C++11 Rocks

GCC Edition

Alex Korban

Contents

Introduction **9**

C++11 guiding principles 9

Type Inference **11**

`auto` 11

Some things are still manual 12

More than syntactic sugar 12

Why else do we need it? 13

Objection, Your Honor! 14

Diving in 14

A little bit of `auto` history 16

`decltype` 17

Side effects 18

`declval` 19

`auto`, `decltype` - how about both at once? 20

Lambda Expressions **22**

Why do we need this thing? 22

Return type 23

Lambda parameters 24

Lambda body 24

Storing lambdas 25

`std::function` to the rescue 25

References to outside context 26

Closures . . . 27
Capturing in C++11 . . . 27
Capturing by reference . . . 28
Default capture modes . . . 29
Capturing class members . . . 30
Limitations of capturing . . . 32
Mutable lambdas . . . 32
Conversion to function pointers, nested lambdas, recursion . . . 34
How not to shoot yourself in the foot . . . 35
Rules of thumb for lambdas . . . 37
Lambda syntax in all its glory . . . 38

Template Features **39**

Variadic templates . . . 39
What else are variadic templates good for? . . . 40
Back to the example . . . 40
Working with parameter packs . . . 41
Traversing template parameter packs . . . 44
Constraining parameter packs to one type . . . 44
More places to expand a parameter pack . . . 45
Nested variadic templates . . . 45
Multiple parameter packs in function templates, non-type parameter packs 46
Template aliases . . . 47
Using `using` instead of `typedef` . . . 49
Closing angle brackets are officially allowed to tail-gate . . . 49
Local and unnamed types as template arguments . . . 50
`extern` templates . . . 51
Default values for function template parameters . . . 52
Arbitrary expressions in template deduction contexts . . . 53

Class Features **55**

In-class initializers for non-static data members . . . 55

Inheriting constructors . . . 58

Delegating constructors . . . 61

Default methods . . . 63

Deleted methods . . . 65

`override` and `final` . . . 67

Extended friend declarations . . . 68

Nested class access rights . . . 70

The Dream of Uniform Initialization **72**

Embrace the braces . . . 73

`initializer_list` . . . 75

Narrowing conversions . . . 77

Distortions of the uniformity continuum . . . 77

Want a move-only type in your `vector`? . . . 78

`auto` + `{}` . . . 78

`<>` + `{}` = ? . . . 79

Surprising consequences of narrowing . . . 80

What's the verdict? . . . 80

Move Semantics **82**

What are the benefits? . . . 82

How does this stuff work? . . . 83

Revision part 1: lvalue vs. rvalue . . . 83

Revision part 2: const attribute + lvalue/rvalue . . . 85

Revision part 3: reference initialization . . . 86

Rvalue references . . . 87

Overload resolution . . . 88
Implementing move semantics . . . 90
Compiler generated move operations . . . 92
Implementing your own move operations . . . 92
`std::move` . . . 95
Moving it right . . . 95
Rvalue references to `const` values . . . 95
Derived class construction . . . 96
Move construction in terms of assignment . . . 96
Check for self-assignment . . . 97
Don't make move constructors `explicit` . . . 98
Reference qualifiers for member functions . . . 98
Move-only types . . . 101

Perfect Forwarding Problem and Solution **103**
Reference collapsing and rvalues in templates . . . 105
How the `forward` template works . . . 106
Bonus: implementation of `std::move` . . . 109

`constexpr` Mechanism **111**
What else is it good for? . . . 111
What's in a constant expression? . . . 111
`constexpr` variables . . . 111
`const` and `constexpr` . . . 112
`constexpr` functions . . . 112
Literal types . . . 114
A couple more notes on `constexpr` functions . . . 115

