C Programming Language: Chapter Notes

Daniel Duggans

February 28, 2026

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:

```
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:

```
  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:

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

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:

$$\%[\textit{width}][\textit{length}]\textit{specifier}$$

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 3 Notes: Formatted Input and Output

C provides two powerful formatted I/O functions in `<stdio.h>`:

- `printf()` – formatted output
- `scanf()` – formatted input

Both functions use **format strings** containing ordinary characters and **conversion specifications** beginning with %.

3.1 3.1 The printf() Function

The general form is:

```
printf("format string", expression1, expression2, ...);
```

The format string may contain:

- Ordinary characters (printed exactly as written)
- Conversion specifications (placeholders beginning with %)

Basic Example

```
#include <stdio.h>

int main(void) {
    int i = 10, j = 20;
    float x = 43.2892f, y = 5527.0f;

    printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
    return 0;
}
```

Output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

Important Warning

The compiler does **not** check that:

- The number of conversion specifiers matches the number of variables.
- The specifier matches the variable type.

Examples of incorrect usage:

```
printf("%d %d\n", i);        // too few arguments
printf("%d\n", i, j);       // too many arguments
printf("%f %d\n", i, x);    // wrong types
```

These lead to undefined or meaningless output.

3.1.1 Conversion Specification Structure

General form:

$$\%[flags][width][.precision]specifier$$

Common specifiers:

Specifier	Meaning
%d	Integer (decimal)
%f	Floating-point (fixed decimal)
%e	Floating-point (scientific notation)
%g	Automatically chooses %f or %e
%c	Character
%s	String

Field Width and Precision

- **Width (m)**: Minimum number of characters to print.
- **Precision (p)**:
  - For %d: Minimum number of digits.
  - For %f, %e: Number of digits after decimal.
  - For %g: Maximum significant digits.
- Default alignment is right-justified.
- Use `-` for left-justification.

Examples:

```
printf("|%d|\n", 40);
printf("|%5d|\n", 40);
printf("|%-5d|\n", 40);
printf("|%5.3d|\n", 40);
printf("|%10.3f|\n", 839.21);
printf("|%-10g|\n", 839.21);
```

Key behaviours:

- `%5d` → minimum width 5
- `%-5d` → left aligned
- `%5.3d` → width 5, at least 3 digits
- `%10.3f` → width 10, 3 decimal places
- `%g` removes trailing zeros

3.1.2 Escape Sequences

Escape sequences begin with \.

Sequence	Meaning
`\n`	New line
`\t`	Horizontal tab
`\a`	Alert (bell)
`\b`	Backspace
`\"`	Double quote
`\\`	Backslash

Example:

```
printf("\"Hello!\"\n");
printf("\\\n");
```

3.2 3.2 The scanf() Function

The general form is:

```c
scanf("format string", &variable1, &variable2, ...);
```

Important: Variables must usually be preceded by & (address-of operator).

3.2.1 Basic Example

```c
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

If the user enters:

```
1 -20 0.3 -4.0e3
```

Values stored:

- i = 1
- j = -20
- x = 0.3
- y = -4000.0

3.2.2 How scanf() Works

- Processes format string from left to right.
- Skips leading whitespace for most specifiers.
- Stops reading when an invalid character is encountered.
- If a match fails, scanning stops immediately.

Recognition rules:
Integer (%d):

- Optional + or -
- One or more digits

Floating-point (%f, %e, %g):

- Optional sign
- Digits (optional decimal point)
- Optional exponent (e or E with optional sign)

3.2.3 Ordinary Characters in Format Strings

In `scanf`:

- Whitespace in format string matches any amount of whitespace in input.
- Non-whitespace characters must match exactly.

Example:

```c
scanf("%d/%d", &num, &denom);
```

Input:

```
5/6
```

Works correctly because / must match exactly.

3.2.4 Common Pitfalls

- Forgetting &:

