

Informatik 2

Assignment Project Exam Help

<https://tutorcs.com>
02 – Modulare Integer-Arithmetik

WeChat: cstutorcs

Sommer 2021

Torsten Grust
Universität Tübingen, Germany

1 | Ein Programm für die Ewigkeit...

Das Prädikat `x >= 0` des `while`-Loops in Programm `eternity.c0` sollte eigentlich immer `true` ergeben:

```
#use <conio>

int main() {
    int x = 0;

    while (x >= 0) {
        x = x + 1;
    }
    /* -----8<-----cut here----->8-----*/
    println("We'll never get here!");

    return 0;
}
```

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

⚠ Das Ergebnis mag überraschen. Was *genau* passiert hier?

2 | Bit-Darstellung von positiven Integers

Intern repräsentiert C0 einen Wert $x \geq 0$ des Typs `int` durch Sequenzen von n Bits b_i (auch: *binary digit*, $b_i \in \{0,1\}$):¹

$$\ast^+ \quad x = \sum_{i=0}^{n-1} (2^i \times b_i)$$

Wertigkeit von...
 • Bit b_0 : $2^0 = 1$
 • Bit b_{i+1} : $2 \times$ Wertigkeit von b_i

<https://tutorcs.com>

- Vergleiche mit Dezimalsystem und Ziffernwertigkeit 10^i .

WeChat: [cstutorcs](#)

- **Beispiel:** Interne Darstellung von $x = 214$ (sei $n = 8$):

| 2^i | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | \ast^+ |
|-------|-----|----|----|----|---|---|---|---|----------|
| b_i | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 214 |

¹ NB: \ast^+ ist nicht die endgültige Repräsentation. Wir modifizieren diese auf den kommenden Slides.

(Einschub: `#use "file"`)

- Ein **Pragma** `#use "file"` wird vor Compilation und Interpretation durch den Inhalt des Files `file` ersetzt:

| | | |
|---|---|---|
| <pre> : /* before #use */ #use "file" /* after #use */ : </pre> | <div style="border-left: 1px solid black; border-right: 1px solid black; height: 100px; margin: 0 auto; width: 2px;"></div> | <pre> : /* before #use */ [Inhalt von file] /* after #use */ : </pre> |
|---|---|---|

- Fehler werden weiterhin in Quelltext **1** lokalisiert.
- `#use ...` muss jeder weiteren Deklaration vorausgehen.
- Mehrfache identische `#use ...` Pragmas werden—auch bei Schachtelung—erkannt und ignoriert. (Danke, C0!)

3 | C0: 32-Bit Integers

- Jede reale Sprache/Maschine begrenzt die Länge n der Bitsequenz zur Darstellung von Integers. In C0: $n = 32$.
 - Damit gilt $\text{int} \subsetneq \mathbb{Z}$ und unendliche viele ganze Zahlen x sind nicht darstellbar.
Assignment Project Exam Help
 - Und: Die Integer-Repräsentation (\ast^+ , Slide 03) stellt $x \geq 0$ ($\in \mathbb{N}$) dar.
<https://tutorcs.com> **Darstellung negativer Zahlen $x < 0$?**
WeChat: cstutores
- Programm `eternity.c0` endet mit $x = -2147483648$.
 \Rightarrow C0 setzt *nicht* \ast^+ als Repräsentation für Typ `int` ein.
 - 💡 Nutze `printbits(x)` für $x < 0$.

Bit-Darstellung von negativen + positiven Integers

Interne Darstellung eines Werts $x \leq 0$ des Typs *int* durch Sequenz von n Bits b_i (Vorzeichenbit b_{n-1}):

$$\ast \quad x = -2^{n-1} \times b_{n-1} + \underbrace{\sum_{i=0}^{n-2} (2^i \times b_i)}_{n-1 \text{ Bits}}$$

Assignment Project Exam Help
<https://tutores.com>
 Vorzeichenbit

- Darstellbarer Wertebereich mit n Bits [C0: $n = 32$]:

| | | | |
|-----------------|---|---------------------|-----------------|
| Minimum: | 10000...0000 | $\equiv -2^{n-1}$ | $[-2147483648]$ |
| | 0: 000000000000 | | |
| Maximum: | 01111...1111 | $\equiv +2^{n-1}-1$ | $[+2147483647]$ |
| | <div style="border-top: 1px solid black; width: 150px; margin: 0 auto;"></div> <div style="text-align: center;">$n-1$ Bits</div> | | |

