Random Numbers in C++

1. Introduction

Today, we’re going to learn how to generate random numbers in C++. Whether you're building a game, simulating data, or shuffling a playlist — randomness is essential.

We use pseudorandom number generators (PRNGs) — algorithms that produce sequences of numbers that appear random.

In modern C++, we have a powerful and flexible library for this: the `<random>` header.

2. The Problem with rand() and srand()

You may have seen this classic approach:

```
#include <cstdlib>
#include <ctime>

int main() {
    srand(time(0)); // Seed once
    int random_num = rand() % 100; // 0 to 99
    return 0;
}
```

⚠ But this method has serious drawbacks:

2. The Problem with rand() and srand()

ISSUE	EXPLANATION
Poor randomness quality	`rand()` often uses a simple linear congruential generator — predictable patterns
Limited range	`RAND_MAX` is only guaranteed to be ≥ 32767
Non-uniform distribution	Using `%` introduces bias (especially with small ranges)
Global state	Affects all code in the program
Not thread-safe	Unsafe in multithreaded programs

✅ Conclusion: Avoid `rand()` in modern C++. Use `<random>` instead.

3. The Modern Way: `<random>` Header

C++11 introduced the <random> library — a complete toolkit for generating high-quality pseudorandom numbers.

It separates two key components:

✅ 1. Engine — Generates raw random numbers
✅ 2. Distribution — Shapes numbers into a desired form (e.g., uniform, normal)

This separation gives us flexibility, quality, and control.

4. Common Random Engines

An **engine** is an algorithm that produces a sequence of pseudorandom bits.

ENGINE	DESCRIPTION
std::default_random_engine	Default choice — implementation-defined (often mt19937)
std::mt19937	Mersenne Twister — excellent quality, fast, widely used
std::linear_congruential_engine	Simpler, faster, lower quality
std::random_device	True hardware randomness(if available) — used for seeding

🔧 Tip: Use std::mt19937 for most purposes.

5. Common Distributions

A distribution transforms engine output into useful values.

DISTRIBUTION	PURPOSE
std::uniform_int_distribution<T>	Random integers in a range
std::uniform_real_distribution<T>	Random floating-point numbers
std::normal_distribution<T>	Bell curve (Gaussian) distribution
std::bernoulli_distribution	true/falsewith given probability

6. Example: Generate a Random Integer (1 to 100)

```
#include <iostream>
#include <random>

int main() {
    // Step 1: Create a random device for seeding
    std::random_device rd;

    // Step 2: Initialize the engine
    std::mt19937 gen(rd());

    // Step 3: Define the distribution
    std::uniform_int_distribution<int> dist(1, 100);

    // Step 4: Generate random numbers
    for (int i = 0; i < 5; ++i) {
        int num = dist(gen);
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

7. Seeding: Why It Matters

The seed determines the starting point of the random sequence.

- Same seed → same sequence (good for reproducibility).
- Different seed → different sequence (good for real randomness).

Best Practice: Use `std::random_device`

```
std::random_device rd;  // True random seed
                        // (hardware)
std::mt19937 gen(rd()); // Seed the engine
```

For testing, you can use a fixed seed:

```
std::mt19937 gen(12345); // Reproducible results
```

8. Floating-Point Random Numbers

Want a random double between 0.0 and 1.0?

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<double> dist(0.0,
1.0);

double value = dist(gen);
std::cout << value << std::endl; // e.g., 0.3742
```

Useful for probabilities, simulations, etc.

9. Reuse Engines, Not Recreate Them

✅ Good: Create engine once, reuse it:

```
std::mt19937 gen(std::random_device{}());
// Reuse in loops or functions
for (int i = 0; i < 100; ++i) {
    std::uniform_int_distribution<int> d(1, 6);
    std::cout << d(gen) << " "; // Roll a die
}
```

❌ Bad: Creating new engine every time:

```
// DON'T DO THIS!
for (int i = 0; i < 100; ++i) {
    std::mt19937 bad_gen(std::random_device{}());
    // Each will likely have same seed due to clock resolution!
}
```

⚠ Risk: Poor randomness if seeded too frequently.

10. Shuffling Containers: std::shuffle

```
#include <algorithm>
#include <vector>
#include <random>

std::vector<int> v = {1, 2, 3, 4, 5};
std::random_device rd;
std::mt19937 gen(rd());
std::shuffle(v.begin(), v.end(), gen);
// Now v is randomly ordered
for (int x : v) std::cout << x << " ";
```

Great for card games, quizzes, playlists.

Lecture: Arrays and Pointers in C++

1. Introduction

Today, we'll explore two of the most fundamental and powerful concepts in C++: arrays and pointers.

