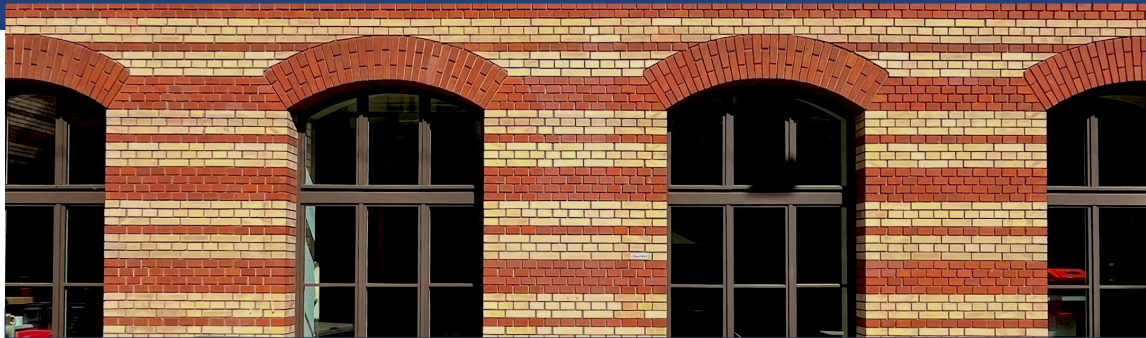




Informatik

Ganzzahldivision, Modulo, Ausdrücke

Adel Gavranović

ETH Zürich — 2026

Überblick

Ganzzahlige Division und Modulo

Logische Ausdrücke

Kurzschlussauswertung (“Short Circuit”)

Ausdrücke mit gemischten Typen

Gültige und ungültige Ausdrücke



`adelgavranovic.ch`

 Webpage

 Mail

1. Follow-up

Follow-up aus letzter Übungsstunde

Follow-up aus letzter Übungsstunde

- $\text{\LaTeX}$ -Tutorial (optional, *nicht* prüfungsrelevant)  Youtube Tutorial

2. Feedback zu **code** expert

Allgemeines bezüglich **code expert**

Allgemeines bezüglich **code expert**

- Läuft alles?
- Wer hat das C++-Tutorial versucht?

Fragen bezüglich **code expert** eurerseits?

3. Lernziele

Ziele

Ziele

- ☐ Ganzzahl- und Modulo-Division verstehen und berechnen können
- ☐ (Gemischte) Ausdrücke von Hand evaluieren können und Evaluationsschritte begründen können
- ☐ Kurzschlussauswertung verstehen

4. Zusammenfassung

Getting on the same page

Getting on the same page

- Was wurde diese Woche behandelt?

5. Ganzzahlige Division und Modulo

Integer Division & Modulo

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”

Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

Modulo ($a\%b$)

Division mit Rest, aber *nur* den Rest.

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

7/3

Modulo ($a\%b$)

Division mit Rest, aber *nur* den Rest.

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

$7/3 == 2$
 $15/4$

Modulo ($a\%b$)

Division mit Rest, aber *nur* den Rest.

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

7/3 == 2

15/4 == 3

16/4

Modulo (a%b)

Division mit Rest, aber *nur* den Rest.

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

7/3 == 2

15/4 == 3

16/4 == 4

Modulo (a%b)

Division mit Rest, aber *nur* den Rest.

7%3

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

$$7/3 == 2$$

$$15/4 == 3$$

$$16/4 == 4$$

Modulo (a%b)

Division mit Rest, aber *nur* den Rest.

$$7\%3 == 1$$

$$15\%4$$

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

$$7/3 == 2$$

$$15/4 == 3$$

$$16/4 == 4$$

Modulo (a%b)

Division mit Rest, aber *nur* den Rest.

$$7\%3 == 1$$

$$15\%4 == 3$$

$$16\%4$$

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

$$7/3 == 2$$

$$15/4 == 3$$

$$16/4 == 4$$

Modulo (a%b)

Division mit Rest, aber *nur* den Rest.

$$7\%3 == 1$$

$$15\%4 == 3$$

$$16\%4 == 0$$

Integer Division & Modulo

Integer Division (a/b)

Der Compiler “ignoriert”
Dezimalstellen, wenn er (**unsigned**)
int durch (**unsigned**) **int** teilt.

$$7/3 == 2$$

$$15/4 == 3$$

$$16/4 == 4$$

Modulo ($a\%b$)

Division mit Rest, aber *nur* den Rest.

$$7\%3 == 1$$

$$15\%4 == 3$$

$$16\%4 == 0$$

Wichtige Identität

$$(a / b) * b + a \% b == a$$

Fragen/Unklarheiten?

Code Snippet Quiz

Frage: Welche Frage beantwortet dieses Code Snippet?

```
int a;  
std::cin >> a;  
  
if (a % 2 == 0) {  
    std::cout << "Yes" << std::endl;  
} else {  
    std::cout << "No" << std::endl;  
}
```

Antwort:

Code Snippet Quiz

Frage: Welche Frage beantwortet dieses Code Snippet?

```
int a;
std::cin >> a;

if (a % 2 == 0) {
    std::cout << "Yes" << std::endl;
} else {
    std::cout << "No" << std::endl;
}
```

Antwort: Die Ausgabe beantwortet, ob die eingegebene Zahl (die in a gespeichert wird) eine gerade Zahl ist.

