Systems Programming

2 Basics of Data Types, Pointers, and Operators

Secure Systems Group, LIT Secure and Correct Systems Lab
Johannes Kepler University Linz
stefan.rass@jku.at

Contents

Slides adapted from the lecture by Michael Sonntag

Fig. 2.1: hello-world.c

```c
#include<stdio.h>
int main() {
        /* Stream comment;
        ...multi-line (C90) */
        printf("Hello World\n\tNot again :-)\n");
        return 0;  // comment until end of line (C99)
}
```

Which data types can you find in this example?

- Number: 0
  - Integer
  - Literal

Hello World revisited 2

- String: `"Hello World. \n \t Not again! \n"`
  - Sequence of characters
  - Literal
- (Type of the) Return value: **`int`**

Data Types in C 1

- Five important data types in C90

`void`	associated with no data type
`char`	character
`int`	integer
`float`	floating-point number
`double`	double precision floating-point number

- Attention: there is no native `String` datatype! Strings are treated as arrays of type **`char`**.
- Also, note that C (also C++) can distinguish, unlike many other languages, signed from unsigned values (by default, values are always signed). You can declare a variable `x` as **`unsigned int`** `x;` making sure that `x` can never take on negative values. Why and when is that useful?
  - General advice: never declare a variable as **`unsigned`**, unless you are very sure that it will only need positive values. The typical example are addresses in memory.

Data Types in C 2

– Also, turn on compiler warnings (e.g., `-Wconversion`, `-Wabsolute-value`, ...) to get an idication when you unintentionally mix up signed and unsigned values.
– Unsigned integers may have an advantage in cases of overflows. Exercise: try out the following code with variable `i` as **`int`** (as shown), but also as **`unsigned int`** (not shown).

Fig. 2.2: `int-overflow.c`

```c
#include <limits.h>        /* to get INT_MAX */
int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

- Added in C99

long long	min. 64 Bits
uint8_t/int32_t	and similar types with defined sizes
bool	Boolean type
	also adds **true**, **false** as Boolean literals
	requires **#include** <stdbool.h>

- Added in C11

char16_t, char32_t	Unicode UTF-16/-32 text

Type modifiers:

- Several of the basic types can be modified using **signed**, **unsigned**, **short**, and **long**
- When one of these type modifiers is used by itself, a data type of **int** is assumed.
- A complete list of possible data types follows:

```
char               unsigned char          signed char
int                unsigned int           signed int
  short int        unsigned short int     signed short int
  long int         unsigned long int      signed long int
float
double
  long double
```

Numeric Ranges 1

Type	Bytes	Range
`char`, `signed char`	1	-128 . . . 127
`unsigned char`	1	0 . . . 255
`short`, `short int`, `signed short`, `signed short int`	2 (≥ 2)	-32768 . . . 32767
`unsigned short`, `unsigned short int`	2 (≥ 2)	0 . . . 65535
`int`, `signed`, `signed int`	4 (≥ 2)	-2147483648 . . . 2147483647
`unsigned`, `unsigned int`	4 (≥ 2)	0 . . . 4294967295
`long`, `long int`, `signed long`, `signed long int`	8 (≥ 4)	-9223372036854775808 . . . 9223372036854775807
`unsigned long`, `unsigned long int`	8 (≥ 4)	0 . . . 18446744073709551615

Type	Bytes	Range
`long long`, `long long int`, `signed long long`, `signed long long int`	8 (≥ 8)	-9223372036854775808 ...9223372036854775807
`unsigned long long`, `unsigned long long int`	8 (≥ 8)	0 ...18446744073709551615
`float`	4	$1 \times 10^{-37} \ldots 1 \times 10^{37}$
`double`	8	$1 \times 10^{-308} \ldots 1 \times 10^{308}$
`long double`	16	$1 \times 10^{-4932} \ldots 1 \times 10^{4932}$
`void`	1	-
`void*`	8	`0x00...000...0xff...ff`

Remarks:

- The values are for 64 Bit Linux.
- Floating point number types of course allow for positive and negative values; we list the numeric ranges here only in absolute terms (the negative range is the same as the positive).
- In many cases, it is useful to have the maximum and minimum values explicitly, e.g., to avoid an over- or underflow. These limits become available upon `#include<limits.h>`

- In C, a series of instructions enclosed in curly brackets { ... } defines a scope.
- A scope is a domain in which local variables can be declared. Their region of visibility is then bounded to the scope, meaning that the variable is not visible after the closing }
- However, all variables declared outside a scope are accessible inside the scope
- If a variable is, inside a scope, re-declared with the same name, it shadows the outer variable, meaning that it overrides the variable from outside with the same name. The "outside variable" remains untouched, i.e., retains its value.

Example 2.1:

