C Control Flow

Table of contents

The three shapes . . . 1
Truth . . . 2
Selection — `if` . . . 2
Repetition — `while` . . . 3
Repetition — `for` . . . 3
Infinite loops, deliberately . . . 4
Other control statements . . . 4
See also . . . 4

The three shapes a C program's control can take — **sequential**, **selection** and **repetition** — with the statements the course uses for each: `if … else`, `while` and `for`. First met in 2026-09-07-introduction-to-embedded-c-programming[1] and 2026-09-10-more-c-programming-on-the-avr[2]; the assembly equivalents are in avr-program-structure[3].

The three shapes

	Shape	Statements
sequential	statements run one after another	the default — any list of statements
selection (decision)	a test picks one of two paths	`if … else`
repetition	a body runs again while a test holds	`for`, `while`

[1] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[2] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[3] courses/csse2010/avr-program-structure.pdf

All three are drawn as flowcharts in 2026-09-07-introduction-to-embedded-c-programming[4]. Every program is built from nothing but these, nested.

Truth

There is no boolean type. **C interprets any 0 value as false, anything else as true.** So:

Written	Means
`if(a)`	`if(a != 0)`
`if(!a)`	`if(a == 0)`

Comparisons yield 1 or 0, so `if (a > b)` and `if ((a > b) != 0)` are the same test. A bitwise test like `PINC & 0x01` yields either `0x01` or `0x00`, which is why it can sit directly inside an `if` — anything non-zero is true, and it does not need to be exactly 1.

To the hardware a truth test is a compare against zero and a branch on **Z** — `tst`/`cp` then `breq`/`brne` in avr-instructions[5]. `if(a)` is the C spelling of *branch if not zero*.

Selection — `if`

```
if (expression)
    stmt
else
    stmt
```

- **The `else` clause is optional.**
- **`stmt` can be replaced by multiple statements enclosed in braces `{ }`.** Without the braces, only the *one* statement following the `if` (or `else`) is conditional — a second line at the same indentation runs unconditionally, whatever the indentation suggests. Always use the braces.

An `if` used as a guard:

```
if(digit < 10) {
    PORTA = seven_seg[digit];
} else {
    PORTA = 0;
}
```

[4] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[5] courses/csse2010/avr-instructions.pdf

Four input bits can hold 0–15, the array has ten entries, and the `if` is what keeps `seven_seg[digit]` inside it (c-pointers-and-arrays[6] — C does not check).

An `else if` chain is an `else` whose single statement is another `if`; nothing new is needed for it.

Repetition — `while`

```
while (expression)
    stmt
```

Test first; if true, run the body, then test again; stop when the test is false. If the test is false to begin with, the body never runs.

`while(1)` — a condition that is always true — is the shape of every embedded program:

```
int main(void) {
    /* set-up, once */
    DDRA = 0xFF;
    while(1) {
        /* read inputs, compute, write outputs - forever */
        digit = PINC & 0x0F;
        …
    }
}
```

On the AVR `main` has nowhere to return to. The program is *set-up, then loop forever*, and the loop body is one pass of reading inputs and writing outputs. In assembly this was a label and an unconditional `rjmp` back to it (avr-program-structure[7]); `while(1)` is the same jump with the label hidden.

Repetition — `for`

```
for (init; test; step)
    stmt
```

is exactly

[6] courses/csse2010/c-pointers-and-arrays.pdf
[7] courses/csse2010/avr-program-structure.pdf

```
init;
while (test) {
    stmt
    step;
}
```

— the `for` only gathers the three parts of a counted loop onto one line so they cannot drift apart. Two traversals of the array `a`:

```
for(index=0; index<10; index++) {     /* init: index=0; test: index<10; step: index++
 ↪  */
    a[index] = 0;
}

for(ptr=a; ptr < a+10; ptr++) {       /* same loop, counting in addresses */
    *ptr = 0;
}
```

The bound is the thing to get right. `index < 10` runs the body for 0 through 9, the ten legal indices of `a[10]`; `index <= 10` runs it once more, for `a[10]`, which does not exist. The pointer form has the same edge: `ptr < a+10` stops at the address one past the end without ever touching it.

Infinite loops, deliberately

The macro `#define EVER ;;` makes `for(EVER)` a spelling of `for(;;)` — a `for` with no init, no test and no step, which loops forever (c-preprocessor-and-avr-headers[8]). It is `while(1)` in a different costume; the course's programs use `while(1)`.

Other control statements

`switch`, `do … while`, `break` and `continue` are all real C. They are not in these notes yet.

