C Handout

English edition

Contents

1 C basics **3**
- 1.1 main, printf & functions . . . 3
- 1.2 Variables, types & arithmetic . . . 4
- 1.3 Comments & style . . . 5

2 Selection **6**
- 2.1 if / else . . . 6
- 2.2 switch . . . 7

3 Loops **8**
- 3.1 while & for loops . . . 8
- 3.2 Accumulation . . . 9
- 3.3 Nested loops & patterns . . . 9

4 Functions **10**
- 4.1 Parameters & return values . . . 10
- 4.2 Prototypes & scope . . . 11

5 Arrays **12**
- 5.1 1-D arrays . . . 12
- 5.2 Array algorithms . . . 13
- 5.3 2-D arrays . . . 13

6 Pointers **14**
- 6.1 &, * and pass-by-pointer . . . 14

7 Strings **16**
- 7.1 Char arrays & the null terminator . . . 16
- 7.2 The string.h library . . . 17

8 Structs **18**
- 8.1 struct, typedef, . vs -> . . . 18

9 Dynamic memory **19**
- 9.1 malloc, free & realloc . . . 19

10 Data structures **21**
- 10.1 Linked lists . . . 21
- 10.2 Stacks & queues . . . 23

11 Searching, sorting & recursion **24**
- 11.1 Linear & binary search . . . 24
- 11.2 Sorting . . . 25

11.3 Recursion 26

12 Files & errors **27**
12.1 Text files 27
12.2 Error handling with return codes 28

13 Bits & data **29**
13.1 Bits, binary & bitwise operators 29
13.2 Run-length encoding 30

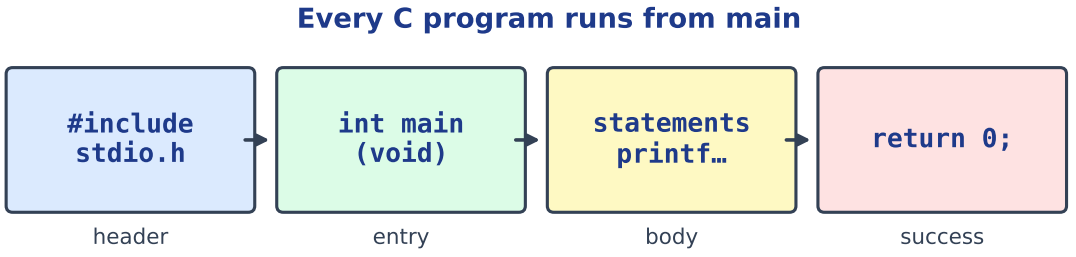
14 Preprocessor & qualifiers **31**
14.1 Preprocessor, const, static, enum 31

1 C basics

1.1 ■ main, printf & functions

Every C program starts in `main`. `#include <stdio.h>` pulls in a header so you can use `printf` to print. A function is a named block you can call; `\n` starts a new line. `main` returns `0` to mean "success".

```c
#include <stdio.h>

void greet(void) {
    printf("Hello, world!\n");
}

int main(void) {
    greet();
    printf("I am learning C.\n");
    return 0;
}
```



Exam tip: #include for libraries; main returns 0 for success; end statements with ;

Every C program runs from main after including headers

1.2 ■ Variables, types & arithmetic

C needs a type for every variable: `int` (integer), `double` (floating-point), `char` (one character). `printf` uses a format specifier – `%d`, `%f`, `%c` – for each value. `/` between two ints is integer division; `%` is the remainder .

```c
#include <stdio.h>

int main(void) {
    int n = 17;
    double pi = 3.14;
    char grade = 'A';
    printf("%d %.2f %c\n", n, pi, grade);   // 17 3.14 A
    printf("%d %d\n", 7 / 2, 7 % 2);        // 3 1
    return 0;
}
```

The format string drives both printing and reading:

Specifier	Type	Example output
`%d`	`int`	`17`
`%f`	`double` (`%.2f` = 2 decimal places)	`3.14`
`%c`	`char`	`A`
`%s`	a string	`Mei`
`%zu`	a size (from `sizeof` / `strlen`)	`3`
`%x`	`unsigned int`, printed in base 16	`ff`

`scanf("%d", &n)` reads an `int` through its address (note the `&`). Input and output formats are not identical: `printf("%f", x)` prints a `double`, but `scanf("%lf", &x)` reads a `double`; input `%f` needs a `float *`. Check the returned count of successfully assigned items. Its sibling `sscanf` parses values out of a string:

```c
#include <stdio.h>

int main(void) {
    char line[] = "Mei 88";
    char name[20];
    int score;
    if (sscanf(line, "%19s %d", name, &score) != 2) return 1;
    printf("%s scored %d\n", name, score);   // Mei scored 88
    return 0;
}
```