```c
int main() {  // <- this opens the scope of main()
  int a = 1, b = 2; // <- local variables inside the
                    // scope; invisble outside,
                    // i.e., not visible in any other function
  int i;
  for(i = 1; i <= 10; i++) {
    int a = 7; // this variable shadows "a" as defined above
    // the variable "b" remains visible in this inner scope
    int c; // this variable is only visible inside the for-body
  }
  // c is not visible here!
  c = 1; // this will cause a compiler error
}
```

Declarations vs. Definitions

- In C, declarations and definitions are not the same thing
- Declaration introduces a name (= identifier) to the compiler → this is the main purpose of header-files; cf. slide 1-28
- Creates an entry in the compiler's list of "things to assign an address"
- Definitions allocate storage for the name. This meaning applies for both variables and functions (see slide 4-41 for further remarks)
  - For a variable → space is reserved in memory to hold the data
  - For a function → the compiler generates code, which ends up occupying storage in memory
- You can declare a variable or a function in many different places, but there must be only one definition per item in C (this is sometimes called the ODR: one-definition rule) This is checked when the linker is uniting all the object modules
- A definition can also be a declaration. If the compiler has not seen the name `x` before and you define `int x`, the compiler sees it as a declaration and allocates storage for it all at once.

Additional Data Type Facts

- Integer lengths (in bytes) satisfy: **`char`** $\leq$ **`short`** $\leq$ **`int`** $\leq$ **`long`**
- The sizes given on slides 2-9ff are indicative only; the actual values are platform- and compiler-specific.
- But there are definitions in the libraries which are precise (e.g., `int64_t` designates an exact-width 64-bit integer; see C99)
- To determine the size of any variable type in bytes, you can use the **`sizeof`** operator:

```
printf("Size of int is %d\n", sizeof(int));
```

- There was no Boolean primitive until C99
  - 0 = **`false`**
  - Everything else = **`true`**
- There is no "byte" primitive in C, so you have to use **`char`**, or more usually **`unsigned char`** instead
- Divide by zero run-time errors and illegal numbers may be returned as: `+/-INF` (infinity) or `IND` (indetermined) or `NaN` (not a number) depending on the compiler.

Literals 1

Strings: Written in double quotes, e.g., `"abc"`.
Variables to store strings are arrays of char-type, whose last element must be zero to indicate the end of the string. Consequently, storing the 3-letter string "ABC" takes up 4 bytes: 65 66 67 00 ← zero-terminated

Characters: Enclosed in single quotes, e.g., `'a'`; An assignment `x = 'a';` would put the ASCII code of the character "a" into `x`. No zero-termination required, since it is always only 1 byte.

Integers: specified differently, depending on the radix

- Decimal notation (the default): `160`
- Hexadecimal notation: `0x100` (starts with "0x")
- Octal notation: `0100` (starts with "0")
- Modifiers for **`long`** and **`unsigned long`** are suffixes L and UL: `160L` and `160UL`

Literals 2

Real numbers: use a dot (not comma) to separate integer from fractional part

- Like integers, followed by the decimal part: `160.1`
- If no decimal part is present, the dot must still be added: `160.` or `160.0`

Attention: there are no further literals (e.g., like convenience notations to specify arrays or similar)

Naming Conventions

- Identifier = name of
  - Variable
  - Function
  - Parameter
  - Template tag of structures/unions/enums
  - Member of structures/unions
  - Type definition
- Can consist of
  - Upper- and lower-case ASCII letters
  - Decimal digits
  - Underscore character
- Has to start with letter
- Maximum of 31 characters
- Must not be one of the reserved keywords

Reserved Keywords – ANSI C (C89), ISO C (C90)

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

C99

+ `_Bool` / `bool`
+ `_Complex`
+ `_Imaginary`
+ `inline`
+ `restrict`

What exactly are variables? 1

- A variable in a program is a name, by which a certain memory cell is accessible.
- The compiler and linker typically take care of associating addresses with variable names, internally hosting lookup tables to map textual names to blocks of memory.
- When we declare a variable we inform the compiler of:
  - the name of the variable, and
  - the type of the variable
- For example, we declare an integer variable with the name `k` by writing: `int k;`
  - The compiler sets aside 4 bytes of memory to hold the value (IA-32)
  - It also sets up a symbol table, and adds the symbol `k` and the relative address in memory where those 4 bytes were set aside
  - Thus, later if we write: `k = 2133;` we expect that, at run time when this statement is executed, the value 2133 will be placed in that memory location reserved for the storage of the value of `k`
  - In C we refer to a variable such as the integer `k` as an "object"

What exactly are variables? 2

- In our previous example, in a sense there are two "values" associated with the object `k`
- One is the value of the integer stored there (2133 in the above example)
- The other the "value" of the memory location, i.e., the address of `k`
- Some texts refer to these two values respectively as
  - rvalue (right value, pronounced "are value") = `value` and
  - lvalue (left value, pronounced "el value") = `address`
- In some languages, the lvalue is the value permitted on the left side of the assignment operator "=", i.e. the address where the result of evaluation of the right side ends up.
- The rvalue is the expression on the right side of the assignment statement, the 2133 above. rvalues cannot be used on the left side of the assignment statement; thus: `2 = k;` is illegal. some languages make this more explicit via the syntax, such as R, which uses the <- operator for assignments, or Pascal, in which assignments are made with :=

What exactly are variables? 3

- Actually, the above definition of lvalue is somewhat modified for C into understanding it as a "locator value", i.e., any object that occupies some memory address or processor register. An rvalue is understood as any intermediate result that is not necessarily stored somewhere (equivalently defined by exclusion: anything that is not an lvalue is an rvalue) We will revisit these in a later Chapter (slide 5-8)

- In C, every Boolean condition evaluates to the integer 0 to represent **false** and 1 to represent **true**. Any nonzero value appearing in a condition is treated as **true**, e.g., if we have `a = 10`, then `if (a){ ... }` will enter the "then" clause of the condition
- For this reason, logical conditions can appear on the right hand side of assignments, i.e., we can write
  ```
  int a = (i > j);
  ```
  as a valid instruction to get `a = 0` if $i \leq j$ and `a = 1` if $i > j$.
- An assignment itself evaluates, as an expression, to the value being assigned. For example, if we write `var = 10`, then the whole expression evaluates to the (assigned) value 10 in any outer expression. For this reason, we can write
  - multiple assignments like `a = (b = c)`, which first assigns the value of `c` to `b`, and then the value of `b` to `a`.
  - inner assignments in Boolean conditions, like `while((c = getchar())!= EOF){ ... }` to first read a value into the variable `c`, and then check the loop condition on this value.

Type Conversions 1

- Implicit (automatic) type conversions
  - Happens automatically during the course of evaluating an expression
  - Preserve precision (i.e., are always "widening" conversions)
  - If the result value does not "fit" into the result type, it is silently truncated (i.e., no error message!) $\rightarrow$ inputs are adjusted, not the output!
  - Tiebreaker for types that are otherwise the same width **`signed`** $\rightarrow$ **`unsigned`**
  - Assignment conversions
    - Happen as part of an assignment
    - They do not necessarily preserve precision and no error is signaled when truncation occurs

Example 2.2:

`int num = 312;`

`char ch = num;` $\Rightarrow$ what is the value of `ch`?

◇

Type Conversions 2

- Explicit type conversions
  - Like in Java: `(type) expression`
  - You can typecast from any type to any type

    ```
    char c = (char)some_int;
    ```

    ⇒ So be careful!

Pointer Variables

- A pointer variable is a variable to hold an lvalue (an address)
  - The size required to hold such a value depends on the system
  - The actual size required is not too important, as long as we have a way of informing the compiler that what we want to store is an address
- Pointer variables in C:
  - We define a pointer variable by preceding its name with an asterisk
  - In C we also give our pointer a type which refers to the type of data stored at the address we will be storing in our pointer.
- Example:

```c
int *ptr;
```

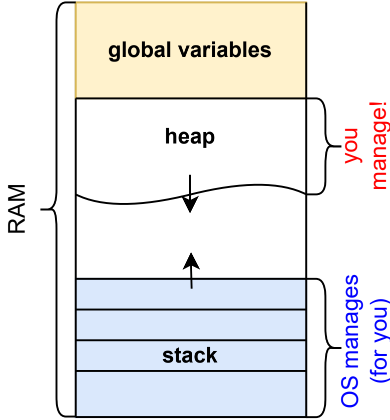
- Such a pointer is said to "point to" an integer

Pointer Types

- Why do we need to identify the type of variable that a pointer refers to?
- One reason for this is, that the compiler knows how many bytes to copy into or out of that memory location in the future
- For example, if `ptr` points to an **`int`** value, `*ptr = 2;` would result in 4 bytes being copied into the memory location contained in `ptr`
- This assumes that an **`int`** takes up 4 bytes (depends on the target hardware platform)
- But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code

The entire RAM is divided into three basic sections:

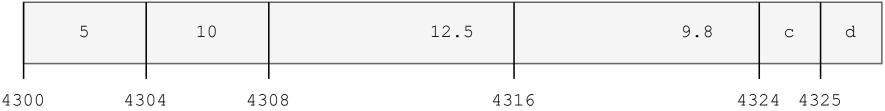
- Global variables
- Heap: space to dynamically allocate and manage memory → to be done manually by the programmer, or (for many modern programming languages) automated by the compiler or libraries
  C lets you (!) take responsibility for the memory management
- Stack: last-in-first-out (LIFO) organized memory, managed by the operating system (for you) to handle function calls. Stores local variables for functions (only), and auxiliary information for the program control flow.



Memory layout and addresses

Local variables of a function are kept on the stack, while global variables and dynamically allocated memory blocks reside in the heap. This distinction is important to keep in mind!

