C Programming Language: Chapter Notes

Daniel Duggan

November 30, 2025

1 Chapter 1: Overview of Some Basic Concepts

1.1 Operator v Statement

An operator is a symbol that acts of data in some way. For example +, -, % etc.

A statement is a complete instruction that tells a program to do something e.g. an if statement, a while statement etc.

1.2 Strongly Typed v Weakly Typed Language

A strongly typed language is one whereby the type of every value is strictly enforced, and does not allow one to mix types without an explicit conversion. Ada is a strongly typed language. C is not.

```
#include <stdio.h>

x = float
y = string

z = x + y \\ Error in STL

z = x + float(y) \\Acceptable in most STL
```

Listing 1: Example of Strongly Typed Language

1.3 Static v Dynamic Typed

Static means that when a variable is assigned a type, it keeps that type and its type is checked at compile time. Example languages are C, Rust, Go and Ada

```
int x = 5;
x = "hello";   // compile-time error
```

Listing 2: Example of Static Typed Language

So, strongly typed –> cannot mix variable types and operations
Statically typed –> Once a variable is assigned a type it maintains that type and this is checked at compile time.

Dynamically typed means a varibale can be assigned one type somewhere, and re-assigned another type later on. Example languages are Python, JavaScript, Ruby.

```
int x = 5;
x = "hello";   // NO compile-time error
```

Listing 3: Example of Static Typed Language

1.4 Procedural v Functional Language

Procedural languages are based around writing functions that operate on data. Examples are C, Python, Ada. Use looping structures, and can be viewed as carrying out programs step-by-step.
Functional languages are based on mathematical functions and expressions. Do not have, e.g., for loops, instead using recursion. Example languages are Haskell, OCaml. They are less step-by-step and more based around writing and calling functions which are the mian building blocks.

1.5 Imperative v Declarative Programming

Imperative programming focuses on how to do something by providing step-by-step instructions. Example languages are C, Python, Ada.
Declarative programming focuses on what the desired outcome is without specifying the exact steps e.g. Coq (Gallina), SQL, Haskell.
Difference between i– and –i:

```c
int i = 5;

int a = i--;    // a = 5, i becomes 4 (post-decrement)
int b = --i;    // i becomes 3, b = 3 (pre-decrement)
```

Listing 4: Pre- vs Post-Decrement

2 Chapter 2 Notes: Input and Output Functions

C provides powerful yet simple functions for performing input and output operations. Two of the most commonly used are `printf()` and `scanf()`, both defined in the `<stdio.h>` header file.

2.1 The `printf()` Function

The `printf()` function is used to send formatted output to the standard output stream (usually the screen). Its general form is:

```c
 printf("control string", expressions);
```

Listing 5: General form of printf()

The control string contains ordinary characters, which are printed as they appear, and conversion specifications, each beginning with a `%` character, which tell `printf()` how to format and display the corresponding expressions.

Specifier	**Meaning**
%d	prints an integer value
%f	prints a floating-point value
%c	prints a single character
%s	prints a string of characters
%02d	prints an integer padded with zeros (e.g. 05)
%.2f	prints a floating-point number to 2 decimal places
%.3f	prints a floating-point number to 3 decimal places
%6.2f	prints a floating-point number in a field width of 6, with 2 digits after the decimal point
%5d	prints an integer in a field width of 5 characters (right-aligned by default)

The general form of a format specification is:

$$\%[\textit{flags}][\textit{width}][.\textit{precision}][\textit{length}]\textit{specifier}$$

where:

- **width** specifies the minimum number of characters to print.
- **precision** controls the number of digits after the decimal point for floating-point numbers, or the maximum number of characters printed for strings.
- **0** flag pads numbers with zeros instead of spaces.
- The output is right-justified by default; use the `-` flag for left-justification.

For example:

```c
#include <stdio.h>

int main(void) {
    int hour = 9, minute = 5;
    float temperature = 18.437;

    printf("The time is %d:%02d\n", hour, minute);
    printf("Temperature: %.2f\\textdegree C\n", temperature);
    printf("Temperature (3dp): %.3f\\textdegree C\n", temperature);
    printf("Right-aligned field (width 6): %6.2f\\n", temperature);
    printf("Left-aligned field: %-6.2f\\n", temperature);

    return 0;
}
```

Listing 6: Example of printf() usage

This will print:

```
The time is 9:05
Temperature: 18.44°C
Temperature (3dp): 18.437°C
Right-aligned field (width 6):  18.44
Left-aligned field: 18.44
```

Here:

- `%.2f` rounds to 2 decimal places.
- `%.3f` rounds to 3 decimal places.
- `%6.2f` ensures the total printed field occupies at least 6 characters.
- `%-6.2f` left-aligns the number within that 6-character field.
- `%02d` pads with leading zeros if the integer has fewer than 2 digits.

2.2 The `scanf()` Function

The `scanf()` function reads formatted input from the standard input stream (usually the keyboard). Its general form is:

```c
scanf("control string", address list);
```

Listing 7: General form of scanf()

The control string works similarly to that of `printf()`, using format specifiers to determine the type of data being read. Each variable to be read must be preceded by an `&` (address-of operator), because `scanf()` needs to know where to store the input value.

Specifier	Meaning
%d	reads an integer value
%f	reads a floating-point value
%c	reads a single character
%s	reads a string of characters (stops at whitespace)
%lf	reads a double-precision floating-point number
%3d	reads at most 3 digits of an integer (useful for limiting input length)
%5s	reads at most 5 characters into a string (prevents overflow)

The general form of a `scanf()` conversion specification is:

$$\%[width][length]specifier$$

where:

- **width** limits the number of characters read for that input item.
- **length** is used for type modifiers such as `l` (for long) or `lf` (for double).

For example:

```c
#include <stdio.h>

int main(void) {
    int hour, minute;
    float temperature;

    printf("Enter a 24-hour time (hh:mm): ");
    scanf("%d:%d", &hour, &minute);

    printf("Enter current temperature: ");
    scanf("%f", &temperature);

    printf("You entered %02d:%02d and %.3f\\textdegree C\n", hour, minute, temperature);

    return 0;
}
```

Listing 8: Example of scanf() usage

If the user inputs:

```
09:30
18.437
```

the program will output:

```
You entered 09:30 and 18.437°C
```

The precision specifier in `scanf()` (such as `%3d` or `%5s`) does not control rounding or decimal places — instead, it limits the number of characters `scanf()` reads from the input buffer. This prevents reading too many characters into a variable or string.

2.3 Important Notes and Common Pitfalls

- **Always match format specifiers to variable types.** Using the wrong one (for example, reading a float with `%d`) can lead to undefined behaviour.
- **Use `&` in `scanf()`.** Forgetting it causes the program to overwrite memory at an undefined location, often leading to crashes. `scanf("%d", n);` is incorrect — it should be `scanf("%d", &n);`.
- **Width and precision serve different purposes.** In `printf()`, width controls field size and precision controls decimal places. In `scanf()`, width controls the *maximum number of characters read*.
- **Whitespace handling:** `scanf()` skips leading whitespace (spaces, newlines, tabs) for most format specifiers except `%c`, which reads the next character literally — even if it's a newline.
- **`%s` stops at whitespace.** It reads characters until the first space, tab, or newline. To read full lines of text including spaces, use `fgets()` instead.
- **Input validation:** If the user types a non-numeric value when `%d` or `%f` is expected, `scanf()` fails and leaves variables unchanged. Always check its return value (the number of items successfully read).
- **Clearing the input buffer:** After using `scanf()`, unwanted newline characters can remain in the input buffer and affect the next input. To safely handle mixed input (e.g. reading both numbers and characters), use:

  ```c
  while (getchar() != '\n'); // clear leftover characters

  ```

- **Printing special characters:** To print a percent sign, use `%%`. For example:

  ```c
  printf("Progress: 100%% complete\n");

  ```

2.4 Summary

- `printf()` is used for formatted output; `scanf()` for formatted input.
- Both functions use format specifiers beginning with `%` to handle different data types.
- Field width and precision control output spacing and rounding for `printf()`, but input limits for `scanf()`.
- Each variable in `scanf()` must be preceded by `&`.
- `scanf()` skips whitespace for most specifiers but not for `%c`.
- `printf()` allows control of output precision, padding, and alignment.
- Always ensure the types of variables match their format specifiers.

3 Chapter 5 Notes: Selection Statements

3.1 Relational Operators

Symbol	**Meaning**
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

The relational operators produce 0 when false and 1 when true e.g. `1 < 2` produces 1 (true). `1> 2` produces 0 (false).
The expression `i < j < k` doesn´t have the meaning one may expect, as < is left associative. It is equivalent to `(i<j)<k` i.e. it tests whether i is less than j, and then the 1 or 0 is compared with the value of k.
To test if j lies within i and k, the correct expression is `i<j && j<k`.

3.2 Equality Operators

Symbol	**Meaning**
==	equal to
!=	not equal to

Like the relational operators, the equality operators are left associative and produce 0 or 1. However, they have *lower precedence* than the relational operators, meaning the expression `i<j == j<k` is equivalent to `(i<j) == (j<k)`.

3.3 Logical Operators

Symbol	**Meaning**
!	logical negation
&&	logical *and*
||	logical *or*

The logical operators often produce 0 or 1 as there result. They behave as follows:

- `!expr` has the value 1 if *expr* has the value 0.
- `expr1 && expr2` has the value 1 if both expressions are non-zero.
- `expr1 || expr2` has the value 1 if either `expr1` or `expr2` (or both) has a non-zero value.

3.4 if statement

An if statement takes the form

```
if (expression) statement
```

Listing 9: if statement

If you want numeroues statements, one can use the curly braces:

```
if (line_num == max_lines){

  line_num = 0;
  page_num++;
}
```

Curly braces can be add to if and if else statements even when not strictly required to aid in readability:

```
#include <stdio.h>

int main(void) {
    int score = 85;

    // Even though braces aren't strictly required for single statements,
    // adding them improves clarity and helps prevent future errors.
    if (score >= 90) {
        printf("Grade: A\n");
    } else if (score >= 80) {
        printf("Grade: B\n");
    } else {
        printf("Grade: C or below\n");
    }

    return 0;
}
```

Listing 10: Using curly braces for readability in if-else statements

3.4.1 Dangling else Problem

An else statement belongs to the nearest if statement that hasn´t already been paired with an else. Using curly braces aids in recognising which else statement belongs to which if statement.

3.5 Conditional Expressions

C provides an operator that allows an expression to produce one of two values depending on the value of a condition. This operator cosists of two symbols (? and :) which have to be used in the following way.

```
expr1 ? expr2 : expr3
```

Listing 11: Using curly braces for readability in if-else statements

The expressions can be of any type. This operator is unique in C in that it requires 3 operands instead of the usual one or two and is therefore called the ternary operator. It is read as *"if expr1 thhen expr2 else expr3"*.
expr1 is evaluated, if it is not 0, then expr2 is evaluated and it´s value is the value of the expression. If expr1 is 0, then the value of expr3 is the value of the expression as a whole. For example:

```
i = 1;
j = 2;
k = i>j ? i : j; // if i is greater than j, k is value of i (1), else it is value of
    j (2).
                 // so here, k is 2
k = (i>=0 ? i : 0) + j; // k is now 3 (i is greater than 0, so expr2 is evaluated, and
    has +j added to it)
```

3.6 Boolean Types

None in C89. Need to define a macro ifelsebraces

```
#define BOOL int
.
.
.
BOOL flag
```

In C99 there are boolean types declared basicstyle

```
_Bool flag
```

where *_Bool* is an unsigned integer type that can only be assigned 0 or 1. Giving value greater than 1 usually causes it to be assigned 1. The header file $< stdbool.h >$ can also be used which makes it easier to work with boolean values. It provides a macro, bool, that stands for *_Bool*.

