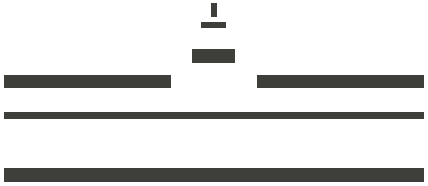


Westfälische
Wilhelms-Universität
Münster

Übung zur Vorlesung Wissenschaftliches Rechnen — Sommersemester 2012

Auffrischung zur Programmierung in C++, 1. Teil

living.knowledge
WWU Münster

Sebastian Westerheide

11. April 2012

Organisatorisches

- Ausgabe der Übungszettel jeweils Dienstags
- Erster Übungszettel am 17.04.

Organisatorisches

- Ausgabe der Übungszettel jeweils Dienstags
- Erster Übungszettel am 17.04.

- Abgabe Dienstags, pünktlich 10:00 Uhr, Briefkasten 89
- Abgabe in festen Zweiergruppen (jetzt Partnersuche)

Organisatorisches

- Ausgabe der Übungszettel jeweils Dienstags
- Erster Übungszettel am 17.04.

- Abgabe Dienstags, pünktlich 10:00 Uhr, Briefkasten 89
- Abgabe in festen Zweiergruppen (jetzt Partnersuche)

- Schriftliche Abgabe, Code zusätzlich per e-mail
- Code muss kompilieren, sonst wird er nicht bewertet

Organisatorisches

- Ausgabe der Übungszettel jeweils Dienstags
- Erster Übungszettel am 17.04.

- Abgabe Dienstags, pünktlich 10:00 Uhr, Briefkasten 89
- Abgabe in festen Zweiergruppen (jetzt Partnersuche)

- Schriftliche Abgabe, Code zusätzlich per e-mail
- Code muss kompilieren, sonst wird er nicht bewertet

- e-mail an `sebastian.westerheide@uni-muenster.de`

Eingliederung C++

Klassifikation höherer Programmiersprachen nach Paradigmen:

- Deklarative Programmiersprachen (funktionale und logische P.)

Eingliederung C++

Klassifikation höherer Programmiersprachen nach Paradigmen:

- Deklarative Programmiersprachen (funktionale und logische P.)
- Imperative Programmiersprachen
  - *Grundidee*: Programm bewirkt Speichertransformation
  - Rein prozedurale Programmiersprachen

    z.B. C, Fortran, Pascal, Basic, Cobol, Algol, . . . und Matlab
  - Objektorientierte Programmiersprachen

    z.B. C++, Java, C#, Eiffel, Smalltalk, . . . und neuerdings auch Matlab

Eingliederung C++

Klassifikation höherer Programmiersprachen nach Paradigmen:

- Deklarative Programmiersprachen (funktionale und logische P.)
- Imperative Programmiersprachen
  - *Grundidee*: Programm bewirkt Speichertransformation
  - Rein prozedurale Programmiersprachen
    z.B. C, Fortran, Pascal, Basic, Cobol, Algol, . . . und Matlab
  - Objektorientierte Programmiersprachen
    z.B. C++, Java, C#, Eiffel, Smalltalk, . . . und neuerdings auch Matlab

C++:

- Imperative Programmiersprache
- Hybrider Ansatz:
  Prozedurale und objektorientierte Programmierung möglich

Überblick über die ersten beiden Übungsstunden

Heute:

- Auffrischung zur prozeduralen Programmierung in C++
  - Grundlegende Syntax
  - Basisdatentypen
  - Wichtige Operatoren
  - Kontrollstrukturen
  - Prozeduren
  - Wertparameter vs. Referenzparameter, Pointer und Referenzen

Nächste Woche:

- Auffrischung zur objektorientierten Programmierung in C++

Grundlegende Syntax

- Syntax für Anweisungen: ‹Anweisung› ;
- Blöcke durch geschweifte Klammern: { ... }

Grundlegende Syntax

- Syntax für Anweisungen: ‹Anweisung› ;
- Blöcke durch geschweifte Klammern: { ... }

```
#include <iostream>

// Hauptroutine
int main()
{
  // Ausgeben des Grusses
  std::cout << "Hallo Welt!" << std::endl;
  // Programm beenden
  return 0;
}
```

