

# The Quanta Programming Language

---

A Developer's Guide to Memory-Safe, Intelligent, and Embedded-Ready Coding

**Author:** Rohan Kumar Rawat

**Edition:** First Edition

---

## About the Author

---

**Rohan Kumar Rawat** is the creator and architect of the Quanta Programming Language. Driven by a vision to bridge the gap between easy-to-use software and low-level system performance, Rohan designed Quanta from scratch. His engineering philosophy centers on creating developer tools that are predictable, fast, and memory-safe out of the box.

Through Quanta, Rohan has pioneered features like the Dual String System—merging the flexibility of standard applications with the strict safety needed for small devices—and intelligent data types that just work the way you expect them to. When he is not working on Quanta, Rohan is passionate about pushing the boundaries of what modern embedded and general-purpose programming can achieve.

---

## Acknowledgments

---

Creating a programming language from scratch is a monumental endeavor, and Quanta would not exist without a deep dedication to simple, effective design.

Thank you to the early testers, developers, and engineers who looked at the early syntax and saw the potential for a new way of writing safe, embedded-ready code. Your rigorous testing of Quanta's memory safety shaped the language into the robust tool it is today.

Finally, thank you to everyone who believes that writing software does not have to be painfully complex, and that there is always room for a language that simply *works*.

---

## Preface

---

Welcome to **The Quanta Programming Language**.

For a long time, developers have been forced to make a difficult choice. High-level languages are easy to read and handle memory for you, but they are often too heavy and slow for small, embedded devices. Low-level languages give you immense power and control to write for those tiny devices, but they are very hard to learn and require you to manage every byte of memory yourself.

Quanta was built to give you the best of both worlds.

I designed Quanta to be incredibly easy to learn for general-purpose programming, while also giving you the exact tools you need if you decide to write software for embedded operating systems. It is a language where you can simply write `x = 100` and let Quanta figure it out, but also explicitly write `int8 tiny = 100;` when every byte matters. It is a language where normal text strings clean up after themselves automatically, but you can also create strict, fixed-size text strings that are perfectly safe for microcontrollers.

This book is a practical, hands-on guide to writing Quanta code. Throughout these pages, we will explore:

- 1. Intelligent Data Types:** How to easily mix automatic typing with strict data types when needed.
- 2. Built-in Safety:** Understanding how Quanta keeps your code safe from common crashes behind the scenes.

3. **The Dual String Approach:** Learning how to use flexible text for daily applications and rigid, safe text for embedded devices.
4. **Clean Code:** Writing logic and loops that are simple, clear, and behave exactly as expected.

Whether you are building a tool for your computer or writing software for a tiny IoT device, Quanta is here to help you get the job done simply and safely.

Let's start coding.

— Rohan Kumar Rawat

(Creator of Quanta)

---

## Full Table of Contents

---

### Chapter 1: Introduction to Quanta

---

- **1.1 What is Quanta?**: Overview of the language philosophy (simplicity, predictability, memory safety).
- **1.2 Setting up your Environment**: How to install the Quanta compiler and run your first script.
- **1.3 Hello, Quanta**: Writing your first `print("Hello Quanta");` program.
- **1.4 Comments and Code Structure**: Understanding the `@` symbol for comments and basic syntax formatting.

### Chapter 2: Variables, Types, and Scoping

---

- **2.1 Variable Declarations**:
  - Implicit assignment (`x = 100`) vs explicit declaration (`int x = 10;`).

- The `var` keyword for inferred typing.
- **2.2 Core Data Types:**
  - Integers (`int`, `int8`, `int32`, `int64`) and their use cases.
  - Floating-point numbers (`float`).
  - Booleans (`true`, `false`).
- **2.3 Type Introspection:** Using `type()` and `bytesize()` to understand your data at runtime.
- **2.4 Lexical Scoping:** How block scope works (`{ ... }`) and variable shadowing.

## Chapter 3: Operators and Expressions

---

- **3.1 Arithmetic Operators:** Addition, subtraction, multiplication, division, and modulo (`%`).
- **3.2 Order of Operations:** BODMAS rules in Quanta.
- **3.3 Increment and Decrement:** Prefix and postfix operators (`++a`, `a++`, `--a`, `a-`).
- **3.4 Relational Logic:** Equality and comparison (`==`, `!=`, `<`, `>`, `<=`, `>=`).
- **3.5 Implicit Conversions:** How Quanta safely handles type promotion (e.g., `int8` to `int32`) and truncation (e.g., `float` to `int`).

## Chapter 4: Control Flow

---

- **4.1 Conditional Branching:** `if`, `elif`, and `else` statements.
- **4.2 Nested Logic:** Managing deep conditional trees safely.
- **4.3 Truthiness:** How Quanta assesses true and false in expressions.

## Chapter 5: Loops and Iteration

---

- **5.1 The `loop` Construct:** Quanta's unified iteration block (`loop (condition) { ... }`).
- **5.2 Managing State:** Incrementing iterators securely within loops.
- **5.3 Common Loop Patterns:** Counting, accumulating, and infinite loops.

## Chapter 6: The Dual String System (Memory Safety Made Simple)

---

Quanta possesses a unique and powerful approach to string memory, accommodating both embedded and dynamic use cases.

- **6.1 Dynamic Strings ( `string` ):**
  - Heap-allocated strings for flexible, high-level programming.
  - The "Auto-Free" mechanism: Automatic memory management without garbage collection pauses.
- **6.2 Static Embedded Strings ( `string[N]` ):**
  - Stack-allocated strings for absolute memory control (e.g., `string[16] deviceName;` ).
  - Built-in overflow protection and safe truncation.
- **6.3 Interoperability:** Mixing dynamic strings (`string`), static strings (`string[N]`), and inferred strings (`var` ).

