1️⃣

Chapter 1: Variables and Data Types

📦 Chapter 1: Variables, Data Types & Input in C

🌱 The Story So Far...

You've met C. You said "Hello, World!"

But your program was like a tourist — it just *spoke* and then left. No memory, no personalization. Just vibes.

Now it's time to give it **memory** — the ability to **store, remember, and use data**. That's where **variables** come in.

🎭 Meet the Variables – The Actors of Your Code

In any story, characters need names and roles.

In C, **variables** are like characters — they're **named storage boxes** that hold data.

```
int a; // Declaring a variable named 'a' of type int
a = 100; // Initializing 'a' with the value 100
```

> 🧠 Think of int a; as "Hey compiler, I'm reserving a box named 'a' to hold an integer."
>
> And `a = 100;` as "Now put 100 inside that box."

🧰 Variable Syntax & Steps

🧾 Step-by-step:

1. **Declaration** – Tells the compiler you're creating a variable:
   `int a;`
2. **Initialization** – Gives the variable a value:
   `a = 100;`
3. **Use/Print/Operate** – Use it in code:
   `printf("%d", a);`

> ⚠️ C Rule: You must declare the variable before using it.
>
> Unlike Python, C doesn't guess your intent.

💡 Example: `variables.c`

```
#include <stdio.h>

int main()
{
    int a;                                  // Variable Declaration
    a = 100;                                // Variable Initialization
    printf("The output of this program is %d", a); // Prints the integer when %d i
```

```
s present
    return 0;
}

// Compiler(GCC) will completely ignore this comment
// Unlike Python, in C we have to first declare the variable, then intialize the va
riable
// And then we can print the value

/*
This is an
Multiline comment
*/
```

✏️ Variable Naming Rules (aka “C’s Baby Name Book”)

```
int fun_car;   // ✅ Valid
int _hi;       // ✅ Valid
int 7hello;    // ❌ Starts with a digit → INVALID
int @me;       // ❌ Special characters → INVALID
```

✅ Valid Names:

- Start with a letter or underscore
- Can include letters, digits, and underscores
- Case-sensitive (`score`, `Score`, `SCORE` → 3 different variables)

❌ Invalid Names:

- Can’t start with a number
- No spaces or special characters except `_`
- Can't use **reserved keywords**

🧙 Reserved Keywords (The VIP List of C 🛡️)

There are **32 reserved words** in C that you **cannot** use as variable names:

```
auto      break     case      char      const     continue
default   do        double    else      enum      extern
float     for       goto      if        int       long
register  return    short     signed    sizeof    static
struct    switch    typedef   union     unsigned  void
volatile  while
```

> 🧠 These are the sacred spells of C. Don't mess with them — use them wisely.

🔣 Data Types – What's in the Box?

Let's meet the core data types of C:

🎯 `int` – Whole numbers

```
int age = 21;
```

🌊 `float` – Decimal values (single precision)

```
float pi = 3.14;
```

🔡 `char` – Single characters

```
char grade = 'A'; // Note: Use single quotes!
```

📏 Example: `data_types.c`

```
#include <stdio.h>

int main()
{
    int a;
    a = 34;

    float b;
    b = 23.7;

    char c;
    c = 'd';

    printf("The value of a is %d\n", a);  // %d for int
    printf("The value of b is %f\n", b);  // %f for float
    printf("The value of c is %c\n", c);  // %c for char

    return 0;
}
```

> 🧠 Don't forget \n for new line!

⚖️ `sizeof` – Checking Box Sizes 📦

Want to know how much memory a variable takes?

```
#include <stdio.h>

int main()
{
    printf("Size of int is: %zu bytes\n", sizeof(int));
    printf("Size of float is: %zu bytes\n", sizeof(float));
    printf("Size of char is: %zu byte\n", sizeof(char));
```

```
    return 0;
}
```

- `sizeof` is a compile-time operator
- `%zu` is the proper format for `size_t` values

> 📏 Helps you optimize memory in embedded systems or large programs.

🎤 Let's Talk: Input from the User

Until now, we fed fixed values into the code.

Let's let the user **talk back**. 🧑‍💻

🧾 Example: `input.c`

```
#include <stdio.h>

int main(){
    int a;
    scanf("%d", &a); // &a is address-of operator → Tells where to store input
    printf("The value of a is %d\n", a);
    return 0;
}
```

🛠️ Breakdown:

- `scanf()` reads input
- `" %d "` tells it to expect an integer
- `&a` gives the **memory address** of `a` to store the result

> 💡 Forgetting & in scanf() is one of the most common C errors.

