CS 137

Programming Principles

Chapter 1

Introduction to C

Victoria Sakhnini

Table of Contents

1. Compiling and Running Our First Program

Let us begin with a program that displays `"Programming is fun"` in your window. The following is a C program to accomplish this task:

```
#include <stdio.h>

int main(void)
{
    printf("Programming is fun.\n");
    return 0;
}
```

- **Lowercase and uppercase letters are distinct.**
- Indentation is not mandatory, but it makes your code more readable. Use Tab characters to indent lines.
- `#include <stdio.h>` should be included at the beginning of just about every program you write. It tells the compiler information about the `printf()` output routine.
- `int main(void)` informs the system that the name of the program is `main()`, which returns an integer value, abbreviated `"int"`.
- `main` is a special name that indicates precisely where the program is to begin execution.
- `void` means that the `main` function takes no arguments. The parentheses specify the start and the end of `main`.
- `printf` prints the followed string to screen.
- `\n` is the *newline* character.
- All program statements in C must be terminated by a semicolon (`;`).
- `return 0;` means finish execution and return to the system a status value of 0. Zero indicates the program was completed successfully; anything non-zero is a failure. You can also omit the return statement — the C11 standard defaults to returning zero.

After writing your program in any editor, save the file with the extension `.c`

Assuming the program above is typed into a file called `prog1.c`, compile it with the GNU C compiler:

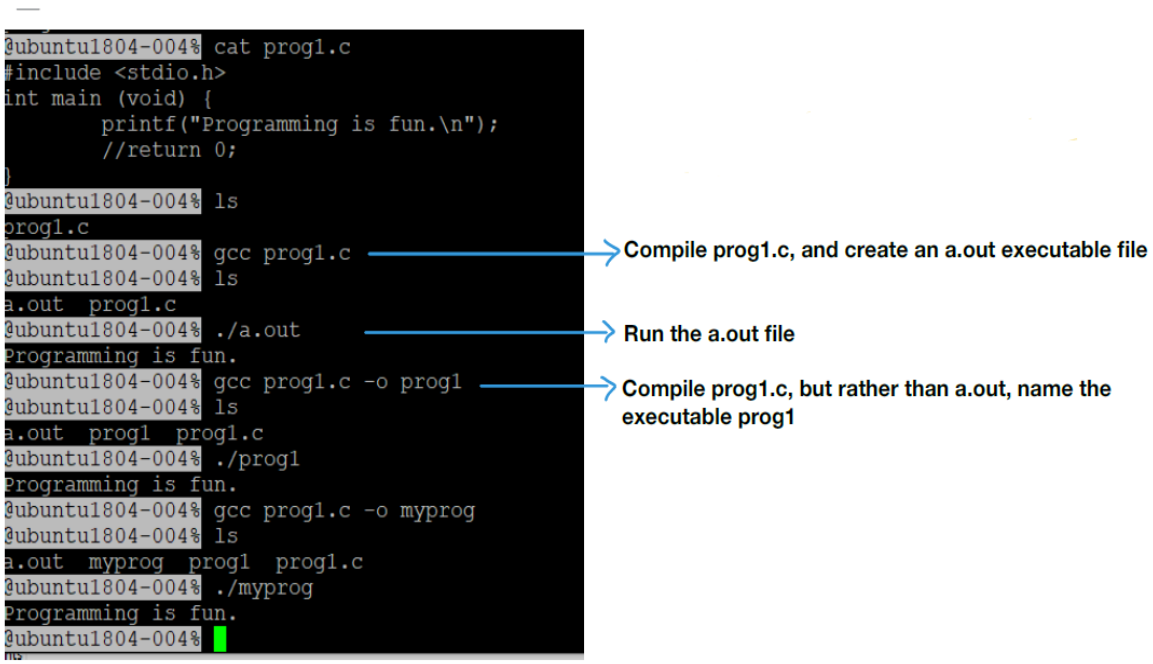
```
gcc prog1.c
```

If no errors are found, the compiler creates an executable called `a.out` by default. Run it with:

```
./a.out
```

You can specify a different name for the executable using the `-o` option:

```
gcc -o prog1 prog1.c
./prog1
```



> ? What is the exact output of the following program? Try to figure it out manually before running the code.

```
#include <stdio.h>

int main(void)
{
    printf("Programming is fun.");
    printf("And programming in C is even more fun.\n");
    printf("This\nis\neven\nmore\nfun!");
    return 0;
}
```

Programming is fun.And programming in C is even more fun.
This
is
even
more
fun!

