Item 7: Distinguish () and {} when creating objects.

Depending on your perspective, syntax choices for object initialization in C++11 embody either an embarrassment of riches or a confusing mess. As a general rule, initialization values may be specified with parentheses, an equals sign, or braces:

```
int x(0);               // initializer is in parentheses
int y = 0;              // initializer follows "="
int z {0};              // initializer is in braces
```

In many cases, it's also possible to use an equals sign and braces together:

```
int z = {0};            // initializer uses "=" and braces
```

For the remainder of this Item, I'll generally ignore the braces-plus-equals-sign syntax, because C++ usually treats it the same as the braces-only version.

The "confusing mess" lobby points out that that the use of an equals sign for initialization often misleads C++ newbies into thinking that an assignment is taking place, even though it's not. For built-in types like `int`, the difference is academic, but for user-defined types, it's important to distinguish initialization from assignment, because different function calls are involved:

```
Widget w1;              // call default constructor
Widget w2 = w1;         // not an assignment; calls copy ctor
w1 = w2;                // an assignment; calls copy operator=
```

Even with several initialization syntaxes, there were some situations where C++98 had no way to express a desired initialization. For example, it wasn't possible to directly indicate that an STL container (e.g., `std::vector<int>`) should be created holding a particular set of values (e.g., 1, 3, and 5).

To address the confusion of multiple initialization syntaxes, as well as the fact that they don't cover all initialization scenarios, C++11 introduces *uniform initialization*: a single initialization syntax that can be used anywhere and can express everything. It's based on braces, and for that reason I prefer the term *braced initializa-*

tion. “Uniform initialization” is a concept. “Braced initialization” is a syntactic construct.

Braced initialization lets you express the formerly inexpressible. Using braces, specifying the initial contents of a container is easy:

```
std::vector<int> v{1, 3, 5};    // v's initial content is 1, 3, 5
```

Braces can also be used to specify default initialization values for non-static data members. This capability—new to C++11—is shared with the “=” initialization syntax, but not with parentheses:

```
class Widget {
  …

private:
  int x{0};                      // fine, x's default value is 0
  int y = 0;                     // also fine
  int z(0);                      // error!
};
```

On the other hand, uncopyable objects (e.g., `std::atomics`) may be initialized using braces or parentheses, but not using "=":

```
std::atomic<int> ai1{0};       // fine

std::atomic<int> ai2(0);       // fine

std::atomic<int> ai3 = 0;      // error!
```

Perhaps now you see why braced initialization is called “uniform.” Of C++’s three ways to designate an initializing expression (braces, parentheses, and “=”), only braces can be used everywhere.

A novel feature of braced initialization is that it prohibits *implicit narrowing conversions*. If the value of an expression in a braced initializer might not be expressible in the type of the object being initialized, the code won’t compile:

```
double x, y, z;

…

int sum1{x + y + z};          // error! sum of doubles may
                              // not be expressible as int
```

Initialization using parentheses and "=" doesn't check for narrowing conversions, because that could break too much legacy code:

```
int sum2 = x + y + z;         // okay (value of expression
                              // truncated to an int)

int sum3(x + y + z);          // ditto
```

Another noteworthy characteristic of braced initialization is its immunity to C++'s *most vexing parse*. A side-effect of C++'s rule that anything that can be parsed as a declaration must be interpreted as one, the most vexing parse most frequently afflicts developers when they want to default-construct an object, but inadvertently end up declaring a function, instead. The root of the problem is that if you want to call a constructor with an argument, you can do it like this,

```
Widget w(10);       // call Widget ctor with argument 10
```

but if you try to call a `Widget` constructor with zero arguments using the analogous syntax, you declare a function instead of an object:

```
Widget w();         // most vexing parse! declares a function
                    // named w that returns a Widget!
```