See also

- c-language-basics[9] · c-operators[10] · c-pointers-and-arrays[11] · c-preprocessor-and-avr-headers[12]

[8] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[9] courses/csse2010/c-language-basics.pdf
[10] courses/csse2010/c-operators.pdf
[11] courses/csse2010/c-pointers-and-arrays.pdf
[12] courses/csse2010/c-preprocessor-and-avr-headers.pdf

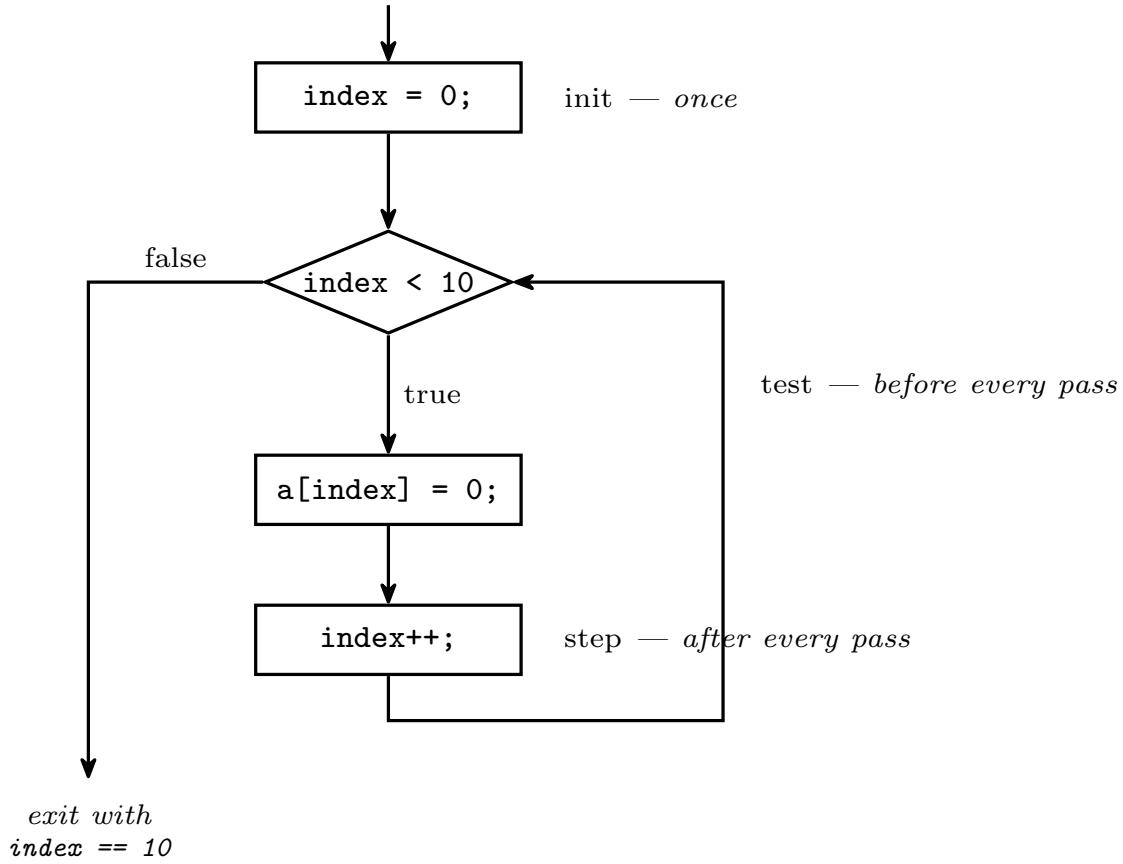


Figure 1: Where the three clauses of a `for` run, drawn on the repetition shape. *init* runs once, before anything; *test* runs before every pass, including the first; *step* runs after every pass, before the next test. The body of `for(index=0; index<10; index++)` therefore runs exactly ten times, with `index` taking 0 through 9 — and when the loop exits `index` is 10, the first value that failed the test. This is the `while` equivalence above, drawn.

- avr-program-structure[13] — labels and branches, the same shapes in assembly
- avr-instructions[14] — `cp`, `breq`, `brne`: what a test compiles to
- 2026-09-07-introduction-to-embedded-c-programming[15] — Lecture 13 · 2026-09-10-more-c-programming-on-the-avr[16] — Lecture 14

[13] courses/csse2010/avr-program-structure.pdf
[14] courses/csse2010/avr-instructions.pdf
[15] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[16] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf