Systems Programming — SNP

Systemnahe Programmierung · Unified Study Summary & Exam Recipes (C)

Author: Simon Jeker · **Use:** open-book paper exam (printed in colour) · **Language:** English · **Code:** ANSI / POSIX C

22 topics · 18 step-by-step exam recipes · complete copy-ready C programs · memory, pointer & process diagrams.

How to read this reference

Each top-level topic starts on a new page with a coloured banner. A **colour band** groups related topics; the band name also prints in the top-right running header so you always know where you are. Colour is always backed by a visible label or border, so the sheet stays usable even if a page prints in grayscale.

- Foundations (§1–3)
- Logic & Functions (§4–6)
- Pointers & Memory (§7–11)
- Build & Quality (§12–14)
- Kernel & Memory (§15–17)
- Processes & IPC (§18–20)
- Shell (§21)
- Appendix (§22)

RECIPE — numbered exam procedure

⚠ **VERIFY** — check against lecture notes

NOTE — added clarification / demo

FIG — diagram caption

C syntax colours: `#include` preprocessor · `int` type · `return` keyword · `"text"` string · `123` number · `/* comment */`

Table of Contents

Index of Exam Recipes

Each recipe is a numbered, step-by-step procedure for a recurring exam task, marked **Use this when…** Recipes live inside the topic they belong to.

1 Computer Systems & Hardware

FOUNDATIONS

Layered system model

From the core outward: **Hardware** → **OS Kernel + hardware drivers** (kernel mode, full hardware access) → **System Calls** (the boundary between kernel and user) → **System Library** → **Programs / OS programs / Shell / user libraries** (user mode). The transition from outer to inner layers **always goes through system calls**.

In SNP terms: an application can be written in anything (e.g. Java); “systems-near” code is C, scripts, or assembler.

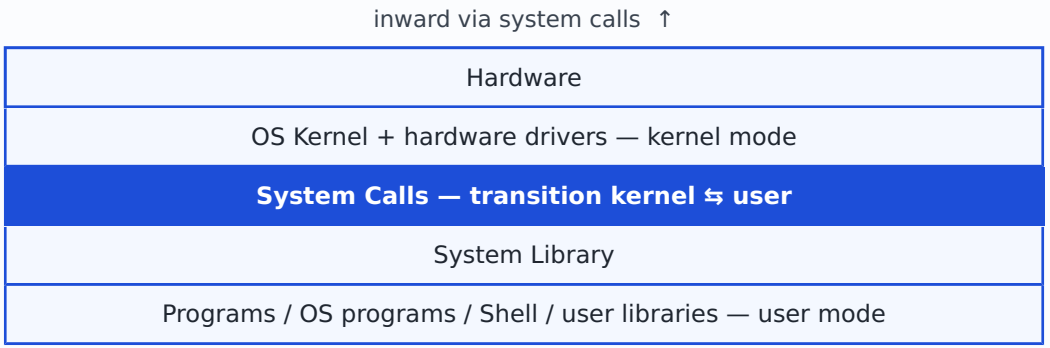


FIG The layered model. Every inward call crosses the system-call boundary; kernel mode has full hardware access, user mode is restricted.

Central hardware elements

- **CPU** (Central Processing Unit) — program execution, data processing, master of the system bus.
- **Memory** — stores data and instructions.
- **Input / Output** — interface to external devices (read/write interface to the outside world).
- **System bus** — the electrical connection between components. The CPU signals the desired accesses (who reads/writes, when, which data). It carries data lines, address lines and control signals.

CPU internals

- **Control Unit** (controls execution): **PC** Program Counter — address of the next instruction; **IR** Instruction Register — the instruction currently executed.
- **Data Path** (does the computing): **ALU** (integer arithmetic) and **registers** (fast scratch storage).
- **Bus Interface** — gives access to all system elements.

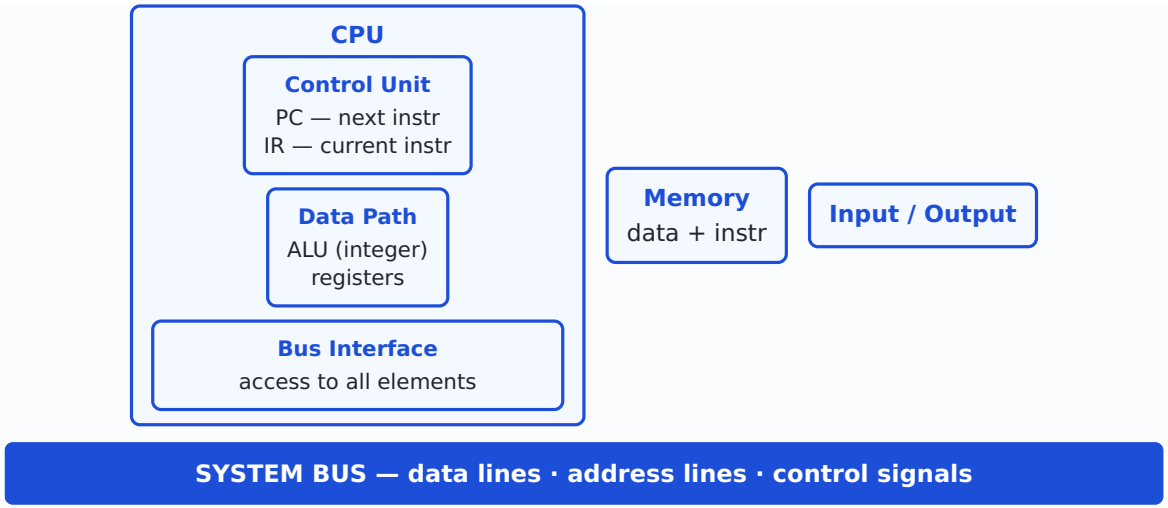


FIG CPU internals on the system bus: control unit (PC/IR), data path (ALU/registers), and the bus interface linking memory and I/O.

Memory types

- **RAM** — every cell individually addressable; *volatile* (keeps data only while powered).
- **ROM** — data defined at production time; retains data independent of power.
- **Variants:** non-volatile RAM (keeps data without power); xxPROM / Flash (ROM programmable at runtime).

System performance & CPU offloading

- **Acceleration inside the CPU: Cache** — faster access to temporarily stored data; **Pipeline** — faster execution via staggered/overlapped instruction stages.
- **Offloading work from the CPU: IC** Interrupt Controller; **DMA** Direct Memory Access (copies data without the CPU); **FPU** Floating Point Unit; **DSP** Digital Signal Processor; **GPU** Graphics Processing Unit; **MPU** Memory Protection Unit (monitors address accesses).
- **Interrupt** = an interruption of the running program to execute a function, after which execution continues at the interrupted point.

Operating system (OS)

- **Tasks:** manage hardware resources and abstract them through OS concepts.
- **Resources:** CPU, memory, I/O, plus acceleration units (cache, pipeline, IC, DMA, …).
- **Abstractions:** tasks (parallel processing, isolation), memory (protection, virtual memory), filesystem (hardware-independent files).

2 C Language Elements

FOUNDATIONS

C compared to Java

- C is **procedural** (not OO): no classes, functions instead of methods; a program is a collection of functions. No classes/namespaces → all functions must have **distinct names**.
- Every program has a `main()` (the entry point); leaving `main` terminates the program. `main` is a special function that calls the others.
- The C compiler produces **machine code** (platform-dependent, no VM); Java produces bytecode. C is usually faster.
- C has no I/O syntax — I/O is done via standard-library functions (e.g. `printf`).
- C is more error-prone than Java: no garbage collection; no array-bounds checking; pointers can be misused; variables live on heap or stack; `String` and `boolean` types are missing (strings are char arrays; booleans from `<stdbool.h>` since C99). Source files end in `.c`; compilers often emit no warning (e.g. loss of precision) by default.

Minimal program & compiling

```c
/* helloworld.c */
#include <stdio.h>   // printf
#include <stdlib.h>  // EXIT_SUCCESS
int main(void) {
    (void)printf("Hello\n");
    return EXIT_SUCCESS;
}
```

```
# build & run
gcc -o helloworld helloworld.c
./helloworld
```

Lexical elements, comments, names

- **Separators:** space, tab, end-of-line.
- **Comments:** `/* ... */` (multi-line) and `// ...` (since C99, to end of line).
- **Names:** letters, digits, underscore; first character must be a letter or `_`; case-sensitive; cannot be a keyword.

Basic data types (typical sizes)

Sizes are hardware-dependent. Four base types: `char`, `int`, `float`, `double`.

Type	Bytes	Range (typical)
`char`	1	−128…127 (signed) or 0…255 (unsigned)
`int`	4	-2^{31} … $2^{31}-1$
`float`	4	$\pm 3.4 \cdot 10^{38}$
`double`	8	$\pm 1.79 \cdot 10^{308}$
`short [int]`	2	−32768 … 32767
`long [int]`	8	-2^{63} … $2^{63}-1$
`long long`	8	-2^{63} … $2^{63}-1$

Modifiers: `signed`, `unsigned`, `short`, `long`, `long long`. An unsigned type covers 0 … 2^n-1.

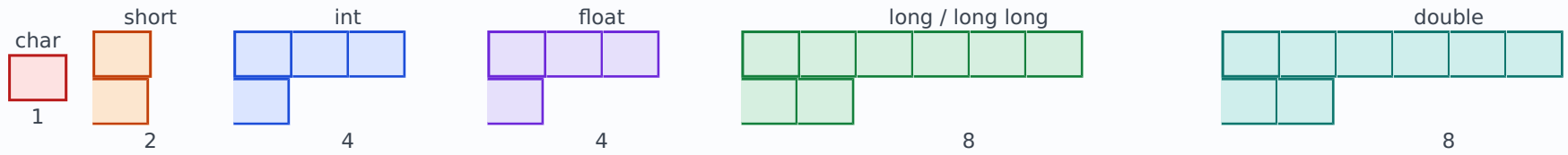


FIG Byte sizes of the base types (typical 64-bit Linux, LP64). One cell = one byte.

⚠ **VERIFY** The original summary listed `long` as 8 bytes. True on typical 64-bit Linux (LP64), but `long` is only guaranteed ≥ 4 bytes and is 4 bytes on Windows (LLP64) and 32-bit systems. Treat the table as "typical 64-bit Linux", not universal.

Fixed-size types

- `<stdint.h>`: `int8_t` / `uint8_t` … `int64_t` / `uint64_t` — OS-independent, exact sizes (important for hardware-near code).
- `<stddef.h>`: `size_t` (for `sizeof`, indices), `ptrdiff_t` (pointer subtraction), `NULL`.

Literals

Kind	Form	Notes
Decimal	`1234`, `3987`	
Octal	`0555`, `037`	leading `0`; unsigned
Hexadecimal	`0x3A`, `0x23`	leading `0x`; unsigned
Character (ASCII)	`'A'` = 65	has type `int` in C
const constant	`const double pi = 3.14159;`	initialised once
Symbolic constant	`#define MAX 1000`	pure text replacement

- **Integer suffixes:** `L` =long, `LL` =long long, `U` =unsigned (`123ULL`, `0x8868L`).
- **Floating-point:** `3.1415` / `1.6e3` are double; `F` =float, `L` =long double.
- **Escapes:** `\n \t \\ \' \" \0`; `'\x41'` == `'\101'` == `'A'`.
- **Strings:** not a separate type → `char[]`, NUL-terminated, storage = characters + 1. `"ZHAW"` needs 5 bytes.

Symbolic constants — the classic pitfall

```c
#define MAXLOOP 1000  // no '=', no ';' -> pure text substitution
// Trap:
#define K 10 + 0
int arr[K * 2];       // becomes 10 + 0 * 2 = 10, not 20!
```

A `#define K 10` followed by `int K = 20;` would expand to `int 10 = 20;` — a compile error.

Replacement specifiers in format strings

`%d`, `%i` (int), `%u` (unsigned int), `%c` (char), `%s` (char*, up to `\0`), `%f` (float/double). Width/precision: `%m.df`, e.g. `%.3f` → `5.123`; `%10.3f` →right-justified in 10 columns.

NOTE A small complete program exercising the base types and their format specifiers — the kind you may have to reproduce from memory.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>              // bool, true, false (C99)
int main(void) {
    char c = 'A';                 // 1 byte, prints as char or int
    int i = 1234;                 // %d signed integer
    unsigned u = 42u;             // %u unsigned integer
    double d = 3.14159;           // %f float and double both use %f
    bool ok = true;               // really an int: 1
    printf("c = %c (as int: %d)\n", c, c);    // A (as int: 65)
    printf("i = %d\n", i);                    // 1234
    printf("u = %u\n", u);                    // 42
    printf("d = %.3f width: %10.3f\n", d, d);  // 3.142 ...
3.142
    printf("ok = %d\n", ok);                  // 1
    return EXIT_SUCCESS;
}
```

Declaration, definition, typedef · Boolean

- **Declaration** fixes how a name may be used. **Definition** allocates storage / gives a function body (a definition is always also a declaration). No default values except for global / static variables.
- `typedef int Index;` → an alias (not a new type; interchangeable with `int`).
- Before C99 there was no `bool`: 0 is false, anything else true (`while(x)` ≡ `while(x != 0)`). Since C99: `#include <stdbool.h>` → `bool b = false;`.

3 Operators, Expressions & Type Casts

FOUNDATIONS

Operator groups

Group	Operators
Arithmetic	`+ - * / %`
Relational	`> >= < <=`
Equality	`== !=`
Logical	`&& || !`
Increment / decrement	`++ --`
Bitwise	& (AND) | (OR) ^ (XOR) ~ (one's complement) << >> (shift)
Assignment	`= += -= *= /= %= &= ^= |= <<= >>=`
Conditional (ternary)	`? :`
Address / dereference	& (address-of) * (dereference)

Apart from Java's `>>>` (which C lacks) and the pointer operators, the operators behave as in Java.

Precedence & associativity (high → low)

Operators	Category	Assoc.
`() [] . -> ++ --` (postfix)	Postfix	L → R
`! ~ ++ -- + - *` & (type) `sizeof`	Unary	R → L
`* / %`	Multiplicative	L → R
`+ -`	Additive	L → R
`<< >>`	Shift	L → R
`< <= > >=`	Relational	L → R
`== !=`	Equality	L → R
`& ^ |`	Bit AND / XOR / OR	L → R
`&& ||`	Logical	L → R
`?:`	Conditional	R → L
`= += -= ...`	Assignment	R → L
`,`	Sequence	L → R

Expressions & type casts

- In an expression, operands are implicitly promoted to the **largest type** involved.
- `3/2` → 1 (integer division, remainder discarded); `3.0/2` → 1.5 (double).
- `=` converts the right side to the type of the left: `int i = 3.1415;` → i = 3; `i = 2.99 + 3;` → i = 5.
- **Cast:** `(type) expr`. `(double)i / 3` forces double division. Use casts sparingly (a design smell if overused).

> **NOTE** Integer vs floating division and the effect of a cast — a frequent trace-the-output trap.

