

Software engineering, functions and arrays

Designing software people can rely on
Requirements, testing and modular programs

Prof. Dr. Mehmet Göktürk · Gebze Technical University

A program lives inside a human activity



A student needs to borrow a book.

A librarian applies lending rules.

The system must record the right outcome.

Library example: understanding the work comes before selecting the software.

Software engineering organizes the development and evolution of software systems.

The work includes understanding needs, designing components and gathering evidence that the system behaves correctly.

A working program also needs understandable documentation and a manageable path for change.

Why software becomes difficult

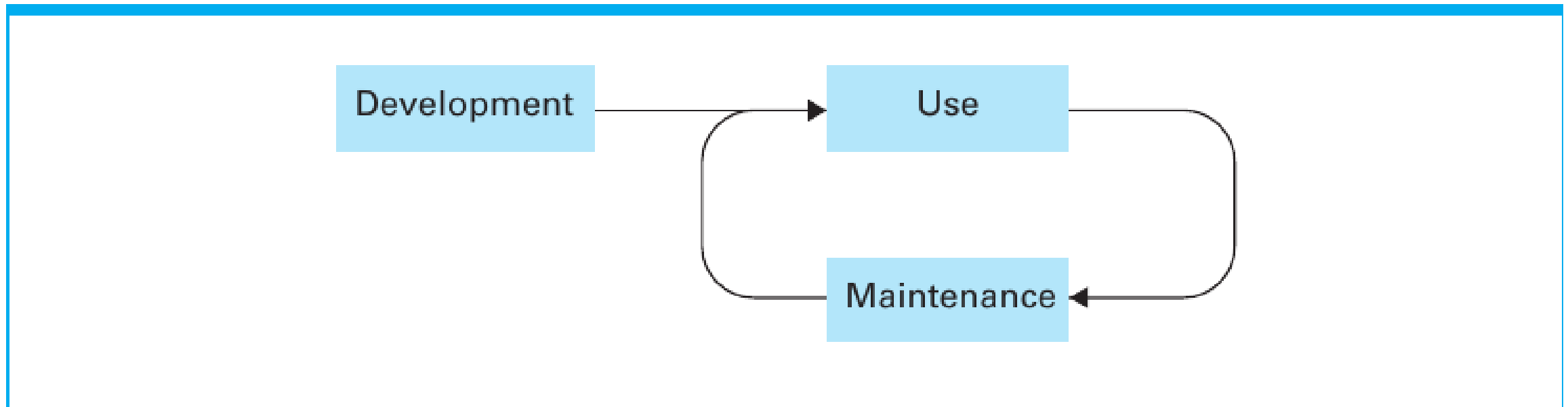
Many people depend on the same system, with different needs.

Parts interact through shared data and interfaces. A local change can affect distant behavior.

Requirements and operating environments change while the software remains in use.

The software life cycle

Figure 7.1 The software life cycle



Development establishes an initial system.

Use reveals faults and new needs.

Maintenance changes the system throughout its life.

Brookshear & Brylow, 12th Global Edition, Fig. 7.1, p. 335.

Stakeholders and their needs

Students want a clear borrowing result and an accurate due date.

Librarians need correct records and a way to handle exceptions.

Administrators need enforceable rules. Maintainers need a system they can understand.

Requirements describe observable behavior

Vague request	A testable statement
“Handle loans properly.”	Allow a loan only for an active member with fewer than 3 current loans.
“Explain a rejection.”	If the member already has 3 loans, show the reason and leave the records unchanged.
“Make it fast.”	Example target: show a result within 2 seconds under a defined test workload.

These are example requirements, not the university library’s actual policies.

A functional requirement specifies behavior: record a successful loan and its due date.

A quality requirement constrains that behavior: response time, accessibility or reliability.

“Reliable” becomes useful only when we specify what failures to tolerate and how to check it.

A loan request as a scenario

Situation	Expected behavior
Active member, 2 loans, book available	Approve and record a third loan.
Active member, 3 loans, book available	Reject. Keep the existing loans unchanged.
Inactive member, book available	Reject regardless of the number of loans.
Active member, 0 loans, book unavailable	Reject without creating a loan.

A scenario exposes missing rules before implementation begins.

Ambiguity can survive perfect execution

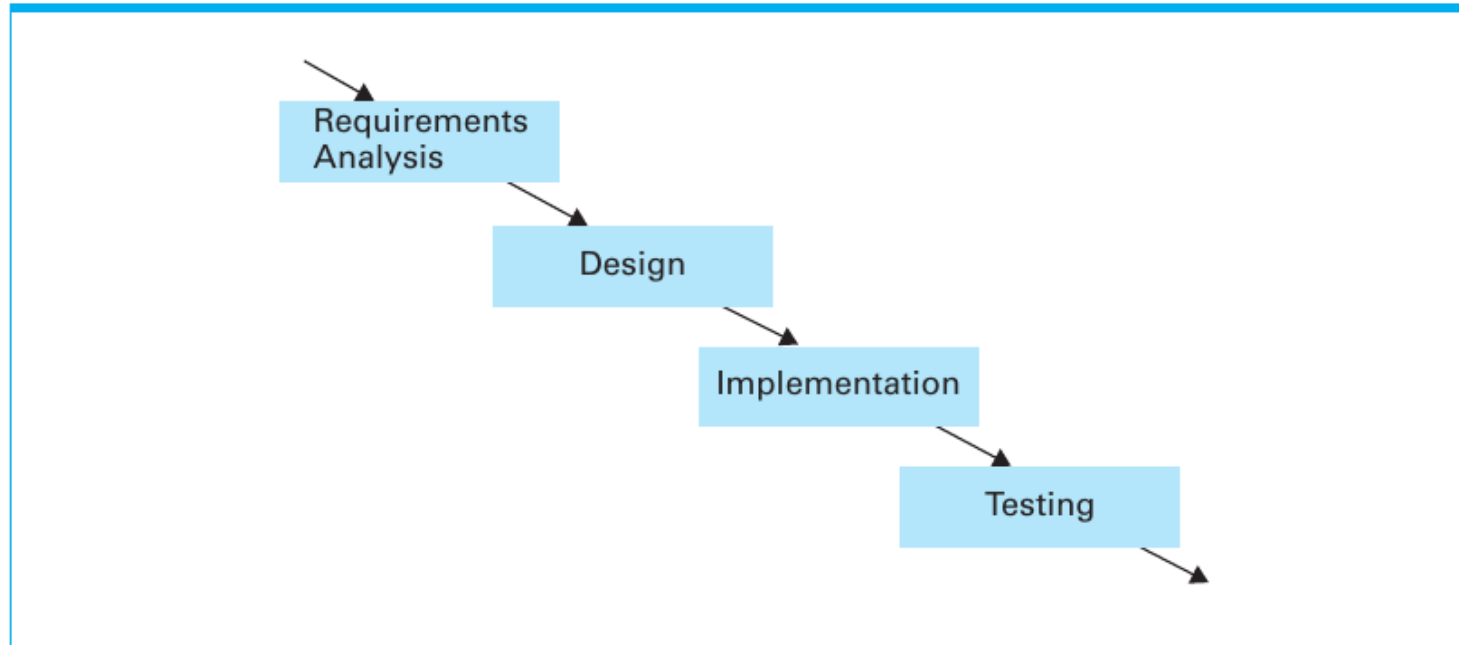
“Count button presses” can mean several different things.

