C++ Prog Lang In A Nutshell Vol # 2

CPP NOTES BY Mr. BILRED

Your One-Day-Left Exam Sidekick

BILRED

C++ Prog Lang In A Nutshell Vol # 2

COMPILED BY BILAL AHMAD KHAN AKA MR. BILRED

CppNotesByBILRED

cout << "*KNOWLEDGE SHOULD BE SHARED\n*

ONLY WITH THE ONE\n

WHO KNOWS ITS TRUE WORTH\n

NOT EVERYONE DESERVES IT" << endl;

"Imagine Everybody Living With Peace
Just Imagine"

"We don't need more divisions; we need unity and love to make this world a BETTER PLACE"

Preface

If you're reading this hand-out, should I assume that

1. **Your exam is right around the corner,** and you're scrambling for help?
2. **You already know a little bit of C++**, enough to make sense of what's written here?
3. **Or maybe you're an extremely ambitious person?**

And yeah, I just wanted to make some stuff that **can** make you **"understand the concept"**. Not just use some HIGH-TECH words that may confuse ya…

Moreover, **this volume is the CONTINUED VERSION of VOL 1**. So if you need to understand the previous stuff, just open the **VOL 1** first.

I can't say these notes will make you a pro overnight, but maybe they might just help you get a quick know-how when you need it most. And yeah, this **DOES NOT COVER EVERYTHING, CONSIDER IT SIDE-KICK, NOT COMBO.**

I used AI to assist in compiling these notes (shout out to tech advancements). And hey, Believe it or not, the idea sparked when someone asked me for C++ notes. That tinkled my **CppNotesByBILRED** sense, and I seized the opportunity to create this resource (maybe maybe maybe). **<that story's about vol 1/>**

Actually, we have mid-term exam tomorrow, and I got the idea, why not make VOL 2 now. This may help me and the world too. Thank me later ;)

www.github.com/bilalahmadkhankhattak/cppnotes

Fun fact: This hand-out was created right before a mid-term

By The Way,

The real challenge today is no longer finding knowledge but discerning what is meaningful, what is true, and what is worth holding on to in a sea of distractions. Knowledge, after all, is not about how much you can access but about how wisely you can use it to make a difference, for yourself and for the world around you…

C++ Prog Lang In A Nutshell VOL # 2

CppNotesByBILRED

Compiled By Bilal Ahmad Khan AKA Mr. BILRED

07th April 2025

PDF Price: Prayers only /-

Table of Contents

Page left blank intentionally

C++ Prog Lang In A Nutshell Vol # 2

Typecasting

Typecasting is the process of **converting** a variable from one data type to another. In simple terms, it's like changing the type of a variable so it can work in a specific way or with certain operations. In C++, typecasting can be done in two ways:

Implicit Typecasting (Automatic):

- This happens **automatically** when you assign a smaller data type to a larger data type.
- For example, if you assign an int to a float, C++ will automatically convert the int to float without any special instructions.
- Example:

```
int num = 5;
float numFloat = num;  // Implicit conversion from int to float
```

Take a look at this one;

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

int main() {
    int a = 15;
    char b = 'x'; // ASCII value of 'x' is 120
    a = a + b; // b is implicitly converted to int
    double c = a + 2.5; // a is implicitly converted to double
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    return 0;
}
```

Output:
a = 135
b = x
c = 137.5

Explicit Typecasting (Manual):

- This is when you tell the compiler **what** type to convert a variable into.
- You have to use **typecasting operators** to manually convert one type into another.

```
float num = 5.7;
int numInt = (int)num;  // Explicit conversion from float to int
```

Note: In this case, the `float` value `5.7` will be converted to `5` (the decimal part will be **lost**).

Why is Typecasting Used?

- **Compatibility**: To make sure that different types of data can work together, like when mixing int and float in calculations.
- **Memory Efficiency**: Sometimes you want to save memory by converting larger data types to smaller ones.

Functions in C++

What is a Function?

A function is like a small program inside your big program. It does a specific job. For example, you might have a function to add two numbers, or to print a message. Functions help you avoid repeating the same code.

Syntax of Functions**:**

```
return_type function_name(parameters) {
    // body of the function
}
```

- **return_type:** This is the type of data the function will give back (e.g., int for numbers, void if no result is returned).
- **function_name:** This is the name of your function.
- **parameters**: These are the values you pass to the function. They help the function do its job.

Example**:**

```
void greet() {
    cout << "Peace Be Upon You, Bilal bhai!" << endl;  // This function prints
a greeting
}
```

- Here, `void` means the function does not return any value. It just prints a message.

User-Defined Functions (UDFs)

A **user-defined function** is a function that you, the programmer, create to perform a specific task. It allows you to group related code into a single block and reuse it in your program. These functions help in organizing code and making it more readable and manageable.