```c
#include <stdio.h>
int main(void) {
    int a = 3, b = 2;
    printf("%d\n",   a / b);          // 1   int / int truncates
    printf("%.1f\n", 3.0 / b);        // 1.5 one double operand
    printf("%.1f\n", (double)a / b);  // 1.5 cast forces double
    int i = 2.99 + 3;                 // assign truncates -> 5
    printf("%d\n", i);                // 5
    printf("%d\n", 7 % 3);            // 1   modulo remainder
    return 0;
}
```

RECIPE R1 — Predict the output of a program / trace printf statements

Use this when… The question shows a short program (or a list of `printf(...)` lines) and asks “Was ist die Ausgabe?” / “What is printed?”, or gives a blank to fill after each line. Tasks mix integer division, casts, `sizeof`, enum values, `++` / `--`, and array/pointer dereferences. *(FS20 A2a, FS16 A2a, FS12 A1)*

1. Walk the statements top-to-bottom; keep a running table of every variable's current value (for pointers: which element it points at).
2. Evaluate each expression by C rules: int÷int truncates (`3/2` →1); if any operand is floating, the result is floating (`3.0/2` →1.5); a cast `(int)x` truncates toward zero.
3. `sizeof`: a `char` is 1; a string literal / `char[]` counts characters +1 for `'\0'` (`"SEPFS16"` →8); `sizeof(array)` = elements×element size *only on the real array* (a parameter is just a pointer).
4. `enum` without explicit values numbers from 0 (`rot` =0, `gruen` =1, `blau` =2).
5. Arrays/pointers: `a[n] == *(a+n)`; `*p` / `p[0]` is the first element; `p[0]-p[2]` subtracts two values (an int) — not pointer arithmetic.
6. Match each specifier: `%c` a character (via ASCII), `%d` the integer value, `%s` from the address up to the next `'\0'`, `%f` 6 decimals by default.

Worked example. `char c[]="SEPFS16"; char *p=c;` → `sizeof(c)` =8 · `p[0]` ='S' · `p[0]-p[2]` = 'S'−'P' = 83−80 = 3 · `p+p[0]-p[2]` = p+3 → prints `"FS16"`.

> ⚠ **VERIFY** Evaluation order of side effects in one expression (e.g. `printf("%d %d", x++, x++)`) is **unspecified** — do not try to predict a single answer; see §14.

RECIPE R2 **Extract one digit of an integer with % and /**

Use this when… A task says “determine the last / second-to-last / n-th digit” and tells you to use the modulo operator. Keywords: *Modulo-Operator*, *Ziffer* (digit), *zweitletzte* (second-to-last). *(FS20 A3a, FS16 A3a)*

1. Last digit = `z % 10`.
2. To reach position p (0 = units), divide it down first: `z/10` drops one digit, `z/100` drops two, …
3. Combine: digit at position p = `(z / 10^p) % 10`. Second-to-last → `(z / 10) % 10`.
4. Mask-and-divide variant (used by the key): `((z % 100) - (z % 10)) / 10` — `%100` keeps the last two digits, subtracting `%10` removes the units, `/10` shifts down.

```c
int z = 123;
int d = (z / 10) % 10;                /* simple form -> 2 */
int e = ((z % 100) - (z % 10)) / 10;  /* mask form   -> 2 */
```

⚠ **VERIFY** The mock's line `z = (z % 100) - (z % 10)) / 10;` has an unbalanced parenthesis; the correct form is `z = ((z % 100) - (z % 10)) / 10;`. Both that and `(z/10)%10` give 2 for `z=123`.

4 Control Structures

LOGIC & FUNCTIONS

Syntax is essentially identical to Java.

```
if (n > 0)        { r = 1; }
else if (n == 0)  { r = 0; }
else              { r = -1; }

for (int i = 1; i <= max; i++) { sum += i; }
while (i <= max)               { sum += i; i++; }
do { sum += i; i++; } while (i <= max);

switch (n) {
    case 1:          r = 1;  break;
    case 2: case 3:  r = 10; break;
    default:         r = 0;  break;
}
```

`break` is required in each case; otherwise execution **falls through** into the following cases.

RECIPE R3 **Translate a worded “rule / exception / exception-to-exception” into nested conditionals**

Use this when… A task gives layered rules of the form “rule, but exception, but exception-to-the-exception” and asks for C that prints YES/NO. The textbook case is leap years (÷4 yes, but ÷100 no, but ÷400 yes). *(FS20 A3b, FS16 A3b)*

1. Read the rules outermost-first; each “exception” becomes a test **nested inside** the previous branch.
2. Build the nest: outer `if` (÷4) → inside `if` (÷100) → inside `if` (÷400) `yes;` `else no;` ; the ÷100's `else yes;` ; the outer `else no;` .
3. Make sure every branch reaches exactly one output (each `else` binds to the nearest unmatched `if`).
4. Test the four boundary years: 2016 (÷4 only → yes), 1900 (÷100 not ÷400 → no), 2000 (÷400 → yes), 2017 (not ÷4 → no).

```
int jahr = 2016;
if (jahr % 4 == 0)
    if (jahr % 100 == 0)
        if (jahr % 400 == 0) printf("YES");  /* e.g. 2000 */
        else                 printf("NO");   /* e.g. 1900 */
    else                     printf("YES");  /* e.g. 2016 */
else                         printf("NO");   /* e.g. 2017 */

/* Equivalent single expression: */
/* (jahr%4==0 && jahr%100!=0) || (jahr%400==0)
   is true exactly for leap years */
```

NOTE A complete, runnable leap-year program built from the recipe above, ready to copy.

```c
#include <stdio.h>
#include <stdbool.h>
/* A year is a leap year if divisible by 4,
   except centuries, unless divisible by 400. */
bool is_leap(int y) {
    if (y % 4 == 0) {
        if (y % 100 == 0) {
            return (y % 400 == 0);  /* 1900 no, 2000 yes */
        }
        return true;                /* e.g. 2016 */
    }
    return false;                   /* not divisible by 4 */
}
int main(void) {
    int years[] = {2016, 1900, 2000, 2017};
    for (int i = 0; i < 4; i++)
        printf("%d: %s\n", years[i],
               is_leap(years[i]) ? "YES" : "NO");
    return 0;
}
```

5 Functions (by value & by reference)

LOGIC & FUNCTIONS

Basics

- Functions structure a program; code that occurs repeatedly is implemented only once.
- **Definition:** `Type Name(parameter-list){ ... }` — the type is the return type (`void`=nothing); parameters act like local variables.
- **Declaration (prototype):** like the definition but the body is replaced by `;`
- **Call:** `Name(argument-list)`, usable inside expressions, e.g. `v = 5 + max(8, a);`

```c
int max(int a, int b);          // declaration (prototype)
int main(void) {
    ... v = max(1000, x); ...   // call
}
int max(int a, int b) {         // definition
    if (a >= b) return a;
    return b;
}
```

Two fundamental rules

- **DBU — Declared-Before-Use:** every name (type, variable, function, `#define`) must be declared before use. Fix: an explicit declaration, or `#include` the right header (`stdio.h`→`printf`, `stdlib.h`→`EXIT_SUCCESS`).
- **ODR — One-Definition-Rule:** each name has exactly one definition in the whole program. You may declare it (identically) any number of times, but define it once. Public declarations belong in headers; headers contain no function/variable definitions. Exception: identical type definitions may appear in several source files.

Parameters & return values

Kind	As parameter	As return value
Basic data types	Valid	Valid
Structures & enums	Valid	Valid
Arrays	Valid (decays to pointer)	**Invalid**
Pointers	Valid	Valid

Arrays cannot be returned by value — only a pointer to the array data (i.e. by reference).

“By value” (central concept)

- Argument values are copied into the function; parameters behave like local variables. Changing them inside has **no effect** on the caller.
- `increment1(int a){ ++a; }` leaves the original 5; `return ++a;` + `a = increment2(a);` yields 6.
- `swap(a,b)` by value does **not** swap the originals — fix: pass by reference (pointers).

By reference (via pointers)

You pass an address (`&var`). The address is still copied by value, but the copy points at the same memory → the referenced variable **can be modified**.

```c
void swap_int(int *a, int *b) {
    int saved = *a; *a = *b; *b = saved;
}
int x = 10, y = 20;
swap_int(&x, &y);  // now x==20, y==10
```

NOTE A self-contained driver that shows by-value leaving the caller untouched while by-reference (pointers) changes it.

```c
#include <stdio.h>
/* by value: the copy changes, the caller does not */
void inc_by_value(int a)  { a = a + 1; }
/* by reference: the caller's variable changes */
void inc_by_ref(int *a)   { *a = *a + 1; }
int main(void) {
    int n = 5;
    inc_by_value(n);
    printf("after by-value: %d\n", n);      // 5 (unchanged)
    inc_by_ref(&n);
    printf("after by-reference: %d\n", n);  // 6 (changed)
    return 0;
}
```

const parameters

- **By value:** `const` is barely relevant (only the local copy is const).

- **By reference:** `const` means the memory at the address is read-only — it prevents accidental writes and documents intent.

```c
int starts_with_capital(const char *s) {
    return s && isupper(*s);  // reading only
}
```

Arrays, structs & functions as parameters

- Arrays can only be passed by reference — the array name is the address of the first element; the parameter must be pointer-to-element. `int a[]` ≡ `int *a`; a size in `[]` is ignored. Inside the function you cannot tell a single variable from an array start → **always pass the length separately**.
- **Multi-dimensional arrays:** all dimensions except the first must be given. `void print_matrix(double (*m)[3]);` ≡ `double m[][3]`.
- **Array of pointers:** `int *a[10]` → `int **`; `func(int *b[])` ≡ `func(int **b)`.
- Structs can be passed by reference or by value. By reference (ideally `const`) avoids copying.
- Functions can be passed by reference (function pointers, below).
- **Variable number of parameters:** the ellipsis `...` is last; at least one normal parameter precedes it. Macros from `<stdarg.h>`.
- **void as parameter list:** `int f(void);` is a truly empty list (the compiler checks the call). `int f();` means no checking → always use `void`.

Function pointers

```c
void logger(char *msg);
void (*out)(char *);     // pointer to a function
out = logger;            // == &logger (the & is optional)
out("Hello");            // == (*out)("Hello") (the * is optional)
typedef void (*log_fp_t)(char *);
log_fp_t f = logger; f("Hello");
```

Functions as function parameters:

```c
// double (*func)(double): pointer to a double -> double function
double trapez(double (*func)(double), double a, double b, int n) {
    double s = 0, t = a, h = (b - a) / n;
    for (int i = 0; i < n; i++) {
        s += (func(t) + func(t+h)) / 2 * h; t += h;
    }
    return s;
}
trapez(sin, 0, pi/2, 100);
```

Variable argument lists (varargs):

```c
int mittelwert(unsigned n, ...) {
    va_list args; int sum = 0;
    va_start(args, n);              // last fixed parameter
    for (unsigned i = 0; i < n; i++)
        sum += va_arg(args, int);   // next value
    va_end(args);                   // mandatory
    return n ? sum / n : 0;
}
mittelwert(3, 4, 5, 6);  // = 5
```

RECIPE R5 **Read / explain a declaration (especially a function pointer)**

Use this when… A task asks “explain the meaning of line N” and the line is a declaration — most often a function-pointer parameter, a static variable, or an array definition. *(FS12 A5)*

1. Read declarations with the **right-left rule**: start at the name; go right (`()` =function, `[]` =array, then read its parameters), then go left (`*` =“pointer to”, type/qualifier last).
2. **Function pointer** `void (*printOut)(int)` : “printOut is a pointer to a function taking an int and returning void.” The parentheses around `*printOut` are what make it pointer-to-function rather than a function returning a pointer.
3. Name the **storage class** if present: file-scope `static int counter;` = a global limited to this file (internal linkage), zero-initialised, lifetime = whole program (§6). `const int test[]={...};` inside a function = an automatic (stack) array, here read-only.

Worked example. Line: `void printStdout(void (*printOut)(int i), int outVal);` → “declaration of printStdout, returns void; parameter 1 is a pointer to a function taking an int and returning void; parameter 2 is an int outVal.”

return

- `return` leaves the function immediately. void→ `return;` ; with a value→ `return -1;`
- A value-returning function without a `return` is a programming error. Return happens **by value**.
- `main` : if `return` is omitted the compiler adds `return 0;` ; the value goes to the calling environment (the shell).

Returning arrays / the lifetime trap

- To “return an array”, return the address of its first element (no `[]` in the return type).
- **Never return the address of a local variable** — its stack memory is gone after return. Allocate with `malloc` instead.