> There are some options for an IDE if you prefer GUI-based learning: Eclipse, VSCode, and Pelles C are popular choices. You can use whatever editor you like.

Back to returning success: By convention, `return 0;` is a success and anything non-zero is a failure. Alternatives include:

```c
#include <stdlib.h>   // tells the compiler about EXIT_SUCCESS and
EXIT_FAILURE

int main(void)
{
    return EXIT_SUCCESS;   // which is 0
    // or: return EXIT_FAILURE;   // which is 1
}
```

2. Comments

A comment statement is used for documentation to enhance a program's readability. There are two ways to insert comments in C:

```c
// This is a single-line comment

/* This is a
   multi-line comment
   spanning three lines */
```

3. Variables, Data Types, and Arithmetic Expressions

3.1 Variables in C

A **variable** is a name given to a storage area that a program can manipulate. Each variable in C has a specific type that determines the valid range of values and the size and layout of the variable's memory.

Rules for variable names:

- Must begin with a letter or an underscore (_).
- After the first character, can be letters, digits, or underscores.
- Case sensitive: `myVar` and `myvar` are different variables.
- Cannot be keywords (e.g., `int`, `while`, `return`).

There are five basic data types in C:

Type	Description	Example Values
int	Integer (whole numbers)	0, 42, -17, 1000
float	Single-precision floating point	3.14, -0.008, 23.0
double	Double-precision floating point	3.14159265358979
char	Single character	'a', ';', '8', '\n'
_Bool	Boolean value (0 = false, 1 = true)	0, 1

Type Specifiers: `long`, `long long`, `short`, `unsigned`, `signed` — these are placed directly before the basic type and change the range of possible values.

> 📌 In C, the integer takes 2 bytes for a 32-bit compiler and 4 bytes for a 64-bit compiler.

Type	Storage Size	Value Range	Numeric
`char`	1 byte	$[-128, 127]$	$[-2^7, 2^7 - 1]$
`unsigned char`	1 byte	$[0, 255]$	$[0, 2^8 - 1]$
`int`	2 bytes	$[-32768, 32767]$	$[-2^{15}, 2^{15} - 1]$
`int`	4 bytes	$[-2147483648, 2147483647]$	$[-2^{31}, 2^{31} - 1]$
`unsigned int`	4 bytes	$[0, 4294967295]$	$[0, 2^{32} - 1]$
`short int`	2 bytes	$[-32768, 32767]$	$[-2^{15}, 2^{15} - 1]$
`unsigned short int`	2 bytes	$[0, 65535]$	$[0, 2^{16} - 1]$
`long int`	4 bytes	$[-2147483648, 2147483647]$	$[-2^{31}, 2^{31} - 1]$
`long int`	8 bytes	$[-9.22 \cdot 10^{18}, 9.22 \cdot 10^{18} - 1]$	$[-2^{63}, 2^{63} - 1]$
`long long int`	8 bytes	$[-9.22 \cdot 10^{18}, 9.22 \cdot 10^{18} - 1]$	$[-2^{63}, 2^{63} - 1]$
`unsigned long long int`	8 bytes	$[0, 1.84 \cdot 10^{19} - 1]$	$[0, 2^{64} - 1]$

3.2 Back to Printing (Output)

We previously learned that `printf("string")` prints a string to the screen. A more powerful format allows printing variables:

```
printf(format_string, argument(s));
```

The format string contains **placeholders** (also called format specifiers) that are replaced by the corresponding argument values:

Placeholder	Meaning
%d	Signed integer
%u	Unsigned integer
%f	Float / double
%c	Character
%s	String
%lf	Double (for scanf)
%ld	Long integer

Example:

```
#include <stdio.h>

int main(void)
{
    int a = 3;
    printf("1 + 2 = %d and %d + %d = 9\n", a, a, 2*a);
    return 0;
}
```

Output: `1 + 2 = 3 and 3 + 6 = 9`

The first `%d` was replaced by `a` (3), the second `%d` by `a` (3), and the third `%d` by `2*a` (6).