```c
scanf("%d", n);    // WRONG
```

- Mixing printf-style formatting with scanf incorrectly:

```c
scanf("%d, %d", &i, &j);
```

  Fails if the comma is not present in the input.

- Adding `\n` at end of format string may cause program to hang.

3.3 Example: Adding Fractions

```c
/* Adds two fractions */
#include <stdio.h>

int main(void) {
    int num1, denom1, num2, denom2;
    int result_num, result_denom;

    printf("Enter first fraction: ");
    scanf("%d/%d", &num1, &denom1);

    printf("Enter second fraction: ");
    scanf("%d/%d", &num2, &denom2);

    result_num = num1 * denom2 + num2 * denom1;
    result_denom = denom1 * denom2;

    printf("The sum is %d/%d\n", result_num, result_denom);

    return 0;
}
```

Sample session:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

3.4 3.3 Summary

- `printf()` formats output using conversion specifications.
- `scanf()` reads formatted input and requires & for variables.
- Width and precision affect spacing and rounding in `printf()`.
- In `scanf()`, width limits characters read (not decimal places).
- The compiler does not check format string correctness.
- Always match specifiers with correct variable types.

4 Chapter 4 Notes: Expressions and Operators

One of C's distinguishing characteristics is its emphasis on **expressions** rather than statements. An expression is a formula that computes a value. Statements often contain expressions, but in C, expressions themselves play a central role.

4.1 4.1 Expressions

The simplest expressions are:

- **Variables** – represent values that may change during program execution.
- **Constants** – fixed values that do not change.

More complex expressions are formed by applying **operators** to **operands** (which are themselves expressions).

Example:

```
a + (b * c)
```

Here:

- `+` is applied to operands `a` and `(b * c)`.
- `b * c` is itself an expression.

Expressions can therefore be nested inside other expressions.

4.2 4.2 Arithmetic Operators

C provides standard arithmetic operators:

Operator	Meaning
$+$	Addition
-	Subtraction
	Multiplication
/	Division
%	Remainder (integers only)

Integer Division

If both operands are integers, division discards the fractional part:

```
7 / 2     // result: 3
```

Remainder Operator (%)

```
7 % 2     // result: 1
```

Only valid for integer operands.

4.3 4.3 Operator Precedence and Associativity

When expressions contain multiple operators, C follows rules of:

- **Precedence** – which operator is evaluated first.
- **Associativity** – direction of evaluation when precedence is equal.

Example:

```
a + b * c
```

Multiplication has higher precedence than addition, so this is interpreted as:

```
a + (b * c)
```

Most arithmetic operators are **left associative**:

```
a - b - c
```

Interpreted as:

```
(a - b) - c
```

Parentheses should be used to improve clarity.

4.4 4.4 Assignment Operators

The assignment operator:

```
=
```

Assigns the value of the right-hand expression to the left-hand variable.

```
i = 10;
```

Assignment is itself an expression and has a value (the value assigned).

```
i = j = k = 0;
```

Assignments associate from right to left.

Compound Assignment Operators

C provides shorthand forms:

Operator	Equivalent To
+=	x = x + y
-=	x = x - y
=	x = x * y
/=	x = x / y
%=	x = x % y

Example:

```
i += 5;    // same as i = i + 5;
```

4.5 4.5 Increment and Decrement Operators

C provides:

Operator	Meaning
++	Increment by 1
–	Decrement by 1

These can appear in two forms:

Post-increment / Post-decrement

```
i++;
i--;
```

The value is used first, then incremented/decremented.

```
int i = 5;
int a = i++;    // a = 5, i becomes 6
```

Pre-increment / Pre-decrement

```
++i;
--i;
```

The value is incremented/decremented first.