```
<return_type> function_name(<parameters>) {
    // Function body
    // Code to perform the task
    return <value>;  // if the function has a return type
}
```

- **<return_type>**: The type of value the function will return. If it doesn't return anything, you use void.
- **function_name**: The name of the function (e.g., addNumbers, printMessage).
- **<parameters>**: The values you pass into the function (optional).

Example of a User-Defined Function:

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

// Function to add two numbers
int addNumbers(int a, int b) {
    return a + b;  // Return the sum of a and b
}

int main() {
    int result = addNumbers(5, 7);  // Calling the user-defined function
    cout << "The sum is: " << result << endl;
    return 0;
}
```

In this example:

- The function addNumbers is user-defined.
- It takes two int parameters (a and b), adds them together, and returns the result.

Built-in Functions

Built-in functions are predefined functions that are provided by the programming language (like C++ or Python) or libraries. You don't need to define them yourself because they are already available for use. They perform common tasks like input/output, math operations, string manipulations, etc.

Examples of Built-in Functions in C++:

1. **`cout`**: Used to display output.
2. **`cin`**: Used to take input from the user.
3. **`sqrt()`**: A math function to calculate the square root of a number (from the `cmath` library).
4. **`strlen()`**: A function from the `cstring` library that returns the length of a string.

Moreover, Learn these two terms (for now)

- **Function Overloading**: You can have multiple functions with the same name, but their parameters should be different. The computer will know which one to use based on the number and types of the parameters.

- **Recursion**: A function that calls itself. This is useful for solving problems that can be broken down into smaller problems (like calculating a factorial).

Pass by Value vs. Pass by Reference

Pass by Value:

- When you pass an argument to a function by value, the function gets a copy of the value. If the function changes the value, it won't affect the original value.
  wait, I know "upar se gai".

Imagine you have a paper with your name on it.
You take a **photocopy** and give that to your friend.

Your friend writes something else on the **copy**, but your **original paper doesn't change**.

Same thing happens in **pass by value**.

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

void changeValue(int x) {
    x = 10;
}

int main() {
    int num = 5;
    changeValue(num);
```

```
    cout << num;  // Output will be 5
    return 0;
}
```

Why is Output 5?

- num was 5.
- changeValue(num) sends a **copy** of num (not the real one).
- Inside the function, the copy becomes 10.
- But the **real num is still 5**.

Pass by Reference:

- When you pass an argument by reference, the function gets the memory address of the original variable, so any changes made inside the function will affect the original value.

Real-Life Example:

You give your friend your **real notebook** (not a copy).
Now if your friend writes or erases something, it happens in your **original notebook**.

This is how **pass by reference** works.

```
#include<iostream>

using namespace std;

void change(int &x) {

    x = 10;

}


int main() {

    int num = 5;

    change(num);

    cout << num;  // Output: 10

}
```

NOTICE THAT “&” BEFORE “x”…….!

I know, I know, this stuff is a lil bit confusing but listen, This x is nothing new, its just another name for “num”. Both are pointing to the same memory address.
For Instance, **see this program**

```cpp
#include<iostream>

using namespace std;

void change(int &x) {
    cout << "Inside function: Address of x = " << &x << endl;
    x = 10;
}