> ? What happens if the number of placeholders is less than the number of arguments, or vice versa?

- If there are more arguments than placeholders: the extra arguments are silently ignored.
- If there are more placeholders than arguments: arbitrary (garbage) values are printed.
  The compiler may issue a warning but will NOT produce an error.

3.3 Basic Operations

Operator	Description	Example
+ - *	Add / Subtract / Multiply	a + b, a - b, a * b
/	Divide (integer ÷ integer = integer; decimal part is lost!)	7 / 2 → 3
%	Modulo: remainder of division	7 % 2 → 1
( )	Parentheses for grouping	(a + b) * c

All operators follow **BEDMAS** (Brackets, Exponents, Division/Multiplication, Addition/Subtraction) and then left-to-right associativity.

> ⚠ There is NO exponentiation operator in C. Use repeated multiplication, or the pow() function from <math.h>.

> ? What is the value of a / b * b + a % b when a and b are positive integers?

a (this is a mathematical identity: (a/b)*b + a%b == a for integer division)

3.4 Assignment Operators

- Done *last* in the order of operations.
- Right-associative: the right-hand side is evaluated first, then assigned.
- Syntax: `variable = expr;  // e.g. a = 3;`
- Combined assignment operators:

Operator	Meaning	Example
+=	a = a + expr	a += 2 → a = a + 2
-=	a = a - expr	a -= 3 → a = a - 3
*=	a = a * expr	a *= b → a = a * b
/=	a = a / expr	a /= b + c → a = a / (b + c)
%=	a = a % expr	a %= 5 → a = a % 5

⚠ Order matters! a += 2 and a =+ 2 are different. The first adds 2 to a; the second assigns +2 to a.

Always assign a value to a variable **before** using it. Do not assume `int x;` gives `x = 0`; the value is undefined.

3.5 Increment/Decrement Operators

Unary operators that add or subtract one. There are two forms: **prefix** (applied before use) and **postfix** (applied after use).

Operator	Form	Behaviour
++a	Prefix	Increment a by 1, then use the new value
--a	Prefix	Decrement a by 1, then use the new value
a++	Postfix	Use the current value of a, then increment by 1
a--	Postfix	Use the current value of a, then decrement by 1

Example:

```
#include <stdio.h>

int main(void)
{
    int a = 3, b = 1;
    b += a++ + 2;
    printf("a = %d\nb = %d\n", a, b);
    return 0;
}
```

Trace: initially a=3, b=1. The expression `b += a++ + 2` evaluates as `b = b + a + 2 = 1 + 3 + 2 = 6`, then `a` is incremented to 4.

Output: `a = 4 b = 6`

? What is the output of this program? (Trace manually before running)

```
#include <stdio.h>
int main(void)
{
    int a = 1, b = 2, c = 3;
    a = b += c;
    printf("%d\n", a);
}
```

5 (c=3, b += c means b = b+c = 5, then a = b = 5)

? What is the output of this program?

```
#include <stdio.h>
int main(void)
{
    int a = 10, b = 15;
    a += ++b;              // prefix: b becomes 16 first, then a = 10+16 = 26
    printf("%d\n", a);    // 26
    a = 10; b = 15;
    a += b++;              // postfix: a = 10+15 = 25, then b becomes 16
    printf("%d\n", a);    // 25
    a = 10; b = 15;
    a += --b;              // prefix: b becomes 14 first, then a = 10+14 = 24
    printf("%d\n", a);    // 24
    return 0;
}
```

```
26
25
24
```

Tricky: What is the output of this program? Do your own investigation.

```
int main(void) {
    int x = 10;
    int y = printf("%d %d\n", printf("%d ", x), x*4);
    printf("%d\n", y);
    return 0;
}
```

3.6 Relational and Logical Operators

Relational operators: `==` (equal) `!=` (not equal) `>` (greater than) `<` (less than) `>=` (greater or equal) `<=` (less or equal)

Logical operators: `!` (NOT) `||` (OR) `&&` (AND)

All relational and logical expressions return `1` for true and `0` for false.

a	b	!a	a || b	a && b
T	T	F	T	T
T	F	F	T	F
F	T	T	T	F
F	F	T	F	F

Short-circuit evaluation: `&&` and `||` only evaluate the right-hand side if necessary.