## Chapter 7: String Manipulation & Standard Library

---

- **7.1 Concatenation:** Combining strings with the `+` operator.
- **7.2 Inspection Functions:** `len()` , `isupper()` , `islower()` , `isalpha()` , `isdigit()` , `isalnum()` , `isspace()` .
- **7.3 Modification Functions:** `upper()` , `lower()` , `reverse()` , `strip()` , `lstrip()` , `rstrip()` , `capitalize()` , `title()` .

- **7.4 Search & Replace:** `find()`, `count()`, `startswith()`, `endswith()`, `replace()`.

## Chapter 8: Functions and Memory

---

- **8.1 Defining Functions:** Syntax, parameters, and return types.
- **8.2 Implicit vs Explicit Signatures:** Mixing strict typing (`int result()`) with implicit typing.
- **8.3 Default Arguments:** Setting fallback values for arguments (e.g., `getArea(width = 10, height = 20)`).
- **8.4 Named Arguments:** Calling functions clearly using parameter names (`greet(rollno=30, name=40)`).
- **8.5 Memory within Functions:** Returning dynamic strings safely (how Quanta's ReturnAST protects returned memory).
- **8.6 Safely passing static strings:** Sending stack memory into functions.

## Chapter 9: Advanced Quanta Practices

---

- **9.1 Writing Embedded-Safe Code:** Best practices for sticking to stack-only structures when memory constraints are tight.
- **9.2 Debugging Common Errors:** Understanding compiler warnings (e.g., illegal String-to-Int casts).
- **9.3 Bridging C and Quanta:** Understanding how the Quanta LLVM backend interfaces.
- **9.4 Future Proofing:** Preparing for upcoming Quanta features (Arrays, Structs, etc.).

## Chapter 1: Introduction to Quanta

---

Welcome to your first steps with **Quanta**. This chapter will guide you through understanding what makes Quanta unique, how to set up your environment, and how to write your very first running script.

By the end of this chapter, you will understand the basic syntax rules of Quanta and know how to print information to your screen.

---

## 1.1 What is Quanta?

---

Quanta is a programming language designed to be beautiful, predictable, and exceptionally safe.

Historically, writing code meant choosing between two extremes:

1. **General Purpose (High-Level):** Languages like Python or JavaScript. They are easy to read and manage memory automatically, but they can be slow and use too much RAM for tiny microcontrollers.
2. **Systems (Low-Level):** Languages like C or Rust. They run fast and fit on tiny devices, but they force you to manually track every piece of memory, leading to complicated code and frequent crashes.

**Quanta bridges this gap.**

Quanta allows you to write code as easily as you would in a high-level language, but it compiles down to highly efficient instructions that run perfectly on both desktop computers and small embedded devices. It features intelligent data types that figure themselves out, alongside strict memory controls you can activate exactly when you need them.

**[!TIP] The Quanta Philosophy** Code should be easy to write for a human, but perfectly precise for the machine. Quanta abstracts away the chaos of memory

*management without hiding the power of the hardware.*

## 1.2 Setting up Your Environment

To write Quanta code, you simply need a text editor and the Quanta compiler.

### Writing the code

You can write Quanta code in any standard text editor (like VS Code, Sublime Text, or even Notepad). Quanta files always use the `.qnt` file extension.

*Example filename:* `hello.qnt`

### Running the code

When you have written a `.qnt` script, you use the Quanta compiler to run it. The compiler reads your text file, translates it into machine code, and executes it.

```
quanta hello.qnt
```

## 1.3 Hello, Quanta

Let's write our very first Quanta program. The tradition in programming is to make the computer print the phrase "Hello, World!" to the screen.

Create a new file called `hello.qnt` and type the following:

```
@ My first Quanta script  
print("Hello, Quanta!");
```

## The Output

When you run this script using the Quanta compiler, the output on your screen will be:

```
Hello, Quanta!
```

## Breaking it down

Let us look closely at the two lines of code we just wrote.

### The Comment ( @ )

```
@ My first Quanta script
```

The `@` symbol is used for **comments**. Whenever the Quanta compiler sees an `@`, it completely ignores everything else on that line. Comments are just notes left by the programmer for themselves or other humans reading the code. They do not affect how the program runs.

### The Print Statement

```
print("Hello, Quanta!");
```

- `print()` is a **function** built directly into Quanta. Its job is to display whatever you put inside the parentheses onto the screen.
  - `"Hello, Quanta!"` is a **string**. In Quanta, any text wrapped in double quotation marks (`" "`) is treated as text data.
  - **Semicolons are Optional!** You might notice the semicolon (`:`) at the end of the line. Unlike strict languages (like C or Java), Quanta does not force you to put a semicolon at the end of every instruction. You can choose to use them if you prefer that style, or completely leave them out. Quanta understands both!
- 

## 1.4 Variables and Basic Printing

---

The `print()` function is incredibly flexible. You can use it to print text, numbers, and variables (which we will learn about deeply in Chapter 2).

Let's look at a script that prints a mix of different things:

```
@ Printing raw numbers
print(100)

@ Printing the result of math
print(10 + 20)

@ Printing multiple things at once separated by commas
print("The score is:", 99)
```

### The Output

```
100
30
```

```
The score is: 99
```

*[!NOTE] Notice how `print(10 + 20)` prints `30` instead of the text "10 + 20". Because it is not wrapped in quotes, Quanta evaluates the mathematical expression before printing the final result!*

## 1.5 Code Structure and Formatting

Quanta was designed to be easily readable. White space (spaces, tabs, and completely blank lines) does not matter to the compiler. You can use spacing to make your code look clean and organized.

These two scripts do the exact same thing:

### Script A (Cluttered):

```
print("Start") print(1)  
print(2) print("End")
```

### Script B (Clean):

```
print("Start")  
  
print(1)  
print(2)
```

```
print("End")
```

**Always choose Script B.** In Quanta, readability is just as important as functionality.

---

## What's Next?

---

You now know how to create a Quanta file, write comments, and print data to the screen. In the next chapter, "**Variables, Types, and Scoping**", we will learn how Quanta manages data, how it cleverly figures out data types for you, and when you should take manual control over your memory.

