Einführung in die Informatik I

Winter 2005/2006

8 Die Programmiersprache C++

Versionsdatum: 14. November 2005

8 Die Programmiersprache C++ ...

Motivation: Ziele dieses Abschnittes

- Einführung in C++ als imperative Sprache (Objektorientierung später)
- Erläuterung und Übung grundlegender Sprachelemente

Herausforderungen:

- Vielzahl neuer Sprachelemente
- erfolgreiches Erlernen der Sprache **nur durch praktische Übung**

Literatur:

- [Ernst] Kapitel 7.3 zu C und 7.4 zu C++ als Erweiterung von C
  Achtung: Wir betrachten keine C-Sprachelemente, die in C++ nicht genutzt werden!
- [Deitel] Deitel, H. und Deitel, P.: C++ How to Program, Pearson Higher Education, 2005
- [Stroustrup] Stroustrup, B.: Die C++ Programmiersprache, Addison-Wesley, 2000
- `www.cplusplus.com`: U.a. ein recht umfassendes Online-Tutorial

Einleitung

- Eine der am weitest verbreiteten höheren Programmiersprachen
- Entwicklung Anfang 80er Jahre unter Leitung von Bjarne Stroustrup bei AT& T
- Unterstützt unter anderem:
  - Prozedurale (= imperative) Programmierung
  - Maschinennahe Programmierung
  - Modulare Programmierung
  - Generische Programmierung (parametrisierte Typen, ***Templates***)
  - Objektorientierte Programmierung
- Basiert auf der Programmiersprache C
- 1. Semester: Konzentration auf imperativen Teil von C++
- 2. Semester: Fokus auf objektorientierte Möglichkeiten

8.1 Ein einführendes Programmbeispiel

Ziele:
- Einfaches Programm zum „ersten Kennenlernen“ der Sprache C++
- Beispiel: Berechnung der Fakultät einer positiven, ganzen Zahl

Definition von Fakultät einer natürlichen Zahl $n \in \mathbb{N}$:

$$n! := \prod_{i=1}^{n} i = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

Algorithmus zur Berechnung von $n!$

1. setze Ergebnis auf 1: `result ← 1`
2. solange $n > 0$:
   2.1. `result ← result * n`
   2.2. `n ← n-1`

Vereinbarung: Wir verwenden für Dateien mit C++-Quellcode die Erweiterung `.cpp`

Programm zur Berechnung der Fakultät

```
#include <iostream>

unsigned int faculty( unsigned int n )
{
  unsigned int result = 1;

  for( unsigned int i=n; i>=1; i-- ) {
    result = result * i;
  }

  return result;
}

int main( int argc, char* argv )
{
  unsigned int n = 10;
  unsigned int result;

  result = faculty(n);

  std::cout << "Fakultaet von " << n
            << ": " << result << std::endl;

  return 0;
}
```

Header-Datei einfügen:

- `iostream` beinhaltet Funktionalität für Ein- und Ausgaben
- „#“ weist auf ***Präprozessor***-Kommando hin
  Hier: Textuelle Ersetzung mit Inhalt der Datei `iostream`

```
#include <iostream>

unsigned int faculty( unsigned int n )
{
  unsigned int result = 1;

  for( unsigned int i=n; i>=1; i-- ) {
    result = result * i;
  }

  return result;
}
```

Funktionsdefinition-Header: Festlegung des Rückgabetyps `unsigned int`, des Funktionsnamens `faculty` und des Übergabeparameter `n` vom Typ `unsigned int`

Funktiondefinition-Rumpf: Befehlsaufrufe in {}-***Block***

- Variablendeklaration: `result` vom Typ `unsigned int`
- Wertzuweisung: `result = 1`
- `for`-Schleife: Wiederholte Ausführung mit ***Laufvariable*** `i` von `i=n` bis `i=1`
- Ergebnisrückgabe: `return result;`

Hauptfunktion `main`:

