Programmieren / Algorithmen & Datenstrukturen

Grundlagen (i), Teil 3

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Grundlagen (i)

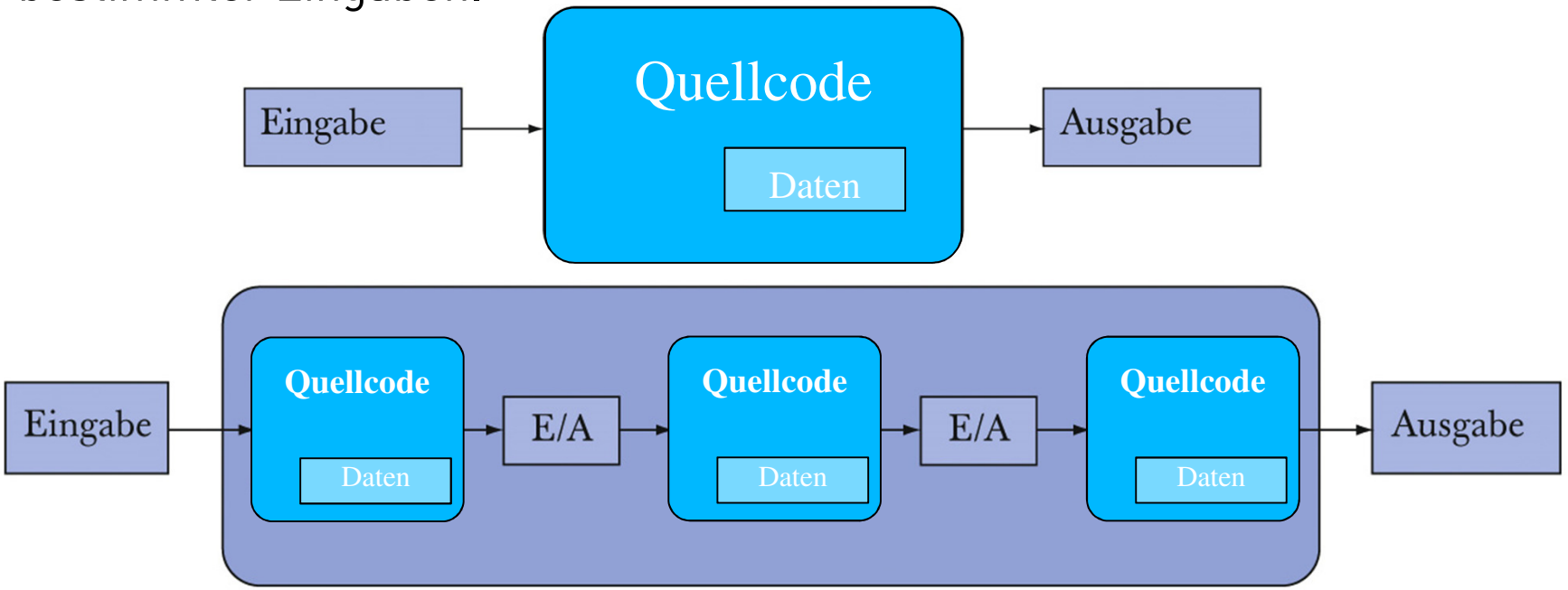
Inhalt.

- Hallo C++
- Objekte, Typen, Werte, und Steuerungsprimitive
- Berechnungen und Anweisungen
- Fehler
- Fallstudie: Taschenrechner
- Funktionen und Programmstruktur
- Klassen

Berechnungen

In gewisser Hinsicht macht ein Programm nichts anderes, als Berechnungen durchzuführen... was ist berechenbar?

- Eine Berechnung ist das Erzeugen von Ausgaben unter Berücksichtigung bestimmter Eingaben.



- Programmiersprachen sind formale Sprachen, die man als sog. *Kalküle* verstehen kann.
- Programme können auch als *Systeme* im Sinn der Systemtheorie verstanden werden.

Bildquelle: Stroustrup.

Ziele und Hilfsmittel

Hauptaufgabe und wichtige Methoden des Programmierens.

- Aufgabe beim Programmieren ist es, Berechnungen als Quellcode auszudrücken.
  1. Korrekt.
  2. Einfach.
  3. Effizient.

  } Beachten Sie die *Reihenfolge.*
- Wichtige Ansätze zur Bewältigung dieser *schwierigen* Aufgabe z.B.:
  - *Abstraktion*: Irrelevantes wird eliminiert, Relevantes wird typisiert, Details werden hinter Schnittstellen verborgen.
  - *Divide et impera*: die Aufgabe wird in mehrere kleinere Teilaufgaben zerlegt, die jeweils unabhängig voneinander gelöst und danach zur Gesamtlösung zusammen gesetzt werden können.
  - *Wiederverwendung*: bereits existierende, bewährte Lösungen werden übernommen und eingesetzt, eigene Lösungen werden möglichst wiederverwendbar gestaltet.

Sprachmittel in C++

Ideen zur Darstellung von Berechnungen.

- Die Sprachmittel in C++ repräsentieren fundamentale Ideen, um Berechnungen als Quellcode auszudrücken, z.B.:
  - Sequenz / Reihenfolge:
    "mach zuerst dies, danach das, und zuletzt jenes".
    ```
    statement; statement; statement;
    ```
  - Selektion / Verzweigung:
    "falls das stimmt, mach hier weiter, falls nicht, mach dort weiter".
    ```
    if( expression ) statement else statement;
    ```
  - Iteration / Wiederholung
    "mach das 100 Mal", "mach solange weiter, bis alles erledigt ist".
    ```
    while( expression ) statement;
    ```
  - Funktionsaufruf:
    "berechne das und komm mit dem Ergebnis wieder".
    ```
    result = function( x );
    ```
  - Arithmetik:
    $d = a + bc$
    ```
    d = a+b*c;
    ```

Ausdrücke

Ausdrücke (*expressions*) haben einen Typ und einen Wert, der sich aus der Auswertung der im Ausdruck enthaltenen Operanden und Operatoren ergibt.

