Chapter 1

Foundations

1.1 Prerequisites

Before exploring this chapter, it's important to understand the distinction between ***the foundations of programming*** *and* ***an introduction to the basics of programming****. This chapter focuses on the former.*

If the material feels overwhelming, consider skipping text or using a language model (LLM) to clarify key concepts. This text is not for beginners but aims to help build a strong conceptual foundation in computers and programming.

1.2 Foundations of Programming

We begin with the fundamental concepts that form the foundation of elementary programming: *variables*, *control structures*, and *functions*. A variable is a named storage location in memory that holds data. The value stored in a variable can change during program execution, a "variable." Control structures are constructs that determine the flow of execution in a program. They allow a program to make decisions, repeat actions, and branch into different paths based on conditions. Functions are sequences of instructions stored in memory that the CPU can execute. Typically, when a function is called, the computer temporarily pauses the current sequence of instructions, switches to the function's instructions, and then returns to the original sequence once the function has completed. Functions in computers can perform tasks ranging from basic arithmetic to complex algorithms, and they can invoke other functions, enabling intricate chains of computation.

1.3 Simple data types

You've probably heard that digital computers use binary numbers, which are composed of ones and zeros, to represent all types of data and instructions. This binary system, based on just two states (on/off, true/false, or 1/0), is fundamental to how computers operate.

Let's explore how different types of data, like integers, floating-point numbers, characters, and strings, are represented in this binary system, particularly within the context of an 8-bit word length, which is a common unit of data in computing.

When we talk about an 8-bit word length in computing, we're referring to the amount of data that a computer's processor can handle or process in a single operation. In this context, a "word" is a fixed-sized group of bits that are processed together as a unit by the CPU. The length of the word determines how much data the CPU can process at one time. The 8-bit word length wasn't initially established as a standard but rather evolved over time due to practical considerations and technological advancements. Data with the 8-bit length is called a "byte." Today an ordinary computer you would use such as a laptop, probably has a 64-bit processor, thus operating with 64 bit words.

1.3.1 Integers in binary

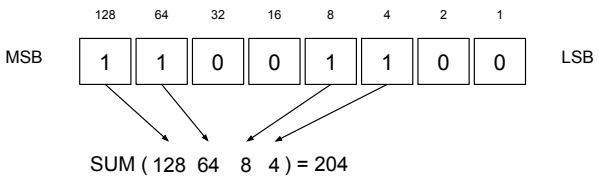
In an 8-bit system, each integer is represented by a sequence of eight binary digits (bits). Each bit can be either 0 or 1, and the combination of these bits determines the integer value. The leftmost bit in the sequence is the most significant bit, and the rightmost bit is the least significant bit. Here's how it works:

0 in binary is 00000000.

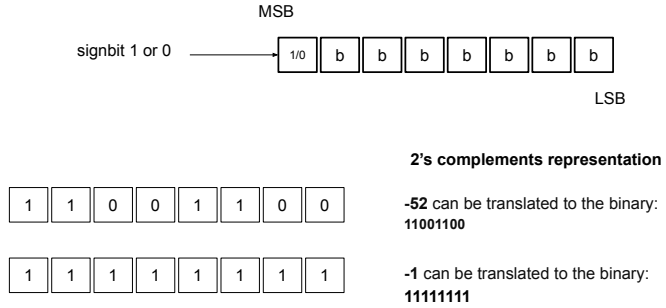
1 in binary is 00000001.

2 in binary is 00000010.

45 in binary is 00101101.



In this system, you can represent any integer from 0 to 255 (for unsigned integers) using 8 bits. If signed integers are used, one bit is reserved for indicating the sign (positive or negative), allowing representation of values from -128 to 127.



2's complement representation. When using positive numbers the highest bit, most significant bit (MSB), is 0. The remaining bits represent the number in binary form, just like unsigned integers. When using negative numbers MSB is 1. The remaining bits are used to represent the magnitude of the negative number using a specific method.

1's complement representation. In the 1's complement system, integers are represented in a way that the most significant bit (MSB) acts as the sign bit. A 0 in the MSB indicates a positive number, while a 1 indicates a negative number. Positive numbers are represented the same way as unsigned binary numbers, but negative numbers are represented by flipping all the bits of the corresponding positive number.

For example, in an 8-bit system:

0 in binary is 00000000.

1 in binary is 00000001.

−1 in binary is 11111110.

−2 in binary is 11111101.

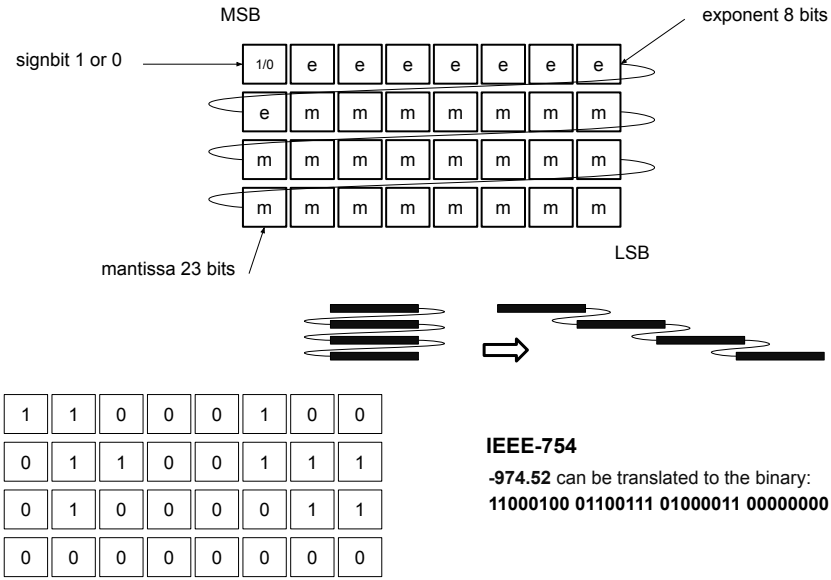
−45 in binary is 11010010.