Range-based `for` Loop **117**

`nullptr` **120**

What's the advantage over `NULL`? . . . 121

`enum` Changes **122**

Scoped `enums` . . . 122

Specifying the underlying type . . . 123

Forward declaration . . . 123

Compile Time Assertions **127**

Literals **129**

Unicode support and literals . . . 129

Unicode character literals . . . 130

Raw literals . . . 130

User defined literals . . . 132

Types of literals . . . 133

Literal operators for integers . . . 134

Character and string literal operators . . . 138

`noexcept` **140**

`noexcept` for your own functions . . . 140

`noexcept` operator . . . 141

When to use `noexcept` . . . 142

A couple more notes on `noexcept` . . . 143

Explicit Conversion Operators **144**

Inline Namespaces **145**

Why not just add a `using` statement? . . . 146

Alignment **148**

`alignof` 148

`alignas` 148

`sizeof` Applied to Non-static Data Members **150**

Memory Model **151**

Multi-threading related semantics 151

Thread local storage **152**

Thread local initialization 153

Thread local destruction 154

Attributes **155**

Standard attributes 156

POD Types, Trivial Types, and Standard Layout Types **157**

Trivial classes 157

Trivially copyable classes 157

Trivial operations 158

Standard layout types and classes 158

Changed restrictions on `unions` 159

Discriminated unions 161

C99 Compatibility Features **165**

Deprecated and Removed Features **166**

Writing Cross-Platform Code **167**

C++14 **169**

Return type deduction for functions 169

Generic lambdas 170

Extended capturing in lambdas 170

Revised restrictions on `constexpr` functions 171

`constexpr` variable templates 171

More language changes 172

Beyond C++14 172

Conclusion **174**

Contact Information and License Agreement **175**

License agreement 175

Introduction

This book will cover the C++ language features added to the C++11 version of the standard, and their implementation in version 4.8 of the GCC compiler. Some of these features began making their way into the compiler way before the C++11 standard was finalized, starting with GCC 4.3.

However, GCC 4.8 is the first version of the compiler to have completely implemented all the new features and changes added to the language by the C++11 standard. Other compilers have also been improving support for the latest standard, so making use of the majority of C++11 features in cross-platform projects is now a viable option.

I thought it would be useful to describe all of these changes in one place, and this book is the result.

Just to be clear, I won't be talking about *all* of C++ in this book. Instead, I'll talk about the changes since C++03.

Please note that this C++ standard also includes extensive additions and changes to the C++ standard library. However, in this book I will focus specifically on language changes, and only mention a few library additions where they work in tandem with language features.

To compile the example code in this book, you'll need to have GCC 4.8.1 or higher installed on your target platform. My assumption is that you are already familiar with using it to compile C++ code.

I will only mention one compiler switch here. In order to compile code which uses C++11 features, you need to use the `-std=c++11` compiler flag.

C++11 guiding principles

According to Bjarne Stroustrup, *"C++11 feels like a new language: the pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever."*

I think this quote encapsulates the spirit of the changes quite well.

Some of the principles the C++ committee used to guide the development of the new standard include:

- Preferring standard library additions over changes to the language. For example, C++11 acquired a lot of machinery to support concurrency but the vast majority of it is provided in libraries. Even so, there are plenty of changes to the language as well.
- The next principle is to improve abstraction mechanisms, rather than solve narrow use cases.

The other principles are:

- Increasing type safety
- Improving performance
- Maintaining the zero overhead principle, which means no overhead from unused features
- And finally, maintaining backwards compatibility.

The new features and changes span different aspects of the language, from a new form of the `for` loop, to various template improvements.

It's hard to say which features have the most impact. It will depend on your code base and your coding style. I have ordered the new features roughly by how much difference the make to the expressiveness, type safety and performance of your code. I also attempted to minimize forward references to features which I haven't already discussed.

Please bear in mind that the examples in this book are written to illustrate the point being discussed, rather than to show the best possible solution or the best coding style. `using namespace std` is assumed to apply to the examples for brevity. Data member names in classes are prefixed with an underscore.

Type Inference

The first thing I'm going to talk about is type inference. C++11 provides mechanisms which make the compiler deduce the types of expressions. These features allow you to make your code more concise, more flexible, and to remove some of the redundancy in the form of repeated type names.

There are actually two different constructs, each with its own keyword: `auto` and `decltype`. As you will see, there is some overlap between them but generally they are intended for different uses. `auto` is limited to declaring variables, while `decltype` is a more general tool.

`auto`

I'm sure you have been annoyed at least once by having to type out types like this:

```
std::map<std::string, std::vector<int>>::iterator
```

And this isn't even a particularly bad example. The verbosity of type names, particularly when templates are involved, can be quite a drag.

The new keyword `auto` solves this problem by inferring the type of a variable from the initializing expression:

```
auto a = 5;
auto plane = JetPlane("Boeing 737");
cout << plane.model();

for (auto i = plane.engines().begin();
    i != plane.engines().end(); ++i)
    i->set_power_level(Engine::max_power_level);
```

In this example, `a` is going to be an `int`, and `plane` is an instance of the `JetPlane` class. The loop variable `i` is going to be an iterator type. As you can see, working with STL iterators becomes quite a bit nicer this way.

To be precise, the keyword `auto` isn't a new keyword. It has actually been in C++ since the beginning, and used to mean "this is a local variable". However, that meaning has proven to be of no use and has now been removed from the language.

Variables declared with `auto` still have a static type just like any other variables. The compiler needs to have enough information to figure the type out - and this is provided by the initializing expression.

Some things are still manual

Because you need to provide sufficient type information to `auto`, it can't be used everywhere. None of these examples will compile:

```
void invalid(auto i) {}     // error

class A
{
    auto _m;                // error
};

int main()
{
    auto arr[10];           // error
}
```

You can't use `auto` for function parameters or member variables, or declare arrays with `auto` because in these cases there is no initializing expression to provide the type information.

More than syntactic sugar

`auto` isn't merely syntactic sugar, however. In some situations it can be not just inconvenient, but actually difficult or impossible to write C++11 code without `auto`. For example, when the code in a template depends on the types of the arguments, it can be hard to specify the type explicitly. But the compiler can choose the appropriate types once it knows the types of the arguments:

```
template<typename X, typename Y>
void multiply(const X& x, const Y& y)
{
    auto result = x * y; // without auto, it can be hard
                         // to specify the type of result
    // ...
}
```

Why else do we need it?

Let's look at the full list of reasons for using this keyword. In fact, it has many advantages over specifying types explicitly:

- It reduces verbosity of the code and makes it more DRY. DRY stands for "Don't Repeat Yourself". This very useful principle was defined in the book called The Pragmatic Programmer by Dave Thomas and Andy Hunt. Consider that any duplication in code is likely to multiply your work.
- It also promotes a higher level of abstraction and coding to interfaces, similarly to template metaprogramming. If you use `auto`, you are more concerned with what types *do* rather than with type names.
- It helps future-proof the code as it allows types to change without necessarily requiring code that *uses* them to change. This means less work for you in the future!
- It allows for easier refactoring, for example when changing class names or function return types.
- It allows template code to be simpler because now you can use `auto` to avoid specifying some of the intermediate expression types.
- It's much easier to declare variables of undocumented types.
- It allows you to store lambda expressions for later use. I'll talk about lambdas in a later chapter, but the key thing here is that you can't know the type of lambda expressions - so you can't declare a variable to assign a lambda expression to without `auto`.

As you can see, the benefits are numerous. So in line with the recommendation from Scott Meyers, I also suggest always using `auto` unless you require a type conversion, that is, when the type inferred by `auto` wouldn't be the type you want. A type conversion

may be required, for example, when you get some proxy type from a template and you want it to decay.

Objection, Your Honor!

At this point you might be disagreeing. A common objection to `auto` is that it obscures type information and makes code hard to comprehend for a new programmer who is trying to familiarize themselves with the code base. It is a valid consideration, but I believe the benefits I've outlined outweigh it.

The fact that the types of variables are less obvious is mitigated to a large degree by code introspection tools in modern editors and IDEs. You also need to keep in mind that reduced type name noise means that the *structure* of the code and what it actually *does* can come to the fore. This may well make the familiarization process *easier* for that new programmer.

Additionally, it's worth considering that similar objections have been made for a long time about other C++ features such as operator overloading, and yet the consensus is that these features contribute to more powerful and expressive code. I believe that `auto` is in the same camp.

Diving in

Now that you're hopefully convinced of the merits of `auto`, let's look at the details of using it.

Using `auto`, you can declare multiple variables in the same statement, as long as all the initializing expressions result in the same deduced type:

```
auto a = 5.0, b = 10.0;
auto i = 1.0, *ptr = &a, &ref = b; // regular variable, pointer
                                   // & reference declarations
auto j = 10, str = "error";        // error: initializing
                                   // expressions of
                                   // different types
```

The first line defines `a` and `b` as `doubles`. The second line defines a regular variable, a pointer and a reference. In contrast, the third line will cause a compilation error because the initializing expressions of the two variables have different types.

See how on the second line I had to add pointer and reference qualifiers? `auto` infers the unadorned type, so if you want a reference or a pointer, you have to specify it explicitly, like this:

```
map<string, int> index;

const auto j = index;
auto& ref = index;
const auto& cref = index;
auto* ptr = &index;
```

Similarly, you can add `const` and `volatile` qualifiers if you need them.

