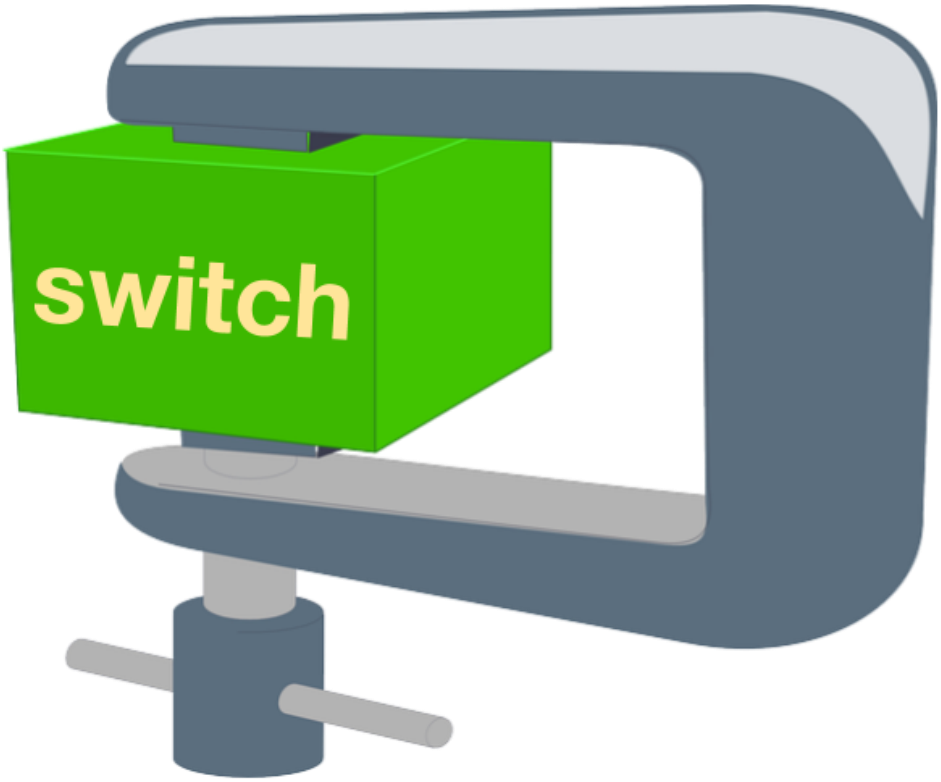
BASIC CONSTRUCTS

- **How to Flatten Out A Nested Switch Statement**
- **Getting Along With The Comma Operator in C++**
- **How to Design Early Returns in C++ (Based on Procedural Programming**

Fluent {C++}

How to Flatten Out A Nested Switch Statement

With my team we've recently come across an annoying switch nested in another switch statement, and I want to show a solution for flattening out this sort of structure.



Motivation

Let's consider two enums representing the size and color of a shirt. While I don't work in the clothing industry, using a simple example by stripping away all the domain specifics allows to focus on the C++ technique here.

The shirts come in two colors and three sizes:

```
enum class Color
{
    Red,
    Blue
};
```

```
enum class Size
{
    Small,
    Medium,
    Large
};
```

We do a specific treatment for each of the combination of color and size of a shirt.

Expressing this with switches gives something like this:

```
switch (color)
{
    case Color::Red:
    {
        switch (size)
        {
            case Size::Small:
            {
                // code for color red and size Small
                break;
            }
            case Size::Medium:
            {
                // code for color red and size Medium
                break;
            }
            case Size::Large:
            {
                // code for color red and size Large
                break;
            }
            default:
            {
                throw WrongEnumValues();
            }
        }
    }
    case Color::Blue:
    {
        switch (size)
        {
            case Size::Small:
            {
                // code for color blue and size Small
                break;
            }
            case Size::Medium:
            {
                // code for color blue and size Medium
                break;
            }
            case Size::Large:
```

```
                {
                    // code for color blue and size Large
                    break;
                }
                default:
                {
                    throw WrongEnumValues();
                }
            }
        }
}
```

Several things are damaging the expressiveness of this piece of code:

- it's lengthy but without containing a lot of information,
- the associated colors and sizes are far apart from one another: for example the `case Size::Large` within the `case Color::Red` is closer to the `case Color::Blue` in terms of lines of code than from the `case Color::Red` to which it belongs.
- this design doesn't scale: imagine that a third enum was involved. The code would then become even harder to read.

