

Programmieren 1

Expressions

Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations and intervals	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
online → 11	23.12.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	13.1.	List and tree operations (filter, map, reduce), objects, object lists	12	19.1. 23:59
13	20.1.	Dynamic data structures (stacks, queues, maps, sets), iterators, documentation tools	(13)	
14	27.1.	C language (remaining C keywords), finite state machines, quicksort	(14)	

Review

- Execution model
 - C closer to hardware, typical CPUs: register machines
- Variables and constants: declaration, definition
 - Variables need to be declared (type) and defined (value)
- Functions: declaration, definition
 - Function header declares signature, function body
- Conditional execution: if statement
 - if ... else, dangling else
- Programming I C library
- Atomic Data (in C)
- Loops: while, for, do-while

Preview

- Binary, octal, hexadecimal, decimal numbers
- Data types and sizes
- Operators, precedence, associativity
- Type conversions
- Expressions, evaluation diagrams
- Recipe for enumerations
- Recipe for intervals

017, 0x1F

char, int, float, ...

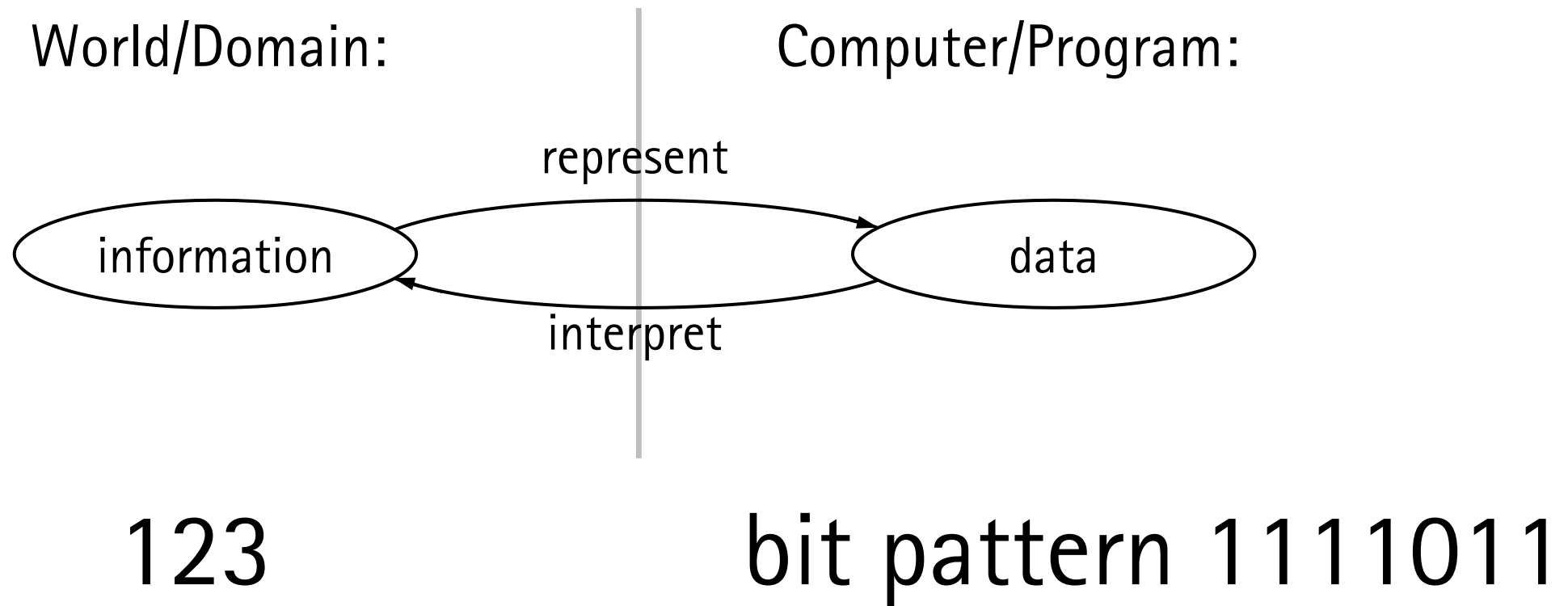
1+2*3

int i = (int)3.14;

5-4/3%2

PLACE-VALUE NOTATION (STELLENWERTSYSTEME)

Representing Numbers



Place-Value Notation: Decimal and Binary

- Place-value notation: Value of a symbol depends on its position

$$h_m h_{m-1} \cdots h_0 . h_{-1} \cdots h_{-n} = \sum_{i=-n}^m h_i \cdot b^i \quad m, n, b \in \mathbb{N} \quad h_i \in \{0, 1, \dots, b-1\}$$

- Decimal system

- Base $b=10$ (decimal)
- Digit at position i is multiplied by 10^i
- **Example:** 123.45_{10} means $1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$

