



Gurukul Technicals Institute

Computer education · Robotics · Science projects · Delhi

FREE EBOOK

PROGRAMMING AND WEB

C Practice Programs

Thirty programs in C with the logic explained line by line

INSIDE THIS BOOK

- How a C program is built
- Decision programs
- Loop and pattern programs
- Array programs
- String programs
- ...and 3 more chapters, practice questions and a glossary

Beginner to intermediate

Students preparing for C practicals and exams

engineergurukul.com

+91 97116 99511

C Practice Programs

Practise C with thirty complete programs, from your first printf to loops, patterns, arrays, strings, recursion, pointers, structures and files. Each comes with sample output and a plain explanation, so you can write, run and explain similar programs in practicals and exams.

WHO IT IS FOR	Students preparing for C practicals and exams
LEVEL	Beginner to intermediate
SUBJECT AREA	Programming and web
WHAT IS INSIDE	8 chapters, practice questions with answers, and a glossary

CONTENTS

1	How a C program is built	3
2	Decision programs	6
3	Loop and pattern programs	10
4	Array programs	16
5	String programs	19
6	Functions and recursion	21
7	Pointers and structures	23
8	File handling programs	26
	Practice questions	27
	Answers	27
	Glossary	28

Free ebook: you may read, print and share it for learning; it may not be sold or changed. © 2026 Gurukul Technicals Institute. First edition, October 2026. Contact details are on the back cover.

How a C program is built

Every C program has the same basic shape. Type each program yourself: your mistakes teach you what the compiler expects.

Program 1: Hello

C

```
#include <stdio.h>
int main(void) {
    printf("Namaste, C!\n");
    return 0;
}
```

Namaste, C!

- `#include <stdio.h>` brings in the standard input/output library, which holds `printf` and `scanf`. Lines starting with `#` are handled by the **preprocessor**, a step that runs before compiling.
- `int main(void)` is where every program starts; `int` means it hands a whole number back to the operating system.
- `printf` prints text. `\n` is the **newline** character, which moves to the next line.
- `return 0;` reports that the program finished without errors.

Compiling with gcc

A **compiler** turns your source code (the text you typed) into an **executable**, a file the computer can run.

1. Save the program as `hello.c`.
2. Open a terminal (Command Prompt on Windows) in that folder.
3. Type `gcc -Wall hello.c -o hello` and press Enter. `-o hello` names the output file and `-Wall` turns on warnings.
4. Run it with `./hello` on Linux or macOS, or `hello` on Windows.

DID YOU KNOW

Turbo C's `void main()` and `conio.h` are not standard C, so this book never uses them.

Program 2: Sum of two numbers

C

```
#include <stdio.h>
int main(void) {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    printf("Sum = %d\n", a + b);
    return 0;
}
```

Enter two numbers: 25 17
Sum = 42

`int a, b;` creates two **variables**, named boxes in memory for whole numbers. In `scanf`, each `%d` is a **format specifier** meaning "an int", and `&a` is the address of `a`, telling `scanf` where to store the value. Forgetting `&` is the most common `scanf` mistake.

REMEMBER

Every statement ends with a semicolon. When one is missing, gcc's message often points at the next line, so check the line above too.

Input and output

Each type has its own format specifier for `printf` (output) and `scanf` (input).

Type	printf	scanf	Usual size
int (whole number)	%d	%d	4 bytes
float (decimal)	%f	%f	4 bytes
double (precise decimal)	%f	%lf	8 bytes
char (one character)	%c	%c	1 byte

`%.2f` prints two digits after the decimal point, which suits rupees and paise. `sizeof(int)` gives the bytes a type uses on your computer, printed with `%zu`. Only `char` is fixed at 1 byte; the others can differ between computers.

Program 3: Area and circumference of a circle

C

```
#include <stdio.h>
#define PI 3.14159
int main(void) {
    double r;
    printf("Enter radius: ");
    scanf("%lf", &r);
    printf("Area = %.2f\n", PI * r * r);
    printf("Circumference = %.2f\n", 2 * PI * r);
    return 0;
}
```

```
Enter radius: 7
Area = 153.94
Circumference = 43.98
```

`#define PI 3.14159` creates a **macro**: the preprocessor replaces every `PI` with 3.14159 before compiling. Note `%lf` in `scanf` for a double, but `%f` in `printf`.