Mal schauen, was ihr gelernt habt

- Geht auf `expert.ethz.ch`
- Logt euch ein
- Geht zu “Code Examples”
- Unter “Week 2: Exercise Session”, öffnet “Last Three Digits”

Mal schauen, was ihr gelernt habt

- Geht auf `expert.ethz.ch`
- Logt euch ein
- Geht zu “Code Examples”
- Unter “Week 2: Exercise Session”, öffnet “Last Three Digits”
- Überlegt euch alleine, wie ihr die Aufgabe mit Stift und Papier lösen würdet (5 Minuten)

Tipp: Spielt zuerst mit ein paar konkreten Zahlen herum!

Mal schauen, was ihr gelernt habt

- Geht auf `expert.ethz.ch`
- Logt euch ein
- Geht zu “Code Examples”
- Unter “Week 2: Exercise Session”, öffnet “Last Three Digits”
- Überlegt euch alleine, wie ihr die Aufgabe mit Stift und Papier lösen würdet (5 Minuten)
Tipp: Spielt zuerst mit ein paar konkreten Zahlen herum!
- Bildet Zweiergruppen mit euren Sitznachbar:innen, diskutiert eure Ideen und schreibt zusammen das Programm (10 Minuten)

Aufgabe

Aufgabe “Last Three Digits”

Schreibt ein Programm, das eine ganze Zahl a grösser als 1000 einliest und ihre letzten drei Ziffern, getrennt durch Leerzeichen, ausgibt.

Zum Beispiel soll für $a = 14325$ die Ausgabe `3 2 5` sein.

Lösung

```
#include <iostream>
int main() {
    int a;
    // input
    std::cin >> a;
    // computation
    int digit0 = a % 10; //3
    int remainder0 = a / 10;
    int digit1 = remainder0 % 10;
    int remainder1 = remainder0 / 10;
    int digit2 = remainder1 % 10;
    // output
    std::cout << digit2 << " " << digit1 << " " << digit0 << std::endl; //
    return 0;
}
```

Handwritten diagram illustrating the digit extraction process for the number 52713:

$a = 52713$

Indices (4, 3, 2, 1, 0) are written above the digits.

The digits are 5, 2, 7, 1, 3. The last digit (3) is underlined in blue.

The digits 5, 2, 7, and 1 are written below the first four digits, respectively, in blue.

A pink minus sign is written below the last digit (3).

Wichtig bezüglich **code** expert

Wichtig bezüglich **code expert**

- Der Autograder kann sehr pingelig sein: Trennt ihr die Ziffern z.B. mit Zeilenumbrüchen (`\n`) statt mit Leerzeichen, wird das Programm nicht akzeptiert.

Wichtig bezüglich **code expert**

- Der Autograder kann sehr pingelig sein: Trennt ihr die Ziffern z.B. mit Zeilenumbrüchen (`\n`) statt mit Leerzeichen, wird das Programm nicht akzeptiert.
- XP gibt es *nur*, wenn der Autograder sie vergibt. Stellt also sicher, dass euer Code alle Tests besteht.

Wichtig bezüglich **code expert**

- Der Autograder kann sehr pingelig sein: Trennt ihr die Ziffern z.B. mit Zeilenumbrüchen (`\n`) statt mit Leerzeichen, wird das Programm nicht akzeptiert.
- XP gibt es *nur*, wenn der Autograder sie vergibt. Stellt also sicher, dass euer Code alle Tests besteht.
- Unklar, was bei der Ausführung genau passiert? Mit Python Tutor könnt ihr das Programm Schritt für Schritt ausführen.  [Python Tutor](#)

Fragen/Unklarheiten?

6. Logische Ausdrücke

Was sind Ausdrücke (“Expressions”)?

Expressions

...können evaluiert werden. Sie kommen oft in Form von mathematischen Ausdrücken vor, die man dann Stück für Stück evaluiert.

Präzedenz

Mit dem Typ `bool` werden Ausdrücke komplizierter. Manchmal muss man sie sorgfältig Schritt für Schritt auswerten und dabei auf *Präzedenz* und *Assoziativität* achten.

Mit dem Typ `bool` werden Ausdrücke komplizierter. Manchmal muss man sie sorgfältig Schritt für Schritt auswerten und dabei auf *Präzedenz* und *Assoziativität* achten.

Präzedenz (aus der Vorlesung)

Binäre arithmetische Operatoren (z.B. `+`) binden stärker als relationale Operatoren (z.B. `<`), und diese binden stärker als binäre logische Operatoren (z.B. `&&`).

`“+”` `>` `“<”` `>` `“&&”`

Beispiel

Wie können wir mit diesem Wissen den folgenden Ausdruck in Klammern setzen?

$(3 < (4 + 1) \&\& (2 < 3))$

Beispiel

Wie können wir mit diesem Wissen den folgenden Ausdruck in Klammern setzen?

$3 < 4 + 1 \ \&\& \ 2 < 3$

$(3 < (4 + 1)) \ \&\& \ (2 < 3)$

Beispiel

Wie können wir mit diesem Wissen den folgenden Ausdruck in Klammern setzen?