int main() {
    int num = 5;
    cout << "In main: Address of num = " << &num << endl;
    change(num);
    cout << "num = " << num << endl;
}
```

Output:
In main: Address of num = 0x7ffffcbfc

Inside function: Address of x = 0x7ffffcbfc

num = 10

In Asaan Alfaaz:
“Jab hum &x (reference) dete hain, function **asli variable ke memory address** pe kaam karta hai. Isliye changes **directly original variable** ko effect karte hain.”

In English:
When we pass &x (a reference), the function works on the **actual memory address** of the original variable.
That’s why any changes inside the function **directly affect the original variable**.

When to Use Which?

- **Pass by Value**: When you don’t want the function to change the original value.
- **Pass by Reference**: When you want the function to change the original value, or if you're passing large objects (to avoid copying them).

Searching Algorithms

Obviously, I couldn’t cover every algorithm, but for now, some of em are below…

UNDERSTANDING: Searching algorithms in C++ are methods used to find a specific element in a collection. The most common ones (or I can say, easy) are Linear Search and Binary Search. Linear Search goes through each element sequentially, while Binary Search works on sorted collections by repeatedly dividing the search range in half. Both are basic algorithms used in various scenarios to locate elements efficiently.

Linear Search:

- This is the simplest way to search. It checks each item in a list one by one until it finds the item you're looking for

```
int linearSearch(int arr[], int size, int key) { // this func takes three
parameters
    for(int i = 0; i < size; i++) {
        if(arr[i] == key)
            return i; // Element found
    }
    return -1; // Not found
}
```

This is slow for large stuff because it checks each item.

Binary Search:

- This is faster than linear search **but works only on sorted lists**. It starts by checking the middle item, then divides the list into two halves and repeats the process.

```
// Binary Search cuts the array in half each time to find the target.
// Only works on sorted arrays.
// Faster than Linear Search.

// Binary search is a super-efficient algorithm used to find the position
// of a target element in a sorted array. Unlike linear search,
// which examines every element one by one, binary search divides the
search range in half with each step.

// Function to perform Binary Search
int binarySearch(int arr[], int size, int key){
    int low = 0, high = size -1; // Set initial search boundaries

    while(low <= high) {  // Searches if whole the range is valid...
        int mid = (low + high ) / 2; // find the middle index

        if (arr[mid] == key) {  // if element is found at mid, return index
            return mid;
        } else if (arr[mid] < key){
            low = mid + 1; // this one searches right half
        }
```

```
        else {
            high = mid - 1; // searches left half
        }
    }
    return -1; // not found....
```

Visit Github Repo for .cpp: https://github.com/BilalAhmadKhanKhattak/CppNotes

Sorting Algorithms

Bubble Sort

Bubble Sort is a simple way to sort things (like numbers) by **repeatedly swapping** them if they are in the **wrong order**.

Bubble Sort is an easy way to arrange data in either **ascending** (small to large) or **descending** (large to small) order. It's called **Bubble Sort** because, as the algorithm moves through the array, some numbers **"bubble" to the end** with each pass. If sorting in **ascending order**, the **larger numbers** move toward the end. If sorting in **descending order**, the **smaller numbers** move toward the end. In this section, we'll look at how Bubble Sort works to sort an array in **ascending order**.

```
// Bubble Sort:
// a sorting techinque that compares each pair of adjacent elements
// and swaps them if they are in the wrong order. It repeats this process
until the list is sorted.

// Disadvantage: It's not very efficient for large lists because it keeps
comparing even if the list is already sorted.

#include <iostream>
using namespace std;

// Function to perform Bubble Sort on the array
void bubbleSort(int numbers[], int size) { // takes two parameters

    // Outer loop: runs for each pass through the array
    for (int pass = 0; pass < size - 1; pass++) {

        // Inner loop: compare adjacent elements
        for (int i = 0; i < size - pass - 1; i++) {

            // If the current element is greater than the next element, swap
them
            if (numbers[i] > numbers[i + 1]) {
                int temp = numbers[i];       // Store current element in temp
```

```
                numbers[i] = numbers[i + 1];   // Replace current element with
next element
                numbers[i + 1] = temp;         // Put the stored value into
the next element's place
            // the above stuff can be understood by the example,
            // if u have two cups, one of tea, and the other of coffee, and u
want to replace the cups
            // then you'll temporarily need a third cup
            }
        }
    }
}

int main() {
    // Create an array of numbers to sort
    int numbers[] = { 5, 2, 8, 1, 4 };
    // Calculate the size of the array
    int size = sizeof(numbers) / sizeof(numbers[0]);

    // Print the original array
    cout << "Original array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    // Sort the array using Bubble Sort
    bubbleSort(numbers, size);