Six observations are 0, 1, 1, 0, 1, 0. Three samples show the pressed state.

There are two observed 0-to-1 transitions. Counting events needs a different rule and adequate sampling.

Activities in software development

Figure 7.2 The traditional development phase of the software life cycle



Requirements describe the intended behavior.

Design organizes the solution.

Implementation and testing provide a working result and evidence.

Brookshear & Brylow, Fig. 7.2, p. 336. A traditional staged model.

Iteration and incremental development

Iteration revisits a solution using feedback. A prototype may reveal an unclear rule.

An increment adds a usable part: first loans, then returns, then renewals.

Small changes make review and testing more manageable. They still need requirements and documentation.

A prototype tests an assumption

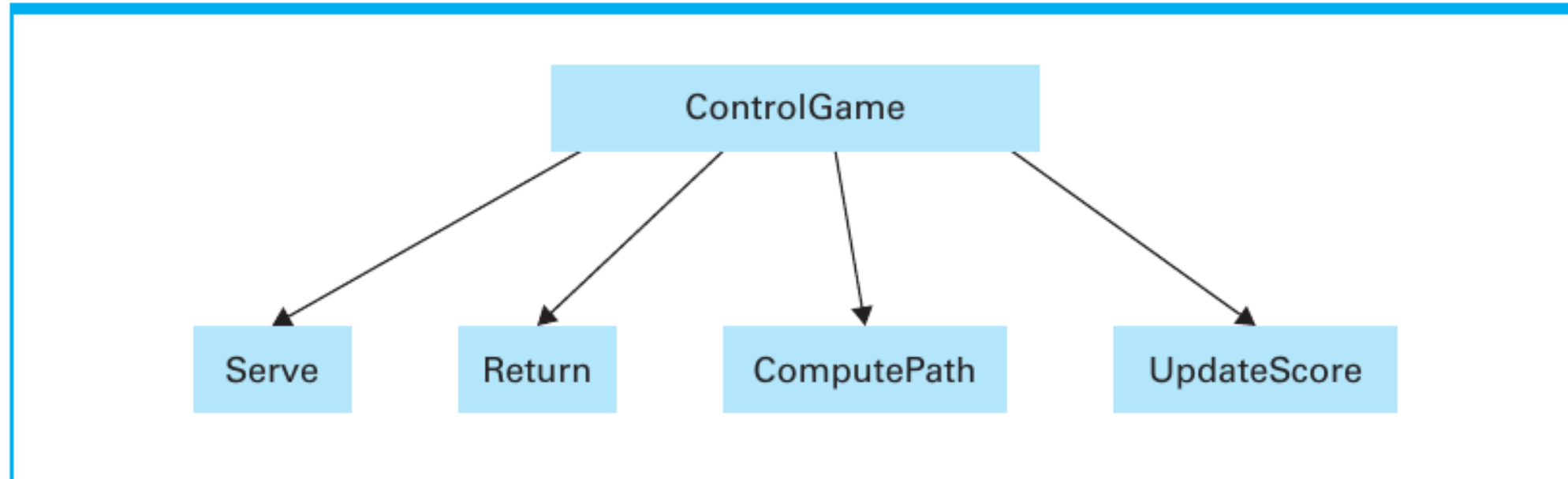
A paper sketch can show whether users understand an approval or rejection.

A small executable model can test a difficult interaction before building the full system.

A convincing demonstration is limited evidence: it may omit persistence, failures or simultaneous users.

Decomposition into responsibilities

Figure 7.3 A simple structure chart



One coordinating component uses smaller services.

Each service has a focused responsibility.

The chart shows dependencies, not the complete execution sequence.

Brookshear & Brylow, Fig. 7.3, p. 341. This is a structure chart.

A possible library decomposition

Component	Responsibility
Check eligibility	Decide whether the member may borrow.
Check availability	Decide whether this book can be issued.
Record loan	Update the lending record after approval.
Show result	Present success or the reason for rejection.

A component can contain several functions. Decomposition works at more than one scale.

An interface says what another component may ask for and what it receives.

A contract adds meaning: valid inputs, expected results and allowed side effects.

For a loan decision, agree on the meaning of “active,” the loan count and the returned decision.

Cohesion and coupling

Design property	Meaning	Library example
Cohesion	How closely a component's responsibilities belong together.	Eligibility rules stay together.
Coupling	How strongly components depend on each other's details.	The display receives a decision rather than editing loan records.

Focused responsibilities and clear interfaces make changes easier to reason about.

Other components use an agreed interface rather than the internal representation.

The records might move from one storage method to another while the decision interface stays stable.

An interface reduces what callers need to know. It does not remove the need to check compatibility.

Reuse carries assumptions

A reused function saves implementation work, but its input rules still apply.

A library developed for one device or workload may behave differently in a new environment.

The integration needs review and tests even when the component already has its own tests.

Ariane 501: a system-level failure



The inertial-reference software stopped supplying valid guidance information.

Both reference systems failed after the same software problem.

ESA identified specification/design faults and insufficiently representative tests.

Ariane 501, 1996. Photo: ©ESA/CNES, CSG. Source: ESA Bulletin 89.

What the Ariane case teaches

A value exceeded the range that the software could represent in an alignment function.

The failure response shut down the processor. A backup with the same vulnerability also failed.

Engineering evidence must cover actual operating conditions, interfaces and failure behavior.

Verification and validation

Question	Example evidence
Verification: does the implementation meet the stated rule?	Tests show that a member with 3 loans cannot receive a fourth.
Validation: does the system meet the intended need?	Librarians review realistic borrowing situations and confirm that the rules and workflow make sense.

Both activities can begin before the final program exists.

Testing needs an expected result

A test specifies the starting state, inputs and expected behavior.

The expected result comes from the requirement, not from copying the program's current output.

For a rejected loan, check both the message and the unchanged records.

Boundary tests for the loan limit

Current loans	Active member, available book	Expected
0	Clearly below the limit	Approve
2	Just below the limit of 3	Approve
3	Exactly at the limit	Reject
4	Above the limit	Reject

A separate input-validation rule should handle impossible or corrupted values such as a negative count.

Black-box and white-box testing

Black-box tests use the specified inputs and outputs without relying on the implementation.

White-box tests use knowledge of branches, paths and internal structure.

Executing every statement does not demonstrate correct behavior for every input.

Tests at different levels

Level	What it examines
Unit	One eligibility function with prepared values.
Integration	Eligibility and record updating work together.
System	The complete borrowing workflow behaves correctly.
Acceptance	Stakeholders evaluate agreed scenarios for intended use.

Passing a unit test does not establish that the complete system works.

Coverage and the limits of testing

Testing every case becomes expensive quickly. Twenty binary inputs already give 1,048,576 combinations.

Selected tests should cover meaningful categories, boundaries and important failure modes.

