

Programming languages and introduction to C

How people describe computation
How computers execute it

Prof. Dr. Mehmet Göktürk · Gebze Technical University

Two perspectives on programming

Languages shape how we express a solution and communicate it to other people.

Translators and runtimes connect those descriptions to a particular computer.

Our C examples let us observe values, calculations and output directly.

Theory first, then a bounded C lesson. Additional worked examples follow as optional practice.

Why programming languages exist

A processor executes instructions defined by its architecture.

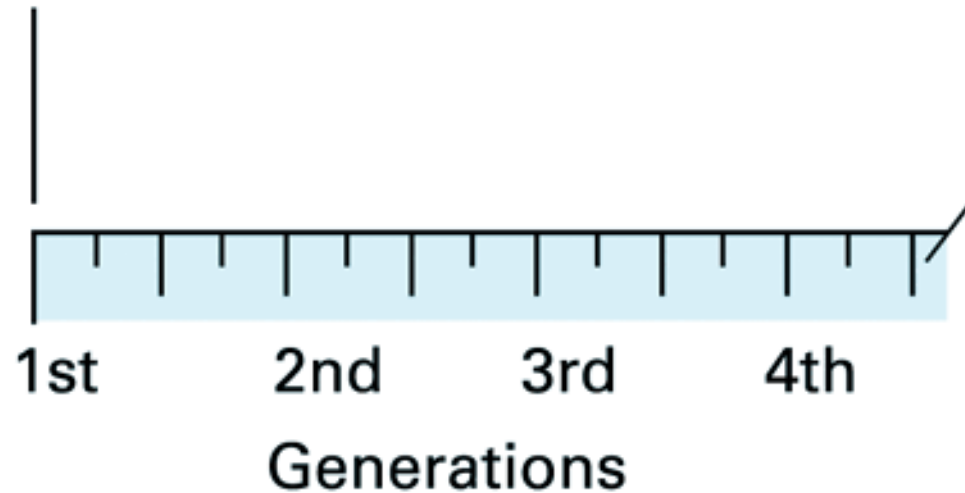
People need names, expressions and ways to organize a large solution.

A programming language defines precise rules for describing computation.

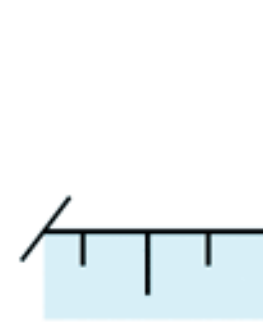
Abstraction lets a programmer work with a useful model while tools handle lower-level details.

Language abstraction

Problems solved in an environment
in which the human must conform
to the machine's characteristics



Problems solved in an environment
in which the machine conforms
to the human's characteristics



Generations describe a historical direction toward higher-level expression, not a ranking of every language.

Different levels of description

Level	What the author works with	Translation or execution
Machine language	Encoded instructions for a processor	Processor executes instructions
Assembly language	Mnemonics, registers and labels	Assembler produces machine code
High-level language	Variables, expressions and structured operations	Compiler and/or interpreter implements the language

One high-level operation can require many machine instructions. No new assembly programming is needed today.

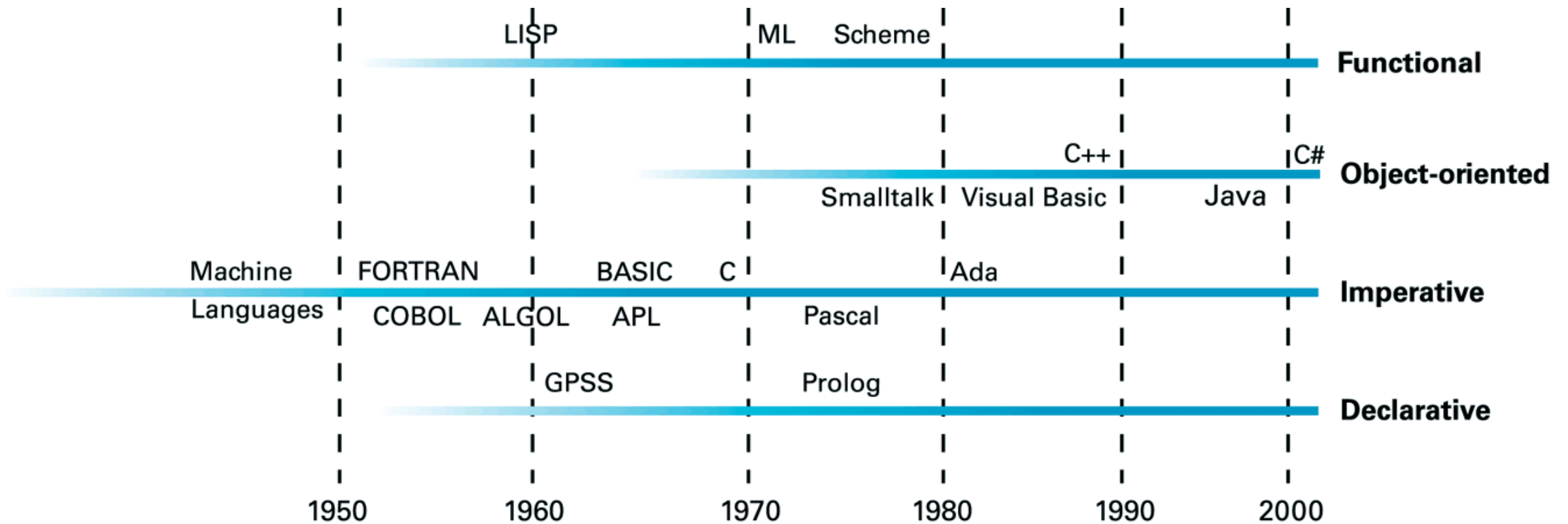
A paradigm is a way to organize a computational solution.

Imperative programming describes operations that change program state.

Other approaches emphasize expressions, interacting objects, or desired relationships.

A language can support more than one paradigm. These are overlapping approaches, not exclusive boxes.

A historical map of language families



Brookshear's historical chart ends around 2000. Read the overlapping approaches, not a current popularity ranking.

Several ways to describe the same task

Approach	Example: identify readings above a limit
Imperative	Visit each reading and add qualifying readings to a result.
Functional	Apply a filtering function to the collection.
Object-oriented	Ask a readings object for its above-limit readings.
Declarative	Specify the condition that the required readings satisfy.