```c
int *create_copy(const int a[], int n) {
    int *cp = malloc(n * sizeof(int));
    if (cp) memcpy(cp, a, n * sizeof(int));
    return cp;
}
```

RECIPE R4 **Diagnose & fix “function returns address of a local variable”**

Use this when… A function returns `&local` (or a pointer to a local variable / local array) and the task asks “what mistake did the programmer make?” and/or “name three ways to fix it”. The compiler hint is `warning: function returns address of local variable` . *(FS12 A3a/b)*

1. **State the bug:** the local lives on the stack; its memory is reclaimed when the function returns, so the returned pointer **dangles** (points at memory that may be reused) — undefined behaviour.
2. **Fix 1:** declare the local `static` → it lives for the whole program (but is then shared across calls).
3. **Fix 2:** allocate on the heap inside the function with `malloc` and return that pointer (the caller must `free`).
4. **Fix 3:** let the caller allocate the storage and pass in a pointer for the function to fill.

```c
struct point3D* initPoint3D(void) {
    struct point3D p;     /* <-- local on the stack */
    p.x = 5; p.y = 10; p.z = 20;
    return &p;            /* BUG: address of a local */
}
```

⚠ **VERIFY** Do not “fix” it by returning the address anyway. It is undefined behaviour even if it appears to work on some runs.

Recursion

A function calls itself directly or indirectly. Each call gets its parameters by value again and its own local variables per level.

```c
int fakultaet(int n) {   /* factorial, recursive */
    if (n < 2) return 1;
    else return n * fakultaet(n - 1);
}
```

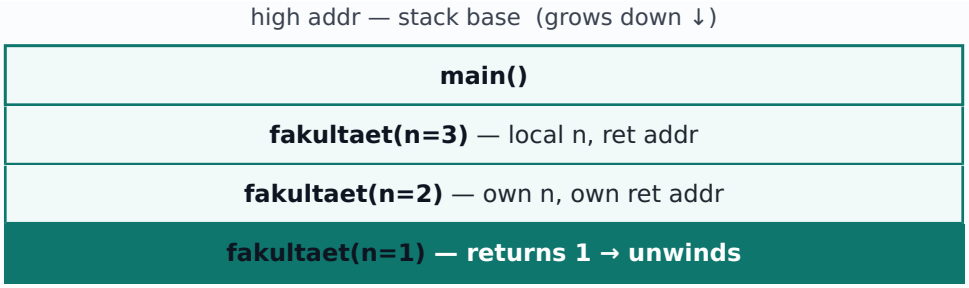


FIG Each recursive call gets its own stack frame (own n, locals and return address). Frames are pushed as the recursion deepens and popped as it unwinds.

6 Variables: Scope, Lifetime & Storage

LOGIC & FUNCTIONS

Visibility & lifetime

Type	Visibility	Notes
Local (automatic)	Block / function	Lives until end of block; initial value **undefined** (garbage); on the stack. Equal names in different functions are no conflict.
Local-static	Block / function	Value **persists across calls** (function keeps state); init 0.
Global	Source file / whole program	Defined outside all functions; visible everywhere; implicitly init 0; use with care.
Global-static	Program (own file only)	Like global but visible only in its own source file; value persists; init 0.

“Hiding” (an inner declaration shadows an outer one) is poor style.

Attributes — inside functions / blocks

Specifier	Storage	Init	Visible	Lifetime
`auto` / (none)	Stack	garbage	function / block	until end of block
`register`	CPU register	garbage	function / block	until end of block
`static`	Data segment	0	function / block	until end of program

Attributes — outside functions

Specifier	Storage	Init	Visible	Lifetime
(none) = global	Data segment	0	whole program	until end of program
`static`	Data segment	0	own module only	until end of program

Storage-class keywords & guidance

- `register` is a hint to keep a variable in a CPU register (today the compiler optimises on its own).
- `extern` is a **declaration** (refers to a global defined elsewhere), **not a definition**.
- **Guidance:** `auto`/`register` are practically never needed. `static` *outside* functions: use whenever possible. `static` *inside* functions: use sparingly (the function carries state). `extern` is usually a code smell; prefer accessor functions.

NOTE A demo of a function-local `static` keeping state between calls, plus scope shadowing.

```c
#include <stdio.h>

int counter(void) {
    static int n = 0;   /* init once; survives between calls */
    n = n + 1;
    return n;
}

int x = 100;            /* global */

int main(void) {
    printf("%d %d %d\n", counter(), counter(), counter()); /* 1 2 3 */

    int x = 5;          /* local x shadows the global x */
    printf("local x = %d\n", x);     /* 5 */
    {
        int x = 9;      /* inner block shadows again */
        printf("inner x = %d\n", x); /* 9 */
    }
    printf("local x = %d\n", x);     /* 5 again */
    return 0;
}
```

7 Pointers & Arrays

POINTERS & MEMORY

Memory & sizeof

- Memory is a sequence of consecutively numbered bytes. An element's address is the **lowest** of its byte addresses.
- `sizeof` gives the byte size of a type/expression — **at compile time**, not runtime.
- `sizeof(pointer)` is always the same on a given system (8 on 64-bit, 4 on 32-bit), regardless of what it points to.
- **Little-endian:** the least-significant byte is stored at the lowest address first. `int v = 0x4D2;` → bytes `D2 04 00 00`.
- **Standard assumptions:** `sizeof(char)=1`, `sizeof(int)=4`, `sizeof(double)=8`, `sizeof(pointer)=8` (4 on 32-bit).

`int v = 0x000004D2;` (value = 1234) — stored little-endian, least-significant byte first

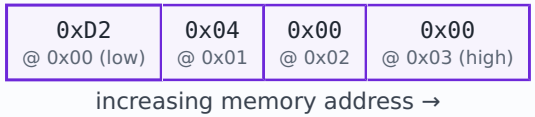


FIG Little-endian storage: the least-significant byte (0xD2) sits at the lowest address; bytes go up as the address increases.

Arrays

- An array is a sequence of equal elements, **contiguous** in memory. Indices 0…N-1; it occupies N·sizeof(element) bytes.
- An array **does not know its own length** and there is no bounds checking: `a[8]` or `a[-3]` cause no runtime error.
- The array name is a **fixed start address**, not modifiable (`a = b;` and `a++;` are compile errors).

```c
int a[5] = {4,7,12,77,2};  // all set
int b[]  = {4,7,12};       // length implicitly 3
int c[5] = {4,3};          // rest = 0
int d[5] = {1,2,3,4,5,6};  // compile error (too many)
a[3] = 3;                  // assign single element
```

Other array properties

- Arrays cannot be directly compared nor assigned (only element-wise, or via `memcpy`/`memcmp`). Structs *can* be assigned (`s = s2;`).
- No default values; no exceptions; no length function. When an array is passed, **only the pointer is passed**.

Determining length

- `sizeof(a) / sizeof(a[0])` — works **only at the definition**.
- It does **not** work for array parameters: there a is just a pointer → the length must be passed as a parameter.

Pointers

- A pointer is its own variable that holds an **address** → it "points at an object".
- A pointer has a **type**, so it knows how far the referenced value reaches.
- Operators: * dereference, & address-of.
- **Declaration trap:** `int *p, q;` → q is an `int`, not a pointer (the * binds to the name).
- An uninitialised pointer is undefined → crash when dereferenced.

```c
int var;            // a variable of type int
int *pt;            // a pointer to int
pt = &var;          // assign an address

int x = 1, y = 2;
int *pt2 = &x;      // pt2 -> x
y = *pt2;           // y = 1 (deref)
```

FIG A pointer is a variable holding the address of another variable. `*pt` reads/writes the pointed-to object; `&var` yields its address.

Types of pointers

- **void pointer** (`void*`) — points at a “naked” address; assignable to/from any pointer without a cast.
- **NULL pointer** — stands for address 0 (“points at nothing”); used to signal an error.

```
char a[]  = "hello";  // content modifiable, address FIXED
char *pa  = "hello";  // pointer variable: can be re-pointed later
a  = pa;              // compile error (a is fixed)
pa = a;               // OK
```

const & pointers

Declaration	Pointer	Pointed-to object
`int * const p`	const	variable
`const int * p`	variable	const
`const int * const p`	const	const

Structures & pointers

`->` accesses members of a struct given via a pointer: `sp->x` is short for `(*sp).x`.

Array ↔ pointer relationship

- In an expression an array **decays** to a pointer to its first element: a and `&a[0]` have the same value.
- Index access is always pointer arithmetic: `x[n] == *(x + n)` — whether x is an array or a pointer.

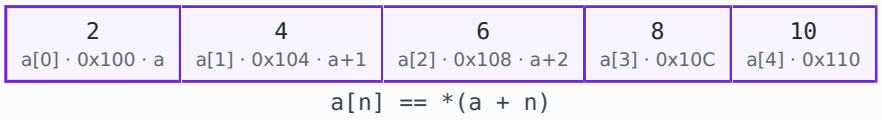


FIG An array decays to a pointer to its first element, so `a[n]` is just `*(a+n)`. Each +1 step advances by one element (here 4 bytes).

RECIPE R6 **Array-processing function using pointer arithmetic (not indexing) + const**

Use this when… A task says “implement function X; use **pointer arithmetic, NOT array notation**; the elements must not be changed.” Keywords: *Pointer-Arithmetik, nicht die Array Notation, nicht verändert* (→ add `const`). (FS12 A2 — sumOfArrayElements)

1. **Signature:** pass `const ElemType *array` (read-only) and a separate `int length` (a function never knows an array's length on its own).
2. Declare a **walking pointer** `const ElemType *p;` plus your accumulator.
3. Loop with pointer comparison: `for (p = array; p < array + length; p++)` — `array+length` is the one-past-the-end address (legal to compare, never to dereference).
4. Access the current element with `*p` (never `array[i]`).
5. Return the result.

```c
int sumOfArrayElements(const int *array, int arrayLength) {
    int sum = 0;
    const int *p;
    for (p = array; p < (array + arrayLength); p++) {
        sum += *p;   /* pointer arithmetic, not array[i] */
    }
    return sum;
}
```

`const` on a by-reference parameter = “I only read this”; it earns a point and documents intent (§5).

8 Pointer Arithmetic

POINTERS & MEMORY

Core rule

If p points to the first element of an array, then `(p + i)` points to the i-th element. The compiler computes `p + i·sizeof(element)` bytes. Compiler conversion: `x[n]` → `*(x + n)`.

Operation	Result
`pointer + int` / `int + pointer`	new address (pointer)
`pointer - int`	new address (pointer)
`pointer - pointer`	number of **elements** (offset), not bytes
`pointer * pointer`, `pointer /`	not allowed
`== != < <= >= >`	compare addresses (meaningful within the same array)

`a + 5` = the address just past the last element of `a[5]` — valid for comparisons, but must **not** be dereferenced.

Worked example

```c
int array[5] = {2, 4, 6, 8, 10};
int *pointer;
pointer = array + 3;   // same as &array[3]
*(pointer + 1) = 17;   // equivalent to p[1]=17, i.e. a[4]=17
```

Declaration examples

```c
int *p;
char *d[20];        // array of pointers
double (*e)[20];    // pointer to an array
char **ppc;         // pointer to pointer
```

Multi-dimensional arrays

- When an array is used in an expression it is implicitly converted to a pointer to the first element (of the first dimension).
- Regular 2-D `int m[2][3]`: contiguous, row by row. `m[i][j] ≡ *(*(m+i)+j)`.
- In parameters, all dimensions **except the first** must be given.

```c
a[2]    == *(a + 2)
a[2][3] == (a[2])[3] == *(*(a + 2) + 3)

int m[2][3] = {{1,2,3},{4,5,6}};
```

← row 0 ———→ ← row 1 ———→ · stored row by row, contiguous · `m[i][j] == *(*(m+i)+j)`

FIG A real 2-D array is stored row by row in one contiguous block, so `m[i][j]` is `*(*(m+i)+j)`.

Jagged arrays

Jagged arrays can hold differing numbers of elements; represented as one-dimensional arrays of pointers to separate memory regions (e.g. month names).

```c
char *str[] = {"Monday","Tuesday","Wednesday","Thursday",
               "Friday","Saturday","Sunday"};
```

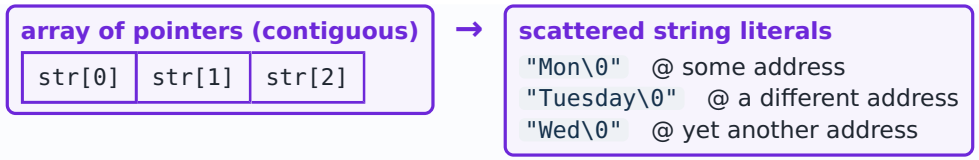


FIG An array of pointers: the pointers are contiguous, but the strings they point to are scattered, so address differences between the literals are **undefined**.

⚠ **VERIFY** The original summary wrote `*(p+1)[2]` to mean "3rd char of Tuesday". Because `[]` binds tighter than unary *, `*(p+1)[2]` actually parses as `*((p+1)[2]) = *(p[3])` = the first char of `"Thursday"`. To get the 3rd char of `"Tuesday"` you need `(*(p+1))[2]` or `p[1][2]`. Double-check which the lecturer intended.

Pointer difference: elements vs. bytes

Rule: `qt - pt` divides the byte difference by `sizeof(element)`. Casting the pointers to `int` makes you compute in bytes.

```
double table[10];
double *pt = table, *qt = table + 10;
printf("%d", qt - pt);              // 10 (elements)
printf("%d", (int)qt - (int)pt);    // 80 (bytes = 10*8)
```

Out-of-bounds & const examples

```
// Example 1 -- no runtime checks
int a[5] = {1,2,3,4,5};
a[3] = 4;     // fine
a[8] = 8;     // no error reported!
a[-3] = -3;   // no error reported!

// Example 2
const int b[5];
b[0] = 33;    // compile error

// Example 3
void *vp;
double *dp = vp;  // no error (void* assigns freely)
```

RECIPE R7 **Fill an address / content table as code runs (the "array memory" task)**

Use this when… A table of cells (often headed *Adresse [HEX] / Inhalt [HEX]*) is given, an array is declared at a stated start address, and a few lines mutate it step by step. You fill in each cell after each line. Keywords: *Ergänzen Sie die Adressen und Inhalte*, a start address like 0x68, "values carry over from one sub-task to the next." (FS20 A1, FS16 A1)

1. **Write the addresses first.** Step per element: `char` +1, `int` +4, `double` +8 (work in hex!). E.g. char array at 0x68 → cells 0x68, 0x69, 0x6A.
2. Enter the initial contents: `{'\0'}` → every cell 00.
3. Process each code line in order, carrying values forward into the table.
4. For `*p` or `*(p+i)`: first decide which cell (the address), then read/write the content there.
5. Convert number ↔ character via ASCII. When an int value is stored into a char cell, keep only the **low byte** (mod 256).
6. At the end, print as char if asked.

```
// txt[3] @ 0x68 (char => +1) -> cells 68 69 6A
char txt[3] = {'\0'};   // => 00 00 00
txt[2] = 80;            // 80 dec = 0x50 => 00 00 50
char *p = txt;
*p = *(p+2) + 3;        // 0x50=80,+3=83=0x53 => 53 00 50
*(p+1) = (int)txt - 3;  // 0x68=104,-3=101=0x65 => 53 65 50
// as char: 53='S' 65='e' 50='P' => "SeP"
```

ASCII anchors: `'0'=48=0x30`, `'A'=65=0x41`, `'a'=97=0x61`, space=32=0x20, `'\0'=0`. 1 hex digit = 4 bits; `0x50 = 5·16 = 80`; `0x65 = 6·16+5 = 101`.

⚠ **VERIFY** FS16 A1(d) reads `*(p+1) = (int)txt + 1;` with the array at 0x100, yet the key shows content 0x65. That only fits an array at 0x64; with 0x100 the low byte would be 0x01. The FS16 answer key looks inconsistent — trust the method (keep the low byte) over that one printed value.

RECIPE R8 Decide “Is this expression pointer arithmetic?” (YES / NO)

Use this when… A line is shown and the question is literally “*Ist dieser Ausdruck Pointerarithmetik?*” (answer YES/NO). (FS20 A2b/c, FS16 A2b/c)

1. **YES** if at least one operand is an address / pointer and + or - is used → the result is an address.
2. **NO** if only contents (values) are combined → the result is an ordinary number.

```
p[0] - p[2]        // NO: two char VALUES => int
p + p[0] - p[2]    // YES: pointer + int => address
```

RECIPE R9 Explain the meaning of a pointer-operation line

Use this when… A list of lines such as `pi1 = pi2 + i;` / `i = pi1 * pi2;` / `i = pi1 - pi2;` is given and you must “*Erklären Sie die Bedeutung*” (explain what each does / whether it is legal). (FS20 A4, FS16 A4)

1. `pi1 = pi2 + i;` — pointer + int → a new address: “the address in `pi2`, advanced by `i` elements, is stored as a new address in `pi1`.” `i + pi2` is the same (commutative).
2. `i = pi1 - pi2;` — pointer - pointer → a number: the offset between them, in element units (type `ptrdiff_t`); meaningful only inside one array.
3. `i = pi1 * pi2;` — pointer * pointer is **not allowed** → compile error (“invalid operands to binary *”). Multiplying/dividing addresses is meaningless.
4. `i = pi1 + pi2;` — pointer + pointer is **not allowed** → compile error. Adding two addresses is meaningless.
5. Comparisons (`< <= == …`) compare addresses (meaningful within the same array).

Line	Meaning
`pi1 = pi2 + i;`	address + int → new address in `pi1`
`pi1 = i + pi2;`	int + address → new address (commutative)
`i = pi1 * pi2;`	pointer multiplication forbidden → compile error
`i = pi1 - pi2;`	difference of two addresses → offset in element units

RECIPE R10 **sizeof & address differences: 2-D arrays vs. pointer arrays**

Use this when… A 2-D-looking declaration is given and you must evaluate `sizeof(…)` or `&x[i][j]-&x[0][0]` for several rows. First decide whether it is a **real 2-D array** `char d[7][10]` or a **pointer array** `char *p[7]` — the answers differ.

1. **Real 2-D array:** `sizeof` = whole bytes (rows×cols×elem). Element differences are exact (contiguous memory).
2. **Pointer array:** `sizeof` = (#pointers)×pointer-size. The pointed-to literals are scattered → their address differences are undefined.
3. After a decay (e.g. d+1) `sizeof` yields the pointer size, not the row size.

Expression	d[7][10]	*p[7]
`sizeof(.)`	70	56 (7·8)
`sizeof(.+1)`	8 (decays→ptr)	8
`sizeof(.[1])`	10 (`char[10]`)	8 (`char*`)
`.[4]-.[1]`	30 = (4-1)·10	arbitrary*
`&.[2][3]-&.[0][0]`	23 = 2·10+3	arbitrary*

* A pointer array points at scattered literals → the difference is undefined.

9 Strings & char-Arrays

POINTERS & MEMORY

No string type

- There is **no string type**: a string is a `char` array terminated with the NUL character `'\0'`.
- A literal occupies the visible characters **plus** `'\0'`. `"abc"` → `sizeof = 4`; `"SEPFS16"` → `sizeof = 8`.
- Declared with a string literal; the last character is `'\0'`.
- Library functions in `<string.h>` work up to the `'\0'`: length `strlen`, compare `strcmp`, copy `strcpy`, concatenate `strcat`.

```c
char array[] = "Hello World";
// index: 0 1 2 3 4 5 6 7 8 9 10 11
//        H e l l o   W o r l d  \0
char h1[]   = "hello, world";  // 13 bytes incl. \0
char h3[12] = "hello, world";  // \0 does NOT fit -> unterminated!
```

`char array[] = "Hello World";` (12 bytes incl. terminator)

H	e	l	l	o	␣	W	o	r	l	d	\0
0	1	2	3	4	5	6	7	8	9	10	11

index 0…N-1 · `strlen` counts up to (not incl.) the `'\0'`

FIG A string is a char array ending in `'\0'`. `"Hello World"` needs 12 bytes — 11 visible characters plus the terminator.

⚠ **VERIFY** `char h3[12] = "hello, world";` is a known trap: the 12 visible characters fit exactly, leaving no room for the terminating `'\0'`, so the array is **not** a valid C string. Some compilers accept this silently. Verify the exact wording your course uses.

RECIPE R11 Reverse a string (swap characters) in place

Use this when… A function must reverse / invert a char array — "*Buchstaben von hinten nach vorne umtauschen*", "*invertieren*". Signature is typically `void inverse(char * const pc)`. (FS20 A2d, FS16 A2d)

1. **Find the length** by walking to the terminator (you may not use `strlen` if forbidden): `for (i = 0; *(pc + i); i++, len++);` — an empty body; it stops at `'\0'`.
2. **Iterate only to the middle:** `for (i = 0; i < len/2; i++)`. Going to the end would swap everything back.
3. **Swap** with a temp the mirror pair `pc[i] ↔ pc[len-1-i]`.
4. Do **not** touch the terminating `'\0'` — it must stay at the end.

```
void inverse(char * const pc) {  /* pc: const pointer, cannot re-
point */
    int i = 0, len = 0, temp = 0;
    for (i = 0; *(pc + i); i++, len++);    /* 1. measure length
*/
    for (i = 0; i < len / 2; i++)
{        /* 2. up to the middle */
        temp = *(pc + i);                  /* 3. classic 3-step
swap */
        *(pc + i) = *(pc + len - i - 1);
        *(pc + len - i - 1) = temp;
    }
}
```

Grading splits 1P each for: length, iterate-to-middle, temp variable, correct swap — so write all four clearly.

RECIPE R12 Per-word character transform with strtok + SWAPCHAR (Typoglycemia)

Use this when… You must process a text word by word and swap letters by per-word length rules — keywords *Typoglycemia, strtok, SWAPCHAR, Kopie (issue) des Textes*, or "evaluate command-line arguments". (FS16 A5 = copy of a string; FS16 A6 = argv)

1. The macro (given): `#define SWAPCHAR(x,y) ((x)^=(y),(y)^=(x),(x)^=(y))` — XOR-swaps two chars without a temp.
2. The two length rules (read carefully): **rule 1** — `len ≥ 5` → swap letters 2 and 3 (`word[1]↔word[2]`); **rule 2** — `len > 5` → also swap `word[len-2]↔word[len-3]`.
3. **Flavour A** — build a copy (*issue*) of a string variable: `malloc(sizeof(origin)+1)`, check NULL, cast to `char*`, `strcpy(issue,"")` to empty it, then a `strtok(origin," ")` loop: per token apply the rules, `strcat(issue,word)` + `strcat(issue," ");` finally `free(issue)`.
4. **Flavour B** — read from `argv`: guard `if (argc < 2) return EXIT_FAILURE;`, then `argv++;` and `while (*argv) { word = *argv; … printf("%s ",word); argv++; }`.
5. If the question asks for an "alternative control structure" in the second part, implement the rules with a different construct than the first (e.g. a `switch` with fall-through in A vs. an if-chain in B).

```c
/* Flavour B (argv), if-chain version */
int main(int argc, char **argv) {
    if (argc < 2) return EXIT_FAILURE;   /* need at least one
word */
    int len = 0; char *word;
    argv++;                              /* skip program name */
    while (*argv) {
        word = *argv;
        len = strlen(word);
        if (len > 4) SWAPCHAR(word[1],
word[2]);          /* rule 1 */
        if (len > 5) SWAPCHAR(word[len-2],
word[len-3]);  /* rule 2 */
        printf("%s ", word);
        argv++;                          /* next argument */
    }
    return EXIT_SUCCESS;
}
```

⚠ **VERIFY** The prose says “rule 1: len = 5”, but the model code uses `if (len > 4)`, i.e. rule 1 fires for `len ≥ 5` (not only exactly 5). Follow the code. Also: XOR-swap corrupts the value if the two indices alias the same cell — harmless here since the indices always differ.

10 Structures, Enums & typedef

POINTERS & MEMORY

Structures (struct)

- A structure is a new “data type” that groups elements of different types (≈ a Java class with all members public and no methods).
- Can be nested; whole structures are **assignable** (a copy is made).

```c
struct point3D { double x, y, z; };
struct point3D p = {2.0, 4.0, 6.0};
p.x = 5;                    // member access with .
struct point3D q = p;       // copies the whole struct

// typedef saves the 'struct' keyword:
typedef struct { double x, y; } Point2D;
Point2D pt = {2.0, 4.0};
```

struct point3D { double x, y, z; }; — struct p (8+8+8 = 24 bytes, contiguous)

2.0	4.0	6.0
x · @+0	y · @+8	z · @+16

sp = &p · through a pointer use sp->x (== (*sp).x) · on a value use the dot p.x

FIG A struct lays its members out contiguously. Through a pointer use `sp->x` (== `(*sp).x`); on a value use the dot `p.x`.

Enumeration type (enum)

- Defines a list of constant `int` values usable in expressions.
- Values may be set explicitly and need not be unique. Assignment does **not** enforce the range (`enum wochentage w = 77;` works).
- Combined with `typedef` it saves the `enum` keyword.

```c
enum weekday { Monday = 1, Tuesday = 2, Wednesday = 3 };
enum weekday { Monday, Tuesday, Wednesday };      // 0, 1, 2
enum frucht  { Apfel = 5, Birne, Zitrone };       // 5, 6, 7
typedef enum { Monday, Tuesday, Wednesday } weekday;
printf("%i\n", Monday);
```

RECIPE R13 Access & print struct members via a pointer

Use this when… You are given `struct T *ptr;` that points at an initialised record and must print / use its fields — “*Codefragment, das die Koordinaten auf stdout ausdruckt*”. (FS12 A3c)

1. `#include <stdio.h>` for `printf`.
2. Reach a field through a pointer with `ptr->field` — this is exactly `(*ptr).field`. (Use the dot `.` only on a struct value, not a pointer.)
3. Match each field type to its conversion specifier (`int`→`%d`, `double`→`%f`, …).