This trap is particularly odious, because an empty set of parentheses sometimes *does* call a constructor with zero arguments:

```
void f(const Widget& w = Widget());  // w's default value is a
                                     // default-constructed
                                     // Widget
```

Braced initialization eliminates the most vexing parse, yet has no effect on the meaning of initializations that already do what's desired:

```
Widget w{10};       // as before, calls Widget ctor with arg 10

Widget w{};         // now calls Widget ctor with no args

void f(const Widget& w = Widget{});  // as before, w's default
                                     // value is a default-
                                     // constructed Widget
```

There's thus a lot to be said for braced initialization. It's the syntax that can be used in the widest variety of contexts, it prevents implicit narrowing conversions, and it's immune to C++ most vexing parse. A trifecta of goodness, right? So why isn't this Item entitled something like "Use braced initialization syntax"?

The drawback to braced initialization is the sometimes-surprising behavior that accompanies it. Such behavior grows out of the unusually tangled relationship among braced initializers, `std::initializer_lists`, and constructor overload resolution. Their interactions can lead to code that seems like it should do one thing, but actually does another. For example, Item 5 explains that when an `auto`-declared variable has a braced initializer, the type deduced is `std::initializer_list`, even though other ways of declaring a variable with the same initializer would cause `auto` to deduce the type of the initializer:

```
auto v1 = -1;          // -1's type is int, and so is v1's

auto v2(-1);           // -1's type is int, and so is v2's

auto v3{-1};           // -1's type is still int, but
                       // v3's type is std::initializer_list<int>

auto v4 = {-1};        // -1's type remains int, but
                       // v4's type is std::initializer_list<int>
```

In constructor calls, parentheses and braces have the same meaning as long as `std::initializer_list` parameters are not involved:

```
class Widget {
public:
  Widget(int i, bool b);      // ctors not declaring
  Widget(int i, double d);    // std::initializer_list params
  …
};

Widget w1(10, true);          // calls first ctor

Widget w2{10, true};          // also calls first ctor

Widget w3(10, 5.0);           // calls second ctor

Widget w4{10, 5.0};           // also calls second ctor
```

If, however, one or more constructors declares a parameter of type `std::initializer_list`, calls using the braced initialization syntax strongly prefer the overloads taking `std::initializer_lists`. *Strongly*. If there's *any way* for compilers to construe a call using a braced initializer to be to a constructor taking a std::initializer_list, compilers will employ that interpretation. If the Widget class above is augmented with a constructor taking a std::initializer_list<long double>, for example,

```
class Widget {
public:
  Widget(int i, bool b);                           // as before
  Widget(int i, double d);                         // as before

  Widget(std::initializer_list<long double> il);   // added

  …

};
```

Widgets w2 and w4 will be constructed using the new constructor, even though the type of the std::initializer_list elements (long double) is, compared to the non-std::initializer_list constructors, a worse match for both arguments!

```
Widget w1(10, true);        // uses parens and, as before,
                            // calls first ctor

Widget w2{10, true};        // uses braces, but now calls
                            // std::init_list ctor (10 and
                            // true convert to long double)

Widget w3(10, 5.0);         // uses parens and, as before,
                            // calls second ctor

Widget w4{10, 5.0};         // uses braces, but now calls
                            // std::init_list ctor (10 and
                            // 5.0 convert to long double)
```

Compilers' determination to match braced initializers with constructors taking std::initializer_lists is so strong, it prevails even if the best-match std::initializer_list constructor can't be called. For example, consider this slightly-revised example:

```
class Widget {
public:
  Widget(int i, bool b);                     // as before
  Widget(int i, double d);                   // as before

  Widget(std::initializer_list<bool> il);   // std::init_list
  …                                          // element type is
};                                           // now bool
```

```
Widget w{10, 5.0};      // error! requires narrowing conversions
```

Here, compilers will ignore the first two constructors (the second of which offers an exact match on both argument types) and try to call the constructor taking a `std::initializer_list<bool>`. Calling that constructor would require converting an `int` (`10`) and a `double` (`5.0`) to `bool`s. Both conversions would be narrowing (`bool` can't exactly represent either value), and narrowing conversions are prohibited inside braced initializers, so the call is invalid, and the code is rejected.