🚫 Common Mistakes Beginners Make

Mistake	Why It Happens	Fix
Using variable before declaring	Python habit	Always declare first in C
`scanf("%d", a);`	Missing `&`	Use `&a` to give memory address
Using wrong format specifier	`%s` for int? Nah	`%d` → int, `%f` → float, `%c` → char
Using double quotes for char	`"A"` instead of `'A'`	Use single quotes for `char`
Misspelling `main` or `printf`	C is case-sensitive	Check spelling & cases

---

📚 Chapter Summary Table

Concept	Keyword/Function	Format
Declare int	`int a;`	`int` type
Assign value	`a = 10;`	Use `=`
Output int	`printf("%d", a);`	`%d` for integers
Input int	`scanf("%d", &a);`	`&a` gets memory address
Data Types	`int`, `float`, `char`	Built-in types
Check size	`sizeof(int)`	Memory size in bytes
Invalid Names	`7abc`, `@val`	Must start with letter or `_`
Reserved Words	`int`, `return`, `while`, etc.	32 total

💬 Final Thoughts

> "Variables are how your program remembers things. Data types are how it understands them. Input/output is how it talks to the world."

Mastering these three gives you *80% of what you need* to build real programs.

You've just learned how to **give memory, meaning, and voice** to your code.

🧠 DEEP DIVE: Variables, Data Types, and Input in C

Chapter 1 – Advanced Theory & Internal Mechanics

🎬 Opening Scene: What Actually Happens When You Write `int a = 5;` ?

So you write:

```
int a = 5;
```


But what does the **compiler** and your **CPU** actually *do* with this?

1. 🔧 **Compiler:** Allocates a chunk of memory big enough to hold an `int` (usually 4 bytes).
2. 🧠 **Gives that memory a label** – `a` is just a name for this address.
3. 📦 **Stores the value** `5` in that memory slot.
4. 🗺️ **Generates machine code** to fetch and use it later.

> 💡 A variable is not the value, it's the label for a memory address that holds the value. Think of it like a sticky note on a box.

🧮 Memory Model – Think Like the Computer 🧠💾

Visualize your RAM like a **long row of mailboxes** (memory cells):

Address	Content
`0x100`	5
`0x104`	(next int)

If `a` is stored at `0x100`, then `&a` gives you that **exact address**. C lets you access this using the `&` (address-of) and `*` (dereference) operators. That's where

pointers come in (we'll get to that later 😉).

🧪 Data Types: Behind the Curtain

Every data type defines:

1. **Size in memory**
2. **Range of values**
3. **Operations allowed**

Type	Bytes (Typical)	Value Range
`int`	4	−2,147,483,648 to 2,147,483,647
`float`	4	3.4e−38 to 3.4e+38 (7 digits precision)
`double`	8	15–16 digits of precision
`char`	1	ASCII characters (0 to 255 or −128 to 127)

> ⚠️ Size may vary based on architecture (32-bit vs 64-bit). Use sizeof() to be sure.

🔠 Signed vs Unsigned

Every numeric type can be **signed** or **unsigned**:

- `signed int`: Can store both positive and negative values.
- `unsigned int`: Only positive, but **twice the range**.

```
unsigned int age = 4294967295; // max value without sign bit
```

> 💡 The first bit in signed integers is the sign bit. 0 means positive, 1 means negative.

🧱 Memory Alignment (Why Size Matters)

Ever wondered *why* an `int` is 4 bytes?

- The CPU fetches memory in **blocks**.
- Aligning data on word boundaries (4 or 8 bytes) **speeds up** performance.
- Misaligned data can lead to **extra cycles** or even **crashes** on older systems.

> 🧠 That's why sizeof(char) is 1, but sizeof(struct) might be padded to 8 or 12 bytes!

📌 Variable Lifecycle – What Happens Where?

When you declare a variable inside `main()`, it lives in the **stack**:

```
int a = 10; // stack variable
```

But if you use `malloc()` (later in dynamic memory), it lives in the **heap**.

Memory Region	Used For
Stack	Local variables, function calls
Heap	Dynamically allocated memory
Data Segment	Global/static variables
Code Segment	Actual program code

> 📌 Stack = fast, automatic
>
> **Heap = flexible, manual (but dangerous!)**

🧙 Keyword Wisdom – Reserved for the Compiler Gods

Let's go **behind** those 32 keywords:

- **Control Flow**: `if`, `else`, `while`, `for`, `switch`, `case`, `break`, `continue`, `goto`, `default`
- **Data Types**: `int`, `float`, `char`, `double`, `void`, `short`, `long`, `signed`, `unsigned`
- **Memory & Scope**: `static`, `auto`, `register`, `extern`, `const`, `sizeof`, `volatile`

- **Structures:** `struct`, `union`, `typedef`, `enum`
- **Functions:** `return`

> 🧠 These are non-negotiables. They're hardwired into the compiler — redefining them will break everything.