These are descriptions of approaches, not four new language syntaxes to learn. A real program may combine them.

Syntax, meaning and intention

Question	Example	What it tells us
Is the form allowed?	$\text{total} = \text{price} + ;$	The expression is incomplete.
What does it mean?	$\text{total} = \text{price} + \text{tax};$	Evaluate the sum and assign its value.
Does it solve our problem?	$\text{total} = \text{price} - \text{tax};$	Legal form can still encode the wrong rule.

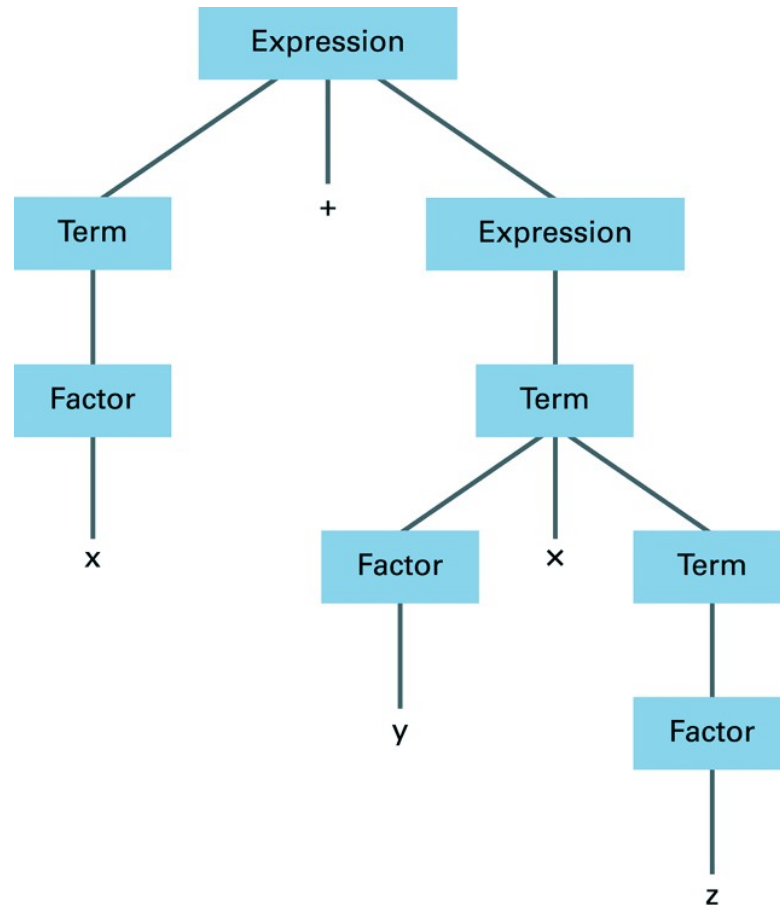
A translator checks language rules. Tests and reasoning must also check the intended task.

Recognizing tokens

Text in total = price + 12;	Role
total and price	Identifiers
=	Assignment operator
+	Addition operator
12	Integer literal
;	Statement terminator

Lexical analysis groups characters into meaningful tokens. The characters 1 and 2 form one literal here.

Parsing an expression



Tokens alone do not specify the grouping.

The grammar gives the expression a structure.

Parentheses can request a different grouping: $(x + y) \times z$ gives 20.

The tree represents $x + (y \times z)$. With $x = 2$, $y = 3$, $z = 4$, the result is 14.

Meaning also depends on types

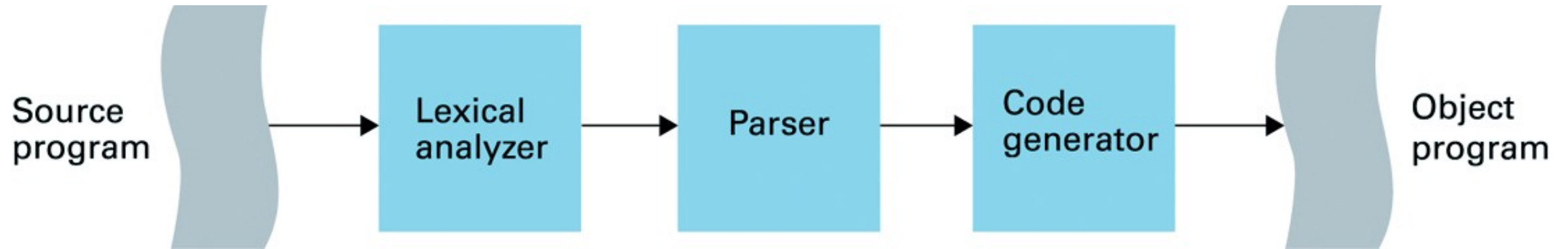
A name must refer to an appropriate declaration in its scope.

Types constrain which operations make sense and how expressions behave.

For example, C evaluates $7 / 2$ as integer division, producing 3.

Correct syntax and valid types still do not establish that the result meets a requirement.

Inside a translator



A simplified model: recognize tokens, determine structure, then generate target code.

A compiler maps the program's operations to a target representation.

An intermediate representation can separate language analysis from target-specific work.

Optimization can change the instruction sequence while preserving the required behavior.

The generated instruction count need not match the number of source statements.

Compilation and interpretation

Mechanism	Main action	What runs
Compilation	Translate a program into another representation	The resulting code runs later or within a runtime.
Interpretation	Carry out operations described by a program representation	An interpreter executes those operations.
Combined implementation	Translate first, then interpret or compile further	Several stages may contribute to execution.

These describe implementation mechanisms. A language name alone does not determine the entire execution path.

What an interpreter does

An interpreter is a running program that carries out another program's operations.

It works from source-derived structures or intermediate instructions such as bytecode.

It keeps track of values and control flow, and invokes the operations requested by the program.

“One source line at a time” is only a rough analogy. Parsing, bytecode and optimization can sit between source and execution.

A small interpreter trace

Toy program operation	Interpreter action	State or output
SET x 3	Associate x with the value 3.	x = 3
ADD x 2	Read x, add 2 and store the result.	x = 5
PRINT x	Read x and request output.	Print 5

Illustrative language, not C or a real bytecode listing. The interpreter itself runs as machine code.

