C++ Cheatsheet: Intermediate Level

Deepak Ranolia · Follow

12 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++.

> ***02 Intermediate Level***
>
> *In the Intermediate Level, you'll delve into more advanced topics such as object-oriented programming, pointers, and templates. This level aims to strengthen your understanding of C++ fundamentals and prepare you for more complex*

> *scenarios.*

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

01 c++ Cheatsheet: Basic Level

16 *Object-Oriented Programming (OOP)* is a programming paradigm that uses objects and classes for organizing code. A class is a blueprint for creating objects, and an object is an instance of a class. In C++, classes encapsulate data and behavior (methods) into a single unit.

```
#include <iostream>

// Definition of the Rectangle class
class Rectangle {
public:
    int length;
    int width;
    // Method to calculate the area of the rectangle
    int area() {
        return length * width;
    }
};
// Main function
int main() {
    // Creating an object of the Rectangle class
    Rectangle rect;
    // Assigning values to the object's attributes
    rect.length = 4;
    rect.width = 5;
    // Calculating and displaying the area of the rectangle
    std::cout << "Rectangle Area: " << rect.area() << std::endl;
    return 0;
}
```

Description:

- The `Rectangle` class is defined with two public attributes (`length` and `width`) and a method (`area`) to calculate the area of the rectangle.
- `public` specifies the access level, allowing attributes and methods to be accessed outside the class.
- An object of the `Rectangle` class named `rect` is created.
- Values are assigned to the attributes of the `rect` object.
- The `area` method is called on the `rect` object to calculate the area.
- The result is displayed using `std::cout` .
- The program returns 0 to indicate successful execution.

Note: In summary, the code demonstrates the creation of a `Rectangle` class with attributes and a method. An object of this class is then created, and its attributes are assigned values. Finally, the area of the rectangle is calculated and displayed.

17 *Operator overloading* in C++ allows you to define how operators should behave for user-defined types. In the provided code, the `+` operator is overloaded for the `Complex` class, enabling the addition of two complex numbers using the `+` operator.

```
#include <iostream>

// Definition of the Complex class
class Complex {
  public:
    int real;
    int imag;
    // Operator overloading for addition
    Complex operator+(const Complex &c) {
        Complex temp;
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        return temp;
    }
};

// Main function
int main() {
    // Creating objects of the Complex class
    Complex a, b, result;
    // Assigning values to complex numbers
    a.real = 3; a.imag = 2;
    b.real = 1; b.imag = 7;
    // Using the overloaded + operator to add complex numbers
    result = a + b;
    // Displaying the sum of complex numbers
    std::cout << "Sum: " << result.real << " + " << result.imag << "i" << std::
    return 0;
}
```

Description:

- The `Complex` class is defined with two public attributes (`real` and `imag`).
- The `+` operator is overloaded as a member function, taking another `Complex` object (`c`) as a parameter.

- Objects `a`, `b`, and `result` of the `Complex` class are created, and values are assigned to their attributes.
- The overloaded `+` operator is used to add the complex numbers `a` and `b`, and the result is stored in the `result` object.
- The sum of complex numbers is displayed using `std::cout`.
- The program returns 0 to indicate successful execution.

Note: In summary, the code showcases operator overloading for the addition of complex numbers. The `+` operator is defined to perform addition on `Complex` objects, allowing a natural way to add instances of the `Complex` class.

18

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class). The code provided demonstrates a simple example of inheritance in C++.

```
#include <iostream>

// Base class: Shape
class Shape {
public:
    // Member function in the base class
    void display() {
        std::cout << "Shape" << std::endl;
    }
};
```

```
// Derived class: Circle, inherits from Shape
class Circle : public Shape {
public:
    // Overriding the display function in the derived class
    void display() {
        std::cout << "Circle" << std::endl;
    }
};
// Main function
int main() {
    // Creating an object of the derived class Circle
    Circle circle;
    // Calling the overridden display function in the Circle class
    circle.display();
    return 0;
}
```

Description:

- The `Shape` class is defined with a member function `display`.
- The `Circle` class is defined, indicating that it publicly inherits from the `Shape` class.
- The `display` function in the `Circle` class overrides the `display` function in the `Shape` class.
- An object `circle` of the `Circle` class is created.
- The `display` function in the `Circle` class is called, which overrides the function in the base class `Shape`.
- The program outputs “Circle” because the overridden `display` function in the `Circle` class is called.