Using 1's complement representation, the range of integers in an 8-bit system is from −127 (11111110) to +127 (01111111). However, a notable feature of 1's complement is that the value 0 has two representations: 00000000 (positive zero) and 11111111 (negative zero).

1.3.2 Floating-point numbers

Floating-point numbers, which represent real numbers (like 3.14 or -2.7), are more complex. They are stored using a standard format known as IEEE 754 in most systems, which divides the number into three parts: the sign, the exponent, and the mantissa.

This structure allows computers to represent a wide range of real numbers, including very large or very small values, by encoding them in scientific notation -9.7452×10^2 into binary 32-bits approximation.



1.3.3 Characters and ASCII

Characters in computers, such as letters, numbers, and punctuation marks, have historically been represented using standardised encoding systems. One of the earliest and most influential of these systems is ASCII (American Standard Code for Information Interchange).

For example:

The letter `A` is represented as `01000001`.

The letter `a` is represented as `01100001`.

The character `0` (zero) is represented as `00110000`.

The space character is `00100000`.

The English alphabet, with its Latin characters, has the advantage of being relatively limited in scope, which simplifies its representation in digital form. However, representing other languages has proven more challenging, leading to various modifications, replacements, and adaptations.

UTF-8 (8-bit Unicode Transformation Format) was introduced in the early 1990s as part of the Unicode standard, which aimed to solve the limitations of ASCII by providing a single, unified character encoding system that could represent virtually every character used in human languages.

1.3.4 Strings

A string is simply a sequence of characters, so it's stored as a sequence of 8-bit binary numbers, one for each character. For example, the string "Hello" would be stored in memory as:

`H`: `01001000`

`e`: `01100101`

`l`: `01101100`

`l`: `01101100`

`o`: `01101111`

These binary sequences are stored consecutively in memory, and a special character, often called a null character (with a binary value of `00000000`), is used to indicate the end of the string.[1]

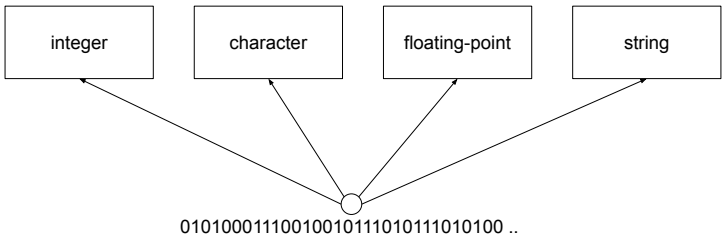
1.3.5 Representations and types

In essence, everything in a computer—whether it's numbers, text, or instructions—is represented by sequences of binary digits. Understanding these basic concepts of binary representation helps in grasping how computers store and manipulate different types of data, whether it's an integer being processed in a calculation, a floating-point number being used in scientific computation, or a string of text being displayed on a screen.

To distinguish between different binary representations, we use types. Types not only define the kind of data (such as integers, floating-point numbers, or

[1] There are actually other ways to terminate a string, such as having the length of the string occupy the first bytes of the representation.

characters) but also how that data is interpreted and manipulated. In some cases, types can be converted from one to another, depending on their compatibility. Additionally, types can encompass more than just raw binary data; they can also include the operations that can be performed on them. For instance, in object-oriented programming, types can represent "objects" that combine both data and the methods (or tools) that operate on that data.



1.3.6 Summary

Foundational aspects on computers and programming includes:

Simple data types. Digital computers use binary (ones and zeros) to represent all types of data. An 8-bit word length (a group of 8 binary digits) is commonly used to represent a unit of data, called a byte. Different types of data, like integers, floating-point numbers, characters, and strings, are represented using binary in distinct ways.

Integers. Integers are represented by a sequence of binary digits, and in an 8-bit system, each integer value corresponds to a combination of 0s and 1s. You can represent values from 0 to 255 (unsigned) or -128 to 127 (signed). Two's complement is a common way to represent signed integers.

Characters. Characters, such as letters and symbols, are encoded in binary using systems like ASCII. Each character corresponds to a unique binary value. ASCII was extended by Unicode (UTF-8) to represent more characters from various languages.

Floating-point numbers. Floating-point numbers represent real numbers, stored using a format like IEEE 754, which divides the number into sign, exponent, and mantissa. This format allows for a wide range of values by storing numbers in a scientific notation-like structure.

Strings. Strings are sequences of characters, and each character in a string is represented as a binary value. A special character, typically a null character (00000000), is used to indicate the end of the string.

Representation and types. Everything in a computer is represented by binary digits. Data types define how this binary data is interpreted and manipulated. In some cases, data types can be converted between formats if compatible. Types can also represent more complex structures, like objects in object-oriented programming.

