Vorkurs C++ Programmierung

Operatoren
Blöcke
Kontrollstrukturen

UNIVERSITÄT SIEGEN

Letzte Stunde...

- Programmierung allgemein
  - Vielfältige Anwendungen, Sprachen, ...
- Ein erstes Programm (Fakultätsberechnung)
  - Variablen
  - Datentypen
  - Operatoren (arithmetisch, Vergleichsoperatoren)
  - Prioritäten und Klammern
  - Anweisungen
- Kompiliervorgang und Ausführung
  - g++

Heute

- Noch mehr Operatoren
- Kontrollstrukturen
- Blöcke und Geltungsbereiche
- Codingstyle

Logische Verknüpfungen

- Gestern: Vergleichsoperatoren
  - Umsetzung von logischen Vergleichen
  - Symbole: `==` , `!=` , `>` , `<` , `>=` , `<=`
  - Liefern als Ergebnistyp immer **`bool`** (also **`true`** oder **`false`**)
- Formulierung und Verknüpfung logischer Aussagen

Symbol	Bedeutung
&&	Konjunktion („Und-Verknüpfung“)
||	Disjunktion („Oder-Verknüpfung“)
!	Negation

Beispiel:

```
bool a = ( 3 < 5 ) && ( 5 != 6 ); //a ist true
bool b = ( 3 > 5 ) && ( 5 != 6 ); //b ist false
```

Zuweisungsoperatoren

- Zusammengesetzte Operatoren
  - Ziel: Zunächst Ausführen einer Operation, dann Zuweisung
  - Symbole: `+=` , `-=` , `*=` , `/=`, `%=`
  - `i += 3` entspricht der Anweisung `i = i + 3;`
  - Beispiel:

```
int i = 1; //initiale Zuweisung
i += 3;
```

→ i hat den Wert 4.

Operatoren...

- Es gibt noch mehr Operatoren...
  - z.B. bitweise Konjunktion, Disjunktion, XOR, Negation
  - man kann auch selbst Operatoren definieren (→„Operatorüberladung“)
- ...aber die bisher genannten reichen für den Anfang!

Kontrollstrukturen

- Bisher: Sequenzielle Abarbeitung von Anweisungen
- Anfangsbeispiel (Fakultätsberechnung):

```
#include <iostream>

int main ( int argc, char* argv )
{
   unsigned int result;
   result = 1;
   unsigned int n = 10;
   for ( unsigned int i = n; i >= 1; i-- )
   {
     result = result * i;
   }
   std::cout << „Fakultaet von “ << n << „: “ << result;

   return 0;
}
```

Berechnung Hier: Eine Schleife mit n Durchläufen

Kontrollstrukturen

- Für anspruchsvollere Dinge braucht man deutlich mehr Flexibilität
  - Ausführung von Befehlen nur unter bestimmten Bedingungen:
    - `if`-Anweisung
    - `switch`: Auswahl aus verschiedenen Optionen
  - Wiederholte Ausführung eines Befehls oder einer Befehlssequenz:
    - bedingte Schleife: `while` , Sonderform `do ... while`
    - Schleife mit Laufvariable: `for`-Schleife

Kontrollstrukturen: Bedingte Anweisung

- Bedingte Anweisung (**`if`** und **`if/else`**)
  - Ziel: Bedingte Ausführung von Programmteilen
- **`condition`** muss als bool'scher Wert interpretierbar sein!
  - wenn **`condition == true`** gilt, wird der **`if-code`** ausgeführt, wenn **`condition == false`**, wird der Block übersprungen.
- Der optionale(!) **`else`**-Teil wird, wenn vorhanden, ausgeführt, wenn **`condition == false`**

```
//...
if(condition)
{
   //if-code
}
//...
```

ODER

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


Kontrollstrukturen: Bedingte Anweisung

- Besteht der if-code lediglich aus einer Anweisung, *können* die Blockklammern weggelassen werden...
- **`if`**-Anweisungen können geschachtelt werden
- zur besseren Übersicht
  - Klammern
  - Einrückungen
  - → Einschub!

```
//...
if(condition)
{
   if(condition)
   {
        //...
   }
   else
   {
        //...
   }
}
//...
```