- Binary system

- Base $b=2$ (binary, dual)
- Digit at position i is multiplied by 2^i
- **Example:** 101.101_2 means $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$

Binary System

Developed by Gottfried Wilhelm Leibniz
(article "Explication de l'Arithmétique
Binaire", 1703)

- 1646 born in Leipzig
- 1672/73 completed work on computing machine
 - Add / subtract / multiply / divide
 - Difficult to build at the time
- 1676 became Hofrat and Hofbibliothekar in Hannover
- 1703 published work on binary system
 - Basis for modern computer systems
- 1716 died in Hannover



TABLE 86 MEMOIRES DE L'ACADEMIE ROYALE
DES NOMBRES.

bres entiers au-dessous du double du plus haut degré. Car icy, c'est comme si on disoit, par exemple, que 111 ou 7 est la somme de quatre, de deux & d'un. Et que 1101 ou 13 est la somme de huit, quatre & un. Cette propriété sert aux Essayeurs pour peser toutes sortes de masses avec peu de poids, & pourroit servir dans les monnoyes pour donner plusieurs valeurs avec peu de pieces.

Cette expression des Nombres étant établie, sert à faire tres-facilement toutes sortes d'operations.

Pour l'Addition par exemple.

Pour la Soustraction.

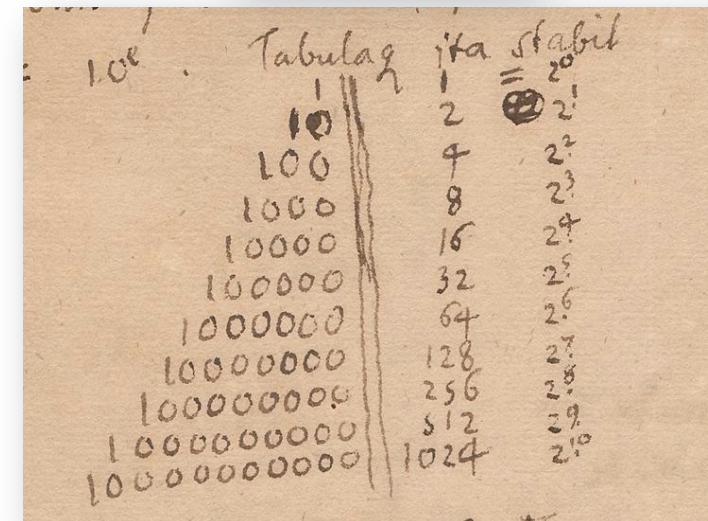
Pour la Multiplication.

Pour la Division.

Et toutes ces operations sont si aisées, qu'on n'a jamais besoin de rien essayer ni deviner, comme il faut faire dans la division ordinaire. On n'a point besoin non-plus de rien apprendre par cœur icy, comme il faut faire dans le calcul ordinaire, où il faut sçavoir, par exemple, que 6 & 7 pris ensemble font 13; & que 5 multiplié par 3 donne 15, suivant la Table d'une fois un est un, qu'on appelle Pythagorique. Mais icy tout cela se trouve & se prouve de source, comme l'on voit dans les exemples précédens sous les signes ⊕ & ⊙.

Powers of 2

2^x	binary	decimal
2^{10}	100000000000	1024
2^9	10000000000	512
2^8	1000000000	256
2^7	100000000	128
2^6	10000000	64
2^5	100000	32
2^4	10000	16
2^3	1000	8
2^2	100	4
2^1	10	2
2^0	1	1
2^{-1}	0.1	0.5
2^{-2}	0.01	0.25
2^{-3}	0.001	0.125



Leibniz: „Alles aus dem Nichts zu entwickeln genügt Eins (Omnibus ex nihilo ducendis sufficit unum).“

Place-Value Notation: Octal and Hexadecimal

- Octal system
 - Base $b=8$ (octal)
 - 8 symbols: 0, 1, 2, 3, 4, 5, 6, 7
 - Digit at position i is multiplied by 8^i
 - **Example:** 76.54_8 means $7 \cdot 8^1 + 6 \cdot 8^0 + 5 \cdot 8^{-1} + 4 \cdot 8^{-2}$
- Hexadecimal system
 - Base $b=16$ (hexadecimal, sedezimal)
 - 16 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
 - Digit at position i is multiplied by 16^i
 - **Example:** $A9.F8_{16}$ means $10 \cdot 16^1 + 9 \cdot 16^0 + 15 \cdot 16^{-1} + 8 \cdot 16^{-2}$

Powers of 16

16^x	hexadecimal	decimal
16^8	100000000	4294967296
16^7	10000000	268435456
16^6	1000000	16777216
16^5	100000	1048576
16^4	10000	65536
16^3	1000	4096
16^2	100	256
16^1	10	16
16^0	1	1
16^{-1}	0.1	0.0625
16^{-2}	0.01	0.00390625

Counting in Binary

decimal	binary	comment
0	0	need 1 st position
1	1	
2	10	need 2 nd position
3	11	
4	100	need 3 rd position
5	101	
6	110	
7	111	
8	1000	need 4 th position
9	1001	
10	1010	
11	1011	
12	1100	
13	1101	
14	1110	
15	1111	

assume we have only 4 bits available,
what happens if we increment 1111?