Grundlegende Syntax

- Syntax für Anweisungen: ‹Anweisung› ;
- Blöcke durch geschweifte Klammern: { ... }
- Zeilenkommentar: // Kommentar
- Blockkommentare: /* Potentiell mehrzeiliger Kommentar */

```
#include ‹iostream›

// Hauptroutine
int main()
{
  // Ausgeben des Grusses
  std::cout ‹‹ "Hallo Welt!" ‹‹ std::endl;
  // Programm beenden
  return 0;
}
```

Basisdatentypen

Name	Wertebereich	Genauigkeit
bool	false (0) bis true (1)	
(signed) char	-128 bis 127	
unsigned char	0 bis 255	
(signed) short int	-32.768 bis 32.767	
unsigned short int	0 bis 65.535	
(signed) (long) int	-2.147.483.648 bis 2.147.483.647	
unsigned (long) int	0 bis 4.294.967.295	
float	3,4E-38 bis 3,4E+38	8 Stellen
double	1,7E-308 bis 1,7E+308	16 Stellen

- Repräsentation von Booleschen Zuständen, Zeichen, ganzen Zahlen und reellen Zahlen in Fließkommadarstellung

Basisdatentypen

Name	Wertebereich	Genauigkeit
bool	false (0) bis true (1)	
(signed) char	-128 bis 127	
unsigned char	0 bis 255	
(signed) short int	-32.768 bis 32.767	
unsigned short int	0 bis 65.535	
(signed) (long) int	-2.147.483.648 bis 2.147.483.647	
unsigned (long) int	0 bis 4.294.967.295	
float	3,4E-38 bis 3,4E+38	8 Stellen
double	1,7E-308 bis 1,7E+308	16 Stellen

- Repräsentation von Booleschen Zuständen, Zeichen, ganzen Zahlen und reellen Zahlen in Fließkommadarstellung
- `unsigned short int i = 13; int j = -45000; j+= i; float eps; eps = 1E-10;`

Basisdatentypen

Name	Wertebereich	Genauigkeit
bool	false (0) bis true (1)	
(signed) char	-128 bis 127	
unsigned char	0 bis 255	
(signed) short int	-32.768 bis 32.767	
unsigned short int	0 bis 65.535	
(signed) (long) int	-2.147.483.648 bis 2.147.483.647	
unsigned (long) int	0 bis 4.294.967.295	
float	3,4E-38 bis 3,4E+38	8 Stellen
double	1,7E-308 bis 1,7E+308	16 Stellen

- Repräsentation von Booleschen Zuständen, Zeichen, ganzen Zahlen und reellen Zahlen in Fließkommadarstellung
- `unsigned short int i = 13; int j = -45000; j+= i; float eps; eps = 1E-10; char ch = 'A'; ch++; // ch wird 'B'`

Basisdatentypen

Name	Wertebereich	Genauigkeit
bool	false (0) bis true (1)	
(signed) char	-128 bis 127	
unsigned char	0 bis 255	
(signed) short int	-32.768 bis 32.767	
unsigned short int	0 bis 65.535	
(signed) (long) int	-2.147.483.648 bis 2.147.483.647	
unsigned (long) int	0 bis 4.294.967.295	
float	3,4E-38 bis 3,4E+38	8 Stellen
double	1,7E-308 bis 1,7E+308	16 Stellen

- Repräsentation von Booleschen Zuständen, Zeichen, ganzen Zahlen und reellen Zahlen in Fließkommadarstellung
- ```
  unsigned short int i = 13; int j = -45000; j+= i; float eps; eps = 1E-10;
  char ch = 'A'; ch++; // ch wird 'B'
  bool bedingung = true; bool bed2 = (j > eps); // bed2 wird false
  ```
  ```

Wichtige Operatoren (1/2):

- Arithmetische Operatoren für zwei Zahlen a und b
  - a += b Kurzform für a = a + b
  - a −= b Kurzform für a = a − b
  - a *= b Kurzform für a = a * b
  - a /= b Kurzform für a = a / b
  - a %= b Kurzform für a = a % b (nur für ganze Zahlen)
  - a++ a inkrementieren (a = a + 1)
  - a−− a dekrementieren (a = a − 1)

Wichtige Operatoren (2/2):

- Vergleichsoperatoren (relationale Operatoren) zum Vergleich von zwei Zahlen `a` und `b`
  - `a == b` Ist `a` gleich `b`?
  - `a != b` Ist `a` ungleich `b`?
  - `a < b` Ist `a` kleiner als `b`?
  - `a > b` Ist `a` größer als `b`?
  - `a <= b` Ist `a` kleiner oder gleich `b`?
  - `a >= b` Ist `a` größer oder gleich `b`?

Wichtige Operatoren (2/2):

- Vergleichsoperatoren (relationale Operatoren) zum Vergleich von zwei Zahlen `a` und `b`
  - `a == b` Ist `a` gleich `b`?
  - `a != b` Ist `a` ungleich `b`?
  - `a < b` Ist `a` kleiner als `b`?
  - `a > b` Ist `a` größer als `b`?
  - `a <= b` Ist `a` kleiner oder gleich `b`?
  - `a >= b` Ist `a` größer oder gleich `b`?
