CheckedInt: A policy-based range-checked integer

Hubert Matthews, hubert@oxyware.com

Recently, I wanted a short example to show the canonical form for operators on value classes. In other words, I wanted to show how postincrement should be related to preincrement, how `operator+=` and `operator+` fit together, which functions should be members and which not, and so on. Having also been reading Alexandrescu's excellent book, I decided to make this exercise a little more interesting (for me and for the students) by incorporating something about policies and generic programming. What came out was a small range-checked integer type called `CheckedInt`. Although nothing remarkable, it turns out to be both flexible and useful, and something that in retrospect I could have used myself on several occasions.

For those who are not so familiar with operators, this class shows how all of the arithmetic operators can (or maybe even should) be implemented in terms of one fundamental operation: `operator+=`. This ensures consistency between operators, thereby avoiding potential surprising arithmetic inconsistencies. (For reasons of space, I show only the addition-based operations. Implementation of the others is left, in time-honoured fashion, to you, Gentle Reader™.)

For those already familiar with operators, the policy aspect is more interesting. What should happen when you try to take a range-checked integer or enum out of its defined range or even just modify it? For our range-checked integer, a number of possibilities sprang to mind:

- ?? allow silent overflow
- ?? throw an exception
- ?? saturate at the limit value
- ?? saturate at the limit and log the event
- ?? wrap around using modular arithmetic
- ?? log the event for debugging purposes
- ?? etc.

This little class template allows us to choose which behaviour we want by means of a policy class. Allowing silent overflow is the default for integers so there's no need to write a class for that. Throwing an exception when straying from the promised range is possibly indicative of a programming error. Saturating at the limit could be useful for a digital volume control; one that sticks tenaciously to 10 when you try to set it to 11. And wrapping around is very useful when dealing with ring buffers, dates, etc.

This is a simple example of feature-driven modelling and domain analysis, as described in "Generative Programming" and "Multi-Paradigm Design for C++" where families of types are created with variations described in policies.

So, here's the abbreviated code:

```
template <int low, int high>
class OutOfBoundsThrower {
```

```
public:
	static int RangeCheck(int newVal) {
		if (newVal < low || newVal > high)
			throw std::out_of_range(“RangeCheck failed”);
		return newVal;
	}
};

template <int low, int high>
class ModularArithmetic {
public:
	static int RangeCheck(int newVal) {
		while (newVal > high)
			newVal -= high - low;
		while (newVal < low)
			newVal += high - low;
		return newVal;
	}
};

template <int low, int high>
class SaturatedArithmetic {
public:
	static int RangeCheck(int newVal) {
		if (newVal > high)
			newVal = high;
		else if (newVal < low)
			newVal = low;
		return newVal;
	}
};

template <int low, int high,
	   template <int, int> class ValueChecker = OutOfBoundsThrower>
class CheckedInt : protected ValueChecker<low, hi gh> {
	int value;
public:
	explicit CheckedInt(int i = low) : value(RangeCheck(i)) {}

	CheckedInt & operator+=(int incr) {
		value = RangeCheck(value + incr);
		return *this;
	}
	CheckedInt & operator++() {
		*this += 1;
		return *this;
	}
	const CheckedInt operator++(int) {
		CheckedInt temp(*this);
		++*this;
		return temp;
	}
	CheckedInt & operator-=(int incr) {
		*this += - incr;
		return *this;
	}
	operator int() const { return value; }

	CheckedInt & operator=(int i) {
		value = RangeCheck(i);
		return *this;
```

```
	}
	const CheckedInt operator+(const CheckedInt & other) const {
		return CheckedInt(*this) += other;
	}
};
```

Construction and member functions

Note that the constructor is, like most single argument constructors, marked as explicit. This is to avoid implicit conversions that muddy the type system. Consider what would happen with `CheckedInt<0,10>(5) + 27` if `27` could be explicitly converted. What should its template parameters be? Should it throw an exception? An explicit constructor avoids these problems and forces us to state what we want to happen. The explicit nature of object creation is particularly useful when we wish to constrain the underlying int to a given range as we do not want to create erroneous values. Some might bemoan the inability to `write CheckedInt<0,10> ci = 5;` but I think that safety is more important than ease this time. Choosing low as the default parameter is purely arbitrary and it is arguable that we should force the user to give an initial value anyway.