Bit-Darstellung von negativen + positiven Integers

Beispiele ($n = 8$ Bits):

| $(-)2^i$ | -128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | * |
|-----------|------|----|----|----|---|---|---|---|-----|
| b_i | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | -42 |

$$x = -42$$

| $(-)2^i$ | -128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | * |
|-----------|------|----|----|----|---|---|---|---|-----|
| b_i | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 127 |

$$x = 127 = 2^{8-1} - 1 \text{ (Maximum)}$$

| $(-)2^i$ | -128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | * |
|-----------|------|----|----|----|---|---|---|---|------|
| b_i | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -128 |

$$x = -128 = -2^{8-1} \text{ (Minimum)}$$

- Q: Darstellung von -1?

Zweierkomplement: Negation eines Integers

Q: Gegeben ein Integer x , wie berechnet sich dann $-x$?

A: Durch das **Zweierkomplement** (auch *two's complement*):

$$x \equiv b_{n-1} b_{n-2} \cdots b_1 b_0$$

$$-x \equiv \bar{b}_{n-1} \bar{b}_{n-2} \cdots \bar{b}_1 \bar{b}_0 + 1 \quad \left(\begin{array}{l} \text{Bit-Invertierung:} \\ 0 = 1, 1 = 0 \end{array} \right)$$

<https://tutorcs.com>

Beispiel ($n = 8$, $x = -42$):²

WeChat: cstutorcs

| $(-)2^i$ | -128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | * |
|-------------|------|----|----|----|---|---|---|---|-----|
| b_i | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | -42 |
| \bar{b}_i | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 41 |
| +1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 42 |

² Addition von Bits: $0+0 = 0$, $1+0 = 1$, $0+1 = 1$, $1+1 = 10$ (Übertrag 1).

Zweierkomplement: Negation eines Integers

Check: Gilt mit dem Zweierkomplement $x + (-x) = 0$?

$$\begin{aligned}
 & \underbrace{x}_{\text{X}} + \underbrace{-x}_{-X} \\
 &= -2^{n-1} \times b_{n-1} + \sum_{i=0}^{n-2} (2^i \times b_i) - 2^{n-1} \times \bar{b}_{n-1} + \sum_{i=0}^{n-2} (2^i \times \bar{b}_i) + 1 \\
 &= -2^{n-1} \times \underbrace{(b_{n-1} + \bar{b}_{n-1})}_{=1} + \sum_{i=0}^{n-2} (2^i \times \underbrace{(b_i + \bar{b}_i)}_{=1}) + 1 \\
 &= -2^{n-1} + \sum_{i=0}^{n-2} (2^i) + 1 \\
 &= -2^{n-1} + (2^{n-1} - 1) + 1 \\
 &= 0 \quad \checkmark
 \end{aligned}$$

4 : Modulare Arithmetik

Q: Wie verhalten sich die arithmetischen Operatoren (+, -, *), falls ihr Ergebnis nicht mit n (= 32) Bits darstellbar ist?

