C Language Basics

Table of contents

Where C sits 1
The shape of a program 2
Variables and types 2
Functions 3
Where a variable lives 6
Constants 6
Floating point on the AVR 8
See also 8

What a C program is made of: the shape of a program, variables and their types, functions, and constants. Operators are in c-operators[1], the `if`/`while`/`for` statements in c-control-flow[2], arrays and pointers in c-pointers-and-arrays[3], and the preprocessor in c-preprocessor-and-avr-headers[4]. First met in 2026-09-07-introduction-to-embedded-c-programming[5].

Where C sits

C is the top layer of the course's three blocks — *high-level programming* over *low-level programming* over *digital logic design* — and the course's C is **embedded C for the ATmega324A**, the same chip the assembly targeted. The difference from assembly is one extra tool. An assembler turns each line into exactly one instruction; a **compiler** turns each C statement into however many instructions it needs, chooses the registers, and hands the result to the assembler.

[1] courses/csse2010/c-operators.pdf
[2] courses/csse2010/c-control-flow.pdf
[3] courses/csse2010/c-pointers-and-arrays.pdf
[4] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[5] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf

The chain from C statement down to control unit is drawn in 2026-09-07-introduction-to-embedded-c-programming[6], and the steps after the compiler, object files and the linker, are in compilation-and-linking[7].

What is expected: write small C programs and functions from scratch, understand the meaning of C programs, and write or modify C programs — practised in Labs 11–16, which all use C.

The shape of a program

- There must be a function **called `main`**; it is executed when the program starts.
- Blocks of code are enclosed by **braces `{ }`**.
- Statements **end with a semicolon `;`**.
- C is **case sensitive** — `variable` and `Variable` are different names.
- Comments are `/* … */` (any length, may span lines) or `//` to the end of the line.

```c
/* the smallest useful embedded program: */
#include <avr/io.h>

int main(void) {
    DDRA = 0xFF;        /* a statement */
    while(1) {          /* a block */
        PORTA = 0x55;
    }
}
```

On the AVR `main` never returns — there is no operating system to return to — so every program ends in an infinite loop (c-control-flow[8]). `#include` is the preprocessor's (c-preprocessor-and-avr-headers[9]).

Variables and types

Declaration: `type-name variable-name, variable-name …;`

A declaration reserves room in data memory (memory-models[10]) and gives it a name and a type. The type decides two things: **how many cells** the variable takes, and **how the bits are read** — the signed/unsigned question of binary-number-representations[11], now answered once at the declaration instead of instruction by instruction.

[6] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[7] courses/csse2010/compilation-and-linking.pdf
[8] courses/csse2010/c-control-flow.pdf
[9] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[10] courses/csse2010/memory-models.pdf
[11] courses/csse2010/binary-number-representations.pdf

Declaration	Size	Range	
`char c;`	1 byte	−128 … 127	two's complement
`unsigned char x;`	1 byte	0 … 255	
`int day;`	**machine dependent** — 16 bits on the AVR	−32768 … 32767	two's complement
`unsigned int count;`	as `int`	0 … 65535 on the AVR	
`float`	32 bits	about $\pm 3.4 \times 10^{38}$, 7 digits	IEEE single precision; no AVR hardware — done in software
`double`	64 bits in general; 32 on avr-gcc		likewise

On avr-gcc `float` and `double` are interchangeable: both are 32 bits (below).

Three consequences worth carrying around:

- **`char` is an integer.** It is exactly the 8-bit two's complement byte of the hardware block; the name only promises it can hold a character code. Arithmetic on it is byte arithmetic, and `'A' + 1` is `'B'` because `'A'` is 65.
- **`int` is not one size.** Write code that assumes 16 bits and it breaks on a desktop; assume 32 and it breaks on the AVR. The AVR headers provide `uint8_t`, `int16_t` and friends — types with the width in the name — and the course's example program uses `uint8_t` for exactly this reason.
- **Nothing is checked.** Storing 300 in an `unsigned char` does not fail; it stores 300 mod 256, i.e. 44. (For a signed `char` the C standard leaves the result implementation-defined; gcc also gives 44.) The type is a reading rule, not a guard.

Booleans. There is no boolean type in the course's C. **Any value can be treated as a boolean: zero means false, non-zero means true.** This is what lets `if(a)` mean `if(a != 0)` (c-control-flow[12]).

Functions

```
return-type function-name (arg-decl)
{
    variable-declarations;
    …
```

