Verlässliche Systemsoftware

Übungen zur Vorlesung

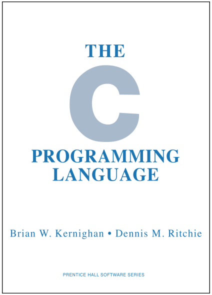
Rolle der Programmiersprache

Phillip Raffeck, Simon Schuster, Peter Ulbrich

Technische Universität Dortmund
Lehrstuhl für Informatik 12 (Arbeitsgruppe Systemsoftware)
`https://sys.cs.tu-dortmund.de`

Sommersemester 2021

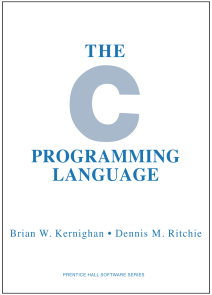
Die Programmiersprache C



Programmieren in C

- ihr könnt alle in C programmieren
- ihr habt alle schon mit C gearbeitet
- diverse Veranstaltungen: SP, SPiC, EZS, …

Die Programmiersprache C



Programmieren in C

- ihr könnt alle in C programmieren
- ihr habt alle schon mit C gearbeitet
- diverse Veranstaltungen: SP, SPiC, EZS, . . .

⇒ Dann sollte man sich ja auch mit C auskennen?

Frage 2 [1]

Zu was wird `1U > -1` ausgewertet?

1. `0`
2. 1
3. nicht definiert

Frage 2 [1]

Zu was wird `1U > -1` ausgewertet?

1. `0`
2. 1
3. nicht definiert

Erklärung

- `unsigned` gewinnt bei impliziter Typumwandlung.

$\leadsto$ `1U > -1U` $\Rightarrow$ `1U > UINT_MAX`

Frage 6 [1]

Zu was wird `UINT_MAX + 1` ausgewertet?

1. `0`
2. `1`
3. `INT_MIN`
4. `UINT_MIN`
5. nicht definiert

Frage 6 [1]

Zu was wird `UINT_MAX + 1` ausgewertet?

1. `0`
2. 1
3. `INT_MIN`
4. `UINT_MIN`
5. nicht definiert

Erklärung

Der C-Standard garantiert, dass `UINT_MAX + 1 == 0`

Frage 7 [1]

Zu was wird `INT_MAX + 1` ausgewertet?

1. `0`
2. `1`
3. `INT_MAX`
4. `UINT_MAX`
5. nicht definiert

Frage 7 [1]

Zu was wird `INT_MAX + 1` ausgewertet?

1. `0`
2. `1`
3. `INT_MAX`
4. `UINT_MAX`
5. nicht definiert

Erklärung

`signed int`-Überlauf ist nicht definiert.

Frage 10 [1]

Angenommen `x` hat Typ `int` und ist positiv. Ist `x << 1` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von `x`?

Frage 10 [1]

Angenommen `x` hat Typ `int` und ist positiv. Ist `x << 1 ...`

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

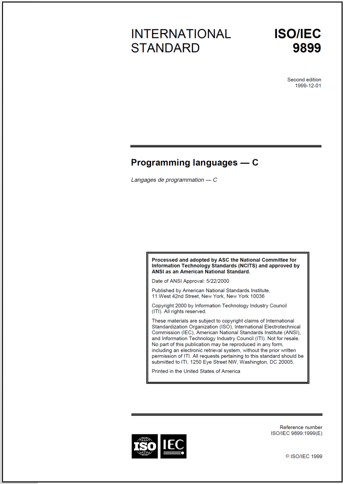
von x?

Erklärung

- Es darf nicht in das Vorzeichenbit hineinverschoben werden

⇒ nicht definiert für große Werte von x

C Standard


INTERNATIONAL STANDARD

ISO/IEC 9899

Second edition
1999-12-01

Programming languages — C

Langages de programmation — C

Processed and adopted by ASC the National Committee for Information Technology Standards (NCITS) and approved by ANSI as an American National Standard.

Date of ANSI Approval: 5/22/2000

Published by American National Standards Institute, 11 West 42nd Street, New York, New York 10036

Copyright 2000 by Information Technology Industry Council (ITI). All rights reserved.

