C++ Cheatsheet: Advanced Level

Deepak Ranolia · Follow

13 min read · Dec 5, 2023

Welcome to the C++ Cheatsheet designed to enhance your proficiency in C++ programming. This is structured into four levels: Basic, Intermediate, Advanced, and Expert. Each level introduces you to progressively sophisticated C++ concepts and practices. The examples provided in each level are diverse, ensuring a comprehensive understanding of C++.

> *03 Advanced Level*
>
> *The Advanced Level introduces advanced features like smart pointers, exception handling, and lambda expressions. This level is tailored for those who want to deepen their C++ expertise and tackle more intricate programming challenges.*

Note: Please check out my C++ previous cheatsheet if you have not read it yet.

01 c++ Cheatsheet: Basic Level

02 c++ Cheatsheet: Intermediate Level

26 ***Multithreading:*** in C++ involves the execution of multiple threads concurrently to improve program performance and responsiveness. The `<thread>` header provides facilities for creating and managing threads. In the given code, a simple example of creating and running a thread is demonstrated.

```
#include <iostream>
```

```cpp
#include <thread>

// Function to be executed in a separate thread
void printNumbers() {
    for (int i = 0; i < 5; ++i) {
        std::cout << "Thread 1: " << i << std::endl;
    }
}

int main() {
    // Creating a thread and associating it with the function 'printNumbers'
    std::thread t1(printNumbers);

    // The main thread waits for 't1' to finish its execution
    t1.join();
    return 0;
}
```

Description:

1. Header Inclusions:

- `#include <iostream>` : Standard input/output stream.
- `#include <thread>` : Header for multithreading support.

2. Thread Function:

- `void printNumbers() { /* ... */ }` : A function (`printNumbers`) that will be executed in a separate thread. In this example, it prints numbers from 0 to 4.

3. Creating a Thread:

- `std::thread t1(printNumbers);` : This line creates a thread (`t1`) and associates it with the function `printNumbers` . The new thread runs concurrently with the main thread.

4. Thread Execution:

- `t1.join();` : The `join` function is called on the thread `t1` . This ensures that the main thread waits for `t1` to finish its execution before proceeding further. Without `join` , the main thread could finish before the thread `t1` completes.

5. Output:

- The program outputs the numbers printed by the separate thread, demonstrating concurrent execution.

Multithreading Features:

Concurrent Execution:

- Multithreading allows multiple threads to execute concurrently, improving the overall performance of the program.

Asynchronous Execution:

- Threads can execute asynchronously, allowing certain tasks to run independently, enhancing program responsiveness.

Parallelism:

- Multithreading enables parallelism, where multiple tasks are performed simultaneously, leveraging multicore processors.

Thread Management:

- C++ provides facilities for creating, joining, and managing threads, allowing developers to control the flow of execution.

Note: Multithreading is a powerful technique used to enhance the performance and responsiveness of applications, particularly in scenarios where tasks can be performed concurrently.

27 ***File handling:*** in C++ involves operations on files, such as reading from and writing to files. The `<iostream>` and `<fstream>` headers provide functionalities for handling files.

```
#include <iostream>
#include <fstream>

int main() {
    // Opening a file for writing
    std::ofstream file("example.txt");
```

```
    // Writing data to the file
    file << "Hello, C++ File Handling!";

    // Closing the file
    file.close();
    return 0;
}
```

Description:

1. Header Inclusions:

- `#include <iostream>` : Standard input/output stream.
- `#include <fstream>` : Header for file stream operations.

2. Opening a File:

- `std::ofstream file("example.txt");` : This line creates an object (`file`) of the `ofstream` class and opens a file named "example.txt" for writing. If the file does not exist, it will be created.

3. Writing to the File:

- `file << "Hello, C++ File Handling!";` : The `<<` operator is used to write the specified text to the file.

4. Closing the File:

- `file.close();` : This line closes the file. It's essential to close the file after writing to ensure that changes are saved.