These features give C++ its efficiency and flexibility — but they can also be confusing at first.

By the end of this lecture, you will understand:

- What arrays and pointers are
- How they are related
- The difference between arrays and pointers
- Common operations and pitfalls
- Modern alternatives (like std::array and std::vector)

2. Arrays in C++

An array is a contiguous block of memory that stores multiple elements of the same type.

Declaration:

```
int numbers[5]; // Array of 5 integers
```

Initialization:

```
int arr[5] = {1, 2, 3, 4, 5};
double values[] = {1.1, 2.2, 3.3};
// Size deduced automatically
```

Accessing Elements:

```
arr[0] = 10;     // First element
arr[2] = 30;     // Third element
```

⚠ No bounds checking!

Accessing arr[10] on a size-5 array leads to **undefined behavior.**

3. Memory Layout of Arrays

Arrays are stored contiguously in memory:

```
int arr[3] = {10, 20, 30};
```

Memory layout:

```
Address:   ... | 0x1000 | 0x1004 | 0x1008 | ...
Value:         |   10   |   20   |   30   |
```

Each int takes 4 bytes (typically), so elements are spaced evenly.

4. Pointers: What Is a Pointer?

A **pointer** is a variable that stores the **memory address** of another variable.

Declaration:

```
int* ptr;       // Pointer to an integer
double* dptr;   // Pointer to a double
```

Assigning an Address:

```
int x = 42;
int* ptr = &x; // 'ptr' holds the address of 'x’
```

Dereferencing:

```
cout << *ptr;  // Outputs 42 — value at the address
*ptr = 100;    // Changes 'x' to 100
```

🔑 Key operators:

& — address-of operator

* — dereference operator

5. The Connection Between Arrays and Pointers

Here's where things get interesting:
👉 An array name is implicitly convertible to a pointer to its first element.

```
int arr[5] = {1, 2, 3, 4, 5};
int* ptr = arr; // Same as &arr[0]
```

Now both `arr[i]` and `*(ptr + i)` give the same result.

This is known as **pointer arithmetic**.

6. Pointer Arithmetic

You can perform arithmetic on pointers:

```
int arr[3] = {10, 20, 30};
int* p = arr;
cout << *p;      // 10
cout << *(p+1); // 20
cout << *(p+2); // 30
```

Rules:

- p + 1 → moves forward by sizeof(type) bytes
- For int*, p + 1 adds 4 bytes (if int is 4 bytes)
- Works for any type: double*, char*, etc.

✅ This is why arr[i] is equivalent to *(arr + i)

✅ And why i[arr] is valid (yes, really!): same as *(i + arr)

7. Arrays vs. Pointers: Key Differences

FEATURE	ARRAY	POINTER
Type	Fixed-size block	Variable storing an address
Size	Known at compile time (sizeof(arr) works)	sizeof(ptr) gives pointer size (e.g., 8 bytes)
Reassignment	❌ Cannot change `arr = ..`	✅ Can reassign: `ptr = &x;`
Memory	Stores actual data	Stores address of data
Decay	❗ Decays to pointer when passed to function	Already a pointer

Example:

```
void func(int arr[]) { // Actually: void func(int* arr)
    cout << sizeof(arr); // Prints 8 (size of pointer), not size of array!
}
```

⚠ When an array is passed to a function, it decays into a pointer — you lose size information!

8. Passing Arrays to Functions

Because arrays decay to pointers, always pass size separately:

```
void printArray(const int* arr, int size) {
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Usage:
int data[] = {1, 2, 3, 4, 5};
printArray(data, 5);
```

9. Dynamic Arrays with `new` and `delete`

Sometimes you don’t know the size at compile time.

Use dynamic allocation:

```
int n;
cout << "Enter size: ";
cin >> n;
int* dynArr = new int[n]; // Allocate on heap
for (int i = 0; i < n; ++i) {
    dynArr[i] = i * i;
}
// Use the array...
delete[] dynArr; // Must use delete[] for arrays!
```

9. Dynamic Arrays with `new` and `delete`

⚠ Rules:

- Use `new[]` → **must use** `delete[]`
- Use `new` → **must use** `delete`
- Never forget `delete[]` → causes memory leaks

10. Common Pitfalls

❌ Buffer overflow:

```
int arr[5];
arr[10] = 100; // Undefined behavior!
```

❌ Dangling pointer:

```
int* ptr = new int(42);
delete ptr;
*ptr = 10; // Crash! Pointer points to freed memory
```

❌ Memory leak:

```
int* arr = new int[10];
delete arr; // ❌ Wrong! Should be delete[]
```