Languages commonly compiled to native code

Language / family	Example application	Typical execution route
C, C++, Rust, Go	Systems and applications	Native compilation is a common deployment route.
Fortran	Scientific and numerical work	Compilers such as GNU Fortran produce native code.
COBOL	Business records and transactions	GnuCOBOL translates to C, then builds native code.
Pascal, Ada	Teaching / dependable systems	Free Pascal and GNAT illustrate native compilers.

Compilation can have several stages. A compiler may translate through another language before producing machine code.

Web, application and scripting languages

Language / family	Example application	Typical execution route
Java, C#	Applications and services	Java bytecode / .NET IL, then runtime execution, often with JIT compilation.
JavaScript	Web pages and web servers	Engines such as V8 combine bytecode interpretation with JIT compilation.
TypeScript	Larger JavaScript applications	Usually translates to JavaScript, which a JavaScript engine executes.
Lua	Embedded scripting and games	Standard Lua compiles to bytecode and interprets it in a virtual machine.

Bytecode compilation and interpretation can occur in the same system. JIT compilation produces machine code during execution.

Language / family	Example application	Typical execution route
Python, R	Data analysis and automation	Commonly called interpreted. CPython and R also use internal bytecode compilation.
Julia	Numerical computing	Usually JIT-compiler specialized code to native machine code using LLVM.
MATLAB	Engineering and matrix computing	Its interactive execution engine includes JIT compilation.
APL	Array-oriented computation	Dyalog is an APL interpreter. Other implementations may differ.

An interactive prompt describes how we use a system. It does not tell us whether the system compiles code internally.

Languages for data processing and analysis

Language / family	Example application	Typical execution route
SQL	Relational database queries	A database parses, plans and executes the query. Engines may also use compilation.
SAS language	Data preparation and statistics	SAS compiles and executes DATA steps. PROC steps invoke procedures.
SPSS command syntax	Statistical analysis	The SPSS runtime processes commands that request transformations and analyses.

SAS and SPSS are also software products. Here we mean their programming/command languages, alongside SQL as a query language.

BASIC, Forth and Lisp: implementation matters

Language family	Concrete example	What happens
BASIC	GW-BASIC / FreeBASIC	GW-BASIC interprets programs. FreeBASIC compiles programs into executables.
Forth	Gforth	The text interpreter supports interpretation and compilation states. Compiled definitions can then execute.
Lisp	Common Lisp with SBCL	SBCL normally compiles to native code, including interactive evaluation. It also offers an interpreter.

A language can have several implementations. An interactive prompt does not prove that execution uses only interpretation.

Python is commonly described as interpreted

Python's own documentation calls it an interpreted language.

In ordinary CPython, source first becomes Python bytecode. The bytecode interpreter then executes it.

The command `python program.py` performs these steps for the user. A separate native executable build is usually unnecessary.

Both descriptions are accurate: Python is commonly called interpreted, and CPython uses internal bytecode compilation.

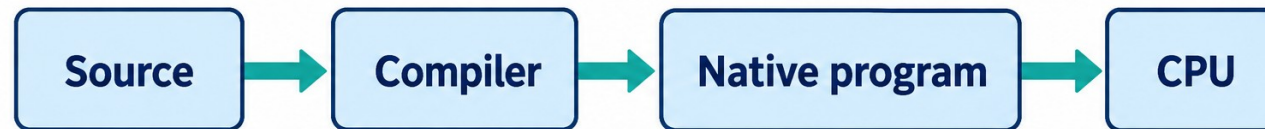
What happens after intermediate compilation?

Example	Intermediate form	Typical execution
Java on a JVM	JVM bytecode	Runtime may interpret and/or JIT-compile.
C# on .NET	Common Intermediate Language (CIL)	Runtime commonly JIT-compiles to native code.
Python on CPython	Python bytecode	Bytecode interpreter executes operations.

JIT means just-in-time compilation during execution. Other configurations, including ahead-of-time compilation, exist.

Two execution paths

Ahead-of-time compilation



Bytecode execution



The runtime may interpret bytecode or compile it to native code.

Both paths ultimately cause the physical CPU to execute machine instructions.

Bytecode targets an abstract execution model rather than one physical processor.

A compatible runtime supplies the operations that model requires.

CPython compiles Python source into bytecode for its interpreter.

A runtime can combine interpretation with just-in-time compilation. Compatibility still depends on the runtime and its version.

Translation, linking and loading



Translation produces code. Linking connects components. Loading prepares the program for execution.

Libraries and the tools around a language

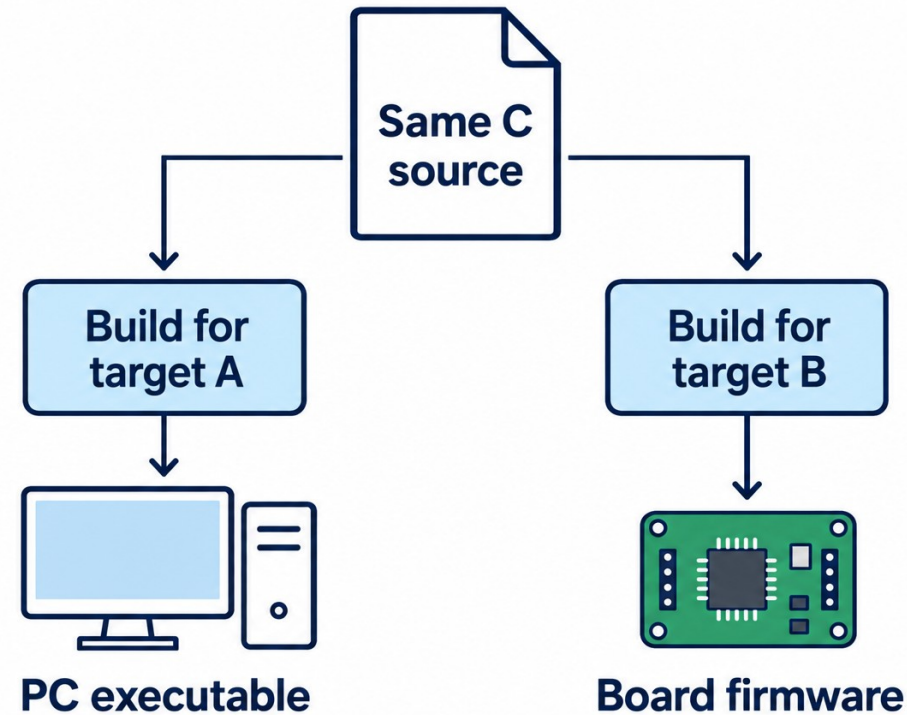
A language defines constructs such as expressions, statements and types.