1.3 ■ Comments & style

A comment is a note for humans; the compiler ignores it. Use `//` for one line and `/* ... */` for a block. Good indentation and clear names make code easy to read.

```c
#include <stdio.h>

int main(void) {
    // a single-line comment
    /* a block
       comment */
    int total = 3 + 4;        // clear names help
    printf("%d\n", total);    // 7
    return 0;
}
```

Common mistakes

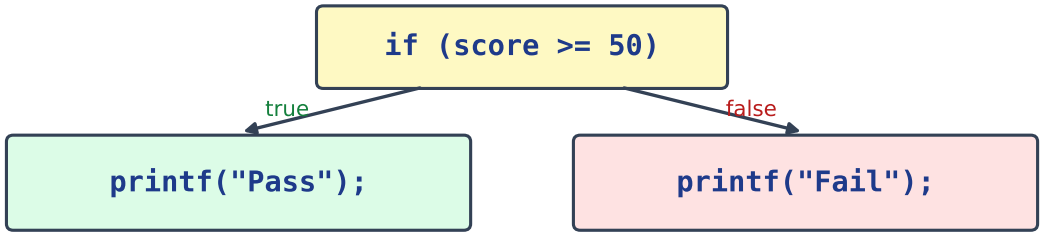
- Declarations and expression statements end with `;`; a compound block `{ ... }` needs no extra semicolon. `main` should return `int` (`return 0;`).
- `printf` needs a format string: `%d` for `int`, `%f` for `double`, and `\n` for a new line.
- A wrong format like `%d` for a `double` causes undefined behaviour; it is not guaranteed merely to print garbage. Match the argument type.

2 Selection

2.1 ■ if / else

An `if` runs a block when a condition is true. A condition is tested as a number: `0` is false and any non-zero value is true. (C99's `<stdbool.h>` adds a boolean type, `bool` with `true` and `false`, but these are still `1` and `0` underneath.) Chain choices with `else if` and `else`. Compare with `==`, `!=`, `<`, `>`, `<=`, `>=`.

```c
#include <stdio.h>

int main(void) {
    int score = 72;
    if (score >= 90) {
        printf("A\n");
    } else if (score >= 60) {
        printf("Pass\n");
    } else {
        printf("Fail\n");
    }
    return 0;
}
```



Exam tip: braces { } group the body; only one branch runs; else is optional

if chooses the true branch; else the false branch

2.2 ■ switch

A `switch` picks one `case` by an integer value. Use `break` to jump out when a case should end. Without a control transfer, execution falls through into the next case; this can be intentional. `default` runs when nothing matches.

```c
#include <stdio.h>

int main(void) {
    int day = 3;
    switch (day) {
        case 1: printf("Mon\n"); break;
        case 2: printf("Tue\n"); break;
        case 3: printf("Wed\n"); break;
        default: printf("Other\n");
    }
    return 0;
}
```

Common mistakes

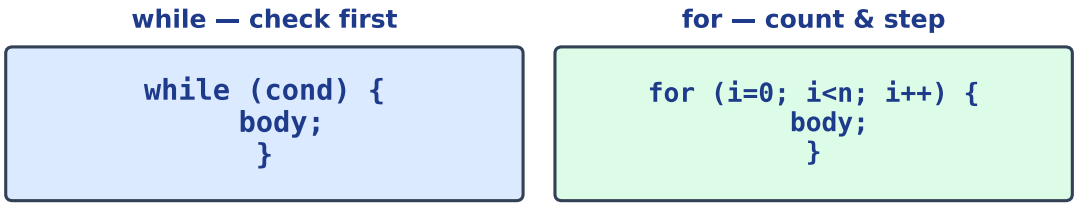
- `if (x = 5)` assigns and is always true; use `==` to compare.
- End a `switch` case with `break;` when fall-through into the next case is not intended.
- `0` is false and any non-zero value is true in a condition.

3 Loops

3.1 ■ while & for loops

A loop repeats a block. A `while` loop runs as long as a condition is true. A `for` loop packs the start, the test, and the increment (`i++`) into one line – best when you know how many times to repeat.

```c
#include <stdio.h>

int main(void) {
    int i = 1;
    while (i <= 3) {
        printf("%d ", i);
        i++;
    }
    printf("\n");                       // 1 2 3
    for (int j = 0; j < 5; j++) {
        printf("%d ", j);
    }
    printf("\n");                       // 0 1 2 3 4
    return 0;
}
```



Exam tip: both need a way to end; for packs init, test, step; while is open-ended

while checks a condition; for counts with a step

3.2 ■ Accumulation

A common pattern: start an accumulator at `0`, then add to it inside a loop. The same idea counts items or finds a running total.

```c
#include <stdio.h>

int main(void) {
    int total = 0;
    for (int i = 1; i <= 5; i++) {
        total += i;            // 1 + 2 + 3 + 4 + 5
    }
    printf("%d\n", total);     // 15
    return 0;
}
```