To make this code more expressive, I'm going to show how to flatten the double switch into a single one.

Collapsing a switch

Here's an easy way to do this: creating an new enum that represents all the combinations of the other enums, and use it in the switch statement.

Let's do it manually once, and then write a generic code to do it for us.

Here is the enum representing the combinations:

```
enum class Color_Size
{
    Blue_Small,
    Blue_Medium,
    Blue_Large,
    Red_Small,
    Red_Medium,
    Red_Large
};
```

The ugly double switch can be encapsulated into a function that does the mapping between the original enum and this new one:

```
constexpr Color_Size combineEnums(Color color, Size size)
{
    switch (color)
    {
        case Color::Red:
        {
            switch (size)
            {
                case Size::Small: return Color_Size::Blue_Small;
                case Size::Medium: return Color_Size::Blue_Medium;
                case Size::Large: return Color_Size::Blue_Large;
                default: throw WrongEnumValues();
            }
        }
        case Color::Blue:
        {
            switch (size)
            {
                case Size::Small: return Color_Size::Red_Small;
                case Size::Medium: return Color_Size::Red_Medium;
                case Size::Large: return Color_Size::Red_Large;
                default: throw WrongEnumValues();
            }
        }
    }
}
```

And then we can do a single switch statement on the combination of values. The key for this to work is that the `combineEnums` function is **constexpr**, so its return value can be put into a switch statement:

```
switch (combineEnums(color, size))
{
    case combineEnums(Color::Red, Size::Small):
    {
        // code for color red and size Small
        break;
    }
    case combineEnums(Color::Red, Size::Medium):
    {
        // code for color red and size Medium
        break;
    }
    case combineEnums(Color::Red, Size::Large):
    {
        // code for color red and size Large
        break;
```

```
    }
    case combineEnums(Color::Blue, Size::Small):
    {
        // code for color blue and size Small
        break;
    }
    case combineEnums(Color::Blue, Size::Medium):
    {
        // code for color blue and size Medium
        break;
    }
    case combineEnums(Color::Blue, Size::Large):
    {
        // code for color blue and size Large
        break;
    }
    default:
    {
        throw WrongEnumValues();
    }
}
```

You'll note that a constexpr function can throw exceptions. While this seems strange at first, it's logical because a constexpr function can also be called at runtime. And if it ever tries to throw at compile time, the program doesn't compile. All this is very well explained in Dietmar Kühl's Constant Fun talk at CppCon on constexpr.

Although the switch statement has been flattened out, there is a lot of code that could be automated here.

Combining the enums automatically

Prerequisite: The generic solution I propose is based on one prerequisite: that the enums have all an extra last element with a consistent name, say "End_", and that its value is not customized (as in `End_ = 42`). We could choose any other name, but I like "End_" because it has the same semantics of "one after the last one" as in the STL. I need this to manipulate the enums together (if you can think of a way to fill the same need without the End_, the comments section is all yours).

So our two enums become:

```
enum class Color
{
    Red,
    Blue,
    End_
};

enum class Size
{
    Small,
    Medium,
    Large,
    End_
};
```

The idea is now to give a unique value for each association of enum values. The most compact (and, in my opinion, the most natural) way to do this is by using the following formula:

combinedValue = (Color value) + (numbers of possible color values) * (Size value)

One way to view this formula is that for each value of the `Size` enum, there are as many values as there are possible `Color`s.

The formula manipulates enum values like numeric values. To do this, we throw away all the type safety brought by the enum classes:

```
template<typename Enum>
constexpr size_t enumValue(Enum e)
{
    return static_cast<size_t>(e);
}
```

This code snippet is supposed to make you feel *very* uneasy. But worry not, we'll put all the type safety back in just a moment.

And here is how to get the number of possible values of an enum:

```
template<typename Enum>
constexpr size_t enumSize()
{
    return enumValue(Enum::End_);
}
```

Hence the need for `End_`.

And here is the implementation of the formula:

```
template<typename Enum1, typename Enum2>
constexpr size_t combineEnums(Enum1 e1, Enum2 e2)
{
    return enumValue(e1) + enumSize<Enum1>() * enumValue(e2);
}
```

which is still constexpr, to be able to fit into the cases of a switch statement.

