Assignment_14-08-2024

Author: ThanhTh10
Date: 16/08/2024

1. Fix the syntax error in the virtual function program. Analyze the results
Code fixed:

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

typedef void (*caller)(void);
class base
{
public:
    base()
    {
        caller vfunc1;
        caller vfunc2;
        cout << "Object base constructed" << endl;
        cout << "Object address is " << this << endl;
        cout << "vfptr address is " << *(void**)this << endl;
        cout << "base::funct1 address is " << *(void**)(*(void**)this) << endl;
        cout << "base::funct2 address is " << *((void**)(*(void**)this)+1) << endl;
        cout << "Calling vfunctions using vptr and vtable:" << endl;
        /* Base class vtable function 1, 2 */
        vfunc1 = (caller)(*(void**)(*(void**)this));
        vfunc2 = (caller)(*((void**)(*(void**)this)+1));
        /* Call base class vtable function 1, 2 */
        vfunc1();
        vfunc2();
    }
    virtual void funct1(void)
    {
        cout << "base::funct1" << endl;
    }
    virtual void funct2(void)
    {
        cout << "base::funct2" << endl;
    }
};
class derived : public base
{
public:
    derived()
    {
        caller vfunc1;
        caller vfunc2;
        cout << "Object derived constructed" << endl;
        cout << "Object address is " << this << endl;
        cout << "vfptr address is " << *(void**)this << endl;
        cout << "derived::funct1 address is " << *(void**)(*(void**)this) << endl;
        cout << "derived::funct2 address is " << *((void**)(*(void**)this)+1) << endl;
        cout << "Calling vfunctions using vptr and vtable:" << endl;
        /* Derived class vtable function 1, 2 */
        vfunc1 = (caller)(*(void**)(*(void**)this));
        vfunc2 = (caller)(*((void**)(*(void**)this)+1));
        /* Call derived class vtable function 1, 2 */
        vfunc1();
        vfunc2();
    }
    void funct1(void)
    {
        cout << "derived::funct1" << endl;
    }
    void funct2(void)
    {
        cout << "derived::funct2" << endl;
    }
};
int main(int argc, char *argv[])
{
    derived d;
    caller vfunc1;
    caller vfunc2;
    void *** vfptr = (void ***) &d;
    void ** vtable = (void **) *vfptr;
    /* class vtable function 1, 2 */
    vfunc1 = (caller)(vtable[0]);
    vfunc2 = (caller)(vtable[1]);
    cout << "From main" << endl;
    cout << "Address of d is " << &d << endl;
    cout << "d.vfptr is " << vtable << endl;
    cout << "Address of d.funct1 is " << vtable[0] << endl;
    cout << "Address of d.funct2 is " << vtable[1] << endl;
    cout << "Calling vfunctions using vptr and vtable:" << endl;
    /* Call vtable function 1, 2 */
    vfunc1();
    vfunc2();
    return 0;
}
```

Analys code:
- Class base: The constructor of base prints the object's address, vfptr address, and the addresses of funct1 and funct2 in the vtable.
It then calls funct1 and funct2 using the vfptr and vtable.

- Class derived: The constructor of derived does the same as base, but with the addresses of funct1 and funct2 in the derived class's vtable.

- main function: Creates an object d of type derived.
Prints the address of d, the vfptr address, and the addresses of funct1 and funct2 in the vtable.
Calls funct1 and funct2 using the vfptr and vtable.

Output : The output will show the addresses of the objects, vfptrs, and vtable entries. The exact addresses will vary depending on the system and compiler.

2. How to fix the memory leak in singleton object creation program

Static Local Variable Method (Thread-Safe): Static local variables are initialized only once in a thread-safe manner. The static local variable method is simple and effective for singleton creation and destruction.

```
#include <iostream>

class Singleton
{
private:
    Singleton()
    {
        std::cout << "Singleton instance created\n";
    }
public:
    static Singleton &getInstance()
    {
        static Singleton instance; // Static local variable
        return instance;
    }
    void doSomething()
    {
        std::cout << "Doing something\n";
    }
};
int main()
{
    Singleton &singleton1 = Singleton::getInstance();
    singleton1.doSomething();
    return 0;
}
```

The static local variable instance is destroyed automatically when the program terminates, ensuring no memory leaks.

3. Explore about Rust. Pro and cons compare to C and C++

Rust is a modern systems programming language designed for performance, reliability, and memory safety. It addresses many of the issues present in older languages like C and C++, especially around memory management and concurrency.

Rust was created by Mozilla Research in 2010 and became popular due to its focus on safety, especially around memory. Its syntax is somewhat similar to C++ but provides unique features such as ownership and borrowing, which enable memory safety without a garbage collector.

Feature	Rust	C	C++
Memory Safety	Strong memory safety with the **ownership** and **borrowing** model; enforced at compile time, preventing common bugs like null pointer dereferencing, dangling pointers, and buffer overflows.	Manual memory management with functions like malloc() and free(), leading to potential memory leaks and vulnerabilities like buffer overflows.	Manual memory management but supported by **smart pointers** (e.g., std::unique_ptr, std::shared_ptr), though still prone to issues like dangling pointers.
Concurrency	**Data race-free concurrency** enforced by the borrow checker at compile time, ensuring safe multithreading.	Prone to **data races** and race conditions unless carefully managed with locks, semaphores, etc.	Supports multithreading with standard libraries (e.g., <thread>), but prone to **data races** and undefined behavior if not carefully managed.
Performance	Performance is **close to C/C++**, with minimal runtime overhead thanks to zero-cost abstractions and no garbage collector. Some overhead from safety checks, though.	**High performance** with minimal abstraction, allowing fine-grained control over hardware and system resources. Ideal for real-time and embedded systems.	Similar performance to C, but **may incur some overhead** due to more complex features (e.g., virtual functions, exceptions, templates).

Feature	Rust	C	C++
Compile Times	**Slower compile times** due to extensive compile-time checks for memory safety and concurrency.	**Very fast compile times** due to its simple structure and lack of complex features like templates or safety checks.	**Slower compile times** compared to C due to its rich feature set (templates, exceptions, etc.) and heavy optimizations.
Ease of Learning	**Steep learning curve** due to the unique ownership model and borrow checker, even though the syntax is similar to C++.	**Simpler to learn** for small programs due to its minimal feature set, but error-prone and lacks modern abstractions.	**Steep learning curve**, especially with complex features like templates, exceptions, and object-oriented programming.
Memory Management	**Automatic memory management** through ownership and borrowing, with no garbage collector. Memory is automatically freed when no longer in use, reducing the risk of leaks.	Manual memory management, requiring explicit allocation and deallocation. Errors can easily result in memory leaks or undefined behavior.	**Manual management**, but with support for **RAII** (Resource Acquisition Is Initialization) and smart pointers to help manage memory more safely.
Tooling & Package Management	**Excellent tooling** with Cargo (build system, dependency manager, and testing framework). Integrated and easy to use.	**Basic tooling**, usually relying on external build systems like Make. No built-in package manager.	**Mature tooling** like CMake for build automation, and package management is improving with tools like Conan and vcpkg.
Ecosystem and Libraries	Growing ecosystem with increasing support, but still **smaller than C/C++**. **Crates.io** offers a centralized package registry.	Vast and **highly mature ecosystem**, with libraries for virtually every domain (e.g., POSIX, networking, embedded systems).	Extremely large and mature ecosystem, with **boost**, **STL**, and many third-party libraries for nearly every use case.
Cross-Platform Support	Cross-platform, with excellent support for modern OSes (Linux, macOS, Windows).	Extremely cross-platform, available on almost every system, including embedded platforms and microcontrollers.	Cross-platform, widely used on all major OSes and embedded platforms, with support for hardware-specific code.
Concurrency Model	Provides a safe concurrency model through **ownership and borrowing**, preventing data races.	**Low-level concurrency**, typically using POSIX threads or equivalent. No built-in safety for race conditions.	Supports higher-level concurrency models (e.g., thread pools, <thread> library) but still prone to **unsafe concurrency** without manual synchronization.
Abstraction Level	Allows for **high-level abstractions** while maintaining zero-cost (i.e., no runtime cost for abstractions).	**Low-level programming** directly interacting with hardware; higher abstractions are typically avoided for performance reasons.	**Supports high-level abstractions** (OOP, templates, etc.) but can incur runtime costs, especially if not carefully optimized.
Error Handling	Uses **Result/Option types** for safe error handling, forcing developers to handle errors explicitly. No exceptions.	Error handling is typically done via **return codes**, which are easy to ignore, leading to unsafe code.	Uses **exceptions** for error handling, which can be complex to manage and introduce performance overhead.
Syntax	**Modern syntax** inspired by C++ and other languages, with a focus on safety and clarity.	**Simple syntax**, but somewhat verbose and outdated (e.g., manual type declarations, lack of	**Complex syntax** due to the large number of features, such as templates, lambdas, and

Feature	Rust	C	C++
		modern features).	overloading, which can be overwhelming for beginners.
Interoperability	**Good interoperability** with C via FFI, but more difficult with C++ due to differences in object models.	**Seamless interoperability** with other C programs, easy to interface with low-level systems and hardware.	Good interoperability with C and assembly, though more complex when integrating with Rust or other languages.
Use Cases	Ideal for **system-level programming** where safety, performance, and concurrency are critical. Great for new systems projects, embedded systems, and web services (e.g., WASM).	Best for **low-level programming**, embedded systems, operating systems, and performance-critical applications.	Suitable for a **wide range of applications**, from game engines and GUI applications to system-level programming and high-performance computing.

4. write c program to handle divide by zero exception with out program termination

```c
#include <stdio.h>

float safeDivide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        printf("Error: Division by zero is not allowed.\n");
        return 0.0;
    }
    return (float)numerator / denominator;
}
int main()
{
    int num1, num2;
    float result;
    printf("Enter numerator: ");
    scanf("%d", &num1);
    printf("Enter denominator: ");
    scanf("%d", &num2);
    result = safeDivide(num1, num2);
    if (num2 != 0)
    {
        printf("Result: %.2f\n", result);
    }
    return 0;
}
```