$3 < 4 + 1 \ \&\& \ 2 < 3$

$(3 < (4 + 1)) \ \&\& \ (2 < 3)$

Nun evaluieren wir ihn Schritt für Schritt (Tipp: Zuerst Klammern setzen hilft oft!)

Beispiel

Wie können wir mit diesem Wissen den folgenden Ausdruck in Klammern setzen?

$3 < 4 + 1 \ \&\& \ 2 < 3$

$(3 < (4 + 1)) \ \&\& \ (2 < 3)$

Nun evaluieren wir ihn Schritt für Schritt (Tipp: Zuerst Klammern setzen hilft oft!)

$(3 < 5) \ \&\& \ (2 < 3)$

Beispiel

Wie können wir mit diesem Wissen den folgenden Ausdruck in Klammern setzen?

`3 < 4 + 1 && 2 < 3`

`(3 < (4 + 1)) && (2 < 3)`

Nun evaluieren wir ihn Schritt für Schritt (Tipp: Zuerst Klammern setzen hilft oft!)

`(3 < 5) && (2 < 3)`

`true && (2 < 3)`

Beispiel

Wie können wir mit diesem Wissen den folgenden Ausdruck in Klammern setzen?

$3 < 4 + 1 \ \&\& \ 2 < 3$

$(3 < (4 + 1)) \ \&\& \ (2 < 3)$

Nun evaluieren wir ihn Schritt für Schritt (Tipp: Zuerst Klammern setzen hilft oft!)

$(3 < 5) \ \&\& \ (2 < 3)$

$\text{true} \ \&\& \ (2 < 3)$

$\text{true} \ \&\& \ \text{true}$

Beispiel

Wie können wir mit diesem Wissen den folgenden Ausdruck in Klammern setzen?

$3 < 4 + 1 \ \&\& \ 2 < 3$

$(3 < (4 + 1)) \ \&\& \ (2 < 3)$

Nun evaluieren wir ihn Schritt für Schritt (Tipp: Zuerst Klammern setzen hilft oft!)

$(3 < 5) \ \&\& \ (2 < 3)$

$\text{true} \ \&\& \ (2 < 3)$

$\text{true} \ \&\& \ \text{true}$

true

Beispiel

Manchmal gibt es zwei Operatoren mit demselben Vorrang, zum Beispiel

```
false && false && true.
```

Beispiel

Manchmal gibt es zwei Operatoren mit demselben Vorrang, zum Beispiel

```
false && false && true.
```

In diesem Fall müssen wir die Assoziativität berücksichtigen, d.h. `&&` ist ein *links*-assoziativer Operator. Das bedeutet hier, dass der linke der beiden Operatoren *zuerst* berechnet werden muss, d.h. die Klammern sind so zu setzen

Beispiel

Manchmal gibt es zwei Operatoren mit demselben Vorrang, zum Beispiel

```
false && false && true.
```

In diesem Fall müssen wir die Assoziativität berücksichtigen, d.h. `&&` ist ein *links*-assoziativer Operator. Das bedeutet hier, dass der linke der beiden Operatoren *zuerst* berechnet werden muss, d.h. die Klammern sind so zu setzen

```
(false && false) && true
```

6. Logische Ausdrücke

6.1. Kurzschlussauswertung (“Short Circuit”)

Kurzschlussauswertung (“Short-Circuit Evaluation”)

Für die Operatoren `&&` und `||` gibt es eine spezielle Regel namens *Kurzschlussauswertung*. Für diese Operatoren garantiert der Compiler, dass er immer den *linken Ausdruck* zuerst auswertet. Wenn das *Gesamtergebnis* allein aus dem linken Ausdruck abgeleitet werden kann, dann wird der rechte Ausdruck *nicht* ausgewertet, was unnötige Berechnungen erspart.

Kurzschlussauswertung (“Short-Circuit Evaluation”)

Für die Operatoren `&&` und `||` gibt es eine spezielle Regel namens *Kurzschlussauswertung*. Für diese Operatoren garantiert der Compiler, dass er immer den *linken Ausdruck* zuerst auswertet. Wenn das *Gesamtergebnis* allein aus dem linken Ausdruck abgeleitet werden kann, dann wird der rechte Ausdruck *nicht* ausgewertet, was unnötige Berechnungen erspart.

`2 > 3 && 17u - 55 <= ++x % y`

Kurzschlussauswertung (“Short-Circuit Evaluation”)

Für die Operatoren `&&` und `||` gibt es eine spezielle Regel namens *Kurzschlussauswertung*. Für diese Operatoren garantiert der Compiler, dass er immer den *linken Ausdruck* zuerst auswertet. Wenn das *Gesamtergebnis* allein aus dem linken Ausdruck abgeleitet werden kann, dann wird der rechte Ausdruck *nicht* ausgewertet, was unnötige Berechnungen erspart.

```
2 > 3 && 17u - 55 <= ++x % y
```

```
(2 > 3) && (17u - 55 <= ++x % y)
```

Kurzschlussauswertung (“Short-Circuit Evaluation”)