3.3 ■ Nested loops & patterns

A loop inside another loop is a nested loop. The inner loop finishes fully for each step of the outer loop – perfect for grids and patterns.

```c
#include <stdio.h>

int main(void) {
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

Common mistakes

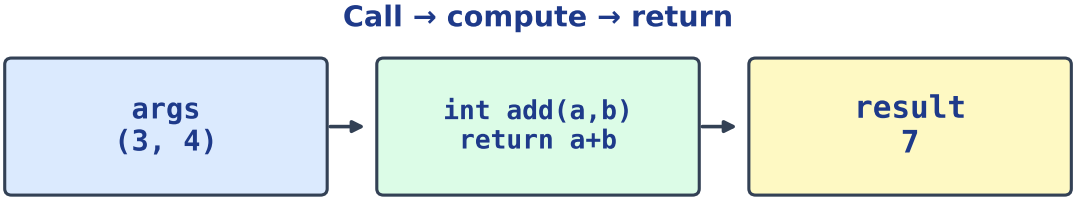
- `for (i = 0; i < n; i++)` runs `n` times; a semicolon right after `for (...)` makes an empty loop.
- A `while` whose condition never becomes false loops forever.
- Declare the counter (`int i`) before or inside the `for` header.

4 Functions

4.1 ■ Parameters & return values

A function takes parameters (inputs) and gives back a return value . The return type comes first (`int`, `double`, …); `void` means it returns nothing.

```c
#include <stdio.h>

int square(int x) {         // x is a parameter
    return x * x;           // hand back a value
}

int main(void) {
    int r = square(5);
    printf("%d\n", r);      // 25
    return 0;
}
```



Exam tip: declare the return type; parameters need types; return ends the function

Parameters in; return value out

4.2 ■ Prototypes & scope

C reads top to bottom, so a function must be known before it is called. A prototype – the header line plus ; – declares it early so you can keep `main` first. A variable's scope is the block it lives in: it is local and disappears when the block ends.

```c
#include <stdio.h>

int add(int a, int b);      // prototype: declared before use

int main(void) {
    printf("%d\n", add(3, 4));   // 7
    return 0;
}

int add(int a, int b) {     // definition comes later
    int sum = a + b;        // sum is local to add
    return sum;
}
```

Common mistakes

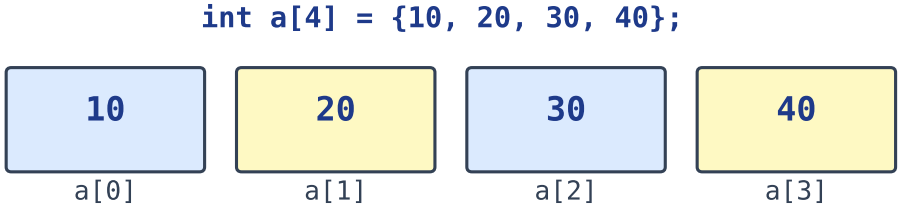
- A function used before it is defined needs a prototype above `main`.
- Arguments are passed BY VALUE – changes inside a function do not affect the caller unless you pass a pointer.
- A variable declared inside a function is local to it.

5 Arrays

5.1 ■ 1-D arrays

An array holds several values of one type. Index from `0`. C does **not** store an array's length, so a common trick computes the element count: `sizeof(a) / sizeof(a[0])`.

```c
#include <stdio.h>

int main(void) {
    int scores[3] = {88, 71, 95};
    printf("%d\n", scores[0]);     // 88
    scores[1] = 100;
    printf("%d\n", scores[1]);     // 100
    int n = sizeof(scores) / sizeof(scores[0]);
    printf("%d\n", n);             // 3
    return 0;
}
```



Exam tip: indices start at 0; a[i] is one element; length is fixed at creation

Array slots are indexed from 0

5.2 ■ Array algorithms

Traverse an array with a `for` loop to find a max, a total, or a count. Always loop from `0` up to `n - 1`.

```c
#include <stdio.h>

int main(void) {
    int a[] = {3, 9, 2, 7};
    int n = sizeof(a) / sizeof(a[0]);
    int max = a[0], total = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] > max) max = a[i];
        total += a[i];
    }
    printf("%d %d\n", max, total);   // 9 21
    return 0;
}
```

5.3 ■ 2-D arrays

A 2-D array is a grid of rows and columns : `grid[row][col]`. Use two nested loops to visit every cell.

```c
#include <stdio.h>

int main(void) {
    int grid[2][3] = {{1, 2, 3}, {4, 5, 6}};
    printf("%d\n", grid[1][2]);    // 6
    for (int r = 0; r < 2; r++) {
        for (int c = 0; c < 3; c++) {
            printf("%d ", grid[r][c]);
        }
    }
    printf("\n");                  // 1 2 3 4 5 6
    return 0;
}
```

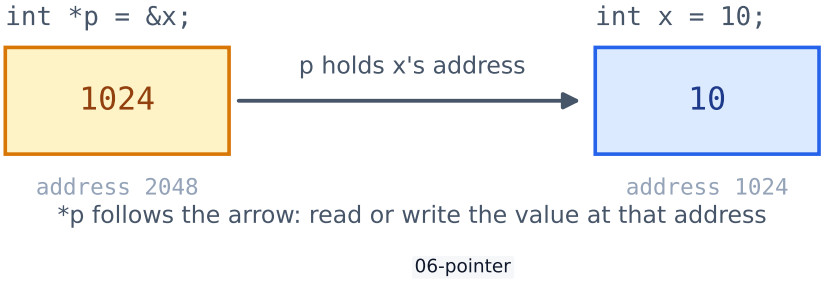
Common mistakes

- C does NOT check array bounds: writing `a[n]` in a size-`n` array is undefined behaviour.
- Valid indexes run from `0` to `n - 1`.
- In most expressions an array converts to a pointer to its first element, but the array itself is not a pointer. `sizeof(a)` gives the whole byte size for an actual array; an array parameter is adjusted to a pointer, so that function needs a separately supplied length.

6 Pointers

6.1 ■ &, * and pass-by-pointer

A pointer stores the address of a variable. `&x` gives the address of `x`; `*p` dereferences the pointer – it reads or writes the value stored there. Passing a pointer lets a function change the caller's variable (pass-by-pointer).



A pointer stores an address; dereferencing it follows the arrow to the value

```c
#include <stdio.h>

void addOne(int *p) {     // p holds an address
    *p = *p + 1;          // change the value at that address
}

int main(void) {
    int x = 10;
    int *ptr = &x;        // & takes the address of x
    printf("%d\n", *ptr); // * reads the value: 10
    addOne(&x);
    printf("%d\n", x);    // 11 - changed through the pointer
    return 0;
}
```

The classic use is a swap function. Passing values only copies them – the swap is lost. Passing pointers lets the function reach the caller's variables:

```c
#include <stdio.h>

void swap_values(int a, int b) {      // copies: the caller sees nothing
    int t = a; a = b; b = t;
}

void swap_pointers(int *a, int *b) {  // addresses: the swap is real
    int t = *a; *a = *b; *b = t;
}

int main(void) {
    int x = 1, y = 2;
    swap_values(x, y);
    printf("%d %d\n", x, y);   // 1 2  (unchanged!)
    swap_pointers(&x, &y);
    printf("%d %d\n", x, y);   // 2 1  (swapped)
    return 0;
}
```

An array name acts as a pointer to its first element, and pointer arithmetic steps by whole elements:

```c
#include <stdio.h>

int main(void) {
    int a[] = {10, 20, 30};
    int *p = a;                  // same as &a[0]
    printf("%d\n", *p);          // 10
    printf("%d\n", *(p + 2));    // 30 - same as a[2]
    return 0;
}
```

- `NULL` is the pointer that points at nothing – check for it before you dereference.

Common mistakes

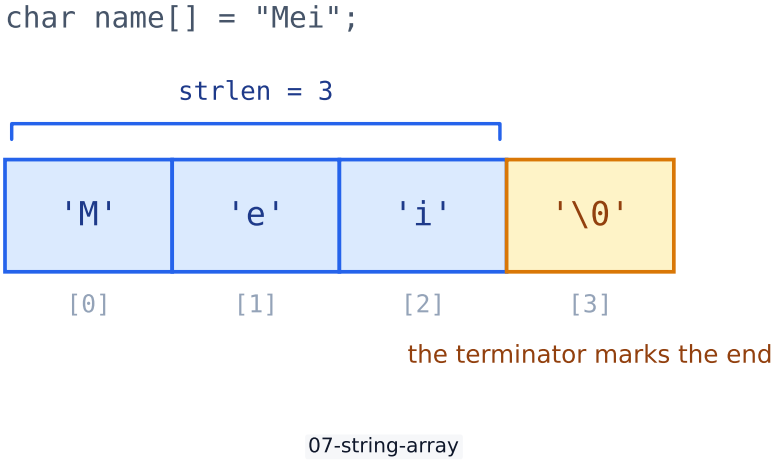
- `&x` is the address of `x`; `*p` is the value stored at `p`.
- Never use `*p` on an uninitialised or `NULL` pointer – it crashes or corrupts memory.
- To change a caller's variable, pass its address and write through the pointer.

7 Strings

7.1 ■ Char arrays & the null terminator

A C string is an array of `char`. Every string ends with a hidden null terminator `'\0'`, which marks where it stops. `strlen` (from `<string.h>`) counts characters up to that `'\0'`. Print a whole string with `%s` and one character with `%c`.

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

int main(void) {
    char name[] = "Mei";            // 'M', 'e', 'i', '\0'
    printf("%s\n", name);           // Mei
    printf("%zu\n", strlen(name));  // 3 (stops at '\0')
    printf("%c\n", name[0]);        // M
    return 0;
}
```



