Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Übung03: Informationstechnik (IT)

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Tobias Schwalb / Michael Tansella

Institut für Technik der Informationsverarbeitung (ITIV)

Teil2: Funktionen, Header-Dateien, Gültigkeitsbereiche

Inhalt: Übung03

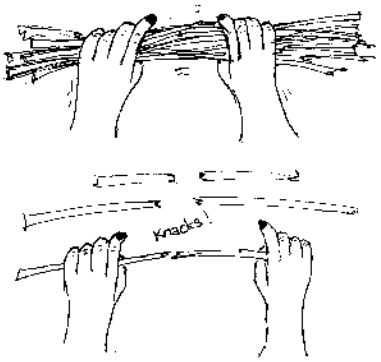
Karlsruher Institut für Technologie

1. Besprechung Übungsaufgaben 2.01-2.06
2. Funktionen
3. Header-Dateien
4. Gültigkeitsbereiche

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



Teile und Herrsche

- Vorgehensweise
  1. Teile: Zerlege das Problem in Teilprobleme, bis die Teilprobleme so klein sein, dass sie einfach zu lösen sind
  2. Herrsche: Löse die Teilprobleme (in einzelnen Funktionen)
  3. Herrsche: Setze die Teilprobleme wieder zu einer Gesamtlösung zusammen

- Beispiel: Vorbereitung auf eine Klausur
  1. Teile: Übungsaufgaben, Vorlesungsinhalt, Praktikumsinhalte, Klausuren aus vorherigen Semestern
  2. Herrsche: Bearbeitung der Übungsaufgaben, Durchlesen mit Notizen zum Vorlesungsinhalt, Proberechnen der Klausuren aus vorherigen Semestern
  3. Herrsche: Schreiben einer Zusammenfassung / Formelsammlung

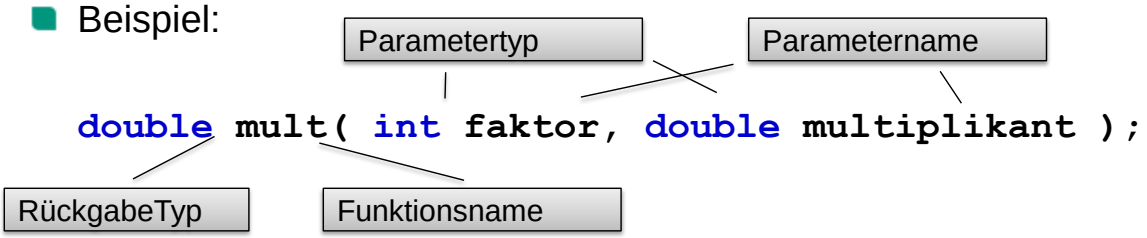
- Erfordert Übung !!

Deklaration von Funktionen

- Muss wie eine Variable deklariert (bekanntgemacht) werden
- Bekanntmachung: Namen, Übergebene Daten & Ergebnistyp

- Syntax: `rueckgabeTyp funktionName( parameterTyp1 parameter1, parameterTyp2 parameter2, ... );`
  - Typen können beliebige Variablentypen sein
  - Übergabeparameter sind optional
  - Sollte kein rückgabeTyp vorhanden sein: `void` verwenden

- Beispiel:



Definition einer Funktion

- Beschreibung der Funktionalität einer Funktion

- Syntax:

```
rueckgabeTyp funktionName( 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 );
...
```

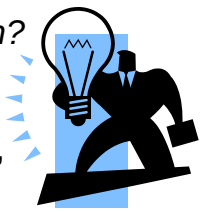
Werte werden übergeben

Ergebnis wird zurückgegeben

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


Zwischenübung08: 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 `cubes()` besitzt einen Parameter `n` vom Typ `int`. Sie summiert die ersten `n` positiven Zahlen „hoch drei“ auf, berechnet also den Wert $1^3+2^3+ \ldots +n^3$, und liefert das Ergebnis als Return-Wert zurück.

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

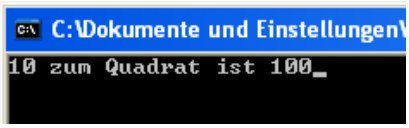
Call by Value: Ablauf

- 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 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 Value: Zusammenfassung

- Werte der übergebenen Variablen werden intern in neue Variablen kopiert, welche innerhalb der Funktion verwendet werden
- Die Änderungen an einer Variablen innerhalb der Funktion betreffen die beim Aufruf verwendete Variable nicht
- Nachteil: die Variable muss in eine Neue kopiert werden
  - kostet Speicher und Rechenzeit
  - Funktion kann nicht auf die übergebenen Variablen verändernd zugreifen, da die Funktion nur Kopien erhält
- Vorteil:
  - Komplette Trennung der Variablen beim Funktionsaufruf