5. Output:

- The program writes the text “Hello, C++ File Handling!” to the file “example.txt.”

File Handling Features:

Modes:

- C++ supports various file modes, such as `std::ifstream` for reading, `std::ofstream` for writing, and `std::fstream` for both reading and writing.

Error Handling:

- File stream objects can be checked for errors using the `.fail()` or `.good()` member functions.

Appending to Files:

- Files can be opened in append mode (`std::ofstream file("example.txt",`

`std::ios::app);`) to add content to an existing file.

Binary File Handling:

- C++ supports binary file handling for reading and writing binary data.

Note: File handling is crucial for data persistence and communication between programs. It allows programs to store and retrieve data from files, enabling data storage and sharing beyond the program's runtime.

28 ***Networking with sockets:*** in C++ involves using the socket API to create, bind, and manage network sockets. The given code shows a basic outline of setting up a server socket.

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <arpa/inet.h>

int main() {
    // Create a server socket
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    // Set up server address struct and bind the socket
    // ... (server-side setup not shown for brevity)

    // Listen for incoming connections
    listen(serverSocket, 5);

    // ... (server-side connection handling not shown for brevity)
    // Close the socket when done
```

```
    close(serverSocket);
    return 0;
}
```

Description:

1. Header Inclusions:

- `#include <iostream>` : Standard input/output stream.
- `#include <cstdlib>` : General utilities.
- `#include <cstring>` : String handling.
- `#include <unistd.h>` : POSIX operating system API.
- `#include <arpa/inet.h>` : Definitions for internet operations.

2. Create a Server Socket:

- `int serverSocket = socket(AF_INET, SOCK_STREAM, 0);` : This line creates a socket using the `socket` system call. It returns a file descriptor (`serverSocket`) representing the socket.

3. Set up Server Address and Bind:

- The code snippet doesn't show the detailed server-side setup and binding, which involves configuring a `sockaddr_in` structure with server

details and binding it to the socket.

4. Listen for Incoming Connections:

- `listen(serverSocket, 5);` : This line sets the server socket to listening mode, allowing it to accept incoming connections. The second parameter (5 in this case) specifies the maximum length of the queue for pending connections.

5. Server-Side Connection Handling:

- The actual server-side connection handling (accepting connections, reading/writing data) is not shown for brevity.

6. Close the Socket:

- `close(serverSocket);` : After the server has finished handling connections, it closes the server socket using the `close` system call.

Note:

- Detailed server-side setup, including address struct initialization, binding, and actual connection handling, is omitted for brevity.
- This example demonstrates the server socket's setup phase, and you would typically include error handling and additional logic in a

complete server application.

- Socket programming in C++ involves a combination of system calls and data structures for handling communication over a network. It is a low-level, powerful way to enable network communication between processes.

29 ***Custom memory allocators:*** in C++ allow developers to override the default `new` and `delete` operators, providing control over memory allocation and deallocation.

```
#include <iostream>
#include <cstdlib>

// Custom memory allocation function (override new operator)
void* operator new(size_t size) {
    std::cout << "Custom Memory Allocation: " << size << " bytes" << std::endl;
    return malloc(size);
}

// Custom memory deallocation function (override delete operator)
void operator delete(void* ptr) noexcept {
    std::cout << "Custom Memory Deallocation" << std::endl;
    free(ptr);
}

int main() {
    // Allocate an array of integers using the custom allocator
    int* arr = new int[5];