```
int    x = 5,
       y = 10;
double g = 12.5,
       h = 9.8;
char   c ='c',
       d ='d';
```



A pointer...

...is a variable containing the address of another variable

Two unary prefix-operators:

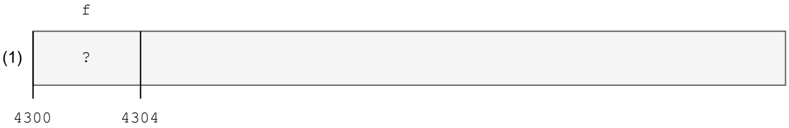
- & to get a variable's address, e.g. `&a` (read-only) returns the address of the variable `a`
- $*$ to access an address stored in a variable, e.g., a read or write to `*a` will get or set the value stored at the address in `a`.

The $*$ modifier also appears in suffix notation as type modifiers: **`int`**$*$, **`float`**$*$, etc., declare variables as pointers to other variables of the given type.

A pointer. . .

. . . is a variable containing the address of another variable

(1) `float f;` data variable

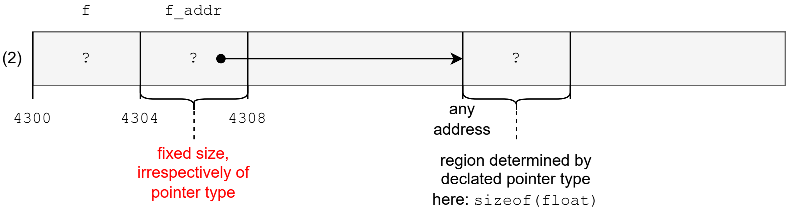


A pointer. . .

. . . is a variable containing the address of another variable

(1) **float** f; data variable

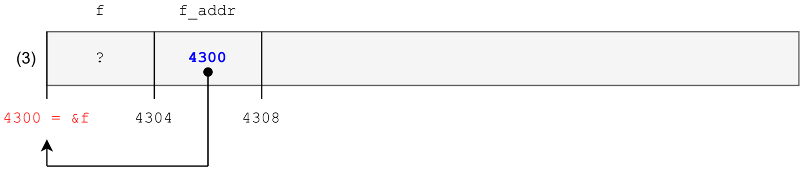
(2) **float** $*$f_addr; pointer variable



A pointer…

…is a variable containing the address of another variable

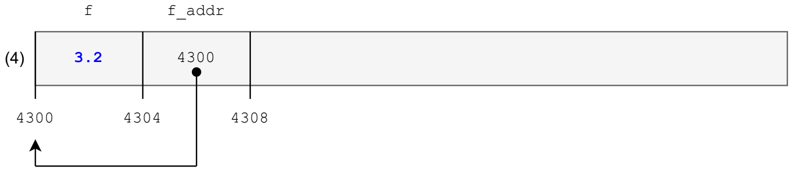
(1) `float f;`	data variable
(2) `float *f_addr;`	pointer variable
(3) `f_addr = &f;`	& is the address-of operator



A pointer...

...is a variable containing the address of another variable

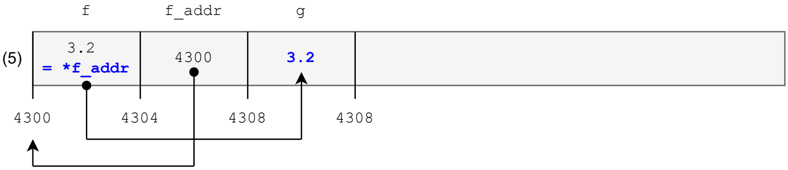
(1) `float f;`	data variable
(2) `float *f_addr;`	pointer variable
(3) `f_addr = &f;`	& is the address-of operator
(4) `*f_addr = 3.2;`	assign a value to the memory cell whose address is in `f_addr`.
	* is the de-referencing or indirection operator



A pointer...

...is a variable containing the address of another variable

(1) `float f;`	data variable
(2) `float *f_addr;`	pointer variable
(3) `f_addr = &f;`	& is the address-of operator
(4) `*f_addr = 3.2;`	assign a value to the memory cell whose address is in `f_addr`.
	∗ is the de-referencing or indirection operator
(5) `float g = *f_addr;`	g is now 3.2

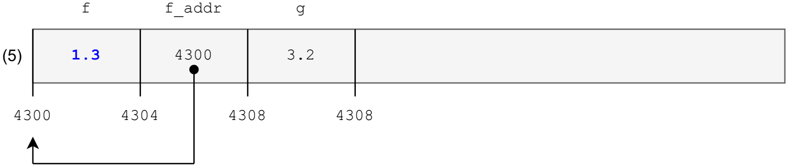


Pointers made (almost) easy 7

A pointer…

…is a variable containing the address of another variable

(1) `float f;`	data variable
(2) `float *f_addr;`	pointer variable
(3) `f_addr = &f;`	& is the address-of operator
(4) `*f_addr = 3.2;`	assign a value to the memory cell whose address is in `f_addr`.
	* is the de-referencing or indirection operator
(5) `float g = *f_addr;`	g is now 3.2
(6) `f = 1.3;`	g is still 3.2!



The "null" pointer and NULL

- If the definition of a pointer variable is made outside of any function (= global/static), ANSI-compliant compilers will initialize it to a value guaranteed to not point to any C object or function. A pointer initialized in this manner is called a "null" pointer.
- Be careful! A null pointer may or may not evaluate to zero
  - This depends on the specific system on which the code is developed.
  - Intuitively, we would expect this to be true; but technically (= ANSI C) the value `0`, upon conversion to a pointer, becomes a "null pointer constant". It points to no object, but it need not be all 0 bits...
  - Therefore, and for lots of other good reasons, C++ (as of version 11) introduced **`nullptr`**, a new constant of a new type (`nullptr_t`)
  - For source compatibility, a macro is used to represent a null pointer
  - That macro goes under the name **`NULL`** and should always be used!
- To guarantee that a pointer has become a null pointer we set its value using the **`NULL`** macro: `ptr = NULL;`
- Similarly, we can test for a null pointer:
  `if (ptr == NULL)...` or even `if (!ptr)...`

Pointer-related operators

- To store in ptr the address of our integer variable `k`, we use the unary `&` (address-of) operator and write:
  `ptr = &k;`
  - The `&` operator retrieves the lvalue (address) of `k`
  - The assignment operator "=" copies that to the contents of `ptr`
  - Now, `ptr` is said to "point to" `k`
- The dereferencing operator is the asterisk; and it is used as follows:
  `*ptr = 7;`
  This statement will copy `7` to the address stored in `ptr`. Thus if `ptr` "points to" `k`, the above statement will set the value of `k` to 7.
- When we use the $*$ this way, we are referring to the value of the memory block with address equal to `ptr`, not the value of the pointer itself!

Putting it all together

Fig. 2.3: `pointer-values.c`

```c
#include <stdio.h>

int j, k;
int *ptr;

int main(void) {
  j = 1;
  k = 2;
  ptr = &k;
```

```
j has the value 1 and is stored at 0x601048
k has the value 2 and is stored at 0x601058
ptr has the value 0x601058 and is stored at 0x601050
The value of the integer pointed to by ptr is 2
```

One-Dimensional Arrays 1

- Array
  - Sequence of elements of same type, any type **int**, **float**, **double**, **char**...
  - Fixed, constant length
  - 0-based access via integer index
    ```
    array[0]
    array[intVar]
    ```
  - No length information → you have to remember it yourself
  - No range checking → silent over-/underwriting or -reading possible ⇒ buffer-overflow/underflows
    ```
    int number[12];
    printf("%d", number[20]);
    ```
    Produces undefined output, may terminate, may not even be detected
  - Always initialize before use → the compiler does not do this for you!
- C99: In functions (and only there; not possible for global variables: located in compile-time-sized "data" section) the length can be set at initialization
  ```
  int vla[strlen(in)];
  ```

Fig. 2.4: `1darray.c`

```
#include <stdio.h>
main(void) {
        int number[12]; /* 12 cells, one cell per student */
        int index, sum = 0;
        /* Always initialize array before use */
        for (index = 0; index < 12; index++) {
            number[index] = index;
        }
        /* now, number[index]=index; will cause error: why ?*/
        for (index = 0; index < 12; index = index + 1) {
            sum += number[index]; /* sum array elements */
        }
        printf("sum: %d\n", sum);
}
```

- 2-dimensional arrays: are allocated with consecutive rows in memory. Be careful on how to iterate through the array, e.g.:
  1) `for each row { for each column ... }` → fast, since the caching can load entire rows for processing
  2) `for each column { for each row ... }` → can be considerably slower, since it makes caching less efficient

Example 2.3: `int weekends[52][2];`

[0][0]	[0][1]	[1][0]	[1][1]	[2][0]	⋯	[51][1]

↰ `weekends`; similar to Java: address (like a reference) accessible via the identifier

◇

- Array access

```
int points[3][4];
points[2][3] = 12;
printf("%d", points[2][3]);
```

do not use `points[3,4]`

- Analogous for higher-dimensional arrays

Strings

- String
  - handled as arrays of type `char`
  - Terminated by the NUL character `'\0'`

```
char name[6];
name = {'C','S','1','0','\0'};
printf("%s", name);
```

  → will print the content of `name` until it finds a `char` with value zero

- Functions to operate on strings
  `strcpy, strncpy, strcmp, strncmp, strcat, strncat, strstr, strchr`
  `#include <string.h>` at program's beginning
- Be careful: many of them exist in various versions, sometimes with and sometimes without buffer limit checking, or similar → look for the "memory-safest variant" that is available.
- We will study strings more deeply in chapter 5

Assignment operators

- `=` direct assignment of values, e.g., `a=b` puts the value of `b` into the variable `a`.
- `+=` add the value on the right hand side to the current value assigned to. Example: `a += 2` is the same as writing `a = a + 2`
  The same "calculate-and-assign" operator is available for all binary arithmetic and logical operators. Exercise: try them out, e.g., what does `-=`, `/=`, `^=`, ... compute?
- `++` incrementation operator. This one come in prefix and postfix form, with different effects. Example:
  - assigning `a++` to any variable, it takes the current value of `a`, and then increments `a` (post-increment).
  - assigning `++a` to any variable, it first increments the value of `a`, and then assigns it to the variable (pre-increment).

  The `--` is the respective decrementation operator

Operators 2

Arithmetic operators

+, −, *, /, %	the binary arithmetic operators
−	the unary sign change operator

Logical operators on bits

<<, >>	left and right bit-shifts
|	bitwise OR
&	bitwise AND
^	bitwise XOR
!	bitwise NOT (unary, prefix)

Operators 3

Comparison operators

- `==` equality (attention: do not mix up with the = in an if clause)
- `!=` inequality
- `<`, `>`, `<=`, `>=` less/greater than, less/greater than or equal
- `||` logical OR
- `&&` logical AND

Access operators → examples and details later in Chapter 3

- `[]` array element access
- `.` access members of **`struct`** and **`union`** data types (similarly to Java's member function syntax: `object.method`)

-> like the . operator, access of a member, but used with pointer variables:

- if p is a (normal) variable, we can write `p.memberField`
- if p is a pointer variable, we write `p->memberField`

Address and Memory operators → examples and details later in Chapter 5

& prefix address operator: given a variable a, we get its address by writing &a

* prefix operator to access a memory cell. If a stores an address, we can read and write content thereto by reading from or assigning a value to `*a`

Other operators

, the sequence operator. This merely declares several instructions as one single piece of code. It is mostly used in for loops, to put several increments into the per-iteration operation (third part of the for loop header, like in Java)

Operator Precedence 1

- For two operators op1 and op2, we write "op1 $\triangleleft$ op2", if op1 is evaluated before op2.
- Operators appearing next to each other like "$\triangleleft$ op1 op2 $\triangleleft$" have the same precedence.

<table>
<tr><th>Operator type</th><th>Operators in order of precedence</th></tr>
<tr><td>Primary Expression Operators</td><td>( ) ◁ [ ] ◁ . ◁ -> ◁ expr++ ◁ expr--</td></tr>
<tr><td>Unary operators†</td><td>sizeof ◁ (typecast) ◁ --expr ◁ ++expr ◁ ~ ◁ ! ◁ - ◁ + ◁ & ◁ *</td></tr>
<tr><td>Binary operators</td><td>* / % ◁ + - ◁ << >> < > <= >= ◁ == != ◁ & ◁ ^ ◁ | ◁ && ◁ ||</td></tr>
<tr><td>Ternary operator</td><td>? :</td></tr>
<tr><td>Assignment operators</td><td>|= ◁ ^= ◁ &= ◁ <<= ◁ >>= ◁ %= ◁ /= ◁ *= ◁ -= ◁ += ◁ =</td></tr>
<tr><td>Comma</td><td>,</td></tr>
</table>

† The unary & and * operators refer to obtaining and accessing memory addresses → later in Chapter 5

Operator Precedence 2

Precedence/Associativity

- If two operators from the same line occur together, they are evaluated in this sequence
- Example `a = b = c` → right-to-left → `a = (b = c)`
- The logical operators `&&` and `||` do short-circuit evaluation, a.k.a., lazy evaluation:
  - Note: The binary operators `&` and `|` do not!
  - If the left operand equals 0, the right operand is not evaluated
  - Important for any side-effects, e.g. `x++`