Program 4: Simple interest

Simple interest is $P \times R \times T \div 100$. Ravi keeps ! 20,000 in a deposit for 3 years at an example rate of 7.5% a year.

C

```
#include <stdio.h>
int main(void) {
    double p, r, t, si;
    printf("Principal, rate, years: ");
    scanf("%lf %lf %lf", &p, &r, &t);
    si = p * r * t / 100;
    printf("Interest = Rs %.2f\nTotal    = Rs %.2f\n", si, p + si);
    return 0;
}
```

```
Principal, rate, years: 20000 7.5 3
Interest = Rs 4500.00
Total    = Rs 24500.00
```

We print "Rs" because some terminals cannot show the ! sign.

Program 5: Celsius to Fahrenheit

C

```
#include <stdio.h>
int main(void) {
    float c, f;
    printf("Temperature in Celsius: ");
    scanf("%f", &c);
    f = c * 9 / 5 + 32;
    printf("%.1f C = %.1f F\n", c, f);
    return 0;
}
```

```
Temperature in Celsius: 45
45.0 C = 113.0 F
```

REMEMBER

In C, an int divided by an int drops the fraction, so `9 / 5` is 1, not 1.8, and `(9 / 5) * c + 32` gives 77 for 45 °C. `c * 9 / 5` works because `c` is a float.

CHAPTER 2

Decision programs

A program decides with `if`, `else if`, `else` and `switch`. Conditions use these operators:

Operator	Meaning	Operator	Meaning
<code>==</code>	equal to	<code>!=</code>	not equal to
<code>&&</code>	AND (both true)	<code> </code>	OR (either true)

`<`, `>`, `<=` and `>=` work as in maths. When a branch is one short statement, this book writes it on the same line as its `if` or `else`.

REMEMBER

`=` stores a value and `==` compares. `if (n = 5)` stores 5 in `n` and is always true. With `-Wall`, `gcc` warns you.

Program 6: Even or odd, positive or negative

C

```
#include <stdio.h>
int main(void) {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    if (n % 2 == 0) printf("%d is even\n", n);
    else printf("%d is odd\n", n);
    if (n > 0) printf("Positive\n");
    else if (n < 0) printf("Negative\n");
    else printf("Zero\n");
    return 0;
}
```

```
Enter a number: -7
-7 is odd
Negative
```

`%` gives the remainder; even numbers leave 0 when divided by 2. Test `== 0`, not `== 1`, because for -7, `n % 2` is -1. The second test is an **else-if ladder**: C checks it from the top and runs only the first true branch, and the final `else` catches the rest.

Program 7: Largest of three numbers

C

```
#include <stdio.h>
int main(void) {
    int a, b, c;
    printf("Enter three numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    if (a >= b && a >= c) printf("Largest = %d\n", a);
    else if (b >= c) printf("Largest = %d\n", b);
    else printf("Largest = %d\n", c);
    return 0;
}
```

```
Enter three numbers: 45 92 67
Largest = 92
```

If `a` beats both others it wins; otherwise one comparison settles it. `>=` handles ties.

Program 8: Leap year

C

```
#include <stdio.h>
int main(void) {
    int year;
    printf("Enter a year: ");
    scanf("%d", &year);
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
        printf("%d is a leap year\n", year);
    else
        printf("%d is not a leap year\n", year);
    return 0;
}
```

```
Enter a year: 1900
1900 is not a leap year
```

Years divisible by 4 are leap years, except century years, which must be divisible by 400: 2024 and 2000 are leap years, 1900 is not.

Program 9: Grade from marks

This uses a sample grading scheme; your school's may differ.

C

```
#include <stdio.h>
int main(void) {
    int marks;
    printf("Enter marks (0-100): ");
    scanf("%d", &marks);
    if (marks < 0 || marks > 100) printf("Invalid marks\n");
    else if (marks >= 90) printf("Grade A\n");
    else if (marks >= 75) printf("Grade B\n");
    else if (marks >= 60) printf("Grade C\n");
    else if (marks >= 40) printf("Grade D\n");
    else printf("Grade E\n");
    return 0;
}
```

```
Enter marks (0-100): 78
Grade B
```

The ladder stops at the first true test, so `marks >= 75` is only reached for marks below 90: it means 75 to 89.