1.4 Variables

In mathematics, a variable is a symbol that represents an unknown or changeable value. Variables are fundamental in expressing mathematical formulas, equations, and functions. For example, in the equation the formula $y = 2x + 3$, x and y are variables, where x can take on different values, and y is determined based on x. Variables allow mathematicians to generalise problems and work with abstract concepts rather than specific numbers.

In computer science though, a variable is a storage location in a computer's memory that is associated with a symbolic name (identifier) and can hold a value. This value can change during the execution of a program, making variables essential for dynamic behaviour in software. Variables can hold various types of data, such as numbers, strings, or more complex structures (like arrays and objects). For example, in a program, you might declare a variable age to store a user's age, and this value can be updated as needed.

As mentioned earlier, variables can store different kinds of values and are therefore classified into different types based on the nature of these values. For instance, a variable of type "8-bit positive integer" may hold a binary number, which could also correspond to an ASCII character. They hold the same space of binary digits. However, the type of the variable determines how we interpret its content—whether as an integer or as a character.

1.4.1 Assignment

Assignment in computer science refers to the operation of *assigning a value to a variable*. It's a fundamental concept in programming, as it allows developers to store data, manipulate it, and control the flow of a program.

Assignment is a basic yet essential operation in programming. It allows developers to store, update, and manipulate data within a program. Understanding assignment, especially in the context of different programming paradigms and data types, is crucial for writing effective and efficient code.

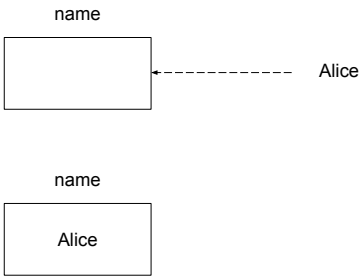
What is assignment? Assignment is the process of setting or updating the value of a variable. In most programming languages, this is done using the assignment operator, typically represented by an equals sign ('='). The general syntax for assignment is in Python:

```
variable_name = value
```

And an example:

```
x = 5
name = "Alice"
```

The variable to the left ("name") is assigned to the value on the right ("Alice").



Memory allocation. When you assign a value to a variable, the computer allocates memory to store that value. The variable name acts as a reference to the location in memory where the data is stored.

Overwriting. If a variable already holds a value and a new value is assigned to it, the old value is overwritten. The memory location is updated with the new value, and the previous data is typically discarded (unless the language has garbage collection or other memory management features).

Simple assignment. A variable assigned a direct value.

```
age = 30
```

Compound assignment. These are shorthand operations that combine an arithmetic operation with assignment. For example:

```
x += 5 # same as x = x + 5
y *= 2 # same as y = y * 2
```

Multiple assignment. Some languages, such as Python, allow assigning multiple variables in a single statement, assigned in sequence.

```
a, b, c = 1, 2, 3
```

Type consistency. In *statically-typed* languages, the type of the variable must be declared beforehand, and the value assigned must be of that type. In *dynamically-typed* languages, the type can be inferred at runtime based on the value assigned.

Static in C:

```
int number = 10; // 'number' must be an integer
```

Dynamic in Python:

```
var = 10 # 'var' can later hold a different type, like a string
var = "hello"
```

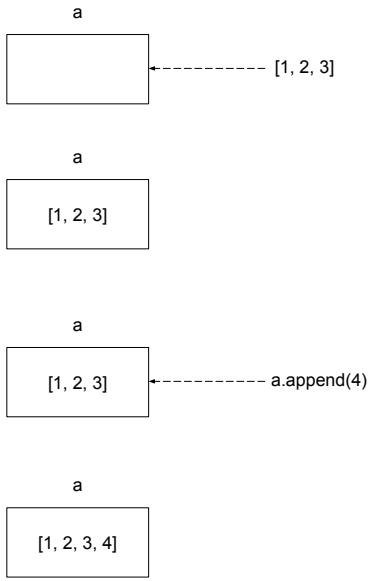
1.4.2 Mutable and immutable variables

Mutable and *immutable* variables are fundamental concepts in programming. Mutable variables can be more memory-efficient since they can be changed in place without allocating new memory. However, immutability can lead to safer, more predictable code, as it reduces the risk of accidental changes to data. Immutable variables are also inherently thread-safe, meaning they can be shared across multiple threads without the need for locks or other synchronization mechanisms, which makes them ideal for concurrent programming.

Mutable variables are those whose values can be changed *after* they have been initialised. When a mutable variable is modified, the program can update the data stored in the same memory location. This means you can alter, add, or remove data from the variable without creating a new instance of it.

In Python, lists and dictionaries are mutable. You can change elements, append new items, or remove items directly.

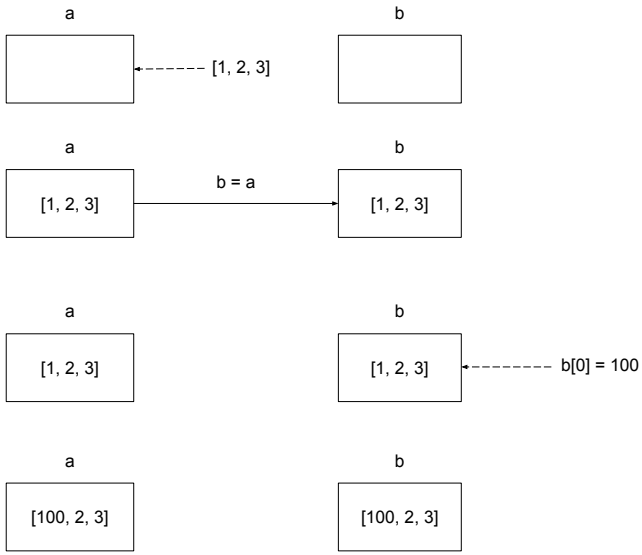
```
a = [1, 2, 3]
a.append(4) # a is [1, 2, 3, 4]
```



When *reference assigning* complex data types like objects or arrays, the variable often holds a reference to the original data. Modifying the new variable can affect the original data.

```
# reference assignment with lists (mutable)
a = [1, 2, 3] # assign a list to variable 'a'
b = a # 'b' points to the same list as 'a'
b[0] = 100 # changing 'b' affects 'a'

print(a) # output: [100, 2, 3] (changed)
print(b) # output: [100, 2, 3]
```



Immutable variables on the other hand, *cannot be changed* once they have been created. Any operation that seems to modify an immutable variable actually creates a new instance with the updated value. When you attempt to change an immutable variable, the program creates a new object with the new value, leaving the original object unchanged.

Examples in Python, strings and tuples are immutable. Any operation that modifies a string will result in a new string.

```
astring = "hello"
astring = astring + " world" # astring is a new string "hello world"
```

When *value assigning* primitive data types, a copy of the value is made. Changing the new variable does not affect the original.

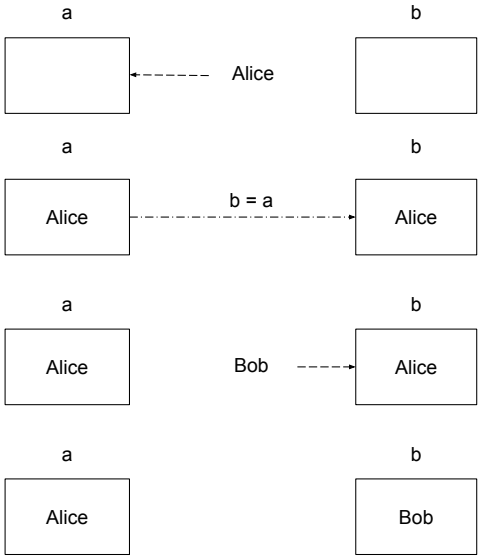
```
# value assignment with integers (immutable)
a = 10 # assign 10 to variable 'a'
b = a # 'b' gets a copy of the value stored in 'a'
b = 20 # changing 'b' from 10 to 20 does not affect 'a'

print(a) # output: 10 (unchanged)
print(b) # output: 20
```

Or another sample with strings.

```
# value assignment with strings (immutable)
a = "Alice" # assign 'Alice' to 'a'
b = a # 'b' gets a copy of the value stored in 'a'
b = "Bob" # changing 'b' from 'Alice' to 'Bob' does not affect 'a'

print(a) # output: Alice (unchanged)
print(b) # output: Bob
```



Assignment in functional programming. In functional programming paradigms, assignment is often discouraged or restricted. Instead of reassigning values, new variables or data structures are created. A variable can be reassigned new data, but the previous data remains unchanged—it is simply discarded. This, known as the above *immutability*, ensures that data remains constant, helping to prevent unintended side effects, and is one of the more beneficial features of functional languages.

1.4.3 Summary

Assignments. Assignment refers to the process of assigning a value to a variable using an assignment operator (like '=' in many programming languages). It

allows values to be stored, updated, and manipulated. Compound and multiple assignments are shorthand techniques used for efficiency. Assignment behaviour varies between statically and dynamically typed languages.

Memory allocation. When a value is assigned to a variable, memory is allocated to store it. Overwriting an existing variable replaces its memory content. In statically typed languages, types must be declared, while dynamically typed languages infer types at runtime.

Mutable versus immutable variables. Mutable variables can be changed after being created, while immutable variables cannot. Changing a mutable variable modifies the original data, but changing an immutable variable creates a new object. Mutability affects memory efficiency and thread safety in concurrent programs.

Mutable variables. Mutable variables can be updated in place, which means changing their value doesn't create a new memory object. Examples include lists and dictionaries in Python. Changing a reference to a mutable object can alter the original data.

Immutable variables. Immutable variables, such as strings and tuples, cannot be changed once created. Modifying an immutable variable creates a new instance rather than updating the existing one. This behavior ensures data safety and prevents unintended side effects.

Functional programming. In functional programming, assignment is often discouraged, and immutability is favoured. Instead of reassigning values, new variables are created. This helps prevent side effects and ensures data consistency throughout the program's lifecycle.

1.5 Control structures

Traditional programming but also applicable concepts has often been based on imperative languages, where statements dictate what should happen in a kind of command-like manner. These instructions typically involve operations like taking two given numbers, multiplying them, and assigning the result to a variable. In this type of programming, it is particularly easy to adopt three "primitives" concerning control structures: *sequence*, *iteration*, and *selection*. As shown in Figure 1.1.

