Contents

1 Type Safety

Type safety refers to the enforcement of correct and consistent use of data types within a program. A language is typically considered type-safe if it prevents operations that are invalid for a given type. In C++, type safety is particularly important because improper use of types can easily result in subtle bugs, crashes, or undefined behavior.

C++ is often described as a statically typed language, meaning type checking is performed at compile time. For example, the following code results in a compile-time error rather than a run-time failure:

```cpp
int x = "forty-two";
```

This behavior prevents many categories of type-related bugs from ever making it into a running program. However, C++ is also a highly flexible language–and this flexibility includes numerous implicit type conversions, some of which can introduce subtle, silent bugs. While this freedom is powerful, it can be dangerous if used carelessly or without full understanding. Consider the following example:

```cpp
#include <iostream>

int main()
{
    int x { -1 };
    unsigned y { 1 };

    std::cout << (x < y) << std::endl;
}
```

Despite what might seem like an intuitive comparison, this program will confidently assert that 1 is less than -1. This happens because when comparing an `int` and an `unsigned`, C++ is forced to reconcile the type difference and does so by turning the signed integer `x` into an unsigned value. Since `x` is negative, the conversion results in a large positive number, which distorts the intended comparison. An even more egregious example:

```cpp
#include <iostream>

int main()
{
    float f = 0.1;
    std::cout << (f == 0.1) << std::endl;
}
```

This program, surprisingly, prints 0, indicating that `f` does not equal the value we assigned to it just a line before. The reason for this discrepancy lies in a subtle detail: `0.1` is a `double` literal, which holds more precision than the `float` `f`. When the `double` value is assigned to the `float`, it loses precision, causing the comparison to fail. For a deeper explanation of this issue, feel free to check out my talk: "*Messing with Floating Point.*"
A type-safe program aims to eliminate these classes of bugs by ensuring that:

- Types are not implicitly converted in ways that compromise correctness.
- Operations are performed only on types that support them.
- The program behaves predictably with respect to the types involved.

In the sections that follow, well explore type-safe programming in C++ and discuss the tools, practices, and patterns we can use to write safer, more predictable code.

1.1 Strong Type Definitions

A **strong typedef** creates a new, distinct type at the language level. This can help enforce type safety by ensuring that different types–even if they share the same underlying representation–can't be mixed up by accident. Consider the following example where we use a function to calculate the final price of a sale:

salestax.cpp

```cpp
#include <iostream>

double final_price(double p, double t)
{
    return (1.0 + t) * p;
}

int main()
{
    double p = 5499.99;
    double t = 0.06;
    std::cout << final_price(t, p) << std::endl;
}
```

Terminal

```
$ clang++ -std=c++23 salestax.cpp
$ ./a.out
330.059
```

The user of this program will no doubt be delighted to discover that their $5500 purchase will only cost them $330. The cause of this error is simple–though easy to miss: We passed the parameters in the wrong order to `final_price()`. C++ gives us the tools to prevent this kind of mix-up by using the type system more intentionally. We can define strong types that make it impossible to confuse parameters with the same underlying type but different meaning.

A naive attempt at a solution might involve introducing type aliases for prices and tax rates, hoping to prevent ourselves from accidentally mixing them up:

salestax2.cpp

```cpp
#include <iostream>

using Price = double;
using TaxRate = double;

Price final_price(Price p, TaxRate t)
{
    return (1.0 + t) * p;
}

int main()
{
    Price p = 5499.99;
    TaxRate t = 0.06;
    std::cout << final_price(t, p) << std::endl;
}
```

However, this solution offers little benefit beyond improved readability. The compiler still treats `Price` and `TaxRate` as identical types, so our bug slips through unnoticed. To make the compiler enforce the distinction, we must define entirely new types–rather than simple aliases:

salestax3.cpp

```cpp
#include <iostream>

struct Price { double value; };
struct TaxRate { double value; };

Price final_price(Price p, TaxRate t)
{
    return { (1.0 + t.value) * p.value };
}

int main()
```

```cpp
{
    Price p { 5499.99 };
    TaxRate t { 0.06 };
    std::cout << final_price(t, p).value << std::endl;
}
```

Terminal

```
$ clang++ -std=c++23 salestax3.cpp
15:15: error: no matching function for call to 'final_price'
   15 |  std::cout << final_price(t, p).value << std::endl;
```

It appears the compiler has finally caught on to our antics, and we'll be forced to pay full price. Thats a win for correctness. However, perhaps we can do better still. A codebase littered with `.value` member accesses isnt exactly one Id be thrilled to work in. Perhaps we can clean this up using templates:

safetype.h

```cpp
template<class T, class Tag>
class SafeType
{
private:
    T value;

public:
    SafeType(T t): value(t) {}

    template<class U, class UTag>
    SafeType(const SafeType<U, UTag>&) = delete;

    operator T() const { return value; }
};
```