A C string is a char array ending in a hidden null terminator

You can traverse a string by looping until you reach `'\0'`.

```c
#include <stdio.h>

int main(void) {
    char word[] = "banana";
    int count = 0;
    for (int i = 0; word[i] != '\0'; i++) {
        if (word[i] == 'a') count++;
    }
    printf("%d\n", count);   // 3
    return 0;
}
```

7.2 ■ The string.h library

`#include <string.h>` gives you the everyday string tools:

Function	Does
`strlen(s)`	the length, not counting the terminator
`strcpy(dst, src)`	copy – `dst` must be big enough
`strcat(dst, src)`	append `src` onto the end of `dst`
`strcmp(a, b)`	compare: `0` when equal, else negative / positive

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

int main(void) {
    char full[40];
    strcpy(full, "Mei");           // full is now "Mei"
    strcat(full, " Chen");         // append -> "Mei Chen"
    printf("%s (%zu)\n", full, strlen(full));       // Mei Chen (8)
    printf("%d\n", strcmp("apple", "apple") == 0);  // 1 (equal)
    return 0;
}
```

- `strcmp` returns `0` for equal, so the test is `strcmp(a, b) == 0` – never `a == b`.
- The destination array must have room for the result **plus** the terminator.

Common mistakes

- A C string needs one extra byte for the `\0` terminator: a 5-letter word needs `char[6]`.
- Copy and compare strings with `strcpy` / `strcmp`, not `=` / `==`.
- Forgetting the `\0` makes `printf("%s", ...)` run past the end of the text.

8 Structs

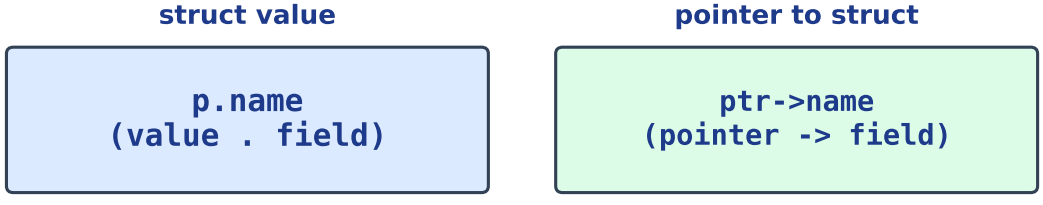
8.1 ■ struct, typedef, . vs ->

A struct groups related values into one type. Each value is a member . `typedef` gives the struct a short name so you can write `Dog` instead of `struct Dog`. Use `.` on a struct value, but `->` on a pointer to a struct.

```c
#include <stdio.h>

typedef struct {
    char name[20];
    int age;
} Dog;

int main(void) {
    Dog d = {"Rex", 3};
    printf("%s is %d\n", d.name, d.age);   // Rex is 3   (. on a value)

    Dog *p = &d;
    p->age = 4;                            // -> on a pointer
    printf("%s is %d\n", d.name, d.age);   // Rex is 4
    return 0;
}
```

Common mistakes

- Use `.` on a struct value and `->` on a struct pointer.
- `typedef` lets you drop the word `struct` when declaring; without it you write `struct Point p;`.
- Assigning one struct to another copies all its fields.



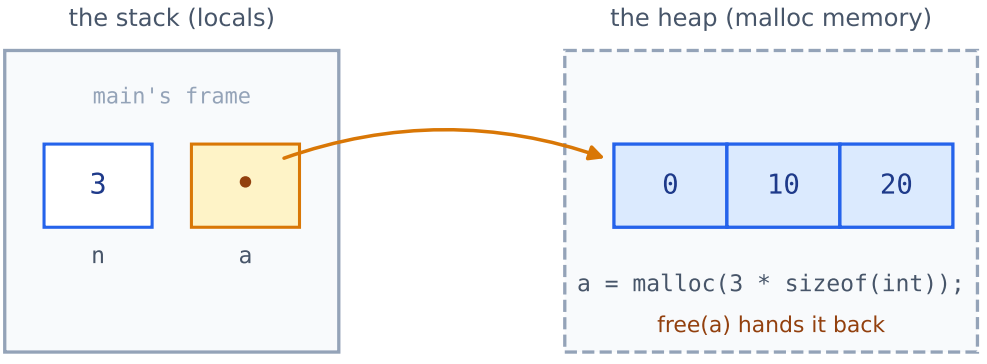
Exam tip: use . on a struct; use -> on a pointer (same as (*ptr).field)

. for struct values; -> for pointers to structs

9 Dynamic memory

9.1 ■ malloc, free & realloc

`malloc` allocates memory on the heap at run time and returns a pointer to it. `realloc` resizes that block; `free` gives it back. Every `malloc` needs a matching `free`, or you get a memory leak . These live in `<stdlib.h>`.



09-stack-heap

Locals live on the stack; malloc memory lives on the heap until you free it

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

int main(void) {
    int n = 3;
    int *a = malloc(n * sizeof(int));    // room for 3 ints
    if (a == NULL) return EXIT_FAILURE;
    for (int i = 0; i < n; i++) a[i] = i * 10;

    int *grown = realloc(a, 4 * sizeof *a); // positive size
    if (grown == NULL) {
        free(a);
        return EXIT_FAILURE;
    }
    a = grown;
    a[3] = 30;

    for (int i = 0; i < 4; i++) printf("%d ", a[i]);
    printf("\n");                           // 0 10 20 30

    free(a);                                // hand the memory back
    return 0;
}
```