Positive and Negative Numbers: Two's Complement

signed decimal	binary
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
<hr/>	
0	0000
<hr/>	
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

- First bit is a "sign bit"
 - first bit 0: number is positive or zero
 - first bit 1: number is negative
- To flip the sign (+x to -x)
 - invert bit pattern, add 1
- Examples
 - $0000_2 (=0_{10}) \rightarrow 1111_2 \rightarrow 0000_2 (=0_{10})$
 - $0001_2 (=1_{10}) \rightarrow 1110_2 \rightarrow 1111_2 (= -1_{10})$
 - $1110_2 (= -2_{10}) \rightarrow 0001_2 \rightarrow 0010_2 (=2_{10})$

assume we have only 4 bits available,
what happens if we increment 0111?

Computing with Binary Numbers

- Representing numbers is not enough, we want compute with them
- Basic arithmetic works fine with binary representations
- Addition: Subtraction:

Addition	Example
$0+0= 0$	1011_2
$0+1= 1$	$+0011_2$
$1+0= 1$	$1\ 1$
$1+1=10_2$	1110_2

Subtraction	Example
$0-0= 0$	0100_2
$0-1=-1$	-0110_2
$1-0= 1$	$1\ 1$
$1-1= 0$	1110_2

or...

Subtraction	Example
add two's complement	$0100_2 - 0110_2$
$x-y = x+(-y)$	$= 0100_2 + 1010_2$
	$= 1110_2 = -2$

Arithmetic Overflow Experiment

- What is the output of this program? Why?

```
#include "base.h"
int main(void) {
    signed char c = 0x7f; // 7F16 = 710 * 161 + 1510 * 160 = 011111112 = 12710
    printf(c);
    c = c + 1;
    printf(c);
    c = 127;
    printf(c);
    c = c + c;
    printf(c);
    return 0;
}
```

Output?

127

Output?

-128

Output?

127

Output?

-2

1111 1110

= -2

1111 1111

= -1

0000 0000

= 0

0000 0001

= 1

0000 0010

= 2

...

0111 1111

+ 0111 1111

= 1111 1110

Converting between Binary Octal, Hexadecimal and Decimal

■ Binary to Octal and Hexadecimal

octal	0			5			7			2		
binary	0	0	0	1	0	1	1	1	1	0	1	0
hexadecimal	1			7			A					

$$\begin{aligned}
 & 5 \cdot 8^2 + 7 \cdot 8^1 + 2 \cdot 8^0 \\
 &= 2^8 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1 \\
 &= 1 \cdot 16^2 + 7 \cdot 16^1 + 10 \cdot 16^0 \\
 &= 378
 \end{aligned}$$

■ to Decimal

$$h_m h_{m-1} \cdots h_0 \cdot h_{-1} \cdots h_{-n} = \sum_{i=-n}^m h_i \cdot b^i \quad m, n, b \in \mathbb{N} \quad h_i \in \{0, 1, \dots, b-1\}$$

■ Decimal to Binary

- while (d != 0) { if (d even) print('0') else print('1'); d = d / 2; }
- Example: d=11₁₀: 11, 5, 2, 1
prints: 1 1 0 1 → (reverse) → 1011₂

DATA TYPES AND SIZES

Interpreting Bit Patterns as Values

- Computers only store bit patterns
 - ... but we are not interested in bit patterns (typically)
 - we are interested in data, representing real-world information
- Need to interpret bit patterns
 - as characters
 - as integers
 - as floating-point numbers
- Limited capacity for each variable
 - cannot represent all integers \mathbb{Z}
 - cannot represent all real numbers \mathbb{R}

How many different values can be stored in an 8-bit variable?

How many different values can be stored in an n-bit variable?

Data Types

- Data types
 - Specify what kind of value a variable stores
 - Give meaning to bit patterns
 - C allows access to raw bit pattern, most other languages don't
 - Allows re-interpretation of bit pattern, flexible but error-prone
- Set of possible values and operations on these values
 - Operators are specific to data types
 - Example: Integer division is different from floating-point division
- Example
 - Bit pattern in memory cell: 01110000
 - For data type char, interpreted as ASCII character 'p'
 - For data type integer, interpreted as $1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 = 112$

Built-in Data Types in C

- Basic types
 - `char` single byte, holds a single ASCII character
 - `int` integer number
 - `float` single precision floating-point number
 - `double` double precision floating-point number
 - `bool` Boolean type (values: true, false)
- Qualifiers
 - `short, long` modify the number of bits used
 - `signed, unsigned` if omitted then signed is assumed
- Example combinations
 - `short int` (= short)
 - `unsigned long int` (= unsigned long)
 - `signed int` (= int)

Typical Sizes of Built-in Data Types

- C size rules (number of bits in memory)
 - $\text{char} < \text{short} \leq \text{int} \leq \text{long}$
 - $8 \text{ bits} \leq \text{char}, 16 \text{ bits} \leq \text{short}, 16 \text{ bits} \leq \text{int}, 32 \text{ bits} \leq \text{long}$
- Actual size depends on hardware (see `<limits.h>`, `<float.h>`)
- Typical integer sizes and ranges

■ char	8 bits	$-2^7..2^7-1$	-128..127
■ unsigned char	8 bits	$0..2^8-1$	0..255
■ short	16 bits	$-2^{15}..2^{15}-1$	-32768..32767
■ unsigned short	16 bits	$0..2^{16}-1$	0..65535
■ int	32 bits	$-2^{31}..2^{31}-1$	-2147483648..2147483647
■ unsigned int	32 bits	$0..2^{32}-1$	0..4294967295
■ long	64 bits	$-2^{63}..2^{63}-1$	
■ unsigned long	64 bits	$0..2^{64}-1$	

Typical Sizes of Built-in Data Types

- Floating point sizes and ranges
 - float 32 bits 6 significant decimal digits, $10^{-38} \dots 10^{38}$
 - double 64 bits 15 significant decimal digits, $10^{-307} \dots 10^{308}$
 - long double 128 bits 18 significant decimal digits, $10^{-4931} \dots 10^{4932}$

Data Type Sizes Experiment

- sizeof operator yields the number of bytes of its operand

- Examples (for gcc and macOS 11.6):

`printf("%d", sizeof(char));` 1 byte

`printf("%d", sizeof(short int));` 2 bytes

`printf("%d", sizeof(int));` 4 bytes

`printf("%d", sizeof(long int));` 8 bytes

`printf("%d", sizeof(bool));` 1 byte

`printf("%d", sizeof(float));` 4 bytes

`printf("%d", sizeof(double));` 8 bytes

`printf("%d", sizeof(long double));` 16 bytes

DATA TYPES AND SIZES: FLOATING-POINT NUMBERS

Floating-Point Numbers (Float and Double)

- Float:

- 32 bits

1	8 bits	23 bits
S	E	M

- Double:

- 64 bits

1	11 bits	52 bits
S	E	M

- IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines

- Arithmetic operations
 - Rounding rules
 - Exceptions (division by zero, overflow)
 - Special values ("not a number" NaN, $+\infty$, $-\infty$)

Institute of Electrical
and Electronics
Engineers, "i-triple-e"

Floating-Point Numbers (float, 32 bits)

- Encode floating-point number x as a bit pattern (of 32 bits)

- Float:

1	8 bits	23 bits
S	E	M

encoded in 32 bits:
sign, integer, shift

- Encode $x = s * m * 2^e$ as a bit pattern $[S, E, M]$:
 - $S = 0$ for positive numbers, $S = 1$ for negative numbers
 - $M = 2^{23} * (m - 1)$, with $1 \leq m < 2$
 - $E = e + 127$
 - Example: $-0.5 = -2^{-1} = -1 * 1 * 2^{-1} \rightarrow S = 1, M = 0, E = 126$

- Decode bit pattern $[S, E, M]$:
 - Define $s = (-1)^S$
 - $m = 1 + M / 2^{23}$

$2^{-23} = 1.192 \times 10^{-7}$
 - $e = E - 127$

Floating-Point Numbers (float, 32 bits)

- Encode floating-point number x as a bit pattern (of 32 bits)
- Write as $x = s * m * 2^e$
 - Sign: $s \in \{-1, 1\}$
 - Mantissa: $1 \leq m < 2$ (German: Mantisse)
 - Exponent: $-127 \leq e < 128$ (how to shift ("float") the point)
 - Example: $x = -0.5 = \underset{s}{-1} * \underset{m}{1} * \underset{2^e}{2^{-1}}$
- Encodable values when floating the point from -2 to 2
 - $e = -2 \Rightarrow x = m * 2^{-2} \Rightarrow 0.25 \leq x < 0.50$
 - $e = -1 \Rightarrow x = m * 2^{-1} \Rightarrow 0.50 \leq x < 1.00$
 - $e = 0 \Rightarrow x = m * 2^0 \Rightarrow 1.00 \leq x < 2.00$
 - $e = 1 \Rightarrow x = m * 2^1 \Rightarrow 2.00 \leq x < 4.00$
 - $e = 2 \Rightarrow x = m * 2^2 \Rightarrow 4.00 \leq x < 8.00$

Floating-Point Precision Experiment

- What is the output of this program? Why?

```
#include "base.h"
```

```
int main(void) {
```

```
    float f = 2e10f + 1;
```

```
    printf(f - 2e10f);
```

Output?