Note: In summary, this code illustrates the concept of inheritance, where the `Circle` class inherits from the `Shape` class, and the overridden function in the derived class is called when an object of the derived class is used.

19 *Polymorphism* is one of the four fundamental principles of object-oriented programming (OOP), allowing objects of different classes to be treated as objects of a common base class. In C++, polymorphism is achieved through virtual functions. The provided code illustrates runtime polymorphism using virtual functions.

```
#include <iostream>

// Base class: Animal
class Animal {
public:
    // Virtual function in the base class
    virtual void sound() {
        std::cout << "Animal Sound" << std::endl;
    }
};
// Derived class: Dog, inherits from Animal
class Dog : public Animal {
public:
    // Overriding the virtual function in the derived class
    void sound() override {
        std::cout << "Bark" << std::endl;
    }
};
// Main function
int main() {
    // Creating a pointer to the base class (Animal) and assigning it the addre
    Animal *ptr = new Dog();
    // Calling the virtual function using the pointer, which resolves to the ov
    ptr->sound();
    // Deleting the dynamically allocated object
```

```
    delete ptr;
    return 0;
}
```

Description:

- The `Animal` class is defined with a virtual function `sound`.
- The `Dog` class is defined, indicating that it publicly inherits from the `Animal` class.
- The `sound` function in the `Dog` class overrides the virtual function in the `Animal` class.
- A pointer `ptr` to the base class (`Animal`) is created and assigned the address of a dynamically allocated object of the derived class (`Dog`).
- The virtual function `sound` is called using the pointer. The call is resolved at runtime to the overridden function in the `Dog` class.
- The dynamically allocated object is deleted to free up memory.
- The program outputs “Bark” because the overridden `sound` function in the `Dog` class is called through the base class pointer.

Note: In summary, this code demonstrates polymorphism by allowing a pointer to the base class to be used to call a virtual function that is overridden by a derived class. The actual function called is determined at runtime based on the type of object the pointer is pointing to.

20 *Templates* in C++ allow you to write generic functions or classes that can work with any data type.

```cpp
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << "Sum: " << add(3, 4) << std::endl;
    std::cout << "Sum: " << add(2.5, 3.5) << std::endl;
    return 0;
}
```

Description:

- `template <typename T>` : This declares a template function where `T` is a placeholder for the data type. It allows the function to work with different types.
- `T add(T a, T b)` : This is the template function named `add` . It takes two parameters of type `T` and returns their sum of the same type.
- `int main()` : The main function.
- `std::cout << "Sum: " << add(3, 4) << std::endl;` : Calls the `add` function with integers 3 and 4. The compiler automatically deduces the type `T` as `int` in this case.

- `std::cout << "Sum: " << add(2.5, 3.5) << std::endl;` : Calls the `add` function with floating-point numbers 2.5 and 3.5. Here, `T` is deduced as `double` by the compiler.
- The program prints the sum of integers and the sum of floating-point numbers.

Note: This demonstrates how the same template function can handle different data types, providing flexibility and code reuse.

21 *The Standard Template Library (STL) - Vector* in C++ provides a collection of template classes and functions that implement many popular and commonly used algorithms and data structures. One of these is the `std::vector` class, which is a dynamic array that can grow or shrink in size.

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

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    numbers.push_back(6);
    std::cout << "Vector Size: " << numbers.size() << std::endl;
    return 0;
}
```

Description:

- `#include <iostream>` : Includes the necessary header file for input/output operations.
- `#include <vector>` : Includes the header file for the `std::vector` class.
- `int main()` : The main function.
- `std::vector<int> numbers = {1, 2, 3, 4, 5};` : Declares a vector named `numbers` of type `int` and initializes it with values 1, 2, 3, 4, and 5.
- `numbers.push_back(6);` : Adds the value 6 to the end of the vector using the `push_back` method. Vectors can dynamically resize to accommodate additional elements.
- `std::cout << "Vector Size: " << numbers.size() << std::endl;` : Prints the current size of the vector using the `size` method.
- `return 0;` : Indicates successful program execution.

Note: This code demonstrates the basic usage of the `std::vector` class. It initializes a vector with some values, adds a new value to it, and prints the updated size of the vector. Vectors are dynamic and can efficiently handle changes in size during program execution.

22 ***STL — MAP*** In C++, the Standard Template Library (STL) provides a `map` container that is implemented as a sorted binary tree. It is used to store key-value pairs where each key must be unique. This container allows efficient lookups, insertions, and deletions.

```cpp
#include <iostream>
#include <map>

int main() {
    // Declare a map with string keys and integer values
    std::map<std::string, int> ages;
    // Insert key-value pairs into the map
    ages["John"] = 30;
    ages["Alice"] = 25;
    // Access and print the value associated with the key "Alice"
    std::cout << "Alice's Age: " << ages["Alice"] << std::endl;
    return 0;
}
```

Description:

1. Include Headers:

- `#include <iostream>` : Header for input and output operations.
- `#include <map>` : Header for the `map` container.

2. Map Declaration:

- `std::map<std::string, int> ages;` : Declares a map named `ages` with string keys and integer values.

3. Insert Key-Value Pairs:

- `ages["John"] = 30;` : Inserts a key-value pair into the map where the key

is "John" and the value is 30.

- `ages["Alice"] = 25;` : Inserts another key-value pair for "Alice" with a value of 25.

4. Access and Print:

- `std::cout << "Alice's Age: " << ages["Alice"] << std::endl;` : Accesses the value associated with the key "Alice" and prints it to the console.

5. Output:

- The program prints “Alice’s Age: 25” to the console.

Note:

- Maps in C++ automatically sort their keys, making them efficient for searching and retrieval based on keys.
- The `[]` operator is used to access values associated with keys. If the key is not present, a new key-value pair is created.
- Maps do not allow duplicate keys; each key must be unique.

23 ***Exception handling*** in C++ allows the program to respond to unexpected events or errors during its execution. The `try`, `catch`, and `throw` keywords are used for this purpose.

```cpp
#include <iostream>

int main() {
    try {
        // Throw a runtime_error exception with a descriptive message
        throw std::runtime_error("An error occurred");
    } catch (const std::exception &e) {
        // Catch and handle the exception
        std::cerr << "Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

Description:

1. `try` Block:

- `try { /* code */ }`: The block of code inside the `try` block is monitored for exceptions.

1. `throw` Statement:

- `throw std::runtime_error("An error occurred");`: This statement throws a `std::runtime_error` exception with the specified error message.

1. `catch` Block:

- `catch (const std::exception &e) { /* code */ }`: If an exception of type `std::exception` (or its derived types) is thrown within the `try` block, the

corresponding `catch` block is executed.

- The `catch` block takes a reference to the exception object (`const std::exception &e`) that provides information about the exception.

1. Exception Handling:

- If an exception occurs in the `try` block, the control is transferred to the appropriate `catch` block.

1. Printing Exception Information:

- `std::cerr << "Exception: " << e.what() << std::endl;` : This line prints the exception information to the standard error stream. The `what()` function of the exception object returns a C-style string describing the exception.

1. Output:

- In this example, the program intentionally throws a `std::runtime_error` exception, and the catch block handles it by printing "Exception: An error occurred" to the error stream.

Note:

- Exception handling is crucial for writing robust and error-tolerant code.

- Catch blocks are selected based on the type of the thrown exception. In this case, it catches exceptions of type `std::exception` or its derived types.
- The `what()` function provides a human-readable description of the exception.

24 *Lambda expressions*, introduced in C++11, provide a concise way to create anonymous functions or function objects. They are particularly useful for short, one-time operations, where defining a separate named function is unnecessary.

```
#include <iostream>

int main() {
    // Lambda expression to add two integers
    auto add = [](int a, int b) { return a + b; };
    // Using the lambda function to add 3 and 4
    std::cout << "Sum: " << add(3, 4) << std::endl;
    return 0;
}
```

Description:

1. Lambda Expression:

- `auto add = [](int a, int b) { return a + b; };` : This line defines a lambda expression that takes two parameters (`int a` and `int b`) and

returns their sum (`a + b`).

- The `auto` keyword is used to automatically deduce the lambda's return type.

2. Lambda Capture (Optional):

- Lambda expressions can capture variables from their enclosing scope. In this example, there is no capture (`[]`), indicating that the lambda doesn't capture any variables.

3. Function Invocation:

- `add(3, 4)` : The lambda function is invoked with arguments `3` and `4` , and the result (sum) is printed.

4. Output:

- The program outputs “Sum: 7” because the lambda expression adds the values `3` and `4` .

Lambda Features:

- Capture Clause:
- The square brackets `[]` can be used to capture variables from the enclosing scope. For example, `[a, b]` captures variables `a` and `b` .

- Mutable Lambdas:
- If modifications to captured variables are needed, use the `mutable` keyword: `[a, b]() mutable { /* modifications */ }`.
- Return Type Deduction:
- The `auto` keyword can be used for the return type when it can be deduced.
- Parameter Types:
- Parameter types can be omitted if they can be deduced or specified explicitly.

Note: Lambda expressions are powerful and flexible, allowing the creation of concise and readable code for various tasks. They are extensively used in modern C++ programming.

25 ***Smart pointers*** in C++ are objects that act as pointers but provide additional features to manage the memory they point to. They automatically handle memory allocation and deallocation, helping to avoid memory leaks and other memory-related issues. Among smart pointers, `std::shared_ptr` is a type of smart pointer that enables shared ownership of a dynamically allocated object.

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

```
int main() {
    // Creating a shared_ptr to an integer with value 42
    std::shared_ptr<int> num = std::make_shared<int>(42);
    // Accessing and printing the value through the shared_ptr
    std::cout << "Value: " << *num << std::endl;
    // The shared_ptr will automatically deallocate memory when it goes out of
    return 0;
}
```

Description:

1. Header Inclusions:

- `#include <iostream>` : Standard input/output stream.
- `#include <memory>` : Header for smart pointers and related functionalities.

2. `std::shared_ptr` Creation:

- `std::shared_ptr<int> num = std::make_shared<int>(42);` : This line creates a shared_ptr named `num` that manages an integer with the value `42` . `std::make_shared` is a recommended way to create a shared_ptr, as it ensures efficient memory allocation.

3. Accessing Value:

- `*num` : The `*` operator is used to dereference the shared_ptr, providing access to the value it points to.

4. Output:

- The program outputs “Value: 42,” demonstrating that the value inside the shared_ptr is successfully accessed and printed.

Smart Pointer Features:

- *Automatic Memory Management*:
- Smart pointers automatically handle memory allocation and deallocation, eliminating the need for explicit `new` and `delete`.
- *Ownership Management*:
- `std::shared_ptr` allows multiple shared_ptrs to share ownership of the same dynamically allocated object. The object is only deallocated when the last shared_ptr pointing to it is destroyed.
- *Avoiding Memory Leaks*:
- Smart pointers help prevent memory leaks by ensuring proper memory deallocation, even in the presence of exceptions or early returns.
- *Simplifying Memory Management*:
- Smart pointers simplify memory management, making the code safer and more readable by reducing manual memory management errors.

Note: Smart pointers are an essential feature of modern C++ programming, providing a safer and more convenient alternative to raw pointers. They

play a crucial role in writing robust and memory-safe code.

This completes the Intermediate 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 (1)

What are your thoughts? Respond

lanjoyner
about 1 year ago

C++ is a self-serving language. All you really learn with C++ is C++, and you never master it, it masters you.

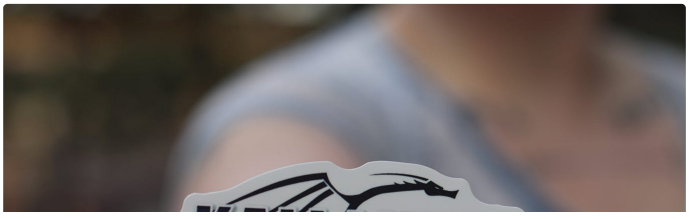
What is needed is a cheatsheet in general programming, particularly OO.