# Chapter 2: Variables, Types, and Scoping

---

In the previous chapter, we learned how to output data using the `print()` function. But real programs need to store, track, and manipulate data over time. In Quanta, we do this using **Variables**.

Understanding how Quanta handles variables is the key to unlocking its power. Quanta gives you the rare ability to write flexible, dynamic code, while simultaneously allowing you to lock down specific variables to perfectly fit microcontroller memory.

---

## 2.1 Variable Declarations

---

In Quanta, declaring a variable simply means telling the computer to save a piece of data under a specific name. There are two primary ways to declare variables in Quanta: **Implicit** and **Explicit**.

## Implicit Declaration (Letting Quanta Decide)

If you are writing a quick script or a high-level application, you probably don't want to think about exact byte sizes. You want things to "just work."

In Quanta, you can simply type a name, an equals sign `=`, and the value. The compiler will instantly and automatically assign the safest type for you.

```
@ Quanta automatically knows this is an integer  
x = 100  
  
@ It automatically knows this is a floating-point (decimal) number  
pi = 3.14159  
  
@ It automatically knows this is a dynamic text string  
msg = "Hello Quanta"  
  
@ It automatically knows this is a boolean (True/False)  
flag = true  
  
print(x)  
print(msg)
```

## Output:

```
100  
Hello Quanta
```

## The `var` Keyword

Sometimes, you want to explicitly state that you are letting Quanta infer the type. You can use the `var` keyword.

In Quanta, both `y = 50` and `var y = 50` do the exact same thing! You can choose to assign a variable with `var` or completely without it based on what you find most readable.

```
@ Directly assigning data without "var"  
x = 100  
  
@ Directly assigning data with "var"  
var y = 50  
  
print(x)  
print(y)
```

## Explicit Declaration (Taking Control)

When writing software for an embedded system, every byte of memory matters. If you know a number will never go above `100`, it is a waste of memory to assign it a massive 64-bit integer type.

Quanta allows you to take absolute control over your memory by declaring the exact type before the variable name.

```
@ Standard integer  
int a = 10  
  
@ A tiny 8-bit integer (perfect to save memory on a microchip)  
int8 tiny_number = 100
```

```
@ A 32-bit integer  
int32 medium_number = 50000
```

## 2.2 Core Data Types

Whether Quanta guesses the type implicitly, or you dictate the type explicitly, every piece of data belongs to a **Data Type**.

Here are the primary numerical and logical types you will use:

1. **Integers** (`int`, `int8`, `int32`, `int64`): Whole numbers without decimals.

- o `int8`: Very small numbers.
- o `int32`: Standard sized numbers.
- o `int64`: Massive numbers.
- o `int`: The default integer size chosen by the compiler based on your system.

2. **Floating-point** (`float`): Numbers that contain a decimal point (e.g., `3.14` or `0.001`).

3. **Booleans**: Logical values that can only be `true` or `false`. Internally, Quanta treats `true` as `1` and `false` as `0`.

**[!TIP] Reassigning Variables** Once a variable is created, you can easily change its value by using the `=` sign again.

```
score = 10  
score = 200
```

```
print(score) @ Outputs 200
```

*Quanta is not creating a new block or a brand new variable here! It is simply changing the data stored in the existing `score` variable.*

## 2.3 Type Introspection

Because Quanta allows you to write quick, implicit variables, you might sometimes forget what type of data is actually stored in a variable, or exactly how much memory it is taking up.

Quanta provides two incredibly powerful built-in functions to look inside your variables:

`type()` and `bytesize()`.

```
my_value = 12345

@ Find out what the compiler named the type
print(type(my_value))

@ Find out exactly how many bytes of memory it uses
print(bytesize(my_value))
```

**Output:**

```
int
8
```

(Note: Output may vary based on whether your system dynamically defaults `int` to 4 bytes or 8 bytes).

---

## 2.4 Lexical Scoping (Blocks)

As your programs get larger, you don't want every variable to live forever in memory. **Scope** dictates where a variable exists and where it dies.

In Quanta, a new scope is created anytime you open a pair of curly braces `{ }`. We call this a "block."

Variables created **outside** of a block are *Global* (they can be seen everywhere). Variables created **inside** a block are *Local* (they only exist inside those braces).

```
global_var = 99

if (1 == 1) {
    @ We are inside a new block now!
    local_var = 1

    @ We can see the global variable from inside the block
    print(global_var + local_var)
}

@ If we tried to print(local_var) out here, Quanta would throw an error!
@ local_var was destroyed the moment the block ended.
```

**Output:**

By safely destroying `local_var` at the end of the block, Quanta prevents your system memory from bloating out of control!

---

## What's Next?

---

Now that you can store data and control how much memory it uses, the next step is interacting with it. In **Chapter 3: Operators and Expressions**, we will learn how to perform mathematics, compare values, and see how Quanta prevents crashes when you mix different integer sizes together.

# Chapter 3: Operators and Expressions

---

Now that we know how to store data in variables, it is time to do something with that data. In Quanta, **Operators** are special symbols (like `+` or `==`) that perform operations on variables and values.

When you combine variables and operators together to produce a new value, that combination is called an **Expression**.

---

## 3.1 Arithmetic Operators

---

Quanta supports all standard mathematical operations out of the box.

```
a = 10
b = 3

print(a + b)  @ Addition: Outputs 13
print(a - b)  @ Subtraction: Outputs 7
print(a * b)  @ Multiplication: Outputs 30

@ Division between two integers results in an integer (the decimal is dropped)
print(a / b)  @ Division: Outputs 3 (Not 3.33)

@ Modulo operator returns the remainder of the division
print(a % b)  @ Modulo: Outputs 1
```

If you want precise decimal division, at least one of your numbers needs to be a floating-point number.

```
f1 = 10.5
f2 = 2.5
print(f1 / f2)  @ Outputs 4.200000
```

## 3.2 Order of Operations (BODMAS)

Quanta strictly follows standard mathematical order of operations, often remembered as **BODMAS** (Brackets, Orders, Division/Multiplication, Addition/Subtraction).

```
@ Multiplication happens before addition  
print(2 + 3 * 4)    @ Outputs 14 (Not 20)  
  
@ Slower operations inside parenthesis happen first  
print((2 + 3) * 4) @ Outputs 20  
  
@ Division happens before subtraction  
print(10 - 10 / 2) @ Outputs 5
```

## 3.3 Increment and Decrement

When writing loops or counters, you frequently need to add exactly `1` or subtract exactly `-1` from a variable. Quanta provides short-hand operators to do this quickly: `++` and `--`.