📥 `scanf()` – Your Program Starts Listening

Here's where input becomes interesting...

```
int age;
scanf("%d", &age);
```

💬 What's really happening:

- `%d` tells the function: "Expect an **int**"
- `&age` gives the **memory address** of `age` (where the input should go)
- Behind the scenes, `scanf()` pulls the input **character by character** from stdin and converts it to binary, storing it in `age`.

> ⚠️ If you forget the &, your program will crash or behave weirdly. Always give the address, not the value.

🧪 Format Specifiers Cheat Sheet

Specifier	Type
`%d`	Integer (`int`)
`%f`	Float (`float`)
`%c`	Character (`char`)
`%lf`	Double (`double`)
`%s`	String (`char[]`)

`%zu`	`size_t` (like in `sizeof`)

> 💡 You must match the specifier to the variable type — no automatic conversion here, unlike Python.

🪤 Input Pitfalls: Traps to Avoid

Mistake	Why It Breaks
`scanf("%d", a);`	No `&` – it needs address
`scanf("%f", &a);` where `a` is `int`	Mismatched types
Typing a float into `%d`	Truncates without warning
Typing text into `%d`	Causes undefined behavior (input buffer issue)

🛠️ Memory: Think Like a Debugger

Let's visualize what happens during this code:

```
int x = 10;
float y = 5.5;
char z = 'A';
```

In Memory:

Variable	Address	Type	Value	Bytes
`x`	`0x1000`	int	10	4
`y`	`0x1004`	float	5.5	4
`z`	`0x1008`	char	'A'	1

This is how the compiler lays things out — **efficiently, in order**.

🛡️ Why C Is Strict About Declaration

Unlike Python:

```
x = 10
```

...which dynamically assigns a type at runtime, C forces you to **declare types explicitly**:

```
int x = 10;
```

Why?

- ✅ Compile-time safety
- ✅ Memory optimization
- ✅ Performance (no runtime type-checking)

> 🧠 You give up convenience for speed and control. That's why C is still used in embedded, OS, and game engines.

🎓 Summary Table: Concepts vs What Actually Happens

Concept	C Code	Behind the Scenes
Declare int	`int x;`	Allocates 4 bytes on stack
Input	`scanf("%d", &x);`	Fills `x` using input buffer
Output	`printf("%d", x);`	Converts int to string & prints
Size	`sizeof(int)`	Returns memory used (in bytes)
Variable	`x`	Label for a memory location
Type	`int`, `float`, etc.	Determines how bits are stored/interpreted

📉 Top 30 Common Mistakes in Chapter 1 (Variables, Data Types & Input)

#	💥 Mistake	😵 Why It Happens	🛠️ Fix

1	`scanf("%d", a);`	Forgot the `&` (address-of)	Use `&a` for input: `scanf("%d", &a);`
2	Using `%d` for a float	Wrong format specifier	Use `%f` for `float`, `%lf` for `double`
3	Using `%f` for `double`	Float and double mismatch	Use `%lf` for `double`
4	Declaring variable after usage	Declaration must come first in C	Declare before using it
5	Using undeclared variable	Compiler doesn’t know the variable	Always declare before use
6	`int 123abc;`	Variable name can't start with a digit	Start names with a letter or underscore
7	`int a@b;`	Invalid characters in name	Only use letters, digits, and `_`
8	Missing semicolon	C requires `;` at end of statements	Always end with `;`
9	Forgetting `#include <stdio.h>`	`printf`, `scanf` undefined	Always include standard header
10	Printing char with `%d`	Wrong specifier	Use `%c` for `char`
11	Typing string into `%d` input	Can't convert text to integer	Ensure correct type input
12	Forgetting newline `\n`	Output looks jumbled	Add `\n` in `printf`
13	Confusing assignment (`=`) with comparison (`==`)	Syntax mix-up	Use `==` in conditions, `=` for assignment
14	Not initializing variables	May contain garbage values	Always assign before using
15	Initializing `char` with `"A"`	`"A"` is a string, not a char	Use `'A'` for single characters
16	Using `int a = 3.5;`	Type mismatch, loses decimal	Use `float` or `double`
17	Using `scanf("%d", &a)` and pressing Enter without	Program hangs waiting for input	Always give correct input as expected