Libraries supply reusable operations such as formatted output.

A build system coordinates the tools and dependencies that make an executable.

An IDE brings editing, building and debugging together. It is one part of the development environment.

Source portability and executable compatibility



Source portability does not mean the same executable.

Target hardware, libraries and interfaces must match.

Portable source still needs suitable language support, libraries and interfaces on each target.

What can change when the target changes?

Dependency	Example	Consequence
Processor and executable format	Different instruction sets or operating systems	Build a suitable executable.
Data representation	The width of int can differ	Check assumptions about ranges and sizes.
Libraries and interfaces	PC terminal output versus board I/O	Adapt the interface to the environment.
Runtime support	A required virtual machine or library version	Provide a compatible runtime.

Source portability requires deliberate choices and testing. Recompilation alone does not repair every dependency.

Arduino as a cross-compilation example

You edit a sketch on the PC and choose the target board.

The Arduino build process prepares C++ source and includes board support.

A compiler and linker produce firmware for that target, then upload tools transfer it.

The board executes the firmware. The PC used to build it need not have the same processor.

Choosing a language and its tools

Question	Why it matters
What problem are we describing?	Numeric work, device control and data queries need different abstractions.
Where must the program run?	Hardware, operating system and runtime constrain deployment.
What must the team maintain?	Clear code, available libraries and tooling affect long-term work.
What resources are limited?	Memory, timing, power and development effort involve tradeoffs.

There is no single ranking that answers every engineering problem. Compare the requirements.

Theory check: three situations

A source file works on one computer. Must its executable run on every other computer?

A language implementation compiles to bytecode. Can it also use an interpreter?

A program builds successfully but subtracts tax when it should add it. Has translation proved correctness?

Explain the mechanism or missing assumption in each case.

Theory check: explanations

An executable depends on its target. Even source may need adaptation for different libraries or interfaces.

Yes. Translation to bytecode and interpretation of that bytecode can occur in one implementation.

No. A valid program can express the wrong rule for the intended task.

Language rules, execution mechanisms and problem correctness answer different questions.

Brookshear and Brylow, Computer Science: An Overview, 12th Global Edition: Chapter 6, especially 6.1 and 6.4.

Brookshear's local 7th-edition chapter slides provide the historical and translation figures.

GCC documentation explains build stages. Python and JVM documentation illustrate runtime mechanisms.

Arduino's sketch build documentation explains the PC-to-board compilation process.

A first working program

Use predictions and test cases to connect language rules with observed results.

Programs that calculate and control



We begin with calculations on the PC. Arduino connects these ideas to physical behavior.

Why C in this course?

Variables and types make representation choices explicit.

Expressions turn a mathematical rule into an executable calculation.

Compiler messages give evidence about mistakes in a program.

These fundamentals carry into Arduino's C++ environment.

Our first complete program

```
#include <stdio.h>

int main(void) {
    printf("Hello, C!\n");
    return 0;
}
```

Save this plain-text file as hello.c. Expected output: Hello, C!

Reading the program

Part	Purpose
<code>#include <stdio.h></code>	Makes declarations such as printf available.
<code>int main(void)</code>	Program entry function; no parameters here.
<code>{ ... }</code>	Groups the statements in the function body.
<code>printf("Hello, C!\n");</code>	Calls a library function to produce output.
<code>return 0;</code>	Reports successful completion to the host.

A header supplies declarations; the library supplies the implementation.

Build and run in the terminal

```
gcc -std=c17 -Wall -Wextra -Wconversion hello.c -o hello  
./hello
```

Hello, C!

First command: build hello. Second command: execute it from this directory.

The edit–build–run cycle

Save the change in hello.c.

Build again and read the compiler messages.

Run the newly built executable only after a successful build.

If an old executable remains after a failed build, it still contains old code.

Statements, comments and layout

```
int count = 3;                // initial number of items
count = count + 1;

/* Explain the reason for a calculation. */
printf("count = %d\n", count);
```

Semicolons end these statements. Indentation helps us read their structure.

A variable names a stored value

`int count = 3;` declares count, gives it type int and initializes it to 3.

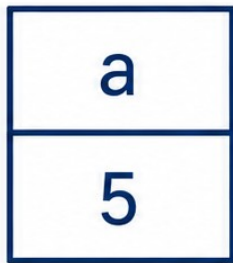
The name lets later statements refer to that object.

An assignment can replace its current value.

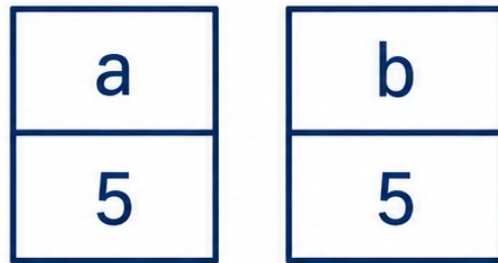
Initialize a local variable before reading its value.

Assignment copies a value

```
int a = 5;
```



```
int b = a;
```



```
a = 9;
```



Assignment copies a value.

After a changes to 9, b still holds the copied value 5.

Assignment is an update

```
int count = 3;  
count = count + 1;  
printf("%d\n", count);
```

Read the old count, compute $3 + 1$, then store 4 in count.

Types describe the values we use

Type	Example declaration	Useful for
int	int count = 12;	Whole-number counts
double	double temperatureC = 23.5;	Fractional calculations
float	float gain = 0.5f;	Floating-point with a different precision/range
char	char grade = 'A';	A small integer type used for character codes

Choose a type for the quantity and its required range, not just its current value.

A type has limits

A fixed-width integer type can represent only a finite range.

Do not rely on signed integer overflow wrapping around in C.

Floating-point values also have finite range and precision.

An expression can overflow before its result reaches a larger destination.

Arithmetic expressions

Operator	Example	Result
+	$7 + 3$	10
-	$7 - 3$	4
*	$7 * 3$	21
/	$7 / 3$	2 with integer operands
%	$7 \% 3$	1, the integer remainder

Here all operands are int values. Division by zero is invalid.