[12] courses/csse2010/c-control-flow.pdf

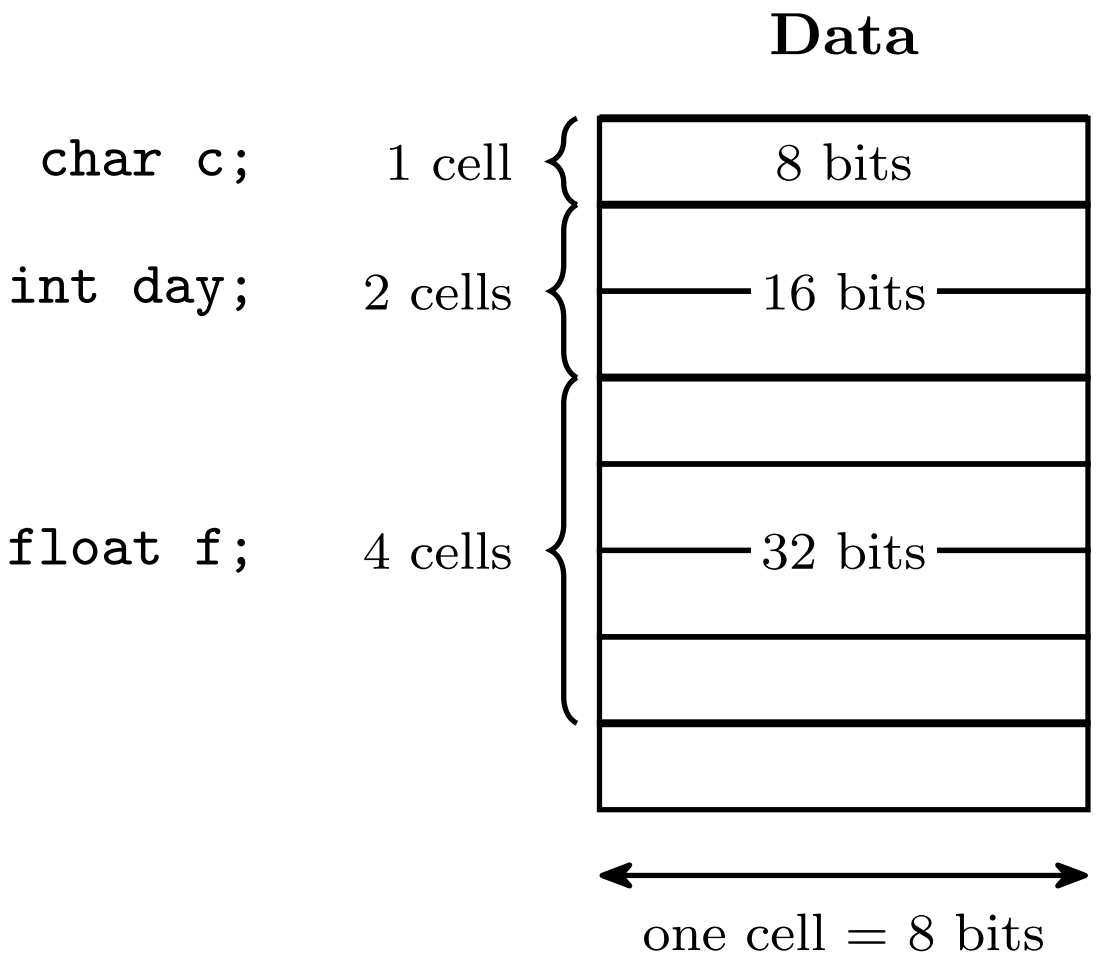


Figure 1: What three declarations reserve in the AVR's 8-bit data memory, drawn as a memory map. `char` is one cell. `int` is two, because `int` is 16 bits on the AVR — which is why a pointer to an `int` steps by two in the pointers note. `float` is four, because single precision is 32 bits. The compiler decides the actual addresses; the addresses here are schematic and the order is arbitrary.

```
    statements;
}
```

Part	Meaning	Example
`return-type`	what type is returned, or `void` for nothing	`int`, `void`
`function-name`	the name; `main` for the entry point	`average`
`arg-decl`	argument (parameter) declarations, comma-separated, each with its type	`int a, float b`
`variable-declarations`	variables used **only** within the function	`int avge;`
`statements`	the code which actually does stuff	

An example:

```
/*
  Return the average of two integers
  (result will be rounded towards 0)
*/
int average(int a, int b) {
    int avge;

    avge = (a+b)/2;

    return avge;
}
```

- `avge = (a+b)/2;` is a **statement** built from **expressions** — `(a+b)`, then `(a+b)/2`, then the assignment; the outermost expression plus its semicolon is the statement.
- `return avge;` hands the value back and ends the function.
- `avge` is **local**: it exists only while `average` is running. Nothing outside can name it.
- **Integer division truncates toward zero.** `average(1, 2)` computes `3/2` and returns `1`; the `.5` never exists. The comment says *rounded towards 0*, and that is the whole of C's rule for `/` on integers — `-3/2` is `-1`, not `-2`.

A function with `void` in front returns nothing and is called for its effect. `int main(void)` is a function whose argument list is explicitly empty.

Where a variable lives

Where a declaration is written decides where the compiler stores the variable and how long it exists.

Declaration	Kind	Stored	Exists
outside every function: `int count;`	**global**	at a fixed address in data memory, from `0x0100` up	for the whole run
inside a function: `int avge;`	**local**	in the call's **stack frame**	only while that call runs
inside a function: `static int calls;`	**`static` local**	with the globals, at a fixed address	for the whole run, keeping its value between calls
outside every function: `static int a;`	**`static` global**	like any global	for the whole run; the name is private to its file

So `avge` in `average` above is made fresh on every call and is gone at the `return`, and a local that is never assigned starts out holding whatever was left on the stack. A `static` local is the way to have a function remember something from one call to the next. Memory obtained from `malloc()` comes from a third place, the heap, which grows from above the globals toward the stack. The full picture is in compilation-and-linking[13], and the frame itself is in procedures-and-the-stack[14].