Für die Operatoren `&&` und `||` gibt es eine spezielle Regel namens *Kurzschlussauswertung*. Für diese Operatoren garantiert der Compiler, dass er immer den *linken Ausdruck* zuerst auswertet. Wenn das *Gesamtergebnis* allein aus dem linken Ausdruck abgeleitet werden kann, dann wird der rechte Ausdruck *nicht* ausgewertet, was unnötige Berechnungen erspart.

```
2 > 3 && 17u - 55 <= ++x % y
```

```
(2 > 3) && (17u - 55 <= ++x % y)
```

```
false && (17u - 55 <= ++x % y)
```

Kurzschlussauswertung (“Short-Circuit Evaluation”)

Für die Operatoren `&&` und `||` gibt es eine spezielle Regel namens *Kurzschlussauswertung*. Für diese Operatoren garantiert der Compiler, dass er immer den *linken Ausdruck* zuerst auswertet. Wenn das *Gesamtergebnis* allein aus dem linken Ausdruck abgeleitet werden kann, dann wird der rechte Ausdruck *nicht* ausgewertet, was unnötige Berechnungen erspart.

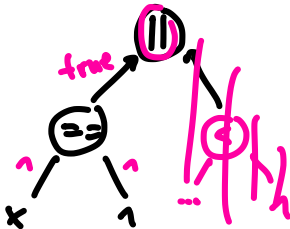
```
2 > 3 && 17u - 55 <= ++x % y  
(2 > 3) && (17u - 55 <= ++x % y)  
false && (17u - 55 <= ++x % y)  
false
```

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

$$(x == 1) \mid \mid ((1 / (x - 1)) < 1)$$

Lösung



Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

$$x == 1 \ || \ 1 / (x - 1) < 1$$

Lösung

$$(x == 1) \ || \ ((1 / (x - 1)) < 1)$$

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

$$x == 1 \ || \ 1 / (x - 1) < 1$$

Lösung

$$(x == 1) \ || \ ((1 / (x - 1)) < 1)$$
$$(1 == 1) \ || \ ((1 / (x - 1)) < 1)$$

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

$$x == 1 \ || \ 1 / (x - 1) < 1$$

Lösung

```
(x == 1) || ((1 / (x - 1)) < 1)
```

```
(1 == 1) || ((1 / (x - 1)) < 1)
```

```
true || ((1 / (x - 1)) < 1)
```

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`x == 1 || 1 / (x - 1) < 1`

Lösung

`(x == 1) || ((1 / (x - 1)) < 1)`

`(1 == 1) || ((1 / (x - 1)) < 1)`

`true || ((1 / (x - 1)) < 1)`

`true`

$a = (b = (c = 7))$



`int a = 7 + 3;`
`int a = 10;`

`10`
`a`
`int`

`7` `7` `7`
`c` `b` `a`

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`x == 1 || 1 / (x - 1) < 1`

Lösung

`(x == 1) || ((1 / (x - 1)) < 1)`

`(1 == 1) || ((1 / (x - 1)) < 1)`

`true || ((1 / (x - 1)) < 1)`

`true` *Short Circuit!*

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`!(1 < 2 && x == 1) + 1`

Lösung

¹
"Verneinung"
`!true = false`

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

$$!(1 < 2 \ \&\& \ x == 1) + 1$$

Lösung

$$(!((1 < 2) \ \&\& \ (x == 1))) + 1$$

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`!(1 < 2 && x == 1) + 1`

Lösung

`(!((1 < 2) && (x == 1))) + 1`

`(!(true && (x == 1))) + 1`

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`!(1 < 2 && x == 1) + 1`

Lösung

`(!((1 < 2) && (x == 1))) + 1`

`(!(true && (x == 1))) + 1`

`(!(true && true)) + 1`

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`!(1 < 2 && x == 1) + 1`

Lösung

`(!((1 < 2) && (x == 1))) + 1`

`(!(true && (x == 1))) + 1`

`(!(true && true)) + 1`

`(!true) + 1`

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`!(1 < 2 && x == 1) + 1`

Lösung

`(!((1 < 2) && (x == 1))) + 1`

`(!(true && (x == 1))) + 1`

`(!(true && true)) + 1`

`(!true) + 1`

`false + 1`

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`!(1 < 2 && x == 1) + 1`

Lösung

`(!((1 < 2) && (x == 1))) + 1`

`(!(true && (x == 1))) + 1`

`(!(true && true)) + 1`

`(!true) + 1`

`false + 1`

`0 + 1`

Quiz

Aufgabe Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte. Ihr könnt davon ausgehen, dass `x` eine Variable vom Typ `int` mit dem Wert `1` ist.

`!(1 < 2 && x == 1) + 1`

Lösung

`!( (1 < 2) && (x == 1) ) + 1`

`!(true && (x == 1)) + 1`

`!(true && true) + 1`

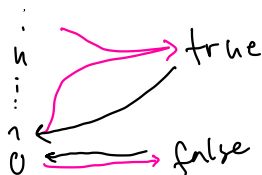
`!true + 1`

false (+) 1

0 + 1

1

return type
input types
`int` `operator +` (`int a`, `int b`)



Quiz

Aufgabe Minimiert den folgenden booleschen Ausdruck, wobei a und b Variablen vom Typ `bool` sind und n ein `int` ist:

Quiz

Aufgabe Minimiert den folgenden booleschen Ausdruck, wobei `a` und `b` Variablen vom Typ `bool` sind und `n` ein `int` ist:

1. `b == true && !(n < 0) && (n != 0) && (n != 6) && (n < 6)`
2. `a && !(a == true) || (b == false)`
3. `!(a != false && !(b == false)) == false`

Quiz

Aufgabe Minimiert den folgenden booleschen Ausdruck, wobei `a` und `b` Variablen vom Typ `bool` sind und `n` ein `int` ist:

1. `b == true && !(n < 0) && (n != 0) && (n != 6) && (n < 6)`
2. `a && !(a == true) || (b == false)`
3. `!(a != false && !(b == false)) == false`

Lösung

Quiz

Aufgabe Minimiert den folgenden booleschen Ausdruck, wobei `a` und `b` Variablen vom Typ `bool` sind und `n` ein `int` ist:

1. `b == true && !(n < 0) && (n != 0) && (n != 6) && (n < 6)`
2. `a && !(a == true) || (b == false)`
3. `!(a != false && !(b == false)) == false`

Lösung

1. `b && n > 0 && n < 6`

Quiz

Aufgabe Minimiert den folgenden booleschen Ausdruck, wobei `a` und `b` Variablen vom Typ `bool` sind und `n` ein `int` ist:

1. `b == true && !(n < 0) && (n != 0) && (n != 6) && (n < 6)`
2. `a && !(a == true) || (b == false)`
3. `!(a != false && !(b == false) == false)`

Lösung

1. `b && n > 0 && n < 6`
2. `!b`

Quiz

Aufgabe Minimiert den folgenden booleschen Ausdruck, wobei `a` und `b` Variablen vom Typ `bool` sind und `n` ein `int` ist:

1. `b == true && !(n < 0) && (n != 0) && (n != 6) && (n < 6)`
2. `a && !(a == true) || (b == false)`
3. `!(a != false && !(b == false) == false)`

Lösung

1. `b && n > 0 && n < 6`
2. `!b`
3. `!a || b`

Fragen/Unklarheiten?

7. Ausdrücke mit gemischten Typen

Types

Bislang haben wir in den Vorlesungen drei Gruppen von Typen gesehen:

- Logische Variablen: `bool`;
- Ganzzahlen: `int`, `unsigned int`;
- Fließkommazahlen: `float`, `double`.

Manchmal sind mehrere Typen in einer Expression. Wie interagieren verschiedene Typen miteinander?

Types

Bislang haben wir in den Vorlesungen drei Gruppen von Typen gesehen:

- Logische Variablen: `bool`;
- Ganzzahlen: `int`, `unsigned int`;
- Fließkommazahlen: `float`, `double`.

Manchmal sind mehrere Typen in einer Expression. Wie interagieren verschiedene Typen miteinander?

Generalitätsreihenfolge der Typen

Types

Bislang haben wir in den Vorlesungen drei Gruppen von Typen gesehen:

- Logische Variablen: `bool`;
- Ganzzahlen: `int`, `unsigned int`;
- Fließkommazahlen: `float`, `double`.

Manchmal sind mehrere Typen in einer Expression. Wie interagieren verschiedene Typen miteinander?

Generalitätsreihenfolge der Typen

`bool <`

Types

Bislang haben wir in den Vorlesungen drei Gruppen von Typen gesehen:

- Logische Variablen: `bool`;
- Ganzzahlen: `int`, `unsigned int`;
- Fließkommazahlen: `float`, `double`.

Manchmal sind mehrere Typen in einer Expression. Wie interagieren verschiedene Typen miteinander?

Generalitätsreihenfolge der Typen

```
bool < int < unsigned int <
```

Types

Bislang haben wir in den Vorlesungen drei Gruppen von Typen gesehen:

- Logische Variablen: `bool`;
- Ganzzahlen: `int`, `unsigned int`;
- Fließkommazahlen: `float`, `double`.

Manchmal sind mehrere Typen in einer Expression. Wie interagieren verschiedene Typen miteinander?

Generalitätsreihenfolge der Typen

`bool < int < unsigned int < float < double`

Typen konvertieren immer zum generelleren Typen der Subexpression

Wie man sich Types vorstellen kann

Type **Literal** **Approximiert**

Wie man sich Types vorstellen kann

Type

`bool`

Literal

Approximiert

`{false, true}`

Wie man sich Types vorstellen kann

Type	Literal	Approximiert
<code>bool</code>		<code>{false, true}</code>
<code>unsigned int</code>	<code>u</code>	$\mathbb{N}$

Wie man sich Types vorstellen kann

Type	Literal	Approximiert
<code>bool</code>		<code>{false, true}</code>
<code>unsigned int</code>	<code>u</code>	$\mathbb{N}$
<code>int</code>		$\mathbb{Z}$

Wie man sich Types vorstellen kann

Type	Literal	Approximiert
bool		{false, true}
unsigned int	u	$\mathbb{N}$
int		$\mathbb{Z}$
float	f	$\mathbb{R}$

Wie man sich Types vorstellen kann

Type	Literal	Approximiert
bool		{false, true}
unsigned int	u	$\mathbb{N}$
int		$\mathbb{Z}$
float	<u>f</u>	$\mathbb{R}$
double		$\mathbb{R}$, aber <i>double</i> Präzision

5u ← unsigned int
5.0f ← double
5.0 ← int
+ 0.0 ← double

Types evaluieren I

```
std::cout << 5.0/2 << std::endl;  
// what type and value will this return and why?
```

Types evaluieren I

```
std::cout << 5.0/2 << std::endl;  
// what type and value will this return and why?
```

Lösung

`double`, `2.5`, weil die `int` `2` zuerst in eine `double` `2.0` konvertiert wird, um diese Expression zu berechnen.

Types evaluieren II

```
std::cout << (1/2)*5.0/2 << std::endl;  
// what type and value will this return and why?
```

Types evaluieren II

```
std::cout << (1/2)*5.0/2 << std::endl;  
// what type and value will this return and why?
```

Lösung

double, 0, weil zuerst die linke Expression $1/2$ evaluiert wird, welche zu 0 evaluiert (Integer-Division). Der Rest ist trivial, weil $0*$ irgendetwas evaluiert zu 0. Aber diese 0 wird vom Type **double** sein.

Literale

Literale

Es gibt bestimmte Buchstaben, die der Compiler mit bestimmten Types verbindet. Wenn ihr dem Compiler sagen möchtet *“Hey, don’t treat this 2.0 as a **double**, but instead as a **float**”* müsst ihr ein `f` am Ende des Werts hinzufügen. Etwa so:

Literale

Es gibt bestimmte Buchstaben, die der Compiler mit bestimmten Types verbindet. Wenn ihr dem Compiler sagen möchtet *“Hey, don’t treat this 2.0 as a **double**, but instead as a **float**”* müsst ihr ein **f** am Ende des Werts hinzufügen. Etwa so:

```
std::cout << (5/2)*5.0f/2 << std::endl;
```

Literale

Es gibt bestimmte Buchstaben, die der Compiler mit bestimmten Types verbindet. Wenn ihr dem Compiler sagen möchtet *“Hey, don’t treat this 2.0 as a **double**, but instead as a **float**”* müsst ihr ein **f** am Ende des Werts hinzufügen. Etwa so:

```
std::cout << (5/2)*5.0f/2 << std::endl;
```

Analog dazu macht ein angehängtes **u** aus einem **int**-Literal ein **unsigned int**-Literal, z.B. 17u.

Types evaluieren III

```
std::cout << (5/2)*5.0f/2 << std::endl;  
// what type and value will this return and why?
```

Types evaluieren III

```
std::cout << (5/2)*5.0f/2 << std::endl;  
// what type and value will this return and why?
```

Lösung

`float`, `5.0`, (kann als `5.0f` geschrieben werden).

Zuerst, wird `5/2` evaluiert, was zu `2` wird (integer division). Dann wird `2.0f*5.0f` berechnet: Die `int 2` wurde zu einer `float 2`, weil `float` der generellere Type (in dieser Expression) ist. Dito für `/2` später.

Exercise I

1. Welche der folgenden Zeichenfolgen sind keine C++ Expression, und warum nicht? Hierbei seien `x` und `y` Variablen vom Typ `int`.
- a) `(y++ < 0 && y < 0) + 2.0`
 - b) `y = (x++ = 3)`
 - c) `3.0 + 3 - 4 + 5`
 - d) `5 % 4 * 3.0 + true * x++`
- Handwritten notes and diagrams:
- A pink arrow points from the expression `(x++ = 3)` in option b) to the right.
 - Handwritten code snippets:
 - `y = x++ = 3`
 - `y = (x++ = 3)` (underlined in pink)
 - A diagram showing a pink box containing the number `3`. Below the box is the text `int x`. An arrow points from the box to a symbol that looks like a combination of `&` and `3`.
2. Für alle gültigen Expression, die oben identifiziert wurden, entscheidet, ob es sich um L-Werte oder R-Werte handelt und begründet eure Entscheidung.
3. Bestimmt die Werte der Ausdrücke und erklärt, wie diese Werte erhalten werden. Angenommen, dass anfangs `x == 1` und `y == -1`.

Expression Evaluation - Lösungen a)

`(y++ < 0 && y < 0) + 2.0`

Expression Evaluation - Lösungen a)

`(y++ < 0 && y < 0) + 2.0`

`(-1 < 0 && y < 0) + 2.0 // after this step: y==0`

Expression Evaluation - Lösungen a)

`(y++ < 0 && y < 0) + 2.0`

`(-1 < 0 && y < 0) + 2.0` // after this step: `y==0`

`(true && y < 0) + 2.0`

Expression Evaluation - Lösungen a)

`(y++ < 0 && y < 0) + 2.0`

`(-1 < 0 && y < 0) + 2.0` // after this step: `y==0`

`(true && y < 0) + 2.0`

`(true && false) + 2.0`

Expression Evaluation - Lösungen a)

`(y++ < 0 && y < 0) + 2.0`

`(-1 < 0 && y < 0) + 2.0` // after this step: `y==0`

`(true && y < 0) + 2.0`

`(true && false) + 2.0`

`(false) + 2.0`

Expression Evaluation - Lösungen a)

`(y++ < 0 && y < 0) + 2.0`

`(-1 < 0 && y < 0) + 2.0` // after this step: `y==0`

`(true && y < 0) + 2.0`

`(true && false) + 2.0`

`(false) + 2.0`

`0.0 + 2.0`

Expression Evaluation - Lösungen a)

`(y++ < 0 && y < 0) + 2.0`

`(-1 < 0 && y < 0) + 2.0` // after this step: `y==0`

`(true && y < 0) + 2.0`

`(true && false) + 2.0`

`(false) + 2.0`

`0.0 + 2.0`

`2.0`

Expression Evaluation - Lösungen a)

`(y++ < 0 && y < 0) + 2.0`

`(-1 < 0 && y < 0) + 2.0` // after this step: `y==0`

`(true && y < 0) + 2.0`

`(true && false) + 2.0`

`(false) + 2.0`

`0.0 + 2.0`

`2.0`

R-Wert

Expression Evaluation - Lösungen b)

`y = (x++ = 3)`

Expression Evaluation - Lösungen b)

`y = (x++ = 3)`

Invalid

Expression Evaluation - Lösungen b)

`y = (x++ = 3)`

Invalid ...aber wieso?

Expression Evaluation - Lösungen b)

`y = (x++ = 3)`

Invalid ... aber wieso?

Weil `x++` einen R-Wert zurückgibt und wir einem R-Wert nichts zuweisen (`=`) können.

Expression Evaluation - Lösungen c)

$$3.0 + 3 - 4 + 5$$

Expression Evaluation - Lösungen c)

$3.0 + 3 - 4 + 5$

$((3.0 + 3) - 4) + 5$

Expression Evaluation - Lösungen c)

$$3.0 + 3 - 4 + 5$$

$$((3.0 + 3) - 4) + 5$$

$$((3.0 + 3.0) - 4) + 5$$

Expression Evaluation - Lösungen c)

$$3.0 + 3 - 4 + 5$$

$$((3.0 + 3) - 4) + 5$$

$$((3.0 + 3.0) - 4) + 5$$

$$(6.0 - 4) + 5$$

Expression Evaluation - Lösungen c)

$$3.0 + 3 - 4 + 5$$

$$((3.0 + 3) - 4) + 5$$

$$((3.0 + 3.0) - 4) + 5$$

$$(6.0 - 4) + 5$$

$$(6.0 - 4.0) + 5$$

Expression Evaluation - Lösungen c)

$$3.0 + 3 - 4 + 5$$

$$((3.0 + 3) - 4) + 5$$

$$((3.0 + 3.0) - 4) + 5$$

$$(6.0 - 4) + 5$$

$$(6.0 - 4.0) + 5$$

$$2.0 + 5$$

Expression Evaluation - Lösungen c)

$$3.0 + 3 - 4 + 5$$

$$((3.0 + 3) - 4) + 5$$

$$((3.0 + 3.0) - 4) + 5$$

$$(6.0 - 4) + 5$$

$$(6.0 - 4.0) + 5$$

$$2.0 + 5$$

$$2.0 + 5.0$$

Expression Evaluation - Lösungen c)

$3.0 + 3 - 4 + 5$

$((3.0 + 3) - 4) + 5$

$((3.0 + 3.0) - 4) + 5$

$(6.0 - 4) + 5$

$(6.0 - 4.0) + 5$

$2.0 + 5$

$2.0 + 5.0$

7.0

Expression Evaluation - Lösungen c)

$3.0 + 3 - 4 + 5$

$((3.0 + 3) - 4) + 5$

$((3.0 + 3.0) - 4) + 5$

$(6.0 - 4) + 5$

$(6.0 - 4.0) + 5$

$2.0 + 5$

$2.0 + 5.0$

7.0

R-Wert

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

Expression Evaluation - Lösungen d)

`5 % 4 * 3.0 + true * x++`

`((5 % 4) * 3.0) + (true * (x++))`

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

(1.0 * 3.0) + (true * (x++))

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

(1.0 * 3.0) + (true * (x++))

3.0 + (true * (x++))

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

(1.0 * 3.0) + (true * (x++))

3.0 + (true * (x++))

3.0 + (true * 1)

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

(1.0 * 3.0) + (true * (x++))

3.0 + (true * (x++))

3.0 + (true * 1)

3.0 + (1 * 1)

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

(1.0 * 3.0) + (true * (x++))

3.0 + (true * (x++))

3.0 + (true * 1)

3.0 + (1 * 1)

3.0 + 1

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

(1.0 * 3.0) + (true * (x++))

3.0 + (true * (x++))

3.0 + (true * 1)

3.0 + (1 * 1)

3.0 + 1

3.0 + 1.0

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

(1.0 * 3.0) + (true * (x++))

3.0 + (true * (x++))

3.0 + (true * 1)

3.0 + (1 * 1)

3.0 + 1

3.0 + 1.0

4.0

Expression Evaluation - Lösungen d)

5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))

(1 * 3.0) + (true * (x++))

(1.0 * 3.0) + (true * (x++))

3.0 + (true * (x++))

3.0 + (true * 1)

3.0 + (1 * 1)

3.0 + 1

3.0 + 1.0

4.0

R-Wert

Fragen/Unklarheiten?