Einschub: Blöcke und Geltungsbereiche

- Code ist strukturiert durch unterschiedliche Blöcke, die ineinander verschachtelt sein können.
- Variablen sind nur innerhalb des aktuellen Blocks bekannt und gültig, in dem sie *deklariert* sind!

```
#include <iostream>

unsigned int _count = 2; //globale Variable

int main(int argc, char* argv[])
{                          //BLOCKEBENE 1
   float f;                //lokale Variable
   int count = 1;          //lokale Variable
   _count = 12;            //ändert globale
                           //Variable
   if (count > 0)
   {                       //BLOCKEBENE 2
     float    f = 1;      //Neue lokale Variable!
     int _count = 1;      //Neue lokale Variable!
         _count = 23;     //ändert lokale Variable
              f = 1.1212;//ändert lokale Variable
     std::cout << _count; //Ausgabe: 23
   } //hier gehen lokale Versionen von f und
     //_count verloren
   std::cout << _count; //Ausgabe: 12
   std::cout << f;      //Was passiert hier?
} //hier gehen auch f und count verloren.
```


Einschub: Blöcke und Geltungsbereiche

- Bezeichner müssen innerhalb von Blöcken eindeutig sein!
- Alles, was außerhalb von jedem Block steht, ist global, alles andere ist lokal
- (Variablen sollten nur in begründeten Fällen global sein!)

```
#include <iostream>

unsigned int _count = 2; //globale Variable

int main(int argc, char* argv[])
{                          //BLOCKEBENE 1
   float f;                //lokale Variable
   int count = 1;          //lokale Variable
   _count = 12;            //ändert globale
                           //Variable
   if (count > 0)
   {                       //BLOCKEBENE 2
     float    f = 1;      //Neue lokale Variable!
     int _count = 1;      //Neue lokale Variable!
         _count = 23;     //ändert lokale Variable
              f = 1.1212;//ändert lokale Variable
     std::cout << _count; //Ausgabe: 23
   } //hier gehen lokale Versionen von f und
     //_count verloren
   std::cout << _count; //Ausgabe: 12
   std::cout << f;      //Was passiert hier?
} //hier gehen auch f und count verloren.
```

Einschub: Codingstyle für C++

- „Codingstyle“: Eine Konvention zur Festlegung von...
  - **Bezeichnern** für Variablen, Funktionen, Konstanten und Typen
  - **Layout** des Quellcodes zur besseren Lesbarkeit
  - Blöcke auf gleicher „logischer Ebene“ sollten auch im Code auf der gleichen Einrückungsebene stehen.
- Namenskonvention für Bezeichner
  - Datentyp einer Variablen ist nur bei der Deklaration zu sehen
    - das kann zu Fehlern führen, wenn man die Variable später benutzt
  - deshalb sinnvoll: Typ im Namen unterbringen, z.B.

```
float fPi = 1.1415;
bool  bFinished = true;
```

- Sprechende Namen!
- All das ändert nichts am Programm, erleichtert aber die Arbeit um ein Vielfaches, vor allem in großen Projekten.

Guter Stil vs. **schlechter Stil**

```
#include <iostream>

int main(int argc, char* argv[])
{ float x = 5000.0; float y = 3.0;
float z = 0.0;
int a = 3;
float d = 0.0;
for (int i = 0; i < a; i++)
{z = x * (y/100.0);
d = x + z;
x = d;} std::cout << d;
return 0; }
```


Guter Stil vs. schlechter Stil

```
#include <iostream>

int main(int argc, char* argv[])
{
  float fStartkapital = 5000.0;
  float fZinssatz = 3.75;
  float fZinsen = 0.0;
  int nLaufzeitInJahren = 3;
  float fEndkapital = 0.0;
  for (int i = 0; i < nLaufzeitInJahren; i++)
  {
      fZinsen = fStartkapital * (fZinssatz/100.0);
      fEndkapital = fStartkapital + fZinsen;
      fStartkapital = fEndkapital;
  }
  std::cout << fEndkapital;
  return 0;
}
```


Kontrollstrukturen: `switch`-Auswahl

- Sinn: Auswahl aus verschiedenen Optionen
- *int-expression* muss ein **int**, **short** oder **char**-Typ sein
- bei jedem **case** *const_i* wird geprüft:
  int-expression == const_i ?
  Wenn **true**, wird nachfolgender Code bis zum nächsten **break** ausgeführt
  Wenn kein **case true** liefert, wird der Code unter **default** ausgeführt, sofern vorhanden.