0.0

```
    double d = 2e10 + 1;
```

```
    printf(d - 2e10);
```

Output?

1.0

```
    return 0;
```

```
}
```

20000000000.0 encoded as:

S	E	M
0	161	1377017

20000000001.0 encoded as:

S	E	M
0	161	1377017

S	E	M
0	161	137701 <u>8</u>

decodes to 20000002048.0

OPERATORS

Operators in C

	Operators
function call	()
array access	[]
member access	-> .
(de-)referencing	* (indirection) & (address of)
arithmetic	+ - * / %(mod.) ++(incr.) --(decr.) +/-(unary)
relational	< <= > >=
equality	== (equals) != (not equals)
logical	&& (and) (or) !(not)
bitwise	& (and) (or) ^ (exclusive or) ~ (flip bits)
bit shift	<< >>
type cast	(type)
sizeof	sizeof(type)
conditional expression	test ? a : b
assignment operators	= += -= *= /= %= &t= ^= = <<= >>=
comma	,

Operators and Expressions

- Operators
 - Combine operands to a new value
 - Unary operators take one operand
 - Example: -5 (unary integer minus operator)
 - Binary operators take two operands
 - Example: 1 + 2 (binary integer plus operator)
 - Ternary operators take three operands
 - Example: valid ? 10 : 20 (ternary conditional operator)
- Expressions
 - Example: (1 + 2) * 3 + f(7)
 - Consist of operands and operators, including function calls
 - Are evaluated to a new value

Arithmetic, Relational, and Equality Operators

- Binary arithmetic operators: `+`, `-`, `*`, `/`, `%`

- `/` integer division truncates the fractional part
- `%` is the modulo operator (rest of integer division)

3 / 2 is 1 (not 1.5)

- Relational operators

- `>` `>=` `<` `<=`
- yield 1 if true, 0 otherwise

no **boolean** type in traditional C:
integer 0 represents false, all
other integers represent true

- Equality operators

- `==` `!=`
- yield 1 if true, 0 otherwise

no built-in boolean
type in traditional C

Increment and Decrement Operators

- ++ --
- prefix (++n): changes variable before value is used
- postfix (n++): changes variable after value has been used
- Example:

```
int n, x;
```

```
n = 5;
```

```
x = n++; // assigns value of n to x, then increments n
```

```
// assertion: x == 5, n == 6
```

```
n = 5;
```

```
x = ++n; // increments n, then assigns (incremented) value of n to x
```

```
// assertion: x == 6, n == 6
```

Logical Operators

- Logical operators
 - `&&` (and) `||` (or)
 - 0 means false, not-0 means true
 - yield 1 if true, 0 otherwise
- Expressions connected by logical operators
 - Evaluation is left to right and stops as soon as truth value is known
 - Short-circuit evaluation
 - Example: `if (a == 1 || b == 2 || c == 3)...`
evaluation stops after first equality operator if `a == 1`
- Unary negation operator ("not")
 - `!` converts non-zero operand to 0, converts 0 operand to 1
 - Example: `if (!valid)...` means "if not valid..."

a	b	a && b	a b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Bitwise Operators

- **&** bitwise AND
 - example: 0101_2
 $\quad \quad \quad \& 0110_2$
 $\quad \quad \quad \underline{\hspace{1cm}}$
 $\quad \quad \quad = 0100_2$
- **|** bitwise inclusive OR
 - example: 0101_2
 $\quad \quad \quad | 0110_2$
 $\quad \quad \quad \underline{\hspace{1cm}}$
 $\quad \quad \quad = 0111_2$
- **^** bitwise exclusive OR
 - example: $0101_2 \wedge 0110_2 = 0011_2$
- **~** not (flip bits, one's complement)
 - example $\sim 0101_2 = 1010_2$

a	b	a & b	a b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bit Shift Operators

- << left shift
 - $x \ll n$ shifts x left by n bit positions
 - fills with 0s
 - Example: $1010_2 \ll 2 = 1010\mathbf{00}_2$

- >> right shift
 - $x \gg n$ shifts x right by n bit positions
 - fills with 0s for unsigned quantities
 - Example: $1110_2 \gg 2 = 11_2$
 - fills with sign bit or 0s depending on machine behavior for signed quantities

Assignment Operators

- Assignment operators and expressions
 - $\text{expr}_1 \text{ op} = \text{expr}_2$ is equivalent to $\text{expr}_1 = (\text{expr}_1) \text{ op } (\text{expr}_2)$
 - where **op** is one of `+` `-` `*` `/` `%` `<<` `>>` `&` `^` `|`

- Examples
 - `x *= 3;` means `x = x * 3;`
 - `x -= 5;` means `x = x - 5;`
 - `x *= 3 + y;` means `x = x * (3 + y);`

- Assignment operators have a type and a value
 - type is that of expr_1
 - value is that after the assignment

Conditional Operator

- Conditional operator
 - $\text{expr}_1 ? \text{expr}_2 : \text{expr}_3$
 - if expr_1 is true, evaluate expr_2 otherwise evaluate expr_3
 - Example: $z = (a > b) ? a : b;$ // $z = \max(a,b)$
- Sometimes useful replacement for
if (expr) ... else ...