```
int i = 5;
int a = ++i;    // i becomes 6, a = 6
```

This distinction is important inside expressions.

4.6 4.6 Order of Evaluation

C does not always specify the order in which parts of an expression are evaluated.
Example:

```c
i = i++ + 1;    // undefined behaviour
```

Modifying a variable more than once between sequence points leads to undefined behaviour.
To avoid ambiguity:

- Keep expressions simple.
- Avoid modifying the same variable multiple times in one expression.

4.7 4.7 Expression Statements

An unusual feature of C is that **any expression can serve as a statement**.
Example:

```c
i = 10;
i++;
a + b;      // valid but pointless
```

An expression becomes a statement when followed by a semicolon.
Expression statements are commonly used for:

- Assignments
- Function calls
- Increment/decrement operations

4.8 4.8 Summary

- C emphasizes expressions as the core of computation.
- Operators combine operands to form more complex expressions.
- Arithmetic operators follow precedence and associativity rules.
- Assignment is itself an expression and associates right-to-left.
- Compound assignment operators provide shorthand notation.
- Pre- and post-increment operators behave differently inside expressions.
- Avoid undefined behaviour caused by modifying a variable multiple times in one expression.
- Any expression can function as a statement in C.

5 Chapter 5 Notes: Selection Statements

5.1 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`.

5.2 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)`.

5.3 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.

5.4 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

5.4.1 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.

5.5 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)
```

5.6 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*.

5.7 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.

6 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.

6.1 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.

6.2 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.

6.3 For Loop

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

```c

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. For loops can omit expr´s however the semi-colon must always be present e.g.

```c

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

Listing 17: For Loop

is valid, as i has been initialised previously. One may want to have two or more intialisation expressions or one that increments several expressions each time through the loop. This can be achieved via a comma expression as the first or third expression in the for loop. Forexample

```c

i=10;
for(i=1, j=2, k=i+j; k>0 ; k--)
    printf("T minus %d and counting\n", i);
```

Listing 18: For Loop

where the comma expression is left associative so i is ¨read¨ first.

6.4 Break Statement

The break statement allows one to exit a loop. Further, it can only exit one level of nesting, so a break in an inner nested loop will break out to the next level loop.

6.5 Continue Statement

Continue statements do not exit a loop or statement, transferring control to a point just before the end of a loop. This is in contrast to the break statement which transfers control to a point past the end of a loop. Thus, with a continue statement, control remains inside the loop. Also, unlike break statments which can be used in switch statements and loops (while, do, for), the continue statement can only be used in the latter. An example usage of the continue statement is:

```c

n = 0;
sum = 0;
while (n < 10){
    scanf(%d,&i);
    if (i==0)
        continue;
    sum += i;
    n++;
    /* continue jumps to here */
}
```

Listing 19: Continue Statement

Without continue, the above would have to be written such as:

```c

n = 0;
sum = 0;
while (n < 10){
    scanf(%d,&i);
    if (i!=0){
        continue;
    sum += i;
    n++;
    }
}
```

Listing 20: Continue Statement 2

i.e. the if statement would have to contain the body of the loop, so that the statements for updating the sum and incrementing n are executed only when the condition is satisfied.

6.6 goto Statement

Both the break and and continue statements are jump statements that transfer control from one point in the program to another. Both are restricted; break retruns one to a point just after the end of the closing loop, whereas continue to a point just before the end of the loop. In contrast, the goto statement is capable of jumping to any statement in a function, provided that the statement has a label.
A label is an identifier placed at the beginning of a statement - identifier : statement. A statement may have more than one label. The goto statement itself has the form - goto identifier ; . As an example:

```c

    for(d = 2; d < n; d++)
        if (n % d == 0)
            goto done;

            done:
            if (d < n)
                printf("%d is divisible by %d\n", n, d);
            else
                printf("%d is prime\n", n, d);
```

Listing 21: goto statement example

7 Basic Types

7.1 Numeric Types

7.1.1 Signed, unsigned, long, short and long long int

C supports two fundamentally different numeric types: integer types and floating types. Integer types are whole numbers whereas values of a floating type can have fractional parts as well. Integer types in turn are divided into two categories; signed and unsigned. With signed, the MSB is in fact a sign bit, 0 being positive and 1 being negative. By default, integer variables in C are signed by default, and one must use the reserved word unsigned to tell the compiler that there is no sign bit.