However, *where* you place the operator matters!

### Postfix (`a++` and `a--`)

Placing the operator *after* the variable evaluates the expression using the **old** value first, and then changes the variable.

```
a = 10  
print(a++) @ Outputs 10 (It prints, THEN increments)  
print(a)    @ Outputs 11
```

### Prefix (`++a` and `--a`)

Placing the operator *before* the variable changes the variable first, and then evaluates the expression using the **new** value.

```
x = 10
print(++x) @ Outputs 11 (It increments, THEN prints)
print(x)    @ Outputs 11
```

## 3.4 Relational Logic

Often, you don't just want to do math; you want to compare two values. Relational operators compare two pieces of data and return a **Boolean** (`true` or `false`).

Remember, Quanta prints `1` for true and `0` for false.

```
val = 10

print(val == 10)  @ Equal to: Outputs 1 (True)
print(val != 20)  @ Not equal to: Outputs 1 (True)
print(val > 20)   @ Greater than: Outputs 0 (False)
print(val < 20)   @ Less than: Outputs 1 (True)
print(val >= 10)  @ Greater than or equal to: Outputs 1 (True)
print(val <= 5)   @ Less than or equal to: Outputs 0 (False)
```

## 3.5 Implicit Conversions (Safe Type Casting)

What happens when you mix different types together? For example, adding a small `int8` to a massive `int64`, or adding an integer to a float?

In older systems languages, this could crash your program or create "undefined behavior." Quanta handles this safely through **Implicit Conversion**. The compiler automatically detects a size or type mismatch and casts the smaller/simpler type into the larger/more complex type behind the scenes.

## Promoting to Float

If you mix an integer and a float, Quanta will automatically promote the integer into a float to ensure you don't lose any math precision.

```
print(10 + 2.5) @ Outputs 12.500000
```

## Promoting Integer Sizes (Small to Big)

If you try to return an `int8` inside a function requiring a massive `int32`, Quanta happily accepts it because a small number can easily fit inside a larger memory bucket.

```
int8 tiny = 100  
int32 medium = tiny @ Quanta safely changes tiny into a 32-bit number
```

## Truncation (Big to Small)

What if you try to stuff a massive 64-bit number into a tiny 8-bit bucket? Quanta will execute a safe *truncation*. It will chop off the extra binary bits that don't fit.

```
int64 massive = 200  
int8 tiny = massive @ The number 200 overflows int8 bounds (-128 to +127)
```

```
print(tiny) @ Outputs -56 (Safely wrapped around via truncation)
```

*[!WARNING] While Quanta handles these conversions safely, dumping massive numbers into tiny blocks of memory naturally changes the value of the number via binary truncation. Always be sure your variables are sized appropriately for the data they will hold!*

## What's Next?

We can now do math and compare values. The real magic of programming happens when we use those comparisons to make decisions. In **Chapter 4: Control Flow**, we will learn how to make Quanta run different blocks of code depending on whether those comparisons result in `true` or `false`.

# Chapter 4: Control Flow

Up until now, our Quanta scripts have executed in a straight line—running line 1, then line 2, and so on. But real applications require programs to make decisions. They need to execute certain blocks of code only if certain conditions are met.

This decision-making process is called **Control Flow**, and in Quanta, it is handled by `if` , `elif` , and `else` statements.

## 4.1 Conditional Branching ( `if` )

The simplest way to control the flow of your program is the `if` statement. An `if` statement evaluates a logical expression (like the ones we learned in Chapter 3). If that expression is true ( `1` ), Quanta executes the code block inside the curly braces `{ }`. If it is false ( `0` ), Quanta skips the block entirely.

```
check = 50

@ This condition is true, so the block will run!
if (check == 50) {
    print("Check is exactly 50")
}

@ This condition is false, so Quanta ignores this block!
if (check > 100) {
    print("Check is a massive number")
}
```

### Output:

```
Check is exactly 50
```

## 4.2 Handling Alternatives ( `else` )

Sometimes, you want to explicitly tell Quanta what to do if a condition fails. You do this by attaching an `else` block directly after your `if` block.

Only one of these blocks will ever run. It is an either/or decision.

```
account_balance = 10
item_price = 20

if (account_balance >= item_price) {
    print("Item purchased!")
} else {
    print("Insufficient funds.")
}
```

**Output:**

```
Insufficient funds.
```

## 4.3 Chaining Conditions (`elif`)

Life is rarely a simple choice between two options. When you have a complex scenario with multiple possibilities, you can use `elif` (short for "else if") to chain conditions together.

Quanta checks these conditions starting from the top. The moment it finds a condition that evaluates to `true`, it runs that specific block and ignores the rest of the chain.

```
score = 75

if (score > 90) {
    print("Grade A")
} elif (score > 70) {
```

```
    print("Grade B")
} elif (score > 50) {
    print("Grade C")
} else {
    print("Fail")
}
```

## Output:

```
Grade B
```

*Note: Even though `score > 50` is also technically true, Quanta already executed the `score > 70` block, so it skips the rest of the chain.*

## The Magic of the Missing "Switch"

In many traditional languages (like C, Java, or C++), if you have a long chain of conditions, you are forced to use a complex `switch` or `case` statement to ensure the program runs fast. These statements require annoying syntax, extra keywords like `break`, and create messy code.