    // Deallocate the array using the custom deallocator
    delete[] arr;
    return 0;
}
```

Description:

1. Header Inclusions:

- `#include <iostream>` : Standard input/output stream.
- `#include <cstdlib>` : General utilities.

2. Custom Memory Allocation (new Operator Override):

- `void* operator new(size_t size)` : This function is an override for the global `new` operator. It is called when memory allocation is requested. Here, it prints a message indicating the custom memory allocation and returns a pointer to the allocated memory.

3. Custom Memory Deallocation (delete Operator Override):

- `void operator delete(void* ptr) noexcept` : This function is an override for the global `delete` operator. It is called when memory needs to be deallocated. Here, it prints a message indicating the custom memory deallocation and frees the memory using `free()` .

4. Main Function:

- `int* arr = new int[5];` : Allocates an array of integers using the custom allocator. The custom `new` operator is called, and a message is printed.

- `delete[] arr;` : Deallocates the array using the custom deallocator. The custom `delete` operator is called, and a message is printed.

Note:

- This example demonstrates a basic custom memory allocator for educational purposes.
- In a real-world scenario, custom allocators might involve more sophisticated strategies, such as pooling or specialized memory management.
- Use custom allocators judiciously, considering the specific requirements of your application.

30 ***Regular expressions:*** (regex) in C++ provide a powerful mechanism for pattern matching and manipulation of strings. The given code demonstrates the use of regular expressions to find occurrences of a specific pattern in a text.

```
#include <iostream>
#include <regex>

int main() {
    // Input text containing the pattern
    std::string text = "C++ is powerful and C++ is fast!";

    // Regular expression pattern to match "C++"
    std::regex pattern("C\\+\\+");
```

```
    // Iterator for finding matches in the text
    std::sregex_iterator it(text.begin(), text.end(), pattern);

    // Iterator to represent the end of the matches
    std::sregex_iterator end;

    // Loop through matches and print found instances
    while (it != end) {
        std::cout << "Found: " << it->str() << std::endl;
        ++it;
    }
    return 0;
}
```

Description:

1. Header Inclusions:

- `#include <iostream>` : Standard input/output stream.
- `#include <regex>` : Regular expression support.

2. Input Text and Regular Expression Pattern:

- `std::string text = "C++ is powerful and C++ is fast!";` : Defines a string containing the text to be searched.
- `std::regex pattern("C\\+\\+");` : Defines a regular expression pattern to match occurrences of "C++" in the text. Note the escape sequence `\\` to represent a literal plus sign.

3. Search for Matches:

- `std::sregex_iterator it(text.begin(), text.end(), pattern);` : Initializes an iterator `it` to find matches of the pattern in the text.
- `std::sregex_iterator end;` : Represents the end of the matches.

4. Iterate and Print Matches:

- The `while` loop iterates through the matches and prints each found instance.
- `it->str()` : Retrieves the matched substring.

Note:

- Regular expressions provide a flexible and expressive way to describe patterns in strings.
- Escape characters are used to represent special characters in the regular expression pattern.
- The `std::sregex_iterator` is used to iterate over the sequence of matches.

31 ***RAII:*** is a programming idiom in C++ that stands for "*Resource Acquisition Is Initialization*." It is a technique where resource management (such as memory allocation, file handling, etc.) is tied to the

lifespan of an object. The given code demonstrates the RAII principle with a `FileHandler` class for handling file operations.

```
#include <iostream>
#include <fstream>

class FileHandler {
public:
    // Constructor: Opens the file
    FileHandler(const std::string& filename) : file(filename) {
        std::cout << "File opened: " << filename << std::endl;
    }

    // Destructor: Closes the file
    ~FileHandler() {
        file.close();
        std::cout << "File closed." << std::endl;
    }

    // Member function to write data to the file
    void writeData(const std::string& data) {
        file << data;
    }
private:
    std::ofstream file; // File stream as a private member
};

int main() {
    // Creating an instance of FileHandler opens the file
    FileHandler file("example.txt");

    // Writing data to the file using the member function
    file.writeData("RAII is awesome!");