```c
#include <stdio.h>
(void) printf("Point is %d %d %d\n", pptr->x, pptr->y, pptr->z);
```

Grading: printf+include 1P, format string with conversion chars 1P, struct access 2P.

11 Dynamic Memory (Heap & Stack) POINTERS & MEMORY

Where data lives

- **Code** · **Global/Static** · **Heap** (dynamic, `malloc`) · **Stack** (automatic: local variables + parameter passing).
- Default = automatic (stack). Alternative = dynamic (heap).

high address 0xFFFF…

Stack — locals + params, per call (grows down ↓)
(free space) — stack & heap grow toward each other
Heap — `malloc`/`free` (dynamic) (grows up ↑)
Global / Static — init 0, whole run
Code (text) — executed by CPU

low address 0x0000…

FIG Process memory regions in virtual memory (conventional layout, high address on top). The stack grows down and the heap grows up, toward the free space between them.

⚠ VERIFY Growth direction is implementation-specific. The course diagram shows heap and stack growing toward each other; on most real systems the stack grows downward (toward lower addresses) and the heap grows upward. Don't over-rely on a fixed direction.

Stack — automatic memory

- Allocated automatically; each call reserves space for locals + parameter passing (size known at compile time).
- Each call has its own instances → recursion is possible. Also stored: management info incl. the **return address**.
- The stack's requirement changes continuously (nesting) and it can **overflow**.

Heap — dynamic memory

- `malloc` returns an untyped `void*`, or `NULL` on failure → **always check**.
- Always return memory with `free()`, else a leak. Call `free()` exactly once per block (double-free corrupts the heap).

```c
// stdlib.h
void *malloc(size_t size);             // size bytes, content undefined
void *calloc(size_t n, size_t size);   // n*size bytes, zero-initialised
void *realloc(void *p, size_t size);   // grow/shrink a block
void  free(void *p);                   // release

node_t *node_ptr = malloc(sizeof(node_t));
if (node_ptr == NULL) { /* error handling */ }
free(node_ptr);   // release exactly once per block
```

`malloc` → `void*` (NULL on failure) · `free(p)` releases it exactly once per block

FIG A stack pointer holds the address of a block that `malloc` carved out of the heap. `free(p)` returns that block; do it exactly once.

each node points to the next; the last → NULL

FIG A singly linked list built from heap nodes: each node stores data plus a pointer to the next; the last points to NULL.

RECIPE R14 **Allocate on the heap, copy data in, return the pointer**

Use this when… A function must dynamically allocate space, copy a parameter into it, and return the pointer — keywords *auf dem Heap allozieren, kopieren, Pointer zurückgeben; Standard-Library verwenden*. (FS12 A4 — copyStringToHeap)

1. **Size:** for a string it is `strlen(src) + 1` — the +1 is for the terminating `'\0'`. For n items of type T it is `n*sizeof(T)`.
2. **Allocate & cast:** `T *p = (T*) malloc(size);`
3. **Check:** `if (p == NULL) { /* exit / return NULL */ }`
4. **Copy:** for a string use `strcpy(p, src)` (copies through `'\0'`); for raw bytes use `memcpy(p, src, size)`.
5. **Return p.** The caller is responsible for `free()`.

```c
#include <stdlib.h>
#include <string.h>

char* copyStringToHeap(const char *sourceString) {
    char *stringOnHeap;
    int length = strlen(sourceString);    /* without the '\0' */
    /* malloc length+1 bytes (the +1 is for the terminator): */
    stringOnHeap = (char *) malloc((length + 1) * sizeof(char));
    if (NULL == stringOnHeap) { exit(1); }
    strcpy(stringOnHeap, sourceString);   /* copies incl. '\0' */
    return stringOnHeap;
}
```

⚠ **VERIFY** The FS12 answer key ends with `return strncpy(stringOnHeap, sourceString, length);`. That copies exactly `length` bytes and does **not** append `'\0'` → the result is an unterminated string (a bug). Fix it with `strcpy`, or `strncpy(…, length+1)`, or set `stringOnHeap[length] = '\0';` after copying.

12 Modular Programming & the Toolchain

BUILD & QUALITY

From source code to an executable

- **Preprocessor** — textual substitutions. Commands begin with #: text inclusion (`#include`), text replacement (`#define`), conditional inclusion (`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`). `gcc -E` shows the result.
- **Compiler** — source → object files (`.c→.o`). Does syntax & static type checking, emits errors/warnings. An object file holds machine code + external symbols + debug info and may still contain unresolved calls — not yet executable. One `.o` per module.
- **Linker** — resolves the open calls (functions/variables from other `.o` or libraries), sets addresses, generates the executable.

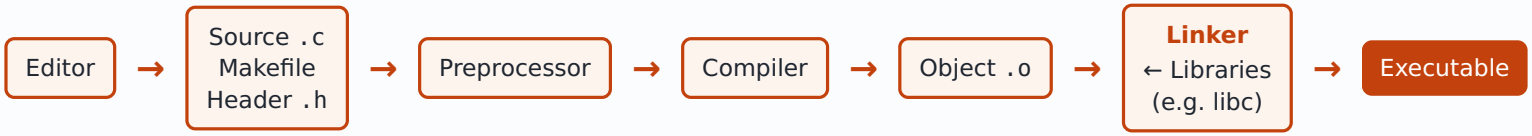


FIG The build pipeline: editor → sources → preprocessor → compiler → object files → linker (pulling in libraries) → executable.

gcc invocations

```
gcc -c hello.c          # compile only -> hello.o
gcc -o hello hello.o    # link -> executable 'hello'
gcc -o hello hello.c    # compile + link directly
gcc -o prog main.c doit.c   # multiple modules
gcc -Idir -o prog main.c    # header in another directory
```

Preprocessor details

- `#include <file>` = system/compiler header; `#include "file"` = your own header.
- `#define MAXLOOP 1000` = text replacement; `#undef DEBUG` removes a constant.
- Conditional compilation: `#if defined DEBUG … #else … #endif`; `#ifdef` is shorthand for `#if defined`.

Modules & headers

- Divide and conquer: split the source into modules; one header (`.h`) per module is the interface.
- A header contains: `#define` constants, function declarations, and user-defined types (`struct`, `enum`). Headers contain **no** function/variable definitions.
- Advantage: constants/declarations exist once; each `.c` does `#include "header.h"`.

Include guard (avoid multiple inclusion)

Problem: a header gets included directly and indirectly → a struct is defined twice → compile error. **Remedy:** set a constant on first inclusion, then check it → each header is processed at most once.

```c
/* Header output.h */
#ifndef OUTPUT_H            // include-guard start
#define OUTPUT_H
#include <stdlib.h>
#include "data.h"           // other headers
void output_dot(data_t data);  // function head (declaration only)
#endif                      // include-guard end
```

NOTE A complete three-file module: a header (interface) guarded against double inclusion, its implementation, and a `main` that uses it. Build with `gcc -o calc main.c add.c`.

```c
/* ---------- add.h : the module interface ---------- */
#ifndef ADD_H              /* include guard: process once */
#define ADD_H
int add(int a, int b);     /* declaration only (no body) */
#endif

/* ---------- add.c : the implementation ------------ */
#include "add.h"           /* check the prototype matches */
int add(int a, int b) {    /* the one definition (ODR) */
    return a + b;
}

/* ---------- main.c : a user of the module --------- */
#include <stdio.h>
#include "add.h"           /* just the declaration */
int main(void) {
    printf("%d\n", add(2, 3));   /* prints 5 */
    return 0;
}
```

Libraries

- The standard library (e.g. `/usr/lib/libc.a`) is precompiled object files; the headers live in `/usr/include`.
- `#include <stdio.h>` inserts declarations/constants; the linker uses the library's object code.
- **Build your own:** `ar -r libown.a doit.o`, then `gcc -Ldir -lown -o prog main.c` (`-L` = directory, `-l` = library).

Useful libraries

Header	Purpose	Header	Purpose
`<stdio.h>`	I/O (printf, scanf, getchar)	`<string.h>`	strlen, strcpy, strcat, strcmp
`<stdint.h>`	exact-size integer types	`<assert.h>`	assert
`<stddef.h>`	ptrdiff_t, size_t, NULL	`<ctype.h>`	isalnum, isdigit, islower
`<stdbool.h>`	bool	`<errno.h>`	errno
`<stdlib.h>`	malloc, free, atoi, abs, rand	`<limits.h>` / `<float.h>`	INT_MAX... / DBL_MAX...
`<math.h>`	log, pow, sqrt, tan	`<stdarg.h>`	va_list, va_start, va_arg, va_end
`<signal.h>`	signals	(also)	`<setjmp.h>`, `<locale.h>`, `<time.h>`

13 make & Makefiles

BUILD & QUALITY

Purpose & mechanism

- make builds programs from several modules **incrementally** — only out-of-date parts are regenerated.
- An object is **out of date** when at least one component has a newer date than the object.
- On make, the rules of the Makefile in the current directory are processed recursively.

Makefile structure & rules

- Line-oriented: comments (#), variable definitions, explicit rules.
- A rule = `target: dependencies` + a <TAB> command. Commands run whenever a dependency has a newer date than the target.
- Before each command there must be a real **TAB** (not spaces).

```
rechner: add.o sub.o main.o
        gcc -o rechner add.o sub.o main.o
add.o: add.c def.h Makefile
        gcc -c add.c
clean:
        rm -f *.o rechner
```

Variables & line continuation

- `var = value` — recursively expanded (re-expanded on every use).
- `var := value` — simply expanded (evaluated once).
- Line break via continuation: \ as the last character of a line.

```
TARGET  = my-program
OBJECTS = main.o view.o ctrl.o
$(TARGET): $(OBJECTS)
        gcc -o $(TARGET) $(OBJECTS)
```

.PHONY, automatic variables & patterns

- The first target runs when nothing is specified; common ones: `all`, `default`, `clean`, `test`.
- `.PHONY: clean` → not interpreted as a file; the commands always run even if a file named `clean` exists.
- **Automatic variables:** `$@` = target · `$^` = all dependencies · `$<` = first dependency.
- **Substitution:** `$(VAR:%.c=%.o)` = replace a trailing `.c` with `.o` in each word.

```
.PHONY: default all clean test
OBJECTS = $(SOURCES:%.c=%.o)
my-program: main.o view.o ctrl.o
        gcc -o $@ $^
%.png: %.dot          # pattern / suffix rule
        dot -Tpng $< >$@
```

Errors, echo & options

- **Ignore an error:** `-` before the command (e.g. `-rm -f $(OBJECTS)`) or call `make -k`.
- **Suppress echo** of the command: `@` before it (e.g. `@echo "done"`).
- **Dry run:** `make -n` (shows what would happen, changes nothing). List rules/variables: `make -p` (e.g. `make -n -p | grep CFLAGS`).

> **NOTE** A complete, idiomatic Makefile for a small 3-module program, using variables, automatic variables, a pattern rule, and phony targets — the kind you may have to write by hand.

```
# ---- a complete small Makefile ----
CC      = gcc
CFLAGS  = -Wall -Wextra -std=c11   # strict warnings
TARGET  = calc
OBJECTS = main.o add.o sub.o

.PHONY: all clean          # not real files

all: $(TARGET)             # default target

$(TARGET): $(OBJECTS)      # link step
        $(CC) -o $@ $^          # $@=target $^=all deps

%.o: %.c                   # pattern: any .c -> .o
        $(CC) $(CFLAGS) -c $<   # $<=first dependency

clean:
        rm -f $(OBJECTS) $(TARGET)
```

14 Code Quality & Testing

BUILD & QUALITY

C behavior categories

- **Unspecified:** the standard allows several variants but does not choose one (e.g. order of sub-expression evaluation).
- **Undefined:** the standard makes no statement — anything is allowed, even a crash (e.g. modifying a string literal).
- **Implementation-defined:** the compiler decides and documents (e.g. whether `char` is signed or unsigned).
- **Goal:** avoid undefined and unspecified constructs.

```c
int x = 5;
printf("%d %d", x++, x++);  // 5 5? 5 6? 6 5? -- unspecified
char *p = "Beispiel Text";
p[0] = 'W';                 // undefined behaviour
```

Ensuring code quality

- **Design:** a document review checks that all requirements / architecture constraints are covered.
- **Implementation:** apply coding guidelines (an accepted subset of the language + usage rules); do code review (manual and/or tool); switch on compiler warnings and eliminate them.
- **Test:** execution + code coverage on several levels.
- Quality is a trade-off against cost; ask at every level “when is it good enough?” (remote control vs. cardiac pacemaker).

Tool support & questionable constructs

- The C compiler emits many warnings; switch them on as strictly as possible: `-Wall -Wextra -Werror`.
- **Lint** (historically separate, allowed to be slow) is still useful, e.g. against industry standards such as MISRA.

Construct	Category & warnings
`c = (i++) + a[i];`	Unspecified evaluation order. Compiler `-Wsequence-point`; Lint 564; MISRA 12.x.
`addr = ntohl(*((uint32_t*)msg_start));`	Undefined — invalid pointer. Lint 826: suspicious pointer conversion (area too small).
`while (x = 1) { … }`	Pitfall — = instead of ==. Compiler `-Wparentheses`; Lint 720; MISRA 13.1.
`unsigned a=1; int b=-1; if (a > b)`	signed/unsigned mixed: b becomes huge unsigned, block never runs. `-Wsign-compare`; Lint 737.

Testing

- **Test hierarchy** (IEEE / V-model): System test → Integration test → Component/Unit test. Orthogonal to agile (each iteration may touch all levels).
- Proving correctness is impossible; testing serves to **find errors** — the grossest ones as early as possible.
- **Anatomy of a test:** Stimulus → UUT → Actual Response, compared against the Expected Response → Pass/Fail. Needs both **controllability** (set inputs) and **observability** (observe outputs); a complete test needs a stimulus AND an expected behaviour.