Evaluation Order

- C does **not** specify the order in which operands of an operator are evaluated
 - Example: $x = f() + g()$
 - f may be called before g ...
 - ... or g may be called before f

Type Conversions

- Operands of different types are converted to a common type
- Rules for binary operators with **signed** operands
 - if either is long double, convert other to long double
 - else if either is double, convert other to double
 - else if either is float, convert other to float
 - else convert char and short to int
 - then, if either is long, convert other to long
- Examples
 - `double d; float f; char c; int i; long l;`
 - `d * f` // convert f to double, double-multiplication
 - `i * f` // convert i to float, float-multiplication
 - `c * i` // convert c to int, int-multiplication

Type Conversion Examples

Type conversions when evaluating expressions

double d; float f; char c; int i; long l;

```

c * i + d
char * int + double
int * int + double
int + double
double + double
double

```

```

(c - '0') * 3.14
(char - char) * double
(int - int) * double
int * double
double * double
double

```

Type Conversions in Assignments and for Arguments

- Conversion rules also apply across assignments
 - `int i; char c;`
 - `i = c;` // sign extension depends on whether char on machine is signed
 - `c = i;` // high-order bits are dropped
- ... and when passing arguments to functions
 - `double sqrt(double x);` // method declaration
 - `int i = 5;`
 - `int j = sqrt(i);` // convert i to double

Type Cast Operator

- Explicit type conversions

- (type name) expression

- Example:

```
double d = 123.456;
```

```
int i = (int)(d+0.5); // rounding for positive numbers
```

- Example:

```
int i = 5;
```

```
int j = 123;
```

```
double d;
```

```
d = i / j; // integer division, d == 0
```

```
d = (double) i / j; // force floating-point division, d == 0.04065...
```

Type Cast Operator

- Casting to a smaller type means that the set of representable values becomes smaller
 - Values might get truncated

- Example:

```
short s = 0x1234;    // 16 bits
char c = 0;          // 8 bits
println(s);          // prints 4660 = 0x1234
c = s;
println(c);          // prints 52 = 0x34
```

EXPRESSIONS

Operator Precedence (Kernighan, Ritchie, Table 2-1)

	Operators
highest	() [] -> .
	! ~ ++ -- ++(unary) *(indirection) &(address of) (type) sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	&&
	?:
	= += -= *= /= %= &= ^= = <<= >>=
lowest	,

expressions are
evaluated in order of
operator precedence
and associativity

use parentheses (...)
if in doubt

Precedence and Associativity

- **Precedence** defines order of evaluation of different operators
 - $1 + 2 * 3$ means $1 + (2 * 3)$, $*$ has higher precedence than $+$
 - value of expression is 7
- **Associativity** defines order of evaluation of operators of the same precedence
 - $1 - 2 + 3$ means $(1 - 2) + 3$, $+$ and $-$ are left-to-right associative
 - value of expression is 2
- Evaluation diagrams (*Auswertungsdiagramme*) show order of evaluation

$$\begin{array}{r} 1 + 2 * 3 \\ 1 + 6 \\ \hline 7 \end{array}$$

$$\begin{array}{r} 1 - 2 + 3 \\ -1 + 3 \\ \hline 2 \end{array}$$

sometimes
multiple
valid orders

Integer Division

- Integer division first, then integer multiplication:

$$\begin{array}{r} 5 / 3 * 2 \\ \hline 1 * 2 \\ \hline 2 \end{array}$$

- Integer multiplication first, then integer division:

$$\begin{array}{r} 5 * 2 / 3 \\ \hline 10 / 3 \\ \hline 3 \end{array}$$

- Typically integer division should be last in an expression

Assignments (are expressions as well)

- In C, an assignment is an expression and has a value
- Left-hand side of an assignment have a memory location ("Lvalue")
- Assignment is right-to-left associative

$$\begin{array}{r}
 a = b = c = 0 \\
 \hline
 a = b = 0 \\
 \hline
 a = 0 \\
 \hline
 0
 \end{array}$$

- Using assignments in conditions:
 - `if ((i = i_input()) != 0) ...`
- Can easily get unreadable:
 - `a = 3 * (b = (c = 5) + 1);`
 - better: `c = 5; b = c + 1; a = 3 * b;`

Assignments (are expressions as well)

- With parentheses:
if ((i = i_input()) != 0) ...

i = 5

!= 0

5

!= 0

1

5 → i

	Operators
highest	() [] -> .
	! ~ ++ -- +-(unary) *(indirection)
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	&&
	?:
	= += -= *= /= %= &= ^= =
lowest	,