1.5.1 Conventional control structures

In computer science, control structures are essential constructs that dictate the flow of execution in a program. They allow a program to make decisions, repeat actions, and branch into different paths based on conditions. Even if the three

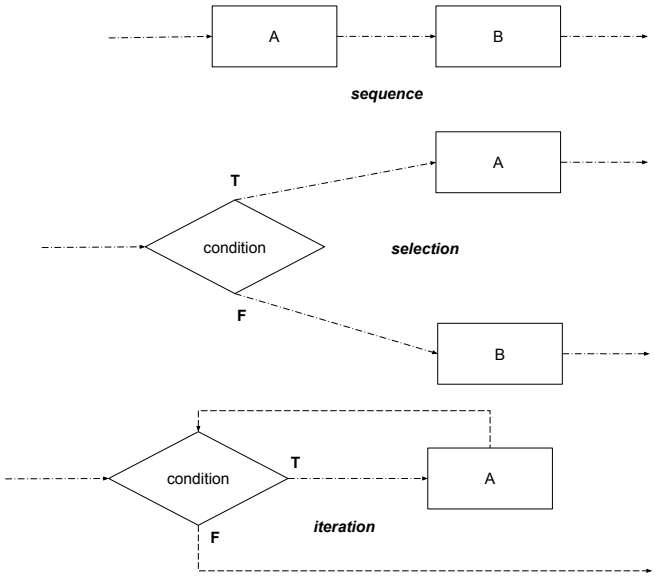


Figure 1.1: Sequence, Iteration, Selection.

primitives are exhaustive for many variations of control structures, there are some convenient structures that often are used in programming languages.

Sequential Execution is the most basic form of control, where instructions are executed one after the other in the order they appear. For example, in a simple program that calculates the sum of two numbers, the operations are performed sequentially.

Conditional Statements (`if-then-else`) allow a program to execute certain blocks of code only if specific conditions are met. For example, an if-else statement might check if a user input is valid and then proceed with different actions based on that input. Examples of these kinds of flows, can be seen in Figure 1.2.

Loops (`for`, `while`, `do-while`) enable the repetition of a block of code multiple times, either a fixed number of times (for loop) or until a condition is met (while loop). For example, a for loop might be used to iterate over an array and perform operations on each element. See Figure 1.3.

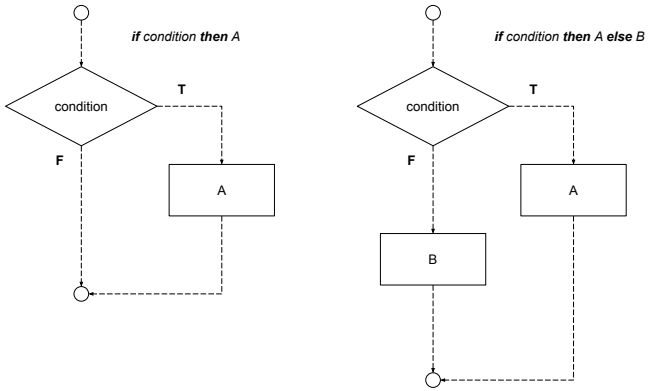


Figure 1.2: Selection.

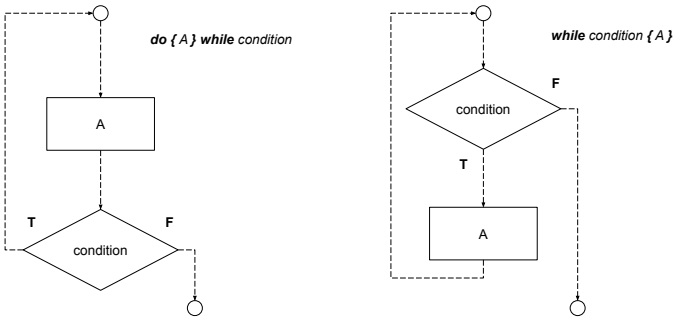


Figure 1.3: Iteration.

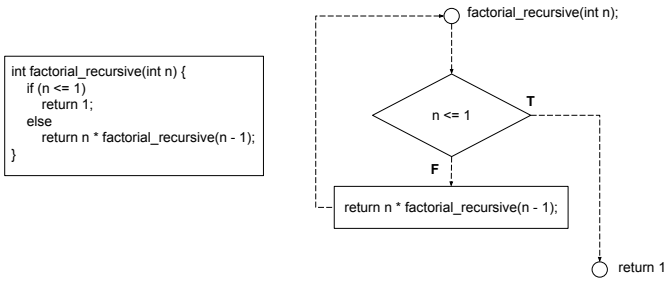


Figure 1.4: Recursive Factorial.

Recursion is a control structure where a function calls itself to solve smaller instances of the same problem. This is particularly useful for problems that can be divided into similar subproblems, like traversing a tree data structure or calculating the factorial of a number. An illustration of the latter can be seen in Figure 1.4 and Figure 1.5. Recursive calls build up, and then collapse back to give the final result. In this example: $5 \times 4 \times 3 \times 2 \times 1 = 120$.

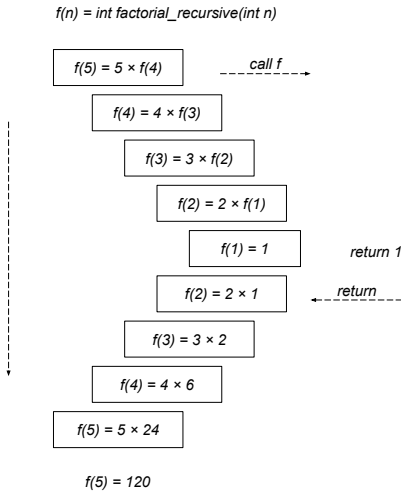


Figure 1.5: Illustration of Running a Recursive Factorial.