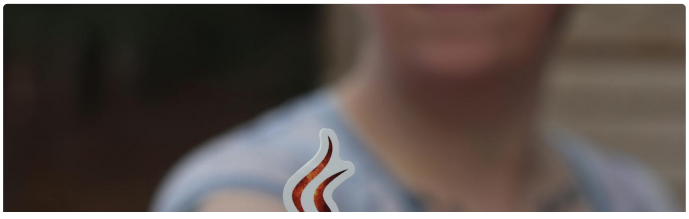
C++ is so arcane because it just perpetuates and magnifies the.....

Read More

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...

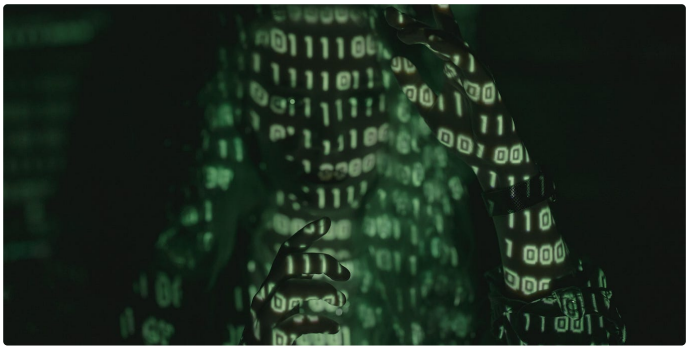
Nov 13, 2023 19

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: Advanced Level

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

Dec 5, 2023 115 2

See all from Deepak Ranolia

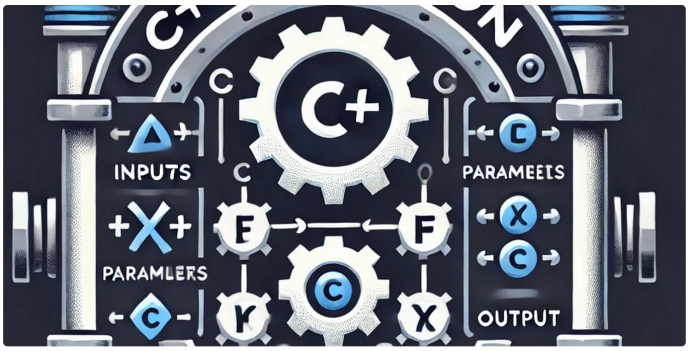
Recommended from Medium

anzhi zhu

100 C++ interview questions and answers

Below are 100 C++ interview questions, each with simple answers. Please note that these...

Jul 18 13 1



md

Lecture 3: C++ Functions

To create something in C++, you need to use functions. A function is a specific operation...

Dec 3 50 1

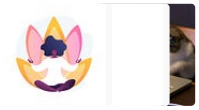
Lists

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

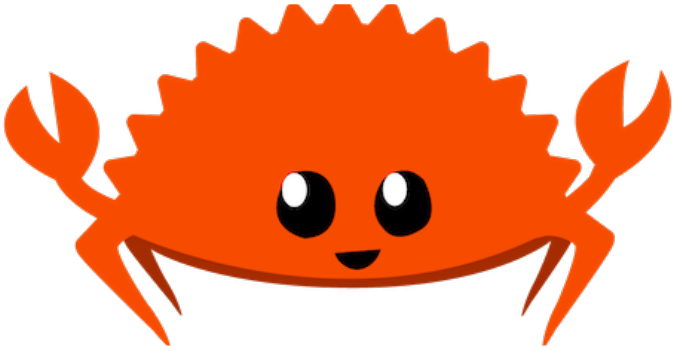
21 stories · 908 saves

In Nerd For Tech by Mohit Mishra

Implementing Shared Pointers in C for Robust Memory Management

C Can Be Smart Too, Taking Control of Memory

Oct 24 52 1



In CodeX by Austin Starks

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

Dec 5 269 7

Linking

Advanced Techniques in C Programming—2

23. Using Macros for Code Reusability

Oct 26

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

See more recommendations