C++ Variables & Data Types Mastery Notes

1. What is a Variable?

> A variable in C++ is like a **labeled box** in your computer's memory that stores a value you can use or change.

```
int age = 25;
```

Think of it as:

> "Hey computer! Save a value `25` in a box called `age`, and I might change or use it later."

---

2. Data Types in C++

Different types of boxes store different kinds of stuff:

Data Type	Description	Example	Emoji Analogy
`int`	Integer numbers	`42`	Whole numbers
`float`	Decimal numbers (single precision)	`3.14f`	Approx values
`double`	Decimal (double precision)	`3.1415926`	More precise
`char`	Single character	`'A'`	One letter
`bool`	Boolean true/false	`true`	or
`string`	Sequence of characters	`"Hello"`	Text

---

3. Memory Size of Data Types ( )

Data type sizes **depend on system architecture** (32-bit vs 64-bit) and compiler.

Common Sizes in C++:

Data Type	32-bit Size	64-bit Size	Range (approx)
`char`	1 byte	1 byte	-128 to 127
`int`	4 bytes	4 bytes	-2,147,483,648 to 2,147,483,647
`float`	4 bytes	4 bytes	±3.4e±38 (7 digits precision)
`double`	8 bytes	8 bytes	±1.7e±308 (15 digits precision)
`bool`	1 byte	1 byte	true or false

Data Type	32-bit Size	64-bit Size	Range (approx)
`pointer`	4 bytes	8 bytes	Depends on architecture (address)

⛓️ **Note**: `sizeof()` operator helps to find actual size at runtime.

```
#include <iostream>
using namespace std;

int main() {
    cout << "Size of int: " << sizeof(int) << " bytes" << endl;
    cout << "Size of float: " << sizeof(float) << " bytes" << endl;
    cout << "Size of double: " << sizeof(double) << " bytes" << endl;
    cout << "Size of char: " << sizeof(char) << " byte" << endl;
    cout << "Size of bool: " << sizeof(bool) << " byte" << endl;
    return 0;
}
```

◇ 4. How Data is Stored in Memory 🧠

Memory is like a **row of lockers** 🗄️ with addresses, and variables are stored in these lockers.

📦 Memory Representation:

```
int x = 5;
```

🧠 Internally:

Address	Value
0x100	00000101 (binary)
0x101	00000000
0x102	00000000
0x103	00000000

(Stored in **little-endian** format on most machines)

⚙️ Endianness

- **Little Endian**: Least significant byte stored first.
- **Big Endian**: Most significant byte stored first.

🧃 **Analogy**: Think of numbers as juices in cartons:

- Little Endian pours from the smallest flavor.
- Big Endian pours from the biggest flavor first.

◇ 5. Table: Summary of Variable Storage 📝

Data Type	Size	Binary Example	Stored As (little-endian)	Description
`int`	4B	`5 → 00000101`	0x00, 0x00, 0x00, 0x05	Integer in 4 bytes
`char`	1B	`'A' → 01000001`	0x41	ASCII representation
`bool`	1B	`true → 00000001`	0x01	1 for true
`float`	4B	`3.14 → IEEE754`	0xC3, 0xF5, 0x48, 0x40	IEEE floating point
`double`	8B	`...`	8 bytes as per IEEE 754	More precision

◇ 6. Code Demo: Check Sizes ⌨️

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 10;
    float b = 3.14f;
    char c = 'A';
    double d = 3.14159;
    bool e = true;

    cout << "int a = " << a << " | size: " << sizeof(a) << " bytes" << endl;
    cout << "float b = " << b << " | size: " << sizeof(b) << " bytes" << endl;
    cout << "char c = " << c << " | size: " << sizeof(c) << " byte" << endl;
    cout << "double d = " << d << " | size: " << sizeof(d) << " bytes" << endl;
    cout << "bool e = " << e << " | size: " << sizeof(e) << " byte" << endl;

    return 0;
}
```

🧠 7. Bonus: How CPU Reads Memory? 🔍

Imagine:

📍 CPU asks: "What's in locker `0x100`?" 💾 RAM responds: "Here's 00000101 (that's 5!)" 🖥️ CPU processes it based on instruction.

✨ 8. Recap Cheat Sheet

Concept	Key Point
Variable	A named box to store data in memory
Data Type	Specifies type & size of data in variable
Size	Depends on architecture and compiler
Memory	Stored in binary, little-endian format
`sizeof()`	Used to find size at runtime

🚀 9. Practice Task

Try printing sizes and addresses:

```
#include <iostream>
using namespace std;

int main() {
    int x = 10;
    cout << "Value: " << x << ", Address: " << &x << endl;
    return 0;
}
```

🧠 C++ Data Type Memory Storage

📦 1. `int` – Integer

☑ How stored:

- Typically **4 bytes (32 bits)** on most systems.
- Stored in **binary** (2's complement for negatives).
- Stored as **little-endian** on most platforms (e.g., Intel CPUs).

🧪 Example:

```
int x = 5;
```

Binary Representation (32-bit):

```
5 → 00000000 00000000 00000000 00000101
```

Memory (Little Endian):

Address	Value
0x1000	0x05
0x1001	0x00
0x1002	0x00
0x1003	0x00

Think of it like a carton poured starting from the smallest end (LSB first).

---

2. `char` – Character

How stored:

- Exactly **1 byte (8 bits)**.
- Stores ASCII or Unicode character code.

Example:

```
char c = 'A';
```

ASCII Code:

```
'A' = 65 → 01000001
```

Memory:

Address	Value
0x1004	0x41

`'A'` is stored directly as its binary ASCII code.

---

3. `bool` – Boolean

How stored:

- Typically **1 byte (8 bits)**.
- Stored as:
  - `0x00` for `false`
  - `0x01` for `true`

Example:

```
bool b = true;
```

Memory:

Address	Value
0x1005	0x01

⚠ Only 1 bit is logically required, but stored in 1 byte for alignment.

🧠 4. `float` – Floating Point (IEEE 754, 32-bit)

☑ How stored:

- Stored in **4 bytes**.
- Format: IEEE 754 **single precision** (1 sign bit, 8 exponent bits, 23 mantissa bits)

🧪 Example:

```
float f = 3.14f;
```

Binary (approx):

```
3.14 → 01000000 01001000 11110111 11011110
```

Memory (Little Endian):

Address	Value (Hex)
0x1006	0xC3
0x1007	0xF5
0x1008	0x48
0x1009	0x40

🤯 CPU decodes the value using IEEE 754 math rules.

🧠🧠 5. `double` – Double Precision Float (IEEE 754, 64-bit)

☑ How stored:

- Stored in **8 bytes**
- Format: IEEE 754 **double precision** (1 sign bit, 11 exponent bits, 52 mantissa bits)

Example:

```
double d = 3.14;
```

Binary (approx):

```
3.14 → 0100000000001001000111101011100001010001111010111000010100011111
```

Memory (Little Endian):

Address	Value (Hex)
0x1010	0x1F
0x1011	0x85
0x1012	0xEB
0x1013	0x51
0x1014	0xB8
0x1015	0x1E
0x1016	0x09
0x1017	0x40

Summary Table

Data Type	Size (Bytes)	Format	Stored As Example (`3.14`)
`int`	4	Binary (signed)	05 00 00 00 (Little Endian)
`char`	1	ASCII/Unicode	0x41 for 'A'
`bool`	1	0 or 1	0x01 for true
`float`	4	IEEE 754	0xC3 0xF5 0x48 0x40
`double`	8	IEEE 754	8 bytes in Little Endian

Bonus Code to See Memory Bytes

```
#include <iostream>
using namespace std;

void printBytes(void* ptr, int size) {
    unsigned char* p = (unsigned char*) ptr;
```

```
    for (int i = 0; i < size; ++i) {
        printf("%02X ", p[i]);
    }
    cout << endl;
}

int main() {
    int a = 5;
    float b = 3.14f;
    double c = 3.14;
    char d = 'A';
    bool e = true;

    cout << "int: "; printBytes(&a, sizeof(a));
    cout << "float: "; printBytes(&b, sizeof(b));
    cout << "double: "; printBytes(&c, sizeof(c));
    cout << "char: "; printBytes(&d, sizeof(d));
    cout << "bool: "; printBytes(&e, sizeof(e));

    return 0;
}
```

🧠 IEEE 754 Format for `float` and `double`

🧠 `float` – 32-bit Floating Point

Part	Size (bits)	Description
Sign bit	1	0 for positive, 1 for negative
Exponent	8	Biased exponent (bias = 127)
Mantissa	23	Fractional part (leading 1 hidden)

🧪 Example: `float f = 3.14f;`

Step-by-step:

1. **Decimal to Binary**:
   - `3.14` in binary ≈ `11.0010001111011...`
2. **Normalized Form**:
   - `1.10010001111011 × 2¹`
3. **Sign bit**:
   - `3.14` is positive → `0`

4. **Exponent**:
    - Actual exponent = `1`
    - Biased exponent = `1 + 127 = 128`
    - Binary = `10000000`
5. **Mantissa (after leading 1)**:
    - `10010001111010111000011` (fits 23 bits)

☑ Final 32-bit Representation