An integer is typically 32 bits (though this is machine dependent, bieng 16 bits on old CPUs). C also provides long integers, which have more than 32 bits. Short integers are also available if one wants to store an integer in less space than normal. The different types can be combined, however, only only the following six combinations actually produce different types:

- short int
- unsigned short int
- int
- unsigned int
- long int
- unsigned long int

Other combinations are synonyms for one of these six types. (For example, long signed int is the same as long int, since integers are always signed unless otherwise specified.) Incidentally, the order of the specifiers doesn't matter; unsigned short int is the same as short unsigned int. C allows us to abbreviate the names of integer types by dropping the word int. For example, unsigned short int may be abbreviated to unsigned short, and long int may be abbreviated to just 1ong. Typical value ranges for the vrious it types are given in figure 1.

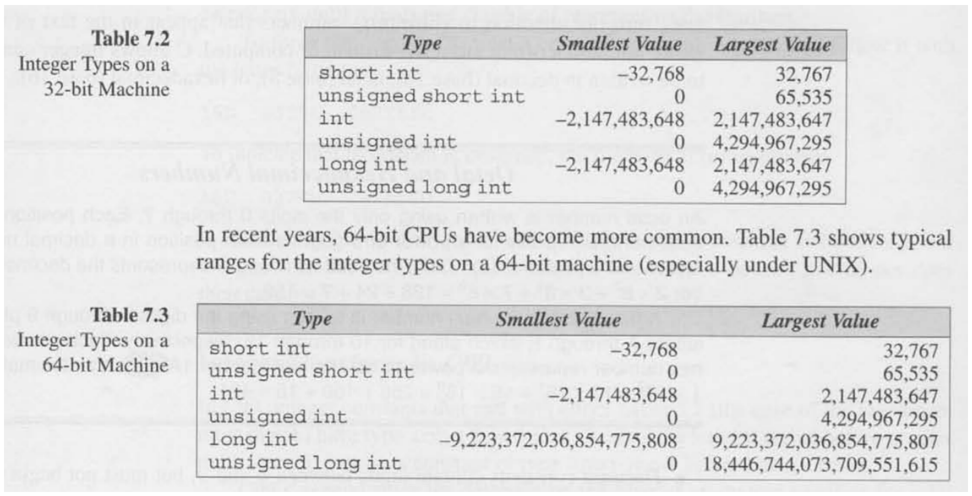

Table 7.2
Integer Types on a 32-bit Machine

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

In recent years, 64-bit CPUs have become more common. Table 7.3 shows typical ranges for the integer types on a 64-bit machine (especially under UNIX).

Table 7.3
Integer Types on a 64-bit Machine

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	–9,223,372,036,854,775,808	9,223,372,036,854,775,807
`unsigned long int`	0	18,446,744,073,709,551,615

Figure 1: Typical integer ranges for 16 and 32 bit machines. Taken from *C: A Modern Approach.*

The above ranges vary between compilers. To check the ranges for the various numeric types, one can check the limits.h header file. C99 added long long int and unsigned long long int. Both long long types are required to be at least 64 bits wide.

7.1.2 Constants

Constants can be written in decimal (base 10), octal (base 8) or hexadecimal (base 16). Integers can start with any number except 0. Octal uses any digit between 0 and 7 and must begin with 0. Hexadecimal can contain any digit between 0 and 9, and the letters A to F, and begin with 0x. The different forms can be mixed together when, e.g., adding or subtracting numbers. To force the compiler to treat a constant as a long integer, just follow it with the letter L (or 1): 15L 0377L 0x7fffL To indicate that a constant is unsigned, put the letter U (or u) after it: 15U 0377U 0x7fffU L and U may be used in combination to show that a constant is both long and unsigned: OxffffffffUL. (The order of the L and U doesn't matter, nor does their case.) Integer Constants in C99 In C99, integer constants that end with either LL or 11 (the case of the two letters must match) have type long long int. Adding the letter U (or u) before or after the LL or 11 denotes a constant of type unsigned long long int.