A passing test provides evidence for its conditions. It does not prove the absence of all defects.

A useful defect report

State the version, initial conditions and steps that reproduce the problem.

Separate the expected result from the observed result. Include a small reproducible example.

After a fix, preserve a regression test so later changes can reveal the same failure again.

Maintenance changes more than code

Change	Related work
Correct a rejected valid loan	Repair the rule and add a regression test.
Increase the limit from 3 to 5	Update the requirement, boundary tests and explanation.
Replace the record-storage component	Check the interface and integration behavior.
Clarify a confusing message	Review it with users and update documentation.

A change can be correct locally and still break an assumption elsewhere.

Version control records changes and allows comparison with an earlier state.

A review asks whether the change matches the requirement and affects other behavior.

A meaningful change description records why the system changed. Version history complements backups.

A small traceability record

Requirement	Design location	Evidence
Active member, fewer than 3 loans	Eligibility component	Cases: inactive, 0, 2, 3, 4 loans
Rejected loan changes no records	Loan coordinator	Compare records before and after rejection
Explain the rejection	Result presentation	Review message for each rejection reason

Traceability helps identify what needs review when a requirement changes.

Design review: the limit becomes five

Which parts should change? The requirement, decision rule, tests and user explanation.

Which tests become especially useful? Four, five and six current loans.

What should remain true? Inactive members still cannot borrow, and rejection still leaves records unchanged.

Small components, explicit data

Understanding calls, contracts and traversal

A function names a computation

The caller supplies arguments. Parameters give those values local names inside the function.

A return value carries a result back to the caller. A void function returns no value.

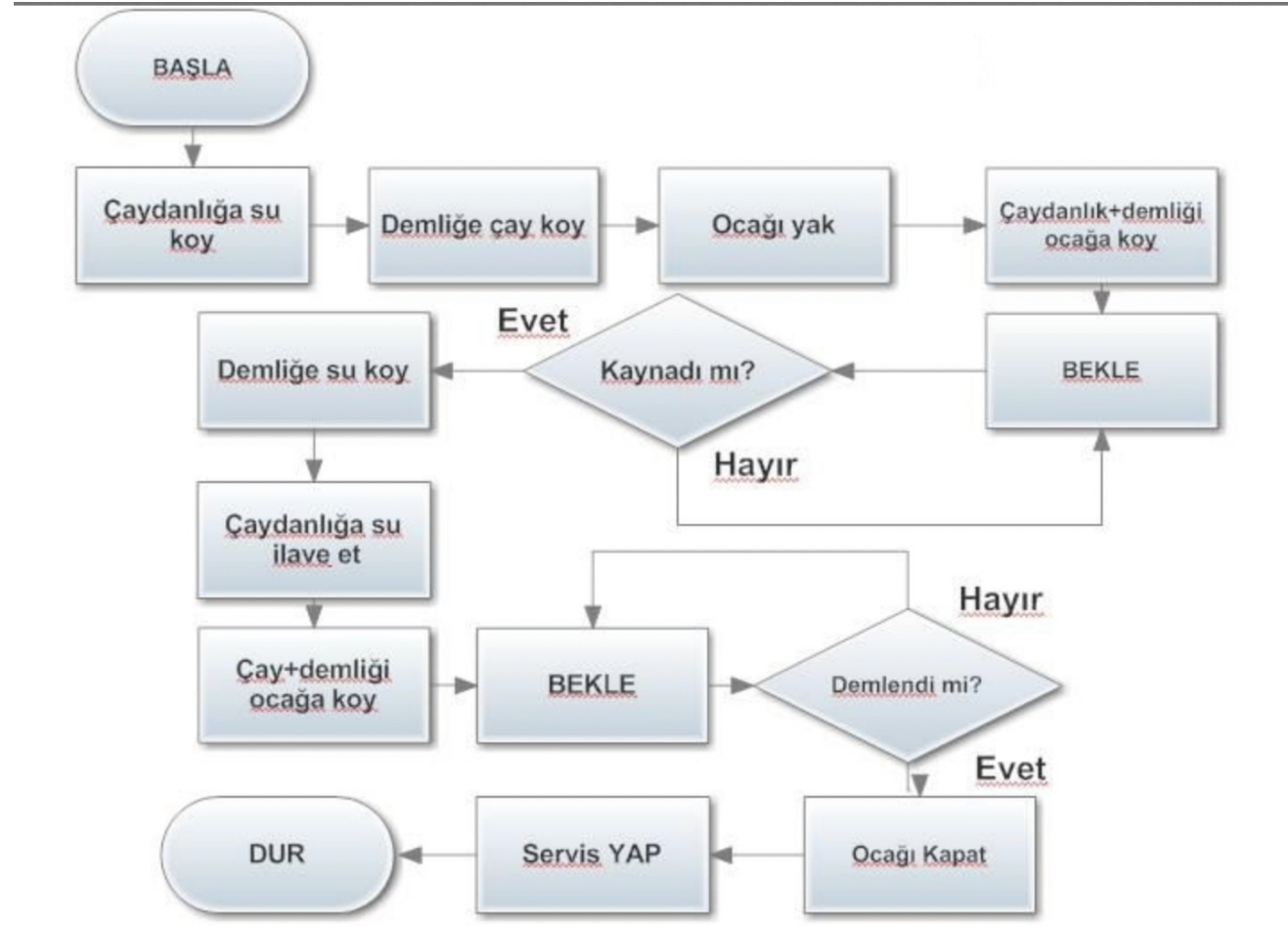
A useful function separates one responsibility so we can understand and test it.

