Contents

1 Memory Safety **2**
- 1.1 Common Memory Safety Issues . . . 2
  - 1.1.1 Buffer Overflows . . . 2
  - 1.1.2 `nullptr` Dereferencing . . . 2
  - 1.1.3 Dangling Pointers . . . 3
  - 1.1.4 Memory Leaks . . . 3
- 1.2 Smart Pointers . . . 3
  - 1.2.1 Unique Pointer . . . 3
  - 1.2.2 Shared Pointer . . . 5
  - 1.2.3 Weak Pointer . . . 5

1 Memory Safety

Memory safety refers to the concept of ensuring that a program does not access memory in an unintended or unsafe manner. This can involve issues such as out-of-bounds access, null pointer dereferencing, or memory leaks, all of which can lead to undefined behavior, crashes, or security vulnerabilities. In C++, memory safety is a critical concern due to the language's low-level features and manual memory management capabilities, which allow developers more control but also more responsibility.

1.1 Common Memory Safety Issues

Before diving into strategies for improving memory safety, let's examine some of the most common issues developers face in C++ programs.

1.1.1 Buffer Overflows

Buffer overflows occur when a program writes past the end of an array (a buffer), corrupting adjacent memory. Theyre a common source of security vulnerabilities, as attackers can exploit them to inject or execute arbitrary code.

overflow.cpp

```cpp
#include <iostream>

int main()
{
    char buffer[10];
    strcpy(buffer, "This is too long for the buffer");
}
```

1.1.2 `nullptr` Dereferencing

Dereferencing a `NULL` pointer causes undefined behavior and can lead to program crashes:

```cpp
int* ptr { nullptr };
*ptr = 10;
```

1.1.3 Dangling Pointers

A dangling pointer is a pointer that references a memory location after the object that it points to has been deallocated. Dereferencing a pointer leads to undefined behavior:

```cpp
int* ptr { new int };
delete ptr;
*ptr = 10;
```

1.1.4 Memory Leaks

Memory leaks occur when a program allocates memory but fails to deallocate it, causing the program to consume ever-increasing amounts of memory. Usually this happens when a call to `delete` does not follow a call to `new`.

1.2 Smart Pointers

Smart pointers are wrapper classes that manage the lifetime of dynamically allocated objects. Unlike raw pointers, which require explicit memory management (using `new` and `delete`), smart pointers automatically handle memory cleanup. Smart pointers are a critical component of RAII (Resource Acquisition Is Initialization), a C++ programming paradigm that ensures resources are properly cleaned up when an object goes out of scope.

C++ offers several types of smart pointers, each designed for different use cases. The most commonly used smart pointers are `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`, all of which are defined in the `<memory>` header.

1.2.1 Unique Pointer

Unique pointers are smart pointers that own an object exclusively. They help to ensure that the resource managed is freed when the pointer goes out of scope.

A unique pointer cannot be copied. Here is a simplified implementation of the unique pointer class:

uniqueptr.h

```
template<class T>
class UniquePtr
{
private:
    T* ptr;

public:
    unique_ptr(): ptr(nullptr) {};
    unique_ptr(T* ptr): ptr(ptr) {}

    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;

    ~unique_ptr()
    {
        delete ptr;
    }

    T* operator->() { return ptr; }
};
```

Here is an example of using `std::unique_ptr`:

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

int main()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(10);
    std::cout << *ptr << std::endl;
}
```

1.2.2 Shared Pointer

A shared pointer allows multiple pointers to share ownership of a resource. The resource is only freed when the last reference is destroyed. This is managed by a reference count, which tracks how many shared pointers are sharing the resource:

shared.cpp

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

class MyClass
{
public:
    MyClass() { std::cout << "constructed" << std::endl; }
    ~MyClass() { std::cout << "destroyed" << std::endl; }
};

int main()
{
    std::shared_ptr<MyClass> ptr { nullptr };
    {
        std::shared_ptr<MyClass> myobj {
            std::make_shared<MyClass>() };
        ptr = myobj;
    }
    std::cout << "left scope" << std::endl;
}
```

1.2.3 Weak Pointer

A `std::weak_ptr` is used to "observe" an object that is managed by a shared pointer without affecting its reference count. Weak pointers can be constructed from shared pointers:

weak.cpp

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

int main()
{
    std::weak_ptr<int> weak;
    std::shared_ptr<int> s;

    {
        std::shared_ptr<int> shared =
           std::make_shared<int>(42);
        weak = shared;

        std::cout << "Inside block: weak expired? "
                  << std::boolalpha << weak.expired() <<
                     "\n";
    }

    std::cout << "Outside block: weak expired? "
              << weak.expired() << "\n";

    if (auto locked = weak.lock())
    {
        std::cout << "Value: " << *locked << "\n";
    } else
    {
        std::cout << "Object no longer exists.\n";
    }
}
```