- Boolesche Operatoren zur Verknüpfung von Booleschen Variablen oder Ausdrücken `a` und `b`
  - `!a` bzw. `not a` Ist `a` falsch?
  - `a && b` bzw. `a and b` Sind `a` und `b` wahr?
  - `a || b` bzw. `a or b` Ist `a` oder `b` wahr?

Kontrollstrukturen

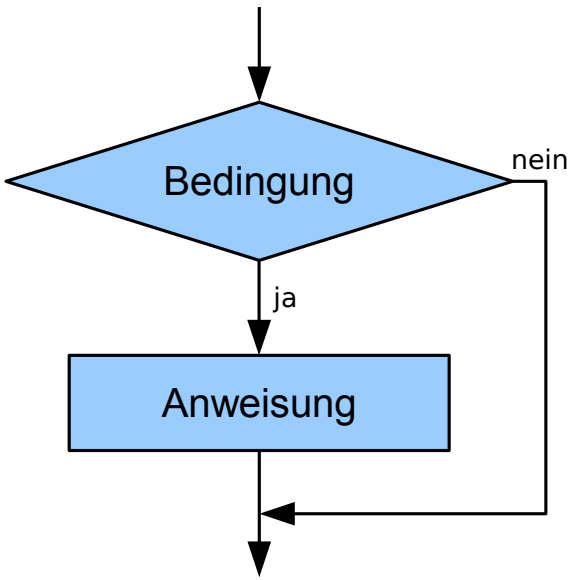
- Mit Kontrollstrukturen kann der Programmablauf in Abhängigkeit vom Wert einer oder mehreren Variablen gesteuert werden

Kontrollstrukturen

- Mit Kontrollstrukturen kann der Programmablauf in Abhängigkeit vom Wert einer oder mehreren Variablen gesteuert werden
- Es gibt folgende Möglichkeiten in C++:
  - Auswahlanweisungen:
    - Bedingte Anweisung: if-else - Anweisung
    - Fallunterscheidung: switch - Anweisung
  - Schleifen (Iterationsanweisungen):
    - Kopfgesteuerte Schleife: while - Schleife
    - Fußgesteuerte Schleife: do-while - Schleife
    - Zählschleife: for - Schleife

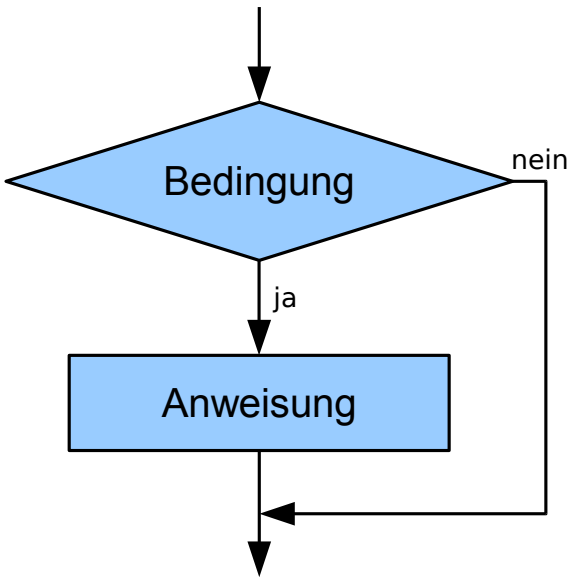
Bedingte Anweisung

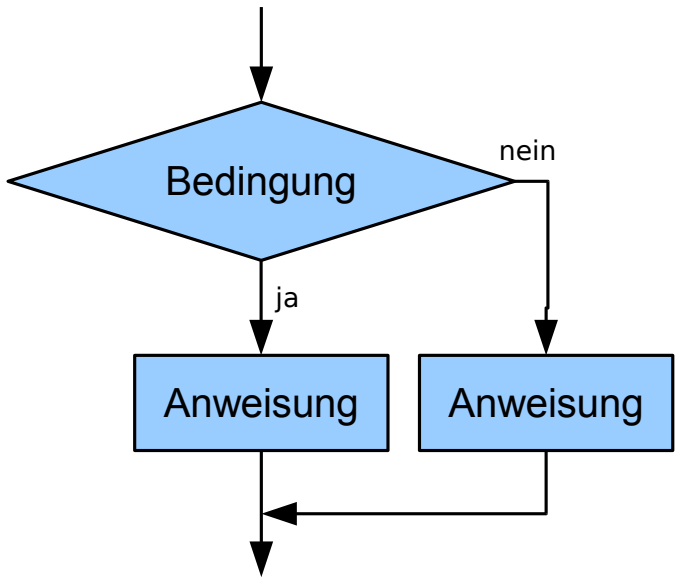
- if (‹Bedingung›) {‹Anweisung›}



Bedingte Anweisung

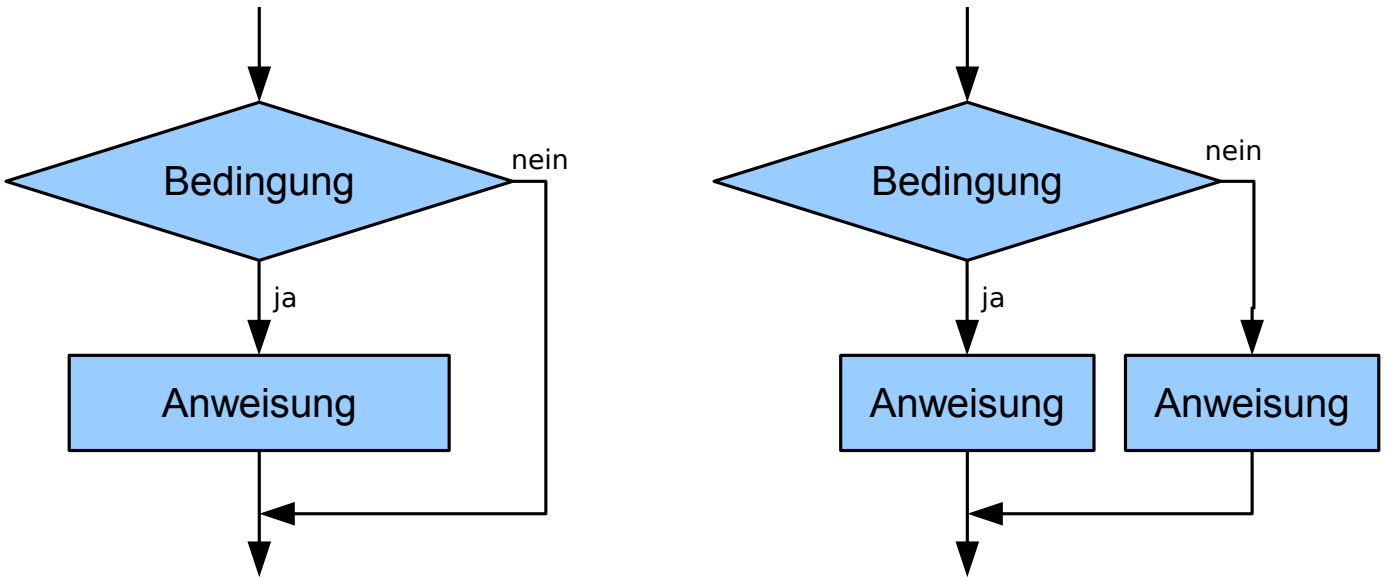
- if (‹Bedingung›) {‹Anweisung›}
- if (‹Bedingung›) {‹Anweisung›} else {‹Anweisung›}





Bedingte Anweisung

- if (‹Bedingung›) {‹Anweisung›}
- if (‹Bedingung›) {‹Anweisung›} else {‹Anweisung›}



- ‹Anweisung› kann einen Block von Anweisungen enthalten

Beispiel