7.2 Floating Types

Integer types are not suitable for all applications. In many cases, we need to represent numbers with fractional parts or values that are extremely large or small. Such numbers are stored using floating-point representation, so called because the position of the decimal point can "float." C provides three floating-point types:

- float
- double
- long double

These correspond to single-precision, double-precision, and extended-precision floating-point formats, respectively. The `float` type is appropriate when limited precision is acceptable, while `double` offers greater precision and is sufficient for most programs. The `long double` type provides the highest precision, but is used less frequently.

The C standard does not specify the exact precision or range of floating types, as these are machine dependent. Most modern systems follow the IEEE Standard 754 for floating-point arithmetic. Under this standard, floating-point numbers are stored in a form of scientific notation consisting of three components: a sign, an exponent, and a fraction (mantissa). The size of the exponent determines the range of representable values, while the size of the fraction determines precision.

In IEEE 754 single-precision format (32 bits), the exponent is 8 bits and the fraction is 23 bits, giving a maximum value of approximately 3.4×10^{38} and about 6 decimal digits of precision. Double-precision format (64 bits) provides roughly 15 decimal digits of precision and a much larger range. The size of `long double` varies by implementation, commonly being 80 or 128 bits.

The characteristics of floating types on a given system can be obtained from the `<float.h>` header file. On systems that do not follow IEEE 754, the ranges and precision may differ.

7.2.1 Floating Constants

Floating constants may be written using a decimal point, an exponent, or both. For example, the following are all valid floating constants:

```
57.0   57.   57.0e0   57E0   5.7e1   570.e-1
```

By default, floating constants are treated as type `double`. To force a constant to be stored as a `float`, append the letter `F` or `f` (e.g., `57.0F`). To store a constant as a `long double`, append `L` or `l` (e.g., `57.0L`). C99 also allows hexadecimal floating constants, although they are rarely used.

7.2.2 Reading and Writing Floating-Point Values

When reading floating-point values using `scanf`, different conversion specifiers are required depending on the type. For values of type `double`, the letter `l` must precede the conversion specifier:

```
double d;
scanf("%lf", &d);
```

For values of type `long double`, the letter `L` must be used:

```
long double ld;
scanf("%Lf", &ld);
printf("%Lf", ld);
```

In calls to `printf`, the `e`, `f`, and `g` conversion specifiers may be used for both `float` and `double` values.

7.3 Character Types

The only remaining basic type is char, the character type. The values of char differ from one machine to the next as different machines may have different underlying character sets. The most popular is the ASCII set. C treats characters as small integers.

8 Arrays

8.1 Overview

- Variables encountered so far have been **scalar variables**, capable of holding a single value.
- C also supports **aggregate variables**, which store collections of values.
- The two aggregate types in C are:
  - Arrays
  - Structures
- This chapter covers **arrays**.

8.2 One-Dimensional Arrays

- An **array** is a data structure containing multiple values of the same type.
- Individual values are called **elements**.
- Elements are accessed using an **index (subscript)**.

8.2.1 Array Declaration

To declare an array, specify the element type and number of elements:

```c
int a[10];
```

Listing 22: One-dimensional array declaration

- The array contains 10 integers.
- Arrays may be of any type.
- Elements are indexed starting at 0.
- Valid indices for an array of size n are:

$$0 to n - 1$$

Example:

```c
a[0] = 1;    /* assigns 1 to first element */
```

8.2.2 Common Array Operations

Clearing an array:

```c
for (int i = 0; i < 10; i++)
    a[i] = 0;
```

Listing 23: Clearing an array

Reading data into an array:

```c
for (int i = 0; i < 10; i++)
    scanf("%d", &a[i]);
```

Listing 24: Reading data into an array

Summing elements:

```c
int sum = 0;

for (int i = 0; i < N; i++)
    sum += a[i];
```

Listing 25: Summing array elements

8.2.3 Array Bounds

- C does **not check array bounds**.
- Accessing outside the array produces **undefined behaviour**.
- Common mistake:
  - Using indices 1 to n instead of 0 to $n - 1$.

8.2.4 Array Initialization

Arrays may be initialized when declared:

```c
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

Listing 26: Array initializer

- Initializer values must be constant expressions.
- If fewer values are given, the remaining elements become 0.

Example:

```c
int a[10] = {0};    /* all elements initialized to 0 */
```

8.2.5 Designated Initializers

Specific elements may be initialized using designators:

```c
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

Listing 27: Designated initializers

- Values not specified become 0.
- Designators must be integer constant expressions.
- Valid designators range from 0 to $n - 1$.
- If the array length is omitted, the compiler determines the size from the largest designator.

8.2.6 The `sizeof` Operator

- `sizeof` determines the size of an array in bytes.

Example:

```c
sizeof(a)
```

- If `a` contains 10 integers and each integer is 4 bytes:

$$sizeof(a) = 40$$

- Element size:

  ```c
  sizeof(a[0])
  ```

- Number of elements:

  ```c
  sizeof(a) / sizeof(a[0])
  ```

8.3 Multidimensional Arrays

- C supports multidimensional arrays.
- Two-dimensional arrays are often called **matrices**.

Example:

```c
int a[5][9];
```

Listing 28: Two-dimensional array

- 5 rows
- 9 columns

8.3.1 Storage Order

- Arrays are stored in **row-major order**.
- Row 0 is stored first, then row 1, etc.

8.3.2 Initializing a Matrix

Example: identity matrix

```c
#define N 10

double ident[N][N];

for (int row = 0; row < N; row++)
    for (int col = 0; col < N; col++)
        if (row == col)
            ident[row][col] = 1.0;
        else
            ident[row][col] = 0.0;
```

Listing 29: Identity matrix

- Multidimensional arrays are less common in C than in some languages.
- Arrays of pointers often provide more flexibility.

8.3.3 Multidimensional Initializers

Example:

```c
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

Listing 30: Multidimensional initializer

8.4 Variable-Length Arrays

- Normally array size must be a constant expression.
- In C99, arrays may use a non-constant size.
- These are called **variable-length arrays (VLAs)**.

Example:

```c
#include <stdio.h>

int main(void)
{
    int n;

    printf("How many numbers do you want to reverse? ");
    scanf("%d", &n);

    int a[n];   /* size determined at runtime */

    printf("Enter %d numbers: ", n);

    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("In reverse order:");

    for (int i = n-1; i >= 0; i--)
        printf(" %d", a[i]);

    printf("\n");

    return 0;
}
```

Listing 31: Variable-length array example

8.5 Important Rules and Pitfalls

- Array indices start at 0.
- C does not perform bounds checking.
- Accessing outside an array causes undefined behaviour.
- Arrays store elements contiguously in memory.
- Use `sizeof(a)/sizeof(a[0])` to determine array length.

9 Functions

9.1 Introduction

- A **function** is a named group of statements.
- Functions are the basic building blocks of C programs.
- A function:
  - May have parameters
  - May return a value
  - May also do neither
- Each function has:
  - Its own variables
  - Its own statements
- Benefits of functions:
  - Modularity
  - Easier debugging
  - Code reuse
  - Improved readability

9.2 Defining and Calling Functions (9.1)

9.2.1 Function Definitions

General form:

```
return-type function-name(parameter declarations)
{
    declarations
    statements
}
```

Example:

```
double average(double a, double b)
{
    return (a + b) / 2;
}
```

- The **return type** specifies the type of value returned.
- If no value is returned, the return type is `void`.
- Parameter declarations specify type and name.

9.2.2 Function Calls

Example:

```
printf("Average: %g\n", average(x, y));
```

- A function call consists of:
  - Function name
  - Parentheses
  - Arguments (optional)
- The returned value can:
  - Be assigned
  - Be printed
  - Be used in expressions

9.2.3 Execution of Function Calls

- Control transfers to the called function.
- Arguments are evaluated before the call.
- Parameters receive copies of argument values.
- Execution resumes after the call completes.

9.2.4 Local Variables

- Variables declared inside a function are **local**.
- Local variables:
  - Exist only during function execution
  - Are not accessible outside the function

9.2.5 Scope

- Scope = region where an identifier is visible.
- Parameters and local variables are visible only inside the function.
- Different functions may use the same variable names.

9.2.6 The `main` Function

- Every program must contain `main`.
- Execution begins in `main`.
- `main` normally returns an integer.

9.3 Function Declarations (9.2)

9.3.1 Function Prototypes

Example:

```
double average(double a, double b);
```

- A function declaration specifies:
  - Function name
  - Return type
  - Parameter types
- A declaration ends with a semicolon.
- No function body is included.

9.3.2 Purpose of Prototypes

- Allow functions to be called before definition.
- Enable compiler type checking.

9.3.3 Parameter Names in Prototypes

```
double average(double, double);
```

- Parameter names are optional.
- Types are required.

9.3.4 Compiler Checking

The compiler verifies:

- Correct number of arguments
- Correct argument types
- Correct return type usage

9.4 Arguments (9.3)

9.4.1 Call by Value

- C passes arguments by value.
- Parameters receive copies.

  Example:

```
void f(int x)
{
    x = 10;
}
```

- Changing `x` does not affect the caller.

9.4.2 Parameter Matching

- Arguments are matched by position.
- Number and types must agree with prototype.

9.4.3 Type Conversions

- Automatic conversions occur if needed.

Example:

```
double f(double x);

f(3);
```

- Integer 3 is converted to double.

9.4.4 Array Arguments

```
int sum_array(int a[], int n)
{
    int i, sum = 0;

    for (i = 0; i < n; i++)
        sum += a[i];

    return sum;
}
```

- Arrays are passed by reference (address is passed).
- Changes inside the function affect the original array.

9.5 The `return` Statement (9.4)

9.5.1 Returning a Value

```
return expression;
```

- Expression type must match return type.

Example:

```
return (a + b) / 2;
```

9.5.2 Returning Nothing

```
return;
```

- Used in `void` functions.

9.5.3 Multiple Returns

- A function may contain multiple return statements.

Example:

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

9.6 Program Termination (9.5)

9.6.1 Returning from `main`

```
return 0;
```

- Indicates successful termination.

9.6.2 Reaching End of `main`

- Equivalent to:

  ```
  return 0;
  ```

9.6.3 The `exit` Function

```
exit(status);
```

- Immediately terminates the program.
- Declared in `<stdlib.h>`.

Common values:

```
exit(EXIT_SUCCESS);
exit(EXIT_FAILURE);
```

9.7 Recursion (9.6)

9.7.1 Definition

- A recursive function calls itself.

Example:

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

9.7.2 Essential Requirements

- Base case (stopping condition)
- Recursive case

9.7.3 How Recursion Works

- Each call creates a new set of local variables.
- Calls are stored on the program stack.
- Calls return in reverse order.

9.7.4 Advantages

- Natural for divide-and-conquer problems.
- Often simpler and clearer.

9.7.5 Disadvantages

- Uses more memory.
- Usually slower than iteration.
- Risk of infinite recursion.

9.8 Important Rules and Pitfalls

- Always declare functions before use.
- Ensure argument types match prototypes.
- Remember arguments are passed by value.
- Local variables are not preserved between calls.
- Recursive functions must terminate.