```c
#include <CUnit/Basic.h>
static int setup(void) {
    remove_file_if_exists(OUTFILE); return 0;  // success
}
static int teardown(void) { return 0; }
static void test_person_compare(void) {
    CU_ASSERT_TRUE(person_compare(&b, &a) > 0);
    CU_ASSERT_PTR_NOT_EQUAL(anchor, anchor->next->next->next);
    CU_ASSERT_PTR_EQUAL(anchor, anchor->next->next->next->next);
}
```

NOTE A self-contained assert-based test, the simplest test harness when no framework is available. `assert` aborts (via SIGABRT) and prints the file/line if the condition is false; compile with `-DNDEBUG` to disable all asserts in a release.

```c
#include <assert.h>
#include <stdio.h>
int add(int a, int b) { return a + b; }   /* unit under test */
int main(void) {
    /* stimulus -> expected response, checked with assert */
    assert(add(2, 3) == 5);    /* normal case */
    assert(add(-1, 1) == 0);   /* boundary */
    assert(add(0, 0) == 0);    /* zero */
    printf("all tests passed\n");
    return 0;
}
```

⚠ **VERIFY** Evaluation order inside one expression — e.g. `printf("%d %d", x++, x++)` or `c = (i++) + a[i];` — is **unspecified**. Don't predict a single answer; this is the category R1 flags for trace-the-output traps.

15 System Calls & System Libraries

KERNEL & MEMORY

Isolation, user mode & kernel mode

- **Isolation:** applications and the OS each have “private” memory.
- **Kernel/Supervisor mode:** all instructions allowed (full hardware access). **User mode:** restricted (no direct hardware access, no MPU/ISR reconfig, no direct switch to supervisor mode).
- Mode switch via a synchronous interrupt / trap → into the ISR in supervisor mode; on return, control passes back in a controlled way.
- At startup the system runs in supervisor mode until the OS starts the first user task. **Mode** = privilege separation; **Space** = memory-region protection.

System calls

- The fundamental API: a call from a user task into the kernel; it encapsulates the mode switch.
- **Anatomy:** each syscall has a number; number + arguments go into CPU registers → the OS function is invoked. CPU-architecture- and OS-specific (its own ABI); Linux has ~300 syscalls.
- Usually called via wrapper functions in the system library (glibc), not directly.

```
long syscall(long sys_no, ...) {
    long res = MODE_SWITCH(sys_no, ARGS(...));
    if (res < 0) { errno = -res; res = -1; }  // -1 + errno on error
    return res;
}
```

- On error `syscall()` returns `-1` and sets `errno`; otherwise 0 or positive. `errno` is a user-land variable (the kernel does not know it); on success it is not modified. Some wrappers return e.g. a `NULL` pointer on error.

NOTE A complete program that triggers an error, then reads `errno` and prints it two ways. Always check the return value first, then read `errno` (a later successful call may overwrite it).

```
#include <stdio.h>
#include <errno.h>    // errno
#include <string.h>   // strerror
int main(void) {
    FILE *f = fopen("/no/such/file", "r");
    if (f == NULL) {              // 1. check the result
        int e = errno;            // 2. snapshot errno
        perror("fopen");          // "fopen: No such file ..."
        printf("errno=%d: %s\n", e, strerror(e));
        return 1;
    }
    fclose(f);
    return 0;
}
```

System library & standards

- The system library abstracts system/architecture dependencies (syscall wrappers, processes, threads, file access, I/O, error handling). In SNP it is the boundary to the kernel.
- **Standard C library** — part of the C standard, OS-independent. **POSIX** (IEEE 1003.1) — the C API to Unix-like OSes. **FHS** — defines the filesystem hierarchy (`/bin`, `/dev`, `/etc`, ...).
- **Compatibility kinds:** API (source level) · ABI (binary, same architecture) · filesystem (files in the same places).
- On Linux: **glibc** = C Standard Library + C POSIX Library + GNU extensions + syscall wrappers + CLI-argument parsing.

C revisions & standard options

C revisions: K&R → ANSI-C/C89/C90 → C99 → C11. `gets()` is gone from C11 → use `fgets(line, LSIZE, stdin)`.

Standard	Option	Standard	Option
C90	`-std=c90 -pedantic`	C11	`-std=c11 -pedantic`
C99	`-std=c99 -pedantic`	Default	`-std=gnu11`

16 Memory, Isolation, MMU/MPU

KERNEL & MEMORY

Isolation & virtual memory

- **Isolation of applications:** apps must not affect the OS or each other → private virtual memory. Mechanisms: user/kernel-mode transition via syscalls + virtual memory & memory management.

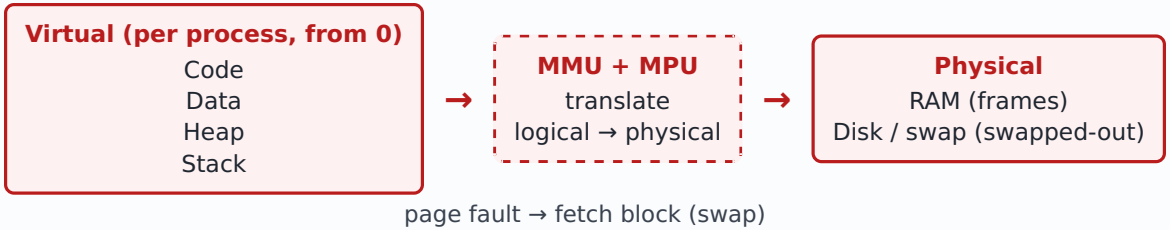


FIG Each process sees its own virtual address space (Code, Data, Heap, Stack) from 0. The MMU (with MPU) translates virtual → physical; a page fault fetches a swapped-out block from disk.

Virtual memory

- All processes have the same virtual range (logically from address 0, but not physically). Backed by physical memory; data resides in RAM or is swapped out.
- Address translation via a block list; a CPU exception signals that a block must be fetched → swap.

MMU & MPU

- **MMU** (Memory Management Unit): translates logical → physical addresses; extends physical memory with swapped-out blocks (illusion of large memory); also contains the MPU functionality.
- **MPU** (Memory Protection Unit): monitors the address bus for illegal accesses → raises an exception. Configurable only in kernel mode. **Free:** your own user data; **protected:** kernel data, other hardware.
- Simple embedded CPUs sometimes have only an MPU and no MMU.

Process memory & virtualization

- **Process memory regions:** Code (executed by the CPU), Global data, Stack (locals during execution), Heap (OS-allocated dynamic regions) — all in virtual memory.
- **Control virtualization:** illusion of sole CPU control; the OS switches via context switch. **Memory virtualization:** illusion of one linear region; the MMU maps virtual → physical (or swapped).

FHS — Filesystem Hierarchy Standard

Path	Contents	Path	Contents
`/bin`	basic commands (all users)	`/opt`	additional software packages
`/boot`	bootloader files	`/root`	root's home
`/dev`	device files	`/run`	data of running processes
`/etc`	configuration files	`/sbin`	important system commands
`/home`	user directories	`/srv`	data of services
`/lib`	kernel modules, dynamic libs	`/tmp`	temporary files
`/media`	mount: removable media	`/usr`	Unix System Resources (bin, lib...)
`/mnt`	temporary mount points	`/var`	cache, log, spool, ...

NOTE Linux startup: BIOS (ROM; tests hardware, loads bootloader) → 1st-stage bootloader (MBR, 446 B) → 2nd-stage bootloader → Linux kernel (inits hardware, scheduler, init process) → init/systemd → user interaction (GUI/login → bash).

17 Filesystem & I/O

KERNEL & MEMORY

“Everything is a file”

- Many interactions happen via reading/writing files. Access pattern: **open → read/write → close**.
- **File descriptor (fd):** an integer ID with which the kernel manages an open file; valid as long as the file is open.

Regular files

- A contiguous, unstructured array of bytes (a byte stream). Can be opened multiple times; the OS provides no synchronization.
- **File position:** start point of each operation; kernel-managed, advanced on each access; 0 on open; settable to a non-negative value.
- Length in bytes; changeable (truncate shorter → rest lost; lengthen → filled with zeros). Writing past the end fills with zeros; reading past the end yields EOF.
- **Shared access:** multiple opens (each its own fd) or a shared fd; the kernel does not synchronize → the program is responsible.

Inode, file name, directories & links

- The **inode** is the management unit (metadata): unique i-number (ino), timestamps, owner, length, physical location — but **not** the file name.
- A directory is a file whose content is a map of name→ino pairs. A name→ino pair = a **hard link**; several links can point to one ino. The inode holds a link counter; the file is removed only when it reaches 0. Directories are modified via link/unlink.
- **Symbolic/soft link:** a special file whose content is a path; FS-spanning, can point at non-existent targets. A hard link is limited to one filesystem.

```
ln file lnk      # another directory entry (hard link)
ln -s file sym   # symbolic link 'sym' pointing at 'file'
ls -l            # shows type + hard-link counter
```

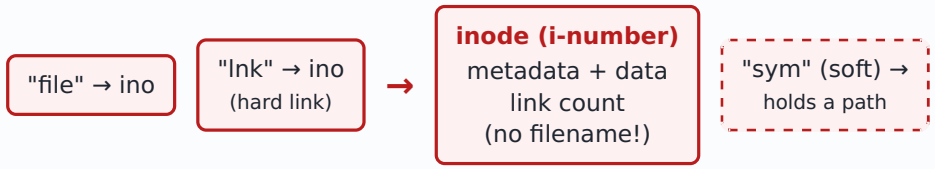


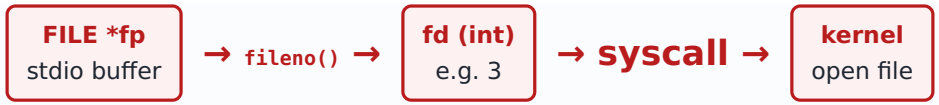
FIG Two hard links are two names for the same inode (they share its i-number and bump its link count). A soft link is a separate file that merely stores a path, so it can cross filesystems and dangle.

Paths, special files & mounting

- A path is names separated by `/`; leading `/` = root (absolute). A process has a **working directory** (set with `cd`, query with `pwd`).
- **Special files** (under `/dev`): Character Devices (byte sequence: keyboard, mouse), Block Devices (byte arrays: mass storage), Named Pipes & Sockets (IPC).
- Other filesystems are attached via `mount` / `umount` onto an empty mount-point dir; `/etc/fstab` records mounts; `findmnt` lists them.
- **Physical FS:** on block devices, access unit = sector (usually 512 B). **Virtual FS** (in memory): `/proc`, `/sys`, `/dev`.

Streams: FILE* vs. fd

- A **stream** (`<stdio.h>`) is a logical byte sequence — an abstraction even for non-Unix systems.
- **fd** = an int, kernel-managed (system calls). **FILE*** = a stdio wrapper (buffer, position, status hidden) — do not dereference it. `fileno(FILE*)` yields the fd; mixing FILE* and fd accesses is unpredictable.



0 = stdin, 1 = stdout, 2 = stderr · never dereference a FILE* · don't mix fp & fd

FIG Three layers: a stdio FILE* (buffered) sits on top of a kernel file descriptor (a small int) which refers to the kernel's open file. `fileno()` bridges FILE* → fd; don't mix the two.

stdio.h vs. system-call I/O

stdio.h purpose	Functions
Open / close	fopen, freopen, tmpfile, fclose, fflush, remove, rename
Position / status	ftell, fgetpos, fseek, rewind, fsetpos, feof, ferror, clearerr, setvbuf
Read / write (unformatted)	fread, fwrite, fgetc, getc, getchar, ungetc, fgets, fputc, putc, fputs, puts
Formatted	fscanf, scanf, sscanf, fprintf, printf, snprintf
Error	perror

System-call I/O	Functions
Open / close / delete	open, creat, close, link, unlink
Status / position	stat, fstat, lseek
Read / write	read, write, recv, send
Attributes / IPC / other	chown, chmod, umask; pipe, socket, bind, listen, accept, connect; mount, mknod, fcntl, mmap, dup

Buffering & blocking

- **Buffering:** unbuffered (direct), fully-buffered (sent when full), line-buffered (sent at end of line). `stdin`/`stdout` are fully-buffered (line-buffered on a terminal); `stderr` is unbuffered on Linux. fd constants in <unistd.h>: `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`.
- **Blocking** (e.g. `getchar()` waits) vs. **non-blocking** (returns -1, `errno=EAGAIN`). Seekable (regular files) vs. non-seekable (serial). Check success after every I/O access (`perror`).

NOTE A complete program that copies one file to another two ways: with low-level descriptors (`open`/`read`/`write`/`close`) and with stdio streams (`fopen`/`fgets`/`fputs`/`fclose`). Both check every call.

