DevSheets

C

Everyday C17, with explicit boundaries.

01 Compile & run

```c
#include <stdio.h>
int main(void) {
  puts("Hello, C!");
  return 0;
}
```

Save as main.c. Compile: gcc -std=c17 main.c -o app. Run ./app. Later examples are independent snippets: place headers above main and statements inside it.

02 Variables & types

```c
#include <stdbool.h>
int count = 3;
double price = 19.95;
bool ready = true;
char grade = 'A';
const int limit = 10;
unsigned flags = 0u;
```

Initialize variables before reading them. Type sizes can vary by platform. const prevents modification through that name; C17 has no automatic type inference.

03 Operators & conversions

```c
int whole = 7 / 2;       // 3
double ratio = 7 / 2.0;  // 3.5
int remainder = 7 % 2;   // 1
int n = (int)ratio;      // 3
int ok = whole > 0 && remainder == 1;
```

Use + - * / %, == != < <= > >=, and && || !. Comparisons produce 0 or 1. Integer division truncates toward zero. Avoid signed overflow and division by zero.

04 Conditions & switch

```c
int score = 82;
if (score >= 60) ++score; else score = 0;
switch (score) {
  case 83: --score; break;
  default: score = 0; break;
}
```

Zero is false; nonzero is true. Use == to compare, not =. switch uses an integer or enum value; break prevents falling into the following case.

05 Loops

```c
int total = 0;
for (int i = 0; i < 3; ++i) {
  total += i;
}
while (total > 0) { --total; }
```

break exits the nearest loop; continue skips to its next iteration. A do/while loop runs its body at least once. Check bounds before accessing an array.

06 Functions

```c
int add(int a, int b) {
  return a + b;
}
int square(int n) { return n * n; }
int total = add(2, 3); // 5
int area = square(4); // 16
```

Define functions outside main, before calls, or declare prototypes first. C passes arguments by value. Use (void) for no parameters; void as a return type means no returned value.

07 Arrays & sizes

```c
#include <stddef.h>
int nums[3] = {10, 20, 30};
size_t count = sizeof nums / sizeof nums[0];
nums[1] = 25;
int zeroes[3] = {0}; // all elements zero
```

Indices run from 0 to count - 1, with no bounds checks. sizeof gives bytes. The element-count formula works for an array object, not a pointer or array parameter.

08 Strings & termination

```c
#include <string.h>
char name[16] = "Maya";
size_t length = strlen(name); // 4
name[0] = 'm';
int same = strcmp(name, "maya") == 0;
char label[] = "hi"; // 3 bytes with \0
```

Strings end with \0; reserve room for it. strlen excludes the terminator; strcmp returns 0 for equality. An array initialized from a literal is writable; the literal itself is not.

09 Bounded formatting

```c
#include <stdio.h>
char text[16];
int used = snprintf(text, sizeof text,
                    "%s %d", "Age", 28);
if (used < 0 || (size_t)used >= sizeof text)
  return 1;
puts(text);
```

snprintf returns the required length without the terminator. A negative result is an error; a result at least the capacity means truncation. Keep format strings under your control.

C

Pointers, ownership, and checked I/O.

10 Pointers & lifetimes

```c
#include <stddef.h>
int value = 7;
int *ptr = &value;
*ptr = 9; // value is now 9
int *empty = NULL;
if (ptr) { *ptr = 10; }
```

& gets an address; * accesses the pointed-to object. A non-NULL pointer can still be invalid. Only dereference live objects; never return the address of a local variable.

11 Pointer parameters

```c
void set_value(int *value) {
  if (value) { *value = 42; }
}
int count = 0;
set_value(&count);
```

Define set_value outside main. The pointer is copied, but it lets the function modify the caller's object. Pass array lengths or buffer capacities separately.

12 Structs & enums

```c
struct Point { int x; int y; };
enum Status { TODO, DONE };
struct Point p = {.x = 2, .y = 3};
struct Point *ptr = &p;
ptr->x = 4;
enum Status state = DONE;
```

Use . for a struct object and -> for a pointer to one. Struct assignment copies member values, including pointers, not the objects they point to. enum names integer constants.

13 Allocate & free memory

```c
#include <stdlib.h>
int *items = calloc(3, sizeof *items);
if (!items) return 1;
items[0] = 42;
free(items);
```

Check for NULL. calloc zeroes bytes; malloc does not. Free owned storage once; its aliases are then invalid.

14 Resize without losing data

```c
#include <stdlib.h>
int *items = malloc(3 * sizeof *items);
if (!items) return 1;
int *grown = realloc(items,
                     6 * sizeof *items);
if (!grown) { free(items); return 1; }
free(grown);
```

Use a temporary pointer. Failure keeps the original allocation valid; success invalidates the old pointer. Newly added bytes are uninitialized.

15 Read bounded text

```c
#include <stdio.h>
#include <string.h>
char line[64];
if (!fgets(line, sizeof line, stdin))
  return 1;
line[strcspn(line, "\n")] = '\0';
puts(line);
```

fgets reads at most capacity - 1 characters and terminates the string on success. A full buffer may contain only part of a line; reject it or continue reading. Check success before use.

16 Files & stream errors

```c
#include <stdio.h>
FILE *file = fopen("notes.txt", "w");
if (!file) return 1;
int failed = fputs("Hello\n", file) == EOF;
if (fclose(file) == EOF) failed = 1;
if (failed) return 1;
```

"w" creates or truncates a file; "r" opens for reading. Check opening, writes, and closing. Pass a FILE* to fgets for bounded line input. Do not use a stream after closing it.

17 Parse numbers & errors

```c
#include <errno.h>
#include <stdlib.h>
char *end; const char *text = "42";
errno = 0;
long value = strtol(text, &end, 10);
if (errno == ERANGE || end == text || *end)
  return 1;
```

Reject range errors, no digits, and trailing text. Unlike atoi, strtol exposes conversion failures. Reset errno before the call; check destination limits before converting long to int.

18 Headers & declarations

```c
#ifndef APP_H
#define APP_H
#define BUFFER_SIZE 64
int add(int a, int b);
#endif
```

Put this in app.h; include it with #include "app.h". Define add once in a .c file. Header guards prevent repeated inclusion. Prefer functions to function-like macros.