**Quanta does not have a `switch` statement.**

Why? Because you don't need one! The Quanta compiler is incredibly intelligent. When it sees a long chain of `if` and `elif` statements, the compiler **automatically converts them into highly optimized switch logic behind the scenes.**

By omitting the `switch` statement, Quanta saves you the time of learning unnecessary syntax and keeps your code perfectly clean, while still delivering the exact same high-speed performance.

## 4.4 Nested Logic

Conditions can become incredibly complex. You might need to check one fact, and only if that is true, check another fact. In Quanta, you can easily nest `if` blocks inside of other `if` blocks.

```
speed = 85
is_raining = true

if (speed > 70) {
    print("You are speeding.")

    @ We only check the rain IF they are already speeding!
    if (is_raining == true) {
        print("Warning: Speeding in the rain is very dangerous!")
    }
} else {
    print("You are driving safely.")
}
```

### Output:

```
You are speeding.
Warning: Speeding in the rain is very dangerous!
```

*[!WARNING] While nesting `if` blocks is a powerful tool, be careful not to nest them too deeply! A block inside a block inside a block becomes very difficult for humans to*

*read and understand. Try to keep your code Structure "flat" and clean whenever possible.*

## 4.5 Truthiness

In Chapter 3, we learned that Quanta considers `1` to be true and `0` to be false. The `if` statement relies entirely on evaluating an expression down to one of those two numbers.

```
@ Since 'true' literally equates to '1', this runs perfectly!
if (true) {
    print("This will always run.")
}

@ Since 'false' equates to '0', this will never run!
if (false) {
    print("This will never print.")
}
```

## What's Next?

We have learned how to execute code only once based on a condition. But what if we want to execute code *multiple times* based on a condition? In **Chapter 5: Loops and Iteration**, we will learn how to use Quanta's unified `loop` structure to iterate safely and repetitively over blocks of code.

# Chapter 5: Loops and Iteration

In Chapters 3 and 4, we learned how to make mathematical decisions and execute specific blocks of code exactly *once* using `if` and `else`.

But what happens when you need to run the exact same block of code 10 times? Or 100 times? Or indefinitely, until a specific sensor is triggered on an embedded device? To achieve this, we use **Iteration**.

Instead of writing out the same `print()` statement a hundred times, Quanta allows us to use **Loops** to repeat actions efficiently.

---

## 5.1 The `loop` Construct

Most programming languages force you to learn multiple different types of loops: `while` loops, `for` loops, `do-while` loops, and so on. They all have different, often complex, syntaxes.

In keeping with the philosophy of extreme simplicity, Quanta has unified all of this into a single, powerful keyword: `loop`.

The `loop` structure is incredibly simple. You provide a single condition inside parentheses. As long as that condition remains `true` (or `1`), the block of code inside the curly braces will run again and again.

```
int i = 0

@ As long as 'i' is less than 5, keep running this block!
loop (i < 5) {
    print(i)

    @ We MUST increment 'i', otherwise the loop will run forever!
```

```
i++  
}
```

## Output:

```
0  
1  
2  
4
```

## 5.2 Managing State

The most critical part of writing a loop is managing its **state**. If the condition you provide to the `loop` command *never* becomes false, the loop will run infinitely, locking up your computer or embedded device.

In the example above, we managed our state perfectly using three distinct steps:

- 1. Initialization:** Before the loop starts, we created a variable to track our state (`int i = 0`).
- 2. Condition:** We told the loop when to stop running (`i < 5`).
- 3. Mutation:** Inside the loop, at the very end of the block, we mutated our state by adding `1` to `i` (`i++`).

If we forgot step 3, `i` would forever remain `0`, and the loop would never end because `0` is always less than `5`.

## A Common Mistake

```
@ DANGEROUS CODE - DO NOT RUN!
int countdown = 10

loop (countdown > 0) {
    print("Countdown is active!")
    @ Oops! We forgot to write 'countdown--'
}
```

## 5.3 Common Loop Patterns

Because Quanta's `loop` is so flexible, you can use it to build any iteration pattern you need.

### 1. Counting

This is the most standard loop, designed to run a specific number of times.

```
count = 1
loop (count <= 3) {
    print("Running...")
    count = count + 1    @ This does the same thing as count++
}
```

#### Output:

```
Running...
Running...
```

Running...

## 2. Accumulating Processing

Loops are incredible for doing repetitive math or building up data over time.

```
sum = 0
current_number = 1

@ Add all numbers from 1 to 5 together
loop (current_number <= 5) {
    sum = sum + current_number
    current_number++
}

print("The total sum is:")
print(sum)
```

### Output:

```
The total sum is:
15
```

## 3. Infinite Loops (For Embedded Devices)

Sometimes, an infinite loop *isn't* a mistake. In fact, if you are writing firmware for a microcontroller or an embedded operating system, the entire program is usually just one massive infinite loop! The device boots up, and then loops forever, constantly checking sensors and processing data.

You can easily write an infinite loop in Quanta by just passing `1` or `true` as the condition.

```
@ A standard embedded device main loop
loop (true) {
    @ Read sensor data
    @ Process inputs
    @ Update the device screen
}
```

## What's Next?

---

You now have the core foundational logical tools of programming: variables to store data, math to change data, branching to make decisions, and loops to repeat tasks.

However, moving data around inside variables has always been the most dangerous part of system programming, specifically when dealing with text. In **Chapter 6: The Dual String System**, we will explore Quanta's crown jewel. We will learn how Quanta manages dynamic strings safely for major applications, and how it strictly limits static strings to protect embedded microcontrollers.

# Chapter 6: The Dual String System

---

Text manipulation is one of the most common tasks in programming, but it is also one of the most dangerous. Text strings are inherently variable in size. A user might enter a 3-character name ("Bob") or a 50-character name.

In traditional low-level languages, you are forced to manually allocate large chunks of memory, keep track of them, and manually destroy them. If you fail to manage them perfectly, you cause "Memory Leaks" (your program eats up all the RAM) or "Buffer Overflows" (your program crashes horribly).

Quanta completely solves this problem through its unique **Dual String System**.