	input		
18	`sizeof` returns unexpected value	Misunderstood type sizes	Use `sizeof(type)` or `sizeof var` correctly
19	Writing `int main()` without `return 0;`	Technically valid but poor practice	Always end `main()` with `return 0;`
20	Using multiple words in variable name with space	`int my var = 10;` → Error	Use underscore or camelCase: `my_var`
21	Using `float` but no decimals	`float f = 5;` → still valid, but confusing	Use `5.0f` to make it clear
22	Wrong order of variable declaration and use	Used before declaration	Always declare before using
23	Confusing `char c = "A";`	`"` is for string, not character	Use `'A'` for char
24	Using space in format specifier → `"% d"`	`% d` is invalid in C	No space inside specifier: `"%d"`
25	Mixing input/output types → `scanf("%f", &i)` where `i` is `int`	Causes wrong results or crash	Ensure format matches variable type
26	Variable shadowing in small scopes	Declaring `int a` again inside block	Avoid redeclaring same name in inner scopes
27	Thinking C will convert types like Python	C is not dynamic!	Always cast explicitly if needed
28	Not using `\n` when expecting new lines	Output gets bunched up	Remember to format your output
29	Forgetting to link `stdio.h` and writing `printf()`	Compiler error: undefined reference	Always add `#include <stdio.h>`
30	Using `int main{}` instead of `int main()`	C requires proper syntax	Always use `()` even if no parameters

---

🧠 Bonus Tips:

- ✅ Add comments like `// This line declares a variable` to explain.
- 🔍 When in doubt, use `gcc -Wall` to show all warnings.

- 🧪 Experiment with `sizeof()` to explore how C handles memory.
- 🛑 Never assume C "understands" what you mean — **you must be precise**.

🧩 Problem 1A: Area of Rectangle (Hardcoded)

📁 `problem1_a.c`

📌 Problem Summary:

Calculate area of a rectangle with fixed dimensions (length = 2, breadth = 5).

🔍 Concepts Tested:

- Integer variables
- Arithmetic operations ()
- Printing output with `%d`

🧑‍🏫 Explanation:

Here, values of `length` and `breadth` are directly written in the code (no user input). It's a great starter example to get a feel of how C calculates things and uses variables.

```
int length = 2;
int breadth = 5;
int area = length * breadth;
```

We use `%d` because `area` is an `int`.

⚠️ Common Mistakes:

- Forgetting to multiply: writing `area = length + breadth`
- Using `%f` instead of `%d`
- Forgetting `;` after each line

🧠 Bonus Tip:

You can also directly do:

```
printf("Area is %d", length*breadth);
```

📝 Output Snapshot:

```
The area of the rectangle is 10
The area of the rectangle is 10
```

🧩 Problem 1B: Area of Rectangle (User Input)

📁 `problem1_b.c`

🔍 Concepts Tested:

- `scanf()` with `%d`
- `&` address-of operator
- Dynamic input from user

🧑‍🏫 Explanation:

You now take `length` and `breadth` from user using:

```
scanf("%d", &length);
scanf("%d", &breadth);
```

The `&` is needed because `scanf()` needs to know *where* in memory to store the value.

⚠️ Common Mistakes:

- Forgetting `&` → `scanf("%d", length)` ❌
- Using `%f` instead of `%d`
- Not guiding user with prompts → always use `printf()` before asking for input

---

🧩 Problem 2A: Area of Circle

📁 `problem2_a.c`

🔍 Concepts Tested:

- Integer input, float output
- `%f` for printing float
- Math: π * r²

🧑‍🏫 Explanation:

The radius is taken as integer, and we use `3.14` as an approximation of π. Area = π × r²

```
printf("Area is %f\n", radius * radius * 3.14);
```

Although radius is `int`, the result becomes `float`.

⚠️ Mistake Trap:

- Forgetting `%f`
- Not including `3.14` as a float (should ideally use `3.14f` or `double`)
- Inputting radius as float but using `int` declaration

🧩 Problem 2B: Volume of Cylinder

📁 `problem2_b.c`

🔍 Concepts Tested:

- Multiple inputs
- Using float math with integers
- π * r² * h (Volume formula)

🧑‍🏫 Explanation:

We get `radius` and `height` as input and calculate:

```
volume = 3.14 * radius * radius * height;
```

We're again using `%f` for output.

💡 Pro Tip:

This is a good example to later try with `float radius` or even `double` for precision.

🧩 Problem 3: Celsius to Fahrenheit Converter

📁 `problem3.c`

🔍 Concepts Tested:

- Float input and output
- Basic formula: °F = °C × 1.8 + 32

👨‍🏫 Explanation:

```
float celsius;
scanf("%f", &celsius);
float fahrenheit = (celsius * 1.8) + 32;
```

Use `%f` for both input and output. Don’t forget the `&`.