- Operatoren mit Operanden als Ausdruck, z.B. der Operator * und der Zuweisungsoperator =.

```
area = area * 100;
```

  - `area` als linker Operand der Zuweisung *(L-Wert)* ist eine Ausdruck für "das Objekt namens `area`".
  - `area*100` als rechter Operand der Zuweisung *(R-Wert)* ist eine Ausdruck für "der Wert, der im `area` Objekt gespeichert ist multipliziert mit dem Wert des Literals `100`".

- Ein Literal ist in C++ auch ein Ausdruck, wie z.B.:
  - `'@'` (`char`-Literal),
  - `10` (`int`-Literal),
  - `-.14159` (`double`-Literal),
  - `"capitalQ"` (Zeichenketten-Literal).

- Die Definition von Variablen ist in C++ auch ein Ausdruck, dessen Wert bei nicht initialisierten Variablen undefiniert ist.

```
int length { 20 };
int width { 40 };
int area { length * width };
```

Zusammengesetzte Ausdrücke

Ausdrücke können als Kombination von Operanden und Operatoren gebildet werden.

- Operanden mit ihren Operatoren bilden Ausdrücke.
  - Operatoren spezifizieren, was auszuführen ist.
  - Operanden spezifizieren, womit die Operatoren arbeiten.
- Der Vorrang der Operatoren ist bei der Auswertung kombinierter Ausdrücke zu beachten.
  - Beispiel: `double average { (a+b)/2.0 };`
  - Informieren Sie sich selbstständig über die Vorränge der C++ Operatoren.
  - Im Zweifel immer runde Klammern `( )` setzen.
- Vermeiden Sie Ausdrücke mit mehr als einem Vergleich.
  - **`a < b < c`** bedeutet z.B. *nicht*, dass "b zwischen a und c" liegt.
  - Sondern wird von C++ *sequentiell* und *links-assoziativ* (von links nach rechts) mit *impliziten Typumwandlungen* ausgewertet.
  - Also `(a<b)<c` und d.h.: `( true < c )` (1) oder `( false < c )` (0).
- Vermeiden Sie unnötig komplizierte Ausdrücke
  - wie `a*b+c/d*(e-f/g)/h+7`.

Operatoren und Operanden

Wichtige Eigenschaften.

- Unäre Operatoren und Zuweisungsoperatoren sind rechts-assoziativ.
  - Beispiel: `a=b=c` bedeutet `a=(b=c)`.
- Alle anderen Operatoren sind links-assoziativ.
  - Beispiel: `a+b+c` bedeutet `(a+b)+c`.
- Die Reihenfolge, in der Unterausdrücke in einem zusammengesetzen C++ Ausdruck ausgewertet werden, ist ***undefiniert***.
  - *Nur* für die drei Operatoren

    `,` Komma,
    `&&` logisches UND (Konjunktion),
    `||` logisches ODER (Alternation)

    ist in C++ garantiert, dass der linke Operand vor dem rechten Operand ausgewertet wird.
- Grundregel für die Praxis:
  - Wenn Sie den Wert einer Variablen in einem Ausdruck verändern, greifen Sie in diesem Ausdruck *nicht* ein weiteres Mal auf die Variable zu.

Exkurs

Etwas Boolesche Algebra (`a` und `b` sind Objekte vom Typ `bool`).

- Wahrheitswerte: `true`, `false`
- Negation: `!true` ergibt `false`, `!false` ergibt `true`

- Konjunktion und Alternation

a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Nach A. de Morgan benannte Regeln:

`(!a)&&(!b)` ist äquivalent zu `!(a||b)`

`(!a)||(!b)` ist äquivalent zu `!(a&&b)`

- Konditional und Bikonditional

a	b	a ⇒ b	a ⇔ b
true	true	true	true
true	false	false	false
false	true	true	false
false	false	true	true

Mit C++ Operatoren:

(a ⇒ b) ist äquivalent zu `(!a)||b`

(a ⇔ b) ist äquivalent zu `(a&&b)||((!a)&&(!b))`

Kurzschreibweise für einige Operatoren

Vor allem für Inkrement und Dekrement üblich.

- C++ kennt Kurzschreibweisen für einige binäre Operatoren.
  - `a -= c` bedeutet `a = a - c`
  - `++v` bedeutet `v += 1`
  - `++v` bedeutet `v = v + 1`
  - `a *= scale` bedeutet `a = a * scale`
- Bevorzugen Sie die Kurzschreibweise, da sie kürzeren und intuitiveren Quellcode ermöglicht.
  - Voraussetzung ist, dass keine Seiteneffekte in kombinierten Ausdrücken auftreten.
- Die Operatoren ++ und -- können als Präfix- oder Postfix-Operatoren verwendet werden.
  - `y = ++x;` bedeutet `y = ( x += 1 );`
    d.h. `y` erhält den neuen (bereits erhöhten) Wert von `x`.
  - `y = x++;` bedeutet `y = ( tmp = x, x += 1, tmp );`
    d.h. `y` erhält den alten (noch nicht erhöhten) Wert von `x`.

Konstante Ausdrücke

Werden auch symbolische Konstanten genannt.

- Konstante Ausdrücke sind benannte Objekte, denen nach der Initialisierung kein neuer Wert zugewiesen werden kann.
  - Beispiel `const double pi { 3.14159 };`
- Bedeutung:
  - Helfen, den Quellcode lesbarer zu machen.
  - Helfen, den Quellcode leichter änderbar zu machen.
- Setzen Sie in Ihrem Quellcode möglichst oft symbolische Konstanten anstelle von Literalen ein.
  - Beispiel:

```
std::string name { "<>" };          // Warum "<>"?

const std::string noName { "<>" }; // <> bedeutet: kein Name,
std::string name { noName };        // intuitiver Quellcode.
```

`{}`-Syntax als Ausdruck

Die `{}`-Syntax (auch Wertkonstruktion) kann oft als Ausdruck eingesetzt werden.