Program 10: ASCII value, vowel or consonant

C

```
#include <stdio.h>
#include <ctype.h>
int main(void) {
    char ch;
    printf("Enter a letter: ");
    scanf(" %c", &ch);
    printf("ASCII value of %c is %d\n", ch, ch);
    if (!isalpha(ch)) {
        printf("%c is not a letter\n", ch);
        return 0;
    }
    switch (tolower(ch)) {
        case 'a': case 'e': case 'i': case 'o': case 'u':
            printf("%c is a vowel\n", ch);
            break;
        default:
            printf("%c is a consonant\n", ch);
    }
    return 0;
}
```

```
Enter a letter: E
ASCII value of E is 69
E is a vowel
```

A `char` is stored as a small number, its **ASCII** code, so `%c` prints the letter and `%d` the number. The space before `%c` skips any leftover spaces or Enter presses. `isalpha` (is it a letter?) and `tolower` (make it a small letter) come from `ctype.h`. Several `case` labels can share one block. Without `break`, C runs on into the next case: this is **fall-through**.

Program 11: Simple calculator

C

```
#include <stdio.h>
int main(void) {
    double a, b;
    char op;
    printf("Enter a sum like 8 * 5: ");
    scanf("%lf %c %lf", &a, &op, &b);
    switch (op) {
        case '+': printf("= %.2f\n", a + b); break;
        case '-': printf("= %.2f\n", a - b); break;
        case '*': printf("= %.2f\n", a * b); break;
        case '/':
            if (b == 0) printf("Cannot divide by zero\n");
            else printf("= %.2f\n", a / b);
            break;
        default: printf("Unknown operator\n");
    }
    return 0;
}
```

```
Enter a sum like 8 * 5: 125 / 4
= 31.25
```

scanf reads a number, a character and a number in one go, and division checks for zero first.

CHAPTER 3

Loop and pattern programs

A **loop** repeats statements. C has three kinds.

Loop	Use it when
<code>for</code>	you know how many times to repeat
<code>while</code>	you repeat until something changes
<code>do-while</code>	the body must run at least once, as in a menu

Every loop needs a starting value, a condition to keep going and a change that brings it closer to stopping.

TIP

Without a change step, a loop never ends. Press Ctrl+C to stop the program.

Program 12: Sum of 1 to n and a multiplication table

C

```
#include <stdio.h>
int main(void) {
    int n, i, sum = 0;
    printf("Enter n: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        sum += i;
    printf("Sum of 1 to %d = %d\n", n, sum);
    for (i = 1; i <= 10; i++)
        printf("%d x %d = %d\n", n, i, n * i);
    return 0;
}
```

```
Enter n: 7
Sum of 1 to 7 = 28
7 x 1 = 7
...
7 x 10 = 70
```

`sum += i` is short for `sum = sum + i`, and `i++` adds 1 to `i`. Check with the formula $n(n+1)/2$: $7 \times 8 / 2 = 28$. The second loop always runs ten times, printing one line of the table on each pass.

Program 13: Factorial

C

```
#include <stdio.h>
int main(void) {
    int n, i;
    unsigned long long fact = 1;
    printf("Enter n (0-20): ");
    scanf("%d", &n);
    for (i = 2; i <= n; i++)
        fact *= i;
    printf("%d! = %llu\n", n, fact);
    return 0;
}
```

```
Enter n (0-20): 10
10! = 3628800
```

`n!` means $1 \times 2 \times \dots \times n$, and `0!` is 1. Factorials grow fast, so we use `unsigned long long` (printed with `%llu`), which still overflows after 20!.

Program 14: Sum of digits, reverse and palindrome

C

```
#include <stdio.h>
int main(void) {
    int n, original, sum = 0, rev = 0;
    printf("Enter a number: ");
    scanf("%d", &n);
    original = n;
    while (n > 0) {
        sum += n % 10;
        rev = rev * 10 + n % 10;
        n /= 10;
    }
    printf("Sum of digits = %d\nReversed = %d\n", sum, rev);
    if (rev == original) printf("%d is a palindrome\n", original);
    else printf("%d is not a palindrome\n", original);
    return 0;
}
```

```
Enter a number: 1234
Sum of digits = 10
Reversed = 4321
1234 is not a palindrome
```

The key trick of number programs: `n % 10` is the last digit and `n /= 10` removes it. The loop peels digits off from the right, adding each to `sum` and appending it to `rev`. A **trace table** shows the variables after each pass:

Pass	sum	rev	n
1	4	4	123
2	7	43	12
3	9	432	1
4	10	4321	0

A **palindrome** reads the same both ways, such as 12321. The loop destroys `n`, so we keep a copy in `original` to compare with the reverse.

Program 15: Armstrong number

C

```
#include <stdio.h>
int main(void) {
    int n, t, digits = 0, sum = 0;
    printf("Enter a number: ");
    scanf("%d", &n);
    for (t = n; t > 0; t /= 10)
        digits++;
    for (t = n; t > 0; t /= 10) {
        int d = t % 10, p = 1, k;
        for (k = 0; k < digits; k++)
            p *= d;
        sum += p;
    }
    if (sum == n) printf("%d is an Armstrong number\n", n);
    else printf("%d is not an Armstrong number\n", n);
    return 0;
}
```

```
Enter a number: 153
153 is an Armstrong number
```

An **Armstrong number** equals the sum of its digits, each raised to the power of the number of digits: $1^3 + 5^3 + 3^3 = 153$. The first loop counts digits; the second raises each digit to that power and adds it. Also try 370 and 1634.

Program 16: Fibonacci series

C

```
#include <stdio.h>
int main(void) {
    int n, i, a = 0, b = 1, next;
    printf("How many terms? ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        printf("%d ", a);
        next = a + b;
        a = b;
        b = next;
    }
    printf("\n");
    return 0;
}
```

```
How many terms? 10
0 1 1 2 3 5 8 13 21 34
```

Each term is the sum of the two before it. `a` and `b` hold the last two; after printing `a`, both slide forward.

Program 17: GCD and LCM

C

```
#include <stdio.h>
int main(void) {
    int a, b, x, y, r;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    x = a;
    y = b;
    while (y != 0) {
        r = x % y;
        x = y;
        y = r;
    }
    printf("GCD = %d\nLCM = %d\n", x, a / x * b);
    return 0;
}
```

```
Enter two numbers: 36 60
GCD = 12
LCM = 180
```

Euclid's method replaces the pair (x, y) with $(y, x \% y)$ until y is 0: $(36, 60), (60, 36), (36, 24), (24, 12), (12, 0)$. The GCD is 12, and $LCM = a \times b \div GCD$.

Pattern programs

Patterns use **nested loops**, a loop inside a loop. The outer loop picks the row; the inner loop prints that row. Always ask: "How many things does row i have?"

Program 18: Star triangle and Floyd's triangle

C

```
#include <stdio.h>
int main(void) {
    int n = 4, i, j, num = 1;
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= i; j++)
            printf("* ");
        printf("\n");
    }
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= i; j++)
            printf("%3d", num++);
        printf("\n");
    }
    return 0;
}
```

```
*
* *
* * *
* * * *
1
2 3
4 5 6
7 8 9 10
```

Row i has i items, so j runs from 1 to i . The `printf("\n")` sits outside the inner loop, so it ends each row. Change `printf("* ")` to `printf("%d ", j)` and the first loops print a number triangle: 1, then 1 2, and so on. In **Floyd's triangle**, `num` is never reset, so it keeps counting across rows; `num++` uses the value, then adds 1. `%3d` pads each number to three characters, lining up the columns.

Program 19: Pyramid

C

```
#include <stdio.h>
int main(void) {
    int n = 4, i, j;
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n - i; j++)
            printf(" ");
        for (j = 1; j <= 2 * i - 1; j++)
            printf("*");
        printf("\n");
    }
    return 0;
}
```

```
*  
***  
*****  
*****
```

Each row has two parts, so there are two inner loops: `n - i` spaces, then `2 * i - 1` stars (1, 3, 5, 7).

TRY THIS

Change one line of Program 18 so the star triangle prints upside down. Hint: start the outer loop at `n` and count down.

CHAPTER 4

Array programs

An **array** stores many values of one type under one name. `int marks[5];` makes five boxes, `marks[0]` to `marks[4]`. The position number is the **index**, and it starts at 0.

SAFETY

C does not check indexes. Writing to `marks[5]` in a five-element array silently damages other memory and may crash the program.

