```
private:
    T d_data[N];  // data initialized at construction

public:
    template <typename F>
    constexpr ConstexprArray(const F &func)
    : d_data{}
    {
        for (int i = 0; i < N; ++i)
        {
            d_data[i] = func(i);
        }
    }

    constexpr const T& operator[](std::size_t ndx) const
    {
        return d_data[ndx];
    }
};
```

The numerous alternative approaches to writing such data structures vary in their complexity, trade-offs, and understandability. In this case, we ~~default-~~initialize our elements before populating them but do not need to rely on any other significant new language infrastructure. Other approaches could be taken; see Section 2.1."**constexpr** Functions" on page 257.

Given this utility class template, we can then precompute at compile time any function that we can express as a **constexpr** function, such as a simple table of the first `N` squares:

```
constexpr int square(int x) { return x * x; }

constexpr ConstexprArray<int, 500> squares(square);

static_assert(squares[1]   == 1,    "");
static_assert(squares[289] == 83521,"");
```

Note that, as with many applications of **constexpr** functions, attempting to initialize a large array of **constexpr** variables will quickly bump up against a number of possible compiler-imposed limits.

Diagnosing undefined behavior at compile time

Avoiding ~~overflow during intermediate calculations~~ is an important consideration, especially from a security perspective, and yet is a generally difficult-to-solve problem. Forcing ~~computations to occur~~ at compile time brings the full power of the compiler to bear in ~~addressing~~ such undefined behavior.