The tea-making example



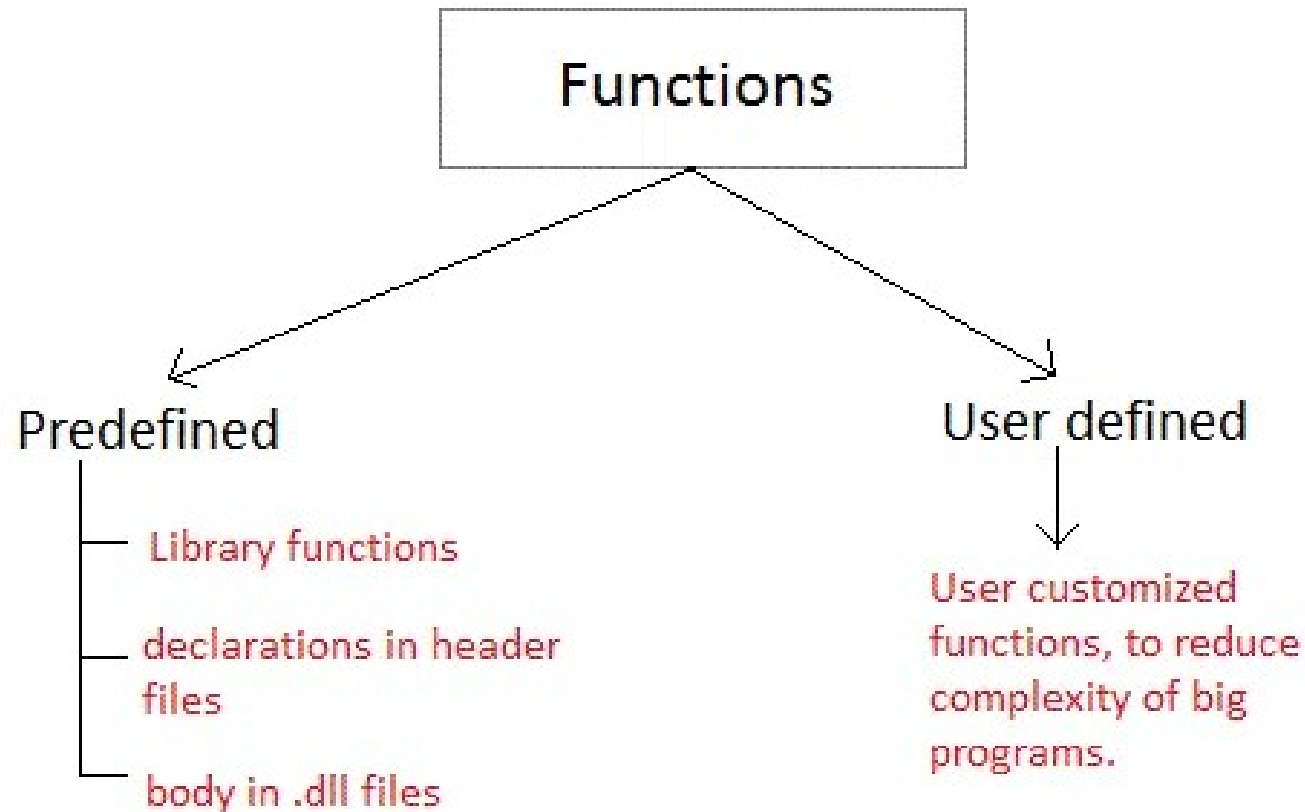
A repeated everyday task can be named and reused as a procedure.

Tea-making flowchart



A familiar task separates decisions, repetition and reusable subprocedures.

Library and user-defined functions



Library and user-defined functions. The DLL wording is platform-specific; C libraries may link statically or dynamically.



One decision as a function

```
int canBorrow(int active, int loans)
{
    return active && loans < 3;
}
```

```
/* Example call */
int allowed = canBorrow(1, 2);
```

Contract: active is 0 or 1;
loans is nonnegative.

The result is 1 or 0. This
helper does not check book
availability.

The call returns 1.

Original function-return illustration

```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

int multiply(int a, int b)
{
    ... ..
    return a*b;
}
```

The value returned by the function must be stored in a variable.

A call can provide a value to an expression. C does not require storing every return value in a variable.

Arguments, parameters and result

At the call	Inside the function	Result
canBorrow(1, 2)	active = 1, loans = 2	1
canBorrow(1, 3)	active = 1, loans = 3	0
canBorrow(0, 0)	active = 0, loans = 0	0

The same function body handles different argument values.

Scalar parameters receive values

```
void change(int value)
{
    value = 99;
}
```

```
/* In the caller */
int x = 4;
change(x);
/* x is still 4 */
```

value is a local parameter.

Assigning to value does not assign to x.

Each ordinary call gets its own automatic local variables.

A declaration tells the compiler the function name, result type and parameter types before a call.

A definition also supplies the body. Placing the definition before its first call is sufficient here.

A local name belongs to its block. Clear parameters and returns reduce reliance on shared global variables.

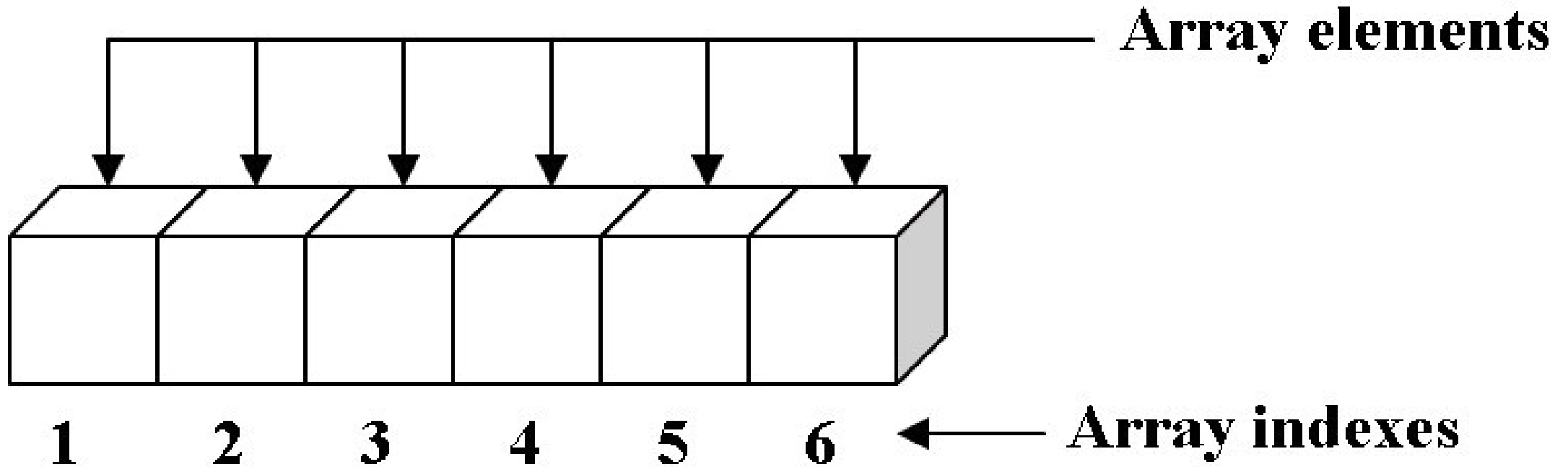
An array stores a sequence

An array groups elements of the same type under one name.

Each index selects one element. For six elements, the valid indices are 0 through 5.

The elements occupy consecutive storage locations. An index is a position, not the stored value.

