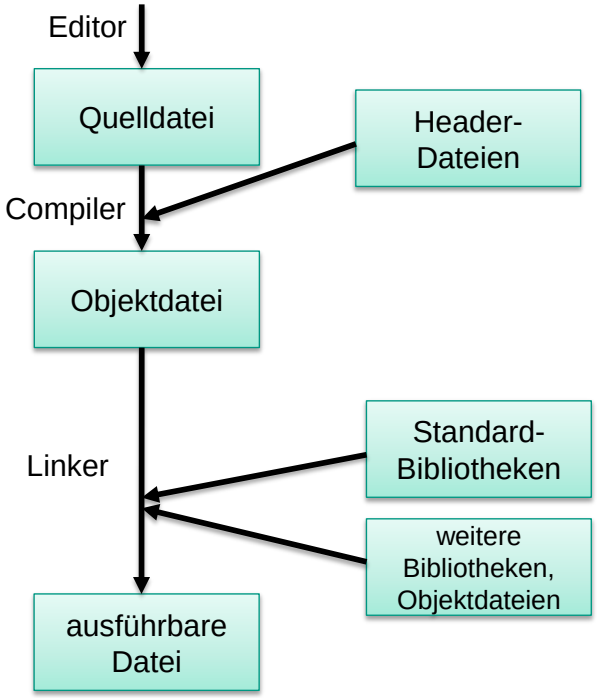
Übergabe als Referenzen!





Erstellen eines C++ Programms

- Editor dient zur Eingabe des Programmcodes
- Mehrere Quelldateien & Headerdateien sind möglich
- Objektdatei enthält den Maschinencode
- Linker führt alle Bausteine zur einer ausführbaren Datei zusammen
- Übliche Dateiendungen:
  - `*.cpp` >> Quelldatei
  - `*.h` >> Headerdatei
  - `*.obj` >> Objektdatei
  - `*.exe` >> ausführbare Datei



Header-Dateien und Linking

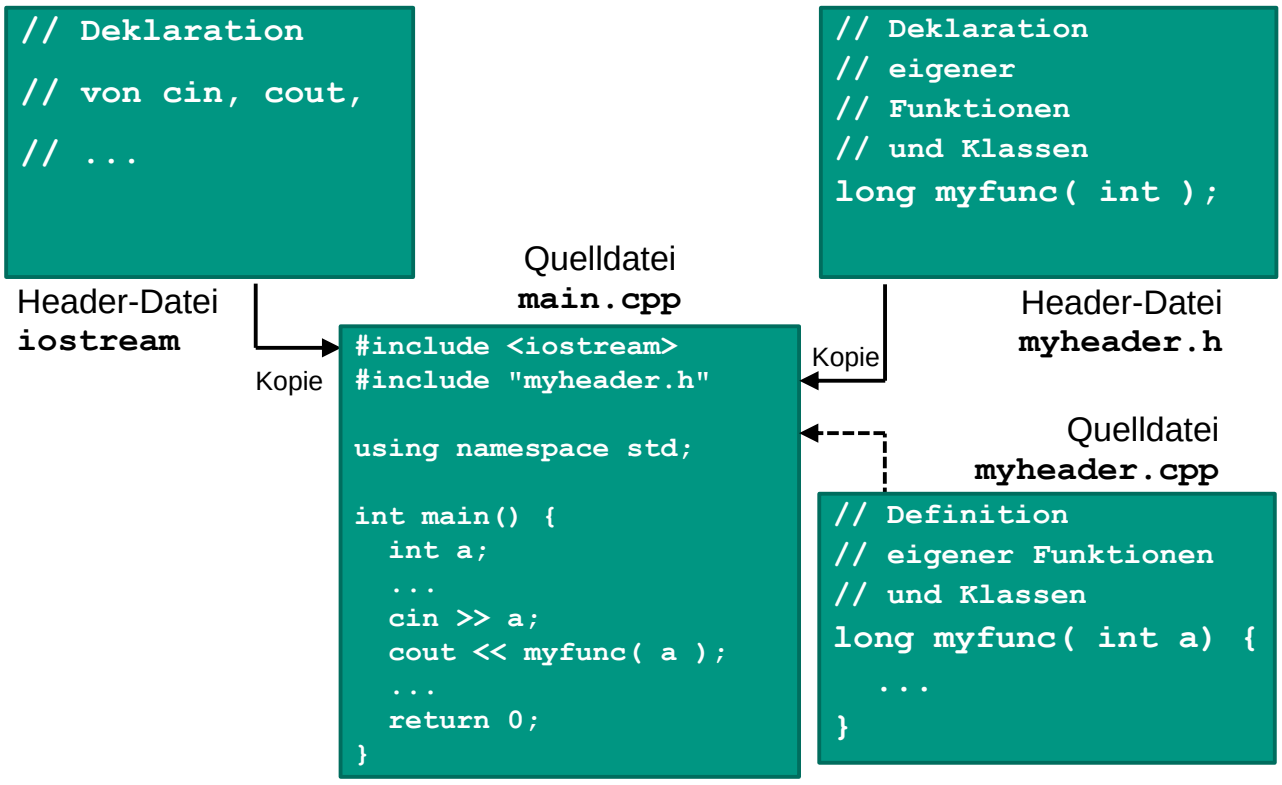
- Aufteilung des Codes in übersichtliche Module
  - Aufteilung in mehrere Quellcode-Dateien
- Zugriff über Prototypen in Header-Dateien
  - Compiler übersetzt einzelne Dateien – Linker bindet sie zusammen
  - Header-Dateien werden in die anderen Quelldateien inkludiert
- Schutz vor mehrfachen Einbinden der Dateien
  - Verwendung von Präprozessor-Direktiven

```
#ifndef _MYHEADER_
#define _MYHEADER_

[...]

#endif
```

Wird vom Compiler / Präprozessor verarbeitet

Inhalt der Header-Datei

Funktionen und Header-Dateien



Referenz & Ausblick

- Kompendium: Kapitel 2 - Variablen, Zeiger und Arrays
  Kapitel 4 - Funktionen
  Kapitel 6 - Gültigkeitsbereiche
- Tutorium: Aufgabe 5 – 10
- Wie können zur Laufzeit mehrere Variablen angelegt und wieder gelöscht werden?
- Wie kann ich die Größe eines Arrays während des Programmablaufs verändern?
- Wie kann ich reale Objekte oder dem Menschen naheliegende Gedankenkonstrukte in Programmstrukturen möglichst sinngemäß abbilden?
- …

Vielen Dank für Ihre Aufmerksamkeit

Harald Bucher
Karlsruher Institut für Technology (KIT) – ITIV
harald.bucher@kit.edu