- Wir haben die `{}`-Syntax schon zur Initialisierung von neu definierten Variablen verwendet
  - Die `{}`-Syntax hatte dabei null oder einen Eintrag, z.B.

```
int tmp {}; // mit 0 initialisiert, leere {}-Syntax
int zwerge {7}; // mit 7 initialisiert, {}-Syntax mit einem Eintrag
std::string nick {"Hegel"}; // {}-Syntax mit einem Eintrag
```

- Die `{}`-Syntax erlaubt auch Initialisierungslisten mit mehreren Einträgen, wie z.B. `{ 1.2, 2.4, 3.6, 4.8 }`.
  - Wir werden in Kürze den `std::vector` Typ kennen lernen, bei dem das sehr nützlich ist.
- Ist der erwartete Typ eindeutig, kann diese `{}`-Syntax u.a. auch
  - als Aufrufparameter in Funktionen,
  - als `return` Wert,
  - als rechter Operand der Zuweisung

  eingesetzt werden.

```
square( {3} );
return {42};
nick = {"Husserl"};
```

Die wohl am häufigsten benötigten Operatoren

Übersicht.

	Bezeichnung	Kommentar
f(a)	Funktionsaufruf	Übergibt a als Argument an f
++lval	Präinkrement	Inkrementieren und den inkrementierten Wert verwenden
--lval	Prädekrement	Dekrementieren und den dekrementierten Wert verwenden
!a	Nicht	Ergebnis ist bool
-a	Unäres Minus	
a*b	Multiplikation	
a/b	Division	
a%b	Modulo (Rest)	Nur für Integer-Typen
a+b	Addition	
a-b	Subtraktion	
out<<b	Schreibt b in out	Wobei out ein ostream ist
in>>b	Liest aus in nach b	Wobei in ein istream ist
a<b	Kleiner als	Ergebnis ist bool
a<=b	Kleiner gleich	Ergebnis ist bool
a>b	Größer als	Ergebnis ist bool
a>=b	Größer gleich	Ergebnis ist bool
a==b	Gleich	Nicht zu verwechseln mit =
a!=b	Ungleich	Ergebnis ist bool
a&&b	Logisches Und	Ergebnis ist bool
a||b	Logisches Oder	Ergebnis ist bool
lval=a	Zuweisung	Nicht zu verwechseln mit ==
lval*=a	Zusammengesetzte Zuweisung	lval=lval*a, gilt auch für /, %, +, -

Bildquelle: Stroustrup.

Anweisungen und Sequenz

Anweisungen (*statements*) bestimmen den Programmablauf.

- Stellen Sie sich eine Anweisung als einen Schritt im Programmablauf vor.
  - Anweisungen werden eingesetzt, um den Programmablauf zu bestimmen.
- Anweisungen
  - werden *sequentiell* in der Reihenfolge ausgewertet, in der sie im Quellcode niedergeschrieben sind
  - und haben im Unterschied zu Ausdrücken *keinen* Wert.
- Ausdruck als Anweisung:
  - Ein Ausdruck gefolgt von einem Semikolon , wie z.B. `++i;`
- Deklaration als Anweisung:
  - Weist einem Objekt einen Namen zu, wie z.B. `int i;`
- Kontrollanweisung:
  - Macht den Programmablauf von Booleschen Bedingungen abhängig, wie z.B. `if( i==0 ) { /*...*/ }`

Sequenz und Blöcke

Mehrere Anweisungen lassen sich durch geschweifte Klammern `{ }` zu Blöcken zusammen fassen.

- Eine Folge von Anweisungen, die zwischen geschweiften Klammern steht, wird Block genannt.
- Ein Block hat seinen eigenen Gültigkeitsbereich.
- Ein *leerer Block* kann nützlich sein um auszudrücken, dass nichts gemacht werden soll.

```
if( min < max )      // alles OK
   { }               // leerer Block, nichts zu tun
else                 // tauschen
   {
      int tmp { min };
      min = max;
      max = tmp;
   }
```

Selektion

Auswahl mit `if()`.

- Die `if` Anweisung
  - Syntax: **`if(`** `Ausdruck` **`)`** `Anweisung` **`else`** `Anweisung`
  - Beispiel:
```
if( a < b )
    max = b;
else
    max = a;
```
  - Soll im `else` Teil nichts passieren,
    - kann er weggelassen werden,
```
if( balance < 0 ) prefix = '-'; // danach direkt weiter
```
    - kann eine leere Anweisung `;` oder ein leerer Block `{}` verwendet werden.
```
if( balance < 0 )
   prefix = '-';
else
   ; // leere Anweisung: kein Praefix '+' fuer Guthaben
```
- Es gibt *keine* spezielle "else if" Semantik in C++.
  - Ein `else` im Quellcode bezieht sich immer auf das ihm unmittelbar vorangehende `if`.

Selektion

Auswahl mit `switch()`: keine moderne Syntax, aber oft klarer als eine Folge verschachtelter `if()`.

- Die `switch` Anweisung an einem kleinen Beispiel:

```
int main( ) {
   const double cm_per_in { 2.54 };  // Zentimeter pro Inch
   int length { 1 };                 // Laenge, in oder cm
   char unit { ' ' };
   std::cout<< "Enter a length followed by a unit (c or i):\n";
   std::cin >> length >> unit;
   switch( unit ) {
       case 'i':
           cout << length << " in == " << cm_p_in*length << " cm\n";
           break;
       case 'c':
           cout << length << " cm == " << length/cm_p_in << " in\n";
           break;
       default:
           cout << "Unknow unit \'" << unit << "\'\n";
           break;
   }
   return 0;
}
```

`break` und `continue`

Informieren Sie sich selbstständig über diese beiden Sprunganweisungen...

Selektion

Wichtige Aspekte der Syntax bei der `switch`-Auswahl mit `case`-Marken.

- Der Wert in den Klammern nach dem Schlüsselwort `switch` muss vom Typ `int`, `char` oder `enum` sein (mehr über `enum` folgt noch).
- Dieser Wert wird mit einer Gruppe von *Konstanten* verglichen, die jeweils Teil der `case`-Marken sind.
  - Es sind *keine* Variablen in den Marken zulässig.
  - Die gleiche Konstante kann *nicht* in mehreren Marken verwendet werden.
  - Mehrere Marken können für einen einzigen Fall verwendet werden.
- Der Programmablauf springt zu der Marke, deren konstanter Wert mit dem Wert in den Klammern nach `switch` übereinstimmt, und wird von dort fortgesetzt.
- Wird keine Übereinstimmung gefunden, dann wird zur `default` Marke gesprungen.
  - Die `default` Marke ist im Quellcode zwar nur optional, soll aber nicht vergessen werden.
- Die `break` Anweisungen beenden den `switch` Block `{ }`.