    // File is automatically closed when the FileHandler instance goes out of s
    return 0;
}
```

Description:

1. Header Inclusions:

- `#include <iostream>` : Standard input/output stream.
- `#include <fstream>` : File stream operations.

2. FileHandler Class:

- Constructor (`FileHandler(const std::string& filename)`)
- Opens the file specified by the filename.
- Prints a message indicating that the file is opened.
- Destructor (`~FileHandler()`)
- Closes the file.
- Prints a message indicating that the file is closed.
- Member Function (`writeData(const std::string& data)`)
- Writes data to the file.

3. Main Function:

- Creates an instance of `FileHandler` with the filename "example.txt," which opens the file.

- Writes the data “RAII is awesome!” to the file using the `writeData` member function.
- The `FileHandler` instance goes out of scope at the end of `main`, and its destructor is automatically called, closing the file.

RAII Principle:

- The resource (file in this case) is acquired (opened) during the initialization of the `FileHandler` object.
- The resource is automatically released (closed) when the `FileHandler` object goes out of scope, thanks to the destructor.
- This ensures that resource management is tied to the object’s lifecycle, making the code more robust and easy to maintain.

32 ***Functions Pointers & Callbacks:*** In C++, a function pointer is a variable that stores the address of a function. Function pointers are powerful mechanisms that enable the use of functions as arguments to other functions, allowing for dynamic behavior and callback functionality. The given code illustrates the usage of function pointers and callbacks.

```
#include <iostream>

// Function to print a message
void printMessage(const char* message) {
    std::cout << "Message: " << message << std::endl;
}
```

```
// Function to process a callback function
void process(void (*callback)(const char*)) {
    callback("Function Pointers");
}

int main() {
    // Passing the printMessage function as a callback to the process function
    process(printMessage);
    return 0;
}
```

Description:

1. *Function Declaration (`void printMessage(const char message)’):**

- Defines a function `printMessage` that takes a `const char*` parameter and prints the message.

2. Function Declaration (`void process(void (*callback)(const char))’):

- Defines a function `process` that takes a function pointer `callback` as a parameter.
- The function pointer is expected to point to a function that takes a `const char*` parameter.

3. Main Function:

- Calls the `process` function with the `printMessage` function as an

argument.

- The `printMessage` function is passed as a function pointer to the `process` function.

Function Pointers:

- `void (*callback)(const char*)` : Declares a function pointer named `callback` that points to a function taking a `const char*` parameter and returning `void` .

Callback Concept:

- In the `process` function, the provided callback function (in this case, `printMessage`) is invoked with the argument "Function Pointers."

Execution:

1. The `process` function is called in `main` .
2. Inside `process` , the provided callback function (`printMessage`) is called with the argument "Function Pointers."
3. The `printMessage` function prints the message.

Function Pointers and Callbacks Usage:

- This pattern is commonly used in scenarios where a function needs to

customize its behavior dynamically.

- Callbacks are essential in event handling, asynchronous programming, and various other situations where dynamic behavior is required.

Note:

- Modern C++ often uses features like `std::function` and lambdas for more expressive and flexible callback mechanisms.

33 *C++17 structured bindings*, which is a convenient way to unpack the elements of a tuple or other data structures into individual variables. This feature improves code readability and simplifies the extraction of multiple values from complex structures. The provided code demonstrates the usage of structured bindings with a tuple.

```
#include <iostream>
#include <tuple>

// Function returning a tuple
std::tuple<int, std::string, double> getPersonInfo() {
    return std::make_tuple(25, "John Doe", 175.5);
}