3.7 Switch Statement

Similar to a case statement in VHDL/Ada. Its syntax is:

```
switch (expression) {
  case constant-expression : statements
  ...
  case constant-exprsion : statements
  default : statements
}
```

For example:

```

  int grade = 2;

switch (grade) {
    case 1:
        printf("Grade: A\n");
        break;
    case 2:
        printf("Grade: B\n");
        break;
    case 3:
        printf("Grade: C\n");
        break;
    default:
        printf("Grade: F (fail)\n");
        break;
}
```

Here, the value of the variable grade is tested against different values. As int grade = 2, when case 2 is executed, the statement inside this case will execute. Each case ends with a break; to prevent "fall-through". Without the break;, the next case is evaluated even if the condition doesn´t match.

4 Chapter 6 Notes: Loops

Chapter 5 covered C´s selection statements, if and switch. This chapter introduces C´s iteration statements, which allows one to set up loops.
A loop is a statement whose job is to repeatedly execute some other statement termed the loop body. In C every loop has a controlling statement, and each time the loop body is executed (termed an iteration of the loop), the controlling statement is evaluated. If the expression is true, that is, has a value that is not 0, the loop continues to execute.
C provides 3 interation staements: while, do and for. The While statement is used for loops whose controlling expression is tested before the loop body is tested. The Do staement is used if the controlling expression is tested after the loop body is executed. The For statemnt is used if one wants to increment or decrement a counting variable.
C also has break, continue and goto staements, which are typically used in conjuction with loops.

4.1 While Statement

This is the most simple and most fundamental statement in C. It has the form *while (controlling expression) statement.* For example:

```

    while(i<n) /* controlling expression*/
     i=i*2    /* loop body */
```

Listing 12: While Statement

When a While statement is executed, its controlling expression is evaluated first. If its value is non-zero (true), the loop body is evaluated, and the controlling expression tested again. To have more than one statement inside the loop body, one can use curly braces:

```

    while(i>0) {
        printf("T minus %d and counting\n", i);
        i--;
    }
```

Listing 13: While Statement Multiple Statements

A while loop can be used to create an infinite loop via while(1), which executes forever unless there is a statement that transfers control out of the loop (e.g. break, goto, return) or calls a function that causes the program to terminate.

4.2 Do Statement

Do statement is essentially a While statement whose controlling expression is tested after each execution of the loop body.

```

    do (statement) while (expression) ;
```

Listing 14: Do Statement

for example:

```

    i = 10;

    do {

        printf("T minus %d and counting\n", i);
        i--
    }   while (i > 0)
```

Listing 15: Do Statement Example

This has the same output as the while statement given in the While statment section i.e. the Do statement and While statement are indistinguishable. The main difference is that the body of a Do statement is always executed at least once, whereas the While statement body may be skipped entirely depending on the controlling expression.

4.3 For Loop

The for loop has the form for (expr1 ; expr2 ; expr3) statement. For example

```

for(i=10 ; i>0 ; i--)
    printf("T minus %d and counting\n", i);
```

Listing 16: For Loop

When this is executed, i is initialised to 10, the loop executes whilst i is greater than 0 and i is decremented by 1 each time the loop executes. The body is executed 10 times in all, with i varying from 10 down to 1. Thus, expr1 is an intialisation step that is performed only once, expr2 controls loop termination (the loop continues to execute as long as the value of expr2 is non-zero i.e. true) and expr3 is an operation to be performed at the end of each loop.