Program 20: Largest, smallest and bubble sort

C

```
#include <stdio.h>
int main(void) {
    int a[6] = {45, 78, 12, 90, 33, 67};
    int n = 6, i, j, t, max = a[0], min = a[0];
    for (i = 1; i < n; i++) {
        if (a[i] > max) max = a[i];
        if (a[i] < min) min = a[i];
    }
    printf("Max = %d, Min = %d\n", max, min);
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - 1 - i; j++)
            if (a[j] > a[j + 1]) {
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

```
Max = 90, Min = 12
12 33 45 67 78 90
```

For the largest and smallest, assume the first element is both, then let every other element challenge it. Starting `max` at 0 fails when all values are negative. **Bubble sort** compares neighbours and swaps them when out of order. Each pass carries the largest remaining value to the end, so the next pass stops one place earlier. For descending order, change `>` to `<`.

Program 21: Linear and binary search

C

```
#include <stdio.h>
int main(void) {
    int a[8] = {5, 12, 19, 27, 33, 48, 56, 71};
    int key, i, pos = -1, low = 0, high = 7, mid;
    printf("Search for: ");
    scanf("%d", &key);
    for (i = 0; i < 8; i++)
        if (a[i] == key) { pos = i; break; }
    printf("Linear search: index %d\n", pos);
    pos = -1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (a[mid] == key) { pos = mid; break; }
        else if (a[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    printf("Binary search: index %d\n", pos);
    return 0;
}
```

```
Search for: 48
Linear search: index 5
Binary search: index 5
```

`pos = -1` means "not found yet", since -1 is never a real index, and `break` leaves a loop as soon as the key is found.

- **Linear search** checks elements one by one. It works on any array.
- **Binary search** needs a sorted array. It checks the middle and discards the half that cannot hold the key, as you do with a dictionary. For 48: index 3 holds 27, too small, so search 4 to 7; index 5 holds 48. Eight elements need at most four checks.

Program 22: Matrix addition and multiplication

C

```
#include <stdio.h>
int main(void) {
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int b[2][3] = {{6, 5, 4}, {3, 2, 1}};
    int m[3][2] = {{7, 8}, {9, 10}, {11, 12}};
    int c[2][2] = {0};
    int i, j, k;
    printf("a + b:\n");
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++)
            printf("%d ", a[i][j] + b[i][j]);
        printf("\n");
    }
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            for (k = 0; k < 3; k++)
                c[i][j] += a[i][k] * m[k][j];
    printf("a x m:\n");
    for (i = 0; i < 2; i++)
        printf("%d %d\n", c[i][0], c[i][1]);
    return 0;
}
```

```
a + b:
7 7 7
7 7 7
a x m:
58 64
139 154
```

A **matrix** is a two-dimensional array: `a[2][3]` has 2 rows and 3 columns, and `a[i][j]` is row `i`, column `j`. Matrices of the same size add element by element. To multiply, the first matrix's columns must equal the second's rows (3 and 3). Each answer cell adds row `i` of `a` times column `j` of `m`: $1 \times 7 + 2 \times 9 + 3 \times 11 = 58$. `= {0}` starts `c` at zero.

CHAPTER 5

String programs

A **string** in C is a character array ending with the **null character** `'\0'`, which marks where the text stops. "Pune" needs five cells: four letters and `'\0'`.

SAFETY

Never use `gets`. It cannot know your array's size, so long input overwrites other memory; it was removed from the C standard in 2011. Use `fgets(s, sizeof s, stdin)`, which never writes past the size you give.

`fgets` keeps the Enter key as `'\n'` at the end. `s[strcspn(s, "\n")] = '\0'`; finds that newline and replaces it with the end marker.

Program 23: Length, palindrome and reverse of a string

C

```
#include <stdio.h>
#include <string.h>
int main(void) {
    char s[100], t;
    int len = 0, i, j, ok = 1;
    printf("Enter a word: ");
    fgets(s, sizeof s, stdin);
    s[strcspn(s, "\n")] = '\0';
    while (s[len] != '\0')
        len++;
    for (i = 0, j = len - 1; i < j; i++, j--)
        if (s[i] != s[j]) { ok = 0; break; }
    for (i = 0, j = len - 1; i < j; i++, j--) {
        t = s[i];
        s[i] = s[j];
        s[j] = t;
    }
    printf("Length = %d\n%s\n", len, ok ? "Palindrome" : "Not a palindrome");
    printf("Reversed: %s\n", s);
    return 0;
}
```