❌ Using delete instead of delete[]:

```
int* arr = new int[10];
delete arr; // ❌ Wrong! Should be delete[]
```

11. Modern C++ Alternatives

In modern C++, prefer safer, higher-level containers:

✅ std::array — Fixed-size array (stack)

```
#include <array>
std::array<int, 5> arr = {1, 2, 3, 4, 5};
cout << arr.size(); // 5
```

- No decay to pointer
- Bounds-checked with .at()
- Stack-allocated

11. Modern C++ Alternatives

✅ **std::vector — Dynamic array (heap)**

```
#include <vector>

std::vector<int> vec = {1, 2, 3};
vec.push_back(4);
cout << vec.size(); // 4
```

- Grows dynamically
- Automatic memory management
- Safe indexing
- Preferred over raw dynamic arrays

12. Summary Table

FEATURE	C-STYLE ARRAY	POINTER	std::array	std::vector
Size fixed?	Yes	N/A	Yes	No
Resizable?	No	Manual	No	Yes
Memory	Stack or static	Any	Stack	Heap
Safe access?	No	No	`.at()` yes	`.at()` yes
Size available?	Yes (in scope)	No	Yes	Yes
Recommended?	Legacy code	Low-level	✔ For fixed size	✔ For dynamic size

13. Best Practices

- ✓ Prefer std::vector and std::array over raw arrays
- ✓ Avoid new/delete unless necessary (use smart pointers)
- ✓ Always free dynamically allocated memory
- ✓ Use const for pointers that shouldn’t change const int* ptr
- ✓ Pass large arrays by pointer/reference, not by value

15. Exercises

- Write a function that finds the maximum value in an array using pointers.
- Dynamically allocate a 2D array and fill it with values.
- Convert a C-style array to std::vector.
- Implement a simple string copy function using pointers.
- Use std::array to store student grades and compute average.

C-Style Strings in C++

1. Introduction

Today, we’re going to talk about C-style strings — the original way of handling text in C and inherited by C++.

You’ve probably seen them:

```
char str[] = "Hello, world!";
```

These are not the modern std::string — they are arrays of characters, terminated by a special null character.

1. Introduction

While C++ provides the safer and more convenient std::string, understanding C-style strings is still important because:

- They appear in legacy code.
- They're used in system APIs (e.g., file operations, command-line arguments).
- They help you understand how strings work under the hood.

1. Introduction

By the end of this lecture, you'll know:

- What C-style strings are
- How to declare, initialize, and manipulate them
- Common functions from <cstring>
- Their limitations and dangers
- When (and when not) to use them

Let's begin!

2. What Is a C-Style String?

A C-style string is an array of characters (char) that ends with a null terminator: \0.

This null character has ASCII value 0 and marks the end of the string.

```
char greeting[] = "Hi";
```

In memory:

```
Index:      0    1    2
           ['H']['i']['\0']
```

✅ The compiler automatically adds \0 when you use a string literal.

❌ Never forget it — otherwise, functions won't know where the string ends!

3. Declaration and Initialization

There are several ways to create a C-string:

```
// 1. From string literal (size auto-deduced)
char name[] = "Alice";
// → size is 6: 'A','l','i','c','e','\0'
// 2. With explicit size
char city[20] = "Paris";
// → Only first 6 chars used; rest zero-initialized
// 3. Character by character (must add \0 manually!)
char msg[6] = {'H','e','l','l','o','\0'};
// 4. Using a pointer (points to string literal)
char* ptr = "Hello";
// ! String literals are read-only! Do NOT modify!
```

3. Declaration and Initialization

⚠ Important:

- Arrays decay to pointers when passed around.
- Size information is lost unless explicitly tracked.

4. Key Rules and Properties

RULE	EXPLANATION
Must be null-terminated	Functions like strlen rely on '\0'
Size = content length + 1	For the null terminator
No bounds checking	Writing past array → undefined behavior
String literals are const	String literals are const
Fixed size	Cannot grow dynamically

5. Common Operations and <cstring>

✅ strlen() — Get length (excluding \0)

```
char str[] = "C++";
cout << strlen(str); // Output: 3
```

✅ strcpy() — Copy a string

```
char dest[20];
strcpy(dest, "Copy me!");
// Now dest contains "Copy me!\0"
```

⚠ Dangerous: No size check → buffer overflow risk.