```
0 10000000 10010001111010111000011
```

Split into:

Field	Bits
Sign	0
Exponent	10000000
Mantissa	10010001111010111000011

🤔 Value =

```
(-1)^0 × 1.10010001111010111000011 × 2^(128 - 127)
= +1.570... × 2¹ = ~3.14
```

🤯🤯 `double` – 64-bit Floating Point

Part	Size (bits)	Description
Sign bit	1	0 for positive, 1 for negative
Exponent	11	Biased exponent (bias = 1023)
Mantissa	52	Fractional part (leading 1 hidden)

🧨 Example: `double d = 3.14;`

1. **Decimal to Binary**:
    - `3.14` ≈ `11.0010001111011...`
2. **Normalized**:
    - `1.10010001111010111000010100011111... × 2¹`

3. **Sign bit** = `0`
4. **Exponent** =
    - Actual: `1`
    - Biased: `1 + 1023 = 1024`
    - Binary: `10000000000`
5. **Mantissa**:
    - `10010001111010111000010100011110101110000101000111111` (52 bits)

☑ Final 64-bit Representation

```
0 10000000000 1001000111101011100001010001111010111000010100011111
```

Split into:

Field	Bits
Sign	0
Exponent	10000000000
Mantissa	1001000111101011100001010001111010111000010100011111

🧠 Value =

```
(+1) × 1.100100011110101... × 2¹ ≈ 3.14
```

🎨 Visual Diagram

```
32-bit float layout
┌──────────┬──────────────────┬──────────────────────────────┐
│ Sign     │ Exponent (8)     │ Mantissa (23 bits)           │
├──────────┼──────────────────┼──────────────────────────────┤
│ 0        │ 10000000         │ 10010001111010111000011      │
└──────────┴──────────────────┴──────────────────────────────┘

64-bit double layout
┌──────────┬──────────────────────┬──────────────────────────────────────────────────────────┐
│ Sign     │ Exponent (11)        │ Mantissa (52 bits)                                       │
├──────────┼──────────────────────┼──────────────────────────────────────────────────────────┤
```

```
| 0      | 10000000000       | 10010001111010111000010100011110101110000 1...
|
└────────┴───────────────────┴──────────────────────────────────────────────────
┘
```

Code to See Float Bits

```cpp
#include <iostream>
#include <bitset>
using namespace std;

void printFloatBits(float num) {
    union {
        float input;
        int   output;
    } data;
    data.input = num;
    cout << "Float bits: " << bitset<32>(data.output) << endl;
}

void printDoubleBits(double num) {
    union {
        double input;
        long long output;
    } data;
    data.input = num;
    cout << "Double bits: " << bitset<64>(data.output) << endl;
}

int main() {
    float f = 3.14f;
    double d = 3.14;

    printFloatBits(f);
    printDoubleBits(d);
    return 0;
}
```

❓ Why does `bool` take 1 **byte (8 bits)** instead of 1 **bit**?

---

1. **Minimum Addressable Unit = 1 Byte**

Modern CPUs are byte-addressable, meaning:

> **Memory is accessed in units of 1 byte (8 bits)** — not bits.

So even if a `bool` only needs 1 bit (`0` or `1`), the smallest chunk of memory that the CPU can read/write is **8 bits**.

🧠 2. **Performance: CPU Optimization**

Accessing individual bits is **slow and complicated**. If multiple `bool` variables were packed tightly bit-by-bit:

- The CPU would need:
  - **Bit masking**
  - **Bit shifting**
- This is **slower and more error-prone** compared to accessing a full byte.

Thus, storing each `bool` in **1 byte** makes the compiler's job easier and programs faster.

💡 3. **Memory Alignment Requirements**

Most modern CPUs have **word alignment rules**. For example:

- 32-bit systems often align data in 4-byte blocks.
- 64-bit systems align in 8-byte blocks.

Storing data with odd alignments (like 1 bit or 2 bits) causes **padding** and **wasted memory** anyway.

🧪 Code Example

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Size of bool: " << sizeof(bool) << " byte(s)" << endl;
    cout << "Size of int: " << sizeof(int) << " byte(s)" << endl;
}
```

🖨️ Output:

```
Size of bool: 1 byte(s)
Size of int: 4 byte(s)
```

📊 Summary Table

Data Type	Logical Size	Actual Memory Used	Why Not Bit-Level?
`bool`	1 bit	1 byte (8 bits)	CPU can't access bits directly
`char`	1 byte	1 byte	Minimum addressable unit
`int`	4 bytes	4 bytes	Word-aligned

Want to Save Bits?