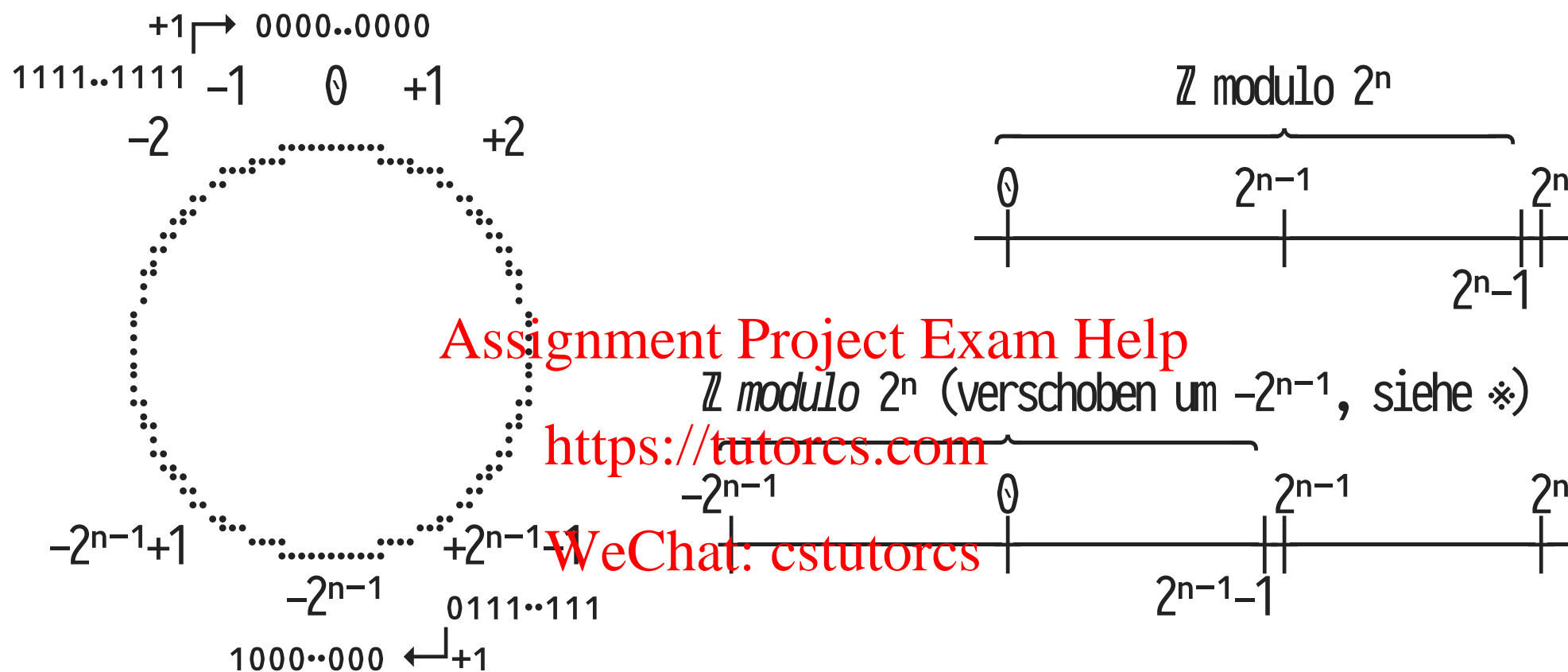
- **Option 1:** Teste jeweils auf Über-/Unterlauf, löse Ausnahmebehandlung (auch: *overflow exception*).
 - Kostspielig zur Programmlaufzeit.
 - Konsequenz: dann gilt u.a. $(x + x) - x \neq x + (x - x)$.
- **Option 2:** Führe alle Operationen *modulo* 2^n aus.
 - Keine Tests, arithmetische Operation \equiv CPU-Instruktion.
 - Auch in \mathbb{Z}_p (Ring “ \mathbb{Z} modulo p ”) gelten viele erwartete algebraische Äquivalenzen.



Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

Modulare Arithmetik



-  *Wrap around*: Für $x = \text{int_max} = 2^{31} - 1$, ist $x + 1 < x$ ( Programm `eternity.c0`).

Modulare Arithmetik bewahrt algebraische Äquivalenzen

Im Ring $\mathbb{Z} \text{ modulo } 2^n$ gelten viele algebraische Äquivalenzen:

| | Äquivalenz |
|-------------------------------|--------------------------------------|
| $x + y = y + x$ | Kommutativität der Addition |
| $(x + y) + z = x + (y + z)$ | Assoziativität der Addition |
| $x + 0 = x$ | neutrales Element der Addition |
| $x + (-x) = 0$ | additives Inverses |
| $-(-x) = x$ | |
| $x * y = y * x$ | Kommutativität der Multiplikation |
| $(x * y) * z = x * (y * z)$ | Assoziativität der Multiplikation |
| $x * 1 = x$ | neutrales Element der Multiplikation |
| $x * (y + z) = x * y + x * z$ | Distributivität |
| $x * 0 = 0$ | |

- Assoziativität und Distributivität gelten i.A. *nicht* für Arithmetik auf Fließkommazahlen (C: `float`, `double`).