- `calloc(n, size)` allocates an array like `malloc` AND fills it with zeros.

Common mistakes

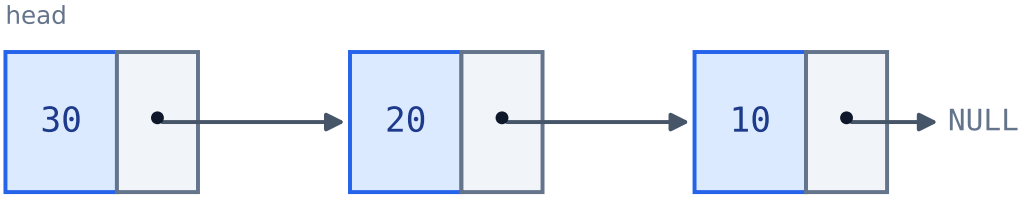
- Every `malloc` needs a matching `free`; forgetting to free leaks memory.
- Using memory after `free` (a "use-after-free") is a serious bug.
- Check that `malloc` did not return `NULL` before you use the memory.

For positive sizes, a failed `realloc` returns `NULL` and leaves the old allocation intact: check a temporary result before replacing your only pointer. Success preserves the smaller of the old/new byte counts; added cells are uninitialised. This example avoids zero-size requests and size overflow. Ordinary non-static locals have automatic storage duration and are usually implemented on a stack; C does not require that layout.

10 Data structures

10.1 ■ Linked lists

A linked list is a chain of nodes . Each node holds a value and a pointer to the `next` node; the last one points to `NULL`. Unlike an array it grows one node at a time with `malloc`. Always `free` every node when done.



10-linked-list

A linked list: each node holds a value and a pointer to the next, ending at NULL

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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

int main(void) {
    Node *head = NULL;
    for (int v = 10; v <= 30; v += 10) {   // prepend 10, 20, 30
        Node *n = malloc(sizeof(Node));
        n->value = v;
        n->next = head;
        head = n;
    }
    for (Node *p = head; p != NULL; p = p->next) {
        printf("%d ", p->value);           // 30 20 10
    }
    printf("\n");

    while (head != NULL) {                 // free the whole list
        Node *t = head;
        head = head->next;
        free(t);
    }
    return 0;
}
```

10.2 ■ Stacks & queues

A stack is LIFO (last in, first out): push and pop at the same end. A queue is FIFO (first in, first out): add at the back, remove from the front. Both are easy to build on an array with index variables.

```c
#include <stdio.h>

int main(void) {
    int stack[10];
    int top = 0;                        // next free slot
    stack[top++] = 1;                   // push
    stack[top++] = 2;
    stack[top++] = 3;
    while (top > 0) {
        printf("%d ", stack[--top]);    // pop: 3 2 1
    }
    printf("\n");
    return 0;
}
```

