C++ Cheatsheet: Expert Level

Deepak Ranolia · Follow

7 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++.

> ***04 Expert Level***
>
> *The Expert Level explores sophisticated C++ concepts like metaprogramming, expression templates, policy-based design, coroutines, and custom allocators. This level is designed for seasoned C++ developers looking to master intricate aspects of the language.*

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

03 c++ Cheatsheet: Advanced Level

36 ***Meta Programming with Template Specialization*** Template specialization is a powerful feature in C++ that allows you to provide custom implementations for specific template parameter values. In the provided code, template specialization is used to create a specialized version of the `printType` function for the `int` type.

```cpp
#include <iostream>

// Primary template for printType
template <typename T>
void printType() {
    std::cout << "Type: " << typeid(T).name() << std::endl;
}

// Template specialization for int type
template <>
void printType<int>() {
    std::cout << "Specialized Type: int" << std::endl;
}

int main() {
    // Call printType with double type
    printType<double>();

    // Call the specialized version of printType for int type
    printType<int>();
    return 0;
}
```

Description:

1. Primary Template (`template <typename T> void printType()`):

- Defines a primary template for the `printType` function, which prints the type name using `typeid` .

2. Template Specialization (`template <> void printType<int>()`):

- Provides a specialized version of the `printType` function specifically for

the `int` type. This version will be used when the template is instantiated with `int` as the type.

3. Main Function:

- Calls `printType` with the `double` type, which uses the primary template.
- Calls `printType` with the `int` type, which uses the specialized version for `int`.

Output:

```
Type: double
Specialized Type: int
```

Meta programming with Template Specialization Features:

- *Customization:* Allows customizing behavior for specific template parameter values.
- *Selective Implementation:* Provides a way to have different implementations for different types.
- *Code Clarity:* Enhances code readability and expressiveness by explicitly defining specialized cases.

Note:

- Template specialization is often used in metaprogramming scenarios where specific behaviors are needed for certain types or conditions. It allows tailoring generic code to specific requirements.

37 ***Expression templates-Efficient Numeric Computations*** in C++ are a powerful technique for optimizing numeric computations by representing mathematical expressions as expression template classes. These classes defer the evaluation of expressions until the final result is needed, allowing for more efficient code generation.

```
#include <iostream>
#include <vector>

// Abstract base class for vector expressions
template <typename T> class VectorExpression {
public:
    virtual T operator[](size_t index) const = 0;
    virtual size_t size() const = 0;
    virtual ~VectorExpression() = default;
};

// Concrete class representing a vector
template <typename T> class Vector : public VectorExpression<T> {
public:
    // Overridden operators for deferred evaluation
    T operator[](size_t index) const override {
        return elements[index];
    }
    size_t size() const override {
        return elements.size();
    }
```

```
    // Additional vector operations can be added here
    private:
        std::vector<T> elements;
    };

    int main() {
        // Create instances of Vector<double>
        Vector<double> v1, v2;
        // Perform vector operations (not shown in the provided code)
        return 0;
    }
```

Description:

1. Abstract Base Class (`VectorExpression`):

- Represents the abstract base class for vector expressions.
- Declares pure virtual functions `operator[]` and `size` to be implemented by derived classes.

2. Concrete Class (`Vector`):

- Inherits from `VectorExpression` to define a concrete vector class.
- Implements `operator[]` and `size` to provide deferred evaluation.

3. Main Function:

- Creates instances of the `Vector<double>` class (v1, v2).

- Performs vector operations (not shown in the provided code) using the expression template approach.

Expression Templates Features:

- *Deferred Evaluation:* Expressions are represented as objects and evaluated only when the final result is required, avoiding unnecessary intermediate computations.
- *Optimization:* Enables optimizations by allowing the compiler to generate efficient code for numeric computations.
- *Customization:* Provides a flexible way to define and extend mathematical operations for specific types.

Note:

- The provided code is a simplified example, and the full power of expression templates is realized when combining multiple expressions and operations. The goal is to generate efficient code by minimizing unnecessary intermediate results.