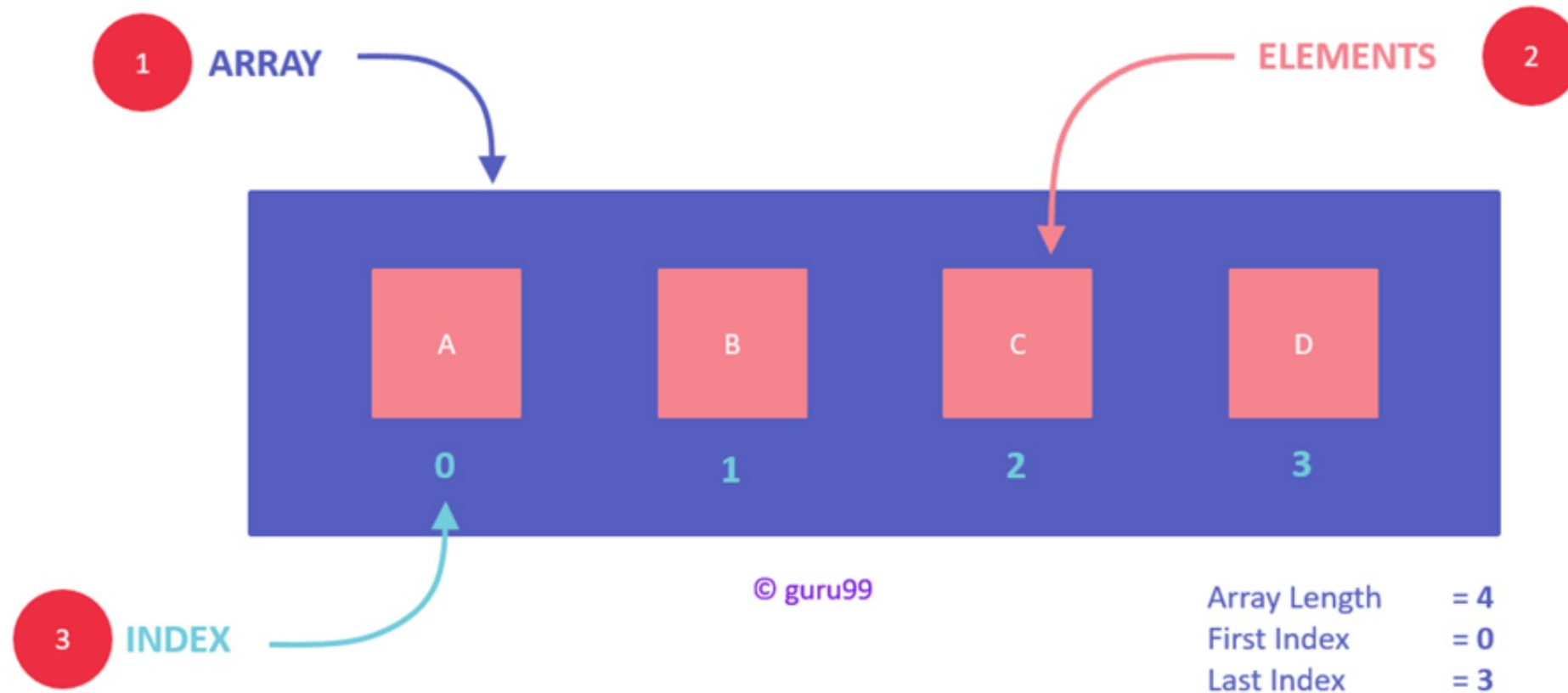
One-dimensional array



One-dimensional array with six elements

Six consecutive elements. In C, their indices are 0–5; the original drawing labels positions 1–6.

Array, element and index



The array has four elements. Its first index is 0 and its last index is 3.

Indices and values are different

Index	0	1	2	3	4	5
<code>samples[index]</code>	0	1	1	0	1	0

`samples[4]` is 1. The sixth element is `samples[5]`. `samples[6]` is outside this array.

Creating and changing a small array

```
int samples[6] = {0, 1, 1, 0, 1, 0};
```

```
int first = samples[0];  
samples[3] = 1;
```

first receives 0.

Writing index 3 changes the fourth element.

After this assignment the values are 0, 1, 1, 1, 1, 0.

Array declaration

PYTHON

```
balance = array.array( 'i', [300,200,100] )
```

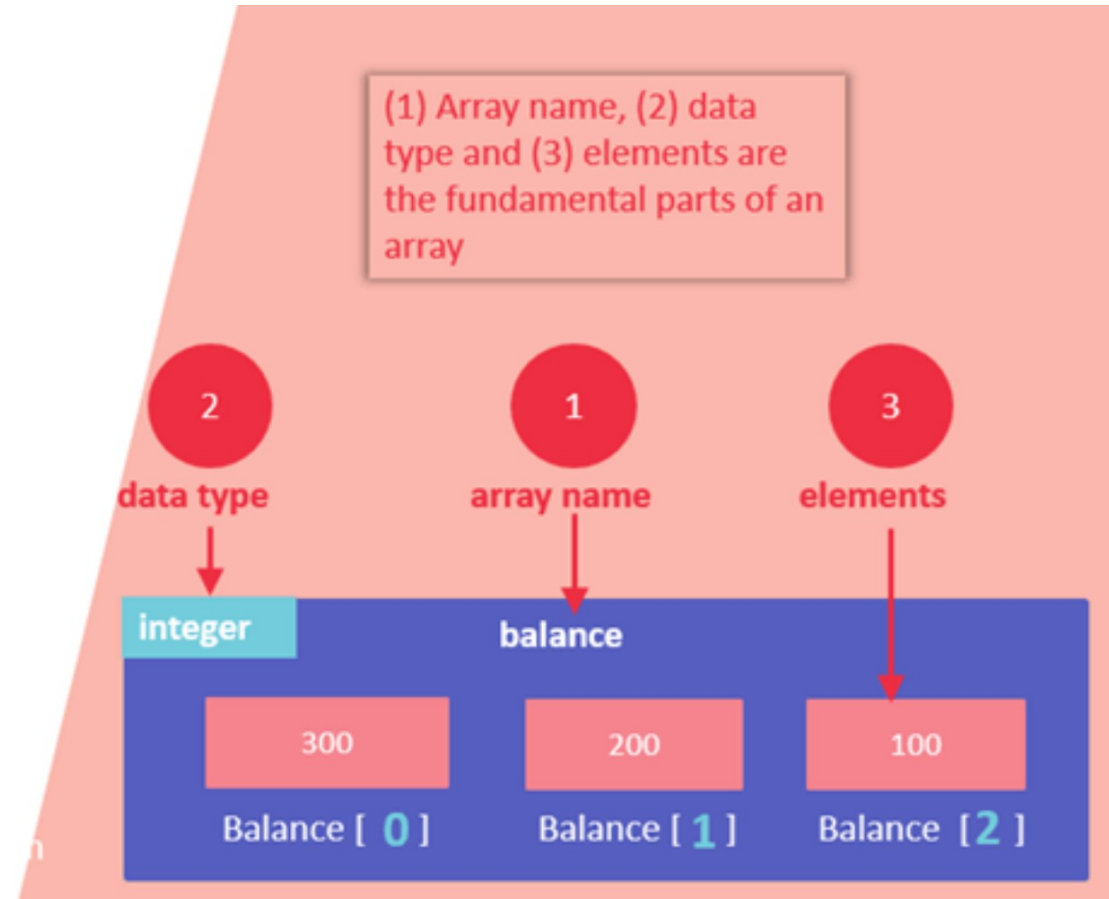
↑ ↑ ↑
array name data type elements

C++

```
int balance[3] = { 300, 200, 100 }
```

↑ ↑ ↑
data type array name elements

© guru99



In C: `int balance[3] = {300, 200, 100};` The Python example uses the array module.

An uninitialized automatic local array does not provide usable initial values. Initialize before reading.

For `int values[6] = {0};` all six elements start at zero.

An out-of-bounds access has undefined behavior in C. A run that appears to work is not evidence of safety.