These materials are subject to copyright claims of International Standardization Organization (ISO), International Electrotechnical Commission (IEC), American National Standards Institute (ANSI), and Information Technology Industry Council (ITI). Not for resale. No part of this publication may be reproduced in any form, including an electronic retrieval system, without the prior written permission of ITI. All requests pertaining to this standard should be submitted to ITI, 1250 Eye Street NW, Washington, DC 20005.

Printed in the United States of America

Reference number
ISO/IEC 9899:1999(E)

© ISO/IEC 1999

- Mehrere Iterationen: C89, C99, C11, C18
- Früher ANSI, heute ISO/IEC Standards:
  - ANSI X3.159-1989
  - ISO/IEC 9899:1990
  - ...
- Unabhängiger Standard, von ISO entwickelt
- Beschreibt C Syntax & Semantik

C Standard II

6.5.5 Multiplicative operators

Syntax

multiplicative-expression:
  cast-expression
  multiplicative-expression `*` *cast-expression*
  multiplicative-expression `/` *cast-expression*
  multiplicative-expression `%` *cast-expression*

Constraints

Each of the operands shall have arithmetic type. The operands of the `%` operator shall have integer type.

Semantics

The usual arithmetic conversions are performed on the operands.

The result of the binary `*` operator is the product of the operands.

The result of the `/` operator is the quotient from the division of the first operand by the second; the result of the `%` operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded.[90)] If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` shall equal `a`.

Source: ISO/IEC 9899:TC3, S.94

C Standard II

6.5.5 Multiplicative operators

Syntax

multiplicative-expression:
cast-expression
multiplicative-expression `*` *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

Constraints

Each of the operands shall have arithmetic type. The operands of the `%` operator shall have integer type.

Semantics

The usual arithmetic conversions are performed on the operands.

The result of the binary `*` operator is the product of the operands.

The result of the `/` operator is the quotient from the division of the first operand by the second; the result of the `%` operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded.[90)] If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` shall equal `a`.

Source: ISO/IEC 9899:TC3, S.94

C Standard II

6.5.5 Multiplicative operators

Syntax

multiplicative-expression:
cast-expression
multiplicative-expression ***** *cast-expression*
multiplicative-expression **/** *cast-expression*
multiplicative-expression **%** *cast-expression*

Constraints

Each of the operands shall have arithmetic type. The operands of the **%** operator shall have integer type.

3.4.3
undefined behavior
behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

EXAMPLE An example of undefined behavior is the behavior on integer overflow.

Source: ISO/IEC 9899:TC3, S.4

IEEE 754 (I)

7.3 Division by zero

The divideByZero exception shall be signaled if and only if an exact infinite result is defined for an operation on finite operands. The default result of divideByZero shall be an ∞ correctly signed according to the operation:

— For **division**, when the divisor is zero and the dividend is a finite non-zero number, the sign of the infinity is the exclusive OR of the operands' signs (see 6.3).

— For **logB**(0) when *logBFormat* is a floating-point format, the sign of the infinity is minus ($-\infty$).

Source: IEEE Standard 754 2019, S.53

IEEE 754 (I)

7.3 Division by zero

The divideByZero exception shall be signaled if and only if an exact infinite result is defined for an operation on finite operands. The default result of divideByZero shall be an ∞ correctly signed according to the operation:

— For **division**, when the divisor is zero and the dividend is a finite non-zero number, the sign of the infinity is the exclusive OR of the operands' signs (see 6.3).

— For **logB**(0) when *logBFormat* is a floating-point format, the sign of the infinity is minus ($-\infty$).

Source: IEEE Standard 754 2019, S.53

IEEE 754 (II)

7.2 Invalid operation

The invalid operation exception is signaled if and only if there is no usefully definable result. In these cases the operands are invalid for the operation to be performed.