```c
#include <stdio.h>
#include <fcntl.h>    // open, O_* flags
#include <unistd.h>   // read, write, close
int main(void) {
    /* ---- low level: file descriptors ---- */
    int in  = open("src.txt", O_RDONLY);
    int out = open("dst.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (in < 0 || out < 0) { perror("open"); return 1; }
    char buf[4096]; ssize_t n;
    while ((n = read(in, buf, sizeof buf)) > 0)
        write(out, buf, (size_t)n);   // (check the return in real code)
    close(in); close(out);

    /* ---- high level: stdio streams ---- */
    FILE *fin  = fopen("dst.txt", "r");
    FILE *fout = fopen("copy.txt", "w");
    if (!fin || !fout) { perror("fopen"); return 1; }
    char line[256];
    while (fgets(line, sizeof line, fin))   // safe; never gets()
        fputs(line, fout);
    fclose(fin); fclose(fout);
    return 0;
}
```

18 Tasks, Processes & Threads

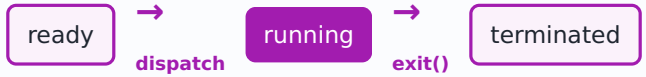
PROCESSES & IPC

Multi-tasking & scheduling

- **Task** = a job for the CPU. **Batch:** tasks run one after another. **Multi-tasking:** tasks run side by side; too few CPUs → the system switches → quasi-parallel (true parallelism is limited by cores).
- **Cooperative scheduling:** each task decides when to yield. **Preemptive:** handover is enforced. The scheduler interrupts tasks and picks the next (round-robin, priority-driven). Goal: no task blocked forever; illusion of sole control.
- **Context switch:** the CPU resumes a previously interrupted task. Saved/restored: program counter, stack pointer, registers, MMU state. For processes the virtual memory must be reconfigured (expensive).

Process vs. thread

- **Process:** one control flow / stack, its own virtual memory (heavyweight). **Thread:** a separate control flow / stack within a process, sharing memory (lightweight) → cheap switch, but no mutual protection.
- **Process states:** running, ready, blocked, terminated. Per process: a process control block (state, context info, resource accesses).
- On Linux threads are processes with shared resources; user-land POSIX threads (pthreads / NPTL, part of glibc).



↑ I/O done → ready · preempt / yield: running → ready · wait for I/O: running → **blocked** → (I/O done) ready

FIG The four process states and the transitions between them: the scheduler dispatches ready→running and preempts running→ready; I/O moves running→blocked and back; `exit()` terminates.

Lifecycle — process API (fork/wait)

Call	Meaning
`fork()`	Duplicates the parent; both continue after the call. Child inherits resources. Returns: `-1` error; `0` in the child; `>0` in the parent (= child's PID).
`wait()`	Waits until a child terminates.
`waitpid()`	Takes a child's exit code (blocking or non-blocking).
`WEXITSTATUS()`	Extracts the exit code from the status returned by `wait()`.
`exit()`	Terminates the process, passes the exit code to `wait…()`.
`exec…()` (e.g. `execv`)	Replaces the current image with a new program; normally does not return (only on a load error).
`system("cmd")`	fork + shell + exec + waitpid in one.
`popen()`/ `pclose()`	Like `system`, but stdin/stdout readable via a pipe (FILE*).

⚠ VERIFY Some source slides said “execv runs a program in a new thread” and “fork returns 1 = Parent”. More precisely: `exec…()` **replaces** the current process image (no new thread), and `fork()` returns the child's PID (a positive number, not literally 1) in the parent.

NOTE A complete fork/exec/wait program: the parent forks, the child replaces itself with `/bin/echo` via `execlp`, and the parent waits and reads the exit code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>      // fork, execlp
#include <sys/wait.h>    // wait, WEXITSTATUS
int main(void) {
    pid_t pid = fork();              // duplicate this process
    if (pid < 0) { perror("fork"); return 1; }
    if (pid == 0) {                  // ---- child (0) ----
        execlp("echo", "echo", "hello from child", NULL);
        perror("execlp");           // only reached on failure
        _exit(127);
    }
    int status;                      // ---- parent (>0) ----
    waitpid(pid, &status, 0);        // wait for the child
    if (WIFEXITED(status))
        printf("child exit code = %d\n", WEXITSTATUS(status));
    return 0;
}
```

Thread API (pthreads) & special situations

- `pthread_create` (start a thread, returns 0 / a positive error code), `pthread_join` (wait + take exit code + free resources), `pthread_detach` (free on termination, no join after), `pthread_exit`, `pthread_cancel`. A missing join/detach → resources never freed.
- **Zombie:** a child terminates before the parent's `wait()` → it stays as a management structure until `wait()`. **Orphan:** the parent terminates first → the child is adopted by init.
- `exit()` in the main thread ends the whole process incl. all threads. FDs are inherited on `fork()` (close unwanted ones or use `O_CLOEXEC`). Multi-threaded + `fork()` → call `exec…()` immediately in the child.
- **Status:** `WIFEXITED`/`WEXITSTATUS` (exit code), `WIFSIGNALED`/`WTERMSIG` (signal).

NOTE A complete pthreads program: start two threads, pass each an argument, join both. Compile with `gcc prog.c -pthread`.

```c
#include <stdio.h>
#include <pthread.h>
void *worker(void *arg) {            // thread entry point
    int id = *(int *)arg;
    printf("thread %d running\n", id);
    return NULL;                     // == pthread_exit(NULL)
}
int main(void) {
    pthread_t t[2];
    int ids[2] = {1, 2};
    for (int i = 0; i < 2; i++)
        pthread_create(&t[i], NULL, worker, &ids[i]);
    for (int i = 0; i < 2; i++)
        pthread_join(t[i], NULL);    // wait + free resources
    printf("both threads done\n");
    return 0;
}
```

RECIPE R15 Count / draw a fork() process tree

Use this when… Code with several `fork()` calls is shown and you must draw the process tree or count how many processes exist. Assume every `fork()` succeeds. (FS20 A5)

1. **n unconditional forks** in a row → 2^n processes (each fork doubles the count). 1 fork=2, 2=4, 3=8, 4=16, …
2. Work level by level: after each `fork()` every existing process has produced one child → the population doubles.
3. A **conditional** fork (e.g. `if (pid == 0) fork();`) only multiplies the processes that satisfy the condition — here only the children of the previous fork (`pid==0`), so add one child per such process.
4. **Drawing:** root = the original process; each fork adds one child edge to every node alive at that point; label the conditional branch separately.

```c
int main(void) {
    fork(); fork(); fork();    /* 3 forks -> 2^3 = 8 */
    pid_t pid = fork();        /* 4th fork -> 16 */
    if (pid == 0) { fork(); }  /* only the 8 children fork */
    exit(0);                   /* -> +8 => 24 total */
}
```

Sanity check. 3 forks → 8; the 4th → 16 (8 old parents + 8 new children); the 8 children each fork once → +8 = 24.

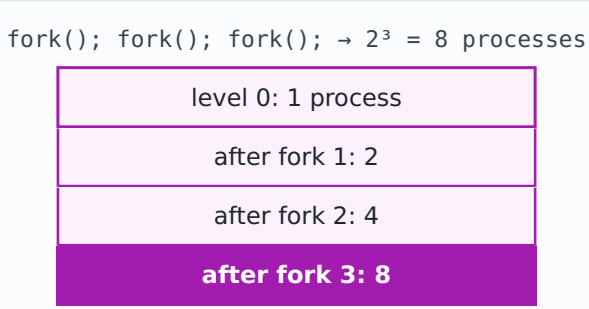


FIG Three unconditional forks double the population at each level: 1 → 2 → 4 → 8 processes. Each node alive at a fork gains one child edge.

19 Thread Synchronization

PROCESSES & IPC

Basics

- Dependent tasks access shared resources or must wait for one another. A **race condition** = behaviour depends on order/timing → non-deterministic. Synchronization avoids races; whoever must wait goes into a **queue**.
- Two kinds: **Locking** = temporally exclusive access (checkout occupied / free) → **Mutex**. **Signaling** = waiting for one another in the flow (wait until the water boils) → **Semaphore**.
- **Critical section (CS):** a sequence with exclusive access to shared data. **Entry** (others wait), **CS** (exclusive), **Exit** (release). **Mutual exclusion:** one task in the CS excludes all others.

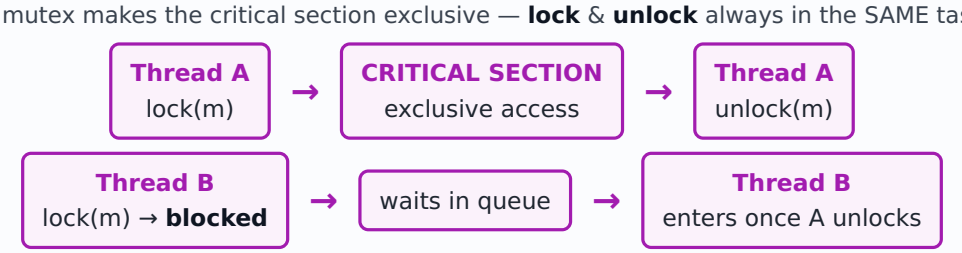


FIG A mutex makes the critical section exclusive: thread A locks and enters; thread B must wait until A unlocks. Lock and unlock always happen in the same task. `lock → enter`; `unlock → release`.

Mutex & Semaphore

- **Mutex:** lock–unlock pair, always in the same task. Lock: free → enter & lock; locked → queue. **Problems:** missing unlock on early return, lock/unlock across tasks, self-deadlock on recursion (unless explicitly recursive).
- **Semaphore:** a counter (a traffic light). Init `0` → blocking from the start; `>0` → that many may pass. **Wait/Down/P:** 0 → block; else `count--`. **Signal/Up/V:** waiter present → wake it; else `count++`. **Binary** (0/1) vs. **counting**.

```c
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);          // Entry
a = value; a += delta; value = a;    // Critical Section
pthread_mutex_unlock(&mutex);        // Exit
```

```c
sem_t sem;
sem_init(&sem, 0, 0);   // init, count = 0
sem_wait(&sem);         // down / P
sem_post(&sem);         // up / V
```

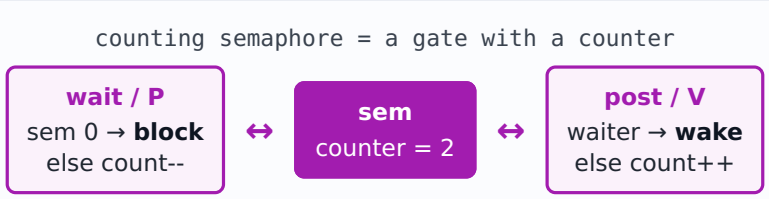


FIG A semaphore is a counter with two operations: `wait/P` blocks when the count is 0 (else decrements), `post/V` wakes a waiter (else increments). Init `0` ⇒ it blocks from the start. Signaling / counting works across tasks.

	Mutex	Semaphore
Purpose	locking (mutual exclusion)	signaling / counting
Operations	lock / unlock	wait / post
Task	only the same task	distributed across tasks
Recursion	possible (explicit)	not possible

Further primitives: Monitor (mutex + condition variable, synchronization built in), **Barrier** (blocks N tasks until all are present; `pthread_barrier_t`), synchronized containers (POSIX message queues).

NOTE Added — a complete two-thread counter. Without the mutex the ++ (read–modify–write) races and the total is wrong; with it the result is always exactly `200000`.

```c
#include <stdio.h>
#include <pthread.h>

static long counter = 0;
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void *bump(void *arg) {
    (void)arg;
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&m);     // Entry
        counter++;                  // Critical Section
        pthread_mutex_unlock(&m);   // Exit
    }
    return NULL;
}

int main(void) {
    pthread_t a, b;
    pthread_create(&a, NULL, bump, NULL);
    pthread_create(&b, NULL, bump, NULL);
    pthread_join(a, NULL); pthread_join(b, NULL);
    printf("counter = %ld\n", counter);   // 200000
    return 0;
}
```

Deadlock & relatives

- **Deadlock** = mutual blockade; all **four conditions** hold: (1) mutual exclusion / in a CS, (2) hold-and-wait, (3) no preemption, (4) circular wait.

  Prevent: take all locks at once, a global fixed lock order, release on failure. Detect & resolve needs OS support; the typical OS stance is to **ignore** it.
- **Livelock** (busy avoiding each other) → solve by design. **Starvation** (never gets a turn) & **priority inversion** (low-prio blocks high-prio → priority inheritance) → solve via the OS.

RECIPE R16 Decide whether a situation is a deadlock (resource-allocation reasoning)

Use this when… A scenario lists processes that hold some resources and want others, and asks “Is this a deadlock? Justify.” (resources allocated exclusively, “want to write” = “request”). (FS20 A7)

1. **Build the graph.** A *held* edge points resource→process (“owned by”); a *request* edge points process→resource (“wants”).
2. **Look for a cycle** that runs through exclusively-held, single-instance resources. A cycle = a deadlock; no cycle = no deadlock.
3. **Justify with the four conditions:** mutual exclusion (exclusive allocation), hold-and-wait (holds one, wants another), no preemption, circular wait (the cycle you found).
4. **Name the parties** in the cycle explicitly.

Worked example. P1 holds T2 and wants D2; P3 holds D2 and wants T2. Cycle: P1 → (wants) D2 → (held by) P3 → (wants) T2 → (held by) P1. All four conditions hold → **deadlock between P1 and P3** (P2, which only holds T1 and wants D2, is not in the cycle).

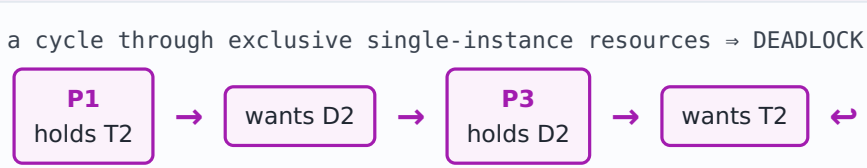


FIG A resource-allocation cycle: P1 wants D2 (held by P3) and P3 wants T2 (held by P1). The closed cycle through exclusive single-instance resources is the deadlock.