Putting type safety back

Now have a look at this example of usage. See anything wrong?

```
switch (combineEnums(color, size))
switch (combineEnums(color, size))
{
    case combineEnums(Color::Red, Size::Small):
    {
        // code for color red and size Small
        break;
    }
    case combineEnums(Color::Red, Size::Medium):
    {
        // code for color red and size Medium
        break;
    }
    case combineEnums(Size::Small, Size::Large):
    {
        // code for color red and size Large
        break;
    }
    case combineEnums(Color::Blue, Size::Small):
    {
        // code for color blue and size Small
        break;
    }
    case combineEnums(Color::Blue, Size::Medium):
    {
        // code for color blue and size Medium
        break;
    }
    case combineEnums(Color::Blue, Size::Large):
    {
        // code for color blue and size Large
        break;
    }
    default:
```

```
    {
        throw WrongEnumValues();
    }
}
```

There is a bug in the third case:

```
case combineEnums(Size::Small, Size::Large):
```

This could happen because I've thrown away type safety a little earlier. I really asked for this one.

A way to put type safety back in place is to **add typing** to the `combineEnums` function. To do this I'm going to:

- transform the `combineEnums` function into a function object
- move the template types corresponding to the enums over to the object rather than the function
- use the same object instance in the whole switch statement.

So for a start, here is the function's code packed into an object:

```
template<typename Enum1, typename Enum2>
struct CombineEnums
{
    constexpr size_t operator()(Enum1 e1, Enum2 e2)
    {
        return enumValue(e1) * enumSize<Enum2>() + enumValue(e2);
    }
};
```

Then we construct the object with the right enum types before the switch statement:

```
CombineEnums<Color, Size> combineEnums;
switch (combineEnums(color, size))
{
    case combineEnums(Color::Red, Size::Small):
    {
        ....
```

and using the wrong enum in a case becomes a compilation error:

```
error: no match for call to '(CombineEnum<Color, Size>) (Size,
Size)'
```

Safety's back.

Going generic

EDIT: I thought that a simple recursion on variadic templates was enough to make this technique work on any number of enums. But as reddit user /u/minirop pointed out with a revealing example, I was wrong. The presented implementation only works for two enums. Therefore I'll leave this section empty and will re-work the implementation to make it more generic. This will be the topic of a later post.

Stepping back

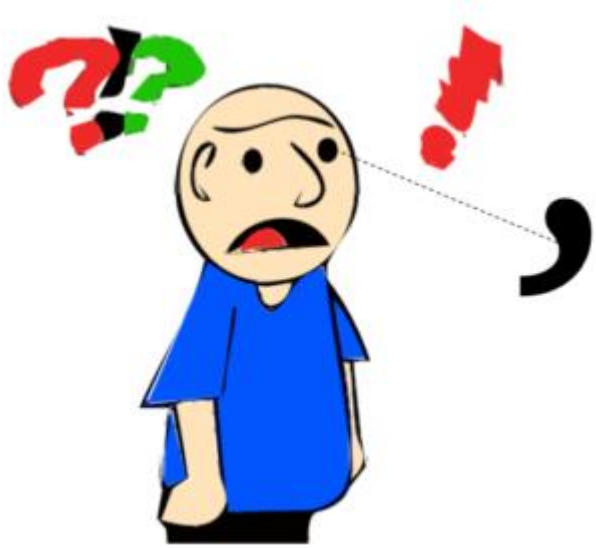
I've found this technique efficient to flatten out switch statements and to bring associated values together in the cases. This really improves code readability.

However, it may not be the right choice for every situation (what is, really). For instance this technique doesn't let you have a case covering a given value of Color for all possible values of Size.

Also, switches on enums often raise the question of hidden polymorphism: wouldn't these enums be better off refactored into types? In this case the need to route on several types draws the code into multiple dispatch, which C++ doesn't support natively. One solution for this is the (much criticized) visitor pattern.

But enums are there for a reason. And when switches start nesting into one another, this technique for ironing them out comes in handy.

Getting Along With The Comma Operator in C++



The comma operator is a curious operator and seldom used, but it happens to encounter it in code. And sometimes by mistake. Such encounters can give a hard time understanding the code.

For this reason it is useful to know what it does, and what it doesn't do.

This article is not made to show how to put `operator,` at the center of your designs, but rather help you get along with it when you find it in code. At some point this will save you some question marks popping up above your head when you read code.

A comma operator?

Yes, there is such as thing as `operator,` in C++, just as much as there is `operator+` or `operator*`.

It has a built-in implementation on all combination of two types, that does the following:

- first evaluate the left hand side expression,
- then evaluate the right hand side expression,
- finally return the result of the evaluation of the right hand side expression.

For example consider the following expression:

```
f(), g()
```

with f and g being two functions. Here the compiler calls f, then calls g, then returns the value returned by g.

Note that even though the return value of f is discarded (meaning that it isn't used) like any other temporary object it persists until the end of the execution of the enclosing statement.

`operator,` is for example seen in the for-loop expressions that maintain several counter variables:

```
for (...; ...; ++i, ++j)
```

Don't overload it (in particular before C++17)

Like its operator siblings, `operator,` can be overloaded for a particular combination of two types.

But as Scott Meyers explains in Item 7 of More Effective C++, you don't want to do that. The reason is that when you overload an operator it is considered like a normal function in terms of order of evaluation. That is, the order of evaluation is unspecified.

And for the operators `&&`, `||` and `,` the order of evaluation is part of their **semantics**. Changing this order kind of breaks those semantics and makes the code even more confusing than the existence of a comma operator in the first place.

This has changed in C++17. Indeed, in C++17 the order of custom operators `&&`, `||` and `,` is the same as the one of built-in types (so left hand side first). So in theory you could more easily override `operator,` in C++17. But I have yet to see a case where it *makes sense* to overload `operator,`.

Note that even with the order of evaluation of C++17, you still wouldn't want to override `operator&&` and `operator||`, because their built-in versions have a short-circuit behaviour: they don't evaluate the second paramater if the first one evaluates to `false` (resp. `true`) for `operator&&` (resp. `operator||`). And the custom versions don't have this short-circuiting behaviour, even in C++17.

Code that doesn't look like what it does

Here is an interesting case that was pointed out to me by my colleague Aadam. Thanks for bringing this up Aadam! I've replaced all domain types and values with `int`s for this example:

```
int sum(int x, int y)
{
    return x + y;
}

int main()
{
    int x = 4;
    int y = 0;
    int z = 0;
    z = sum(x, y);

    std::cout << z << '\n';
}
```

Can you predict the output of this code?

```
Click To Expand Code


MS DOS

4
```

Wasn't hard, right?

Now an unfortunate refactoring introduces a typo in the code: note how the call to the function `sum` has been removed but the second parameter was left by mistake:

```
int sum(int x, int y)
{
    return x + y;
}

int main()
{
    int x = 4;
    int y = 0;
    int z = 0;
    z = x, y;

    std::cout << z << '\n';
}
```

Now can you predict the output of this code?

```
Click To Expand Code
```

```
MS DOS
```

```
4
```

You read it right, it is 4, not 0.

This baffled us: this comma operator should return the right hand side value, so 0, right?

The explanation lies in the precedence rules of operators: **comma is the last operator in terms of precedence**. So in particular it comes after... `operator=` ! Therefore, the assignement statement on z should be read `z=x`, and only after this`operator,` takes the result of that and `y`.

With all this, you're now more equipped to deal with this curious operator next time you encounter it.

How to Design Early Returns in C++ (Based on Procedural Programming)

Travelling back from ACCU conference a couple of weeks ago, one of the insights that I've brought back with me is from Kevlin Henney's talk Procedural Programming: It's Back? It Never Went Away. It's surprisingly simple but surprisingly insightful, and it has to do with early return statements.

Early return statements are controversial in the programming community and, often, deciding whether a given early return is OK comes down to listening to how your gut is feeling about it.

In his presentation about how procedural programming is not just a memory from the past, Kevlin gives a guideline that will help our brain also take part in the decision process of judging an early return statement.

Consider the following two pieces of code that determine whether a year is a leap year:

Code #1:

```
bool isLeapYear(int year)
{
    if (year % 400 == 0)
    {
        return true;
    }
    else if (year % 100 == 0)
    {
        return false;
    }
    else if (year % 4 == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
```

```
}
```

Code #2:

```
bool isLeapYear(int year)
{
    if (year % 400 == 0)
    {
        return true;
    }
    if (year % 100 == 0)
    {
        return false;
    }
    if (year % 4 == 0)
    {
        return true;
    }

    return false;
}
```

The difference between those two pieces of code is that Code #1 is based on `if` / `else if` / `else` structure, while Code #2 has several `if`s followed by a `return`.

Now the question is: which of the two pieces of code is the most readable?

You may think that it is Code #2. After all it has less characters, and even less nesting. In fact, even clang and the LLVM project consider that Code #2 is more readable. Indeed, they even implemented a refactoring in clang-tidy called readability-else-after-return, that removes `else`s after the interruptions of the control flow – such as `return`.

But Kevlin Henney stands for Code #1 when it comes to readability, and draws his argument from procedural programming.

So what makes Code #1 more readable?

Leave your reader on a need-to-know basis

Essentially, the argument for Code #1 is that **you need to know less** to understand the structure of the code.

Indeed, if we fold away the contents of the if statements, Code #1 becomes this:

```
bool isLeapYear(int year)
{
    if (year % 400 == 0) { ... }
    else if (year % 100 == 0) { ... }
    else if (year % 4 == 0) { ... }
    else { ... }
}
```

The structure of the code is very clear. There are 4 different paths based on the `year`, they're independent from each other, and each path will determine the boolean result of the function (if it doesn't throw an exception).

Now let's see how Code #2 looks like when we fold away the if statements:

```
bool isLeapYear(int year)
{
    if (year % 400 == 0) { ... }
    if (year % 100 == 0) { ... }
    if (year % 4 == 0) { ... }

    return false;
}
```

And now we know much less. Do the if statements contain a `return`? Maybe.

Do they depend on each other? Potentially.

Do some of them rely on the last `return false` of the function? Can't tell.

With Code #2, **you need to look inside of the if statement** to understand the structure of the function. For that reason, Code #1 requires a reader to know less to understand the structure. It gives away information more easily than Code #2.

I think this is an interesting angle to look at code expressiveness: how much you need to know to understand the structure of a piece of code. The more you need to know, the less expressive.

In the leap year example, the if blocks are one-line return statements, so you probably wouldn't fold them away anyway, or maybe just mentally. But the differences grows when the code gets bigger.

The Double Responsibility of `return`

Here is another way to compare Code #1 and Code #2. In C++, as well as in other languages, the `return` keyword has two responsibilities:

- interrupting control flow,
- yielding a value.

One could argue that this violates the Single Responsibility Principle, that stipulates that every component in code should have exactly One responsibility.

Note, however, that this is not the case of every language. For example Fortran, quoted by Kevlin, uses two different mechanisms to fulfil those two responsibilities (`RETURN` only interrupts the control flow, while assigning to the name of the function yields a value).

Now if we focus on the second role of `return`, that is yielding a value, and rewrite our function in pseudo-code to only show that value when possible, Code #1 becomes:

```
bool isLeapYear(int year)
{
    if (year % 400 == 0)
    {
        true
    }
    else if (year % 100 == 0)
    {
        false
    }
```

```
    else if (year % 4 == 0)
    {
        true
    }
    else
    {
        false
    }
}
```

We see that we're only using One responsibility of `return`: yielding a value. Code #1 helps `return` respect the SRP.

Now if we do the same thing with Code #2, we can't get rid of the interrupting responsibility of `return`:

```
bool isLeapYear(int year)
{
    if (year % 400 == 0)
    {
        return true;
    }
    if (year % 100 == 0)
    {
        return false;
    }
    if (year % 4 == 0)
    {
        return true;
    }

    false
}
```

Only the last `return` has only responsibility (yielding a value). The other `return`s mix their two responsibilities: interrupting the control flow and returning a value. And mixing responsibilities is not a good thing in programming.

English food and food for thought

This was one of the insights that I took away when I attended the ACCU conference 2018, and that I wanted to share with you. It's a simple example that wraps a deep reflection on several fundamental aspects of programming. If you weren't at ACCU to taste the English food, here is at least some food for thought.

Thanks to Kevlin for reviewing this article. If you'd like to watch his ACCU conference talk in full, here it is.