Parentheses express the intended grouping

Expression	Evaluation	Result
$2 + 3 * 4$	$2 + 12$	14
$(2 + 3) * 4$	$5 * 4$	20
$20 / 5 * 2$	$(20 / 5) * 2$	8
$20 / (5 * 2)$	$20 / 10$	2

Precedence determines grouping. Use parentheses where they clarify the formula.

The operands determine division

Expression	Kind of division	Result
$7 / 2$	Integer	3
$7.0 / 2$	Floating-point	3.5
$7 / 2.0$	Floating-point	3.5
$-7 / 2$	Integer, toward zero	-3

Integer division discards the fractional part; it does not round to nearest.

A double destination cannot recover a fraction

```
double a = 7 / 2;  
double b = 7.0 / 2;  
  
printf("%.1f %.1f\n", a, b);
```

Output: 3.0 3.5. The right-hand expression is evaluated before assignment.

Temperature conversion: define the problem

Given a Fahrenheit temperature F , calculate its Celsius equivalent.

Use $C = (F - 32) \times 5 / 9$.

Check familiar values: 32°F gives 0°C ; 212°F gives 100°C .

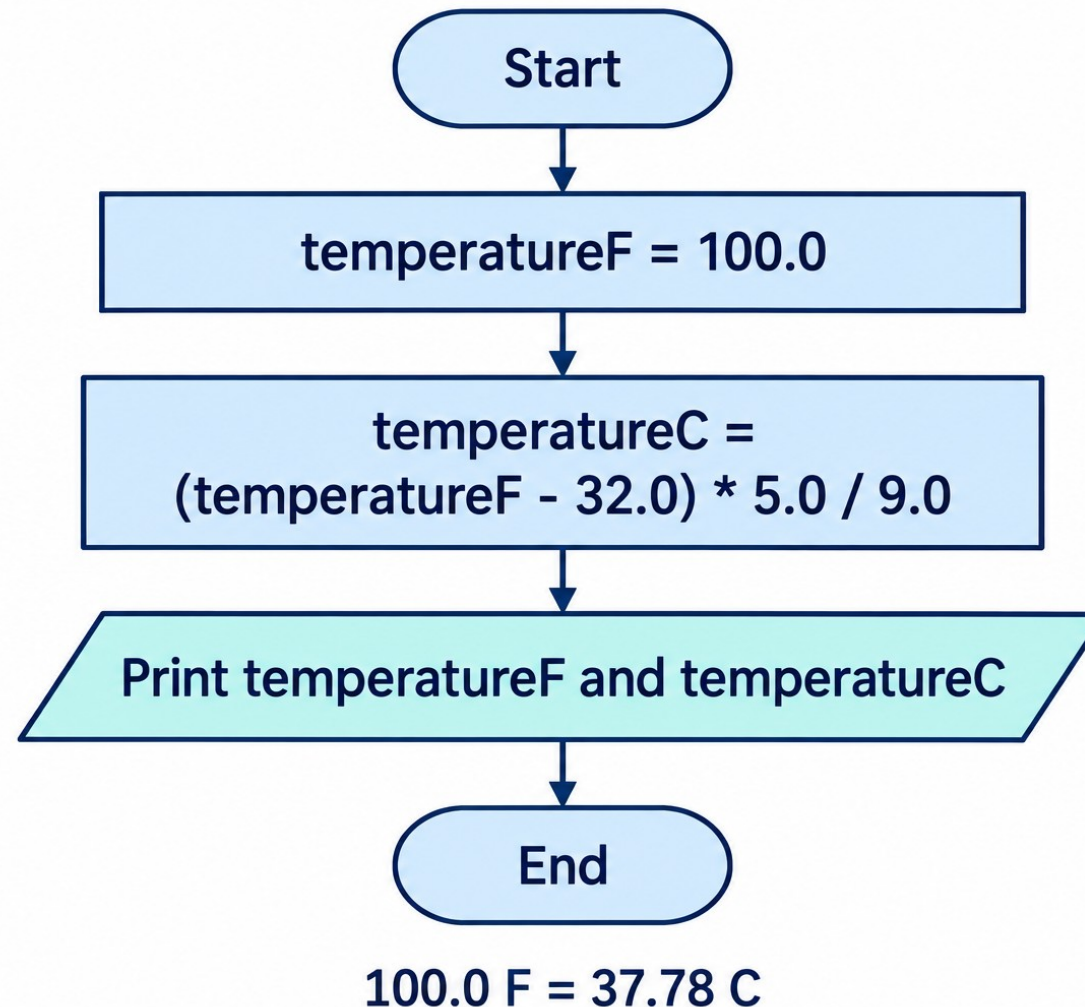
Keep units in names and labels so the result has a clear meaning.

One formula, different C expressions

At F = 100	What C computes	Result
$(5 / 9) * (F - 32)$	$0 * 68$	0
$5 * (F - 32) / 9$	340 / 9 with integers	37
$(F - 32.0) * 5.0 / 9.0$	Floating-point arithmetic	About 37.7778

Changing the destination type alone does not change the operand types.

Temperature conversion flowchart



Each arrow leads to the next operation. The calculation finishes before output.

A complete temperature calculation

```
#include <stdio.h>

int main(void) {
    double temperatureF = 100.0;
    double temperatureC =
        (temperatureF - 32.0) * 5.0 / 9.0;
    printf("%.1f F = %.2f C\n",
           temperatureF, temperatureC);
    return 0;
}
```

Expected output: 100.0 F = 37.78 C

printf matches a format to each argument

Conversion	Argument used here	Example
%d	int	printf("%d", count);
%f	double	printf("%f", temperatureC);
%.2f	double, two fractional digits	printf("%.2f", temperatureC);
%c	Character code promoted to int	printf("%c", grade);
%zu	size_t from sizeof	printf("%zu", sizeof(int));

Match the argument type and the conversion. A mismatch is not a conversion request.

Floating-point is an approximation

Many decimal fractions do not have an exact finite binary representation.

A calculation can accumulate small representation and rounding errors.

More printed digits do not create more accurate input data.

For this exercise, compare expected results to the requested display precision.