Example:

```
int a = 0;
a != 0 && 4/a > 2;  // Safe! Second condition is never evaluated
                    // because a != 0 is false — C skips 4/a
```

Short-circuit evaluation prevents division-by-zero errors in the example above, because 4/a is never reached when a == 0.

3.7 C Operator Precedence

Operators with higher precedence are evaluated first. The table below summarises the most important operators from highest to lowest precedence:

Precedence	Operator(s)	Associativity
1 (highest)	( ) [ ] -> .	Left to right
2	++ -- (postfix)	Left to right
3	++ -- (prefix) ! ~ (unary +/-)	Right to left
4	* / %	Left to right
5	Binary + -	Left to right
6	<< >>	Left to right
7	< <= > >=	Left to right
8	== !=	Left to right
9	& (bitwise AND)	Left to right

Precedence	Operator(s)	Associativity
10	^ (bitwise XOR)	Left to right
11	| (bitwise OR)	Left to right
12	&&	Left to right
13	||	Left to right
14 (lowest)	= += -= *= /= %=	Right to left

Source: http://en.cppreference.com/w/c/language/operator_precedence

3.8 Bit-Wise Operators

3.8.1 The Basics of Bits

- At the smallest scale in a computer, information is stored as **bits** and **bytes**.
- A **bit** can hold either `1` or `0`.
- A **byte** consists of **eight bits**.
- All computer data is represented in **binary** (base 2).

3.8.2 Converting Binary to Decimal

This is a positional number system. Each bit position represents a power of 2:

Bit position (from right)	7	6	5	4	3	2	1	0
Power of 2	2^7=128	2^6=64	2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1

Examples:

- 01001100_2 = 0+64+0+8+4+0+0 = **76_{10}**
- 11111111_2 = 128+64+32+16+8+4+2+1 = **255_{10}**

3.8.3 Converting Decimal to Binary

Repeatedly divide by 2, recording the remainders. Read the remainders from **bottom to top**:

```
Convert 38 to binary:

  38 ÷ 2 = 19  remainder 0
  19 ÷ 2 = 9   remainder 1
   9 ÷ 2 = 4   remainder 1
   4 ÷ 2 = 2   remainder 0
   2 ÷ 2 = 1   remainder 0
   1 ÷ 2 = 0   remainder 1

  Reading bottom to top: 38₁₀ = 00100110₂
```

3.8.4 Bitwise Operators

Suppose `unsigned char a = 5 (00000101), b = 3 (00000011)`

Symbol	Operation	Example	Result (binary)	Result (decimal)
&	Bitwise AND	c = a & b;	00000001	1
|	Bitwise Inclusive OR	c = a | b;	00000111	7
^	Bitwise Exclusive OR	c = a ^ b;	00000110	6
~	Bitwise NOT	c = ~a;	11111010	250 (as unsigned char)
<<	Left shift by 3	c = a << 3;	00101000	40
>>	Right shift by 2	c = a >> 2;	00000001	1

⚠ Be careful using bitwise operators with signed values — shifting can produce unexpected results depending on the compiler.

Example program:

```
#include <stdio.h>

int main(void)
{
    unsigned char a = 5;
    unsigned char b = 3;
    printf("%d\n", a & b);    // 1
    printf("%d\n", a | b);    // 7
    printf("%d\n", a ^ b);    // 6
    printf("%d\n", ~a);       // 250 (as unsigned char) / -6 (as signed)
    printf("%d\n", a << 3);   // 40
    printf("%d\n", a >> 2);   // 1
    return 0;
}
```

3.9 Two's Complement Form

Signed integers are represented in **two's complement** form. To interpret a binary number as a signed integer: if the leading (leftmost) bit is `0`, it is a non-negative number. If the leading bit is `1`, subtract 2^n where n is the number of bits.

To negate a value (compute -x):

1. Take the complement (flip) of all bits.
2. Add 1.

Example: compute -38_{10} in one byte (8 bits):

```
Step 1: Write 38 in binary:          00100110
Step 2: Complement all bits:         11011001
Step 3: Add 1:                       11011010   ← this represents -38

Verify: -2⁷+2⁶+2⁴+2³+2¹ = -128+64+16+8+2 = -38  ✓
```

Shortcut: Find the rightmost `1` bit and flip all bits to its left.

```
  38 =  00100110
           ↑ rightmost 1
 -38 =  11011010   (bits to the left of rightmost 1 are flipped)
```

💡 Arithmetic works naturally in two's complement — any final carry-out is simply discarded. Watch out for overflow errors!

3.10 Arithmetic Overflow and Underflow

An **integer overflow** occurs when you attempt to store a value larger than the maximum (or smaller than the minimum) the variable can hold. The C standard defines this as **undefined behaviour**.

In practice:

- For **unsigned integers**: the value wraps around (adding 1 to the maximum gives 0).
- For **signed integers**: undefined behaviour — most compilers wrap but this is not guaranteed.

Overflows cannot be detected *after* they have happened, so there is no reliable way to tell if a previously computed result is correct.

🎮 Real-world example: In Final Fantasy on the SNES (16-bit console), the final boss had 20,000 HP stored as a short int. Using the Elixir (a full-heal item) caused its HP to overflow to a negative value (-25,536), immediately killing it!

3.11 Endianness

Integers are usually 4 bytes. There are two conventions for storing multi-byte values in memory:

Endianness	Description	Used in
Big-endian	Most significant byte at the LOWEST memory address	MIPS, network protocols
Little-endian	Most significant byte at the HIGHEST memory address	x86, most modern PCs

In C, byte ordering is **implementation-defined** (depends on the computer and compiler). For most portable C programs, byte ordering is irrelevant.

3.12 De Morgan's Theorem

De Morgan's Theorem provides rules for simplifying logical expressions:

```
!(A && B)  ≡  !A || !B
!(A || B)  ≡  !A && !B
```

Example — the following two C expressions are equivalent:

```
!(a > 0) || !(b > 0)
!(a > 0 && b > 0)
```

4. Input – scanf

The `scanf` function reads formatted input from the standard input (keyboard):

```
scanf("%d", &a);   // reads an integer and stores it in a
```

- `&a` refers to the **memory address** of `a`. We will cover this in detail later, but you must include the `&` for `scanf` to work.
- `scanf` ignores all leading whitespace and newline characters.
- If the user doesn't enter the expected type, the previous value remains unchanged and `scanf` returns failure.

Reading formatted input: You can specify an exact input format:

```
scanf("%d/%d", &num, &denom);   // expects input like: 3/4
```

⚠ The input format must match exactly. If the user types '3 4' instead of '3/4', the scanf will fail.

Return value: `scanf` returns the number of items successfully read.

```
int x, y;
int z = scanf("%d %d", &x, &y);
// If user types: 1254 654
// z == 2  (two items read successfully)
```

Full example:

```
#include <stdio.h>

int main(void)
{
    int a, b, c;
    printf("Enter an integer:\n");
    scanf("%d", &a);
    printf("Enter a fraction (num/denom):\n");
    scanf("%d/%d", &b, &c);
    printf("****************\n");
    printf("%d %d %d\n", a, b, c);
    return 0;
}
```

5. Additional Examples (Enrichment)

Example 1 — Area and Circumference of a Disc:

```
/*
 * Calculates and displays the area and circumference
 * of a metal disc given the radius in centimeters.
 */
#include <stdio.h>
#define PI 3.14159

int main(void)
{
    char  part_id1, part_id2, part_id3;
    double radius, area, circum;

    printf("Enter a 3-character part i.d.> ");
    scanf("%c%c%c", &part_id1, &part_id2, &part_id3);

    printf("Enter radius (in centimeters)> ");
    scanf("%lf", &radius);

    area   = PI * radius * radius;
    circum = 2 * PI * radius;

    printf("For part %c%c%c, ", part_id1, part_id2, part_id3);
    printf("the disc area is %f cm^2.\n", area);
    printf("The circumference is %f cm.\n", circum);
    return 0;
}
```