38 ***Policy-Based Design*** is a C++ programming technique that allows you to customize the behavior of a class by providing policy classes as template parameters. This approach promotes flexibility and code reuse by separating concerns and enabling the selection of different policies at compile-time.

```cpp
#include <iostream>

// Policy class for logging to the console
struct ConsoleLogger {
    static void log(const std::string& message) {
        std::cout << "Console: " << message << std::endl;
    }
};

// Policy class for logging to a file
struct FileLogger {
    static void log(const std::string& message) {
        // ... Logging to a file
    }
};

// Template class representing an engine with a logging policy
template <typename LoggerPolicy> class Engine {
public:
    void start() {
        LoggerPolicy::log("Engine started");
        // ... Engine startup logic
    }
};

int main() {
    // Instantiate Engine with ConsoleLogger policy
    Engine<ConsoleLogger> engine1;

    // Instantiate Engine with FileLogger policy
    Engine<FileLogger> engine2;

    // Use engines
    engine1.start();
    engine2.start();
    return 0;
}
```

Decription:

1. Policy Classes:

- `ConsoleLogger` : Implements the logging policy to print messages to the console.
- `FileLogger` : Represents an alternative logging policy that could log messages to a file.

2. Template Class (`Engine`):

- Represents a generic engine class parameterized by a logging policy.
- Provides a method `start()` that utilizes the logging policy to log a message when the engine starts.

3. Main Function:

- Instantiates an `Engine` with `ConsoleLogger` policy (engine1) and another with `FileLogger` policy (engine2).
- Calls the `start()` method on each engine, demonstrating how the behavior is determined by the selected logging policy.

Key Points:

- *Customization:* Policy-Based Design allows the customization of class behavior by selecting different policies at compile-time.

- *Separation of Concerns:* Policies separate concerns, making it easier to modify or extend individual aspects of a class without affecting the core functionality.
- *Compile-Time Decisions:* Policies are chosen at compile-time, providing efficiency and avoiding runtime overhead associated with runtime polymorphism.

Note:

- This design pattern is particularly useful when a class has multiple aspects or behaviors that can vary independently, and it allows for easy adaptation to different requirements without modifying the core class.

39 *Coroutines* in C++ are a feature introduced in C++20 that allows a function to be paused and later resumed. They are particularly useful for asynchronous programming and generator-like functionality.

```
#include <iostream>
#include <coroutine>

// Generator Structure
struct Generator {
    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;

    // Promise Type
    struct promise_type {
        int current_value;

        // Create a generator from the promise
```

```
        auto get_return_object() {
            return Generator{handle_type::from_promise(*this)};
        }

        // Initial suspend point
        auto initial_suspend() {
            return std::suspend_always{};
        }

        // Final suspend point
        auto final_suspend() noexcept {
            return std::suspend_always{};
        }

        // Return void for co_await
        void return_void() {}

        // Yield a value
        auto yield_value(int value) {
            current_value = value;
            return std::suspend_always{};
        }

        // Handle unhandled exceptions
        void unhandled_exception() {
            std::terminate();
        }
    };

    // Move to the next value
    bool move_next() {
        if (coro) {
            coro.resume();
            return !coro.done();
        }
        return false;
    }

    // Get the current value
    int current_value() const {
        return coro.promise().current_value;
    }
private:

    // Constructor with coroutine handle
```

```
    Generator(handle_type h) : coro(h) {}

    // Coroutine handle
    handle_type coro;
};

// Counter Generator Function
Generator counter(int start, int end) {
    for (int i = start; i < end; ++i) {
        co_yield i;
    }
}

// Main Function
int main() {

    // Create a generator
    auto gen = counter(0, 5);

    // Iterate through the generator values
    while (gen.move_next()) {
        std::cout << gen.current_value() << std::endl;
    }
    return 0;
}
```

Description:

1. Generator Structure:

- `promise_type` : Manages the coroutine's promise, providing the necessary functions like `get_return_object`, `initial_suspend`, `final_suspend`, `return_void`, `yield_value`, and `unhandled_exception`.