- Einstiegspunkt des Programms (muß genau einmal auftreten)
- fester Rückgabetyp `int` und Übergabetypen (hier ungenutzt)

Funktionsaufruf:

- Aufruf von `faculty()` mit Variable `n` (Wert 10) als Parameter
- Zuweisung des Rückgabewertes von `faculty()` an `result`

```
int main( int argc, char* argv )
{
  unsigned int n = 10;
  unsigned int result;

  result = faculty(n);

  std::cout << "Fakultaet von " << n
            << ": " << result << std::endl;

  return 0;
}
```

Ausgabe auf Kommandozeile: Funktionalität aus `iostream`

- `std::` legt ***Namespace*** (Namensraum) fest (hier: Standard)
- `cout` legt den ***stream*** (Strom, Kanal) fest (hier: command-out)
- << trennt Elemente, die auszugeben sind; hier:
  - ***konstante Zeichenketten*** durch " eingeklammert
  - Umwandlung von Variablen (`n, result`) in Zeichenketten & Ausgabe
  - Zeilenumbruch wird durch `endl` (end-of-line) erzeugt

Bemerkung: (zu obigem Programm)

Ausgabe des Programmes: Auf der Kommandozeile

```
Fakultaet von 10: 3628800
```

Gleichheitszeichen „=“ entspricht einer Zuweisung; die linke Variable erhält den Wert des Ausdrucks rechts von „=“

Namespace: Fasst logisch zusammengehörende Größen (Typen, Funktionen, Variablen) zusammen mit dem Ziel

- Vermeidung von Namensüberschneidungen
- bewußte Nutzung der Funktionalität eines bestimmten Namensraumes

Operator <<: Entspricht standardmäßig dem Shift-Operator (siehe Logik)

Für Streams wurde ***Operatorüberladung*** verwendet, d.h. der Operator hat eine andere Funktionalität erhalten

Compilieren

Ziel: Übersetzen von C++ Quellcode in ein ausführbares Programm

Syntax: `#>` *`Compiler cpp-Datei`*

Beschreibung:

- `#>` ist ein ***Prompt***: Nachfolgender Text ist ein Shell-Befehl
- *`Compiler`* steht für ein Compilerprogramm, z.B. `g++` (GNU C++ Compiler)
- *`cpp-Datei`* ist die Datei, die den Quellcode enthält.
  Konvention: Für C++-Dateien wird die Dateiendung .cpp verwendet
- Nach erfolgreichem Compilieren befindet sich im gleichen Verzeichnis das lauffähige Programm `a.out`
- Das Programm wird ausgeführt mit: `#> ./a.out`

Vereinbarung:

Wie oben werden im Folgenden abstrakte Kommandos oder Syntax-Elemente *`kursiv`* dargestellt; sie sind Platzhalter für eine konkrete Ausprägungen.

Compilieren (Forts.)

Paramter: Eine kleine Liste der wichtigsten Paramter für den Compiler

`-o` *`file`*	Schreibt das Programm in *`file`* statt in `a.out`
`-c`	es wird kein ausführbares Programm erzeugt, sondern eine sog. ***Objektdatei***. Diese kann später zu einem ausführbaren Programm ***gebunden*** werden (mehr dazu später)
`-Wall`	Es werden zusätzlich zu Fehlern alle Warnungen beim Compilieren ausgegeben
`-g`	Debugging-Ausgaben erzeugen; ermöglicht Fehlersuche mit externen Debugger-Programm
`-O<n>`	Schaltet auf Optimierungslevel `<n>`. Sollte vermieden werden, wenn man debuggen möchte

Beispiel:

```
#> g++ -Wall -o Rational -g Rational.cpp
```

8.2 Kommentare

Einzeiliger Kommentar

Ziel: Einfügen ein- oder mehrzeiliger Kommentare im Quellcode

Syntax: Einzeiliger Kommentar

```
// Kommentartext
```

Text bis zum nächsten Zeilenumbruch wird vom Compiler ignoriert

Syntax: Mehrzeiliger Kommentar

```
/* Kommentarzeile 1
   Kommentarzeile 2
   ...
   Kommentarzeile n */
```