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

Kontrollstrukturen: `switch`-Auswahl

- Sinn: Auswahl aus verschiedenen Optionen
- *int-expression* muss ein **int**, **short** oder **char**-Typ sein
- bei jedem **case** *const_i* wird geprüft:
  int-expression == const_i ?
  Wenn **true**, wird nachfolgender Code bis zum nächsten **break** ausgeführt
  Wenn kein **case true** liefert, wird der Code unter **default** ausgeführt, sofern vorhanden.
- **Anwendungsmöglichkeit:** Realisierung eines einfachen, textbasierten Menüs, Reaktion auf Benutzereingaben.

```
Beispiel:

unsigned int i = 4;
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;
}
```


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
- **Wichtige Anweisungen** zur Anwendung innerhalb von ***code***:
  - **break;** verlässt den **while**-Block unmittelbar
  - **continue;** springt unmittelbar zum Blockanfang zur Prüfung von ***condition***

Bedingte Schleife (**while**): Beispiel

- Fakultätsberechnung für n = 3

```
int n = 3;
int result = 1;

//wiederhole, solange n > 0
while (n > 0)
{
    result *= n;
    n--;
}
```


Bedingte Schleife (**while**): Beispiel

- Fakultätsberechnung für n = 3, etwas weniger übersichtlich...

```
int n = 3;
int result = 1;

//Durchlaufe „ewig“, bis...
while ( true )
{
    result *= n--;
    if ( n == 0 ) break; //... n == 0
}
```

Alternative: `do while`

- Soll *code* mindestens einmal ausgeführt werden, kann auch diese alternative Syntax gewählt werden:

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


Alternative: `do while`

- Beispiel: Feststellen, ob eine Zahl Primzahl ist.

```
//Deklarationen
int n = 127;              //Beispielzahl
int i = n/2;              //Beginne bei n/2
bool isPrime = true;
//Schleife
do
{
   if (n % i == 0)        //wenn Rest bei Division = 0
   {
     isPrime = false;     //ist n keine Primzahl
   }
   else
   {
     i--;                 //sonst fahre fort
   }
} while (isPrime && i > 1); //solange kein Teiler gefunden
                            //UND solange i > 1
```


`for`-Schleife

- Ziel: Wiederholte Programmteile mit Laufvariable
- Syntax:

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


`for`-Schleife

- Ziel: Wiederholte Programmteile mit Laufvariable
- Syntax:

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

- `init` wird zu Beginn der Schleife einmalig ausgeführt
- `init` initialisiert eine Zählvariable

`for`-Schleife

- Ziel: Wiederholte Programmteile mit Laufvariable
- Syntax:

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

- `init` wird zu Beginn der Schleife einmalig ausgeführt
- `init` initialisiert eine Zählvariable
- Die Schleife wird ausgeführt, solange `condition == true` ist

`for`-Schleife

- Ziel: Wiederholte Programmteile mit Laufvariable
- Syntax:

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

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

`for`-Schleife: Beispiel

- Ausgabe der Quadratzahlen bis 10:

```
for (int i = 1; i <= 10; i++)
{
   int result = i * i;
   std::cout << result << „  “;
}
```

```
Ausgabe:
1 4 9 16 25 36 49 64 81 100
```

- In `init` deklarierte Variablen sind innerhalb des `for`-Blocks gültig
- `break;` und `continue;` können wie bei `while` verwendet werden

Zusammenfassung

- Operatoren
  - logische Verknüpfungen
  - Zusammengesetzte Operatoren („Zuweisungsoperatoren“)
- Kontrollstrukturen
  - Bedingte Anweisungen (`if`-Abfrage, `switch`-Konstrukt)
  - Schleifen: `while`, `do...while`, `for`-Schleife
- Blöcke, Geltungsbereiche, Programmierstil