Evaluating Complicated Expressions

- In C, this is a valid expression
 - `4 < 8 && 21 + 3 != 10`
 - What is the result?
- C treats boolean values as integers
 - logical operators yield 0 or 1
- Recommendations
 - For readability add parentheses to make the order of evaluation explicit (...)
 - Simplify expressions
 - Use multiple simpler expressions

$$\begin{array}{rcl}
 4 < 8 & \&\& & 21 + 3 & != & 10 \\
 \hline
 4 < 8 & \&\& & 24 & != & 10 \\
 \hline
 1 & \&\& & 24 & != & 10 \\
 \hline
 1 & \&\& & & & 1 \\
 \hline
 & & & & & & 1
 \end{array}$$

	Operators
highest	() [] -> .
	! ~ ++ -- +(unary) *(value of) &(address of) (type) sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	&&
	?:
	= += -= *= /= %= &= ^= = <<= >>=
lowest	,

RECIPE FOR ENUMERATIONS

Enumerations

- An enumeration type can represent one of a fixed number of distinct values
- Each enumerated value names a category and does not carry any additional data
- Example
 - Suit of cards (diamonds ♦, clubs ♣, hearts ♥, and spades ♠)
 - Continents (Europe, Asia, Africa, etc.)
- In C, enumerations are written as `enum`

enum

- Enumeration constants
 - `enum Continent { EUROPE, ASIA }; // EUROPE == 0, ASIA == 1`
 - `enum Months { JAN = 1, APR = 4, JUL = 7, OCT = 10 };`
- Enums behave like integers
 - Internally, C represents enumeration names as integer constants
 - First name gets value 0, second name gets value 1, etc.
 - Explicit assignment of constant values possible (see above)

1. Problem Statement

- Write down the problem statement as a comment.
 - What is the relevant information?
 - What should the function do with the data?

- Example

/*

Design a function that returns the next color of a traffic light
given the current color of the traffic light.

*/

2. Data Definition

- How should domain information be represented as data in the program? How to interpret the data as real-world information?
- Data definition

```
enum TrafficLight {  
    RED,  
    RED_YELLOW,  
    GREEN,  
    YELLOW  
};
```

3. Function Signature

- Function signature as a comment
 - Parameter types left of the arrow (comma separated)
 - Result type right of the arrow
- Example

```
// enum TrafficLight -> enum TrafficLight
```

4. Function Name

- Preliminary function name
 - Short, non-abbreviated, descriptive name that describes what the function does
- Example
`traffic_light_next`

5. Function Header

- Add function signature to name
- Preliminary parameter names
 - Short, non-abbreviated, descriptive name that describes what the parameter means
- Example
`enum TrafficLight traffic_light_next(enum TrafficLight tl);`

6. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub compiles

- Example

```
enum TrafficLight traffic_light_next(enum TrafficLight tl) {  
    return RED;  
}
```

7. Purpose Statement

- Briefly describes what the function does. Ideally as a single sentence.
- Example
 - // Produces the next color of a traffic light
 - // given the current color of the traffic light.

8. Examples and Expected Results

- Examples

- If the traffic light is RED, expect RED_YELLOW as the next state.
- If the traffic light is RED_YELLOW, expect GREEN as the next state.
- If the traffic light is GREEN, expect YELLOW as the next state.
- If the traffic light is YELLOW, expect RED as the next state.

- Test function

```
void traffic_light_next_test() {  
    test_equal_i(traffic_light_next(RED), RED_YELLOW);  
    test_equal_i(traffic_light_next(RED_YELLOW), GREEN);  
    test_equal_i(traffic_light_next(GREEN), YELLOW);  
    test_equal_i(traffic_light_next(YELLOW), RED);  
}
```

8. Examples and Expected Results

- Program compiles with stub, but produces failed test cases
traffic_light.c, line 18: Actual value 0 differs from expected value 1.
traffic_light.c, line 19: Actual value 0 differs from expected value 2.
traffic_light.c, line 20: check passed
2 of 3 tests failed.
- Note: Output is integer values, not enumeration names

```
enum TrafficLight {
    RED,           ← 0
    RED_YELLOW,    ← 1
    GREEN,         ← 2
    YELLOW         ← 3
};
```


9. Function Body

- Implementation of the function

// Produces the next color of a traffic light

// given the current color of the traffic light.

```
enum TrafficLight traffic_light_next(enum TrafficLight tl) {
    if (tl == RED) {
        return RED_YELLOW;
    } else if (tl == RED_YELLOW) {
        return GREEN;
    } else if (tl == GREEN) {
        return YELLOW;
    } else if (tl == YELLOW) {
        return RED;
    }
    return RED;
}
```

10. Testing

- Main function call test function

```
int main(void) {  
    traffic_light_next_test();  
    return 0;  
}
```