```c
#include <stdio.h>

int main(void) {
    int queue[10];
    int front = 0, back = 0;
    queue[back++] = 1;                          // enqueue
    queue[back++] = 2;
    queue[back++] = 3;
    while (front < back) {
        printf("%d ", queue[front++]);          // dequeue: 1 2 3
    }
    printf("\n");
    return 0;
}
```

Common mistakes

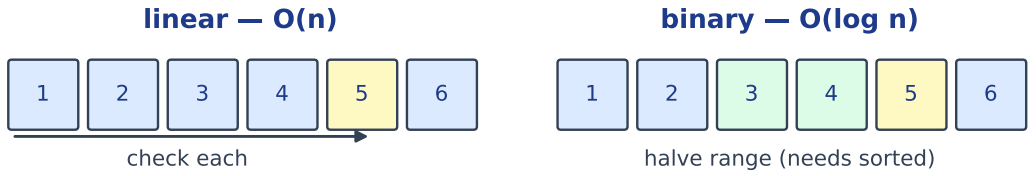
- In a linked list, never lose the `head` pointer, or the whole list is unreachable.
- Free every node when you are done, walking the list before you unlink.
- Check for an empty list (a `NULL` head) before you pop.

11 Searching, sorting & recursion

11.1 ■ Linear & binary search

Linear search checks each element in turn – it works on any array. Binary search is far faster but needs a sorted array: it checks the middle and discards half each step. Both return the index, or `-1` if missing.

```c
#include <stdio.h>

int linear(int a[], int n, int target) {
    for (int i = 0; i < n; i++)
        if (a[i] == target) return i;
    return -1;
}

int binary(int a[], int n, int target) {
    int lo = 0, hi = n - 1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (a[mid] == target) return mid;
        else if (a[mid] < target) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}

int main(void) {
    int a[] = {2, 5, 8, 12, 16, 23};
    int n = sizeof(a) / sizeof(a[0]);
    printf("%d %d %d\n", linear(a, n, 12), binary(a, n, 12), binary(a,
↪   n, 9));
    return 0;   // 3 3 -1
}
```



Exam tip: binary search needs a sorted array; linear works on any order

Linear scans all; binary halves a sorted range

11.2 ■ Sorting

Bubble sort repeatedly compares neighbours and swaps any pair that is out of order. After each pass the largest value "bubbles" to the end.

```c
#include <stdio.h>

int main(void) {
    int a[] = {5, 2, 9, 1, 7};
    int n = sizeof(a) / sizeof(a[0]);
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (a[j] > a[j + 1]) {
                int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
            }
        }
    }
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");   // 1 2 5 7 9
    return 0;
}
```

11.3 ■ Recursion

Recursion is a function that calls itself. It needs a base case to stop, and a recursive call that steps toward it. Without a base case it never ends.

```c
#include <stdio.h>

int factorial(int n) {
    if (n <= 1) return 1;           // base case
    return n * factorial(n - 1);    // recursive call
}

int main(void) {
    printf("%d\n", factorial(5));  // 120
    return 0;
}
```

Common mistakes

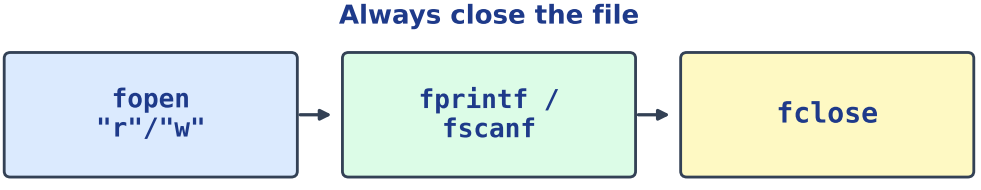
- Binary search needs a **sorted** array; recursion needs a base case.
- Each recursive call must move closer to the base case, or the stack overflows.
- Bubble and insertion sort are $O(n^2)$ – fine to learn, slow at scale.

12 Files & errors

12.1 ■ Text files

`fopen` returns a file pointer ; the mode string says what to do – `"w"` write, `"r"` read, `"a"` append. Write with `fprintf`, read with `fscanf`, and always `fclose` when done.

```c
#include <stdio.h>

int main(void) {
    FILE *out = fopen("scores.txt", "w");   // open for writing
    if (out == NULL) {
        perror("scores.txt");
        return 1;
    }
    fprintf(out, "Alice 80\nBob 95\n");
    fclose(out);

    FILE *in = fopen("scores.txt", "r");    // open for reading
    if (in == NULL) {
        perror("scores.txt");
        return 1;
    }
    char name[20];
    int score;
    while (fscanf(in, "%19s %d", name, &score) == 2) {   // %19s leaves
↪   room for '\0'
        printf("%s -> %d\n", name, score);
    }
    fclose(in);
    return 0;
}
```



Exam tip: check fopen != NULL; mode "r" reads, "w" writes; fclose when done

fopen → read/write → fclose

12.2 ■ Error handling with return codes

C has no built-in exceptions. A function may report its outcome through a return code ; check that function's rules before using the result. The example below uses `0` for success and `-1` for failure. `fopen` instead returns a stream pointer, or `NULL` when it fails.