    // Print the sorted array
    cout << "Sorted array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    return 0;
}

// BILRED BHAI ;)
```

Now you SHOULD have some questions! The explanation may get a lil bit longer but look;

1. **bubbleSort function**:

Outer loop (pass): This loop controls how many times the algorithm goes through the array (aka **passes**). The algorithm runs **size - 1** times because, after each pass, the largest number

bubbles up to its correct position. After **size - 1** passes, the array is sorted, and no more swaps are needed.

Example: If your array has 5 elements, the outer loop will run 4 times (size - 1 = 5 - 1 = 4).

Inner loop (i): This loop compares adjacent elements in the array.

size - pass - 1: This ensures that with each pass, the inner loop compares **fewer elements** since the largest element from the previous pass is already placed at the end of the array.

The - pass part makes sure that with each pass, the algorithm avoids comparing the last few elements that are already in their correct position

Example to Understand:

Suppose we have an array with 5 elements:
[5, 2, 8, 1, 4]

First pass (pass = 0):

- **Inner loop runs 4 times** (size - pass - 1 = 5 - 0 - 1 = 4):
  - Compare 5 and 2 → Swap → [2, 5, 8, 1, 4]
  - Compare 5 and 8 → No swap → [2, 5, 8, 1, 4]
  - Compare 8 and 1 → Swap → [2, 5, 1, 8, 4]
  - Compare 8 and 4 → Swap → [2, 5, 1, 4, 8]

Now, 8 is in its final position.

Second pass (pass = 1):

- **Inner loop runs 3 times** (size - pass - 1 = 5 - 1 - 1 = 3):
  - Compare 2 and 5 → No swap → [2, 5, 1, 4, 8]
  - Compare 5 and 1 → Swap → [2, 1, 5, 4, 8]
  - Compare 5 and 4 → Swap → [2, 1, 4, 5, 8]

Now, 5 is in its final position.

Third pass (pass = 2):

- **Inner loop runs 2 times** (size - pass - 1 = 5 - 2 - 1 = 2):
  - Compare 2 and 1 → Swap → [1, 2, 4, 5, 8]
  - Compare 2 and 4 → No swap → [1, 2, 4, 5, 8]

Now, 2 is in its final position.

Fourth pass (pass = 3):

- **Inner loop runs 1 time** (size - pass - 1 = 5 - 3 - 1 = 1):
  - Compare 1 and 2 → No swap → [1, 2, 4, 5, 8]

At this point, the array is completely sorted.

Why This Works:

Outer loop (pass): Runs **size - 1** times.

Inner loop (i): Compares **adjacent elements** and reduces the range of comparison as more elements get sorted.

After each pass, the largest element gets moved to its correct position at the end of the array.

2. **main function**:

- Creates an array numbers and calculates its size.
- Prints the original array.
- Calls the bubbleSort function to sort the array.
- Prints the sorted array.

But wait, I had this question too,

how **int size = sizeof(numbers) / sizeof(numbers[0]);** Works?

This line is **super common** in C++ and **very useful** — let me explain it in **easy language**

```
int size = sizeof(numbers) / sizeof(numbers[0]);
```

What's going on?

- sizeof(numbers)
  Gives the **total size (in bytes)** of the whole array.
- sizeof(numbers[0])
  Gives the **size of just one element** of the array (e.g., int = 4 bytes usually).

Example:

```
int numbers[] = {5, 2, 8, 1, 4};
```

This array has **5 elements**.

- sizeof(numbers) → 5 elements × 4 bytes = **20**
- sizeof(numbers[0]) → size of 1 int = **4**