Switch-Case is a multi-way branch statement that allows a variable to be tested

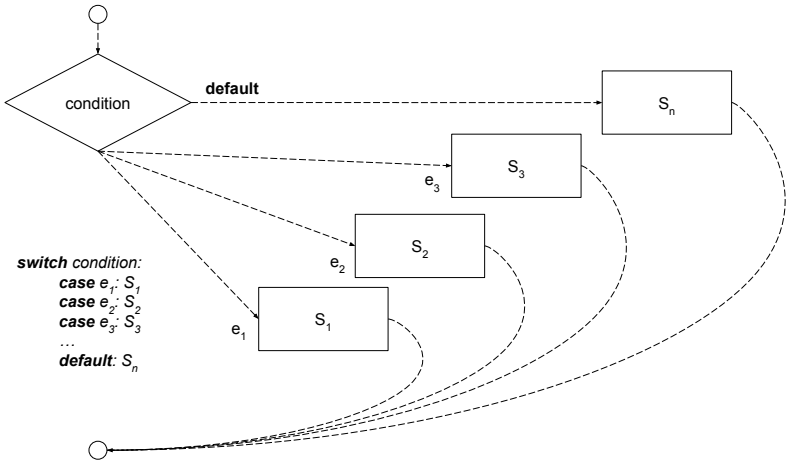


Figure 1.6: Switch-Case.

for equality against a list of values. Each value, known as a case, leads to a block of code that is executed if the case matches the variable. See Figure 1.6.

1.5.2 Control structures in computers

Here we are closing in on the main topic of virtual machines, which will be addressed in later parts. In computers, control structures are implemented at the hardware and software level to manage the flow of instructions executed by the CPU. The CPU uses a combination of instruction pointers, condition flags, and branching mechanisms to implement these control structures.

Instruction pointers. The CPU executes instructions stored in memory sequentially, using an instruction pointer to track the address of the next instruction. Sequential control is managed by simply incrementing the instruction pointer.

Conditional branching. Conditional control structures like if-else statements are implemented using conditional branching. The CPU evaluates a condition and, based on the result, may branch to a different memory address to continue execution. This is typically done using flags in the CPU's status register, which are set or cleared based on the results of comparisons.

Loops and recursion. Loops in software translate to repeated execution of a set of instructions until a condition is met. The CPU handles this by repeatedly evaluating the loop condition and branching back to the start of the loop if the condition is still true. Recursion is handled similarly, but it involves storing return addresses on the stack to ensure the CPU can return to the correct place after the recursive calls are completed.

Hardware implementation. At a lower level, control structures are embedded in the microarchitecture of processors. Conditional branches, loops, and other control mechanisms are managed by a combination of the CPU's control unit, arithmetic logic unit (ALU), and registers, ensuring efficient execution of complex instruction sequences.

1.5.3 Summary

Control structures. Control structures in programming dictate the flow of execution. The three fundamental structures are sequence, iteration, and selection. Control structures guide how a program makes decisions, repeats actions, and branches into different paths based on conditions.

Sequential execution. Sequential execution refers to the simplest form of control, where instructions are executed one after another in the order they appear in the code. This flow is typical for basic programs, such as calculating the sum of two numbers step-by-step.

Conditional statements. Conditional statements allow a program to choose between different actions based on whether a condition is true or false. The if-then-else structure checks a condition and executes different blocks of code based on the result.

Loops. Loops allow a block of code to repeat either a fixed number of times (as in for loops) or until a condition is met (as in while and do-while loops). Loops are fundamental for performing repetitive tasks, like iterating through an array or list.

Recursion. Recursion is a control structure where a function calls itself to solve a problem by breaking it down into smaller subproblems. It is particularly useful for tasks like calculating factorials or traversing hierarchical data structures (like trees).

Switch-case. The switch-case statement allows for multi-way branching based on the value of a variable. It tests the variable against a list of possible cases and executes the corresponding block of code for the matching case.

Control structures in computers. In computing, control structures are implemented at both hardware and software levels. CPUs manage flow control using mechanisms like instruction pointers, conditional branching, and stack-based recursion. These structures ensure efficient execution of instructions and

complex operations.

Instruction pointers. The CPU uses an instruction pointer to track the next instruction to execute. Sequential control is managed by incrementing this pointer after each instruction.

Conditional branching. Conditional branching enables the CPU to change the flow of execution based on evaluated conditions, such as through if-else structures. Branching is achieved using flags in the CPU's status register.

Loops and recursion in hardware. Loops are managed in hardware by repeatedly checking conditions and branching to the beginning of the loop if necessary. Recursion involves storing return addresses on the stack so the CPU can continue execution after recursive calls.

Control in hardware. At the hardware level, control structures are implemented using various CPU components, including the control unit, arithmetic logic unit (ALU), and registers. These components work together to execute instructions, manage branching, and ensure efficient processing of control structures.

1.6 Functions

In mathematics, a function is a relation between a *set of inputs* and a *set of possible outputs* where each input is related to exactly one output. For example, the function $f(x) = x^2$ takes a number x and returns its square. The essential idea is that a function consistently produces the same output for the same input, making it a fundamental concept for understanding relationships between quantities.

