Chapter 1

Safe Features

ch-safe
sec-safe-cpp11 Intro text should be here.

Chapter 1 Safe Features

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

Braced-Initialization Syntax: {}

bracedinit

Braced initialization, a generalization of C++03 initialization syntax, was designed with the intention designed to be used safely and uniformly in any initialization context.

Description

description

List initialization, originally dubbed **uniform initialization**, was conceived to enable a uniform syntax (having the same meaning) to be used generically to initialize objects irrespective of (1) the context in which the syntax is used or (2) the type of the object being initialized. Braced-initialization syntax is the language mechanism that — in close collaboration with the C++ Standard Library's `std::initializer_list` template (see Section 2.1."**??**" on page **??**) — is used to implement list initialization generally. As we will see, this design goal was largely achieved albeit with some idiosyncrasies and rough edges.

C++03 initialization syntax review

ization-syntax-review

Classic C++ affords several forms of initialization, each sporting its own custom syntax, some of which is syntactically interchangeable yet belying subtle differences. At the highest level, there are two dual categories of initialization: (1) **copy/direct** (when you have something from which to initialize) and (2) **default/value** (when you don't).

The first dual category of syntactic/semantic initialization comprises ***copy* initialization** and ***direct* initialization**. *Direct* initialization is produced when initializing an object with one or more arguments within parentheses, such as initializing a data member or base class in a constructor's initializer list, or in a **new** expression. *Copy* initialization happens when initializing from a value without using parentheses, such as passing an argument to a function, or returning a value from a function. Both forms may be used to initialize a variable:

```
#include <cassert>  // standard C assert macro

void test()
{
    int i = 23;  // copy initialization
    int j(23);   // direct initialization

    assert(i == j);
}
```

In both cases above, we are initializing the variable (`i` or `j`) with the literal value `23`.

For scalar types, there is no observable difference between these two dual forms of initialization in C++03, but, for user-defined types, there are. First *direct* initialization *considers* (as part of the overload set) all valid user-defined conversion sequences, whereas *copy* initialization excludes explicit conversion:

```
struct S
{
    explicit S(int);       //     explicit value constructor (from int)
```

```
            S(double);    // non-explicit value constructor (from double)
            S(const S&);  // non-explicit copy  constructor
};

S s1(1);     // direct init of s1: calls S(int)   ; copy constructor is not called
S s2(1.0);   // direct init of s2: calls S(double);   "      "        "   "     "

S s3 = 1;    // copy init of s3:   calls S(double); copy constructor may be called
S s4 = 1.0;  // copy init of s4:   calls S(double);   "      "         "    "   "
```

What's more, *copy* initialization is defined as if a temporary object is constructed; the compiler is permitted to elide this temporary and, in practice, typically does. Note, however, that *copy* initialization is not permitted unless there is an accessible *copy* (or *move*[1]) constructor, even if the temporary would have been elided.[2] Note that function arguments and return values are initialized using *copy* initialization.

Reference types are also initialized by copy and direct initialization, binding the declared reference to an object (or function). For a reference to a non**const** qualified type, the referenced type must match exactly. However, if binding a reference to a **const** qualified type, the compiler may copy initialize a temporary object of the target type of the reference and bind the reference to that temporary; in such cases, the lifetime of the temporary object is extended to the end of the lifetime of the reference.

```
void ref_inits()
{
    int i = 0;          // OK, copy initialization of int
    int& x(i);          // OK, direct initialization of reference
    const long& y = x;  // OK, y binds to a temporary and extends lifetime
    long& z = x;        // Error, incompatible types
}
```

The second dual category of syntactic/semantic initialization comprises ***default* initialization** and ***value* initialization**. Both default and value initialization pertain to situations in which *no* argument is supplied, and are distinguished by the presence or absence of parentheses, where the absence of parentheses indicates default initialization and the presence indicates value initialization. Note that in simple contexts such as declaring a variable, empty parentheses may also indicate a function declaration instead (see *Use Cases — Avoiding the most vexing parse* on page 26):

```
int i;          // default initialization
int j();        // Oops, function declaration
int k = int();  // value initialization

int *pd = new int;    // default initialization of dynamic int object
int *pv = new int();  // value        "         "     "      "    "
```

[1] If the *move* constructor for a user-defined type is declared and not accessible, copy initialization is ill formed; see Section 2.1."**??**" on page **??** and Section 1.1."**??**" on page **??**.

[2] In C++17, direct materialization replaces *copy* initialization in some contexts, thereby obviating the temporary object construction and also the need for an accessible *copy* or *move* constructor.

For **scalar types**, *default* initialization does not actually initialize an object, and *value* initialization will initialize that object as if by the literal `0`. Note that the representation of this value is not necessarily all zero bits, as some platforms use distinct trap values for the null pointer value for pointers and for pointer-to-member objects.

For class types with an accessible **user-provided** default constructor, default initialization and value initialization behave identically, calling the default constructor. If there is no accessible default constructor, both forms produce a compilation error. For objects of class types with an implicitly defined default constructor, each base and member subobject will be default initialized or value initialized according to the form of initialization indicated for the complete object; if any of those initializations produces an error, then the program is ill formed. Note that for a union with an implicitly defined default constructor, the first member of the union will be value initialized as the active member of that union when a union object is value initialized.

```
struct B
{
    int i;
    B() : i() { }  // user-provided default constructor
};

struct C
{
    int i;
    C() { }  // user-provided default constructor
};

struct D : B { int j; };  // derived class with no user-provided constructors

int *pdi = new int;    // default initialization of dynamic int object, *pdi is uninitialized
int *pvi = new int();  // value   initialization of dynamic int object, *pvi is 0
B *pdb = new B;        // default initialization of dynamic B object, b::i is 0
B *pvb = new B();      // value   initialization of dynamic B object, b::i is 0
C *pda = new C;        // default initialization of dynamic C object, a::i is uninitialized
C *pva = new C();      // value   initialization of dynamic C object, a::i is uninitialized
D *pdc = new D;        // default initialization of dynamic D object, c::j is uninitialized
D *pvc = new D();      // value   initialization of dynamic D object, c::j is 0
```

In the case of an object of type `B`, both default and value initialization will invoke the user-provided default constructor, which initializes the subobject `i` to `0`. In the case of an object of type `C`, both default and value initialization will invoke the user-provided default constructor, which does not initialize the subobject `i`. In the case of an object of type `D`, which has an implicitly defined default constructor, the initialization of the subobject `j` depends on whether the `D` object is initialized by default initialization or by value initialization.

Any attempt to read the value of an *uninitialized* object will result in **undefined behavior**. It is a diagnosable error to default initialize a constant object that does not execute a user-provided constructor to initialize each base and member. Note that the top level object, like `D` above, need not have a user-provided constructor as long as all of its bases and members can recursively apply this rule.

```
struct D2 : B { B j; };  // derived class with no user-provided constructors

const D  w;         // Error, w.j is not initialized.
const D  x = D();   // OK, x is value initialized.
const D2 y;         // OK, y is default initialized; sub-objects invoke default ctor.
const D2 z = D2();  // OK, z is value initialized.
```

Objects of static storage duration at file, namespace, or function scope (see Section 1.1."**??**" on page **??**) are ***zero* initialized** before any other initialization takes place. Note that ***default* initialized** static storage duration pointer objects are *zero* initialized to have a *null* address value (see Section 1.1."**??**" on page **??**), even if that representation on the host platform is not numerically zero:

```
struct A
{
    int i;
};

struct B
{
    int i;
    B() : i(42) { }
};

A globalA;
    // Zero initialization also zero initializes globalA::i.
    // Default initialization provides no further initializations.

B globalB;
    // zero initialization initializes globalB.i.
    // After that, default constructor is invoked.

int globalI;
    // Zero initialization initializes globalI.
```

Note the implication that default initialization for a static storage duration object will always initialize an object ready for use, either calling the default constructor of a type with a user-provided default constructor or zero-initializing scalars.

table-bracedinit-cpp3rules

Table 1: Helpful summary of C++03 rules

Initialization Type	No Arguments	>= 1 Arguments
With Parentheses	*value* `int i = int();`	*direct* `int i(23);`
Without Parentheses	*default* `int i;`	*copy* `int i = 23;`

regate-initialization

C++03 aggregate initialization

Aggregates are a special kind of object in C++03 that generally do not use constructors but follow a different set of rules for initialization, typically denoted by braces. There are two varieties of aggregates: (1) arrays and (2) user-defined class types that have no non-public data members that are not static, no base classes, no user-declared constructors, and no virtual functions. Aggregates are very similar to a classic C **`struct`**, potentially with additional, non-**`virtual`** member functions. Note that members of an aggregate are not themselves required to be an aggregate.

```
#include <string>  // std::string

int a[5];          // Arrays are aggregates.

struct A
{
    int         i;  // public data member
    std::string s;  // A is an aggregate even though std::string is not.

private:
    static int j;   // Private data member is static.
    void f();       // Member functions are OK, even if private.
};
```

A quick note on terminology: Strictly speaking, arrays comprise *elements* and classes comprise *members*, but for ease of exposition in this text, we refer to both as *members*.

When an aggregate is copied by either direct or copy initialization, rather than calling the copy constructors, the corresponding members (elements for an array) of each aggregate are copied using direct initialization, which corresponds to the behavior of an implicitly defined copy constructor for a class. Note that this process may be applied recursively, if members are aggregates themselves. Further note that in most cases, arrays do not copy because the argument supplied to the copy operation will undergo **array-to-pointer decay** and so will no longer be an appropriate type to initialize from. However, arrays as data members of classes follow the rules for aggregate initialization and so will copy array data members. This array-copy behavior is one of the motivations for the addition of the `std::array` template in C++11.

When an aggregate is *default* initialized, each of its members/elements is *default* initialized. When an aggregate is *value* initialized, each of its members/elements is *value* initialized. This follows the usual rules for an implicitly defined constructor for a class type and defines the corresponding behavior for array initialization.

```
int n = 17;
int *pid = new int[n];   // default initialization of dynamic array object and its elements
int *piv = new int[n](); // value        "         "     "      "     "     "   "
"

struct A { /*...*/ };
A *pd = new A;           // default initialization of dynamic A object and its members
A *pv = new A();         // value        "         "     "     "     "     "   "
```

"

Otherwise, an aggregate must be *aggregate* initialized by a braced list in the form `= { list-of-values };`, where members of the aggregate will be initialized by *copy* initialization from the corresponding value in the list of values; if the aggregate has more members than are provided by the list, the remaining members are *value* initialized; it is an error to provide more values in the list than there are members in the aggregate. Note that, because a union has only one active member, a union will be initialized by no more than a single value from the list; this becomes relevant for unions as data members of an aggregate initialized by *brace elision*:

```
union U
{
    int i;
    const char* s;
};

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

Let's review the various ways in which we might attempt to initialize an object of aggregate type `A` 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 followed by copy 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;             // default init after i is zero initialized.
}
```

In the sample code above:

- `a1`: `a1` is *default* initialized, which means that 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` will result in undefined behavior.
- `a2` and `a3`: In the cases of both `a2` and `a3`, a temporary of type `A` is first ***value* initialized** and then that 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 not able to create a *value* initialized local variable `a4` by applying parentheses since that would interpreted as declaring a function taking no

arguments and returning an object of type `A` by value; see *Use Cases* — **??** on page **??** [**AUs: there is no section with this name "Avoiding accidentally declaring a function taking no arguments"**] and *Use Cases* — *Avoiding the most vexing parse* on page 26.

- `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 `A` held a second data member, the line initializing `a5` would have resulted in *copy* initialization of the first and *value* initialization of the second.

- `a8` has static storage duration therefore it is first *zero* initialized (`a8.i` is 0) then it is *default* initialized, which is a no-op for the same reasons that `a1` is not initialized at all.

Finally, note that a scalar can be thought of as though it were an array of a single element (though note that scalars never suffer *array-to-pointer decay*); in fact, if we were to take the address of any scalar and add `1` to it, the new pointer value would represent the one-past-the-end iterator for that scalar's implied array (of length `1`). Similarly, scalars can be initialized using aggregate initialization, just as if they were single-element arrays, where the braced list for a scalar may contain zero or one elements.

```
int    i = { };       // OK, i is 0.
int    j = { 1 };     // OK, i is 1.
double k = { 3.14 };  // OK, k is 3.14.
```

itialization-in-c++11

Braced initialization in C++11

Everything we've discussed so far, including braced initialization of aggregates, is well defined in C++03. This same braced initialization syntax — modified slightly so as to preclude narrowing conversions (see the next section) — is extended in C++11 to work consistently and uniformly in many new situations. This enhanced braced initialization syntax is designed to better support the two dual initialization categories discussed in **C++03 initialization syntax review**above as well as entirely new capabilities including language-level support for lists of initial values implemented using the C++ Standard Library's `std::initializer_list` class template. As the opportunity arose by touching the rules for initialization, a few more potential errors, such as narrowing conversions, become diagnosable by the compiler when using braced initialization syntax.

narrowing-conversions

C++11 restrictions on narrowing conversions

Narrowing conversions (a.k.a. **lossy conversions**) are a notorious source of runtime errors. One of the important properties of list initializations implemented using the C++11 braced-initialization syntax is that error-prone narrowing conversions are no longer permitted. Consider, for example, an **`int`** array, `a`, initialized with various built-in (compile-time constant) *literal* values:

```
int ai[] =  //      C++03    C++11
{
    5,      // (0)   OK       OK
    5.0,    // (1)   OK      Error, double to int conversion is not permitted.
    5.5,    // (2)   OK      Error, double to int conversion is not permitted.
    "5",    // (3)  Error    Error, no const char* to int conversion exists.
};
```

In C++03, floating-point literals would be coerced to fit within an integer even if the conversion was known to be lossy — e.g., element (2) in the code snippet above. By contrast, C++11 disallows *any* such implicit conversions in braced initializations even when the conversion is known *not* to be lossy — e.g., element `ai[1]` above.

Narrowing conversions within the integral and floating-point type families, respectively, are generally disallowed except where it can be verified at compile-time that overflow does not occur and, in the case of integers and (classic) **enum**s , the initializer value can be represented exactly[3]:

```
const unsigned long ulc = 1;  // compile-time integral constant: 1UL

short as[] =  //       C++03     C++11                        Stored Value
{
     32767,   // (0)     OK        OK                        as[0] == 32767
     32768,   // (1)     OK      Error, overflow
    -32768,   // (2)     OK        OK                        as[2] == -32768
    -32769,   // (3)   Warning?  Error, underflow
    1UL,      // (4)     OK        OK                        as[4] == 1
    ulc,      // (5)     OK        OK                        as[5] == 1
    1.0       // (6)     OK      Error, narrowing
};
```

Notice that both *overflow* (1) and *underflow* (3) are rejected for integral values in C++11, whereas neither is ill formed in C++03. An integral literal (4) (or an **integral constant** (5)) of a wider type (e.g., **unsigned long**) can be used to initialize a smaller one (e.g., **signed short**) provided that the value can be represented exactly; however, even a floating point literal that can be *represented exactly* (6) is nonetheless rejected in C++11 when used to initialize any integral scalar.

Floating-point initializers, on the other hand, need not be represented precisely so long as overflow does not occur; if, however, what is being initialized is a floating-point scalar and the initializer is integral, then the value must be represented exactly:

```
float af[] =     //       C++03   C++11                  Stored Value
{
     3L,        // (0)     OK     OK                   af[0] == 3
     16777216,  // (1)     OK     OK                   af[1] == 1<<24
     16777217,  // (2)     OK     Error, lossy
     0.75,      // (3)     OK     OK                   af[1] == 0.75
     2.4,       // (4)     OK     OK, but lossy        af[2] != 2.4
```

[3] As of C++20, implicit conversion from either a pointer or pointer-to-member type to **bool** is generally supported in braced initializations.

```
    0.4,       // (5)     OK     OK, but lossy        af[3] != 0.4
    1e-39,     // (6)     OK     OK, but underflow    af[4] != 1e-39
    1e+39,     // (7)     OK     Error, overflow
};
```

In the example above elements (0) – (2) represent initialization from an integral type (**int**), which requires that the initialized value be represented exactly. Elements (3) – (7) are instead initialized from a floating-point type (**double**) and therefore are restricted only from overflow.

When an initializer is *not* a **constant expression**, braced initialization precludes any possibility of such *narrowing* initializations at run time — e.g., initializing a **float** with a double or a **long double**, a **double** with a **long double**, or *any* floating-point type with an integral one. By the same token, an integral type (e.g., **int**) is not permitted to be initialized by non-constant expression integer value of any other potentially larger integral type (e.g., **long**) — even if the number of bits in the representation for the two types on the current platform is the same. Finally, non-constant expression of integral type (e.g., **short**) cannot be used to initialize an unsigned version of the same type (e.g., **unsigned short**) and vice versa.

To illustrate the constraints imposed on non-constant-expressions described above, consider a simple aggregate class, `S`, comprising an **int**, `i`, and a **double**, `d`:

```
struct S  // aggregated class
{
    int    i;  // *integral* scalar type
    double j;  // *floating-point* scalar type
};
```

A function, `test`, declaring a variety of arithmetic parameter types illustrates restrictions imposed by C++11 braced initialization on narrowing initializations that were well formed in C++03:

```
void test(short s, int i, long j, unsigned u, float f, double d, long double e)
{                       //     C++03  C++11
    S s0 = { i, d };  // (0)  OK     OK
    S s1 = { s, f };  // (1)  OK     OK
    S s2 = { u, d };  // (2)  OK    Error, u causes narrowing.
    S s3 = { i, e };  // (3)  OK    Error, e causes narrowing.
    S s4 = { f, d };  // (4)  OK    Error, f causes narrowing.
    S s5 = { i, s };  // (5)  OK    Error, s causes narrowing (theoretically).
};
```

In the `test` function above, lines (0) and (1) are OK because there is no possibility of narrowing on any conforming platform unlike lines (2) through (5) — despite the fact that, in practice, it is more than likely that a **double** will be able to represent exactly every value representable by a **short int**. Note that, just as with the array example above, when the initializing value is a constant expression, it is sufficient that that value be representable exactly in the target type and produce the original value when converted back.

ggregate-initialization

C++11 Aggregate initialization

Aggregate initialization in C++11, including initialization of arrays, is subject to the rules prohibiting narrowing conversions.

```
int  i   = { 1 };        // OK
long j   = { 2 };        // OK

int  a[] = { 0, 1, 2 };  // OK
int  b[] = { 0, i, j };  // Error, cannot narrow j from long to int

struct S { int a; };
S s1 = { 0  };  // OK
S s2 = { i  };  // OK
S s3 = { 0L };  // OK, 0L is an integer constant expression.
S s4 = { j  };  // Error, narrowing
```

In addition, the rules for value initialization now state that members without a specific initializer value in the braced list are "as-if" copy initialized from `{}` [4]. This will result in an error when initializing a member that has an **`explicit`** default constructor according to the new **copy *list* initialization** rules in the next section, which give a meaning for explicit constructors. Note that if the member is of reference type and no initializer is provided, the initialization is ill formed.

Regardless of whether the aggregate itself is initialized using a copy initialization or direct initialization, the members of the aggregate will be copy initialized from the corresponding initializer.

```
struct E { };                          // empty type
struct AE { int x; E y; E z; };        // aggregate comprising several empty objects
struct S { explicit S(int = 0) {} };   // class with explicit default constructor
struct AS{ int x; S y; S z; };         // aggregate comprising several S objects

AE aed;                    // OK
AE ae0 = {};               // OK
AE ae1 = { 0 };            // OK
AE ae2 = { 0, {} };        // OK
AE ae3 = { 0, {}, {} };    // OK

AS asd;                    // OK
AS as0 = {};               // OK in 03; Error in 11 calling explicit ctor for S
AS as1 = { 0 };            // OK in 03; Error in 11 calling explicit ctor for S
AS as2 = { 0, S() };       // OK in 03; Error in 11 calling explicit ctor for S
AS as3 = { 0, S(), S() };  // OK, all aggregate members have an initializer.
```

To better support generalizing the syntax of brace initialization in a style similar to aggregate initialization, an aggregate can make a copy of itself through *aggregate* initialization in C++11 as well as through *direct* initialization per C++03:

```
S x{};      // OK, value initialization
```

[4] From C++ 14 onwards, if the member doesn't have an initializer value, but has a default member initializer, it is initialized from the default member initializer (see Section 2.1."??" on page ??).

```
S y = {x};  // OK in C++11; copy initialization via aggregate initialization syntax
```

Otherwise, initialization of aggregates in C++11 is exactly the same where it would have a meaning in C++03 and is correspondingly extended into new places where braced initialization is permitted, as documented in the following subsections.

Copy list initialization

y-list-initialization

For C++03, only aggregates and scalars could be initialized via braced-initialization syntax:

```
Type var = { /*...*/ };  // C++03-style aggregate (only) initialization
```

The first part of generalizing braced initialization syntax for C++11 is to allow the same syntactic form used to initialize aggregates to be used for *all* user-defined types. This extended form of braced initialization — known as copy *list* initialization — follows the rules of copy initialization:

```
Class var1 = val;      // (C++03) copy initialization
Class var2 = { val };  // (C++11) copy list initialization
```

For a non-aggregate class type, C++11 allows the use of a braced list provided that its sequence of values serves as a suitable argument to a non-**explicit** constructor of the class being initialized. Importantly, this use of *copy* list initialization provides meaning to **explicit** constructors when taking other than a single argument. For example, consider a **struct** S having constructors with 0-3 parameters, only that last of which is **explicit**:

```
struct S
{
            S();                          // default cor
            S(int);                       // 1-value ctor
            S(int, const char*);          // 2-value ctor
    explicit S(int, const char*, double); // 3-value ctor
};
```

We can use *copy* list initialization only if the selected constructor is *not* declared to be **explicit**, e.g., s0, s1, and s2 but not s3:

```
S s0 = { };                // OK, copy list initialization
S s1 = { 1 };              // OK, copy list initialization
S s2 = { 1, "two" };       // OK, copy list initialization
S s3 = { 1, "two", 3.14 }; // Error, constructor is explicit
```

Had we instead declared our default constructor or any of the others to be **explicit**, the corresponding *copy* (or *copy-list*) initialization above would have failed too.

Another important difference between C++11 copy *list* initialization and C++03 copy initialization is that the braced-list syntax considers all constructors, including those that are declared to be **explicit**. Consider a **struct** X having two overloaded single-argument constructors, i.e., (1) one taking an **int** and (2) the other a member template taking a single (deduced) type, T, by value:

```
struct Q  // class containing both explicit and implicit constructor overloads
{
```

```
    explicit Q(int);          // (1) value constructor taking a int
    template <class T> Q(T);  // (2) value constructor taking a T
};
```

Employing *direct* initialization (e.g., `x0` in the code snippet below) selects the most appropriate constructor, regardless of whether it is declared to be **`explicit`**, and successfully uses that one; employing *copy* initialization (e.g., `x1`) drops explicit constructors from the overload set before determining a best match; and employing copy *list* initialization (e.g., `x2`) again includes all constructors in the overloads set but is **ill formed** if the selected constructor is **`explicit`**:

```
Q x0(0);     // OK, direct initialization calls Q(int).
Q x1 = 1;    // OK, copy initialization calls Q(T).
Q x2 = {2};  // Error, copy list initialization selects but cannot call Q(int).
Q x3{3};     // Same idea as x0; direct list initialization calls Q(int).
```

In other words, the presence of the `=` coupled with the braced notation (e.g., `x2` in the code example above) forces the compiler to choose the constructor *as if* it were direct initialization (e.g., `x0`) but then forces a compilation failure if the selected constructor turns out to be **`explicit`**. This "consider-but-fail-if-selected" behavior of copy list initialization is analogous to that of functions declared using **`= delete`**; see Section 1.1."**??**" on page **??**. Using braces but omitting the `=` (e.g., `x3`) puts us back in the realm of *direct* rather than *copy* initialization; see *Direct list initialization* on page 17.

When initializing references, copy list initialization (braced syntax) behaves similarly to copy initialization (no braces) with respect to the generation of temporaries. For example, when using a braced list to initialize an *lvalue* reference — e.g., **`int&`** `ri` or **`const int&`** `cri` in the code example below — to a scalar of a type that exactly matches it (e.g., **`int`** `i`), no temporary is created (just as it would not have been without the braces); otherwise, a temporary will be created, provided that a viable conversion exists and is not narrowing:

```
    assert

void test()
{
    int  i = 2;               assert(i   == 2);
    int& ri = { i };          assert(ri  == 2);  // OK, no temporary created
    ri = 3;                   assert(i   == 3);  // original is affected

    const int& cri = { i };   assert(cri == 3);  // OK, no temporary created
    ri = 4;                   assert(cri == 4);  // other reference is affected

    short s = 5;              assert(s   == 5);
    const int& crs = { s };   assert(crs == 5);  // OK, temporary is created
    s = 6;                    assert(crs == 5);  // note temporary is unchanged

    long j = 7;               assert(j   == 7);
    const int& crj = { j };   // Error, narrowing conversion from long to int
}
```

As evidenced by the C-style asserts above, no temporary is created when initializing either `ri` or `cri` since modifying the reference affects the underlying variable, and vice versa. The

C++ type of `crs`, on the other hand, does *not* match exactly that of the type to which it is bound, a temporary *is* created and hence changing the underlying object does *not* affect its referenced value. Lastly, unlike `s` (of type **`short`**), attempting to initialize a **`const`** *lvalue* reference of type **`int`**, `crj`, with `j` (of type **`long`**) is a narrowing conversion and thus ill formed.

Another consideration involves the standard **`typedef`** `std::size_t` found in the standard header `<cstddef>`, which must have sufficient bits to represent the unsigned difference between any two pointers (into contiguous memory) and is typically, but not necessarily, an alias for an **`unsigned long`**:

```
  std::size_t

std::size_t k = 8;  // alias to implementation-defined type, say, unsigned long

unsigned int&       ra1 = { k };  // Note: Only one of these three lines will
unsigned long&      ra2 = { k };  // compile on any given platform; the other two
unsigned long long& ra3 = { k };  // will necessarily be ill formed and not compile.
```

Historically, a **`long`** has always been of sufficient size to **pun** a pointer value (BAD IDEA) yet, back in the day when **`int`** and **`long`** were the same number of bytes on a given platform, `size_t` was often an alias to an **`unsigned int`** rather than an **`unsigned long`**. Moving forward, one might expect `size_t` to be an alias for an **`unsigned long`** on a 64-bit platform, but there is no such assurance in the C++ Standard and **`unsigned long long`** (see Section 1.1."**??**" on page **??**) is another viable option on any standards-conforming platform.

Finally, **`const`** *lvalue* references to scalars and aggregates initialized via braced lists of literal values follow the rules of aggregates (see *C++11 Aggregate initialization* on page 13); a temporary is materialized having the indicated value and bound permanently to the reference with its lifetime extended coterminously:

```
const int& i0 = { };    // OK, materialized temporary is value initialized.
const int& i1 = { 5 };  // OK,       "           "       " copy      "
```

In the example above, a temporary is *value* initialized (to 0) and bound to `i0`; another temporary is then *copy* initialized (to 5) and bound to `i1`.

Non-modifiable references to *aggregate UDT*s exploit the generalization of *copy* and *direct* list initialization. Consider an aggregate `A` that comprises three **`int`** data members, `i`, `j`, and `k`:

```
struct A          // struct A is an aggregate data type.
{
    int i, j, k;  // This struct contains three data members of type int.
};
```

We can now use braced initialization to materialize a temporary object of aggregate type `A` using aggregate initialization.

```
const A& s0 = { };          // i, j, and k are  value initialized.
const A& s1 = { 1 };        // i copy and j and k are value initialized.
const A& s2 = { 1, 2 };     // i and j copy and k are value initialized.
const A& s3 = { 1, 2, 3 };  // i,j, and k are copy initialized.
```

In the example above, each of the references, `s0` thru `s3`, is initialized to a temporary **`struct`** of type `A` holding the respective aggregate value `{0,0,0}`, `{1,0,0}`, `{1,2,0}`, and `{1,2,3}`.

ect-list-initialization

Direct list initialization

Of the two dual forms of initialization, *direct* versus *copy*, *direct* initialization is the stronger since it enables use of all accessible constructors, i.e., including those declared to be **`explicit`**. The next step in generalizing the use of braced initialization is to allow use of a braced list without the intervening `=` character between the variable and the opening brace to denote direct initialization too:

```
Class var1(/*...*/);  // C++03-style direct initialization
Class var2{/*...*/};  // C++11-style direct list initialization
```

Note that C++ does *not* similarly relax the rules to allow for initialization of aggregates with parentheses.[5]

The syntax suggested in the previous example is known as **direct list initialization** and follows the rules of direct initialization rather than copy initialization in that all constructors of the named `type` are both considered and accessible in the initial overload set:

```
struct Q  // class containing explicit constructor
{
    explicit Q(int);  // value constructor taking a int
    // ...
};

Q x(5);    // OK  direct initialization can call explicit constructors.
Q y{5};    // OK, direct list initialization can call explicit constructors.
Q z = {5}; // Error, copy list initialization can't call explicit constructors.
```

Either form of direct initialization (shown for `x` and `y` in the code example above) may invoke an **`explicit`** constructor of class `Q`, whereas *copy list* initialization will necessarily result in a compile-time error.

However, following the rules of C++11 braced initialization, narrowing conversions are rejected by direct *list* initialization:

```
long a = 3L;

Q b(a);  // OK, direct initialization
Q c{a};  // Error, direct list initialization cannot use a narrowing conversion.
```

Similarly, **`explicit`** conversion operators (see Section 1.1."**??**" on page **??**) can be considered when *direct* list initialization (or direct initialization) is employed on a scalar, but not so with *copy* list initialization (or copy initialization). Consider, for example, a class, `W` that can covert to either an **`int`** or a **`long`** where conversion to **`int`** is explicit and therefore must be *direct*:

```
struct W
```

[5] C++20 finally allows for aggregates to be initialized with parentheses.

```
{
    explicit operator int()  const;  // used via direct initialization only
             operator long() const;  // used via direct or copy initialization
};
```

Initializing a **long** variable with an expression of type W can be accomplished via either *direct* or *copy* initialization (e.g., jDirect and jCopy, respectively, in the code snippet below), but initializing an **int** variable with such an expression can be accomplished only via *direct* initialization (e.g., iDirect):

```
long jDirect {W()};  // OK, considers both operators, calls operator long
long jCopy = {W()};  // OK, considers implicit op only, calls operator long
int  iDirect {W()};  // OK, considers both operators, calls operator int
int  iCopy = {W()};  // Error, considers implicit op only, narrowing conversion
```

In the example above, attempting to use *copy* list initialization (e.g., iCopy) forces the conversion to **long** as the only option, which results in a narrowing conversion and an ill-formed program.

Note that, for **aggregate** types, even direct list initialization will not allow the explicit constructors of the individual member types to be considered since such *member-wise* initialization is invariably *copy* initialization; see *C++11 Aggregate initialization* on page 13.

We may use direct list initialization as part of **member initialization lists** for base classes and member data of a class (note that there is no equivalent to allow copy list initialization in such a context). Consider, for example, an aggregate class, B, a non-aggregate class, C, and a derived class, D, that inherits from B and has an object of type C as a data member, m:

```
struct B { int i; };        // aggregate base type
struct C { C(); C(int); };  // non-aggregate member type

struct D : B  // class publicly derived from B containing C
{
    C m;  // non-aggregate data member

    D()      : B{},  m{}  { }  // direct initialized base/member objects
    D(int x) : B{x}, m{x} { }  //    "       "        "     "       "
};
```

In the definition of class D above, both constructors employ direct list initialization; the first is also an example of *value* initialization for both (aggregate) class B and (non-aggregate) class C. Note that in C++03, aggregate bases and members could only be default initialized, value initialized, or direct initialized, and not initialized to another value.

New expressions are another context in which direct list initialization (braces) or direct initialization (parentheses) can occur and applies similarly to both aggregate and nonaggregate types. If no initializer is provided, the allocated object is default initialized; if empty braces or parentheses are supplied, the object is value initialized; otherwise, the object is initialized from the contents of the braced or (where permitted) parenthesized list.

As an illustrative example, let's consider the scalar type **int** which itself can be default initialized (not initialized), value initialized (to 0) via empty braces or parentheses, or **direct initialized** via a single element within either parentheses or braces:

```
int* s0  = new int;               // default initialized (no initializer)

int* t0 = new int();              // direct (value) initialized from ()
int* t1 = new int{};              // direct (value) list initialized from {}

int* u0 = new int(7);             // direct initialized from 7
int* u1 = new int{7};             // direct list initialized from {7}

int* v0 = new int[5];             // All 5 elements are default initialized.

int* w0 = new int[5]();           // All 5 elements are value initialized.
int* w1 = new int[5]{};           // Array is direct list initialized from {}.

int* x0 = new int[5](9);          // Error, invalid initializer for an array
int* x1 = new int[5]{9};          // Array is direct list initialized from {9}.

int* y1 = new int[5]{1,2,3};      // array direct list initialized from {1,2,3}

int* z1 = new int[5]{1,2,3,4,5};  // direct list initialized from {1,2,3,4,5}
```

All the comments above apply to the object being created in the **new** expression; the pointer set to the address of the dynamically allocated object is copy initialized in all cases. Note that, in C++03, we could default initialize (e.g., `v0`) or value initialize (e.g., `w0`) the elements of an array in a **new** expression but there was no way to initialize the elements of such an array to anything other than their default value (e.g., `x0`); as of C++11, direct list initialization with braces (e.g., `x1`, `y1`, `z1`) makes this more flexible, heterogeneous initialization of array elements in **new** expressions possible.

ect-list-initialization

Contrasting copy and direct list initialization

The difference between *copy* list initialization and *direct* list initialization can be seen in this example:

```
struct C
{
    explicit C() { }
    explicit C(int) { }
};

struct A  // aggregate of C
{
    C x;
    C y;
};

int main()
{
    C c1;           // OK, default initialization
    C c2{};         // OK, value initialization
    C c3{1};        // OK, direct list initialization
```

```
    C c4 = {};      // Error, copy list initialization cannot use explicit default ctor.
    C c5 = {1};     // Error, copy list initialization cannot use explicit ctor.

    C c6[5];        // OK, default initialization
    C c7[5]{};      // Error, aggregate initialization requires a non-explicit default ctor.
    C c8[5]{1};     // Error, aggregate initialization requires non-explicit ctors.
    C c9[5] = {};   // Error, aggregate initialization requires a non-explicit default ctor.
    C ca[5] = {1};  // Error, aggregate initialization requires non-explicit ctors.

    A a1;           // OK, default initialization
    A a2{};         // Error, aggregate initialization requires a non-explicit default ctor.
    A a3{1};        // Error, aggregate initialization requires non-explicit ctors.
    A a4 = {};      // Error, aggregate initialization requires a non-explicit default ctor.
    A a5 = {1};     // Error, aggregate initialization requires non-explicit ctors.
}
```

Note that if the constructors for `C` were not marked explicit, then all of the variables in the example above would be safely initialized. If only the **`int`** constructor of `C` were explicit, then the initializations that did not depend on the **`int`** constructor would be valid:

```
struct C
{
    C() { }
    explicit C(int) { }
};

struct A  // aggregate of C
{
    C x;
    C y;
};

int main()
{
    C c1;           // OK, default initialization
    C c2{};         // OK, value initialization
    C c3{1};        // OK, direct list initialization
    C c4 = {};      // OK, copy list initialization
    C c5 = {1};     // Error, copy list initialization cannot use explicit ctor.

    C c6[5];        // OK, default initialization
    C c7[5]{};      // OK, value initialization
    C c8[5]{1};     // Error, aggregate initialization requires non-explicit ctors.
    C c9[5] = {};   // OK, copy list initialization
    C ca[5] = {1};  // Error, aggregate initialization requires non-explicit ctors.

    A a1;           // OK, default initialization
    A a2{};         // OK, value initialization
    A a3{1};        // Error, aggregate initialization requires non-explicit ctors.
    A a4 = {};      // OK, copy list initialization
```

```
    A a5 = {1};     // Error, aggregate initialization requires non-explicit ctors.
}
```

n-braced-initialization

Integrating default member initialization with braced initialization

Another new feature for C++11 is *default member initializers* for data members in a class. This new syntax supports both copy list initialization and value list initialization. However, initialization with parentheses is not permitted in this context.

```
struct S
{
    int i = { 13 };

    S() { }                         // OK, i == 13.
    explicit S(int x) : i(x) { }  // OK, i == x.
};


struct W
{
    S a{};        // OK, by default j.i == 13.
    S b{42};      // OK, by default j.i == 42.
    S c = {42};   // Error, constructor for S is explicit.
    S d = S{42};  // OK, direct initialization of temporary for initializer
    S e(42);      // Error, fails to parse as a function declaration
    S f();        // OK, declares member function f
};
```

gument-to-a-constructor

List initialization where the list itself is a single argument to a constructor

Another new form of initialization for C++11 is list initialization with a braced list of arguments to populate a container. See Section 2.1."**??**" on page **??** for details. If a braced list of arguments are all of the same type, then the compiler will look for a constructor taking an `std::initializer_list<T>` argument, where `T` is that common type. Similarly, if a braced list of values can be implicitly converted to a common type, then a constructor for an `std::initializer_list` of that common type will be preferred. When initializing from a nonempty braced initializer list, a matching initializer list constructor always wins overload resolution. However, *value* initializing from a pair of empty braces will prefer a default constructor.

std::initializer_list

```
struct S
{
  S() {}
  S(std::initializer_list<int>) {}
  S(int, int);
};

S a;            // default initialization with default constructor
```

```
S b();           // function declaration!
S c{};           // value initialization with default constructor
S d = {};        // copy list initialization with std::initializer_list
S e{1,2,3,4,5};  // direct list initialization with std::initializer_list
S f{1,2};        // direct list initialization with std::initializer_list
S g = {1,2};     // copy list initialization with std::initializer_list
S h(1,2);        // direct initialization with two ints
```

tialization-a-temporary

Omitting the type name when braced initializing a temporary

In addition to supporting new forms of initialization, C++11 allows for braced lists to implicitly construct an object where the type is known by context, such as for function arguments and return values. This use of a braced list without explicitly specifying a type is just like constructing a temporary object by copy initialization, hence using copy list initialization, for the implicit type. As such construction is copy list initialization, it will reject explicit constructors:

```
struct S
{
    S(int, int) {}
    explicit S(const char*, const char*) {}
};

S foo(bool b)
{
    if (b)
    {
        return S{ "hello", "world" };  // OK, direct list initialization of temporary
        return { "hello", "world" };   // Error, copy list initialization cannot call explicit ctor.
    }
    else
    {
        return {0, 0};  // OK, int constructor is not explicit.
    }
}

void bar(S s) { }

int main()
{
   bar( S{0,0} );  // OK, direct list initialization, then copy initialization
   bar(  {0,0} );  // OK, copy list initialization
   bar( S{"Hello", "world"} );  // OK, direct list initialization
   bar(  {"Hello", "world"} );  // Error, copy list initialization cannot use explicit ctor.
}
```

conditional-expressions

Initializing variables in conditional expressions

As a final tweak to make initialization consistent across the language, initializing a variable in the condition of a **while** or **if** statement in C++03 supported only copy initialization and required use of the = token. For C++11, those rules are relaxed to allow any valid form of braced initialization. Conversely, the declaration of a control variable in a **for** loop has supported all forms of initialization permitted for a variable declaration since the original C++ standard[6]:

```
void f()
{
   for (int i = 0; ; ) {}    // OK in all versions of C++
   for (int i = {0}; ; ) {}  // OK for aggregates in C++03 and all types from C++11
   for (int i{0}; ; ) {}     // OK from C++11, direct list initialization
   for (int i{}; ; ) {}      // OK from C++11, value initialization
   for (int i(0); ; ) {}     // OK in all versions of C++
   for (int i(); ; ) {}      // OK in all versions of C++
   for (int i; ; ) {}        // OK in all versions of C++

   if (int i = 0) {}    // OK in all versions of C++
   if (int i = {0}) {}  // OK from C++11, copy list initialization
   if (int i{0}) {}     // OK from C++11, direct list initialization
   if (int i{}) {}      // OK from C++11, value initialization
   if (int i(0)) {}     // Error in all versions of C++
   if (int i()) {}      // Error in all versions of C++
   if (int i) {}        // Error in all versions of C++

   while (int i = 0) {}    // OK in all versions of C++
   while (int i = {0}) {}  // OK from C++11, copy list initialization
   while (int i{0}) {}     // OK from C++11, direct list initialization
   while (int i{}) {}      // OK from C++11, value initialization
   while (int i(0)) {}     // Error in all versions of C++
   while (int i()) {}      // Error in all versions of C++
   while (int i) {}        // Error in all versions of C++
}
```

alization-and-=-default

Default initialization and = default

Another new feature for C++11 is the notion of defaulted constructors, defined by = **default** (see Section 1.1."??" on page ??) to have the same definition as the implicitly defined constructor. This is especially useful when you want to provide constructors for your class without losing the **triviality** otherwise associated with the implicit constructors, notably for the default constructor that would no longer be declared as soon as another constructor is declared.

One important property of = **default** constructors is that, while they are user *declared*, they are not user *provided* as long as this definition occurs within the class definition itself:

```
struct Trivial
```

[6]Note that GCC would traditionally accept the C++11-only syntaxes, even when using C++98/03.

```
{
    int i;
    Trivial() = default;            // user declared but not user provided
};

struct NonTrivial
{
    int j;
    NonTrivial();                    // user declared
};

NonTrivial::NonTrivial() = default;  // user provided
```

Following the rules derived from C++03, a class with no user-provided default constructor may behave differently under *default* initialization compared to *value* initialization:

```
void demo()
{
    const Trivial    a;    // Error, a.i not initialized
    const Trivial    b{};  // OK, b.i = 0.
    const NonTrivial c;    // OK, but c.j never initialized
    const NonTrivial d{};  // OK, but d.j never initialized
}
```

Note that we use function local variables for this example to avoid confusion with *zero* initialization for global variables.

alization-and-scalars

Copy initialization and scalars

With the addition of explicit conversion operators to C++11 (see Section 1.1."**??**" on page **??**), it becomes possible for *copy* initialization and *copy list* initialization to fail for scalars and similarly for *direct list* initialization:

```
struct S
{
    explicit operator int() const { return 1; }
};

S one{};

int a(one);     // OK, a = 1.
int b{one};     // OK, b = 1.
int c = {one};  // Error, copy list initialization used with
                // with explicit conversion operator.
int d = one;    // Error, copy initialization used with
                // with explicit conversion operator.

class C {
    int x;
    int y;
```

```
public:
    C(const S& value) : x(value)  // OK, x = 1
                      , y{value}  // OK, y = 1
    {
    }
};
```

use-cases-bracedinit

Use Cases

ue-initialized-variable

Defining a value-initialized variable

The C++ parser has a pitfall where an attempt to value initialize a variable turns out to be a function declaration:

```
struct S{};

S s1();      // Oops! function declaration
S s2 = S();  // variable declaration using value initialization and then copy initialization
```

The declaration of `s1` looks like an attempt to value initialize a local variable of type `S`, but, in fact, it is a forward declaration for a function `s1` that takes no arguments and returns an `S` object by value. This is particularly surprising for folks who did not realize we could declare (but not define) a function within the body of another function, a feature retained from the original C Standard. Clearly, there would be an ambiguity in the grammar at this point unless the language provided a rule to resolve the ambiguity, and the grammar opts in favor of the function declaration in all circumstances, including at function local scope. Whilst this rule would be essential at namespace/class scope, otherwise functions taking no arguments could not be so easily declared, the same rule also applies at function local scope, first for consistency and second for compatibility with pre-existing C code that we might want to compile more strictly[7] with a C++ compiler.

By switching from parentheses to braces, there is no more risk of confusion between a vexing parse and a variable declaration:

```
S s{};  // object of type S
```

As a digression, it is worth noting that `S` is an empty type. Just as it is ill-formed to rely on default initialization for a **`const`** object of a trivial type such as an **`int`** or an aggregate of just an **`int`**, it was also, prior to Defect Report CWG 253 for C++17 and the introduction of **const-default-constructible** types, ill-formed to rely on default initialization for a **`const`** object of an empty type. Hence, it is often desirable to value initialize such objects, running into the vexing parse. As this Defect Report was resolved at the end of 2016, and applies retroactively to earlier dialects, most current compilers no longer enforce this restriction, and some compilers (notably GCC) had already stopped enforcing this rule several years previously:

```
const S cs1;        // Error on some compilers (as described above)
```

[7] While C and C++ both enforce type safety, C++ enforces strict type safety, where all declared types are distinct; C enforces structural conformance, where two distinct structs with the same sequence of members are treated as the same type.

```
const S cs2{};      // OK, value initialization
const S cs3 = {};   // OK, aggregate initialization
const S cs4 = S();  // OK, copy initialization
```

Avoiding the most vexing parse

the-most-vexing-parse

Value initializing function arguments can lead to another pitfall, often called *the most vexing parse*. C++ will parse the intended value initialization of a function argument as the declaration of an unnamed parameter of a function type instead, which would otherwise not be legal but for the language rule that such a parameter implicitly decays to a pointer to a function of that type:

```
struct V { V(const S&) { } };

void foo()
{
    V v1(S());    // most vexing parse, declares function v1 taking a function pointer
    V v2((S()));  // workaround, object of type V due to non-redundant parentheses on argument

    S x = S();    // declare a variable of type S
    V v3(x);      // workaround, object of type V but argument is now named, with longer lifetime
}
```

In the example above, `v1` is the forward declaration of a function in the surrounding namespace that returns an object of type `V` and has an (unnamed) parameter of type pointer-to-function-returning-`S`-and-taking-no-arguments, `S(*)()`. That is, the declaration is equivalent to:

```
V v1(S(*)());
```

This most vexing of parses can be disambiguated by having the arguments clearly form an expression, not a type. One simple way to force the argument to be parsed as an expression is to add an otherwise redundant pair of parentheses. Note that declaring the constructor of `V` as **`explicit`** in the hopes of forcing a compile error is no help here since the declaration of `v1` is not interpreted as a declaration of an object of type `V`, so the **`explicit`** constructor is never considered.

With the addition of generalized braced initialization in C++11, a coding convention to prefer empty braces, rather than parentheses, for all value initializations avoids the question of the most vexing parse arising:

```
V v4(S{});  // direct initialize object of type V with value initialized temporary
```

Note that the most vexing parse can also apply to constructors taking multiple arguments, but the issue arises less often since any one supplied argument clearly being an expression, rather than a function type, resolves the whole parse:

```
struct W { W(const S&, const S&) { } };

W w1( S(),  S());  // most vexing parse, declares function w1 taking two function pointers
W w2((S()), S());  // workaround, object of type V due to non-redundant parentheses on argument
W w3( S{},  S());  // workaround, even a single use of S{} disambiguates further use of S()
```

ization-in-generic-code

Uniform initialization in generic code

One of the design concerns facing an author of generic code is which form of syntax to choose to initialize objects of a type dependent on template parameters. Different C++ types behave differently and accept different syntaxes, so providing a single consistent syntax for all cases is not possible. Consider the following example of a simple test harness for a unit testing framework:

```
#include <initializer_list>  // std::initializer_list

template <class T, class U>
bool run_test(bool (*test)(const T&), std::initializer_list<U> iL)
{
    for (const auto& val : iL)
    {
        T obj = val;      // initialize the test value
        if (!test(obj))
        {
            return false;
        }
    }

    return true;
}
```

In this example, a test function is provided for an object of parameter type `T` along with an `initializer_list` of test values. The **`for`** loop will construct a test object with each test value, in turn, and call the test function, returning early if any value fails. The question is which syntax to use to create the test object `obj`.

- As written, the example uses *copy* initialization — `T obj = val;` — and so will fail to compile if a non-**`explicit`** constructor cannot be found or if `T` is an aggregate that is not `U`.
- If we switched to *direct* initialization — `T obj(val);` — then explicit constructors would also be considered.
- If we switched to *direct list* initialization — `T obj{val};` — then aggregates would be supported as well as explicit constructors but not narrowing conversions; `initializer_list` constructors are also considered and preferred.
- If we switched to *copy list* initialization — `T obj = {val};` — then aggregates would be supported, but it would be an error if an explicit constructor is the best match, rather than considering the non-**`explicit`** constructors for the best viable match, and it would be an error to rely on a narrowing conversion; `initializer_list` constructors are also considered and preferred.

Table 2 summarizes the different initialization types and highlights the options and trade-offs. In general, there is no one true, universal syntax for initialization in generic (template) code. The library author should make an intentional choice among the trade-offs described in this section and document that as part of their contract.

Table 2: Summary of the different initialization types

-bracedinit-inittypes

Initialization Type	Syntax	Aggregate Support	Explicit Constructor Used	Narrowing	`initializer_list` Constructor Used
Copy	`T obj = val;`	only if `U`	no	allow	no
Direct	`T obj(val);`	only if `U`	yes	allow	no
Direct List	`T obj{val};`	yes	yes	error	yes
Copy List	`T obj = {val}`	yes	error if best match	error	yes

Uniform initialization in factory functions

-in-factory-functions

One of the design concerns facing an author of generic code is which form of syntax to choose to initialize objects of a type dependent on template parameters. Different C++ types behave differently and accept different syntaxes, so providing a single consistent syntax for all cases is not possible. Here we present the different trade-offs to consider when writing a factory function that takes an arbitrary set of parameters to create an object of a user-specified type:

```
#include <utility>  // std::forward

template <class T, class... ARGS>
T factory1(ARGS&&... args)
{
    return T(std::forward<ARGS>(args)...);
}

template <class T, class... ARGS>
T factory2(ARGS&&... args)
{
    return T{std::forward<ARGS>(args)...};
}

template <class T, class... ARGS>
T factory3(ARGS&&... args)
{
    return {std::forward<ARGS>(args)...};
}
```

All three factory functions are defined using **perfect forwarding** (see Section 2.1."**??**" on page **??**) but support different subsets of C++ types and may interpret their arguments differently.

`function1` returns a value created by direct initialization but, because it uses parentheses, cannot return an aggregate unless (as a special case) the `args` list is empty or contains exactly one argument of the same type `T`; otherwise, the attempt to construct the return

value will parse as an error.[8]

`function2` returns an object created by direct list initialization. Hence, `function2` supports the same types as `function1`, plus aggregates. However, due to the use of braced initialization, `function2` will reject any types in `ARGS` that require narrowing conversion when passed to the constructor (or to initialize the aggregate member) of the return value. Also, if the supplied arguments can be converted into a homogeneous `std::initializer_list` that matches a constructor for `T`, then that constructor will be selected, rather than the constructor best matching that list of arguments, despite `function2` being called using parentheses (as for any function call).

`function3` behaves the same as `function2`, except that it uses *copy list initialization* so will also produce a compile error if the selected constructor for the return value is declared as **`explicit`**.

There is no one true form of initialization that works best in all circumstances for such a factory function, and it is for library developers to choose (and document in their contract) the form that best suits their needs. Note that the Standard Library runs into this same problem when implementing factory functions like `std::make_shared` or the `emplace` function of any container. The Standard Library consistently chooses parentheses initialization like `function1` in the code example above, and so these functions do not work for aggregates prior to C++20.

Uniform member initialization in generic code

ization-in-generic-code

With the addition of general braced initialization to C++11, class authors should consider whether constructors should use *direct* initialization or *direct list* initialization to initialize their bases and members. Note that as copy initialization and copy list initialization are not options, whether or not the constructor for a given base or member is **`explicit`** will never be a concern.

Prior to C++11, writing code that initialized aggregate subobjects (including arrays) with a set of data in the constructor's member initializer list was not really possible. We could only *default* initialize, *value* initialize, or *direct* initialize from another aggregate of the same type.

Starting with C++11, we are able to initialize aggregate members with a list of values, using aggregate initialization in place of direct list initialization for members that are aggregates:

```
    std::string

struct S
{
    int         i;
    std::string str;
};

class C
{
    int j;
```

[8] Note that C++20 will allow aggregates to be initialized with parentheses as well as with braces, which will result in this form being accepted for aggregates as well.

```
    int a[3];
    S   s;

public:
    C(int x, int y, int z, int n, const std::string t)
    : j(0)
    , a{ x, y, z }  // Ill-formed in C++03, OK in C++11
    , s{ n, t }     // Ill-formed in C++03, OK in C++11
    {
    }
};
```

Note that as the initializer for `C.j` shows in the code example above, there is no requirement to consistently use either braces or parentheses for all member initializers.

As with the case of factory functions, the class author must make a choice for constructors between adding support for initializing aggregates vs. reporting errors for narrowing conversion. Since member initialization supports only *direct* list initialization, there is never a concern regarding **`explicit`** conversions in this context:

```
    std::forward


template <class T>
class Wrap
{
    T data;

    template <class... ARGS>
    Wrap(ARGS&&... args)
    : data(std::forward<ARGS>(args)...)
        // must be empty list or copy for aggregate T
    {
    }
};

template <class T>
class WrapAggregate
{
    T data;

    template <class... ARGS>
    WrapAggregate(ARGS&&... args)
    : data{std::forward<ARGS>(args)...}  // no narrowing conversions
    {
    }
};
```

Again, there is no universal best answer, and an explicit choice should be made and documented so that consumers of the class know what to expect.

ial-pitfalls-bracedinit

lizer-list-constructors

Potential Pitfalls

Inadvertently calling initializer-list constructors

Classes with an `std::initializer_list` constructors (see Section 2.1."??" on page **??**) follow some special rules to disambiguate overload resolution, which contain subtle pitfalls for the unwary. This pitfall describes how overload resolution might (or might not) select those constructors in surprising ways.

When an object is initialized by braced initialization, the compiler will first look to find an `std::initializer_list` constructor that could be called, with the exception that if the braced list is empty, a default constructor (if available) would have priority:

```
#include <initializer_list>  // std::initializer_list

struct S {
  explicit S() {}
  explicit S(int) {}
  S(std::initializer_list<int> iL) { if (0 == iL.size()) {throw 13;} }
};

S a{};            // OK, value initialization
S b = {};         // Error, default constructor is explicit
S c{1};           // OK, std::initializer_list
S d = {1};        // OK, std::initializer_list
S e{1, 2, 3};     // OK, std::initializer_list
S f = {1, 2, 3};  // OK, std::initializer_list
```

In the presence of initializer list constructors, the overload resolution to select which constructor to call will be a two-step process. First, all initializer-list constructors are considered, and only if no matching `std::initializer_list` constructor has been found, non–initializer-list constructors will be considered. This process has some possibly surprising consequences since implicit conversions are allowed when performing the overload matching. It is possible that an `std::initializer_list` constructor requiring an implicit conversion will be selected over a non–initializer-list constructor that does not require a conversion:

```
#include <initializer_list>  // std::initializer_list

struct S
{
    S(std::initializer_list<int>); // #1
    S(int i, char c);              // #2
};

S s1{1, 'a'};  // calls #1, even though #2 would be a better match
```

In the example above, due to braced initialization preferring initializer-list constructors and because `S` has an `initializer_list` constructor that can match the initializer of `s1`, the constructor that would have been a better match otherwise is not considered.

The other possibly surprising consequence is related to narrowing conversion being checked for only *after* the constructor has been selected. This means that an `initializer_list` constructor that matches but requires a narrowing conversion will cause an error even in the

presence of a non`initializer_list` constructor that would be a match without requiring a narrowing conversion:

```
#include <initializer_list>  // std::initializer_list

struct S
{
    S(std::initializer_list<int>); // #1
    S(int i, double d);            // #2
};

S s2{1, 3.2}; // Error, narrowing conversion when attempting to call #1,
              // even though invoking #2 would be well formed
```

In the example above, due to braced initialization first selecting a constructor and then checking for narrowing conversion, the non-initializer-list constructor, which would not require a narrowing conversion, is not considered.

Both of these situations can be resolved by using parentheses or other forms of initialization than brace lists, which do not prefer initializer-list constructors:

```
#include <initializer_list>  // std::initializer_list

struct S
{
    S(std::initializer_list<int>); // #1
    S(int i, char c);              // #2
    S(int i, double d);            // #3
};

S s3(1, 'c'); // calls #2
S s4(1, 3.2); // calls #3
```

This problem often comes up when talking about `std::vector`:

```
    std::vector

std::vector<std::size_t> v1{5u, 13u};  // Possible bug here.
  // If trying to construct a vector of 5 size_t with value 13,
  // The std::initializer_list constructor is preferred over exact match,
  // so we actually construct a vector with 2 values, 5 and 13.

std::vector<std::size_t> v2(5u, 13u);  // OK, calls the normal constructor
```

lose-aggregate-status

Classes with default member initializers lose aggregate status

An aggregate class is a class with no user-provided constructors, no base classes, no virtual functions, and all public data members. Braced initialization of an aggregate matches each member of the brace list to the corresponding member of the aggregate class in the order of declaration. However, if any of the members has a Section 2.1."**??**" on page **??**, then the class ceases to be an aggregate in C++11, and braced initialization will fail to compile because no matching constructor will be found. C++14 fixes this oversight, so this pitfall

should occur only when supporting code across multiple versions of the language, which is typically more of a concern for library maintainers than application developers:

```
struct S
{
    int a;
    int b;
    int c;
};

struct A  // not an aggregate in C++11 (aggregate in C++14, see next section)
{
    int a{1};
    int b{2};
    int c{3};
};

S s1 = {};         // OK, aggregate initialization, value initializes each member
S s2 = {1, 2, 3};  // OK, aggregate initialization

A a1 = {};         // OK, value initialization with implicit default ctor in C++11
A a2 = {4, 5, 6};  // Error, no matching constructor in C++11
```

ed-in-return-statements

Implicit move and named return value optimization may be disabled in `return` statements

Using extra braces in a return statement around a value may disable the named return value optimization or an implicit move into the returned object.

Named return value optimization (NRVO) is an optimization that compilers are allowed to perform when the operand of a return statement is just the name (id-expression) of a nonvolatile local variable (an object of automatic storage duration that is not a parameter of the function or a catch clause) and the type of that variable, ignoring cv-qualification, is the same as the function return type. In such cases, the compiler is allowed to elide the copy implied by the return expression and initialize the return value directly where the local variable is defined. Naturally this applies only to functions returning objects, not pointers or references. Note that this optimization is allowed to change the meaning of programs that may rely on observable side effects on the elided copy constructor. Most modern compilers are capable of performing this optimization in at least simple circumstances, such as where there is only one return expression for the whole function.

In the example below, we see that the `no_brace()` function returns using the name of a local variable from within that function. As we call `no_brace()` we can observe (with a compiler that performs the optimization) that only one object of the `S` class is created, using its default constructor. There is no copy, no move, and no other object created. Essentially the local variable, `a`, inside the `no_brace()` function is created directly in the memory region of the variable `m1` of the `main()` function.

In the `braced()` function, we use the exact same local variable, but in the return statement we put braces around its name; therefore, the operand of the return is no longer a name (id-expression), and so the rules that allow NRVO do not apply. By calling `braced()`,

we see that now two copies, and so two objects, are created, the first being `a`, the local variable, using the default constructor, and the second being `m2`, which is created as a copy of `a`, demonstrating that NRVO is not in effect:

```
#include <iostream>  // std::cout

struct S
{
    S()          { std::cout << "S()\n"; }
    S(const S &) { std::cout << "S(copy)\n"; }
    S(S &&)      { std::cout << "S(move)\n"; }
};

S no_brace()
{
    S a;
    return a;
}

S braced()
{
    S a;
    return { a };  // disables NRVO
}

int main()
{
    S m1 = no_brace();  // S()
    S m2 = braced();    // S(), S(copy)
}
```

Implicit move (see Section 2.1."**??**" on page **??**) in a return statement is a more subtle operation, so much so that it required a defect report[9] to actually make it work as the original intention. We demonstrate implicit moves in a return statement from a local variable by using two types. The class type `L` will be used for the local variable, whereas the class type `R`, which can be move- or copy-constructed from `L`, is used as the return type. Essentially, we are forcing a type conversion in the return statement, one that may be a copy or a move.

The `no_brace()` function just creates a local variable and returns it. By calling the function, we observe that an `L` object is created, which is then moved into an `R` object. Note that the wording of the ISO standard allows this implicit move only if the return statement's operand is a name (an id-expression).

The `braced()` function is identical to the previous one, except for adding curly braces around the operand of the return statement. Calling the function shows that the *move-from-*`L` return expression turned into a *copy-from-*`L` return expression because a braced initializer is not a name of an object:

```
#include <iostream>  // std::cout
```

[9] **?**

```
struct L
{
    L()        { std::cout << "L()\n"; }
};

struct R
{
    R(const L &) { std::cout << "R(L-copy)\n"; }
    R(L &&)      { std::cout << "R(L-move)\n"; }
};

R no_brace()
{
    L a;
    return a;
}

R braced()
{
    L a;
    return { a };  // disables implicit move from l
}

int main()
{
    R r1 = no_brace();  // L(), R(L-move)
    R r2 = braced();    // L(), R(L-copy)
}
```

ng-deleted-constructors

Surprising behavior of aggregates having deleted constructors

Value initialization of aggregates is allowed with a braced initializer list, even if the default constructor is deleted[10]:

```
struct S
{
    int data;
    S() = delete; // don't want "empty"
};

S s{}; // surprisingly works (until C++20), and 0 == s.data
```

This surprising pitfall occurs for two reasons:

1. A deleted constructor is *user declared* but not *user provided*, so it does not feature in the list of things that stop a class being an aggregate.

[10]Note that C++20 finally addresses the issue so the presence of deleted constructors cause a class to no longer qualify as an aggregate.

2. The rules state that aggregate initialization is not defined in terms of constructors but directly in terms of the initialization of its members.

Annoyances

annoyances

Narrowing aggregate initialization may break C++03 code

-may-break-c++03-code

When compiling existing C++03 code with a C++11 compiler, previously valid code may report errors for narrowing conversion in aggregate (and, therefore, also array) initialization.

```
unsigned u = 128;           // u is computed to an int-friendly value
int ia[] = { 1, 2, u, 9 };  // OK in C++03, narrowing is allowed.
                            // Error in C++11, narrowing conversion.
```

Suppose that the computation in the above code ensures that the value `u` holds at the point of initialization is in the range of values an **`int`** is able to represent. Yet, the code will not compile in C++11 or later modes. Unfortunately each and every case has to be fixed by applying the appropriate type cast or changing the types involved to be "compatible".

No easy way to allow narrowing conversions

narrowing-conversions

In generic code, curly braces have to be used if support for aggregates is required, but, if our interface definition requires supporting narrowing conversions (for example `std::tuple`), there is no direct way to enable them:

```
struct S
{
    short m;
};

class X
{
    S m;

public:
    template <class U>
    X(const U& a) : m{a}  // no narrowing allowed
    {
    }
};

int i;
X x(i);  // Error, would narrow in initializing S.m
```

The workaround is to **`static_cast`** to the target type if it is known or to use parentheses and give up aggregate support in the generic code.[11]

Breaks macro-invocation syntax

cro-invocation-syntax

The macro-invocation syntax of the C++ preprocessor (inherited from C) "understands" parentheses and thus ignores commas within parentheses but does not understand any other

[11] C++20 enables the use of parentheses to initialize aggregates.

list markers, such as braces for braced initialization, square brackets, or the angle bracket notation of templates. If we attempt to use commas in other contexts, the macro parsing will interpret such commas as separators for multiple macro arguments and will likely complain that the macro does not support that many arguments:

```
#define MACRO(oneArg) /*...*/

struct C
{
    C(int, int, int);
};

struct S
{
    int i1, i2, i3;
};

MACRO(C x(a, b, c))         // OK, commas inside parentheses ignored
MACRO(S y{a, b, c})         // Error, 3 arguments but MACRO needs 1
MACRO(std::map<int, int> z) // Error, 2 arguments but MACRO needs 1
```

As the example above demonstrates, on the first macro invocation, the commas within the parentheses are ignored, and the macro is invoked with one argument: `Demo x(a, b, c)`.

In the second macro invocation, we attempt to use braced initialization, but, because the syntax of the preprocessor does not recognize curly braces as special delimiters, the commas are interpreted as separating macro arguments, so we end up with three unusual arguments: first `Demo y{a`, second `b`, and finally the third `c}`. This problematic interaction between braced initialization and macros has existed forever, even back in C code when initializing arrays or **`struct`**s. However, with braced initialization becoming used more widely in C++, it is much more likely that a programmer will encounter this annoyance.

The third invocation of `MACRO` in the example is just a reminder that the same issue exists in C++ with the angle brackets of templates.

The workaround, as is so often the case with the C preprocessor, is more use of the C preprocessor! We need to define macros to help us hide the commas. Such macros will use the **variadic macros** C99 preprocessor feature that was adopted for C++11 to turn a comma-separated list into a braced-initializer list (and similarly for a template instantiation):

```
#define BRACED(...) { __VA_ARGS__ }
#define TEMPLATE(name, ...) name<__VA_ARGS__>

MACRO(X y BRACED(a, b, c));             // OK, X y { a, b, c }
MACRO(TEMPLATE(std::map, int, int) z);  // OK, std::map<int, int> z
```

A common way this annoyance might show up is using the Standard Library `assert` macro:

```
    assert

bool operator==(const C&, const C&);

void f(const C& x, int i, int j, int k)
{
```

```
    assert(C(i, j, k) == x);  // OK
    assert(C{i, j, k} == x);  // Error, too many arguments to assert macro
}
```

not-deduce-array-size

Default member initializer does not deduce array size

Although the syntax looks the same, default member initializers using braced-initializer lists do not deduce the size of an array member:

```
struct S
{
    char s[]{"Idle"};  // Error, must specify array size
};
```

The rationale is that there is no guarantee that the default member initializer will be used to initialize the member; hence, it cannot be a definitive source of information about the size of such a member in the object layout.

per-initializer-lists

No copy list initialization in member initializer lists

The syntax for base and member initializers allows for both direct initialization with parentheses (since C++03) and direct list initialization with braces (since C++11). However, there is no syntax corresponding to copy list initialization, which would allow member initializers to report errors for using an unintended explicit constructor or conversion operator. It would seem relatively intuitive to extend the syntax to support `= { ... }` for member initializers to support such use, but so far there have been no proposals to add this feature to the language. That may be a sign that there is simply no demand, and the authors of this book are the only ones annoyed since this is the only part of the language that supports *direct* initializations without a corresponding syntax for *copy* initializations.

```
class C
{
public:
    explicit C(int);
    C(int, int);
};

class X
{
    C a;
    C b;
    C c;

public:
    X(int i)
    : a(i)       // OK, direct initialization
    , b(i)       // OK, direct list initialization
    , c = (i,i)  // Error, copy list initialization is not allowed.
    {
```

```
    }
};
```

ssed-multiple-arguments

Accidental meaning for explicit constructors passed multiple arguments

In C++03, marking a default or multi-argument constructor explicit, typically as a result of supplying default arguments, had no useful meaning, and compilers did not warn about them because they were harmless. However, C++11 takes notice of the **`explicit`** keyword for such constructors when invoked by copy list initialization. This design point is generally not considered when migrating code from C++03 to C++11 and may require programmers to invest more thought, and potentially split constructors with multiple default arguments into multiple constructors, applying **`explicit`** to only the intended overloads:

```
class C
{
public:
    explicit C(int = 0, int = 0, int = 0);
};

C c0 = {};       // Error, constructor is explicit.
C c1 = {1};      // Error, constructor is explicit.
C c2 = {1,2};    // Error, constructor is explicit.
C c3 = {1,2,3};  // Error, constructor is explicit.

class D
{
public:
    D();
    explicit D(int i) : D(i, 0) { }  // delegating constructor
    D(int, int, int = 0);
};

D d0 = {};       // OK
D d1 = {1};      // Error, constructor is explicit.
D d2 = {1,2};    // OK
D d3 = {1,2,3};  // OK

C f(int i, C arg)
{
    switch (i)
    {
        case 0: return {};         // Error, constructor is explicit.
        case 1: return {1};        // Error, constructor is explicit.
        case 2: return {1, 2};     // Error, constructor is explicit.
        case 3: return {1, 2, 3};  // Error, constructor is explicit.
    }
}

D g(int i, D arg)
```

```
{
    switch (i)
    {
      case 0: return {};         // OK
      case 1: return {1};        // Error, constructor is explicit.
      case 2: return {1, 2};     // OK
      case 3: return {1, 2, 3};  // OK
    }
}

void test()
{
    f(0, {});         // Error, constructor is explicit.
    f(0, {1});        // Error, constructor is explicit.
    f(0, {1, 2});     // Error, constructor is explicit.
    f(0, {1, 2, 3});  // Error, constructor is explicit.

    g(0, {});         // OK
    g(0, {1});        // Error, constructor is explicit.
    g(0, {1, 2});     // OK
    g(0, {1, 2, 3});  // OK
}
```

Note that this topic is deemed an annoyance, rather than a pitfall, because it affects only newly written C++11 (or later) code using the new forms of initialization syntax, so it does not break existing C++03 code recompiled with a more modern language dialect. However, also note that many containers and other types in the C++ Standard Library inherited such a design and have not been refactored into multiple constructors (although some such refactoring occurs in later standards).

Obfuscation due to opaque use of braced-list

ue-use-of-braced-list

Use of braced initializers for function arguments, omitting any hint of the expected object type at the call site, requires deep familiarity with functions being called in order to understand the actual types of arguments being initialized, especially when overload resolution must disambiguate several viable candidates. Such usage may produce more fragile code as further overloads are added, silently changing the type initialized by the brace list as a different function wins overload resolution. Such code is also much harder for a subsequent maintainer, or casual code reader, to understand:

```
#include <initializer_list>  // std::initializer_list

struct C
{
    C(int, int) { }
};

int test(C, long) { return 0; }
```

```
int main()
{
    int a = test({1, 2}, 3);
    return a;
}
```

This program compiles and runs, returning the intended result. However, consider how the behavior changes if we add a second overload during subsequent maintenance:

```
#include <initializer_list>  // std::initializer_list

struct C
{
    C(int, int) { }
};

int test(C, long) { return 0; }

struct A  // additional aggregate class
{
    int x;
    int y;
};

int test(A, int) { return -1; }  // overload for the aggregate class

int callTest1()
{
    int a = test({1, 2}, 3);     // overload resolution prefers the aggregate
    return a;
}
```

Because the overload for `A` must now be considered, overload resolution may pick a different result. If we are lucky, then the choice of the `A` and `C` overloads becomes ambiguous, and an error is diagnosed. However, in this case, there was an integer promotion on the second argument, and the new `A` overload is now the stronger match, producing a different program result. If this overload is added through maintenance of an included header file, this code will have silently changed meaning without touching the file. If the above flexibility is not the desired intent, the simple way to avoid this risk is to always name the type of any temporary variables:

```
int callTest2()
{
    int a = test(C{1,2}, 3);  // Overload resolution prefers struct C.
    return a;
}
```

d-braced-initialization

`auto` deduction and braced initialization

C++11 introduces type inference, where an object's type is deduced from its initialization, using the **`auto`** keyword (see Section 2.1."**??**" on page **??**). When presented with a homoge-

neous, nonempty list using *copy* list initialization, **auto** will deduce the type of the supplied argument list as an `std::initializer_list` of the same type as the list values. When presented with a braced list of a single value using *direct* list initialization, **auto** will deduce the variable type as the same type as the list value:

```
#include <initializer_list>  // std::initializer_list

auto g{1};         // OK, deduces g is int
auto h{1, 2, 3};     // Error, auto requires exactly one element in brace list
auto i = {1};      // OK, deduces i is initializer_list<int>
auto j = {1, 2, 3};  // OK, deduces j is initializer_list<int>
```

Note that the declarations of `i` and `j` in the code example above would also be errors if the `<initializer_list>` header had not been included to supply the `std::initializer_list` class template.

Finally, observe that for **auto** deduction from *direct* list initialization, an `initializer_list` constructor may still be called in preference to copy constructors, even though the syntax seems restricted to making copies:

```
#include <iostream>          // std::cout
#include <initializer_list>  // std::initializer_list

struct S
{
    S() { }
    S(std::initializer_list<S>) { std::cout << "init list\n";}
    S(const S&) { std::cout << "copy\n";}
};

int main()
{
    S s;
    auto s2{s};  // std::initializer_list<S> constructor is called after
                 // deduction. (Note: s2 is deduced to be of type S.)
}
```

s-accept-braced-lists

Compound assignment but not arithmetic operators accept braced lists

Braced initializers can be used to provide arguments to the assignment operator and additionally to compound assignment operators such as `+=`, where they are treated as calls to the overloaded operator function for class types, or as `+= T{value}` for a scalar type `T`.[12]

[12] Although valid, the two `x += {3}` and `x *= {3}` lines in the example compile successfully on Clang but not on GCC or MSVC (applies to all versions at the time of writing). The C++11 standard currently states:

> A braced-init-list may appear on the right-hand side of
>
> — an assignment to a scalar, in which case the initializer list shall have at most a single element. The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of `x=T{v}`. The meaning of `x={}` is `x=T{}`

(**?**, paragraph 9, section 5.17, "Assignment and Compound Assignment Operators," p. 126). There is currently a defect report due to clarify the Standard and explicitly state that this rule also applies to compound

Note that assigning to scalars supports brace lists of no more than a single element and does not support compound assignment for pointer types, since the brace lists are converted to a pointer type, which cannot appear on the right-hand side of a compound assignment operator.

While the intent of compounded assignment is to be semantically equivalent to the expression `a = a + b` (or `* b`, or `- b`, and so on), brace lists cannot be used in regular arithmetic expressions since the grammar does not support brace lists as arbitrary expressions:

```
#include <initializer_list>  // std::initializer_list

struct S
{
    S(std::initializer_list<int>) { }
    S& operator+=(const S&) { return *this; }
};

S operator+(const S&, const S&) { return S{}; }

void demo()
{
    S s1{};               // OK, calls initializer_list constructor
    s1 += {1,2,3};        // OK, equivalent to s1.operator+=({1,2,3})
    s1 = s1 + {1, 2, 3};  // Error, expecting an expression, not an std::initializer_list
    s1 = operator+(s1, {1,2,3}); // OK, braces are allowed as function argument.

    int x = 0;
    x += {3};    // OK, equivalent to x += int{3};.[^cwg_1542].
    x *= {5};    // OK, equivalent to x *= int{5};.[^cwg_1542].

    char y[4] = {1, 2, 3, 4};
    char*p = +y;
    p += {3};    // Error,  equivalent to p += (char*){3};
}
```

see-also

See Also

TODO: Add see also items. Pulled from the feature:

- "??" (§1.1, p. ??) ♦
- "??" (§1.1, p. ??) ♦
- "??" (§1.1, p. ??) ♦
- "??" (§1.1, p. ??) ♦
- "??" (§1.1, p. ??) ♦
- "??" (§2.1, p. ??) ♦

assignments (see **?** and **?**).

- "??" (§2.1, p. ??) ♦
- Implicit move (unknown what this was intended to be)
- "??" (§2.1, p. ??) ♦
- perfect forwarding (unknown what this was intended to be)
- "??" (§2.1, p. ??) ♦
- "??" (§2.1, p. ??) ♦

Further Reading

`further-reading`

TBD

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe

sec-unsafe-cpp11 Intro text should be here.

Chapter 3 Unsafe Features

sec-unsafe-cpp14