```
int size = 20 / 4 = 5
```

Boom — you get the number of elements! (KUCH SAMJH I? Nahi? Read it again)

Simple Summary:

This trick helps you find **how many elements** are in an array, without counting them manually.

Selection Sort

Selection sort is a simple way to sort an array.
It works like this:

"Find the smallest number and put it at the start. Then find the next smallest and put it in the next spot. Repeat until the whole array is sorted."

How it works:

1. Start from the first element.
2. Look through the rest of the array to find the smallest value.
3. Swap it with the current element.
4. Move to the next position and repeat until the end.

Asaan alfaaz:

Imagine apkay paas 5 cards hain:
[29, 10, 14, 37, 13]

1. **Pehla round**:
   - 29 se compare karo baaki sab ko
   - 10 is smaller → so 29 & 10 swap → [10, 29, 14, 37, 13]
2. **Dusra round**:
   - 29 se compare karo 14, 37, 13
   - 13 is smallest → 29 & 13 swap → [10, 13, 14, 37, 29]
3. Repeat...

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

int main() {
    int numbers[] = {23, 4232, 5435, 23, 1};
```

```
    int size = sizeof(numbers)/sizeof(numbers[0]); // this line calculates the
total number of elements in the array

    // Outer looop; goes through each position in the array
    for (int current = 0; current < size - 1; current++) {
        int smallestIndex = current; // just for now, consider the current
element as the smallest

        // Inner Loop - finds the index of the smallest number in the rest of
the array
        for (int next = current + 1; next < size; next++) {
            if (numbers[next] < numbers[smallestIndex]) {
                smallestIndex = next;
            }
        }

        swap(numbers[current], numbers[smallestIndex]);
    }

    cout << "sorted Array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
}
```

Relax, Let me explain:

First, it stores some numbers in an array. Then it uses two loops: one outer loop and one inner loop. The outer loop goes through each number one by one and assumes that number is the smallest. The inner loop checks the remaining numbers in the array to see if there's any number smaller than the current one. If it finds a smaller number, it saves its position. After finishing the inner loop, it swaps the current number with the smallest number it found. This keeps repeating until the whole array is sorted. In the end, it prints the sorted array on the screen. This way, all numbers get arranged from smallest to largest (ascending order).

int numbers[] = {23, 4232, 5435, 23, 1};

This is an **array of integers**. We want to **sort** these numbers.

int size = sizeof(numbers)/sizeof(numbers[0]);

This line calculates the **total number of elements** in the array.

for (int current = 0; current < size - 1; current++) {

This is the outer loop. It goes through each position of the array, except the last one, because the last one will already be in the correct place at the end.

int smallestIndex = current;

We **assume** that the current number is the smallest.

for (int next = current + 1; next < size; next++) {

This is the inner loop.

It checks the rest of the array (after the current index) to find if any number is smaller than the current one.

if (numbers[next] < numbers[smallestIndex]) {

smallestIndex = next;

}

If a **smaller number is found**, we update smallestIndex to remember its position.

swap(numbers[current], numbers[smallestIndex]);

We **swap** the current number with the smallest number we found in the rest of the array.

After each full pass of the outer loop, one number is fixed at its correct position in the sorted array.

cout << "sorted Array: ";

for (int i = 0; i < size; i++) {

cout << numbers[i] << " ";

}

This part prints the final **sorted array**.

Summary:

- You go to each position in the array.
- You look for the **smallest number** in the rest of the array.
- You **swap** it with the current position.
- You repeat this until the array is **sorted**.

Merge Sort (just concept)

Merge Sort is a popular and efficient sorting algorithm that follows the **Divide and Conquer** strategy. It breaks down the problem into smaller sub-problems, solves them, and then combines (or "merges") them to get the final sorted list.

How does Merge Sort work?

1. **Divide**: The array is repeatedly split in half until each sub-array contains only one element. A single element is always considered sorted.
2. **Conquer**: Each of the smaller arrays is merged back together in a sorted manner.
3. **Combine**: The merging process involves comparing the smallest elements from each sub-array and building the sorted array from these comparisons.

Cpp at a glance:

Data Types:

1. int – Whole numbers (e.g., 5)
2. float / double – Decimal numbers (e.g., 3.14)
3. char – Single characters (e.g., 'A')
4. bool – True or false values
5. string – Text (needs #include <string>

Basic I/O

cin >> variable; // Input from user

cout << "Text"; // Output to screen

Operators

1. +, -, *, / – Arithmetic
2. ==, !=, <, > – Comparison
3. &&, ||, ! – Logical (AND, OR, NOT)
4. =, +=, -= – Assignment

Conditional Statements:

if (condition) { }

else if (condition) { }

else { }

Use: Make decisions based on conditions

Loops:

```
for (int i = 0; i < 5; i++) { }
while (condition) { }
do { } while (condition);
```