When going in the opposite direction, i.e. from a `CheckedInt` to an int, there is no danger of breaking any constraints so we can safely use a user-defined conversion – `operator int()` – so that `CheckedInt` appears in a read-only context to behave like an int. This allows us to use all of the existing infrastructure for ints such as `operator<<`, `operator==`, `operator<`, etc. We can now do things like `CheckedInt<0,10>(5) + 27` with impunity and no fear of exceptions.

One small fly swims in the ointment of `operator+=`. There is the possibility that the expression `value + incr` might overflow causing undefined behaviour. This would cause an unexpected problem with saturated arithmetic if someone tried to add a very large number to an instance that was already at its upper limit. Alternatives implementations are possible but more complex.

The more astute of you might have noticed that `operator+` is unusual: it is a member and it is const. The normal advice is to make `operator+` a non-member to allow for implicit conversion of the left-hand operand. However, since we have specifically disallowed that conversion there is no reason not to make it a member and save ourselves a lot of typing! We also return a const value to prevent modification of a temporary whilst still allowing it to be bound to a reference.

Templates versus object-oriented interfaces

An interesting difference in style arises with generic programming rather than a more traditional object-oriented approach. With O-O, one usually ends up with an interface that is the union of all of the sub-interfaces, whereas with a templated version the interface is usually minimal and the intersection of features. This is primarily because with an O-O interface you can combine only those things that you design *a priori* to be combinable, i.e. they must implement all of the stated interface, which can lead to a lot of clutter and "just in case" methods. With templates, you can combine anything that works *a posteriori*. Thus, templates provide compile-time signature-based polymorphism in a manner more reminiscent of Smalltalk than the "one size fits all" of Java interfaces or C++ abstract base classes.

Inheritance versus delegation
Here I have inherited from the policy class rather than delegating to it. Altering the class to use delegation instead:

```
template <int low, int high,
          class VC = OutOfBoundsThrower<low,high> >
class CheckedInt {
public:
      explicit CheckedInt(int i = low) : value(VC::RangeCheck(i)) {}
```

moves us towards a traits-style approach, which some might consider to be cleaner. It is also more digestible by older compilers. In this case because the policy has no state of its own – it is just a wrapper for a function – there is little to choose between the two approaches. The ValueChecker in effect is a compile-time functor analogous to a combination of `bind2nd(), logical_or(), less<int>()` and `greater<int>()`.

Legacy compilers and binding-time issues
Those of us who have to tiptoe around non-standard or ancient compilers will know that template template parameters are off limits. So, how can we adapt `CheckedInt` to be usable? One way is to pass low and high to the `RangeCheck` function at run-time. This has the nice effect of making `ValueChecker` a non-templated class and thereby eliminating some of the compilation problems. Another would be to have a static member of the class that held a pointer to a free function to do the range check. This implementation would also allow the policy to be changed at run-time, turning the class into a classic run-time version of Strategy pattern rather than a compile-time version.

What we are doing is making binding-time choices. By delaying binding from compile time to run time we trade efficiency for the ability to use simpler constructs. We can even change the parameters, so that we could alter the valid range of an object. Whether we wish to do this depends on requirements. As more programmers begin to understand the parallels between different C++ mechanisms and as compilers get better, I believe that we will see binding time become a major design topic, leading people into both feature-driven modelling and domain analysis.

Extensions and additions
Possible extensions include making the underlying type a template parameter, as well as extending the `RangeCheck` function to take the original value as a run-time parameter as well. This would allow us to implement propagating NaN (not a number) behaviour, where if the new value is outside the range we set the value to an out-of-bounds value and keep it there. This is a little bit like the effect of floating-point NaNs which propagate "NaNness" into the results of any calculation.

Summary
I hope this little class template is both useful and instructive. It raises a number of common design issues – relationships between operators, implicit v. explicit conversions – and some others – binding times, policies, implicit v. explicit interfaces, etc – that are less widespread but which I believe will become increasingly common with time. If anyone uses `CheckedInt`, particularly with policies other than these, I would be most interested to hear your experiences.

Acknowledgements
My thanks go to Kevlin Henney and Andrei Alexandrescu for comments on this article.

Bibiliography
Generative Programming, K Czarnecki & UW Eisenecker, Addison-Wesley, 2000
Multi-Paradigm Design for C++, JO Coplien, Addison-Wesley, 1999