2. Generator Functions:

- `counter` : A generator function using the `co_yield` keyword to yield values.

3. Main Function:

- Creates a generator (`gen`) using the `counter` function.
- Iterates through the generator's values using `move_next` and `current_value` .
- Prints the values to the console.

Note: This code demonstrates the use of coroutines in C++ to create a generator that yields values, providing a concise and readable way to implement generators and asynchronous tasks.

40 ***Custom Allocators*** In C++, allocators are responsible for managing memory allocation and deallocation for containers like vectors, lists, and other dynamic data structures. The standard library provides default allocators, but users can also define custom allocators to control memory management according to specific needs.

```
#include <iostream>
#include <memory>

// Custom Allocator Definition
template <typename T>
struct MyAllocator {
    using value_type = T;
```

```
    MyAllocator() noexcept {}
    // Constructor to allow converting between different allocator types
    template <typename U>
    MyAllocator(const MyAllocator<U>&) noexcept {}
    // Allocate memory for 'n' elements of type T
    T* allocate(std::size_t n) {
        if (auto p = std::malloc(n * sizeof(T))) {
            return static_cast<T*>(p);
        }
        throw std::bad_alloc();
    }
    // Deallocate memory
    void deallocate(T* p, std::size_t) noexcept {
        std::free(p);
    }
};

// Comparison operators for allocators
template <typename T, typename U>
bool operator==(const MyAllocator<T>&, const MyAllocator<U>&) noexcept {
    return true;
}

template <typename T, typename U>
bool operator!=(const MyAllocator<T>&, const MyAllocator<U>&) noexcept {
    return false;
}

int main() {
    // Using a vector with the custom allocator
    std::vector<int, MyAllocator<int>> vec;
    vec.push_back(42);
    // ... Use the vector
    return 0;
}
```

Description:

1. *MyAllocator Structure*:

- Defines a custom allocator named `MyAllocator` for type `T`.
- Implements the `allocate` method for memory allocation.
- Implements the `deallocate` method for memory deallocation.
- Provides constructors to allow converting between different allocator types.

2. *Comparison Operators*:

- `operator==` and `operator!=` are defined to compare allocators for equality.

3. *Main Function*:

- Creates a vector (`std::vector`) of integers using the custom allocator (`MyAllocator<int>`).
- Pushes an element (42) into the vector.
- Illustrates the usage of a custom allocator with a standard library container.

Note: This code demonstrates the creation and usage of a custom allocator with a vector. Custom allocators are powerful tools for fine-tuning memory management in C++ programs.

This concludes the Expert 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

Whether you're a beginner aspiring to become proficient in C++ or an experienced developer aiming to refine your skills, this cheatsheet is your comprehensive guide. Dive into each level at your own pace, experiment with the examples provided, and elevate your C++ programming prowess. Happy coding!

Cplusplus Programming

Written by Deepak Ranolia

539 Followers · 12 Following

Follow

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

Responses (1)

What are your thoughts?

Respond

Md Shoriful Islam Ashiq
7 months ago

Thanks. So much informative.

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...

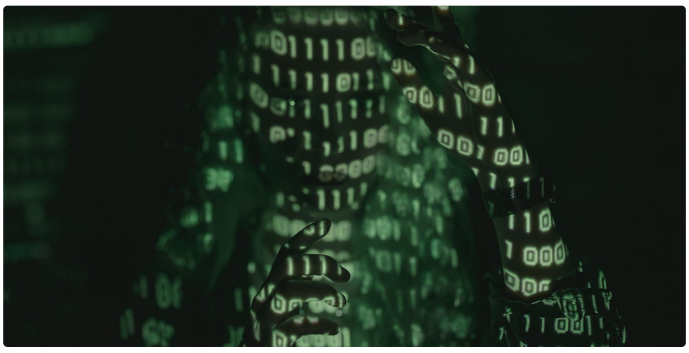
Nov 13, 2023 19

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...

Deepak Ranolia

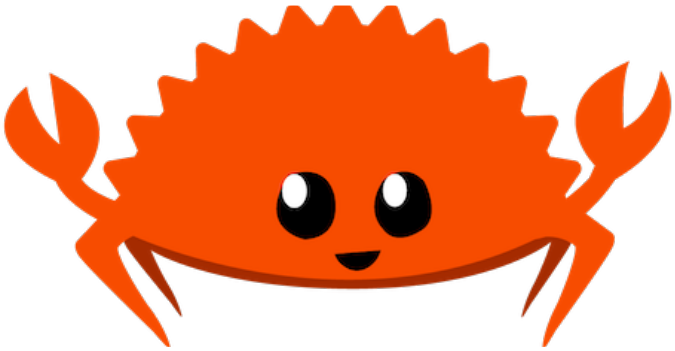
Python Cheatsheet

This Python cheatsheet is designed to be your quick reference guide for Python...

Nov 5, 2023 2 Nov 18, 2023 955 9

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

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

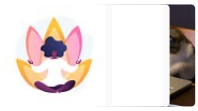
Lists

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

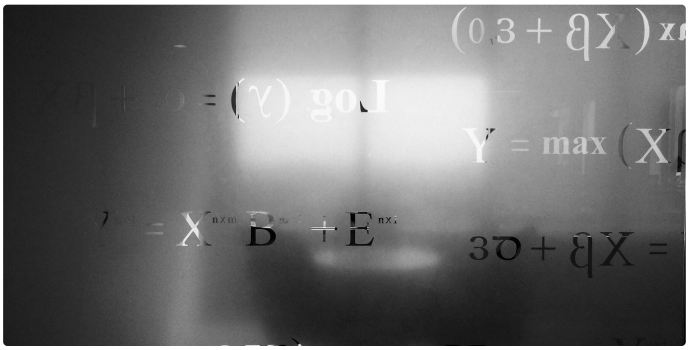
21 stories · 908 saves

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

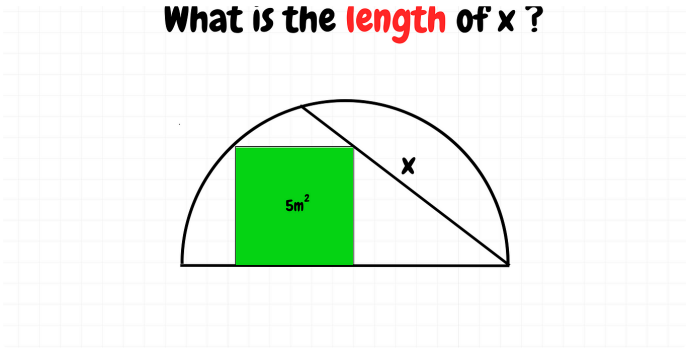


In Puzzle Sphere by Rajul Saxena

The Hidden Math Behind Image Blur

How Simple Math Transforms Images from Crisp to Smooth

Nov 30 698 22

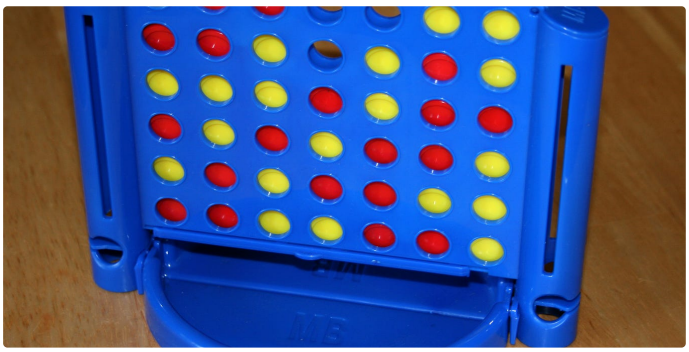


In Think Art by Nnamdi Samuel

A Russian Math Olympiad Problem

Can You Find the Length of x?

Sep 15 291 11



In Science Spectrum by Cole Frederick

How Math Broke Connect 4

A mathematician found a way to guarantee a win

4d ago 669 8