Constants

Character constants

Single quotes: `'a'`, `'b'`, `'1'`. A character constant is **a number** — its ASCII code — so `'1'` is 49, not 1.

Range	Codes
`'A'` … `'Z'`	65 … 90
`'a'` … `'z'`	97 … 122
`'0'` … `'9'`	48 … 57

[13] courses/csse2010/compilation-and-linking.pdf
[14] courses/csse2010/procedures-and-the-stack.pdf

Which makes `c - '0'` the digit value of a digit character, and `c + ('a' - 'A')` the lower-case of an upper-case letter — both plain integer arithmetic.

Special characters are **backslash escaped**:

`'\n'`	newline
`'\''`	single quote
`'\t'`	tab
`'\\'`	backslash

String constants

Double quotes, escapes allowed inside: `"abc \n \" hello\t"`. A string is an array of `char` (c-pointers-and-arrays[15]).

Integer constants

Base	Written	Examples
Decimal	as usual	`3`, `-27`, `65535`, `+5`
Hexadecimal	leading `0x`	`0x5F`, `0xFFFF`, `0xDEADBEEF`
Octal	**leading `0`**	`0377` = 255

The octal rule is a trap: **a leading zero changes the base**. `010` is eight. Do not zero-pad a decimal constant.

Hexadecimal is the natural notation for anything that is really a bit pattern — port values, masks — because each digit is exactly one nibble (binary-number-representations[16]). `0xFF` *is* `11111111` at a glance; `255` is not.

Floating point constants

Include a decimal point and/or an `e` exponent: `3.1416`, `-7.`, `6.02e23`, `-5.2e-2`. **`7` is an integer; `7.` is floating point.** The type of a constant is decided by how it is written, and it decides what arithmetic happens: `7/2` is `3`, `7./2` is `3.5`.

[15] courses/csse2010/c-pointers-and-arrays.pdf
[16] courses/csse2010/binary-number-representations.pdf

Floating point on the AVR

`float` and `double` are usually IEEE single and double precision, but the C standard only asks for minimums: about 6 significant decimal digits for `float` and 10 for `double`, both up to 10^{37}. **avr-gcc makes `double` the same 32-bit single precision as `float`**, which gives about 7 digits and so falls short of what the standard asks of `double`. Code written for a desktop that relies on `double` quietly loses half its digits on the AVR.

The AVR also has **no floating-point hardware**: its only hardware data type is the 8-bit integer (instruction-set-architecture[17]). Every `float` add, multiply or conversion is a call into a software library, hundreds of instructions with branches where an integer operation takes a few. So where integer arithmetic can give the same result, use it:

```
PORTB = PINA / 3;        /* integer: a handful of instructions, no branches */
PORTB = PINA * 0.334f;   /* float: the same results, 300+ instructions      */
```

Scaling to whole units (millivolts, tenths of a degree) or using fixed-point-numbers[18] keeps the fractions without the floats. The formats themselves, and why `0.1` cannot be stored exactly, are in floating-point-numbers[19].

See also

- c-operators[20] · c-control-flow[21] · c-pointers-and-arrays[22] · c-preprocessor-and-avr-headers[23]
- binary-number-representations[24] — the signed/unsigned readings behind `char` and `unsigned char`
- floating-point-numbers[25] — what `float` and `double` hold, and their cost on the AVR · fixed-point-numbers[26] — fractions in integer arithmetic
- memory-models[27] — the 8-bit data cells a declaration reserves
- instruction-set-architecture[28] — the data types the AVR supports in hardware, and the ones it does not

[17] courses/csse2010/instruction-set-architecture.pdf
[18] courses/csse2010/fixed-point-numbers.pdf
[19] courses/csse2010/floating-point-numbers.pdf
[20] courses/csse2010/c-operators.pdf
[21] courses/csse2010/c-control-flow.pdf
[22] courses/csse2010/c-pointers-and-arrays.pdf
[23] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[24] courses/csse2010/binary-number-representations.pdf
[25] courses/csse2010/floating-point-numbers.pdf
[26] courses/csse2010/fixed-point-numbers.pdf
[27] courses/csse2010/memory-models.pdf
[28] courses/csse2010/instruction-set-architecture.pdf

- compilation-and-linking[29] — how the compiler maps globals, locals and statics to memory, and what happens after compiling · procedures-and-the-stack[30] — the stack frame a local lives in
- 2026-09-07-introduction-to-embedded-c-programming[31] — Lecture 13 · 2026-10-15-compilation-and-linking[32] — Lecture 22 · 2026-10-19-floating-point-numbers[33] — Lecture 23

[29] courses/csse2010/compilation-and-linking.pdf
[30] courses/csse2010/procedures-and-the-stack.pdf
[31] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[32] courses/csse2010/2026-10-15-compilation-and-linking.pdf
[33] courses/csse2010/2026-10-19-floating-point-numbers.pdf