```
Enter a word: Kolkata
Length = 7
Not a palindrome
Reversed: atakloK
```

- The `while` loop walks along the array until it meets `'\0'`, just as the library function `strlen` does. Spaces count.
- The palindrome check starts `i` at the first letter and `j` at the last, compares them and moves both inwards; one mismatch is enough. Type `malayalam` and it says `Palindrome`. The check is case-sensitive, so `"Madam"` fails unless you convert with `tolower` first.
- The reverse uses the same two-index walk, swapping instead of comparing, and stops when the indexes meet.

Program 24: Count vowels and consonants

C

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int main(void) {
    char s[100];
    int i, v = 0, c = 0;
    printf("Enter text: ");
    fgets(s, sizeof s, stdin);
    for (i = 0; s[i] != '\0'; i++) {
        char ch = tolower(s[i]);
        if (strchr("aeiou", ch)) v++;
        else if (isalpha(ch)) c++;
    }
    printf("Vowels = %d, Consonants = %d\n", v, c);
    return 0;
}
```

Enter text: Incredible India
Vowels = 7, Consonants = 8

`strchr("aeiou", ch)` is non-zero when `ch` is in the vowel list. Any other letter is a consonant; spaces, digits and the newline are skipped.

CHAPTER 6

Functions and recursion

A **function** is a named block of code that does one job. Write it once and **call** it whenever you need it.

Part	Example	Purpose
Prototype	<code>int isPrime(int n);</code>	tells the compiler it exists
Definition	<code>int isPrime(int n) { ... }</code>	the actual code
Call	<code>isPrime(17)</code>	runs it with a value

The value you pass is an **argument**; the variable that receives it is a **parameter**. C copies the argument, which is **call by value**: changing `n` inside the function leaves the caller's variable alone.

Program 25: Prime check using a function

C

```
#include <stdio.h>
int isPrime(int n);

int main(void) {
    int n, i;
    printf("Enter a number: ");
    scanf("%d", &n);
    printf("%d is %s\n", n, isPrime(n) ? "prime" : "not prime");
    printf("Primes up to 30: ");
    for (i = 1; i <= 30; i++)
        if (isPrime(i)) printf("%d ", i);
    printf("\n");
    return 0;
}

int isPrime(int n) {
    int i;
    if (n < 2) return 0;
    for (i = 2; i * i <= n; i++)
        if (n % i == 0) return 0;
    return 1;
}
```

```
Enter a number: 91
91 is not prime
Primes up to 30: 2 3 5 7 11 13 17 19 23 29
```

A **prime** has exactly two factors, 1 and itself, so 0 and 1 are not prime. Testing up to the square root ($i * i \leq n$) is enough: in $91 = 7 \times 13$, the smaller factor turns up first. `return` ends the function at once, sending back 0 (not prime) as soon as a factor turns up, or 1 (prime) after the loop. Written once, `isPrime` serves both the single check and the range, and `?` : picks one of two words. The prototype lets `main` call a function defined below it.

Recursion

Recursion means a function calls itself on a smaller version of the same problem. It needs a **base case**, an input it answers directly, or it would call itself until the program crashes.

Program 26: Factorial and Fibonacci using recursion

C

```
#include <stdio.h>
unsigned long long fact(int n) {
    if (n <= 1) return 1;
    return n * fact(n - 1);
}

int fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}

int main(void) {
    int i;
    printf("5! = %llu\n", fact(5));
    for (i = 0; i < 10; i++)
        printf("%d ", fib(i));
    printf("\n");
    return 0;
}
```

```
5! = 120
0 1 1 2 3 5 8 13 21 34
```

- **Factorial:** $n! = n \times (n - 1)!$, with $1! = 1$ as the base case. `fact(5)` waits for `fact(4)`, and so on down to `fact(1)`, which returns 1; the answers then multiply back up: 2, 6, 24, 120.
- **Fibonacci:** `fib` follows the definition but is slow for large n , because it recomputes the same terms again and again; Program 16's loop is far faster.

TIP

For recursion, ask two questions: which input can I answer directly, and how does the answer for n use the answer for a smaller input?