Quanta provides two entirely different types of strings: **Dynamic** (for flexible, high-level programming) and **Static** (for rigid, safe embedded systems).

---

## 6.1 Dynamic Strings (`string`)

When you are writing general-purpose applications, you don't want to think about memory. You want strings to grow, shrink, and combine effortlessly. For this, Quanta provides the standard dynamic `string` type.

When you declare a dynamic `string` (or let Quanta implicitly infer a string with `var`), the data is placed on what is called the "Heap." The Heap is a large pool of memory that can hold data of any size.

```
@ The compiler automatically manages this memory
string first = "Hello"
string second = "Quanta"

@ Joining them together automatically requests new memory behind the scenes
string greeting = first + " " + second + "!"

print(greeting)
```

### The Magic of the "Auto-Free" Mechanism

In other languages, every time you combined strings in a loop, you would slowly leak memory until the computer crashed, unless you specifically wrote code to destroy the old strings.

Quanta features a hidden **Auto-Free List**. Behind the scenes, Quanta watches every dynamic string you create. The absolute millisecond a string is no longer needed (like when the program exits a block or finishes a loop), Quanta destroys it and returns the memory to the system.

You get all the safety of an automatic Garbage Collector, but zero of the slow, stuttering performance drops!

```
int i = 0
string builder = "Start-"

@ Watch Quanta securely handle memory inside a loop
loop (i < 5) {
    @ Every loop, this asks the system for new memory...
    builder = builder + "x"
    i++

    @ ...and Quanta's Auto-Free mechanism instantly cleans up the old one
}

print(builder) @ Outputs: Start-xxxxx
```

## 6.2 Static Embedded Strings ( `string[N]` )

What if you are programming a microchip that only has 2 kilobytes of memory? Requesting random, unknown amounts of memory from a "Heap" is a recipe for disaster. Embedded programmers need data that is locked into a fixed size.

For this, Quanta provides the **Static String**.

By placing a number in brackets after the `string` declaration, you tell Quanta to allocate exactly that number of bytes on the computer's "Stack."

```
@ This allocates exactly 16 bytes of memory. No more, no less.  
string[16] deviceName = "Sensor_A1"  
  
print(deviceName)
```

## Automatic Overflow Protection

In languages like C, if you have an 8-byte bucket and try to pour 15 bytes of text into it, the bucket spills over. This "Buffer Overflow" overwrites other parts of your program, causing an instant, fatal crash (a "SegFault").

**Quanta prevents Buffer Overflows entirely.**

If you attempt to stuff too much text into a small static string, the Quanta compiler simply chops off the excess data, ensuring your program never crashes.

```
@ Create an 8 byte bucket  
string[8] smallBuffer = "SuperLongString"  
  
@ Quanta safely truncates the string to fit the 8 bytes!  
print("Truncated:", smallBuffer)
```

### Output:

```
Truncated: SuperLo
```

*(Note: The text is truncated to 7 characters because the 8th byte is secretly reserved by Quanta to mark the absolute end of the string).*

## 6.3 Interoperability

The true beauty of the Dual String System is that you don't have to choose just one. Quanta allows you to seamlessly mix rigid static strings acting alongside flexible dynamic strings without any complex casting!

```
@ A static string safely locked to 8 bytes
string[8] fix = "AB"

@ A standard dynamic heap string
string dynPart = "CD"

@ Combining them works perfectly!
string combined = fix + dynPart

print("Static + Dynamic:", combined)
```

## 6.4 String Indexing and Slicing

Quanta provides robust, built-in tools for extracting specific pieces of text out of a string without needing any external functions.

### Indexing (Getting a Single Character)

You can extract a single character by placing its index inside square brackets `[]`. In programming, counting always starts at `0`. You can also use negative numbers to count backward from the end!

```
string word = "Quanta"

print(word[0])      @ Outputs: Q (the first character)
print(word[2])      @ Outputs: a (the third character)

@ Negative indexing
print(word[-1])    @ Outputs: a (the very last character)
print(word[-2])    @ Outputs: t (the second to last character)
```

## Slicing (Getting a Chunk of Text)

By using a colon `:`, you can create a "slice" of a string from a start point to an end point. Format: `[start : end]` (The start is included, but the end is excluded).

```
string a = "Hello"

print(a[0:2])      @ Outputs: He (Indexes 0 and 1)
print(a[1:4])      @ Outputs: ell (Indexes 1, 2, and 3)

@ You can slice using negative numbers too!
print(a[0:-1])    @ Outputs: Hell (Everything except the last character)
```

## Step Slicing (Skipping Characters)

You can add a second colon to indicate a "step". This tells Quanta to skip characters. A negative step will actually reverse the string! Format: `[start : end : step]`

```
string numbers = "0123456789"

@ Start at 0, go to 10, jumping by 2
print(numbers[0:10:2])  @ Outputs: 02468
```

```
@ Reverse the string completely!
print(numbers[9:0:-1])  @ Outputs: 9876543210
```

---

## What's Next?

---

Now that we understand how Quanta safely stores text across both the Heap and the Stack, it is time to manipulate that text. In **Chapter 7: String Manipulation**, we will dive into all of the powerful, built-in string functions Quanta provides out of the box.

# Chapter 7: String Manipulation & Standard Library

---

Quanta's Dual String System handles all the memory logistics for you. Now, we can focus on actually modifying that text data. The Quanta Standard Library includes a powerful suite of built-in functions designed to work seamlessly across both dynamic (`string`) and static (`string[N]`) text types.

---

## 7.1 Concatenation (Joining Text)

---

We have seen this in previous chapters: you can instantly join multiple strings together using the `+` operator.

Because Quanta is built to be helpful, you can combine Dynamic strings, Static strings, and regular text literals safely.

```
string dyn = "High-Level"
string[16] stat = "Embedded"

@ Quanta handles the conversions internally
string combined = dyn + " & " + stat

print(combined)
```

## Output:

```
High-Level & Embedded
```

## 7.2 Inspection Functions

Sometimes you just need to know facts about the text you have. These functions evaluate your strings and return information (often Booleans).