Sämtlicher Text zwischen den `/*` und `*/` wird vom Compiler ignoriert

Beachte:
- ein Verschachteln von Kommentaren führt fast immer zu Problemen
- `/* */`-Kommentare auch innerhalb einer Zeile oder am Zeilenende möglich

8.3 Anweisungen

Ziel: Ausführen eines Befehls

Syntax: *Anweisung*;

Beschreibung:
- Eine Anweisung ist das grundlegende Element jeder prozeduralen Programmiersprache
- In C/C++ werden Anweisungen mit einem ; abgeschlossen
- Eine Anweisung entspricht einem oder mehreren zusammengesetzten Befehlen an den Computer

8.4 Einfache Datentypen

Typ	Größe	Funktion
Ganzzahlen		
int	32 Bit	Vorzeichenbehaftete Ganzzahl
short	16 Bit	Vorzeichenbehaftete Ganzzahl
long	64 Bit	Vorzeichenbehaftete Ganzzahl
unsigned int	32 Bit	Vorzeichenlose Ganzzahl
unsigned short	16 Bit	Vorzeichenlose Ganzzahl
unsigned long	64 Bit	Vorzeichenlose Ganzzahl
Gleitkommazahlen		
float	32 Bit	Gleitkommazahl (einfache Präzision)
double	64 Bit	Gleitkommazahl (doppelte Präzision)
Wahrheitswerte		
bool	8 Bit	Wahrheitswert (`true`, `false`)
Zeichen		
char	8 Bit	ASCII Zeichen

8.4 Einfache Datentypen ...

Aufzählungstypen

Ziel: Definition eines eigenen Types als Liste von Konstanten

Syntax: `enum` *TypeName* {*Konstantenliste*};
Konstantenliste beinhaltet die Konstantennamen, optional mit Wertzuweisung. Ohne Wertzuweisung wird von **0** aufsteigen gezählt.

Beispiel: 1. Ampelfarbe: `enum AmpelFarbe {ROT, GELB, GRUEN};`
Zugewiesene Werte: `ROT=0; GELB=1; GRUEN=2;`

2. Wochentag: `enum Tag {Mo=1,Di=2,Mi=3,Do=4,Fr=5,Sa=6,So=7};`

Interne Darstellung: Als Integer

Vorteil: Direkter Vergleich mit Konstantennamen möglich

Mit `enum`-Typ

```
enum Farbe {Rot, Gelb, Gruen, Blau};

Farbe farbe = Gelb;
if ( farbe == Gruen ) { ... }
```

Ohne `enum`-Typ

```
// vereinbare: 0=Rot, 1=Gelb, 2=Gruen, 3=Blau

int farbe = 1;  // Gelb
if ( farbe == 2 ) { ... } // wenn Gruen
```

8.5 Variablen

Deklaration: *Typ Name*;

- *Typ* ist der Typ der erstellten Variable
- *Name* ist der Name der Variablen
- C++ reserviert (= ***alloziert***) benötigten Speicher für die Variable
- Achtung: Der Wert einer auf diese Weise deklarierten Variable ist zunächst undefiniert!

Deklaration mit Zuweisung: *Typ Name = Wert*;

- *Wert* ist der initiale Wert der Variablen

Zuweisung: *Name = Wert*;

- *Wert* ist der neue Wert (= Inhalt) der Variablen
- Die Variable *Name* muss zuvor deklariert sein

Beispiele:

```
bool b;
float f = 14.31;
char c = 'a';   // Darstellung von Zeichen in Hochkomma
```

8.5 Variablen ...

Benennung von Variablen

Grundsätzlich werden Variablennamen aus Ziffern, Buchstaben und „_“ (Unterstrich, underscore) gebildet; führende Ziffern sind nicht erlaubt!