int main() {
    // Structured binding to unpack values from the tuple
    auto [age, name, height] = getPersonInfo();
    // Printing the unpacked values
    std::cout << name << " is " << age << " years old and " << height << " cm t
    return 0;
}
```

Description:

1. Tuple Declaration (`std::tuple<int, std::string, double>`):

- Defines a tuple type that holds an integer (age), a string (name), and a double (height).

1. Function (`getPersonInfo`) Returning a Tuple:

- Returns a tuple with sample values for age, name, and height.

1. Main Function:

- Utilizes structured bindings to unpack the values from the tuple returned by `getPersonInfo` .
- `auto [age, name, height]` : Declares and initializes individual variables using structured bindings.

1. Printing Unpacked Values:

- Uses the unpacked variables (`age` , `name` , and `height`) to construct and print a sentence.

Structured Bindings:

- `auto [age, name, height]` : Declares and initializes individual variables by directly binding them to the tuple elements.
- This avoids the need for manual indexing and provides a cleaner syntax.

Benefits:

- Improved readability: Code becomes more expressive and self-documenting.
- Simplified extraction: Easily access tuple elements without needing `std::get` or indexing.

Note:

- Structured bindings work not only with tuples but also with other iterable structures like arrays and user-defined types that provide the necessary interface.

Output:

```
John Doe is 25 years old and 175.5 cm tall.
```

Structured Bindings Usage:

- Particularly useful when dealing with functions returning multiple values in a structured form.
- Enhances code clarity in situations involving complex data structures.

34 *Concurrency — Atomic Operations* in C++ involves dealing with multiple threads executing concurrently. In situations where shared data is accessed and modified by multiple threads simultaneously, atomic operations become crucial to avoid data races and ensure correct behavior. The provided code illustrates the use of atomic operations to increment a shared counter safely.

```
#include <iostream>
#include <thread>
#include <atomic>

// Atomic variable for shared counter
std::atomic<int> counter(0);

// Function to increment the counter
void increment() {
    for (int i = 0; i < 1000000; ++i) {
        // Atomic fetch_add operation to increment counter
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    // Creating two threads to concurrently increment the counter
    std::thread t1(increment);
    std::thread t2(increment);

    // Waiting for both threads to finish
    t1.join();
    t2.join();
```

```
    // Printing the final value of the counter
    std::cout << "Counter: " << counter.load() << std::endl;
    return 0;
}
```

Description:

1. Atomic Counter (`std::atomic<int> counter(0)`):

- Declares an atomic variable `counter` initialized to 0, ensuring atomic operations on this variable.

2. Increment Function (`void increment()`):

- Increments the shared counter in a loop using `fetch_add` , which is an atomic operation.
- `std::memory_order_relaxed` : Specifies the memory order, indicating that no additional synchronization is required.

3. Main Function:

- Creates two threads (`t1` and `t2`) that concurrently execute the `increment` function.

4. Thread Joining (`t1.join()` and `t2.join()`):

- Waits for both threads to complete their execution before proceeding.

5. Printing Counter Value (`std::cout << "Counter: " << counter.load() << std::endl`):

- Loads and prints the final value of the counter using `load` , another atomic operation.

Atomic Operations:

- Atomic operations ensure that the execution of the operation is indivisible, avoiding race conditions.
- In this example, `fetch_add` and `load` are atomic operations provided by the `std::atomic` library.

Memory Order:

- `std::memory_order_relaxed` : Specifies the minimal memory synchronization, suitable for this example.

Output:

```
Counter: 2000000
```

Concurrency and Atomic Operations:

- Essential for thread safety when multiple threads access shared data concurrently.
- Atomic operations guarantee that the operation is completed without interruption, preventing data races.

Note:

- The final counter value is the sum of increments from both threads, demonstrating the necessity of atomicity in concurrent scenarios.

35 *C++20 Concepts*, which is a way to specify constraints on template parameters. Concepts enable more readable and expressive template code while improving error messages. The provided code demonstrates a simple example of using Concepts to constrain a template function to work only with integral types.