However in computer science, a function is a block of organised, reusable code that performs a single, specific task. Functions take input in the form of parameters, process these inputs, and return a result. They are fundamental to structuring programs, allowing for code reuse, modularity, and abstraction. For example, a function in a programming language might take two numbers as input, add them together, and return the sum. Functions help manage complexity in software by encapsulating behaviour that can be invoked repeatedly throughout a program.

What is interesting in the context of practical computers, functions are typically implemented as *sequences of instructions stored in memory that the CPU can execute*. When a function is called, the computer temporarily halts the execution of the current sequence of instructions, switches to the function's instructions, and then returns to the original sequence once the function has completed. Functions in computers can handle everything from basic arithmetic to complex algorithms, and they can call other functions, enabling intricate chains of computation.

Across these domains, mathematics, computer science and practical computers the concept of a function shares a common theme: a *mapping from inputs*

to outputs. In mathematics, this is a clear, abstract relationship between sets; in logic, it's about assigning outcomes based on logical rules; and in computer science and computing, it translates these ideas into actionable *instructions that a machine can execute to produce a result.* This shared concept underpins the theoretical foundations of computing and the practical implementation of software, illustrating the deep connections between these fields.

1.6.1 Calling functions

The easiest way of implementing a call in a stack machine is to use a stack. Either a call stack separate from the stack of operations, or using the same stack as the operations. When doing a call, the return address can be kept on the stack, and to get back the address is popped from the stack and the instruction pointer is set by its value. Then the program immediately continues after the calling address.

But often arguments are carried with the function call. You might have heard of the concepts "call by reference," "call by value," "call by name," and other related mechanisms. They are all ways in which arguments (or parameters) are passed to functions in programming languages. These mechanisms determine how a function interacts with the variables passed to it, which in turn affects the behaviour of the program. Here are some often used variations:

Call by Value *The function receives a copy of the actual data. Any changes made to the variable inside the function do not affect the original variable outside the function.*

Call by Reference *The function is passed a direct reference to the original variable, allowing it to modify the variable's actual data in the calling environment.*

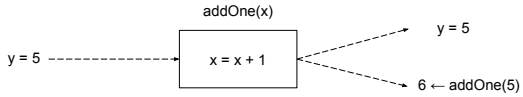
Call by Sharing *Objects are passed by reference, so changes to the object inside the function affect the original, Primitive values are passed as copies, leaving the original values unaffected.*

1. Call by Value

When a function is called by value, the actual value of the argument is passed to the function. This means that the function works on a copy of the data, not the original data. As a result, changes made to the parameter inside the function do not affect the original variable outside the function.

```
! pseudo-code
function addOne(x) {
    x = x + 1;
}
```

```
y = 5;
addOne(y);
```



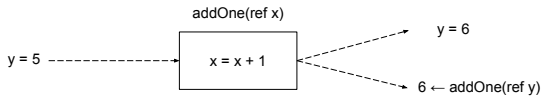
Example: After the function call, ‘y’ remains ‘5’ because the function ‘addOne’ only modifies a copy of ‘y’.

2. Call by Reference

In call by reference, instead of passing the value of an argument, a reference (or pointer) to the actual data is passed. This means the function can directly modify the original variable, and any changes made to the parameter inside the function will affect the variable outside the function.

```
! pseudo-code
function addOne(ref x) {
    x = x + 1;
}

y = 5;
addOne(ref y);
```



Example: After the function call, ‘y’ will be ‘6’ because the function modifies the original data through the reference.

Even though C does not pass by reference when called with primitive types, how the referencing and dereferencing works can be enlightening with an example.

```c
#include <stdio.h>

// function to add one, using a pointer to pass the variable by
    reference
void addOne(int *x) {
    *x = *x + 1;
}

int main() {
    int y = 5;
    addOne(&y); // pass the address of y
    printf("y = %d\n", y); // output: y = 6
    return 0;
}
```

3. Call by sharing, or call by object reference

Call by sharing is common in languages like Python and JavaScript. In this approach, references to objects (like lists or dictionaries) are passed to functions, not the actual objects themselves. This means that the function can modify the contents of the object, and those changes will be reflected outside the function. However, if the reference is reassigned inside the function, it won't affect the original object outside.

For simple data types like integers or booleans, they are passed by value. This means if you try to modify them within the function, only a copy is changed, leaving the original unchanged.

```
! pseudo-code
function modifyList(lst) {
    lst.append(4); ! modifies original list
}

alist = [1, 2, 3];
modifyList(alist);
```

Example: In this case, `alist` is passed by reference, so the `append(4)` operation inside the function modifies the original list. However, if you were to reassign `lst` within the function to a new list (e.g., `lst = [5, 6]`), the original `alist` would remain unchanged outside the function. In Python:

```python
def modify_list(lst):
    lst.append(4) # modifies the original list
```

```
alist = [1, 2, 3]
modify_list(alist)
print(alist) # output: [1, 2, 3, 4]

def reassign_list(lst):
    lst = [5, 6] # does not affect the original list

reassign_list(alist)
print(alist) # output: [1, 2, 3, 4]
```

1.6.2 Summary

Functions. In mathematics, a function is a mapping from inputs to outputs where each input corresponds to one output. In computer science, a function is a reusable block of code that takes input (parameters), processes it, and returns an output. This concept is fundamental to structuring software by promoting modularity and reuse.