- Test results

```
traffic_light.c, line 18: check passed  
traffic_light.c, line 19: check passed  
traffic_light.c, line 20: check passed  
traffic_light.c, line 21: check passed  
All 4 tests passed!
```

11. Review and Revise

- Review the products of the steps
 - Improve function name
 - Improve parameter names
 - Improve purpose statement
 - Improve and extend tests
- Improve / generalize the function
 - switch-statement

Switch-Statement to Handle Multiple Cases

- Handle the cases of an enumeration
 - Compiler can check that all cases of enumeration are handled
- Example

```
enum TrafficLight traffic_light_next(enum TrafficLight tl) {  
    switch (tl) {  
        case RED: return RED_YELLOW;  
        case RED_YELLOW: return GREEN;  
        case GREEN: return YELLOW;  
        case YELLOW: return RED;  
    }  
    return RED;  
}
```

Selection in C: Switch-Statement

- Multi-way decision
- Tests for equality of an expression to integer constants

```
switch (expression) {  
    case const-expression:  
        statements; break;  
    case const-expression:  
        statements; break;  
    ...  
    default: statements  
}
```

case handles explicit cases

default handles all other cases
(similar to else)

- **break** statement exits from switch statement
- "Falls through" without break statement!

The Switch-Statement is Dangerous

- Use with constant integer cases only!
- Use break or return to avoid falling-through cases!



Switch-Statement Falls Through! Break!

Enters all cases after first matching case, unless break or return

```
enum TrafficLight traffic_light_next(enum TrafficLight tl) {
    switch (tl) {
        case RED:
            ...
            break; ← leave the switch-statement!
        case RED_YELLOW:
            ...
            break; ← leave the switch-statement!
        case GREEN:
            ...
            break; ← leave the switch-statement!
        case YELLOW:
            ...
            break; ← leave the switch-statement!
    } ...
}
```

Type Definitions and Enumerations

- Enumeration

```
enum TrafficLight {
    RED, RED_YELLOW, GREEN, YELLOW
};
```

- Type definition

```
typedef enum TrafficLight TrafficLight; ← typedef <construct> NewName
```

- Can now declare function as

```
TrafficLight traffic_light_next(TrafficLight tl)
```

- instead of

```
enum TrafficLight traffic_light_next(enum TrafficLight tl)
```


RECIPE FOR INTERVALS

Intervals

- Intervals represent one or more ranges of numbers
- Pay attention to boundary cases
- Data definitions
 - Enumerations to name each interval
 - Constants to define boundaries

2. Data Definition

- How should domain information be represented as data in the program? How to interpret the data as real-world information?
- Data definition

```
enum TaxStage { // name each interval
    NO_TAX,
    LOW_TAX,
    HIGH_TAX
};
```

```
typedef int Euro; // int represents Euro
```

may omit typedef
and use int directly

```
// interval boundaries
```

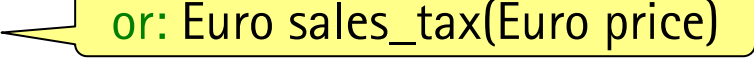
```
const Euro LOW_TAX_BOUNDARY = 1000; // interpret.: price in Euro
```

```
const Euro HIGH_TAX_BOUNDARY = 10000; // interpret.: price in Euro
```

9. Function Body

- Implementation of the function

```
// Return the amount of tax for the given price.
```

```
int sales_tax(int price) {  or: Euro sales_tax(Euro price)
    if (0 <= price && price < 1000) { // NO_TAX: [0, 1000)
        return 0;
    } else if (1000 <= price && price < 10000) { // LOW_TAX: [1000, 10000)
        return round_to_int(0.05 * price);
    } else if (price >= 10000) { // HIGH_TAX: [10000, ∞)
        return round_to_int(0.10 * price);
    }
    printf("sales_tax, error: negative price");
    exit(1); // stop the program
}
```

11. Review and Revise: Constants

```

const double LOW_TAX_RATE = 0.05;
const double HIGH_TAX_RATE = 0.10;
// Return the amount of tax for the given price.
int sales_tax(int price) {
    if (price < 0) {
        printf("sales_tax, error: negative price");
        exit(1);
    } else if (price < LOW_TAX_BOUNDARY) { // NO_TAX: [0, 1000)
        return 0;
    } else if (price < HIGH_TAX_BOUNDARY) { // LOW_TAX: [1000, 10000)
        return round_to_int(LOW_TAX_RATE * price);
    } else { // HIGH_TAX: [10000, ∞)
        return round_to_int(HIGH_TAX_RATE * price);
    }
}

```

Summary

- Binary, octal, hexadecimal, and decimal numbers
- Data types and sizes
- Operators, precedence, and associativity
- Type conversions
- Expressions, evaluation diagrams
- Recipe for enumerations
- Recipe for intervals

C Keywords (ANSI C, ISO C89)

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while