```
#include <iostream>

int main()
{
  std::cout << "Geben Sie eine Zahl zwischen 5 und 10 ein"
            << std::endl;

  // einlesen einer Zahl
  int zahl;
  std::cin >> zahl;

  // pruefen
  if (zahl >= 5 && zahl <= 10)
  {
    std::cout << "Die Zahl ist super" << std::endl;
  }
  else
  {
    std::cout << "Zahl zu klein oder zu gross" << std::endl;
  }

  return 0;
}
```

Schleifen (1/2)

- Wiederholung bestimmter Anweisungen in Abhängigkeit einer Bedingung
- Schleifenkopf `( ... )`, Schleifenrumpf `{ ... }`

Schleifen (1/2)

- Wiederholung bestimmter Anweisungen in Abhängigkeit einer Bedingung
- Schleifenkopf `( ... )`, Schleifenrumpf `{ ... }`
- while - Schleife:
  - while (‹Bedingung›) {‹Anweisung›}
  - Solange ‹Bedingung› erfüllt ist, führe ‹Anweisung› aus

Schleifen (1/2)

- Wiederholung bestimmter Anweisungen in Abhängigkeit einer Bedingung
- Schleifenkopf (...), Schleifenrumpf { ... }
- while - Schleife:
  - while (‹Bedingung›) {‹Anweisung›}
  - Solange ‹Bedingung› erfüllt ist, führe ‹Anweisung› aus
- do-while - Schleife:
  - do {‹Anweisung›} while (‹Bedingung›) ;
  - Führe ‹Anweisung› aus, bis ‹Bedingung› nicht mehr erfüllt
  - Überprüfung von ‹Bedingung› erst nach jedem Durchlauf

Schleifen (1/2)

- Wiederholung bestimmter Anweisungen in Abhängigkeit einer Bedingung
- Schleifenkopf (...), Schleifenrumpf { ... }
- while - Schleife:
  - while (‹Bedingung›) {‹Anweisung›}
  - Solange ‹Bedingung› erfüllt ist, führe ‹Anweisung› aus
- do-while - Schleife:
  - do {‹Anweisung›} while (‹Bedingung›) ;
  - Führe ‹Anweisung› aus, bis ‹Bedingung› nicht mehr erfüllt
  - Überprüfung von ‹Bedingung› erst nach jedem Durchlauf
- Jeweils Endlosschleife, falls ‹Bedingung› nicht irgendwann durch ‹Anweisung› zu false ausgewertet wird

Schleifen (2/2)

- for - Schleife:
  - for (‹ Init › ; ‹Bedingung› ; ‹ Reinit ›) {‹Anweisung›}
  - ‹ Init ›
    Anweisung, die einer Variable einen Anfangswert zuweist
  - ‹Bedingung›
    Solange true, wird ‹Anweisung› im Schleifenrumpf abgearbeitet
  - ‹ Reinit ›
    Legt fest, wie die Variable aus ‹ Init › nach jedem Schleifendurchlauf verändert wird

Schleifen (2/2)

- for - Schleife:
  - for (‹ Init › ; ‹Bedingung› ; ‹ Reinit ›) {‹Anweisung›}
  - ‹ Init ›
    Anweisung, die einer Variable einen Anfangswert zuweist
  - ‹Bedingung›
    Solange true, wird ‹Anweisung› im Schleifenrumpf abgearbeitet
  - ‹ Reinit ›
    Legt fest, wie die Variable aus ‹ Init › nach jedem Schleifendurchlauf verändert wird
- Endlosschleife, falls ‹Bedingung› nicht irgendwann durch ‹ Reinit › oder ‹Anweisung› zu false ausgewertet wird

Beispiel — Summe von 1 bis 10

- while - Schleife

```
int summe = 0;
int i = 0;
while (i <= 10) { summe = summe + i; i++; }
```

- do-while - Schleife

```
int summe = 0;
int i = 0;
do { summe = summe + i; i++; } while (i <= 10);
```

- for - Schleife

```
int summe = 0;
for (int i=0; i<=10; i++) { summe = summe + i; }
```

Prozeduren

- Prozeduren (Funktionen) sind Unterprogramme, d.h. Blöcke, die Teilprobleme einer größeren Aufgabe lösen
- Jede Prozedur hat einen eindeutigen Namen, mit dessen Hilfe sie aufgerufen werden kann
- Prozeduren können einen Wert an das aufrufende Programm zurückgeben und selbst weitere Prozeduren aufgerufen

Prozeduren

- Prozeduren (Funktionen) sind Unterprogramme, d.h. Blöcke, die Teilprobleme einer größeren Aufgabe lösen
- Jede Prozedur hat einen eindeutigen Namen, mit dessen Hilfe sie aufgerufen werden kann
- Prozeduren können einen Wert an das aufrufende Programm zurückgeben und selbst weitere Prozeduren aufgerufen

Vorteile:

- Wiederkehrende Berechnungen müssen nur einmal programmiert werden
- Bessere Lesbarkeit durch Aufteilen des Programms

Beispiel