Test cases for the temperature calculation

Input F	Expected C	What it checks
32	0	Offset at a familiar point
212	100	Scale at another familiar point
-40	-40	Negative input
100	About 37.78	Fractional result
50	10	Ordinary exact reference value

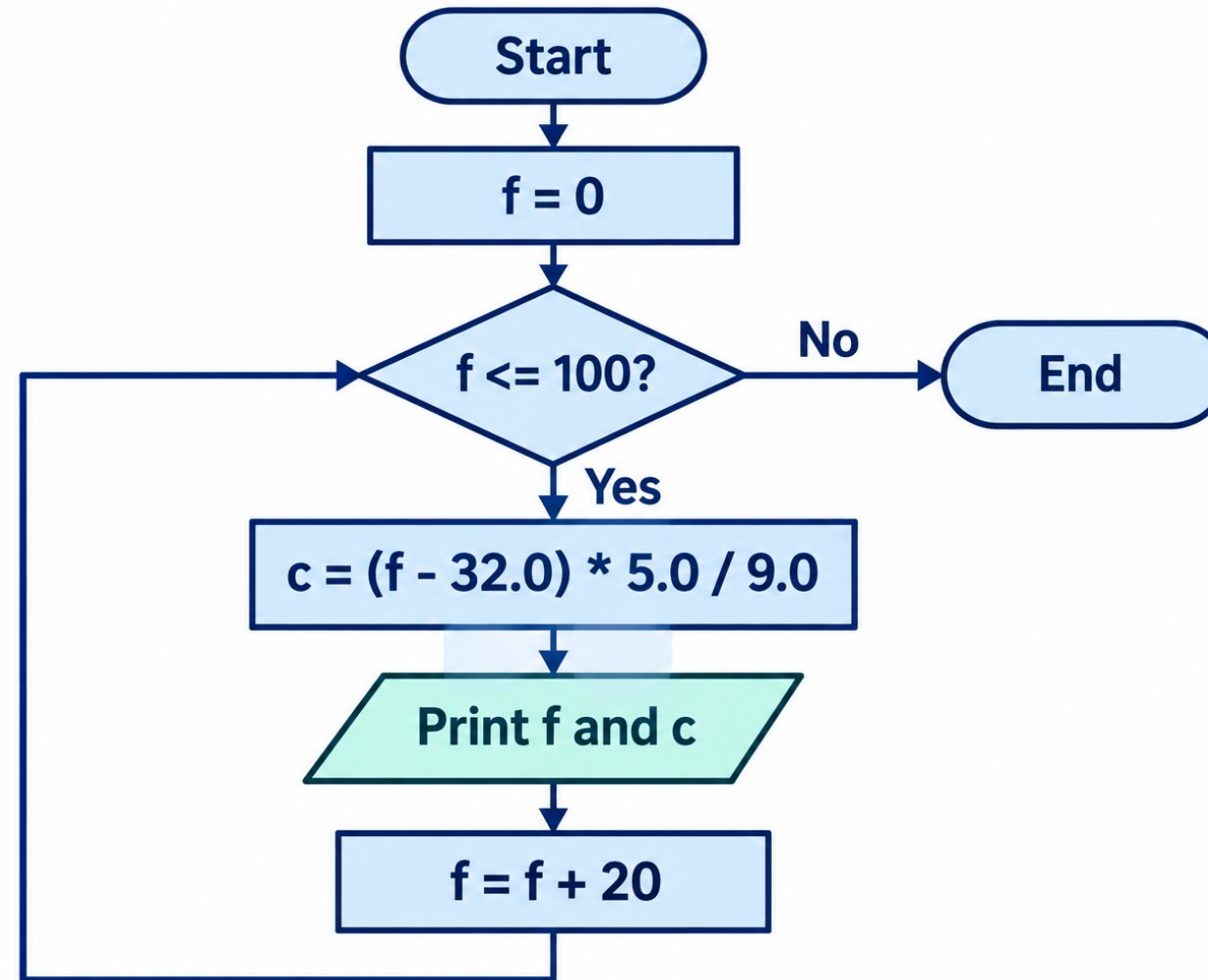
Check more than one input. A wrong formula may pass a convenient test.

Different failures need different evidence

Symptom	Possible cause	First check
Build reports an error	Syntax or declaration mistake	Read the first diagnostic and source line.
Link fails	Missing implementation or wrong symbol	Check function name and build inputs.
Build succeeds, result wrong	Formula or type mistake	Predict intermediate values.
Output stays old after an edit	Old executable was run	Save and confirm the build succeeded.

Warnings are useful evidence. A successful build does not prove the answer is correct.

Temperature-table flowchart



The update returns to the condition. Initialization happens only once.

A supplied loop prints a short table

```
#include <stdio.h>

int main(void) {
    int f = 0;
    while (f <= 100) {
        double c = (f - 32.0) * 5.0 / 9.0;
        printf("%3d %7.2f\n", f, c);
        f = f + 20;
    }
    return 0;
}
```

Trace and modify this supplied structure. Today's focus is the calculation and output.

Trace the supplied repetition

f before test	f <= 100	Printed Celsius	f after update
0	True	-17.78	20
20	True	-6.67	40
...	...	...	...
100	True	37.78	120
120	False	No row	120

Six rows, seven tests. Initialization runs once; the update makes progress.

Ready for recitation and Lab 4

Build, edit and rerun a small C calculation.

Explain integer division, conversion and the output format.

Use a supplied loop without losing the calculation's meaning.

Bring a predicted result and a trace when asking about a bug.

Final check: predict all three results

```
int a = 9;  
int b = 4;  
  
printf("%d\n", a / b);  
printf("%.2f\n", (double) a / b);  
printf("%d\n", a % b);
```

Explain each result using the operand types and the meaning of the operator.

Final check: answers

Expression	Result	Reason
a / b	2	Integer division discards the fraction.
$(\text{double}) a / b$	2.25	Conversion occurs before division.
$a \% b$	1	$9 = 4 \times 2 + 1$.

Next: conditions, loops and Arduino I/O, building on these values and expressions.

Additional practice

Duration calculations, extra traces and Arduino Blink
Use selected examples as needed.

Text and escape sequences

C text	Printed effect
"Hello\n"	Hello, then a new line
"A\tB\n"	A, a tab, B, then a new line
"She said \"Hi\"\n"	She said "Hi", then a new line
"C:\\data\n"	C:\data, then a new line

The backslash begins an escape sequence inside the string.

Trace assignments in order

Statement completed	a	b
int a = 5;	5	Not declared
int b = a;	5	5
a = 9;	9	5
b = b + 2;	9	7

Predict first: what would `printf("%d %d\n", a, b)` print? Answer: 9 7.

