to be negative and therefore optimizes its predictive branching accordingly. Note that even if our hint to the compiler turns out to be misleading at run time, the semantics of every well-formed program remain the same.

Stating explicit assumptions in code to achieve better optimizations

Although the presence of an attribute usually has no effect on the behavior of any well-formed program besides its runtime performance, an attribute sometimes imparts knowledge to the compiler, which, if incorrect, could alter the intended behavior of the program. As an example of this more forceful form of attribute, consider the GCC-specific `[[gnu::const]]` attribute, also available in Clang. When applied to a function, this attribute instructs the compiler to *assume* that the function is a **pure function**, which has no **side effects**. In other words, the function always returns the same value for a given set of arguments, and the globally reachable state of the program is not altered by the function. For example, a function performing a linear interpolation between two values may be annotated with `[[gnu::const]]`:

```
[[gnu::const]]
double linearInterpolation(double start, double end, double factor)
{
    return (start * (1.0 - factor)) + (end * factor);
}
```

More generally, the return value of a function annotated with `[[gnu::const]]` is not permitted to depend on any state that might change between its successive invocations. For example, it is not allowed to examine contents of memory supplied to it by address. In contrast, functions annotated with a similar but more lenient `[[gnu::pure]]` attribute are allowed to return values that depend on any nonvolatile state. Therefore, functions such as `strlen` or `memcmp`, which read but do not modify the observable state, may be annotated with `[[gnu::pure]]` but not `[[gnu::const]]`.

The `vectorLerp` function below performs linear interpolation (referred to as ~~LERP~~) between two bidimensional vectors. The body of this function comprises two invocations to the `linearInterpolation` function in the example above — one per vector component:

```
Vector2D vectorLerp(const Vector2D& start, const Vector2D& end, double factor)
{
    return Vector2D(linearInterpolation(start.x, end.x, factor),
                    linearInterpolation(start.y, end.y, factor));
}
```

If the values of the two components are the same, the compiler is allowed to invoke `linearInterpolation` only once, even if its body is not visible in `vectorLerp`'s translation unit: