

Contents

1	Type Safety	2
1.1	Strong Type Definitions	4
1.1.1	The Type-Safe <code>enum</code>	8
1.2	Type Punning	9
1.3	Type Casting	10
1.3.1	<code>static_cast</code>	10
1.3.2	<code>dynamic_cast</code>	10
1.3.3	<code>const_cast</code>	11

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:

```
1 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:

```
1 #include <iostream>
2
3 int main()
4 {
5     int x { -1 };
6     unsigned y { 1 };
7
8     std::cout << (x < y) << std::endl;
9 }
```

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:

```
1 #include <iostream>
2
3 int main()
4 {
5     float f = 0.1;
6     std::cout << (f == 0.1) << std::endl;
7 }
```

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, we'll 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

1 #include <iostream>
2
3 double final_price(double p, double t)
4 {
5     return (1.0 + t) * p;
6 }
7
8 int main()
9 {
10     double p = 5499.99;
11     double t = 0.06;
12     std::cout << final_price(t, p) << std::endl;
13 }
```

```
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

```
1 #include <iostream>
2
3 using Price = double;
4 using TaxRate = double;
5
6 Price final_price(Price p, TaxRate t)
7 {
8     return (1.0 + t) * p;
9 }
10
11 int main()
12 {
13     Price p = 5499.99;
14     TaxRate t = 0.06;
15     std::cout << final_price(t, p) << std::endl;
16 }
```

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

```
1 #include <iostream>
2
3 struct Price { double value; };
4 struct TaxRate { double value; };
5
6 Price final_price(Price p, TaxRate t)
7 {
8     return { (1.0 + t.value) * p.value };
9 }
10
11 int main()
```

```

12 {
13     Price p { 5499.99 };
14     TaxRate t { 0.06 };
15     std::cout << final_price(t, p).value << std::endl;
16 }

```

Terminal

```

$ clang++ -std=c++23 saletax3.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. That's a win for correctness. However, perhaps we can do better still. A codebase littered with `.value` member accesses isn't exactly one I'd be thrilled to work in. Perhaps we can clean this up using templates:

safetype.h

```

1 template<class T, class Tag>
2 class SafeType
3 {
4 private:
5     T value;
6
7 public:
8     SafeType(T t): value(t) {}
9
10    template<class U, class UTag>
11    SafeType(const SafeType<U, UTag>&) = delete;
12
13    operator T() const { return value; }
14 };

```

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

```
1 #include "safetype.h"
2 #include <iostream>
3
4 using Price = SafeType<double, struct Price_>;
5 using TaxRate = SafeType<double, struct TaxRate_>;
6
7 Price final_price(Price p, TaxRate t)
8 {
9     return { (1.0 + t) * p };
10 }
11
12 int main()
13 {
14     Price p(5499.99);
15     TaxRate t(0.06);
16     std::cout << double(final_price(p, t)) << std::endl;
17 }
```

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

```
1 #include "safetype.h"
2 #include <iostream>
3
4 struct Price: SafeType<double, Price> { using
    SafeType::SafeType; };
5 struct TaxRate: SafeType<double, TaxRate> { using
    SafeType::SafeType; };
6
7 Price final_price(Price p, TaxRate t)
8 {
```

```

9     return { (1.0 + t) * p };
10 }
11
12 int main()
13 {
14     Price p(5499.99);
15     TaxRate t(0.06);
16     std::cout << double(final_price(t, p)) << std::endl;
17 }

```

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`:

```

1 #include <iostream>
2
3 enum class Apple: uint32_t {};
4 enum class Orange: uint32_t {};
5
6 int main()
7 {
8     Apple a { 10 };
9     Orange b { 20 };
10    // error, cannot compare Apples and Oranges
11    std::cout << (a < b) << std::endl;
12 }

```

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 different data type. This can be achieved in a number of ways. Most simply is the `union` construct:

```
1 #include <iostream>
2
3 union FloatInt
4 {
5     float f;
6     int i;
7 };
8
9 int main()
10 {
11     FloatInt pun;
12     pun.f = 3.14;
13     std::cout << pun.i << std::endl;
14 }
```

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

```
1 #include <iostream>
2
3 int main()
4 {
5     int x { 400 };
6     double d { *(double*)&x };
7     std::cout << d << std::endl;
8 }
```

The above example is generally terrible, not least because it reads four bytes beyond the intended 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`.

```
1 int x { 10 };
2 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`.

```
1 class Base
2 {
3     virtual void foo() {}
4 };
5
6 class Derived: public Base {};
7
8 int main()
9 {
10     Base* b = new Derived();
11     Derived* d = dynamic_cast<Derived*>(b);
12 }
```

1.3.3 const_cast

const_cast modifies the constness or volatility of a variable:

```
1 const int x { 20 };  
2 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:

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