### Checking Length `len()`

Returns the exact number of characters currently inside the string. Note that for static strings (`string[N]`), this returns the *current text length*, not the maximum memory size!

```
text = "Quanta"
string[20] stat = "Hi"
```

```
print(len(text)) @ Outputs 6  
print(len(stat)) @ Outputs 2 (Even though it holds up to 20 bytes)
```

## Truth Checks

Quanta includes a suite of boolean checks to analyze the contents of your text. These all return `1` (True) or `0` (False):

- `isupper("HELLO")` -> Checks if all letters are capitalized.
- `islower("hello")` -> Checks if all letters are lowercase.
- `isalpha("abc")` -> Checks if the string only contains alphabetical letters.
- `isdigit("123")` -> Checks if the string only contains numbers.
- `isalnum("abc12")` -> Checks if the string contains only letters and numbers (no punctuation).
- `isspace(" ")` -> Checks if the string is entirely made of spaces.

```
user_input = "12345"  
  
if (isdigit(user_input)) {  
    print("Valid number provided.")  
}
```

## 7.3 Modification Functions

These functions take your original text, modify it, and return a *new* string containing the altered text.

*[!NOTE] These functions do not change your original variable! They create new data.  
If you want to overwrite your old variable, you must reassign it like this: `text = upper(text)`*

- `upper(text)` -> Converts all letters to UPPERCASE.
- `lower(text)` -> Converts all letters to lowercase.
- `reverse(text)` -> Flips the text backwards.
- `capitalize(text)` -> Only capitalizes the very first letter of the string.
- `title(text)` -> Capitalizes the first letter of every word in the string.

```
msg = "hello quanta"

print(upper(msg))      @ Outputs: HELLO QUANTA
print(title(msg))      @ Outputs: Hello Quanta
print(reverse(msg))    @ Outputs: atnaug olleh
```

## Removing Whitespace

Data coming from users or sensors often has messy whitespace at the start or end. Quanta provides tools to clean it:

- `strip(text)` -> Removes spaces from *both* ends.
- `lstrip(text)` -> Removes spaces from the *left* side only.
- `rstrip(text)` -> Removes spaces from the *right* side only.

```
messy = "    Sensor Data    "
clean = strip(messy)
```

```
print(clean) @ Outputs: Sensor Data
```

## 7.4 Search and Replace

Finding specific text within a larger body of text is a foundational programming requirement. Quanta provides tools to scan and replace items.

### Checking Boundaries

- `startswith(text, "query")` -> Returns `1` if the text begins with the query.
- `endswith(text, "query")` -> Returns `1` if the text finishes with the query.

```
filename = "report.qnt"

if (endswith(filename, ".qnt")) {
    print("This is a Quanta file!")
}
```

### Deep Searching

- `find(text, "query")` -> Searches the text and returns the index location (the character position) where it found the query.
- `count(text, "query")` -> Searches the text and tells you how many times the query appeared.

```
data = "error: voltage drop. error: overheat."  
  
print(count(data, "error")) @ Outputs: 2
```

## Replacing Text

- `replace(text, "old", "new")` -> Scans the text, finds every instance of "old", and replaces it with "new".

```
raw = "I hate bugs."  
fixed = replace(raw, "hate", "love")  
  
print(fixed) @ Outputs: I love bugs.
```

## What's Next?

---

We have learned how to use all the tools Quanta provides for us. Now, it is time to build our own tools. In **Chapter 8: Functions**, we will learn how to package our code into reusable blocks, how to manage function arguments, and how Quanta's memory safety rules apply when handing data between functions.

# Chapter 8: Functions and Memory

---

As your programs grow, you will quickly find yourself writing the exact same blocks of code over and over again. To keep your code clean, readable, and easy to fix, you can bundle

that code into a **Function**.

In Chapter 1, we learned about the built-in `print()` function. In this chapter, we will learn how to build our own.

---

## 8.1 Defining Functions

A function is essentially a mini-program inside your main program. It takes inputs (arguments), does some work, and then hands an output back (a return value).

### Implicit vs Explicit Signatures

Just like with variables, Quanta gives you the power to choose between implicit, easy-to-read syntax, or strict, explicit syntax for embedded environments.

**Explicit (The Strict Way):** You tell Quanta exactly what type of data the function will return, and exactly what type of data it accepts.

```
@ This function MUST return a standard integer
int add(int a, int b) {
    return a + b
}

print(add(10, 20)) @ Outputs 30
```

**Implicit (The Easy Way):** You can completely drop the type declarations and let the Quanta compiler figure it out based on what you pass in!

```
add(a, b) {
    return a + b
```

```
}
```

  

```
print(add(10, 20)) @ Outputs 30
```

## 8.2 Positional, Keyword, and Default Arguments

Quanta provides several powerful, modern features for passing data into your functions, saving you from writing unnecessary boilerplate code.

### Positional Arguments

By default, arguments are positional. This means the first value you pass goes to the first argument, the second to the second, and so on.

```
add(a, b) {
    return a + b
}
@ 10 goes to 'a', 20 goes to 'b'
add(10, 20)
```

### Default Arguments

If a function normally uses the same value over and over again, you can set a **Default Argument**. If the user doesn't provide that specific piece of data when calling the function, Quanta will automatically plug in the default.

```
@ If the user doesn't provide width or height, they default to 10 and 20.
int getArea(width = 10, height = 20) {
```

```
        return width * height  
    }  
  
    @ Test 1: Uses the defaults (10 * 20)  
    print(getArea())           @ Outputs: 200  
  
    @ Test 2: Overrides the width, but uses the default height (5 * 20)  
    print(getArea(5))         @ Outputs: 100  
  
    @ Test 3: Overrides both (5 * 5)  
    print(getArea(5, 5))      @ Outputs: 25
```

## Keyword (Named) Arguments

Sometimes functions have many arguments, and it becomes confusing to remember what order they go in. Quanta lets you pass arguments by explicitly naming them using **Keyword Arguments**, no matter what order the function was defined in.

```
string greet(name, rollno) {  
    print("Name:", name, "Roll:", rollno)  
    return "Done"  
}  
  
@ Even though 'rollno' was defined second, we can pass it first!  
greet(rollno=30, name="Rohan")
```

## 8.3 Memory within Functions

In Chapter 6, we learned about Quanta's hidden **Auto-Free Mechanism**. We learned that the microsecond Quanta finishes executing a block of code, it instantly sweeps the room

and destroys any dynamic `string` you created to prevent memory leaks.