Names that explain the calculation

Use names such as totalSeconds, temperatureC and itemCount.

Names are case-sensitive: count and Count differ.

Start beginner names with a letter; use letters, digits and underscores.

Write one clear step at a time rather than compressing several updates.

Measure the target, do not guess

```
#include <stdio.h>
#include <limits.h>

int main(void) {
    printf("int: %zu bytes\n", sizeof(int));
    printf("double: %zu bytes\n", sizeof(double));
    printf("bits per byte: %d\n", CHAR_BIT);
    printf("int range: %d to %d\n", INT_MIN, INT_MAX);
    return 0;
}
```

The output describes this compiler and target. C does not require a four-byte int.

A character and its numeric code

```
#include <stdio.h>

int main(void) {
    char grade = 'A';
    printf("grade = %c\n", grade);
    printf("code = %d\n", grade);
    return 0;
}
```

On an ASCII-compatible target: grade = A, code = 65.

Convert before the division

```
int total = 7;  
int count = 2;  
  
double wrong = total / count;  
double mean = (double) total / count;
```

wrong is 3.0; mean is 3.5. Here count is known to be nonzero.

Width and precision control presentation

Format for 37.777...	Meaning	Visible output
%f	Default six fractional digits	37.777778
%.2f	Two fractional digits	37.78
%8.2f	At least eight character positions	37.78
%.0f	No fractional digits printed	38

Formatting changes the displayed text, not the stored variable.

Printed precision is not stored precision

```
double x = 37.7777777778;  
printf("%.2f\n", x);  
printf("%.6f\n", x);
```

The first line is 37.78; the second is 37.777778. x did not change.

Names for quantities that stay fixed

```
const double serviceFee = 2.50;  
int itemCount = 3;  
double unitPrice = 4.00;  
double total = itemCount * unitPrice + serviceFee;
```

total is 14.50. const prevents assignment through the name serviceFee.

A small design before coding

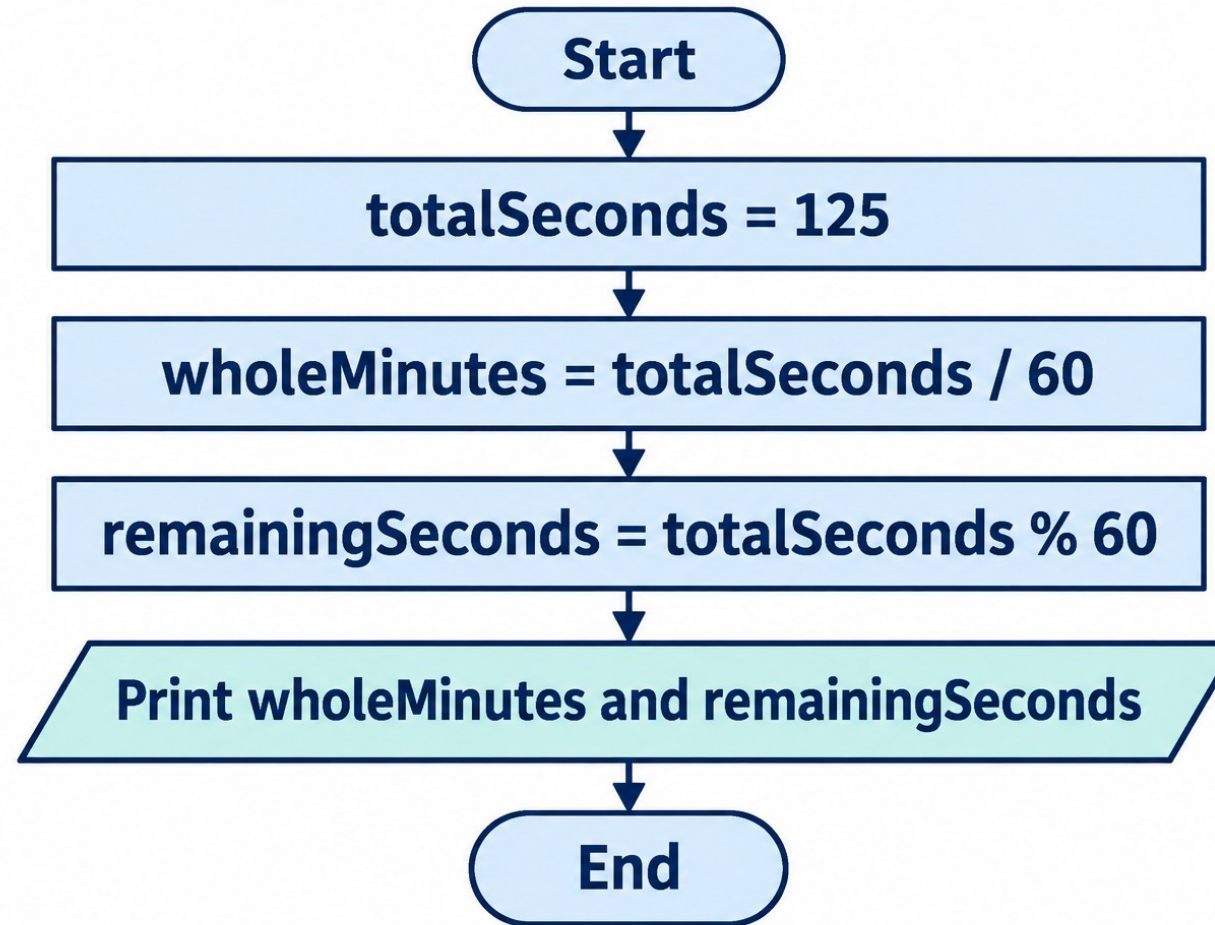
Problem: convert a duration in seconds into whole minutes and leftover seconds.

Input: a nonnegative whole-number total, stored in totalSeconds.

Outputs: wholeMinutes and remainingSeconds.

Example: 125 seconds should become 2 minutes and 5 seconds.

Minutes and seconds flowchart



Integer division: 2 minutes; remainder: 5 seconds

Division gives whole minutes; remainder gives the seconds left over.