RECIPE R17 Use semaphores to enforce a fixed execution order / rendezvous

Use this when… You must write pseudocode with `up(S)`/`down(S)` so that processes run in a prescribed order / pattern (e.g. P0 & P1 in parallel, then P2, repeating) and state the initial values. (FS20 A8)

1. **Initialise:** set to 1 (or k) the semaphores guarding tasks that may start immediately; set to `0` those that must wait.
2. **"B after A":** A does `up(S)` at its end, B does `down(S)` at its start, with S initially `0`.
3. **Barrier (wait for N):** the waiter does `down(S)` N times after the N predecessors each do `up(S)` once.
4. **To loop the pattern:** have the last stage `up()` the semaphores that release the first stages again.

```
Sem s0 = 1; Sem s1 = 1; Sem s2 = 0;   // P0,P1 may start; P2
waits

P0: while (1) { down(s0); working(0); up(s2); }   // signal P2
P1: while (1) { down(s1); working(1); up(s2); }   // signal P2
P2: while (1) {
        down(s2); down(s2);   // wait for BOTH P0 and P1
        working(2);
        up(s0); up(s1);       // release P0 and P1 again
    }
```

⚠ **VERIFY** The exam text mentions semaphores "S0, S1, S3" but the model solution actually uses **S2** (not S3). Read it as S0 / S1 / S2.

20 Inter-Process Communication (IPC) PROCESSES & IPC

Fundamentals

- **IPC** = the kernel's ability to exchange notifications and data between parallel processes. Two basic needs: **events** (report a state change) and **data exchange**.
- **Exchange kinds:** unstructured / byte-based (pipe, socket, shared memory, shared file) vs. **structured** (message queue). **Synchronization:** implicit (message queue, pipe, socket) vs. explicit (shared memory, shared file — secure it yourself).
- **Under Linux:** events → POSIX signals; implicitly synced → pipe, message queue, socket; explicitly synced → shared memory (+ semaphore), shared file (+ file locking).

POSIX signals (<signal.h>)

- A process sends another a **signal** = a number with system-wide meaning; the kernel notifies the target at the next opportunity. **Asynchronous**, predefined numbers, **no payload**.
- Per signal an **action**: **Default** (e.g. terminate), **Ignore**, or a **Handler** (your function). Inside a handler only signal-safe operations.

Signal	Action	Meaning
SIGINT	Term	Interrupt from keyboard (Ctrl-C)
SIGQUIT	Core	Quit from keyboard (Ctrl-\)
SIGABRT	Core	Abort via `abort()` / `assert()`
SIGKILL	Term	Kill — cannot be caught / changed
SIGCONT	Cont	reactivates the process
SIGSEGV	Core	Illegal memory access
SIGALRM	Term	Timer via `alarm()`
SIGTERM	Term	Termination signal
SIGSTOP	Stop	Stops a process — cannot be changed

Sending / pausing: `kill(pid, sig)`, `raise(sig)` = `kill(getpid(), sig)`, `pause()`. `sigaction()` registers a handler (safer than the legacy `signal()`); `sigfillset()` blocks signals during the handler. Handlers: `SIG_IGN`, `SIG_DFL`, or your own. **SIGKILL & SIGSTOP can't be changed**; settings are inherited on `fork()`.

NOTE Set a custom handler:

```c
struct sigaction a = {0};
a.sa_flags = SA_SIGINFO;
a.sa_sigaction = handler;
sigfillset(&a.sa_mask);
sigaction(sig, &a, NULL);
```

NOTE Restore the default action (left) / ignore a signal (right):

```c
/* set the default action */
struct sigaction d = {0};
d.sa_flags = 0;
d.sa_handler = SIG_DFL;
sigfillset(&d.sa_mask);
sigaction(sig, &d, NULL);
```

```c
/* ignore a signal */
struct sigaction g = {0};
g.sa_flags = 0;
g.sa_handler = SIG_IGN;
sigfillset(&g.sa_mask);
sigaction(sig, &g, NULL);
```

RECIPE R18 **Install a signal handler / ignore a signal (parent vs. child after fork)**

Use this when… You must complete code so that on a signal (e.g. Ctrl-C = SIGINT) the parent reacts one way and the child another (e.g. parent prints text, child ignores it). (FS20 A6)

1. **Branch per process:** after `fork()`, set the action inside the matching branch — parent in `if (cpid > 0)`, child in `else` — before the work loop.
2. **Custom handler:** `struct sigaction a = {0}; a.sa_flags = SA_SIGINFO; a.sa_sigaction = handler;` then `sigfillset(&a.sa_mask); sigaction(SIGINT, &a, NULL);`
3. **To IGNORE a signal:** use `a.sa_handler = SIG_IGN;` with `a.sa_flags = 0;` (do *not* set `SA_SIGINFO`). The default is `SIG_DFL`.
4. **Check both calls** (`sigfillset`, `sigaction`) for `-1`.

```c
static void handler(int sig, siginfo_t *si, void *ctx) {
    write(STDOUT_FILENO, "User Interrupt\n",
15);   /* async-safe */
}

/* in the PARENT branch (cpid > 0): install the handler */
struct sigaction a = {0};
a.sa_flags = SA_SIGINFO;
a.sa_sigaction = handler;
sigfillset(&a.sa_mask);
sigaction(SIGINT, &a, NULL);

/* in the CHILD branch (else): ignore it */
struct sigaction b = {0};
b.sa_flags = 0;
b.sa_handler = SIG_IGN;     /* SIG_IGN goes in sa_handler! */
sigfillset(&b.sa_mask);
sigaction(SIGINT, &b, NULL);
```

⚠ **VERIFY** The FS20 answer key writes `a.sa_sigaction = SIG_IGN;` to ignore the signal — that is **wrong**: `SIG_IGN` belongs in `sa_handler`, and `SA_SIGINFO` must be cleared (otherwise `sa_sigaction` is used). Also `printf` in a handler is not async-signal-safe — strictly use `write(STDOUT_FILENO, …)`; most courses tolerate `printf`.

Pipe, message queue, socket, shared memory / file

- **Pipe:** a FIFO byte buffer, **unidirectional**, one fd per end, implicitly synced. **Anonymous:** `pipe(fd[2])` + `fork()`, close the unused end crosswise. **Named (FIFO):** `mkfifo(path, mode)`, normal file I/O.
- **Message queue:** structured (N×M), priorities (high number first, FIFO on equality), multiple R/W. API `mq_open` (name starts with "/", under `/dev/mqueue`), `mq_send` / `mq_receive`, link option `-lrt`.
- **Socket:** IP address + port; datagrams (UDP) or streams (TCP). On one machine: Unix domain sockets. Server: `socket→bind→listen→accept`; client: `socket→connect`.
- **Shared memory:** virtual memory shared between processes; synchronize **explicitly** (semaphores); very fast. **Shared files:** the kernel does not synchronize → use semaphores or file locks (`fcntl`); persistent.

FIG An anonymous pipe after `fork`: the parent writes into `fd[1]`, the child reads from `fd[0]`; each closes the end it does not use. The FIFO buffer synchronizes them implicitly (the reader blocks until data arrives).

NOTE Added — a complete anonymous-pipe program: create the pipe, fork, the parent writes a message, the child reads and prints it. Each side closes the end it doesn't use.

```c
#include <stdio.h>
#include <unistd.h>   // pipe, fork, read, write, close
#include <string.h>

int main(void) {
    int fd[2];   // fd[0]=read, fd[1]=write
    if (pipe(fd) < 0) { perror("pipe"); return 1; }
    pid_t pid = fork();
    if (pid < 0) { perror("fork"); return 1; }
    if (pid > 0) {            // ---- parent writes ----
        close(fd[0]);         // close unused read end
        const char *msg = "hello via pipe";
        write(fd[1], msg, strlen(msg) + 1);
        close(fd[1]);
    } else {                  // ---- child reads ----
        close(fd[1]);         // close unused write end
        char buf[64];
        ssize_t n = read(fd[0], buf, sizeof buf);
        if (n > 0) printf("child got: %s\n", buf);
        close(fd[0]);
    }
    return 0;
}
```

IPC means at a glance

Means	Direction	Structure	Synchronization	Characteristic
Signal	—	no payload	asynchronous	event notification
Pipe	unidirectional	byte stream	implicit	anonymous / named (FIFO)
Message Queue	multi R/W	structured (N×M)	implicit	priorities, needs `-lrt`
Socket	bidirectional	byte stream	implicit	local or distributed
Shared Memory	bidirectional	byte block	explicit (semaphore)	very fast
Shared File	bidirectional	byte block	explicit (file locks)	persistent

21 Shell & Linux Commands SHELL

Shell basics

The **shell** interprets command lines and runs scripts at startup; the `/bin/sh` login shell (bash) is named in `/etc/passwd`. **Flow:** input → tokens (split on space/tab); the first word = a builtin / function or a program (found via PATH). **Exit code** `0` = success / "true", else "false".

Cmd	Meaning	Cmd	Meaning
`echo`	display	`chmod`	change permissions
`ls`	list dir & files	`ps`	process states
`cd`	change directory	`pstree`	process hierarchy
`find`	search & display	`top` / `htop`	process states / + CPU load
`mkdir`	make directory	`lscpu`	list the CPUs
`touch`	create a file	`man`	manual
`nl`	number lines	`du`	disk usage
`pwd`	print working dir	`grep`	filter / search
`wc`	word count	`which`	locate a command
`mount` / `umount`	attach / remove a FS	`findmnt`	list mounted FS
`ln`	link (node)	`rm`	remove (delete file)
`apt`	package manager	`gcc` / `make`	compiler / build utility

man sections: 1 programs, 2 syscalls, 3 library, 4 special files, 5 formats.
Help: `--help` / `-h`, `man bash`.

Redirection, chaining & wildcards

- **Path abbreviations:** `~/` own home, `~name/`, `./` current, `../` parent.
  Wildcards: * (any), ? (exactly one), […] (one of those).
- **fds:** stdin=0, stdout=1, stderr=2.

Syntax	Effect
`… < file`	redirect stdin from a file
`… > file` (1>)	stdout to a new file (overwrite)
`… >> file`	append stdout to a file
`2> errfile`	create a file with stderr
`>& combi`	combine stdout + stderr
`cmd1 | cmd2`	pipe stdout of cmd1 into stdin of cmd2

Chaining: `a ; b` (sequence) · `a && b` (b only if a exits 0) · `a || b` (b only if a fails) · `a &` (background).

Bash scripting

Purpose	Syntax	Example
Set	`var=value`	`usage="usage: cmd arg1 arg2"`
Use	`$var` or `${var}`	`wer=$USER`
Transform	`${name//text/replacement}`	`var=${PATH//:/ }`
Command output	`$(command)`	`n=$(cat myfile.txt | wc -l)`
Arithmetic	`$((expression))`	`i=$((i+1))`

Test	Question it answers	Test	Question it answers
`[ -f "$path" ]`	Does the file exist?	`[ -n "$var" ]`	Is the length non-zero?
`[ -d "$path" ]`	Does the directory exist?	`[ -z "$var" ]`	Is the length zero?
`[ -x "$path" ]`	Execute permission?		

NOTE Added — a complete small bash script tying the pieces together: a usage guard, a loop over arguments, a file test, and command substitution.

```
#!/bin/bash
# count the lines of each file given as an argument
usage="usage: $0 file1 [file2 ...]"
if [ $# -lt 1 ]; then           # need at least one argument
    echo "$usage"
    exit 1
fi
for f in "$@"; do               # loop over all arguments
    if [ -f "$f" ]; then        # only if it is a real file
        n=$(wc -l < "$f")       # command substitution
        echo "$f: $n lines"
    else
        echo "$f: not a file" >&2   # message to stderr
    fi
done
```

22 Appendix: Handy Algorithms & ASCII

APPENDIX

Bubble sort & reverse an array

```c
for (i = 0; i < n-1; i++)            // bubble sort
    for (j = i+1; j < n; j++)
        if (a[i] > a[j]) { t = a[i]; a[i] = a[j]; a[j] = t; }

void reverseArray(int arr[], int n) {   // reverse in place
    int start = 0, end = n - 1;
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++; end--;
    }
}
```

Split a string into tokens

```c
char *token = strtok(string, " ");   // first token
while (token != NULL) {
    printf(" %s\n", token);
    token = strtok(NULL, " ");         // next token
}
```

Default values by storage duration

- **Automatic (local):** without an initializer the value is **indeterminate** (garbage).
- **Static** (global, or declared `static`): initialized to `0`; a function-local `static` keeps its value between calls.
- **Dynamic (`malloc`):** no default — whatever was there; `calloc` zero-initialises. Good practice: always initialise explicitly.
- **PID:** every process has a unique process ID (an integer); obtain it via `getpid()`.

ASCII quick reference

Char	Dec	Hex
`'\0'`	0	0x00
space	32	0x20
`'0'`	48	0x30
`'A'`	65	0x41
`'a'`	97	0x61

1 hex digit = 4 bits (`0x50` = 80) · 1 byte = 8 bits · `'A'` = 65 = `0x41`.

`'A' = 65 = 64 + 1 = 0100 0001 · MSB at bit 7 (left), LSB at bit 0 (right)`

FIG One byte holds 8 bits with place values 128…1; the set bits of `'A'` (64 + 1) give 65 = `0x41`. The MSB is bit 7 (left), the LSB is bit 0 (right).

NOTE Added — a complete program that prints the binary representation of an unsigned byte, MSB first. It shifts a mask from bit 7 down to bit 0 and prints 1 or 0 for each.

```c
#include <stdio.h>
/* print the 8 bits of one byte, most-significant first */
void print_bits(unsigned char v) {
    for (int i = 7; i >= 0; i--)            // bit 7 down to bit 0
        putchar((v >> i) & 1u ? '1' : '0');
    putchar('\n');
}
int main(void) {
    print_bits('A');   // 01000001 (65 = 0x41)
    print_bits(5);     // 00000101
    return 0;
}
```

END End of unified summary + recipes. Items flagged ⚠ **VERIFY** warrant a quick check against your lecture notes. All code is ANSI/POSIX C; on a paper open-book exam, write the numbered steps first, then the code.