For operations producing results in floating-point format, the default result of an operation that signals the invalid operation exception shall be a quiet NaN that should provide some diagnostic information (see 6.2). These operations are:

a) any general-computational operation on a signaling NaN (see 6.2), except for some conversions (see 5.12)

b) **multiplication**: **multiplication**$(0, \infty)$ or **multiplication**$(\infty, 0)$

c) **fusedMultiplyAdd**: **fusedMultiplyAdd**$(0, \infty, c)$ or **fusedMultiplyAdd**$(\infty, 0, c)$ unless c is a quiet NaN; if c is a quiet NaN then it is implementation defined whether the invalid operation exception is signaled

d) **addition** or **subtraction** or **fusedMultiplyAdd**: magnitude subtraction of infinities, such as: **addition**$(+\infty, -\infty)$

e) **division**: **division**$(0, 0)$ or **division**(∞, ∞)

f) **remainder**: **remainder**(x, y), when y is zero or x is infinite and neither is a NaN

g) **squareRoot** if the operand is less than zero

h) **quantize** when the result does not fit in the destination format or when one operand is finite and the other is infinite

Source: IEEE Standard 754 2019, S.52

IEEE 754 (II)

7.2 Invalid operation

The invalid operation exception is signaled if and only if there is no usefully definable result. In these cases the operands are invalid for the operation to be performed.

For operations producing results in floating-point format, the default result of an operation that signals the invalid operation exception shall be a quiet NaN that should provide some diagnostic information (see 6.2). These operations are:

a) any general-computational operation on a signaling NaN (see 6.2), except for some conversions (see 5.12)

b) **multiplication**: **multiplication**(0, ∞) or **multiplication**(∞, 0)

c) **fusedMultiplyAdd**: **fusedMultiplyAdd**(0, ∞, *c*) or **fusedMultiplyAdd**(∞, 0, *c*) unless *c* is a quiet NaN; if *c* is a quiet NaN then it is implementation defined whether the invalid operation exception is signaled

d) **addition** or **subtraction** or **fusedMultiplyAdd**: magnitude subtraction of infinities, such as: **addition**(+∞, −∞)

e) **division**: **division**(0, 0) or **division**(∞, ∞)

f) **remainder**: **remainder**(*x*, *y*), when *y* is zero or *x* is infinite and neither is a NaN

g) **squareRoot** if the operand is less than zero

h) **quantize** when the result does not fit in the destination format or when one operand is finite and the other is infinite

Source: IEEE Standard 754 2019, S.52

MISRA-C

Rule 12.7 (required): **Bitwise operators shall not be applied to operands whose underlying type is signed.**

Source: MISRA-C:2004, S.51

MISRA-C

Rule 12.7 (required): **Bitwise operators shall not be applied to operands whose underlying type is signed.**

Source: MISRA-C:2004, S.51

Rule 14.4 (required): **The *goto* statement shall not be used.**

Rule 14.5 (required): **The *continue* statement shall not be used.**

Rule 14.6 (required): **For any iteration statement there shall be at most one *break* statement used for loop termination.**

Source: MISRA-C:2004, S.57

MISRA-C

Rule 12.7 (required):	**Bitwise operators shall not be applied to operands whose underlying type is signed.**

Source: MISRA-C:2004, S.51

Rule 14.4 (required):	**The *goto* statement shall not be used.**
Rule 14.5 (required):	**The *continue* statement shall not be used.**
Rule 14.6 (required):	**For any iteration statement there shall be at most one *break* statement used for loop termination.**

Source: MISRA-C:2004, S.57

Rule 18.4 (required):	**Unions shall not be used.**

Source: MISRA-C:2004, S.67

Defensive Programmieren

- C bietet viele subtile Fehlermöglichkeiten
- Wie verhält man sich als Programmierer richtig?
- *Defensives Programmieren*

⇝ beispielhaft anhand von Ganzzahloperationen

Addition

Was soll da schon schiefgehen...

```c
unsigned int func(unsigned int a, unsigned int b) {
  return a + b;
}
```

Addition

Was soll da schon schiefgehen...

```c
unsigned int func(unsigned int a, unsigned int b) {
  return a + b;
}
```