5 | C0: Bit-Operatoren

- Die vier **Bit-Operatoren** \sim , $\&$, $|$ und \wedge operieren jeweils auf allen 32 Bits ihrer Argumente des Typs `int`:

Bitweises Invertieren

| \sim | |
|--------|---|
| 0 | 1 |
| 1 | 0 |

`int ~(int)`

Bitweises And

| $\&$ | 0 | 1 |
|------|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

`int &(int,int)`

Bitweises Or

| $ $ | 0 | 1 |
|-----|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

`int |(int,int)`

Bitweises Xor

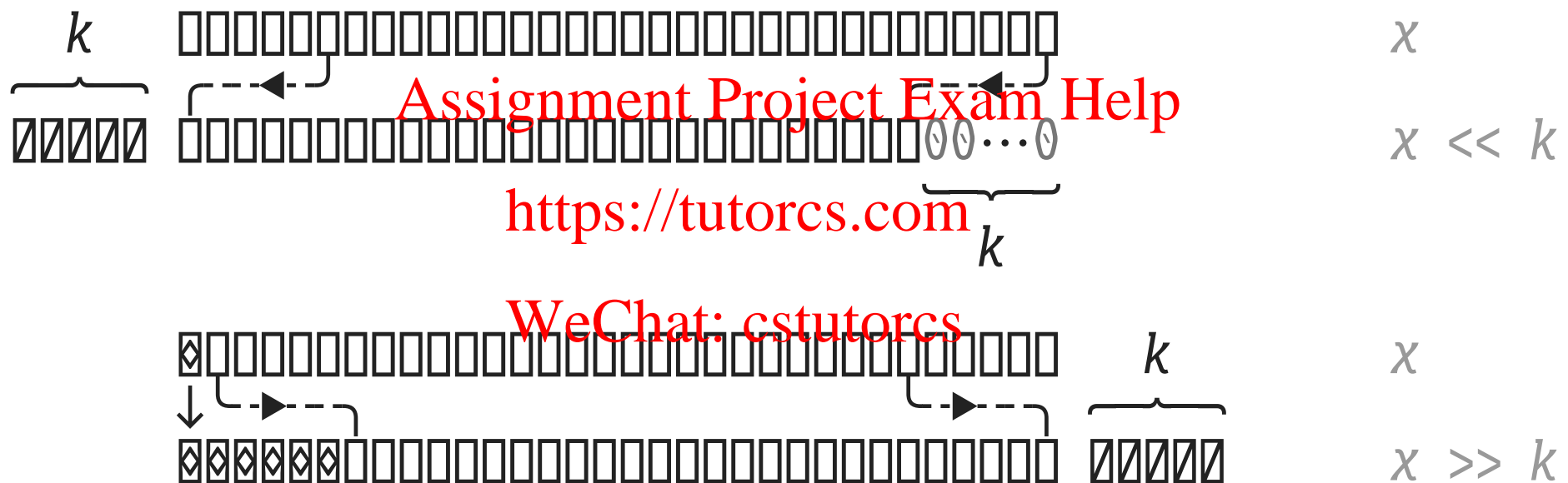
| \wedge | 0 | 1 |
|----------|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

`int ^(int,int)`

- Erinnerung *Informatik 1*: Mittels \sim und $\&$ lassen sich bereits *alle* 16 Booleschen/Bit-Operatoren ausdrücken.

C0: Bit-Shifting

- Die **Bit-Shift-Operatoren** \textcircled{E} $x \ll k$ ($x \gg k$) verschieben die 32 Bits des Operanden x um k Bits nach links (rechts):³



- NB:** Damit gilt $x \ll k = x * 2^k$ und $x \gg k = \lfloor x / 2^k \rfloor$ (Rundung Richtung $-\infty$. Erinnerung: $/$ rundet zur 0).

³ $0 \leq k < 32$.

C0: Update der Operator-Tabelle

| Priorität | Operatoren | Assoziativität | |
|-----------|---------------------------------------|----------------|------------------------------|
| 13 | <code>-, ~</code> | rechts | Negation/Invertierung (unär) |
| 12 | <code>*, /, %</code> | links | |
| 11 | <code>+, -</code> | links | |
| 10 | <code><<, >></code> | links | |
| 9 | <code><, <=, >=, ></code> | links | |
| 8 | <code>==, !=</code> | links | |
| 7 | <code>&</code> | links | |
| 6 | <code>^</code> | links | |
| 5 | <code> </code> | links | Zuweisung |
| 1 | <code>=</code> | — | |

Assignment Project Exam Help
<https://tutorcs.com>
 WeChat: cstutorcs

- Damit ist klar: Klammern `(...)` in `int_max = (1 << 31) - 1` sind notwendig.
- In C0 ist Zuweisung via `=` ein Statement \odot (mit Effekt) und *kein* Ausdruck. Daher illegal: `x = y = e`.