If there's no way to convert the types of the arguments in a braced initializer to the type taken by a `std::initializer_list`, compilers fall back on normal overload resolution. For example, if we replace the `std::initializer_list<bool>` constructor with one taking a `std::initializer_list<std::string>`, the non-`std::initializer_list` constructors become candidates again, because there is no way to convert `int`s and `bool`s to `std::string`s:

```
class Widget {
public:
  Widget(int i, bool b);               // as before
  Widget(int i, double d);             // as before

  // std::init_list element type is now std::string
  Widget(std::initializer_list<std::string> il);
  …
};

Widget w1(10, true);      // uses parens, still calls first ctor

Widget w2{10, true};      // uses braces, now calls first ctor

Widget w3(10, 5.0);       // uses parens, still calls second ctor

Widget w4{10, 5.0};       // uses braces, now calls second ctor
```

There are two additional twists to the tale of constructor overload resolution and braced initializers that are worth knowing about:

- **Empty braces mean no arguments, not an empty std::initializer_list.** Specifying constructor arguments with an empty pair of braces is a request to call the default constructor, not a request to call a constructor with an empty `std::initializer_list`:

```
class Widget {
public:
  Widget();                              // default ctor
  Widget(std::initializer_list<int> il); // std::init_list
  …                                      // ctor
};

Widget w1;          // calls default ctor

Widget w2{};        // also calls default ctor
                    // (doesn't create empty std::init_list)

Widget w3();        // most vexing parse! declares a function!
```

If you *want* to call a `std::initializer_list` constructor with an empty `std::initializer_list`, you do it by making the empty braces a constructor argument—by putting the empty braces inside the parentheses or braces demarcating what you're passing!

```
Widget w4({});      // calls std::init_list ctor
                    // with empty list

Widget w5{{}};      // ditto
```

- **Copy and move constructors are called as usual.** Creating an object from another object of the same type always invokes the conventional copying and moving functions:

```
class Widget {
public:
  Widget(const Widget& rhs);             // copy ctor
  Widget(Widget&& rhs);                  // move ctor

  Widget(std::initializer_list<int> il); // std::init_list
                                         // ctor

  operator int() const;                  // convert to int
  …
};

auto w6{w5};                // calls copy ctor, not
                            // std::init_list <int> ctor, even
                            // though Widget converts to int

auto w7{std::move(w5)};     // ditto, but for move ctor
                            // (Item 28 has info on std::move)
```

At this point, with seemingly arcane rules about braced initializers, `std::initializer_list`s, and constructor overloading burbling about in your brain, you may be wondering how much of this information matters in day-to-day programming. More than you might think. That's because one of the classes directly affected is `std::vector`. `std::vector` has a non-`std::initializer_list` constructor that allows you to specify the initial size of the container and a value each of the initial elements should have, but it also has a constructor taking a `std::initializer_list` that permits you to specify the initial values in the container. If you create a `std::vector` of a numeric type (e.g., a `std::vector<int>`) and you pass two arguments to the constructor, whether you enclose those arguments in parentheses or braces makes a tremendous difference:

```
std::vector<int> v1(10, 20);  // use non-std::init_list ctor:
                              // create 10-element std::vector,
                              // all elements have value of 20

std::vector<int> v2{10, 20};  // use std::init_list ctor:
                              // create 2-element std::vector,
                              // element values are 10 and 20
```

But let's step back from `std::vector` and also from the details of parentheses, braces, and constructor overloading resolution rules. There are two primary takeaways from this discussion. First, as a class author, you need to be aware that if your constructor overloads include one or more functions taking a `std::initializer_list`, client code using braced initialization may see only the `std::initializer_list` overloads. As a result, it's best to design your constructors so that the overload called isn't affected by whether clients use parentheses or braces. In other words, learn from what is now considered to be an error in the design of the `std::vector` interface, and design your classes to avoid it.