But what happens if a function *creates* a `string` and tries to *return* it to the main program?

```
string createMessage() {
    string a = "Memory"
    string b = "Safe"

    @ Joining these creates a brand new string on the Heap!
    return a + " " + b
}
```

If the Auto-Free list destroyed everything when the function ended, your program would crash the moment it tried to read the returned string!

## How Quanta Handles It

The Quanta compiler is incredibly smart. When it analyzes your code, it utilizes a custom `ReturnAST` system.

If Quanta notices that a dynamic `string` is being specifically generated for a `return` statement, it flags that specific bundle of memory as an **Exception**. Quanta will destroy the temporary strings (`a` and `b`), but it will safely hand the combined returned string back to the main program without deleting it.

You get all the safety of an automatic memory manager without ever having to write `malloc` or worry about your returned data vanishing into thin air!

---

## 8.4 Safely Passing Static Strings

Quanta's strict safety checks also apply when moving data *into* a function.

If you have a function that expects a dynamic heap `string`, you can safely pass a hard-locked static `string[16]` into it. Quanta natively understands how to bridge the strict stack memory into the flexible function without requiring confusing type-casting or memory pointers.

```
printLength(string input) {
    print("The length is:", len(input))
}

@ We create a rigid, 10-byte static string on the Stack
string[10] limitedString = "Quanta"

@ We can safely pass it right into the function!
printLength(limitedString)
```

## What's Next?

---

You now understand how to build robust, memory-safe programs in Quanta. In the final chapter, **Chapter 9: Advanced Quanta Practices**, we will cover the best practices for writing code specifically for embedded constraints, debugging compiler warnings, and how Quanta interfaces with its underlying C backend.

# Chapter 9: Advanced Quanta Practices

---

Throughout this book, you have learned the core syntax, data types, and logical structures of Quanta. You now possess the tools required to build fast, memory-safe algorithms and applications.

In this final chapter, we will discuss advanced coding philosophies, best practices for specialized environments (like microcontrollers), how to interpret warnings, and a look at what is coming next for the language.

---

## 9.1 Writing Embedded-Safe Code

---

As discussed in Chapter 6, Quanta is uniquely positioned to handle both high-level processing and low-level firmware. If your goal is to exclusively deploy Quanta code onto constrained environments (such as an IoT sensor with only 4 KB of RAM), you should adopt an **"Embedded-Safe"** mindset.

### 1. Avoid Dynamic Types

While Quanta's "Auto-Free" heap memory is brilliant for desktop software, asking an operating system to manage heap memory costs precious hardware cycles and risks heap fragmentation over long periods.

- **Instead of:** `string msg = "Data"` (Dynamic Heap)
- **Use:** `string[16] msg = "Data"` (Static Stack)

### 2. Lock Down Integer Sizes

Letting the compiler pick an `int` size might default to a 64-bit number depending on the architecture. This wastes memory if you are only tracking numbers from 0 to 10.

- **Instead of:** `count = 5` or `var count = 5`
- **Use:** `int8 count = 5`

### 3. Infinite Control Loops

Embedded devices rarely "finish" a task and exit. They boot up and spin in a continuous loop reading sensors and driving hardware. Rely heavily on the `loop (true)` construct to build your main system architecture.

---

## 9.2 Debugging Common Errors

Quanta's compiler is designed to catch fatal crashes *before* the code ever runs on your machine. Sometimes, this means the compiler will throw a red error and refuse to compile. Understanding these is key to moving fast.

### Illegal Casting

The most common error developers encounter involves type casting. While Quanta gracefully handles safe truncations (e.g., throwing a massive 64-bit integer into an 8-bit bucket), it draws a hard line at illogical conversions.

```
@ ERROR: Cannot convert String to Int!
int value = "This should fail"
```

If you accidentally try to do math on text, Quanta will halt compilation immediately rather than creating bizarre "undefined behavior."

### Out of Bounds Static Slicing

While Quanta safely truncates static strings (`string[8]`) when you assign too much data to them, you must be careful when mathematically indexing them.

```
string[5] data = "123"

@ This is safe. The string has 3 characters.
print(data[2])

@ DANGER: The string does not have a 10th character!
print(data[9])
```

Always ensure your logic verifies the `len()` of a string before dynamically requesting an index position!

---

## 9.3 Future Proofing Your Code

---

Quanta is a rapidly evolving language with a clear ethos: **Developer ease without sacrificing machine performance.** The current foundation prioritizes logic, basic data structures, and the Dual String System.

As the language grows, several major data structures are on the horizon. Writing clean, modular code today ensures you can easily adopt these in the future:

- 1. Arrays and Lists:** While Quanta expertly slices and organizes strings, true data arrays (like `[1, 2, 3]`) are a natural next step for managing datasets.
- 2. Structs and Objects:** The ability to bundle multiple variables under a single custom data type (like a `Player` struct possessing `int health` and `string[10] name`).
- 3. Hardware Integrations:** More native functions exposing raw bit-wise shifting and direct memory registry access for ultra-low-level embedded work.

## Final Note

---

Programming languages are merely tools to help humans instruct machines. Some tools are very heavy to swing; some are too light to make an impact. Quanta was forged to be perfectly balanced.

You now have everything you need to start writing real Quanta applications. Happy coding!

— Rohan Kumar Rawat, *Creator of Quanta*