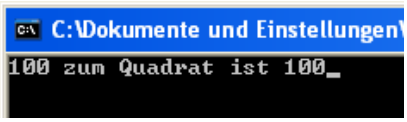
Call by Reference: Ablauf

- 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 Reference: Zusammenfassung

- Werte der übergebenen Variablen werden nicht kopiert, es wird eine Referenz erzeugt – ein anderer Name für die gleiche Variable
- Die Änderungen an einer Variablen innerhalb der Funktion betreffen die beim Aufruf verwendete Variable
- Nachteil:
  - Evtl. unbeabsichtigte Veränderung von Variablen
  - Bei einer Veränderung ist der ursprüngliche Wert überschrieben
- Vorteil:
  - Kostet keinen Speicher und keine Rechenzeit für eine Kopie der Variablen
  - Funktion kann auf die übergebenen Variablen verändernd zugreifen

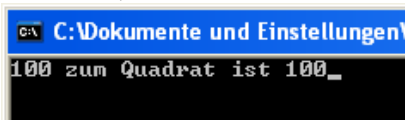
Call by Pointer: Ablauf

- 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



Call by Pointer: Zusammenfassung

- Die Adresse der übergebenen Variable wird in einer Adressvariablen (als Übergabeparameter) gespeichert
- Veränderungen an einer Variablen (über Dereferenzierung innerhalb der Funktion) betreffen die beim Aufruf verwendete Variable
- Nachteil:
  - Kostet Speicher und Rechenzeit
  - Umständlicher Zugriff über Zeiger (Dereferenzierung)
- Vorteil:
  - Funktion kann auf die übergebenen Variablen verändernd zugreifen
  - Zeiger kann verändert werden
  - Übergabe komplexerer Strukturen möglich (Arrays & siehe Übung 4)

Übergabe eines Arrays: Beispiel

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

Äquivalent zu: `long* arr` → Zeigervariable

```
[...]
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;
}
```

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

Äquivalent zu: `*( arr + i )`

Übergabe einer Adresse

Übergabe eines Arrays: Zusammenfassung

- Bei der Übergabe eines Arrays an eine Funktion wird nur die Adresse des ersten Elementes an die Funktion übergeben
- Die Funktion arbeitet mit Zeigern und greift über die Dereferenzierung auf das Array zu
  - Keine Kopie: Veränderung des übergebenen Arrays möglich
- Nachteil:
  - Funktion kennt nicht die Größe des Arrays und kann diese auch nicht ermitteln
- Vorteil:
  - schnelle Übergabe (nur eine Adresse) – auch bei großen Arrays
  - Funktion kann auf das übergebene Array verändernd zugreifen

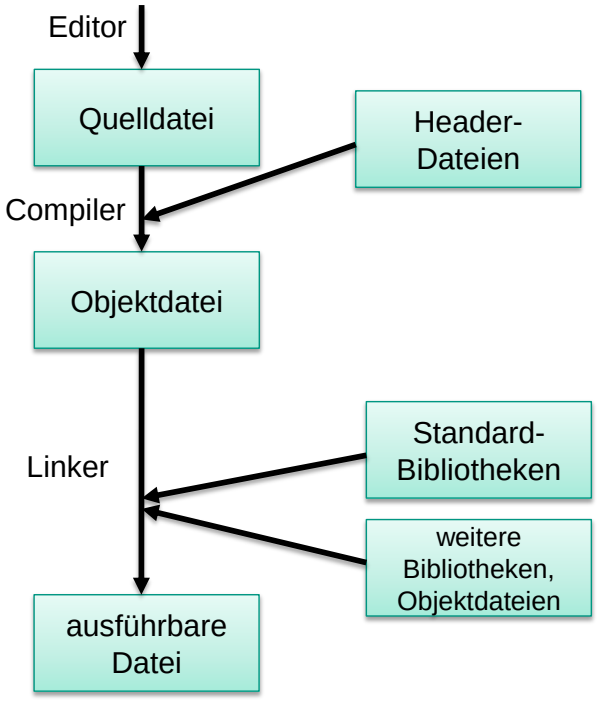
Zwischenübung10: Übergabeparameter

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;
}
```


Erstellen eines C++ Programms (ÜB01)

- Editor dient zur Eingabe des Programmcodes
- Mehrere Quelldateien & Headerdateien sind möglich
- Objektdatei enthält den Maschienencode
- 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 verarbeitet

Inhalt der Header-Datei

Funktionen und Header-Dateien

Header-Datei **iostream**

```
// Deklaration
// von cin, cout,
// ...
```

Kopie

Quelldatei **main.cpp**

```
#include <iostream>
#include "myheader.h"

int main() {
  int a;
  ...
  cin >> a;
  cout << myfunc( a );
  ...
  return 0;
}
```

Header-Datei **myheader.h**

```
// Deklaration
// eigener
// Funktionen
// und Klassen
long myfunc( int );
```

Kopie

Quelldatei **myheader.cpp**