Example 2 — Using Casts to Prevent Integer Division:

```c
/*
 * Using casts to prevent integer division
 */
#include <stdio.h>

int main(void)
{
    int    total_score, num_students;
    double average;

    printf("Enter sum of students' scores> ");
    scanf("%d", &total_score);
    printf("Enter number of students> ");
    scanf("%d", &num_students);

    average = (double)total_score / (double)num_students;
    printf("Average score is %.2f\n", average);
    return 0;
}
```

Casting both operands to double before division ensures a floating-point result. Without the cast, integer division would discard the decimal portion.

6. Extra Practice Problems

Try to manually figure out the answers before running the code or looking at the provided solutions.

Problem 1

```c
#include <stdio.h>
int main(void)
{
    int a, b;
    a = b = 10;
    printf("a = %d\tb = %d\n", a, b);
    printf("a++ = %d\t++b = %d\n", a++, ++b);
    printf("a = %d\tb = %d\n", a, b);
    return 0;
}
```

a = 10 b = 10
a++ = 10 ++b = 11 (a++ uses 10 then increments; ++b increments then uses 11)
a = 11 b = 11

Problem 2

```
#include <stdio.h>
int main(void)
{
    int a, b;
    a = b = 10;
    printf("%d\n", a == b);    // equality
    printf("%d\n", a != b);    // inequality
    printf("%d\n", a << 1);    // left shift
    printf("%d\n", a >> 2);    // right shift
    printf("%d\n", a >> 3);
    printf("%d\n", b >> 5);
    return 0;
}
```

```
1   (10 == 10 is true)
0   (10 != 10 is false)
20   (10 << 1 = 10 × 2 = 20)
2   (10 >> 2 = 10 ÷ 4 = 2, integer)
1   (10 >> 3 = 10 ÷ 8 = 1, integer)
0   (10 >> 5 = 10 ÷ 32 = 0)
```

Problem 3

Complete the program to print the minimum number of buses required to accommodate any number of people. Each bus holds 50 people.

Participants	Buses required
0	0
45	1
50	1
51	2
99	2
100	2
113	3

```
int main(void)
{
    int par, busses;
    printf("Enter the number of participants:\n");
    scanf("%d", &par);
    // Your code here
    printf("Number of busses required: %d\n", busses);
    return 0;
}
```

```
busses = par / 50 + ((par % 50) != 0);
```

Problem 4

Convert the following decimal numbers to 8-bit binary: `27`, `78`, `99`

27 = 00011011
78 = 01001110
99 = 01100011

Problem 5

Convert the following binary numbers to decimal: `01001101`, `11001101` (signed), `01011110`

01001101 = 77
11001101 = -51 (two's complement signed byte)
01011110 = 94

Problem 6

```
// There are TWO errors in this program. Find them!
#include <stdio.h>
int main(void)
{
    int result;
    int a = 8, b = 4;
    result = a \ b;          // Error 1: should be / not \
    printf("%d\n", result);
    return 0;
}
// (result was also not properly initialized before potential error paths)
```

Error 1 (compile-time): Line using '\' — should be '/' for division.
Error 2 (run-time): 'result' is not initialized before use in some code paths.

Problem 7

```
int mystery(int num)
{
    return (num % 4 == 3);
}
```

Returns 1 (true) if num equals 4k+3 for some integer k, i.e., if num mod 4 equals 3.

Problem 8

```
int fun_1(int num)
{
    if ((num & (num - 1)) != 0)
        return 1;
    return 0;
}
```

Returns 1 if num is NOT a power of 2 (or if num == 0).
A power of 2 has exactly one '1' bit; num & (num-1) clears that bit — it equals 0 only for powers of 2.

Problem 9

Write a program that reads two integers `a` and `n` and rotates the bits of `a` to the **right** by `n` positions.

Input	Output
123 23	62976
5 14	1310720

Problem 10

(a) `bitXor(x, y)` — Implement XOR using only `~` and `&`

(b) `tmin()` — Return the minimum two's complement 32-bit integer

(c) `isTmax(x)` — Return 1 if x is the maximum two's complement 32-bit integer