Calling functions. In programming, functions can be called using different argument-passing mechanisms, such as call by value, call by reference, and call by sharing. These mechanisms determine how the function interacts with the variables passed to it and whether or not it can modify the original variables.

Call by value. In call by value, a copy of the argument's value is passed to the function, and changes made to the parameter inside the function do not affect the original variable outside the function.

Call by reference. In call by reference, a reference (or pointer) to the original variable is passed to the function, allowing the function to modify the original data directly. Changes made to the parameter inside the function will affect the original variable outside.

Call by sharing. In call by sharing (or call by object reference), a reference to an object is passed to the function, allowing the function to modify the object. However, if the reference is reassigned inside the function, the original object remains unchanged. Primitive types like integers are passed by value, while complex types like lists or dictionaries are passed by reference.

1.7 Practice

Workbook: Chapter 1.
https://github.com/Feyerabend/bb/tree/main/workbook/ch01
Scan the QR code to access exercises, code, resources and updates.

The first chapter of this book outlines some of the groundwork for understanding fundamental programming concepts. It begins by introducing key elements such as variables, control structures, and functions. It also mentions simple data types, illustrating how different data forms—such as integers, floating-point numbers, characters, and strings—are represented in binary. It highlights the significance of understanding binary representation.

However, the landscape of programming is vast, encompassing numerous fundamental concepts, branches of study, and pathways for further exploration. To navigate this complexity, you should tailor your learning journey based on existing knowledge and specific interests. Consequently, the questions one poses may vary widely depending on these factors.

Example: Floating-point numbers

After reading the section on "Floating-point numbers," consider leveraging tools like Large Language Models (LLMs) to investigate the subject more thoroughly. Here are some thought-provoking questions you might explore:

1. *What is the standard 'IEEE 754'?* Understanding this standard is beneficial for grasping how floating-point numbers are represented and manipulated in computer systems.

2. *How does arithmetic work with floating point numbers?* Delve into the intricacies of floating-point arithmetic, including addition, subtraction, multiplication, and division.

3. *Provide an implementation in Python of floating point numbers arithmetic, but not using built-in operations.* Implementing basic arithmetic manually can deepen your understanding of how floating-point representation functions.

More extensive projects can also be found at the repository. By engaging with these questions and projects, you can further your exploration of floating-point numbers and the broader realm of programming concepts, enhancing your overall comprehension and skills.

Example: Koch snowflakes

Niels Fabian Helge von Koch (1870–1924): Von Koch is famous for creating the Koch snowflake, one of the earliest-recognized fractals. The construction of this snowflake involves a recursive process—each side of a triangle is repeatedly divided to create an infinitely complex boundary. This idea directly links to the concept of recursion, where a process calls itself with smaller subsets, similar to recursive functions in programming. Fractals like the Koch snowflake have inspired recursive algorithms used in computer graphics and complex problem-solving.

Ask a LLM: Can you tell me something about Koch and Koch snowflakes?

Read and ask follow-up questions to enhance your knowledge.

Simple exercise: *Drawing the snowflake with simple loops.*

Objective. Use basic programming loops to create the outline of the Koch snowflake without recursion.

Task. The Koch snowflake can be created by starting with a simple triangle and then applying a transformation to each side. In this task, you will use loops to approximate the Koch snowflake shape based on a list of points for each side of the triangle. Before making a program, explore suitable points on a piece of graph paper, and also draw the first iteration. Going to the program:

1. Start with three points that form an equilateral triangle. Investigate suitable start of the coordinates of these three points.
2. Manually add a few extra points along each side, forming a "V" shape to simulate the fractal's look.
3. Write a loop that connects all these points in sequence to form an outline of the Koch snowflake.
4. Experiment with repeating these loops on smaller segments (like halfway points) to create a rough approximation of more fractal detail.

Goal. Write a Python program that loops through the list of points and draws a rough outline of the snowflake using only for loops and print statements or a simple graphics library. This exercise will help you understand the symmetry and structure of the Koch snowflake without requiring recursion.

Advanced exercise: *Koch snowflake with dynamic levels of recursion.*

Objective. Create a program that adjusts the Koch snowflake's recursion depth in real time based on user input.

Task. The Koch snowflake is a fractal shape that gets more complex with each level of recursion. In this task, you will write a Python program that allows the user to specify the recursion depth of the snowflake and then display it on the screen. You'll use recursion to break each side into more detailed segments based on the chosen depth.

1. *Step 1:* Start by defining a function that draws a single Koch curve segment based on a starting and ending point, and a specified depth.
2. *Step 2:* Write a function that will ask the user for an integer value representing the depth. This value will determine the number of times your Koch function is called recursively, making the snowflake more complex with higher values.
3. *Step 3:* Use this depth value to adjust the number of recursive calls in your program, regenerating the snowflake each time the user inputs a new value. If possible, use a simple graphics library to display the snowflake visually and refresh the display after each input.
4. *Step 4:* Experiment with different recursion depths and observe how the complexity changes. What happens when you use a depth of 1, 2, 3, and so on?

Goal. The goal of this exercise is to help you understand how recursion depth affects fractal detail. Write a program in JavaScript that dynamically updates the Koch snowflake based on user input. Focus on how each additional recursion level changes the visual complexity of the shape and reflect on the computational cost as the depth increases.