Quotient and remainder solve the problem

```
#include <stdio.h>

int main(void) {
    int totalSeconds = 125;
    int wholeMinutes = totalSeconds / 60;
    int remainingSeconds = totalSeconds % 60;
    printf("%d min %d s\n",
           wholeMinutes, remainingSeconds);
    return 0;
}
```

Expected output: 2 min 5 s

Boundary cases explain the design

Seconds	Minutes	Remainder
0	0	0
59	0	59
60	1	0
61	1	1
125	2	5

For these nonnegative inputs: $\text{seconds} = 60 \times \text{minutes} + \text{remainder}$, with $0 \leq \text{remainder} < 60$.

Find the lost fraction

```
int total = 17;  
int samples = 5;  
double average = total / samples;  
printf("%.2f\n", average);
```

Predict the output, explain the cause, then change one expression.

Repair the expression

```
double average = (double) total / samples;  
printf("%.2f\n", average);
```

With total = 17 and samples = 5, output is 3.40.

Read a diagnostic, then inspect the code

```
int count = 3  
printf("%d\n", count);
```

A semicolon is missing after 3. A compiler may point at the next line.

Practice with the supplied program

Change the step from 20 to 25. Predict the rows before running.

Print three digits after the decimal point.

Change the formula to convert Celsius to Fahrenheit: $F = C \times 9 / 5 + 32$.

Keep floating-point operands and rename the output labels to match units.

Expected results after modification

Change	Expected result
Step 25, original F-to-C formula	F values: 0, 25, 50, 75, 100
Print %.3f, F = 100	37.778
Reverse conversion, C = 0	32.00 F
Reverse conversion, C = 100	212.00 F

Use $\text{temperatureF} = \text{temperatureC} * 9.0 / 5.0 + 32.0;$

The same fundamentals on Arduino

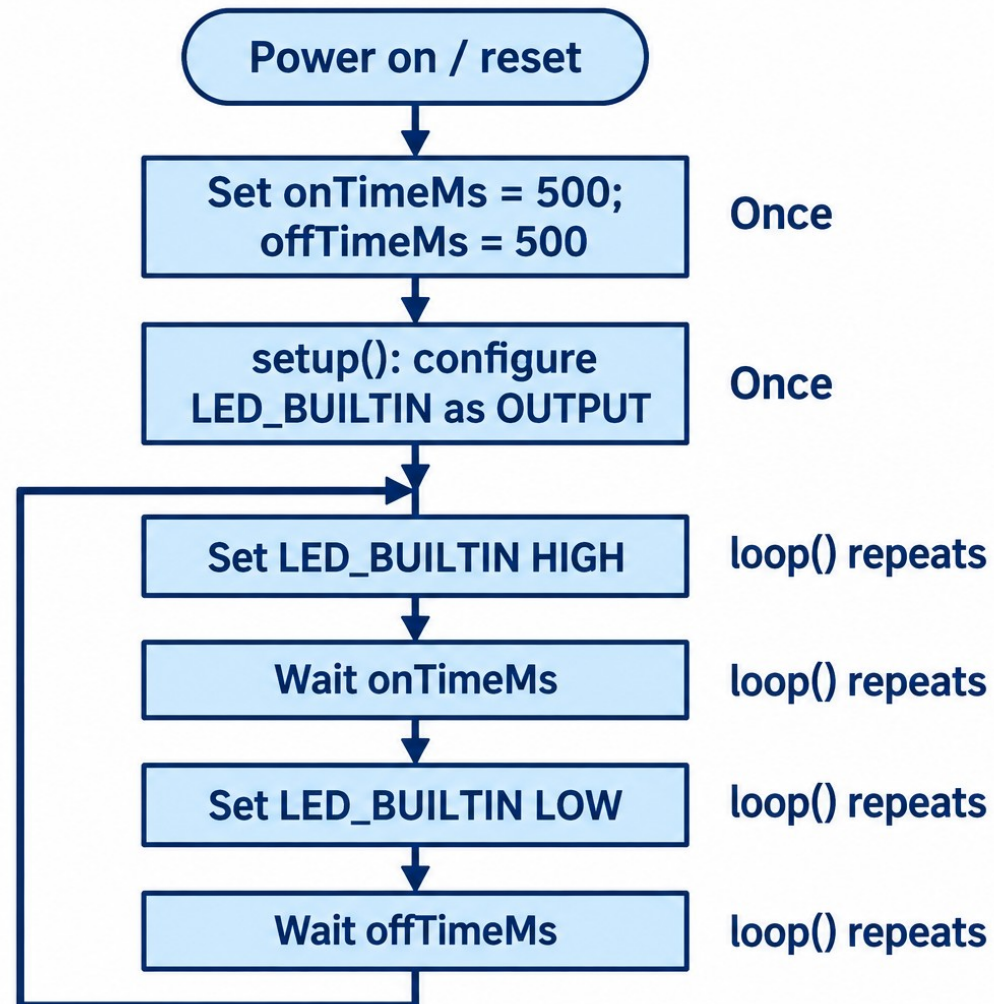
An Arduino sketch uses C++ with Arduino libraries and a board-specific build.

Variables, arithmetic, assignments and control-flow ideas remain familiar.

`setup()` performs initialization; `loop()` is called repeatedly by the framework.

Select the correct board and port before upload.

Arduino Blink flowchart



setup() runs once after reset; the four actions in loop() repeat.

Preview: controlling the built-in LED

```
const unsigned long onTimeMs = 500;
const unsigned long offTimeMs = 500;

void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(onTimeMs);
    digitalWrite(LED_BUILTIN, LOW);
    delay(offTimeMs);
}
```

Arduino sketch, not a PC C program. Uses the selected board's LED_BUILTIN definition.

A timing change is a design choice

On time	Off time	Nominal cycle	Interpretation
500 ms	500 ms	1000 ms	Equal on and off intervals
100 ms	900 ms	1000 ms	Short flash, same nominal period
1000 ms	1000 ms	2000 ms	Slower blinking

Call overhead adds a little time. delay pauses normal sketch execution while waiting.

Design, predict, test, explain

State the required output, units and input assumptions.

Choose types and write the calculation clearly.

Predict ordinary and boundary cases before running.

Change one thing, compare results, and explain any difference.

Instructor Set 4: Hello World, variables and temperature conversion.

GNU C manual: types, expressions and division; GCC manual: build stages.

Harvard CS50 Lecture 1: another introduction to C and common errors.

Arduino documentation: sketches and the built-in Blink example.