8. Gültige und ungültige Ausdrücke

Beispiel

Evaluiert den Ausdruck $5u + 5 * 3u$

Beispiel

Evaluiert den Ausdruck $5u + 5 * 3u$

$$5u + 5 * 3u$$

Beispiel

Evaluiert den Ausdruck $5u + 5 * 3u$

$5u + 5 * 3u$ Punkt vor Strich

Beispiel

Evaluiert den Ausdruck $5u + 5 * 3u$

$5u + 5 * 3u$ Punkt vor Strich

$5u + (5 * 3u)$

Beispiel

Evaluiert den Ausdruck `5u + 5 * 3u`

`5u + 5 * 3u` Punkt vor Strich

`5u + (5 * 3u)` das Resultat wird zu `unsigned int`

Beispiel

Evaluiert den Ausdruck $5u + 5 * 3u$

$5u + 5 * 3u$ Punkt vor Strich

$5u + (5 * 3u)$ das Resultat wird zu **unsigned int**

$5u + 15u$

Beispiel

Evaluiert den Ausdruck $5u + 5 * 3u$

$5u + 5 * 3u$ Punkt vor Strich

$5u + (5 * 3u)$ das Resultat wird zu **unsigned int**

$5u + 15u$ simple Addition

Beispiel

Evaluiert den Ausdruck $5u + 5 * 3u$

$5u + 5 * 3u$ Punkt vor Strich

$5u + (5 * 3u)$ das Resultat wird zu **unsigned int**

$5u + 15u$ simple Addition

$20u$

Valide Expression erkennen

Welche sind eine valide C++-Expression und welche nicht?

1. $1 * (2 * 3)$

2. $(a = 1)$

$x = \underbrace{(a = 1)}_1;$

3. $(1$

4. $\underbrace{(a * 3)}_{\text{R-value}} = \underbrace{(b * 5)}$

Valide Expression erkennen

Welche sind eine valide C++-Expression und welche nicht?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

Valide Expression erkennen

Welche sind eine valide C++-Expression und welche nicht?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

- 3. ist nicht valide, da die Klammer $($ nicht geschlossen wird

Valide Expression erkennen

Welche sind eine valide C++-Expression und welche nicht?

1. $1*(2*3)$

2. $(a=1)$

3. $(1$

4. $(a*3) = (b*5)$

Lösungen

- 3. ist nicht valide, da die Klammer $()$ nicht geschlossen wird
- 4. ist nicht valide, da $(a*3)$ zu einem r-value wird, der Assignment-Operator $(=)$ aber ein l-value zu seiner Linken erwartet

Valide Expression erkennen

Welche sind eine valide C++-Expression und welche nicht?

1. $1*(2*3)$

2. $(a=1)$

3. $(1$

4. $(a*3) = (b*5)$

Lösungen

- 3. ist nicht valide, da die Klammer $()$ nicht geschlossen wird
- 4. ist nicht valide, da $(a*3)$ zu einem r-value wird, der Assignment-Operator $(=)$ aber ein l-value zu seiner Linken erwartet
- 1. und 2. sind valide C++-Expressions

Valide Expression erkennen

Welche sind L-Werte und welche sind R-Werte?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Valide Expression erkennen

Welche sind L-Werte und welche sind R-Werte?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

Valide Expression erkennen

Welche sind L-Werte und welche sind R-Werte?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

- 3. und 4. sind weder noch, da sie keine validen Expressions sind

Valide Expression erkennen

Welche sind L-Werte und welche sind R-Werte?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

- 3. und 4. sind weder noch, da sie keine validen Expressions sind
- 1. ist ein r-value, da der Multiplikation-Operator (*) ein r-value zurückgibt

Valide Expression erkennen

Welche sind L-Werte und welche sind R-Werte?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

- 3. und 4. sind weder noch, da sie keine validen Expressions sind
- 1. ist ein r-value, da der Multiplikation-Operator ($*$) ein r-value zurückgibt
- 2. ist ein l-Value, da der Assignment-Operator ($=$) ein l-Value zurückgibt

Valide Expression erkennen

Zu was werden die Expressions evaluiert?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Valide Expression erkennen

Zu was werden die Expressions evaluiert?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

Valide Expression erkennen

Zu was werden die Expressions evaluiert?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

- 3. und 4. können nicht evaluiert werden, da sie keine validen Expressions sind

Valide Expression erkennen

Zu was werden die Expressions evaluiert?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

- 3. und 4. können nicht evaluiert werden, da sie keine validen Expressions sind
- 1. evaluiert zu 6, also das Ergebnis der simplen Multiplikationen

Valide Expression erkennen

Zu was werden die Expressions evaluiert?

1. $1 * (2 * 3)$

2. $(a = 1)$

3. $(1$

4. $(a * 3) = (b * 5)$

Lösungen

- 3. und 4. können nicht evaluiert werden, da sie keine validen Expressions sind
- 1. evaluiert zu 6, also das Ergebnis der simplen Multiplikationen
- 2. evaluiert zu 1, da a zurückgegeben wird, dem gerade davor der Wert 1 zugewiesen wurde

Warum sind R-Werte und L-Werte wichtig?

Warum sind R-Werte und L-Werte wichtig?

Später im Kurs werden wir numerische Operatoren (wie `operator+` oder `operator=`) für neue numerische Typen (wie eigene Version von `std::complex`) implementieren müssen. Diese Typen und ihre Operatoren sollten sich wie die eingebauten Typen verhalten. So sollte `operator=` eine Rückgabe `*this` machen, aber `operator+` nicht.

9. Outro

Bis nächste Woche!