CHAPTER 7

Pointers and structures

A **pointer** is a variable that stores an address, the location of another variable in memory. Two operators do all the work:

Operator	Read it as	Example
<code>&x</code>	address of x	<code>p = &x;</code>
<code>*p</code>	value at the address in p	<code>*p = 50;</code> changes x

Program 27: Pointers to a variable and an array

C

```
#include <stdio.h>
int main(void) {
    int x = 25, a[5] = {10, 20, 30, 40, 50}, sum = 0, i;
    int *p = &x;
    printf("x = %d, *p = %d\n", x, *p);
    *p = 50;
    printf("x is now %d\n", x);
    printf("Address of x: %p\n", (void *)p);
    p = a;
    for (i = 0; i < 5; i++)
        sum += *(p + i);
    printf("Sum of array = %d\n", sum);
    return 0;
}
```

```
x = 25, *p = 25
x is now 50
Address of x: 0x7ffd5e8a3c4c
Sum of array = 150
```

`int *p` declares a pointer to an `int`, and `p = &x` points it at `x`. Changing `*p` changes `x`, because they are the same box. The `%p` address will be different on your computer. An array's name acts as the address of its first element, so `p = a` points at `a[0]`. Adding 1 to a pointer moves it one element, not one byte, so `*(p + i)` equals `a[i]`: this is **pointer arithmetic**.

Program 28: Swap using pointers

C

```
#include <stdio.h>
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int main(void) {
    int x = 3, y = 8;
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

```
x = 8, y = 3
```

Swapping two glasses of juice needs an empty third glass, and `t` is that glass. With plain `int` parameters, a swap function would swap only its own copies. Passing addresses lets it reach the caller's variables: this is **call by reference**, and it is why `scanf` needs `&`.

SAFETY

Never use a pointer before giving it an address. An uninitialised pointer points somewhere random, and writing through it can crash the program.

Structures

A **structure** groups values of different types that belong together, such as a student's roll number, name and marks. Each item inside is a **member**, reached with a dot: `s.roll`.

Program 29: Student records and the topper

C

```
#include <stdio.h>
struct Student {
    int roll;
    char name[30];
    int marks[3];
};

int main(void) {
    struct Student c[3] = {
        {21, "Priya Sharma", {82, 91, 77}},
        {22, "Aman Verma", {75, 68, 80}},
        {23, "Fatima Khan", {90, 88, 94}}
    };
    int i, j, total[3] = {0}, top = 0;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++)
            total[i] += c[i].marks[j];
        printf("%d %-13s total %d, average %.2f\n",
            c[i].roll, c[i].name, total[i], total[i] / 3.0);
        if (total[i] > total[top]) top = i;
    }
    printf("Topper: %s\n", c[top].name);
    return 0;
}
```

```
21 Priya Sharma  total 250, average 83.33
22 Aman Verma   total 223, average 74.33
23 Fatima Khan  total 272, average 90.67
Topper: Fatima Khan
```

`struct Student { ... };` designs a new type but creates no variable. `c` is an array of three such records, like a class register, and each `{...}` fills one record's members in order.

`c[i].marks[j]` is mark `j` of student `i`. Keeping the topper's index lets us print the name; `%-13s` left-aligns it in 13 spaces, and dividing by `3.0`, not `3`, keeps the decimals.

File handling programs

Variables vanish when a program ends; a **file** keeps data on disk. File work always has four steps: open, check, read or write, close.

Mode	Meaning
"w"	write: creates the file, or empties it if it exists
"a"	append: adds to the end
"r"	read: the file must already exist

`fopen` returns a `FILE *`, a pointer to the file, or `NULL` if opening fails, so always check it.

Program 30: Write marks to a file and read them back

```

c
#include <stdio.h>
int main(void) {
    char name[30];
    int marks, count = 0, total = 0;
    FILE *fp = fopen("marks.txt", "w");
    if (fp == NULL) { printf("Cannot open file\n"); return 1; }
    fprintf(fp, "Aman 78\nFatima 92\nRohit 85\n");
    fclose(fp);
    fp = fopen("marks.txt", "r");
    if (fp == NULL) { printf("Cannot open file\n"); return 1; }
    while (fscanf(fp, "%29s %d", name, &marks) == 2) {
        printf("%-8s %d\n", name, marks);
        total += marks;
        count++;
    }
    fclose(fp);
    if (count > 0) printf("Average = %.2f\n", (double)total / count);
    return 0;
}

```