Accessing an array element

array name (variable)

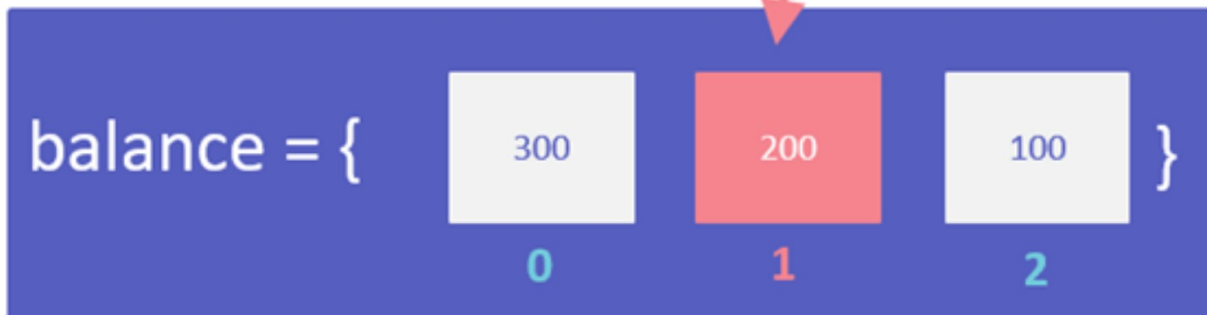
index

`balance[1]`

output = 200

By referring the index number of an element, you can get its value. For example, in this case, value of 200 is located at index 1, fetched via syntax of `balance[1]`

© guru99



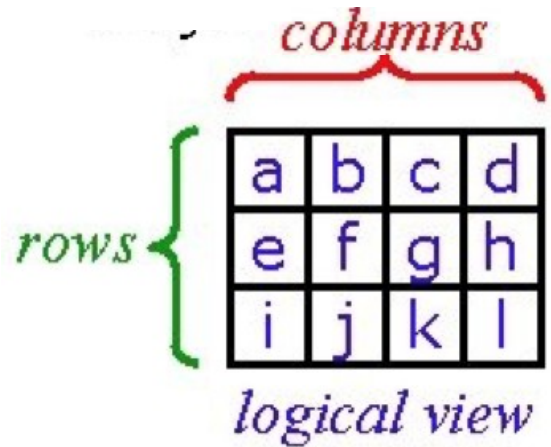
`balance[1]` selects the second element, whose value is 200.

Two-dimensional arrays

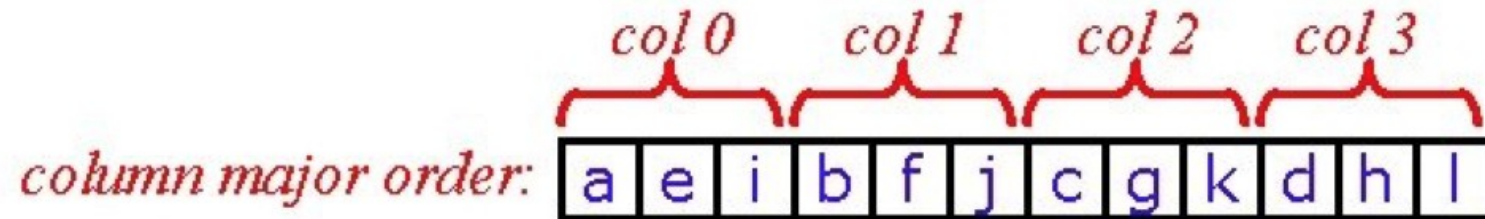
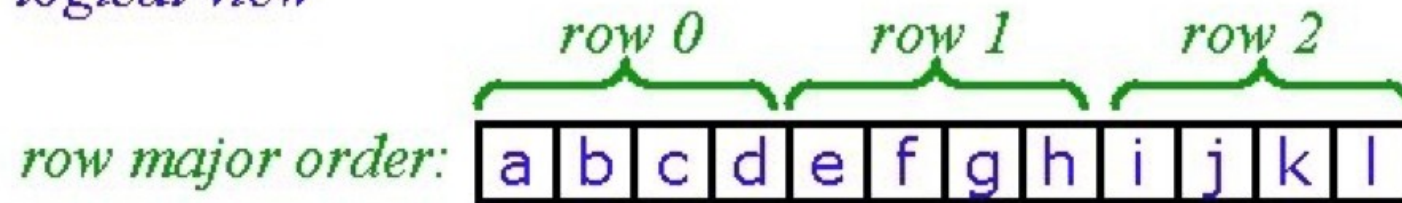
	Col1	Col2	Col3	Col4	
Row1	Arr[0][0]	Arr[0][1]	Arr[0][2]	Arr[0][3]	
Row2	Arr[1][0]	Arr[1][1]	Arr[1][2]	Arr[1][3]	
Row3	Arr[2][0]	Arr[2][1]	Arr[2][2]	Arr[2][3]	
Row4	Arr[3][0]	Arr[3][1]	Arr[3][2]	Arr[3][3]	
⋮					

Arr[row][column] uses two indices. Each index starts at zero.

Two-dimensional arrays in memory



- A two-dimensional array may be stored in one-dimensional computer memory in either of two ways:



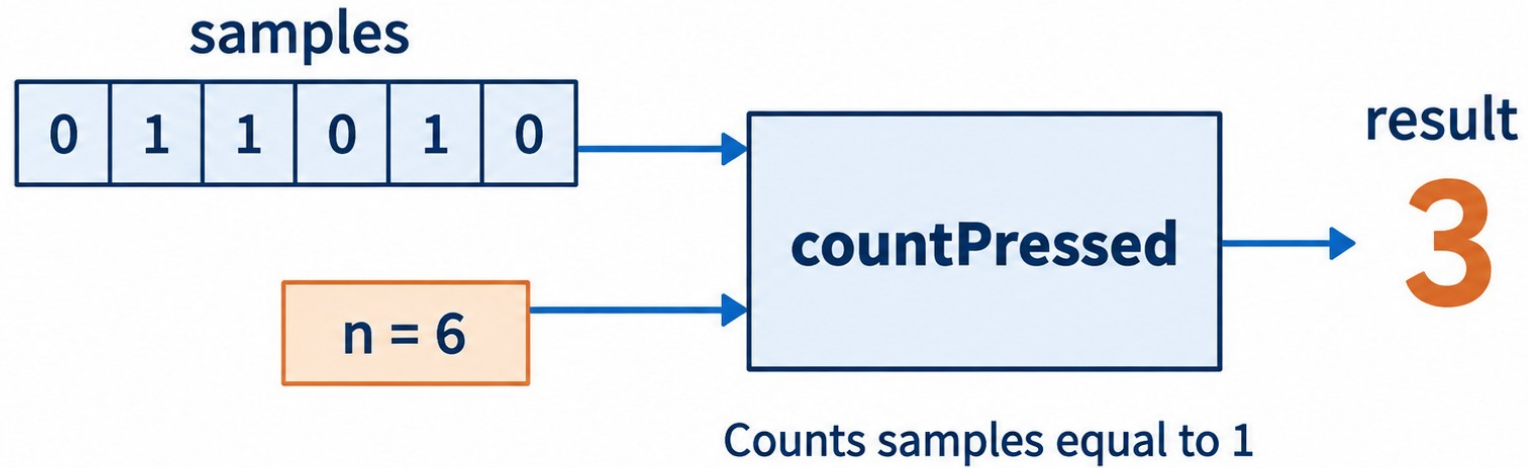
C stores actual multidimensional arrays in row-major order: one complete row follows another.

