```
U x = {    };  // OK, value-initializes x.i = 0
U y = { 1  };  // OK, copy-initializes y.i = 1
U z = { "" };  // Error, cannot initialize z.i with ""
```

Let's review the various ways in which we might attempt to initialize an object of **aggregate type** `A2` in the body of a function, `test`, i.e., defined at function scope:

```
struct A2 { int i; };  // aggregate with a single data member

void test()
{
           A2  a1;             // default init: i is not initialized!
    const  A2& a2 = A2();      // value init: i is 0.
           A2  a3 = A2();      // value init followed by copy init: i is 0.
           A2  a4();           // Oops, function declaration!
           A2  a5 = { 5 };     // aggregate initialization employing copy init
           A2  a6 = { };       //     "            "           "     value  "
           A2  a7 = { 5, 6 };  // Error, too many initializers for aggregate A2
    static A2  a8;             // zero-initialized, then (no-op) default init
}
```

Note the following in the sample code above.

- `a1` — Since `a1` is default initialized, each data member within the aggregate is itself independently default initialized. For scalar types, such as an **`int`**, the effect of default initialization at function scope is a no-op, i.e., `a1.i` is not initialized. Any attempt to access the contents of `a1.i` has undefined behavior.

- `a2` and `a3` — In the cases of both `a2` and `a3`, a temporary of type `A2` is first value initialized. Then, the temporary is bound to a reference for `a2`, extending its lifetime, whereas for `a3`, the temporary is used to copy-initialize the named variable. Both `a2.i` and `a3.i` are initialized to the value `0`.

- `a4` — Notice that we are unable to create a value-initialized local variable, `a4`, by applying parentheses since that would be interpreted as declaring a function taking no arguments and returning an object of type `A2` by value; see *Use Cases — Avoiding the most vexing parse* on page 237.

- `a5`, `a6`, and `a7` — C++03 supports aggregate initialization using braced syntax, as illustrated by `a5`, `a6`, and `a7` in the code snippet above. The local variable `a5` is copy initialized such that `a5.i` has the *user-supplied* value `5`, whereas `a6` is value initialized since there are no supplied initializers; hence, `a6.i` is initialized to 0. Attempting to pass `a7` two values to initialize a single data member results in a compile-time error. Note that had class `A2` held a second data member, the **statement** initializing `a5` would have resulted in copy initialization of the first and value initialization of the second.