The templated class above allows us to define strong types that the compiler treats as distinct, while still allowing implicit conversion to the underlying type when needed. This strikes a practical balance: we get safety from mismatched arguments without giving up ergonomics.

salestax4.cpp

```cpp
#include "safetype.h"
#include <iostream>

using Price = SafeType<double, struct Price_>;
using TaxRate = SafeType<double, struct TaxRate_>;

Price final_price(Price p, TaxRate t)
{
    return { (1.0 + t) * p };
}

int main()
{
    Price p(5499.99);
    TaxRate t(0.06);
    std::cout << double(final_price(p, t)) << std::endl;
}
```

We can take this idea one step further by inheriting from the `SafeType` template instead of using a type alias. This approach tends to produce cleaner and more informative error messages when something goes wrong:

salestax5.cpp

```cpp
#include "safetype.h"
#include <iostream>

struct Price: SafeType<double, Price> { using
    SafeType::SafeType; };
struct TaxRate: SafeType<double, TaxRate> { using
    SafeType::SafeType; };

Price final_price(Price p, TaxRate t)
{
    return { (1.0 + t) * p };
}

int main()
{
    Price p(5499.99);
    TaxRate t(0.06);
    std::cout << double(final_price(t, p)) << std::endl;
}
```

1.1.1 The Type-Safe `enum`

There is an even simpler solution that can be used to create strong type definitions to distinguish between underlying integral types. This is achieved through the use of `enum class`:

```cpp
#include <iostream>

enum class Apple: uint32_t {};
enum class Orange: uint32_t {};

int main()
{
    Apple a { 10 };
    Orange b { 20 };
    // error, cannot compare Apples and Oranges
    std::cout << (a < b) << std::endl;
}
```

In this example, `Apple` and `Orange` are both defined as `enum class` types, backed by `uint32_t`. The key difference here is that `enum class` creates distinct types, preventing the comparison of these two, even though they share the same underlying type. This ensures type safety and avoids unintended type mixing, which can be a common issue in less strict systems.

1.2 Type Punning

Type punning refers to treating memory that holds some data type as if it holds a differnet data type. This can be achived in a number of ways. Most simply is the `union` construct:

```cpp
#include <iostream>

union FloatInt
{
    float f;
    int i;
};

int main()
{
    FloatInt pun;
    pun.f = 3.14;
    std::cout << pun.i << std::endl;
}
```

In this example, `pun.i` and `pun.f` occupy the same memory. Another way to achieve type punning is to use pointer casting:

```cpp
#include <iostream>

int main()
{
    int x { 400 };
    double d { *(double*)(&x) };
    std::cout << d << std::endl;
}
```

The above exmaple is generally terrible, not least because it reads four bytes beyond the indended value of `x`.

1.3 Type Casting

C++ offers several casting operators to convert variables from one type to another. These casts provide more control and type safety compared to C-style casts.

1.3.1 `static_cast`

`static_cast` performs a non-polymorphic conversion between types. It's checked at compile time (hence static), and is generally safe for conversions like `int` to `double`.

```cpp
int x { 10 };
double d { static_cast<double>(x) };
```

1.3.2 `dynamic_cast`

`dynamic_cast` is used for run-time polymorphic type conversions, specifically with inheritance. It checks the type of the object at run-time and returns a valid pointer if the conversion is possible, else `nullptr`.

```cpp
class Base
{
    virtual void foo() {}
};

class Derived: public Base {};

int main()
{
    Base* b = new Derived();
    Derived* d = dynamic_cast<Derived*>(b);
}
```

1.3.3 `const_cast`

`const_cast` modifies the constness or volatility of a variable:

```cpp
const int x { 20 };
int* non_const { const_cast<int*>(&c) };
```

1.3.4 `reinterpret_cast`

`reinterpret_cast` is the most powerful and potentially dangerous cast. It reinterprets the underlying bit pattern as a different type:

```cpp
int* ptr { new int(30) };
unsigned address { reinterpret_cast<unsigned>(ptr) };
```

2 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.

2.1 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.

2.1.1 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");
}
```

2.1.2 `nullptr` Dereferencing

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

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

2.1.3 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;
```

2.1.4 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`.

2.2 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.

2.2.1 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

```cpp
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`:

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

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

2.2.2 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; }
};

```

```cpp
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;
}
```

2.2.3 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";
    }
}
```