Selektion

Die `switch`-Anweisung in einem weiteren Beispiel.

- Die `switch` Anweisung mit mehreren Marken pro Fall:

```
int main( ) {
   cout << "Please enter one digit\n";
   char a { ' ' };
   cin >> a;
   switch( a ) {
       case '0':
           cout << a << " is zero\n";
          break;
       case '2': case '4': case '6': case '8':
           cout <<  a << " is even\n";
          break;
       case '1': case '3': case '5': case '7': case '9':
           cout <<  a << " is odd\n";
          break;
       default:
           cout <<  a << " is not a digit\n";
          break;
   }
   return 0;
}
```

Iteration

Wiederholung mit `while()` Schleifen.

- Die `while` Anweisung:

```
01 int main( ) {
02   int i { 0 };          // Start bei 0
03   while( i < 100 )      // Pruefung, ob 100 erreicht ist
04                         // Falls ja, fertig
05   {
06     cout << i << '\t' << i*i << '\n';  // Berechnung und
07                                       // Ausgabe des Quadrats
08     ++i;                              // Inkrementierung
09   }
10   return 0;
11 }
```

- Bestandteile der typischen `while` Anweisung:
  - *Schleifenzähler* (namens `i`) mit Initialisierung (Zeile 2).
  - Quellcode, der bei jedem Schleifendurchlauf ausgeführt wird.
    - Der auf die `while` Klammer unmittelbar folgende Block (Zeilen 5-9), wird auch *Schleifenkörper* oder *Schleifenrumpf* genannt.
  - Inkrementierung (Zeile 8) des Schleifenzählers im Schleifenkörper.
  - Boolesche Abbruchbedingung im *Schleifenkopf* (Zeile 3).

Iteration

Wiederholung mit `for()` Schleifen.

- Die `for` Anweisung:

```
int main( ) {
   for( int i{0}; i < 100; ++i )
      cout << i << '\t' << i*i << '\n';      // Berechnung und
                                             // Ausgabe des Quadrats.
   return 0;
}
```

- Iterieren über eine geordnete Folge von Zahlen kommt beim Programmieren so oft vor, dass es diese spezielle `for` Syntax dafür gibt.
- Die obige Schleife ist äquivalent zu der eben vorgestellten Schleife mit der `while` Anweisung.
  - Definition, Initialisierung und Inkrementierung des Schleifenzählers sowie die Abbruchbedingung sind syntaktisch bei der `for` Anweisung im Schleifenkopf zusammen gefasst.
- Praxistipp: ändern Sie den Zähler einer `for` Schleife *nicht* im Schleifenkörper.

Funktionen

Funktionen sind benannte Anweisungsfolgen.

- Funktionen
  - Logisch zusammen gehörende Anweisungsfolgen.
  - Sind mit einem *Namen* verbunden.
  - Erwarten oft eine Reihe von Eingaben (*Parameterliste*) beim Aufruf.
  - Liefern normalerweise ein Ergebnis (*Rückgabewert*) zurück.
    - D.h. Funktionen haben einen Wert (ähnlich wie Ausdrücke oder Objekte).
- Sinn von Funktionen
  - Klarere und einfachere Programmstruktur.
  - Möglichkeit der Wiederverwendung.
    - Soll eine bestimmte, nicht triviale Berechnung mehrmals im Programm verwendet werden, ist es i.Allg. ein *Programmierfehler*, falls *keine* Funktion dafür verwendet wird.
- Die C++ Standardbibliothek bietet eine große Menge nützlicher, vordefinierter Funktionen.
- Funktionen, die innerhalb von sog. Klassen programmiert sind.
  - Solche Funktionen heißen auch *Memberfunktionen* oder *Methoden* oder *Operationen* der Klasse.

Funktionen

Ähnlich wie bei Variablen gibt es auch bei Funktionen das Konzept von Deklaration und Definition.