Three-dimensional arrays



In C, the rightmost index varies fastest. Some labels repeat incorrectly in the original bottom row.

The array helper's contract



Each example starts with the original values: 0, 1, 1, 0, 1, 0.

What countPressed promises

Inputs: an array of logical samples and n , the number of elements to examine.

Preconditions: $0 \leq n \leq$ the available array length; every examined sample is 0 or 1.

Result: the number of examined samples equal to 1. The helper leaves the array unchanged.

With $n = 0$, the result is 0. A nonempty allocated array can still have zero elements to process.

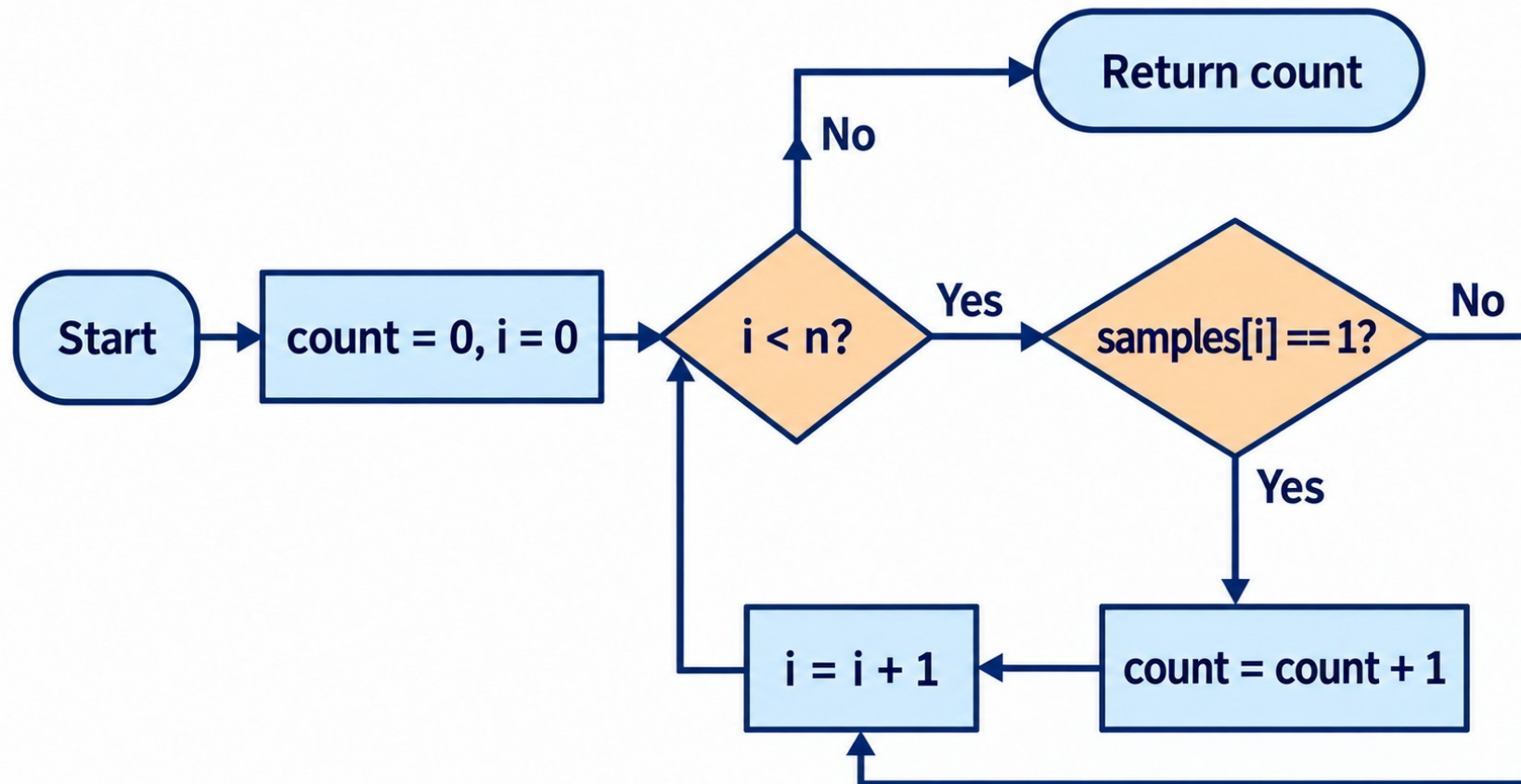
An array argument needs a length

In a C function parameter, `const int samples[]` provides access to the caller's elements. It does not copy the whole array.

Pass `n` separately. The parameter itself does not retain the caller's array length.

`const` prevents this helper from writing through that parameter. Pointer mechanics come later.

Counting by a single traversal



The loop examines exactly the first n elements when the preconditions hold.

The helper in C

```
int countPressed(const int samples[], int n)
{
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (samples[i] == 1) {
            count++;
        }
    }
    return count;
}
```

$i < n$ protects the upper boundary when n is valid.

count increases only for a value of 1.

The same helper works with fixed data before connecting hardware.

A trace of the six samples

i	samples[i]	count after this element
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	0	3

Then i becomes 6. The condition is false, so the function returns 3.

Why the traversal gives the answer

Before each iteration, count equals the number of 1s among the first i elements.

Examining element i preserves that claim for the first $i + 1$ elements.

The index increases until it reaches n . At that point, count describes all n requested elements.

Tests for the helper

Examined values	n	Expected count
No elements examined	0	0
0	1	0
1	1	1
0, 0, 0, 0	4	0
1, 1, 1, 1	4	4
0, 1, 1, 0, 1, 0	6	3

Also test the first three elements of the mixed array: $n = 3$ should return 2.

A boundary defect to recognize

If the loop uses $i \leq n$, it tries to read element n . That element lies outside an n -element array.

If count starts at 1, even zero examined elements produce the wrong answer.

A boundary test and a short trace can reveal different defects. Compiler success alone is insufficient.

Acquisition code supplies logical samples: 1 means pressed and 0 means released.

The helper counts the supplied data. Serial output reports the result.

Electrical input levels, switch bounce and sampling belong to acquisition. Counting pressed samples does not count physical press events.

Ideas to carry into the next lesson

A requirement defines what a correct result means. A contract makes component assumptions visible.

Focused functions and explicit array bounds make behavior easier to explain and test.

The next algorithm question is how the work changes when the input grows. Searching and sorting follow in Week 8.

Brookshear & Brylow, Chapter 7: life cycle, modularity, quality assurance and documentation.

NASA Software Engineering Handbook: requirements validation and verification planning.

ESA Bulletin 89: Ariane-5 flight 501 findings and corrective actions.

GNU C manual and Arduino language reference: functions and array parameters.

Complete examples for independent practice

The core lecture ends before this section.
Examples reuse the ideas already discussed.