6 | C0: Funktionsdefinition

Seien $\tau, \tau_1, \dots, \tau_n$ Typen und f, p_1, \dots, p_n Identifier ($n \geq 0$). Dann definiert

```

 $\tau$   $f(\tau_1 p_1, \dots, \tau_n p_n)$  {
   $\vdots$ 
  return  $e$ ;
   $\vdots$ 
}

```

Assignment Project Exam Help
<https://tutorcs.com>
 Body (Block) von f

eine n -stellige **Funktion** f des Typs $\tau f(\tau_1, \dots, \tau_n)$.

- Ausdruck e des Typs τ definiert das **Funktionsergebnis**.
Ausführung von **return** e ☺: Rückkehr zum Aufrufer von f .
- Typisch (aber nicht zwingend): ein einziges **return** e als letztes Statement im Body von f .

C0: Funktionsaufruf [Ⓔ]

Funktionsaufruf [Ⓔ] (auch: *function call*) einer n -stelligen Funktion f :

| | |
|----------------------|------------------------------|
| $f(e_1, \dots, e_n)$ | liefert Wert des Typs τ |
|----------------------|------------------------------|

Assignment Project Exam Help

- Auswertung des Funktionsaufrufes:

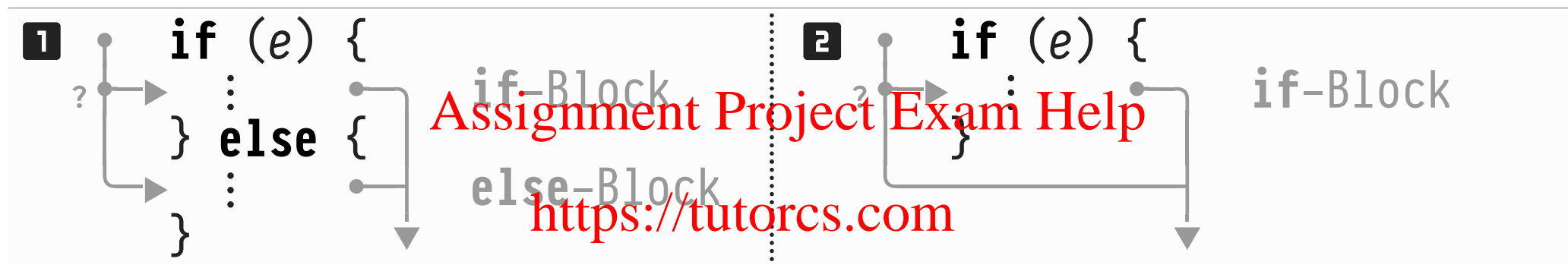
<https://tutorcs.com>

1. **Argumente** e_1, \dots, e_n (in dieser Reihenfolge) auswerten, ergibt Werte v_i des Typs τ_i , $i = 1 \dots n$.
2. Ausführung des Bodys von f . Im Body agieren **Parameter** p_i wie Variablen nach Deklaration $\tau_i \ p_i = v_i$.
3. Wert des Funktionsaufrufes ist das Funktionsergebnis e des Typs τ (**return** e).

WeChat: cstutorcs

7 | C0: Bedingte Anweisung (**if...else**)

Sei e ein Ausdruck des Typs **bool** (Prädikat). **Bedingte Anweisung** ⑤ (auch: *conditional statement*):



WeChat: cstutorcs

- Effekt der bedingten Anweisung:
 - Effekte des **if**-Blocks, falls e zu **true** ausgewertet, ansonsten Effekte des **else**-Blocks.
- Falls der **else**-Block leer **{}** ist, ist Variante 2 eine äquivalente Abkürzung.

Beispiel: Funktion `void printbits(int)`

```
/* Ausgabe der 32 Bits von x (Bit  $b_{31}$  links) */
void printbits(int x) {
    int n = 31;

    while (n >= 0) {
        if (is_bit_set(x, n)) {
            putchar('1');
        }
        else {
            putchar('0');
        }
        n = n - 1;
    }
}
```

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

- Blöcke der Form `{ statement1 }` sind äquivalent zu `statement1`. ⚠ Blocksyntax ist potentielle Fehlerquelle.