5. Common Operations and <cstring>

Use strncpy() instead:

```
strncpy(dest, source, sizeof(dest) - 1);
dest[sizeof(dest)-1] = '\0’;
// Ensure null termination
```

✅ strcat() — Concatenate strings

```
char path[50] = "/home/";
strcat(path, "user");
// Result: "/home/user\0"
```

Safer: strncat(dest, src, max_chars)

5. Common Operations and <cstring>

☑ strcmp() — Compare strings

Returns:

- 0 → equal
- < 0 → first string less than second
- > 0 → first string greater

```
if (strcmp(str1, str2) == 0) {
    cout << "Strings are equal\n";
}
```

Case-insensitive? Use strcasecmp() or stricmp() (non-standard).

6. Input and Output

Using cin and cout

```
char name[50];
cout << "Enter name: ";
cin >> name; // Stops at whitespace!
cout << "Hello, " << name << endl;
```

⚠ Problem: cin >> doesn’t handle spaces.

Safer input: cin.getline()

```
cin.getline(name, 50);
// Reads up to 49 chars + adds \0
```

Limits input to buffer size — prevents overflow.

7. Passing C-Strings to Functions

Since arrays decay to pointers, functions take char* or const char*:

```
void printString(const char* str) {
    cout << str << endl; // 'str' is a pointer
}

// Usage:
printString("Literal");
printString(name);
```

Use const if you don’t modify the string.
To pass size:

```
void safePrint(const char* str, int maxSize) { ... }
```

8. Common Pitfalls and Security Issues

❌ Buffer Overflow

```
char small[5] = "Hi";
strcpy(small, "This is too long!"); // 💥 Crash or exploit!
```

👉 One of the most common sources of security vulnerabilities!

❌ Missing Null Terminator

```
char buf[10];
// ... fill with data but forget \0
cout << buf; // May print garbage until \0 found
```

8. Common Pitfalls and Security Issues

❌ Modifying String Literals

```
char* p = "Hello";
p[0] = 'h'; // ❌ Undefined behavior!
```

❌ Using = to assign

```
char name[20];
name = "Bob";
// ❌ ERROR: can't assign to array!
```

Use strcpy(name, "Bob"); instead.

9. Pointers and C-Strings

You can use pointers to traverse strings:

```
char text[] = "Hello";
char* p = text;
while (*p != '\0') {
    cout << *p;
    p++;
}
// Output: Hello
```

Or use pointer arithmetic:

```
for (int i = 0; text[i] != '\0'; ++i) { ... }
```

Both are valid — C-strings and pointers go hand-in-hand.

10. C-Strings vs. std::string

FEATURE	C-STRING	std::string
Memory management	Manual	Automatic
Size	Fixed	Dynamic
Bounds checking	None	`.at()` provides it
Concatenation	`strcat()`	+operator
Comparison	`strcmp()`	==,<, etc.
Safety	Low (buffer overflows)	High
Ease of use	Harder	Easier
Performance	Very fast	Slight overhead

✅ Use std::string for new code.
✅ Use C-strings only when required (e.g., interfacing with C libraries).

11. When Are C-Strings Still Used?

Despite their dangers, C-strings are still relevant in:

- Operating system APIs (e.g., open(), exec())
- Embedded systems (limited resources)
- Legacy C code
- Command-line arguments: int main(int argc, char* argv[])
- File paths, environment variables

So you will encounter them — even in modern C++.

12. Best Practices

- ✓ Always ensure en`cin.getline()` ough space for \0
- ✓ Use instead of cin >> for full lines
- ✓ Prefer `strncpy, strncat, snprintf` over unsafe versions
- ✓ Use `const char*` for input parameters
- ✓ Avoid modifying string literals
- ✓ Prefer `std::string` in application-level code
- ✓ Validate input length before copying

13. Complete Example: Simple String Manipulation

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    char first[20], last[20], full[50];
    cout << "First name: ";
    cin.getline(first, 20);
    cout << "Last name: ";
    cin.getline(last, 20);
    strcpy(full, first);
    strcat(full, " ");
    strcat(full, last);
    cout << "Full name: " << full << endl;
    cout << "Length: " << strlen(full) << endl;
    return 0;
}
```

15. Exercises

- Write a function myStrlen(const char* str) that returns string length.
- Implement myStrcpy(char* dest, const char* src).
- Check if a C-string is a palindrome.
- Convert a string to uppercase using pointers.
- Safely concatenate three strings into a fixed buffer.

Conclusion

- C-style strings are a low-level, powerful, but dangerous way to handle text.
- They give you fine control — but require careful memory management and awareness of pitfalls like buffer overflows.
- In modern C++, always prefer std::string for general-purpose string andling.
- But never forget C-strings — they're part of C++'s heritage and still appear everywhere in system programming.

Now you understand both worlds: the old and the new.