Use: Repeat a block of code

Arrays:

```
int arr[5] = {1, 2, 3, 4, 5};
```

Use: Store multiple values of same type

Functions:

```
returnType functionName(parameters) {
  // code
  return value;
}
```

Use: Reuse code with inputs/outputs

Typecasting:

```
float x = (float)10 / 3;
```

Use: Convert one type to another manually

Pass by Value vs Reference:

```
void func(int x) { }      // Value - makes a copy
void func(int &x) { }     // Reference - modifies original
```

Useful Headers:

```
#include <iostream> – Input/output
#include <string> – For using strings
#include <cmath> – Math functions
```

Searching Algorithms:

Algorithm	How it Works	Use-case
Linear Search	Check each element of the array one by one until the target is found or the end of the array is reached.	Best for small or unsorted data, when you don't know where the element might be.
Binary Search	Split the sorted array into halves, repeatedly narrowing down the search area based on comparison.	Used on **sorted** arrays. Efficient for large datasets.

Sorting Algorithms:

Algorithm	How it Works	Use-case
Bubble Sort	Compare adjacent elements and swap if needed, repeat until no swaps are made.	Small datasets, when simplicity matters more than efficiency.
Selection Sort	Find the smallest element in the unsorted part of the list and swap it with the first unsorted element.	Simple but inefficient for larger datasets. Great for educational purposes.
Merge Sort	Recursively divide the array into halves, sort each half, and merge them back together.	Efficient for larger datasets and **divide-and-conquer** problems.

When to Choose Which?

- **Linear Search**: When data is small or unordered. Works perfectly when you don't know where the item is.
- **Binary Search**: Best choice when data is already sorted, especially when you're dealing with large datasets and need speed.
- **Bubble Sort**: If simplicity is your priority, and the dataset is small enough. Avoid for large datasets.
- **Selection Sort**: Great for learning and small lists, but avoid it for larger datasets.
- **Merge Sort**: Use it for larger datasets or when dealing with recursive problems. Its time complexity guarantees that it performs well even with big data.

Big O Notation Cheat Sheet

Notation	Description	Example	Use Case	Time Taken
O(1)	Constant Time	Array access	Direct access or simple operations	Takes the same time, no matter the size of input
O(n)	Linear Time	Linear Search	Traversing through all elements	Time increases linearly with input size
O(log n)	Logarithmic Time	Binary Search	Efficient searching in sorted data	Time increases very slowly with input size
O(n^2)	Quadratic Time	Bubble Sort	Nested loops (e.g., sorting algorithms)	Time increases rapidly with input size
O(n log n)	Linearithmic Time	Merge Sort	Efficient sorting algorithms	Time increases slowly, but faster than linear

A Final Thought

So, here we are at the end. If you've made it this far, you're not just someone looking to pass the exam, maybe you're someone who *cares enough to prepare*. Obviously, These notes can't turn you into a C++ master overnight. But they can give you that *one final boost* when time is short and pressure is high.

I just wanted to make a difference, many students just run for the GPA, and sure, that's fine. But I guess we should try some kind of **"exceptional things"** too. There might be a lot of reasons behind creating this hand-out, but one of the reasons I did this, because **I WANTED TO!** Don't ask me why, I just **wanted to, so I did it!**

Thanks To ALLAH ALMIGHTY, Who granted me strength to do this, Alhamdulillah!

And Yeah

Life's short… Go Do It MAN!

And Yeah, Live Today Dear, No One Has Promised You Tomorrow!

And Yeah, Who am I?
I am Bilal Ahmad Khan AKA Mr BILRED

GOOD LUCK!
TAKE CARE!

About The Author

Bilal Ahmad Khan, also known as Mr. BILRED, is a self-taught programmer with a deep passion for simplifying complex technologies. His expertise spans across C++, Python, and GoLang—along with interests in malware analysis and graphic design.Bilal's journey into coding began as early as 7th(or 8th) grade with C++ (a spark he credits to his brother). Even as a Pre-Medical student in FSc, he boldly pursued Python programming—because when ambition calls, you answer.He's also the founder of the clothing brand ZeeJayByBilred, reflecting not just his creativity but his desire to build something meaningful and personal.His story is still unfolding, but we'll wrap it here with one of his own quotes:

"Knowledge should be shared—only with the one who knows its true worth. Not everyone deserves it."

BILRED