```
#include <iostream>

// Prototyp fuer die Prozedur quadrat
// quadrat(x) berechnet x*x
double quadrat (double x);

// Implementierung der Prozedur quadrat
double quadrat (double x)
{
  return x*x;
}

int main()
{
  for (int i=0; i<10; i++)
  {
    std::cout << quadrat(i) << std::endl;
  }
}
```

Syntax

- Rueckgabetyp FktName (Parameter1, Parameter2, ..., ParameterN) {‹Anweisung›}
- Prozeduren ohne Rückgabewert durch Rückgabetyp void
- return ‹Wert›; beendet die Prozedur und gibt ‹Wert› an den übergeordneten Programmteil zurück
- Parameter bestehen jeweils aus dem Datentyp und einem Variablennamen

Syntax

- Rueckgabetyp FktName (Parameter1, Parameter2, ..., ParameterN) {‹Anweisung›}
- Prozeduren ohne Rückgabewert durch Rückgabetyp void
- return ‹Wert›; beendet die Prozedur und gibt ‹Wert› an den übergeordneten Programmteil zurück
- Parameter bestehen jeweils aus dem Datentyp und einem Variablennamen

Prozedurprototyp:

- Mit einem Semikolon abgeschlossener Kopf der Prozedur
- Implementierung in anschließender Prozedurdefinition
- Zweck: Bevor eine Prozedur verwendet werden kann, muss sie deklariert sein

Wertparameter vs. Referenzparameter

Wertparameter (call-by-value):

- Parameter werden kopiert und sind *lokale Variablen*
  - Variablen in der Prozedur (lokale Variablen) sind ausserhalb der Prozedur nicht sichtbar
  - Der Gültigkeitsbereich (Skope) ist auf die Prozedur beschränkt
  - Die Variablen des aufrufenden Programmteils bleiben unverändert

Wertparameter vs. Referenzparameter

Wertparameter (call-by-value):

- Parameter werden kopiert und sind *lokale Variablen*
  - Variablen in der Prozedur (lokale Variablen) sind ausserhalb der Prozedur nicht sichtbar
  - Der Gültigkeitsbereich (Skope) ist auf die Prozedur beschränkt
  - Die Variablen des aufrufenden Programmteils bleiben unverändert

Referenzparameter (call-by-reference):

- Parameter sind Referenzen
  - Eine Referenz zeigt auf den gleichen Speicher wie die ursprüngliche Variable
  - Wenn ich eine Referenz ändere, ändert sich auch der Wert der Variablen im aufrufenden Programmteil
  - Eine Referenz auf eine Variable vom Typ `type` wird mit `type&` benannt, z.B. `int& n`

Beispiel — Wertparameter

```
#include <iostream>

// deklaration und definition gemeinsam
void swap (int x, int y)
{
  int temp = x;
  x = y;
  y = temp;
}

int main()
{
  int x = 5, y = 10;
  swap(x,y);
  std::cout << "Nach der Vertauschung x=" << x << ", y= " << y
       << std::endl;
  return 0;
}
```

- Ausgabe:

Beispiel — Wertparameter

```
#include <iostream>

// deklaration und definition gemeinsam
void swap (int x, int y)
{
  int temp = x;
  x = y;
  y = temp;
}

int main()
{
  int x = 5, y = 10;
  swap(x,y);
  std::cout << "Nach der Vertauschung x=" << x << ", y= " << y
       << std::endl;
  return 0;
}
```

- Ausgabe: Nach der Vertauschung x=5, y=10

Lösung 1 — Referenzparameter

```
#include <iostream>

// deklaration und definition gemeinsam
void swap (int& rx, int& ry)
{
  int temp = rx;
  rx = ry;
  ry = temp;
}

int main()
{
  int x = 5, y = 10;
  swap(x,y);
  std::cout << "Nach der Vertauschung x=" << x << ", y= " << y
       << std::endl;
  return 0;
}
```

- **Ausgabe:** Nach der Vertauschung x=10, y=5

Lösung 2 — Pointer

```
#include <iostream>

// deklaration und definition gemeinsam
void swap (int* px, int* py)
{
  int temp = *px;
  *px = *py;
  *py = temp;
}

int main()
{
  int x = 5, y = 10;
  swap(&x,&y);
  std::cout << "Nach der Vertauschung x=" << x << ", y= " << y
       << std::endl;
  return 0;
}
```

- **Ausgabe:** Nach der Vertauschung x=10, y=5