An implication is that if you have a class with no `std::initializer_list` constructor and you add one, client code using braced initialization may find that calls that used to resolve to non-`std::initializer_list` constructors now resolve to the new function. Of course, this kind of thing can happen any time you add a new function to a set of overloads: calls that used to resolve to one of the old overloads might start calling the new one. The difference with `std::initializer_list` constructor overloads is that a `std::initializer_list` overload doesn't just compete with other overloads, it overshadows them to the point that the other overloads may not even be considered. So add such overloads only with great deliberation.

The second lesson is that as a class client, you must choose carefully between parentheses and braces when creating objects. Most developers end up choosing one kind of delimiter as a default, using the other only when they have to. Braces-by-default folks are attracted by their wide applicability, their prevention of narrowing conversions, and their avoidance of C++'s most vexing parse. Such folks understand that in some cases (e.g., creation of a `std::vector` with a given size and initial element value), parentheses are required. In contrast, the go-parentheses-go crowd embraces parentheses as their default argument delimiter. They're attracted to its consistency with the C++98 syntactic tradition, its avoidance of the `auto`-deduced-a-`std::initializer_list` problem, and the knowledge that their object creation calls won't be inadvertently waylaid by `std::initializer_list` constructors. They concede that sometimes only braces will do (e.g., when creating a container with particular values). Neither approach is rigorously better than the other. My advice is to pick one and apply it consistently.†

If you're a template author, the parentheses-braces duality for object creation can be especially frustrating, because, in general, it's not possible to know which form should be used. For example, suppose you'd like to create an object of an arbitrary type from an arbitrary number of arguments. A variadic template makes this conceptually straightforward:

```
template<typename T,                // type of object to create
         typename... Args>          // types of arguments to use
void doSomeWork(const T& obj, Args&&... args)
{
  create local T object from args...

  …

}
```

There are two ways to turn the line of pseudocode into real code (see Item 30 for information about `std::forward`):

† The examples in this book reveal that I'm a parentheses-by-default person.

```
T localObject(std::forward<Args>(args)...);     // using parens
T localObject{std::forward<Args>(args)...};     // using braces
```

So consider this calling code:

```
std::vector<int> v;
…
doSomeWork(v, 10, 20);
```

If `doSomeWork` uses parentheses when creating `localObject`, the result is a `std::vector` with 10 elements. If `doSomeWork` uses braces, the result is a `std::vector` with 2 elements. Which is correct? The author of `doSomeWork` can't know. Only the caller can.

This is precisely the problem faced by the Standard Library functions `std::make_unique` and `std::make_shared` (see Item 23). These functions resolve the problem by internally using parentheses and documenting this decision as part of their interfaces. This is not the only way of dealing with the issue, however. Alternative designs permit callers to determine whether parentheses or braces should be used in functions generated from a template. A common component of such designs is tag dispatch, which is described in Item 32.[†]

Things to Remember

- Braced initialization is the most widely applicable initialization syntax, it prevents narrowing conversions, and it's immune to C++'s most vexing parse.
- As detailed in Item 5, braced initializers yield `std::initializer_lists` for `auto`-declared objects.
- During constructor overload resolution, braced initializers are matched to `std::initializer_list` parameters, even if other constructors offer seemingly better matches.
- An example of where the choice between parentheses and braces can make a significant difference is creating a `std::vector` with two arguments.

[†] The treatment in Item 32 is general. For an example of how it can be specifically applied to functions like `doSomeWork`, see the 5 June 2013 entry of *Andrzej's C++ blog*, "Intuitive interface — Part I."

- Choosing between parentheses and braces for object creation inside templates can be challenging.