```
// Definition
// eigener Funktionen
// und Klassen
long myfunc( int ) {
  ...
}
```

Gültigkeitsbereiche

- Problemstellung:
  - Welche Variable existiert zu welchem Zeitpunkt?
  - Wo ist welche Variable sichtbar?
  - Wann kann ich auf welche Variable zugreifen?
- Lösung: Gültigkeitsbereiche
  - Bereich, in dem eine Variable / Objekt im Speicher existiert
  - Abhängig vom Ort und Art der Variablen
- Generelle Unterscheidung
  - Globale Variablen
  - Lokale Variablen
- Regel: Variablen immer so lokal wie möglich deklarieren

Globaler Gültigkeitsbereich (1)

- Globales Objekt oder auch global deklariertes Objekt
  - Objekt wird bei Programmstart erzeugt, belegt während der ganzen Ausführungszeit Speicher, erst bei Programmende wieder gelöscht
  - Deklaration außerhalb der geschweiften Klammern { }
- Vorteil:
  - Objekt kann überall in der gleichen Quelldatei genutzt werden
- Nachteile:
  - Objekt belegt ständig Speicherplatz
  - Verlust der Übersicht bei größeren Programmen
- Verwendung globaler Variablen minimieren & begründen

Globaler Gültigkeitsbereich (2)

- Beispiel:

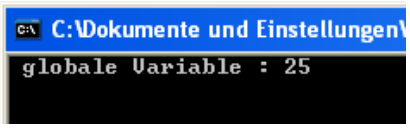
```
[...]
int zahl;
void machWas();

int main() {
  zahl = 25;
  machWas();
  return 0;
}

void machWas() {
  cout << " globale Variable : " << zahl << endl ;
}
```

Globale Variable

Funktionsdeklaration

Hauptprogramm

Funktionsdefinition



Lokaler Gültigkeitsbereich (1)

- Lokales Objekt oder auch lokal deklariertes Objekt
  - wird in einem vom Blockoperator { } umrandeten Bereich deklariert
  - Kann sich innerhalb einer Funktionsdefinition, Schleife (oder einer anderen Kontrollstruktur) oder innerhalb einer Klasse befinden
- Existenz
  - Variable wird erst angelegt, wenn ihre Deklaration erreicht wird
  - lebt nur solange, wie sich der Programmablauf innerhalb der { }-Klammern befindet – außerhalb unbekannt / nicht verwendbar
- Vorteile:
  - Speicherplatz wird nur dann verbraucht, wenn er benötigt wird
- Nachteile:
  - Variablen / Werte müssen übergeben werden, wenn sie in anderen Funktion verwendet werden sollen (siehe Folie 11-18)

Lokaler Gültigkeitsbereich (2)

- Beispiel:

```
[...]
void machWas();

int main() {
  int zahl;
  zahl = 25;
  machWas();
  zahl = 15;
  return 0;
}

void machWas() {
  cout << "globale Variable: " << zahl << endl;
}
```

Funktionsdeklaration

Hauptprogramm

Lokale Variable

Zugriff innerhalb der Funktion auf die lokale Variable möglich

Funktionsdefinition

Zugriff außerhalb der Funktion auf die Variable nicht möglich → Variable unbekannt

Syntaxfehler: Programm wird nicht kompiliert

Speicherklassen-Spezifizierer

- Statische Objekte = permanente Lebensdauer
  - Bleibt nur innerhalb des Blocks sichtbar
  - Speicherbereich / Wert bleibt allerdings erhalten
  - Kennzeichnung durch das Schlüsselwort **`static`**

Beispiel: `static long summe;`

- Globale Objekte = Definition außerhalb einer Funktion
  - Informationsaustausch ohne Argumente
  - Überall in den Quelldateien eines Programms zugreifbar
    - Effekt auf das gesamte Programm → sparsam verwenden
  - Importieren von globalen Variablen aus anderen Quelldateien über den Speicherklassen-Spezifizierer **`extern`**

Beispiel: `extern long linie;`

Zwischenübung11: Verdeckung

Was ist die Ausgabe des folgenden Programms?

```
[...]
int main() {
  int i = 100;

  for( int i = 0; i <= 5; i++ ) {
    int i = 1;
    i += 2;
    cout << i << ",";
  }

  cout << endl << endl;
  cout << "i = " << i;

  return 0;
}
```


Referenz & Ausblick

- Kompendium: Kapitel 4 – Funktionen
- Tutorium: Aufgabe 9 – 14

- Wie können zur Laufzeit mehrere Variablen anlegt und wieder gelöscht werden?
- Wie kann ich die Größe eines Arrays während des Programmauflaufs verändern?
- Wie kann ich Zeichenketten in C++ anlegen?
- Wie kann ich die STL (Standard Template Library) *string* benutzen?
- Welche nützliche Fähigkeiten bzw. Methoden besitzt die Klasse *string*?
- …