```

Aman    78
Fatima  92
Rohit   85
Average = 85.00

```

- `fprintf` works like `printf` but writes into the file; open `marks.txt` in Notepad to check. `fclose` finishes writing and releases the file.
- `fscanf` returns how many values it read, so the loop stops when it can no longer read a name and a number. `%29s` leaves room for `'\0'`.

- `(double)total` is a **cast**: it converts `total` so the division keeps decimals.
- With `"a"` in place of `"w"`, each run adds three more lines and the loop reads them all.

CHECK YOURSELF

Practice questions

1. What does `#include <stdio.h>` do?
2. Why do we write `&n`, not `n`, in `scanf("%d", &n)`?
3. What does `printf("%d", 7 / 2);` print, and why?
4. What is the difference between `=` and `==`?
5. Is 2100 a leap year? Why?
6. What happens if a `switch` case has no `break`?
7. Which loop always runs its body at least once?
8. Which expressions give the last digit of `n` and remove it?
9. Why does Program 14 copy `n` into `original`?
10. Why is testing divisors up to the square root enough for a prime check?
11. Why should `max` start as `a[0]`, not 0?
12. What must be true of an array before binary search?
13. Why is `gets` dangerous, and what replaces it?
14. What is a base case?
15. How does call by value differ from call by reference?
16. In `int *p = &x;`, what is in `p`, and what is `*p`?
17. How is a structure different from an array?
18. What does `"a"` mode do in `fopen`, and why check for `NULL`?

ANSWER KEY

Answers

1. It includes the standard input/output header, which declares `printf`, `scanf` and the file functions.
2. `scanf` needs the variable's address to store the value; `n` alone gives only its current value.
3. It prints 3: both are ints, so C does integer division and drops the fraction.
4. `=` stores a value; `==` compares two values.
5. No. It is divisible by 100 but not by 400.
6. It falls through and runs the next case too.

7. The `do-while` loop, which tests its condition after the body.
8. `n % 10` gives the last digit; `n /= 10` removes it.
9. The loop reduces `n` to 0, losing the value to compare with.
10. If `n = a × b`, one factor is at most the square root, so it turns up first.
11. If every value is negative, 0 would wrongly beat them all.
12. It must be sorted (ascending, for this version).
13. It cannot limit how much it reads, so long input overflows the array. Use `fgets`.
14. The input a recursive function answers directly, which stops the recursion.
15. Call by value passes a copy, so the caller's variable cannot change; call by reference passes the address, so it can.
16. `p` holds the address of `x`; `*p` is the value stored there, which is `x`.
17. An array holds values of one type, reached by index; a structure groups different types, reached by member name.
18. It adds to the end of the file without erasing it. `fopen` returns `NULL` on failure, and using `NULL` can crash the program.

WORD LIST

Glossary

Argument

a value passed to a function when it is called.

Array

values of one type stored under one name.

Base case

the input recursion answers without calling itself.

Cast

a type in brackets that converts a value:
`(double)x`.

Compiler

a program that turns source code into an executable.

Format specifier

a code such as `%d` that names a value's type.

Function

a named block of code that does one job.

Index

an element's position in an array, counted from 0.

Macro

a `#define` name that the preprocessor replaces.

Null character

`'\0'`, which marks the end of a string.

Parameter

a function's variable that receives an argument.

Pointer

a variable that stores another variable's address.

Preprocessor

the step before compiling that handles `#` lines.

Recursion

a function calling itself on a smaller problem.

Structure

a user-made type that groups different types.



Gurukul Technicals Institute

Computer education institute in Delhi, with authorised member centres, robotics classes and science projects, working models and DIY kits.

COURSES

Computer & Office Skills · Accounting & Business Software · Programming & Web Development · Design & Digital Skills · Hardware & Networking · Robotics & Coding · Other Programmes

PROJECTS AND KITS

Ready-made school projects · DIY and STEM kits · Robotics and coding · Components and spares · Engineering projects

CONTACT

Head office: A-10, Chief Commissioner Colony, opposite Rana Pratap Bagh, Delhi 110007

Phone: 011 2721 1512 · **Mobile and WhatsApp:** +91 97116 99511

Email: engineergurukul@yahoo.co.in

Website: engineergurukul.com · **Instagram:** [@science_projects_club](https://www.instagram.com/science_projects_club)

More free ebooks for computer courses, robotics and science projects: engineergurukul.com/resources/ebooks/

Free to read, print and share. Not for sale.