```
constexpr struct D { int i; } x{1};  // brace-initialized aggregate x
constexpr int k = x.i;  // Subobjects of constexpr objects are constexpr.
```

Initializer undefined behavior

It is important to note the significance of one of the differences between a **constexpr** integral variable and a **const** integral variable. Because the initializer of a **constexpr** variable is *required* to be a constant expression, it is not subject to the possibility of undefined behavior, e.g., integer overflow or out-of-bounds array access, at run time and will instead result in a compile-time error:

```
          const int iA = 1 << 15;  // 2^15 = 32,768 fits in 2 bytes.
          const int jA = iA * iA;  // OK

          const int iB = 1 << 16;  // 2^16 = 65,536 doesn't fit in 2 bytes.
          const int jB = iB * iB;  // Bug, overflow (might warn)

constexpr const int iC = 1 << 16;
constexpr const int jC = iC * iC;  // Error, overflow in constant expression

constexpr       int iD = 1 << 16;  // Example D is the same as C, above.
constexpr       int jD = iD * iD;  // Error, overflow in constant expression
```

The code example above shows that an integer constant-expression overflow, absent **constexpr**, is not required by the C++ Standard to be treated as ill formed. When signed integer overflow happens in an initializer of a **constexpr** variable, however, the compiler is required to report it as an error (not just a warning).

A strong association exists between **constexpr** variables and functions; see Section 2.1. "**constexpr** Functions" on page 257. Using a **constexpr** variable rather than just a **const** one forces the compiler to detect overflow — and more generally, any undefined behavior — within the body of a **constexpr** function and report that overflow as an error in a way that the compiler would not otherwise be required to do.

For example, suppose we have two similar functions, `squareA` and `squareB`, defined for the built-in type **int** that each return the integral product of multiplying the single argument with itself:

```
          int squareA(int i) { return i * i; }  // nonconstexpr function
constexpr int squareB(int i) { return i * i; }  // constexpr function
```

Declaring a variable to be just **const** — and not **constexpr** — does nothing to force the compiler to check the evaluation of either function for overflow:

```
               int xA0 = squareA(1 << 15);  // OK
          const int xA1 = squareA(1 << 15);  // OK
constexpr       int xA2 = squareA(1 << 15);  // Error, squareA, not constexpr
```