Reference: complete loan decision program

```
#include <stdio.h>

int canBorrow(int active, int loans)
{
    return active && loans < 3;
}

int main(void)
{
    printf("%d\n", canBorrow(1, 2));
    printf("%d\n", canBorrow(1, 3));
    printf("%d\n", canBorrow(0, 0));
    return 0;
}
```

PC C program.

Output, on separate lines: 1,
0, 0.

This component checks
membership and count, not
book availability.

Reference: complete array-count program

```
#include <stdio.h>
int countPressed(const int samples[], int n)
{
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (samples[i] == 1) count++;
    }
    return count;
}
int main(void)
{
    int samples[6] = {0, 1, 1, 0, 1, 0};
    printf("%d\n", countPressed(samples, 6));
    return 0;
}
```

PC C program.

Output: 3.

The fixed array gives a repeatable test without hardware.

Reference: Arduino with fixed data

```
int countPressed(const int samples[], int n)
{
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (samples[i] == 1) count++;
    }
    return count;
}

void setup()
{
    Serial.begin(9600);
    int samples[6] = {0, 1, 1, 0, 1, 0};
    Serial.println(countPressed(samples, 6));
}

void loop() {}
```

Arduino C++ sketch, using the same helper.

Set Serial Monitor to 9600 baud. Reset to print 3 again.

Fixed logical samples isolate computation from acquisition.

Reference: counting digits with an array

```
#include <stdio.h>

/* count digits, white space, others */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("digits =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
}
```

Original worked example. Modern C uses `int main(void)`; the counters illustrate array indexing.

Reference: digit-counter output

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

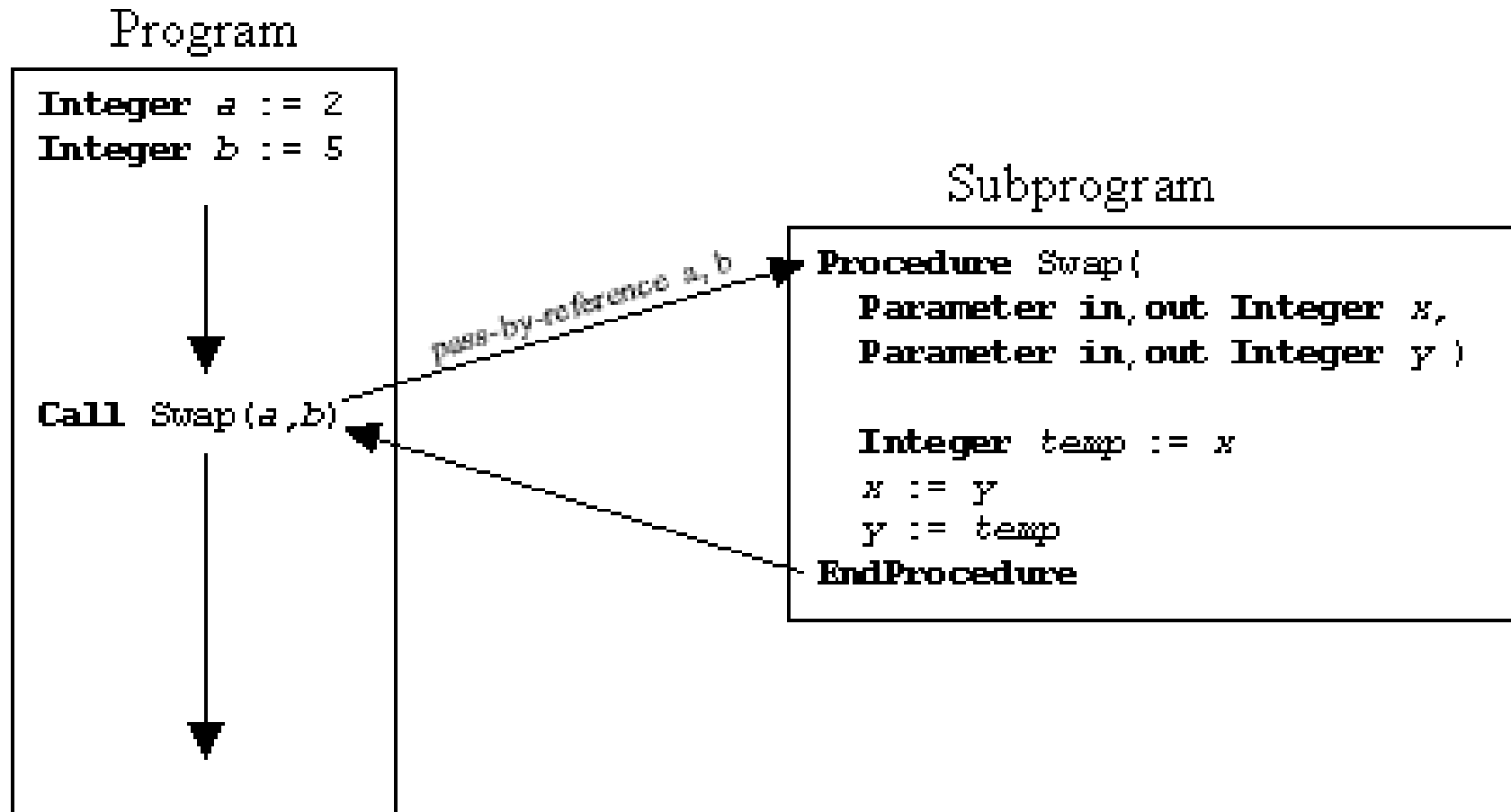
Ten counts correspond to digits 0–9, followed by whitespace and other-character counts.

Reference: character codes and array indices

46	056	2E	00101110	.	.	.	Period, dot or full stop
47	057	2F	00101111	/	/	/	Slash or divide
48	060	30	00110000	0	0		Zero
49	061	31	00110001	1	1		One
50	062	32	00110010	2	2		Two
51	063	33	00110011	3	3		Three
52	064	34	00110100	4	4		Four
53	065	35	00110101	5	5		Five
54	066	36	00110110	6	6		Six
55	067	37	00110111	7	7		Seven
56	070	38	00111000	8	8		Eight
57	071	39	00111001	9	9		Nine
58	072	3A	00111010	:	:	:	Colon
59	073	3B	00111011	;	;	;	Semicolon
60	074	3C	00111100	<	<	<	Less than (or open angled bracket)

For a digit character `c`, `c - '0'` gives an index from 0 to 9. C guarantees consecutive digit codes.

Parameter passing in pseudocode



This diagram uses pass-by-reference pseudocode. Ordinary C parameters receive values; caller mutation is a later pointer topic.

Historical call-and-return illustration

```
1 0 movv sp 80
2 1 movv a 100
3 2 movv b 400
4 3 call 20
5 4 add a b
6 5 out acc
7 6 halt
8
9
10 20 push a
11 21 push b
12 22 push acc
13 23 movv a 20
14 24 movv b 80
15 25 add a b
16 26 out acc
17 27 pop acc
18 28 pop b
19 29 pop a
20 30 ret
```

Archive reference from the source deck. No simulator or assembly exercise is part of Week 7.