- Deklaration
  - Enthält die Informationen, die benötigt werden, um die Funktion syntaktisch korrekt aufzurufen (Deklarationen stehen z.B. in den Header-Dateien der Bibliotheksfunktionen).
  - `int max( int, int ); // Deklaration: Name und Signatur`
- Definition
  - Der vollständige Quellcode der Funktion.
  - ```
    int max( int a, int b ) {
       if( a < b ) return b;
       return a;
    }
    ```
- Bibliotheksfunktionen werden üblicherweise durch Einbindung der Deklarationen in den Headern (`#include`-Direktiven) nutzbar.
  - Oft liegt bei Bibliotheken keine Definition als Quellcode, sondern nur Deklaration und Objektcode vor.
    ```

Funktionen

Parameterübergabe "pass-by-value".

- Beispiel

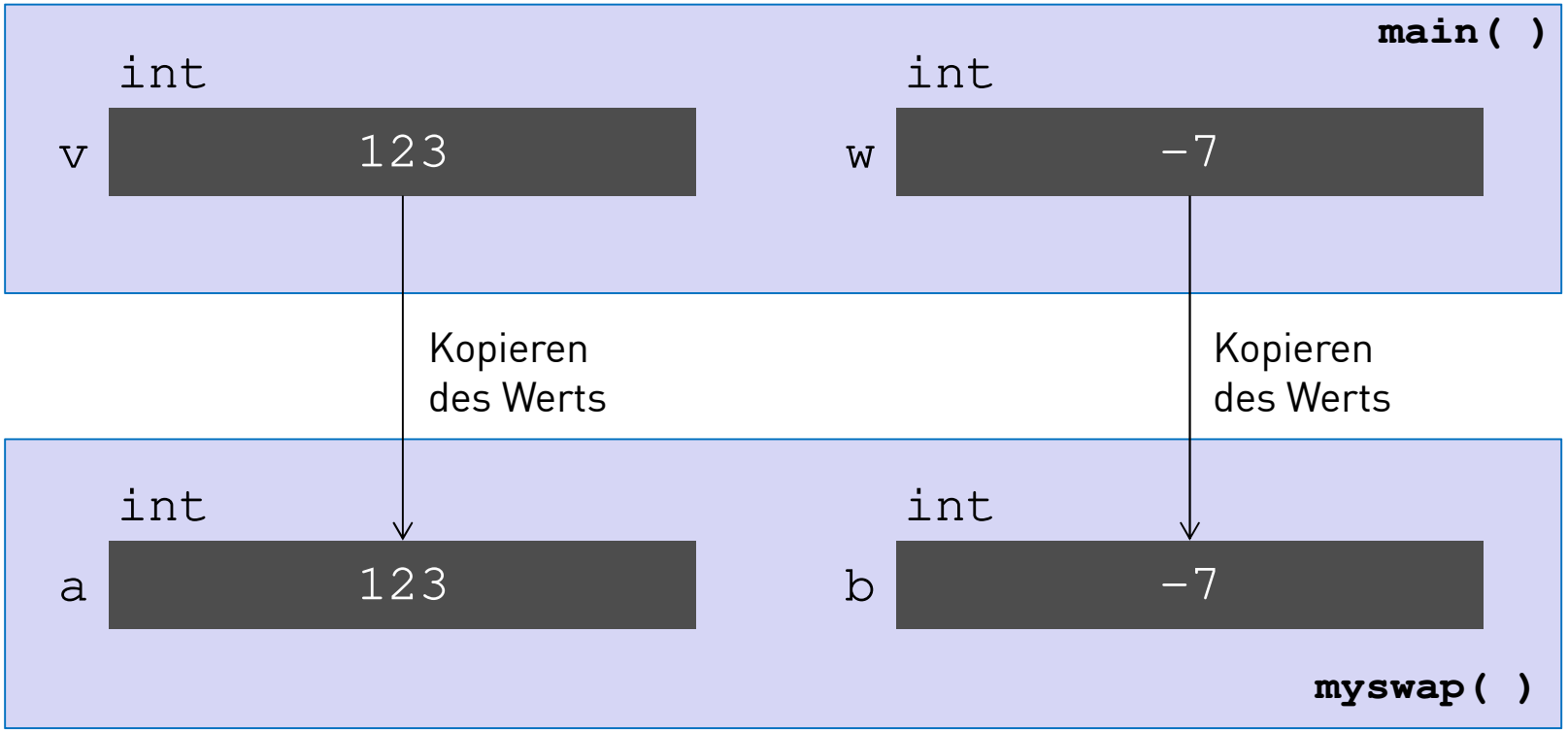
```
01 void myswap( int a, int b ) {
02       int tmp { a }; a = b; b = tmp;
03 }
04 int main( ) {
05    int v { 123 };
06    int w { -7 };
07    myswap( v, w );
08    cout << "v ist " << v << ", w ist " << w << endl;
09    return 0;
10 }
```

- Warum funktioniert der obige Quellcode nicht wie vielleicht intuitiv erwartet?
- Die Parameterwerte werden beim Funktionsaufruf als *Kopien* übergeben.
  - D.h. sie sind nur in der aufgerufenen Funktion lokal gültig und werden bei jedem Aufruf mit einer Kopie des jeweils übergebenen Werts neu initialisiert.

Funktionen

Parameterübergabe "pass-by-value".

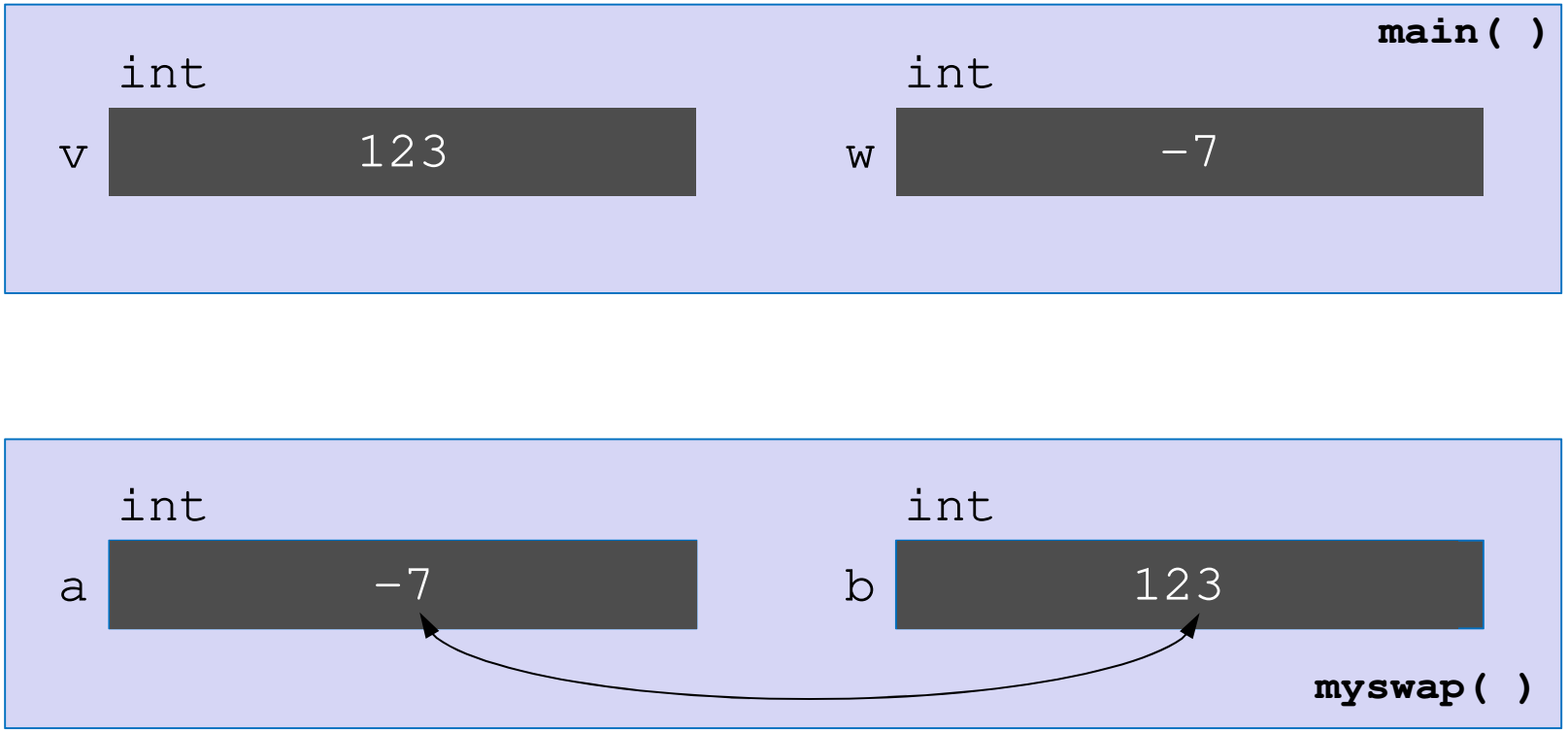
- Sog. "*pass-by-value*"-Prinzip (Wertübergabe).
  - Grund: eine Funktion soll ihre Aufrufparameter nicht ändern.



Funktionen

Parameterübergabe "pass-by-value".

- Sog. "*pass-by-value*"-Prinzip (Wertübergabe).
  - Grund: eine Funktion soll ihre Aufrufparameter nicht ändern.



Funktionen

Parameterübergabe "pass-by-reference" (sog. L-Referenz).

- Eine andere Möglichkeit in C++ ist "pass-by-reference".

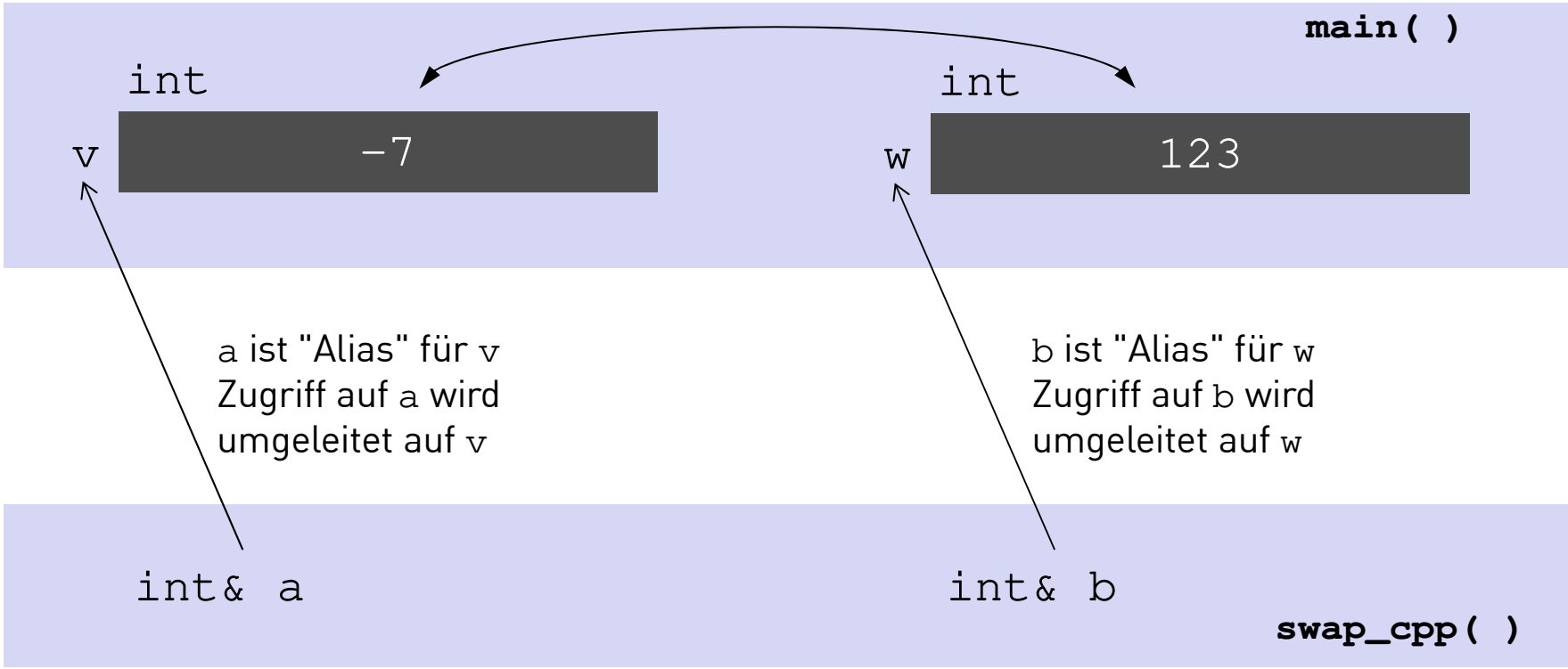
```
01 void swap_cpp( int& a, int& b ) { // a, b sind neue Namen
02    int tmp { a }; a = b; b = tmp;
03 }
04 int main( ) {
05   int v { 123 };
06   int w { -7 };
07   swap_cpp( v, w );
08   cout << "v ist " << v << ", w ist " << w << endl;
09   return 0;
10 }
```

- `a` und `b` sind hier sog. *L-Referenzen*, die man sich als weiteren Namen ("Alias") für ein übergebenes Objekt vorstellen kann.
- Verwenden Sie diese "*pass-by-reference*"-Parameterübergabe nur, wenn es wirklich erforderlich ist (was durchaus vorkommt).
  - Folgen Sie als Einsteiger i.Allg. dem Prinzip, dass eine aufgerufenen Funktion ihre Aufrufparameter nicht ändern sollte.

Funktionen

Parameterübergabe "pass-by-reference" (sog. L-Referenz).

- Sog. "*pass-by-reference*"-Zugriff (L-Referenz-Übergabe).
  - Damit kann eine Funktion ihre Aufrufparameter ändern.
  - Was sie aber nicht ohne guten Grund tun soll.



Funktionen

Parameterübergabe "pass-by-const-reference".

- Auch sehr große Objekte werden beim Funktionsaufruf in Parameterlisten als Kopien an aufgerufenen Funktionen übergeben.
  - Z.B. lange Zeichenketten, große Bitmaps, usw., was aufgrund der Kopieroperation der vielen Daten problematisch werden kann.
- Möchte man derartige Objekte übergeben, ohne sie zeit- und speicheraufwändig zu kopieren, kann man die C++ Möglichkeit des "*pass-by-const-reference*" einsetzen.

```
01 int countWords( const string& x ) { // ggf. ganze Buecher
02   int words { 0 };
03   // zaehle die Woerter durch
04   return words;
05 }
```

  - Die aufgerufene Funktion `countWords()` arbeitet dann mit einer L-Referenz auf das ursprüngliche, große Objekt, d.h. es wird nicht kopiert.
  - `countWords()` kann dieses Objekt nicht (besonders auch nicht versehentlich) verändern, da der Typ `const` ist.

Übung

Parameterübergabe an Funktionen.

- Schreiben sie drei Funktionen:

```
void swap_W( int, int )
void swap_R( int&, int& )
void swap_CR( const int&, const int& )
```

- Alle drei Funktionen sollen den folgenden Funktionsrumpf haben:

```
{ int tmp{a}; a = b; b = tmp; }
```

- Welche der folgenden Aufrufe werden kompiliert und warum, bzw. warum nicht? Welche Werte wurden beim Aufrufer vertauscht und warum, bzw. warum nicht? (Ersetzen Sie ■ durch `W` bzw. `R` bzw. `CR`).

```
int x{7}; int y{9}; swap_■( x, y );
const int cx{7}; const int cy{9}; swap_■( cx, cy );
double dx{7.7}; double dy{9.9}; swap_■( dx, dy );
swap_■( 7, 9 ); swap_■( 7.7, 9.9 );
```

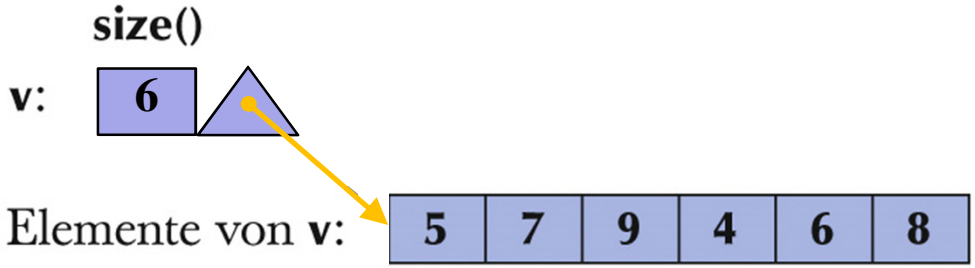
- Versuchen Sie, das Verhalten beim Kompilieren und bei den Aufrufen der Funktionen zu erklären.

LEERE SEITE

Der `vector`-Containertyp

`std::vector` ist ein einfacher und sehr nützlicher Typ zur Handhabung einer Sequenz von Elementen (nötige Headerdatei: `vector`).

- Ein `vector` ist ein Typ für eine Folge von Elementen gleichen Typs, auf die man über einen Index zugreifen kann.




```
// Anlegen eines Objekts namens v
// vom Typ vector von int-Werten,
// der 6 Elemente enthaelt:
std::vector<int> v { 5, 7, 9, 4, 6, 8 };
```

- Beim Anlegen wird der Typ der Elemente in spitzen Klammern hinter der Typbezeichnung mit dem Schlüsselwort `vector` angegeben.
- Die Anfangselemente können beim Anlegen durch Wertkonstruktion mittels einer `{}`-Liste hinter dem Namen des `vector` Objekts angegeben werden.

Bildquelle: Stroustrup.

Der `vector`-Containertyp

Die StdLib stellt für Objekte vom `vector` Typ viele Operationen bereit.

- Zwei weitere der vielen Möglichkeiten, einen `vector` anzulegen:
  - ```
    std::vector<char> vc( 6 ); // ein sog. Konstruktoraufruf:
                               // vector namens vc mit 6 char Werten
    vc[0]='a'; // das erste Element hat den Index 0
    vc[1]='b'; // Indexoperator [], alternativ: vc.at(1) = 'b';
    vc[2]='c';
    vc[3]='x';
    vc[4]='y';
    vc[5]='z';
    ```

    Ein `vector` mit den `char` Werten `'a'`, `'b'`, `'c'`, `'x'`, `'y'` und `'z'`.

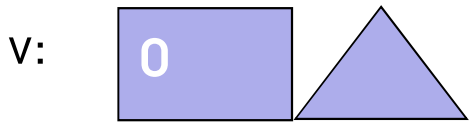
  - ```
    vector<string> vstr1( 1000, "noName" ); // Konstruktoraufruf
    ```

    Ein `vector` mit 1000 `string` Werten, alle mit `"noName"` initialisiert.

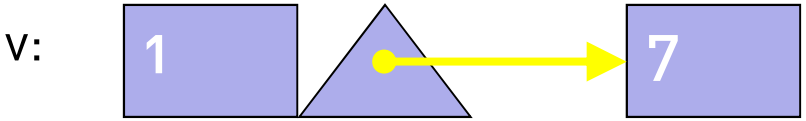
Der `vector`-Containertyp

Die Anzahl der Elemente in einem `vector` Objekt ist dynamisch.

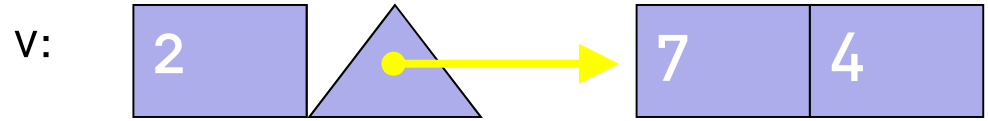
```
vector<int> v{};
// anfangs leer
```



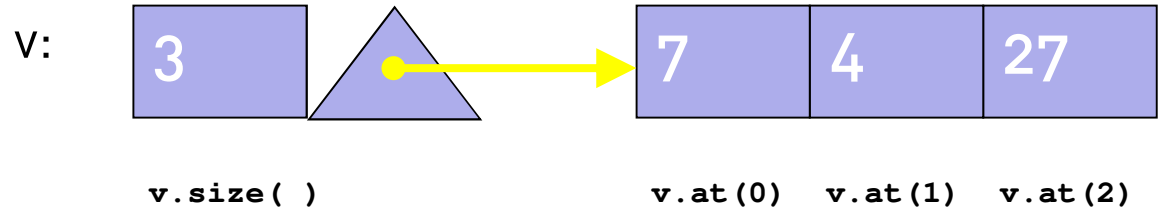

```
v.push_back(7);
// ein int mit Wert 7 am Ende einfuegen
```




```
v.push_back(4);
// ein int mit Wert 4 am Ende einfuegen
```




```
v.push_back(27);
// ein int mit Wert 27 am Ende einfuegen
```



Bildquelle: Stroustrup.

Der `vector`-Containertyp

Die StdLib stellt für Objekte vom `vector` Typ viele Operationen bereit.

- Aufruf der (nur) für `vector` Objekte verfügbaren Operationen (sog. Methoden) durch *Punktsyntax*.