You **can** tightly pack `bool` values using **bit fields** or **bitsets**:

Bitset Example

```
#include <bitset>
#include <iostream>
using namespace std;

int main() {
    bitset<8> flags;
    flags[0] = 1;
    flags[1] = 0;
    flags[2] = 1;

    cout << "Bitset: " << flags << endl;
}
```

This way, 8 `bool` values take **only 1 byte** total!

TL;DR (Too Long; Didn't Read)

- `bool` uses 1 byte because CPU memory is **byte-addressable**.
- Accessing individual bits is **slow and complex**.
- Byte-level access improves **performance and simplicity**.
- Use `bitset` or `bit fields` if you **really want to save memory**.

C++ Variable Naming Conventions

General Rules (Syntax-level)

Rule	Example
Must start with a **letter** or `_`	`myVar`, `_temp`
Can contain letters, digits, `_`	`value123`, `count_1`

Rule	Example
No spaces or special characters	❌ `total value`
Case-sensitive	`data` ≠ `Data`
Cannot use **keywords**	❌ `int`, `while`

🧠 Naming Styles & Conventions

1. **camelCase** (💼 Preferred in C++ for variables)

```
int userAge;
float accountBalance;
```

> 🗣️ Common in functions and variable names

2. **snake_case** (🐍 Sometimes used in test or config code)

```
int total_count;
bool is_logged_in;
```

> 📦 Common in some C-style legacy or config-heavy codebases.

3. **PascalCase** (🎩 Usually used for Class Names)

```
class BankAccount;
class StudentInfo;
```

4. **ALL_CAPS** (🔒 Constants or Macros)

```
const int MAX_USERS = 100;
#define PI 3.14159
```

🔥 Best Practices for Variable Naming

Practice	👍 Do This	👎 Don't Do This
☑️ Be descriptive	`userScore`	`x`, `temp`, `data`

Practice	👍 Do This	👎 Don't Do This
☑ Use consistent style	`camelCase` or `snake_case`	Mixing styles `My_var1`
☑ Avoid abbreviations	`totalCount`	`tc`, `cnt`
☑ Avoid long names	`studentAge`	`ageOfTheStudentCurrentlyLoggedIn`
☑ Use nouns for variables	`carSpeed`	`runSpeedFunc`

🏏 Real Examples

```
// ☑ Good
int userId;
float interestRate;
bool isActive;

// ⃠ Bad
int u;       // unclear
float tRate; // abbreviation
bool X;      // single letter
```

🧠 Memory Hack – Name Like You Explain to a Friend!

> Pretend you're explaining the code to a friend out loud.

```
// What's easier to understand?
int d;           // ⃠ what is 'd'?
int downloadSpeed;  // ☑ oh! it's download speed!
```

🔧 Naming in Different Contexts

Context	Convention	Example
Variables	`camelCase`	`fileName`, `itemCount`
Constants	`ALL_CAPS`	`MAX_RETRIES`, `PI`
Classes	`PascalCase`	`UserManager`, `Logger`
Functions	`camelCase`	`calculateTotal()`
Private Members	`camelCase` + `_`	`filePath_`, `count_`

🚨 Common Mistakes to Avoid

- ❌ Using reserved words:

```
int class; // ❌ error!
```

- ❌ Using too short or single-letter names (except in loops like `i`, `j`)
- ❌ Including type in name (e.g., `intNumber`, `strName`) – it's unnecessary

---

☑ Summary Cheat Sheet

Rule	Example
Use `camelCase`	`userId`, `isRunning`
Use `PascalCase` for classes	`GameEngine`, `StudentInfo`
Use `ALL_CAPS` for constants	`MAX_USERS`, `PI`
Avoid cryptic/short names	❌ `x`, `a1`, `ptr2`
Keep names meaningful	☑ `fileCount`, `isVerified`