BNF-Syntax für Variablennamen

```
<identifier>   ::= <letter> [ { <letter+digit> } ]
<letter+digit> ::= <letter> | <digit>
<letter>       ::= _ | a | ... | z | A | ... | Z
<digit>        ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

C++-Schlüsselwörter können nicht als Variablennamen verwendet werden:

and	and_eq	asm	auto	bitand	bitor	bool	break	case	catch
char	class	compl	const	const_cast	continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export	extern	false	float	for	
friend	goto	if	inline	int	long	mutable	namespace	new	not
not_eq	operator	or	or_eq	private	protected	public	register	reinterpret_cast	
return	short	signed	sizeof	static	static_cast	struct	switch	template	this
throw	true	try	typedef	typeid	typename	union	unsigned	using	virtual
void	volatile	wchar_t	while	xor	xor_eq				

Typumwandlung

Wertzuweisung zunächst für Variable und Werte ***gleichen Typs***

Problem: Zuweisung von Variablen bzw. Werten unterschiedlichen Typs notwendig

Explizite Typumwandlung: Angabe der Umwandlung im Programm

C-Variante:

```
float f = 1.7231;
int   i;
i = (int) f;  // i == 1
```

Funktionale Variante:

```
float f = 1.7231;
int   i;
i = int(f);   // i == 1
```

Implizite Typumwandlung: Automatisch („unbemerkte“) Umwandlung aller einfachen Datentypen ineinander

```
int   i = 123456789;
short s;

s = i; // s == -13035
```

```
float f = 3.4e38;
int   i;

i = f; // i == -2147483648
```

```
float         f = -12.34;
unsigned int ui;

ui = f; // ui == 65524
```

Geltungsbereich von Variablen

Ziel: Festlegung von Gültigkeitsbereichen z.B. für Variablen

Ein Block bildet eine Zusammenfassung von Befehlen, Deklarationen von Variablen und Funktionen und von Typdefinitionen

Syntax: $\{Blockinhalt\}$

Beschreibung: Variablen, Konstanten, Funktionen und Typen sind nur innerhalb des aktuellen Blocks bekannt und benutzbar

Man spricht auch von ***Scope*** von Variablen, Konstanten, Funktionen, Typen

Globaler Geltungsbereich: Alles was außerhalb von jedem Block steht

Globale Variablen: Möglichst sparsam einsetzen!

Lokaler Geltungsbereich: Elemente innerhalb (mind.) eines Blocks

Am Blockende werden lokale Variablen gelöscht

Beispiel: Geltungsbereich

```
// Deklaration von globalen Variablen und Funktionen
unsigned int _count = 2;     // führendes '_' ist Konvention

void doSomething()
{
  float ff;         // lokale Variable
  int   count=1;    // lokale Variable

  _count = 12;      // ändert globale Variable

  if ( count > 0 ) {
    float        ff    =1;  // Achtung: Neue lokale Variable!!
    unsigned int _count=1;  // Achtung: Neue lokale Variable!!

    _count = 23;     // ändert lokale Variable aus diesem Block!
    ff     = 1.1212; // ändert lokale Variable aus diesem Block!
  } // hier gehen lokale Versionen ff, _count verloren
} // hier gehen ff, count verloren
```

Bemerkung: Codingstyle für C++

Ziel: Einheitliches Erscheinungsbild von Quellcode zum schnellen Verständnis

Codingstyle: Eine Konvention mit Regel zur Festlegung von

- Namen für Variablen, Funktionen, Konstanten und Typen
- Layout des Quellcodes zur besseren Lesbarkeit

Layout des Quelltextes i.w. durch Blöcke bestimmt; häufige Konventionen:

Variante 1:

```
Funktions-Dekl.{
  Blockinhalt
}
```

Variante 2:

```
Funktions-Dekl.
{
  Blockinhalt
}
```

Variante 3:

```
Funktions-Dekl.
  {
    Blockinhalt
  }
```

Beachte: ***Whitespaces*** (Leerzeichen, Tabulator, Zeilenumbruch) haben, außer bei Präprozessor-Kommandos (beginnend mit #) keinen Einfluss.

Ziel einer Namenskonvention:
- Handelt es sich um (globale oder lokale) Variable, Funktion, Typ oder Konstante?
- Was ist die Aufgabe der Funktion etc.?

Namensräume (Namespaces)

Problematik:

- Grundsätzlich kann jeder Variablen-, Konstanten-, Funktions- oder Typname in einem Block bzw. global nur einmal definiert werden
- Vergibt man einen Namen mehrmals in ***verschachtelten Blöcken***, ist nur das Element im aktuellen Block erreichbar

Namensräume/Namespace ermöglichen eine saubere Trennung

Nutzung von Namespaces

1. Angabe bei jedem Auftreten:
   namespaceName`::`*term*
   term steht für Variable, Konstante, Funktion oder Typ
2. Freigabe des Namespaces in einer Datei mit dieser Anweisung:
   `using namespace` *namespaceName*`;`

Definition von Namespaces behandeln wir hier nicht

8.6 Konstanten

Ziel: Festlegung von unveränderlichen Werten, z.B. für Zuweisungen

Beschreibung:

- Integer-Typen (`int, short, long`):
  - direkt als Dezimalzahl, z.B. `12, -23`
  - als Hexadezimalzahl mit führender `0x`-Sequenz, z.B. `0x8f`
- Float-Typen (`float, double`):
  - direkt als Fließkommazahl, z.B. `1.542`
  - in wissenschaftlicher Schreibweise, z.B. `1.34e-25`
- Bool'sche Werte: `true`, `false`
- ASCII-Zeichen: In einfachen Anführungsstrichen, z.B. `'a', 'z'`
- Zeichenketten: In doppelten Anführungsstrichen, z.B. `"Hallo Welt"`
- Spezielle Zeichenkonstanten, insb. zur Nutzung in Zeichenketten:

`\n` Zeilenumbruch	`\r` Cursor auf Zeilenanfang	`\f` Form feed (Seitenumbruch)
`\a` alert (beep)	`\t` Tabulator	`\'` Einfaches Anführungszeichen
`\v` vertikaler Tabulator	`\b` Backspace (Löschen)	`\"` Doppeltes Anführungszeichen
`\?` Fragezeichen	`\\` Backslash	

Selbstdefinierte Konstanten

Ziel: Deklaration von eigenen unveränderlichen Werten

Syntax: Variante 1

```
#define name value
```

Syntax: Variante 2

```
const type name =value;
```

Zu Variante 1:
- ❍ das `#define` bewirkt eine „blinde“ Textersetzung im gesamten nachfolgenden Quelltext durch den Präprozessor
- ❍ Problem: Textersetzung hat u.U. Nebeneffekt
  ```
  #define int Hugo
  ```
- ❍ bei der Ersetzung findet keine Typüberprüfung statt!

Zu Variante 2: Deklaration einer Variablen mit max. Typsicherheit

8.7 Operatoren

Bit-Operatoren wurden bereits im Kapitel Logik vorgestellt. Zusätzlich gibt es jedoch noch folgende:

Arithmetik

Ziel: Umsetzung von arithmetischen Operationen

Symbole: `+,-,*,/`

Modulo

Ziel: Rest der ganzzahligen Division

Syntax: *zahl_1* `%` *zahl_2*

Beispiel: $7:3=2$ Rest $1 \rightarrow$ `7 % 3 == 1`

Zusammengesetzte Operatoren

Ziel: Zunächst Ausführen einer Operation, dann Zuweisung

Symbole:

- Arithmetische Operatoren: `+=,-=,*=,/=,%=`
- Shift-Operatoren: `>>=,<<=`
- Bitoperatoren: `&=,|=`

Beispiele:

```
int i = 11; // initiale Zuweisung: i == 11

i  %= 3;    // Modulo: i == 11 % 3 == 2
i <<= 5;    // Linksshift um 5 Binärstellen: i == 64
i  |= 0x33; // Bitweise ODER mit 0x33 == 51
            // 64 == 1000000
            // 51 == 0110011
            // ODER: 1110011 == 115
```

`sizeof`-Operator

Ziel: Bestimmen der Größe einer Variablen im Speicher

Syntax: `sizeof(`*term*`)`

Beschreibung:

- *term* kann entweder eine Variable oder ein Typ sein
- Der `sizeof`-Operator liefert die nötige Größe der Variable / des Typs im Speicher in der Einheit Byte

Beispiel:

```
std::cout << "Die Größe eines int beträgt "
          << sizeof(int) << " (Byte)\n";
```

Ergebnis: `sizeof(int) == 4`

Logische Verknüfungen

Ziel: Umsetzung von logischen Verknüfungen

Symbole:

Symbol	Bedeutung
`&&`	Konjunktion (Und)
`||`	Disjunktion (Oder)
`!`	Negation

Beschreibung:

- In C++ wird jeder Wert $\neq 0$ als `true` und nur 0 als `false` interpretiert
  $\Rightarrow$ jeder Wert (Zahl, Zeichen, Zeichenkette) als bool'sch interpretierbar
- die Konstanten `true` und `false` können auch direkt verwendet werden

Beispiel:

```
bool a = true;
bool b = !a;      // false
bool c = a && b; // false
```

Logische Vergleiche

Ziel: Umsetzung von logischen Vergleichen

Symbole: `==,!=,>,<,>=,<=`

Beschreibung:

- Setzen vergleichende Operationen um
- Liefern als Ergebnistyp immer `bool`
- Bedeutungen:

`==`	gleich	`!=`	ungleich
`<`	kleiner	`<=`	kleiner oder gleich
`>`	größer	`>=`	größer oder gleich

Beispiel: `bool b = ( 3 < 5 ) && ( 5 != 6 );  // true`

Inkrement und Dekrement

Ziel: Erhöhen und Verringern einer Ganzzahl

Syntax:

```
number++ // Variante 1
number--
++number // Variante 2
--number
```

Beschreibung:

- Erhöhen bzw. Verringern die entsprechende Variable um 1
- Variante 1: Zuerst Ausführung der umliegenden Operation, dann des Inkrements bzw. Dekrements
- Variante 2: Zuerst Ausführung des Inkrements bzw. Dekrements, dann der umliegenden Operation

Beispiel:

```
int a,b; // a=?, b=?
a = 1;   // a=1, b=?
b = 2;   // a=1, b=2
a = b++; // a=2, b=3
a = ++b; // a=4, b=4
```

Bedingte Anweisung (`if` und `else`)

Ziel: Bedingte Ausführung von Programmteilen

Syntax:

```
if (condition) {
  if-code
}
else {
  else-code
}
```

Beschreibung:

- *condition* muss als bool'scher Wert interpretierbar sein
  Achtung: `true` bzw. `false` in C++ heißt 0 bzw. $\neq 0$
- Wenn *condition* `true` ist, wird *if-code* ausgeführt
- Der optionale `else` Teil wird, wenn vorhanden, ausgeführt, wenn *condition* `false` ist

Beispiel:

In Blockform:

```
int i = 5;
if( i == 5 ) {
  i++;
}
else {
  i--;
}
```

Einzeilige Variante:

```
int i = 5;
if( i == 5 )
  i++;
else
  i--;
```

Gefährliche Verschachtelung:

```
int i = 5, j = 8;
if( i == 5 )
if ( j != 8 ) i++;
else j--;
else i--;
```

Bemerkung:

- `if` Anweisungen können selbstverständlich geschachtelt werden
- Besteht *if-code* lediglich aus einer Anweisung, können die Block-Klammern (`{` und `}`) weggelassen werden.

Vorsicht: In geschachtelten `if` Anweisungen ohne Block-Klammern ist oft nicht mehr klar ersichtlich, welches `else` zu welchem `if` gehört (sog. dangling-else)

Bedingte Schleife (`while`)

Ziel: Wiederholte Ausführung von Programmteilen

Syntax:

```
while (condition) {
  code
}
```

Beschreibung:

- *condition* muss ein bool'scher Wert sein
- Solange *condition* `true` ist, wird *code* ausgeführt

Beispiel: Berechnung der Fakultät

```
// Deklarationen
int n = 3;          // Zahl deren Fakultät berechnet werden soll
int i = n;          // Zählvariable
int result = 1;     // Das Ergebnis
// Wiederhole Schleife, solange i größer 0 ist
while( i > 0 ) {
  result *= i;
  i--;
}
```

Alternative: `do while` Soll *code* mindestens einmal ausgeführt werden, kann auch eine alternative Syntax gewählt werden:

```
do {
  code
} while (condition);
```

Beispiel: Feststellen ob eine Zahl Primzahl ist

```
// Deklarationen
int n = 127;                    // Beispielzahl
int i = n - 1;                  // Lasse Zahl selbst aus
bool isPrime = true;

// Schleife
do {
  if( number % i == 0 ) {       // Wenn Rest bei Division = 0
    isPrime = false;            // ist number keine Primzahl
  }
  else {
    i--;                        // sonst fahre fort
  }
} while ( isPrime && i > 1 ); // bis i == 2
```

`for`-Schleife

Ziel: Wiederholte Programmteile mit Laufvariable

Syntax:

```
for( init; condition; increment ) {
  code
}
```

Beschreibung:

- *init* wird zu Beginn der Schleife einmalig ausgeführt
- *init* initialisiert gewöhnlich eine Zählvariable
- Die Schleife wird ausgeführt, solange *condition* `true` ist
- *increment* wird nach jedem Schleifendurchlauf ausgeführt
- *increment* wird i.a. verwendet, um die Zählvariable zu verändern

Beispiel: Alle Großbuchstaben ausgeben

```
for( int i = 65; i < 91; i++ ) {
  std::cout << char(i) << "\n";
}
```

```
for( char c = 'A'; c <= 'Z'; c++ ) {
  std::cout << c << "\n";
}
```

Bemerkung:

- Wird ein `int` auf `char` gecastet, so entsteht das Zeichen, das dem ASCII Wert des `int` entspricht

Auswahl (`switch`)

Ziel: Auswahl aus verschiedenen Optionen

Syntax:

```
switch( int-expression ) {
  case const_1:
    code_1
    break;
  ...
  case const_n:
    code_n
    break;
  default:
    Default
    break;
}
```

Beschreibung:

- *int-expression* muss ein int oder ein direkt in int konvertierbarer Typ sein (z.B. short oder char)
- bei jedem `case` *const_i* wird geprüft: *int-expression*==*const_i*
  Wenn `true`, dann wird nachfolgender Code bis zum nächsten `break` oder bis zum Ende des `switch`-Blocks ausgeführt.
- Wenn kein `case true` liefert, wird *Default* ausgeführt (sofern vorhanden)
- `break` springt direkt aus dem `switch`-Block. So lassen sich mehrere Optionen mit gleichem Code verwirklichen.

`break`-Kommando kann auch zum direkten Verlassen von `while-, do-while-` und `for`-Schleifen benutzt werden

Beispiel:

```
switch( i ) {
  case 1:
  case 3:
  case 5:
    std::cout << "Ungerade Zahl < 5\n";
    break;
  case 0:
  case 2:
  case 4:
    std::cout << "Gerade Zahl   < 5\n";
    break;
  default:
    std::cout << "Zahl > 5\n";
    break;
}
```