```
#include <iostream>

// Concept definition for Integral types
template <typename T>
concept Integral = std::is_integral<T>::value;

// Function template constrained by the Integral concept
template <Integral T>
T add(T a, T b) {
    return a + b;
}
```

```
int main() {
    // Using the add function with integral types (int)
    std::cout << "Sum: " << add(3, 4) << std::endl;

    // Uncommenting the line below will result in a compilation error
    // std::cout << "Sum: " << add(2.5, 3.5) << std::endl;

    return 0;
}
```

Description:

1. Concept Definition (`template <typename T> concept Integral = std::is_integral<T>::value;`):

- Defines a concept named `Integral` that checks if the given type `T` is integral using `std::is_integral`.

2. Function Template with Concept (`template <Integral T> T add(T a, T b)`):

- The `add` function template is constrained by the `Integral` concept, meaning it only accepts types that satisfy the concept.

3. Main Function:

- Uses the `add` function with integral types (`int`) and prints the sum.

4. Uncommented Line (`// std::cout << "Sum: " << add(2.5, 3.5) <<`

`std::endl;`):

- Uncommenting this line would result in a compilation error because the `add` function is constrained to integral types, and using it with non-integral types (e.g., `double`) violates the concept.

C++20 Concepts Features:

- Readability: Concepts provide a more readable way to express template constraints.
- Compilation Errors: Improved error messages during compilation, making it easier to understand and fix template-related issues.
- Code Intent: Clearly specifies the requirements on template parameters.

Output:

```
Sum: 7
```

Note:

- Concepts enhance template programming by expressing the intended constraints directly in the code, promoting better code understanding and maintainability.

This concludes the Advanced level of the C++ cheatsheet.

Note: Please check out my C++ cheatsheet if you have not read it yet.

01 c++ Cheatsheet: Basic Level

02 c++ Cheatsheet: Intermediate Level

03 c++ Cheatsheet: Advanced Level

04 c++ Cheatsheet: Expert Level

Cplusplus Programming

Written by Deepak Ranolia

539 Followers · 12 Following

Strong technical skills, such as Coding, Software Engineering, Product Management & Finance. Talk about finance, technology & life
https://rb.gy/9tod91

Follow

Responses (2)

What are your thoughts? Respond

Greg Herlihy
12 months ago (edited)

The atomic example is overly complicated. First, there is a typedef for an std::atomic<int>, std::atomic_int.

Since the example is incrementing the counter, it can simplly do so in the ordinary way:

++counter; // atomically pre increment counter.....

Read More

1 Reply

Rob Watson
9 months ago

Is it me or does every programming post on Medium just consist of regurgitated snippets of info taken from the web, probably just to give the blog poster another blog to add to their portfolio?

Reply

More from Deepak Ranolia

Deepak Ranolia

Understanding Proxychains4.conf & anonsurf in Kali Linux

Proxychains is a powerful tool that enables users to run any application through a prox…

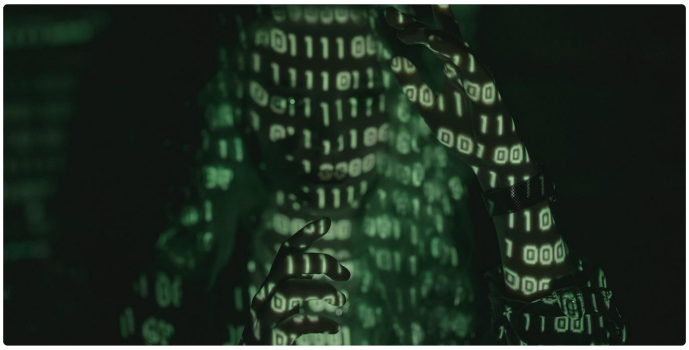
Nov 13, 2023 19

Deepak Ranolia

Step-by-Step Guide: Creating a Java Server for Backend Web…

Step 1: Install Prerequisites

Dec 4, 2023 23



Deepak Ranolia

Nikto: Scanning Web Servers for Vulnerabilities

In an increasingly digitized world, web servers have become the backbone of…

Nov 5, 2023 2

Deepak Ranolia

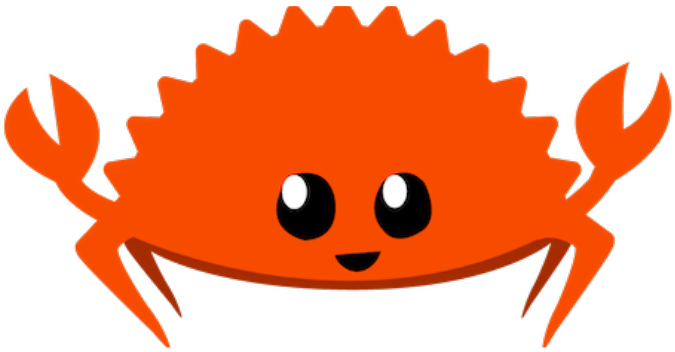
C++ Cheatsheet: Expert Level

Welcome to the C++ Cheatsheet designed to enhance your proficiency in C++…

Dec 5, 2023 141 1

See all from Deepak Ranolia

Recommended from Medium



In CodeX by Austin Starks

I spent 2 years rebuilding my algorithmic trading platform in...

Dec 5 269 7

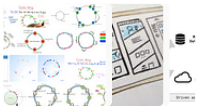
Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 25 7.4K 111

Lists

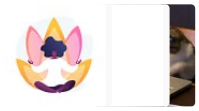


General Coding Knowledge

20 stories · 1807 saves

Coding & Development

11 stories · 935 saves



Stories to Help You Grow as a Software Developer

19 stories · 1517 saves

ChatGPT

21 stories · 908 saves

Keith McNulty

Modeling the Movement of Projectiles

What happens when an object is launched into the air and acted upon only by gravity?

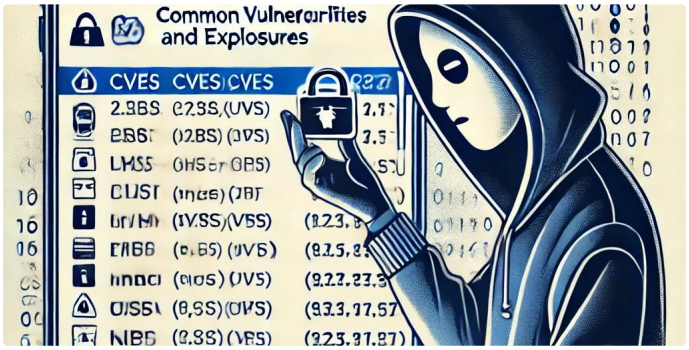
4d ago 447 12

In Stackademic by Oliver Foster

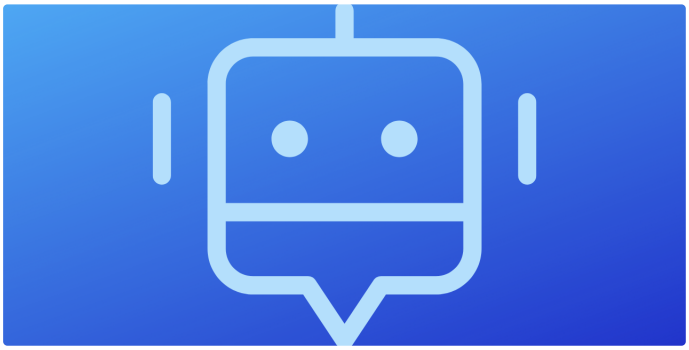
What's the Difference Between localhost and 127.0.0.1?

My article is open to everyone; non-member readers can click this link to read the full text.

Feb 1 6.4K 67



Jonathan Mondaut



In Better Programming by David Amos

How ChatGPT Turned Me into a Hacker

Discover how ChatGPT helped me become a hacker, from gathering resources to tacklin...

Jun 18 2.2K 83

ChatGPT Is an Extra-Ordinary Python Programmer

Not extraordinary. Extra-ordinary. As in, its ordinariness abounds.

Feb 14, 2023 1.3K 19

See more recommendations