  ```
  objektname.methodenname( parameterliste )
  ```

  - Operation **`push_back( x )`**
    - Speichert am Ende des `vector<T>` Objekts einen weiteren Wert `x` vom Typ `T`.
    - `v.push_back( 8 );`
  - Operation **`size( )`**
    - Liefert die Anzahl der Elemente des `vector<T>` Objekts.
    - `int n{ v.size() };`
  - Operationen für den direkten Elementzugriff:
    Indexoperator **`[ ]`** oder Operation **`at( )`**
    - `int first{ v[0] }; // falls v.size() groesser als 0 ist`
    - `int third{ v.at(2) };`

Der `vector`-Containertyp

Typisches Beispiel für den Einsatz der Methoden des `vector`-Typs.

- Beispiel
  - Schleife über alle Elemente eines `vector` Objekts namens `v`.
  - Elementzugriff im Schleifenrumpf über den `vector`-Indexoperator `[]`.

```
for( unsigned int i{0U}; i < v.size(); ++i )
    cout << "v[" << i << "] ist " << v[i] << endl;
```

- *Wenn Sie bereits Programmiererfahrung haben:*
  - Der C++ `vector` Typ erinnert Sie vielleicht an Datenfelder (Arrays).
  - Wichtigster der vielen Vorteile des `vector` Typs gegenüber Datenfeldern: die dynamische Speicherverwaltung zur Laufzeit übernimmt C++, während sich bei dynamischen Datenfeldern der Programmierer selbst um alles kümmern muss (d.h. *new / delete* im Heap).
  - Vorteile von Datenfeldern gegenüber `vector`: im Normalfall keine.

Der `vector`-Containertyp: Zahlenbeispiel

Einlesen von beliebig vielen Werten in eine Variable vom `vector`-Typ und einfache Auswertung der Zahlenreihe.

```
int main( ) {
   std::vector<double> vTemps{}; // Temperaturwerte
   double temp { 0.0 };
   while( std::cin >> temp ) // beliebig viele einlesen
       vTemps.push_back( temp ); // speichern

   // arithm. Mittel:
   double sum { 0.0 };
   for( unsigned int i{0U}; i<vTemps.size(); ++i )
       sum += vTemps[i];
   std::cout << "Temperaturmittel: "
             << sum/vTemps.size() << std::endl;

   // sortieren mit Funktion aus StdLib, Header algorithm
   std::sort( vTemps.begin(), vTemps.end() );
   // Median ausgeben (nicht ganz korrekt)
   std::cout << "Median: "
             << vTemps.at( vTemps.size()/2 ) << std::endl;
   return 0;
}
```

Der `vector`-Containertyp: Textbeispiel

Einfaches Wörterbuch durch `vector`-Typ realisiert.

```
int main( ) {
   std::vector<string> vWords{}; // Woerter
   std::string temp{};
   while( std::cin >> temp && temp != "quit" ) // beliebig viele einlesen
       vWords.push_back( temp ); // speichern

   // Anzahl der Woerter
   std::cout << "Anzahl: " << vWords.size() << std::endl;

   std::sort( vWords.begin(), vWords.end() );          // sortieren
   for( unsigned int i{0U}; i<vWords.size(); ++i )     // ausgeben, aber
       if( i==0 || vWords.at(i-1)!=vWords.at(i) )      // die Doppelten
           std::cout << vWords.at(i) << std::endl;     // unterdruecken
   return 0;
}
```

Einige Beispielfragen

Berechnungen und Anweisungen.

- Was ist eine Berechnung? Was sind Ein- und Ausgaben für Berechnungen?
- Nennen Sie mindestens fünf eigene Beispiele für Berechnungen.
- Was macht ein Ausdruck? Was ist ein konstanter Ausdruck?
- Was ist ein L-Wert? Nennen Sie Operatoren, die einen L-Wert erfordern. Erklären Sie, warum diese Operatoren einen L-Wert brauchen.
- Nennen Sie einige Operatoren, die für `string`-Objekte verwendet werden können.
- Welchen grundsätzlichen Nachteil hat die `switch`-Anweisung? Welchen Vorteil hat die `switch`-Anweisung dennoch im Vergleich mit `if`?
- Erklären Sie die Quellcodezeile: `bool genProbSolv( bool, bool );`
- Wann sollten Sie für Quellcodeteile eine separate Funktion definieren? Nennen Sie Gründe.
- Erklären Sie die beiden Begriffe Definition und Deklaration einer Funktion.

Einige Beispielfragen

Berechnungen und Anweisungen.

- Erklären Sie die Unterschiede zwischen "pass-by-value", "pass-by-reference" und "pass-by-const-reference" Parameterübergaben anhand eines *eigenen* Beispiels.
- Wie lautet der Index für das siebte Element eines `vector`?
- Wie könnte eine Quellcodezeile aussehen, die das letzte Element eines `vector` namens `vB` von `bool`-Werten mit `true` vergleicht?
- Erklären Sie die Quellcodezeile: `vector<char> alphabet(26);`
- Beschreiben Sie, wie die Memberfunktion `push_back()` einen `vector` verändert.
- Welche Aufgaben haben die `vector`-Memberfunktionen `begin()`, `end()` und `size()`?
- Warum ist der `vector`-Typ so nützlich?
- Wie sortieren Sie die Elemente eines `vector`-Objekts?

Nächste Einheit:
Fehler