🧩 Problem 4: Simple Interest Calculator

📁 `problem4.c`

🔍 Concepts Tested:

- Multiple float inputs
- Formula: (P × R × T) / 100
- Input prompts

👨‍🏫 Explanation:

Here's the full formula breakdown:

```
simple_interest = (principal * interest * years) / 100;
```

And remember to escape `%` in the string with `\%`:

```
printf("Enter rate in \%: ");
```

🧠 Bonus Tips:

- Try inputting with decimals like `12.5` to see how float precision works
- Switch `float` to `double` and test difference

🧠 Summary Table

Problem	Input Type	Output Type	Core Concept
1A	None (Hardcoded)	Integer	Variable, Arithmetic
1B	Integer	Integer	`scanf`, `printf`
2A	Integer	Float	$\pi * r^2$
2B	Integer	Float	$\pi * r^2 * h$
3	Float	Float	Celsius to Fahrenheit
4	Float	Float	Simple Interest Formula

🧠 MASTER SUMMARY TABLE — CHAPTER 1

🔢 Topic	🔍 Concept	✅ Syntax / Note	⚠️ Gotcha / Tip
1. Variable Declaration	Declaring variables before use	`int a;`	Must declare before using in C
2. Variable Initialization	Assigning value	`a = 5;` or `int a = 5;`	Uninitialized variables = garbage value

3. Data Types	Integer, Float, Char	`int` , `float` , `char`	Use correct format specifier with each
4. Input Function	Getting user input	`scanf("%d", &a);`	Don't forget `&` for address
5. Output Function	Displaying results	`printf("Value: %d", a);`	Use `%d` , `%f` , `%c` accordingly
6. Format Specifiers	For `printf()` and `scanf()`	`%d` , `%f` , `%c` , `%lf` , `%s`	Mismatched format = wrong output or crash
7. Comments	Explaining code	`// single` , `/* multi */`	Compiler ignores them
8. Variable Naming	Valid names	`int _hi;` `int funCar7;`	No special chars, can't start with number
9. sizeof Operator	Memory in bytes	`sizeof(int)`	Result type: `size_t`
10. Reserved Keywords	Built-in words	`int` , `char` , `return` , etc.	Can't be used as variable names

---

🗝️ Full List of 32 Reserved Keywords in C (for reference):

```
auto      break     case      char      const
continue  default   do        double    else
enum      extern    float     for       goto
if        int       long      register  return
short     signed    sizeof    static    struct
switch    typedef   union     unsigned  void
volatile  while
```

🎯 10 INTERESTING PRACTICE PROBLEMS

Let's go beyond rectangles and circles and enter ✨ **real-life + logic-driven problems** to make learning fun and applicable.

🧩 1. Odd-Even Detector

📌 Write a program to check if a number is even or odd using `%` operator.

🧠 Tip: Try it with both hardcoded and user input.

🧩 2. Swap Two Numbers Without Third Variable

📌 Use math logic (addition & subtraction) to swap two variables.

🤯 Mind-bender for logic practice!

🧩 3. Days to Weeks & Days Converter

📌 User inputs number of days → Convert to weeks + remaining days.

Ex: `45 days = 6 weeks and 3 days`

🧩 4. ASCII Value Finder

📌 Input a character → Print its ASCII value.

🎁 Bonus: Try printing all alphabets with their ASCII values in a loop (you'll love it in future chapters!)

🧩 5. Perimeter of Circle and Rectangle

📌 Ask user to choose shape, then calculate perimeter accordingly.

➕ Use formulas:

- Circle: `2πr`
- Rectangle: `2(l + b)`

🧩 6. Salary Calculator

📌 Input: Basic salary → Output: HRA = 20%, DA = 50%, Total salary.

🧠 Great real-life math usage of percentages.

🧩 7. Minutes to Hours and Minutes

📌 Convert given minutes (like 135) to hours and minutes.

⏱️ Use division and modulo.

🧩 8. Find Last Digit of a Number

📌 Input: 527 → Output: 7

🧠 Use `%` operator for extracting digits.

🧩 9. Find Area of Triangle using Heron's Formula

📌 Input three sides of triangle → Apply Heron's formula

🤓 Bonus: Calculate `s = (a+b+c)/2` then `Area = sqrt(s(s-a)(s-b)(s-c))`

🧩 10. Calculate Your Age in Days

📌 Input age in years → Convert to days, weeks, and months

⚠️ Ignore leap years (make it simple)