```c
#include <stdio.h>

// returns 0 on success, -1 if the divisor is 0
int safe_divide(int a, int b, int *result) {
    if (b == 0) return -1;          // error code
    *result = a / b;
    return 0;                       // success
}

int main(void) {
    int r;
    if (safe_divide(10, 2, &r) == 0)
        printf("10 / 2 = %d\n", r);          // 10 / 2 = 5
    if (safe_divide(10, 0, &r) != 0)
        printf("cannot divide by zero\n");   // error reported
    return 0;
}
```

Common mistakes

- Check that `fopen` did not return `NULL` before reading or writing.
- Always `fclose` a file when you are done.
- Return a non-zero code from `main` to signal an error to the caller.

13 Bits & data

13.1 ■ Bits, binary & bitwise operators

A bit is a single 0 or 1; numbers are stored in binary . Bitwise operators work on the bits directly: `&` AND, `|` OR, `^` XOR, `<<` shift left (×2 each step), `>>` shift right (÷2).

```c
#include <stdio.h>

int main(void) {
    int a = 12;               // 1100
    int b = 10;               // 1010
    printf("%d\n", a & b);    // 8   AND  -> 1000
    printf("%d\n", a | b);    // 14  OR   -> 1110
    printf("%d\n", a ^ b);    // 6   XOR  -> 0110
    printf("%d\n", a << 1);   // 24  left shift
    printf("%d\n", a >> 1);   // 6   right shift
    return 0;
}
```

Hexadecimal (base 16) writes binary compactly: one hex digit is exactly four bits. Write hex literals with `0x`; print with `%x`:

```c
#include <stdio.h>

int main(void) {
    printf("%d\n", 0xFF);   // 255  (0x marks a hex literal)
    printf("%x\n", 255);    // ff
    printf("%x\n", 12);     // c
    return 0;
}
```

Exam tip: bitwise ops work on bits of integers; << multiplies by 2, >> divides by 2

Bitwise operators work on the bits of integers

13.2 ■ Run-length encoding

Run-length encoding (RLE) is a simple compression method: replace each run of repeated characters with a count and the character. It is lossless – the original is fully recoverable.

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

int main(void) {
    char *s = "aaabbc";
    int n = strlen(s);
    for (int i = 0; i < n; ) {
        char c = s[i];
        int run = 0;
        while (i < n && s[i] == c) { run++; i++; }
        printf("%d%c", run, c);     // 3a2b1c
    }
    printf("\n");
    return 0;
}
```

Common mistakes

- `&` is bitwise AND and `&&` is logical AND – do not confuse them.
- A left shift `<< 1` doubles a value; a right shift `>> 1` halves it.
- Bit `n` has value `1 << n` (C has no `**` power operator); check a bit with `(x >> n) & 1`.

14 Preprocessor & qualifiers

14.1 ■ Preprocessor, const, static, enum

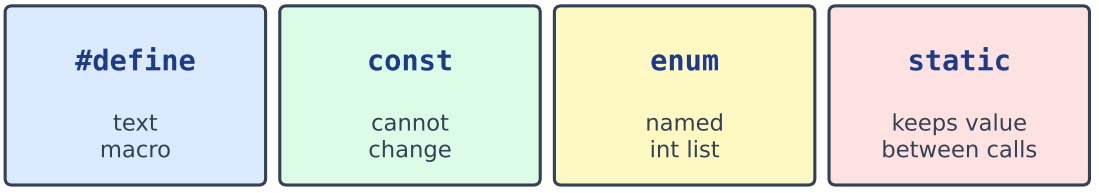
The preprocessor runs before compiling. `#define` makes a macro – plain text replaced everywhere. `const` makes a constant that cannot change. `static` inside a function keeps a variable's value between calls. An `enum` (enumeration) names a set of integers starting at `0`.

```c
#include <stdio.h>

#define MAX 100                     // macro: text replaced before
↪  compiling

enum Color { RED, GREEN, BLUE };    // named integers 0, 1, 2

int next_id(void) {
    static int count = 0;           // keeps its value between calls
    count++;
    return count;
}

int main(void) {
    const double PI = 3.14;         // cannot be changed
    printf("%d %.2f\n", MAX, PI);           // 100 3.14
    printf("%d %d %d\n", RED, GREEN, BLUE); // 0 1 2
    int a = next_id();
    int b = next_id();
    printf("%d %d\n", a, b);                // 1 2
    return 0;
}
```

Bigger programs split across files: declarations go in a header file (`mine.h`), included with `#include "mine.h"` – quotes mean your own file, `< >` means the standard library. An include guard (`#ifndef MINE_H` / `#define MINE_H` / `#endif`) stops a header being included twice.

Common mistakes

- `#define` does plain text replacement with no type checking; wrap macro bodies and arguments in parentheses.
- A `const` value cannot be changed after it is set.
- A `static` local variable keeps its value between calls to the function.



Exam tip: preprocessor runs first; enum values start at 0 unless set

#define, const, enum, and static keep code clear