Vorbedingungstest

```c
#include <limits.h>
unsigned int func(unsigned int a, unsigned int b) {
  if (UINT_MAX - a < b) { raise("wraparound"); }
  return a + b;
}
```

Nachbedingungstest

```c
unsigned int func(unsigned int a, unsigned int b) {
  unsigned int ret = a + b;
  if (ret < a) { raise("wraparound"); }
  return ret;
}
```

Multiplikation

Was soll da schon schiefgehen...

```
unsigned int func(unsigned int a, unsigned int b) {
  return a * b;
}
```

Multiplikation

Was soll da schon schiefgehen...

```c
unsigned int func(unsigned int a, unsigned int b) {
  return a * b;
}
```

Vorbedingungstest

```c
#include <limits.h>
unsigned int func(unsigned int a, unsigned int b) {
  if (a == 0 or b == 0) { return 0; }
  if (UINT_MAX / a < b) { raise("wraparound"); }
  return a * b;
}
```

Addition

Was soll da schon schiefgehen...

```c
signed int func(signed int a, signed int b) {
  return a + b;
}
```

Addition

Was soll da schon schiefgehen...

```c
signed int func(signed int a, signed int b) {
  return a + b;
}
```

Vorbedingungstest

```c
#include <iso646.h>
#include <limits.h>
signed int func(signed int a, signed int b) {
  if ((b > 0 and a > INT_MAX - b)
   or (b < 0 and a < (INT_MIN - b))) { raise("overflow"); }
  return a + b;
}
```

Division

Was soll da schon schiefgehen...

```c
signed long func(signed long a, signed long b) {
  return a / b;
}
```

Division

Was soll da schon schiefgehen...

```
signed long func(signed long a, signed long b) {
  return a / b;
}
```

Vorbedingungstest

```
#include <iso646.h>
#include <limits.h>
signed long func(signed long a, signed long b) {
  if (b == 0) { raise("division by 0"); }
  return a / b;
}
```

Division

- Reicht das schon?

Was soll da schon schiefgehen...

```
signed long func(signed long a, signed long b) {
  if (b == 0) { raise("division by 0"); }
  return a / b;
}
```

Division

- Reicht das schon?

Was soll da schon schiefgehen...

```c
signed long func(signed long a, signed long b) {
  if (b == 0) { raise("division by 0"); }
  return a / b;
}
```

Vorbedingungstest

```c
#include <iso646.h>
#include <limits.h>
signed long func(signed long a, signed long b) {
  if (b == 0) { raise("division by zero"); }
  if (a == LONG_MIN and b == -1) { raise("overflow"); }
  return a / b;
}
```

Weitere Maßnahmen (I)

Konstruktiver Ausschluss

- Einhalten der Grenzbereiche durch Verifikation sichergestellt
- *beweisbare* Sicherheit

Garantiertes Ausnahmeverhalten

- auf Sprachebene
  - Rust: Operationen mit Überprüfung (bspw. `checked_add`)
  - D: Operationen mit Überprüfung: `checkedint`
  - Ada: Constraint_Error bei Überläufen
- durch die Hardware $\rightsquigarrow$ MIPS

Weitere Maßnahmen (II)

Softwareseitige Maßnahmen

- compilergestützt
  - `gcc` built-in functions
    - `__builtin_{add,sub,mul}_overflow`
  - spezielle Warnungen nutzen
    - `-W-sign-compare`, `-W-sign-conversion`
    - `-W-strict-overflow`, `-W-shift-overflow`
- mittels Bibliotheken
  - bspw. *Safe Numerics* von boost.org

Keine Patentlösung

- abhängig von Anwendung und System
- muss beim *Systementwurf* bedacht werden
- zieht sich durch die *gesamte Systementwicklung*
- C macht es einem hier nicht einfach

Fazit

Rolle der Programmiersprache

- definiertes Verhalten in Sprachstandards
- Grenzen dieses Verhaltens
  - ⇝ undefiniertes Verhalten
- C ist zweischneidige Wahl für *verlässliche, eingebettete* Entwicklung
- Konventionen und Werkzeuge *nötig* und *sinnvoll*

Literatur

John Regher.
A quiz about integers in c.