If you aren't declaring a reference, some type transformations are applied automatically:

- `const` and `volatile` specifiers are removed from the initializing type
- arrays and functions are turned into pointers

Let's look at a couple of examples:

```
const vector<int> values;
auto a = values;        // type of a is vector<int>
auto& b = values;       // type of b is const vector<int>&
```

Here I have a `const` vector called `values`. The type of `a` is going to be `vector<int>` without the `const`. `b` is going to be a `const` reference to the `vector`.

If I have a `volatile` variable, and I assign it to an `auto` variable, `volatile` isn't included in the inferred type:

```
volatile long clock = 0;
auto c = clock;     // c is not volatile!
```

Next, suppose I have an array:

```
JetPlane fleet[10];
auto e = fleet;     // type of e is JetPlane*
auto& f = fleet;    // type of f is JetPlane(&)[10] - a reference
```

When I assign it to a plain `auto` variable `e`, `e` is going to be a pointer. If I add a reference specifier as I've done with `f`, then I'm going to get a reference to the array.

Finally, I can use `auto` with functions:

```
int func(double) { return 10; }
auto g = func;      // type of g is int(*)(double)
auto& h = func;     // type of h is int(&)(double)
```

Similarly to arrays, a plain `auto` declaration gives me a pointer to the function, and if I want a reference, I need to specify it explicitly.

You can use either assignment or copy initialization syntax to initialize your `auto` variables:

```
int i = 10;

auto a = i;
auto b(i);
```

However, generally you should prefer using the assignment syntax due to the way `auto` interacts with `initializer_list`. I'll talk about `initializer_list` later in the book.

One exception to this rule is if the inferred type has an explicit copy constructor:

```
struct Expl
{
    Expl() {}
    explicit Expl(const Expl&) {}
};

Expl e;
auto c = e;  // illegal
```

In this situation you can only use copy initialization.

A little bit of `auto` history

This is all you need to know to use `auto`, and I just want to mention a couple of interesting things before moving on. `auto` happens to have a long history. According to Bjarne

Stroustrup, he had the `auto` feature working all the way back in 1984, but had to remove it because of C compatibility issues. Those issues went away when C++98 and C99 started requiring every variable and function to be defined with an explicit type.

Even in the absence of `auto`, compilers have had the ability to infer types for a long time as they had to figure out the types of template arguments.

```
template<typename T>
void f(T t)
{}

f(expr);            // T is deduced from expr
auto var = expr;    // type of var is deduced from expr,
                    // same as above
```

The mechanism behind the type inference performed for `auto` declarations is the same as that used for templates.

Let's now take a look at the other mechanism for type inference, `decltype`.

`decltype`

The second construct that deals with inferring types is the `decltype` specifier. You pass an expression to it, and it figures out the type of the expression:

```
auto i = 10;
cout << typeid(decltype(i + 1.0)).name(); // outputs "d" which
                                          // stands for "double"
```

In this example, `decltype` infers the type of expression to be `double`, so the `typeid.name` outputs "d".

Compared to `auto`, `decltype` is a more general purpose construct. While `auto` can only be used in definitions, `decltype` is meant to be a type specifier, so the intention is for you to be able to use `decltype(expr)` in place of a type name anywhere. For example, in the following bit of code, I'm using `decltype(a)` instead of `vector<int>`:

```
vector<int> a;
decltype(a) b;   // b is vector<int>, same as a
b.push_back(10);
decltype(a)::iterator itr = a.end();  // itr is
                                      // vector<int>::iterator
```

This characteristic of `decltype` is particularly useful when writing templated functions where the return type depends on the template arguments:

```
template<typename X, typename Y>
auto multiply(X x, Y y) -> decltype(x * y)
{
    return x * y;
}
```

This function definition might look strange to you because it uses the new syntax for declaring functions (with a trailing return type). I'll explain this syntax a bit further down. For now, the point is that in C++11 you have the ability to define a function template with a return type which is derived from its template arguments with the help of `decltype`.

Side effects

`decltype` does *not* evaluate the expression that's given to it. So, for example, here the value of `a` won't change outside the context of `decltype`:

```
auto a = 10;
decltype(a++) b;
cout << a << endl;      // outputs 10 as value of a is unchanged
```

Nonetheless, even though the expression isn't evaluated, `decltype` can still have a potential side effect due to template instantiation. Consider this example:

```
template <int I>
struct Num
{
```

```
    static const auto c = I;
    decltype(I) _member;
    Num() : _member(c) {}
};
int i;
decltype(Num<1>::c, i) var = i;     // type of var is int&
```

I passed an expression with the comma operator to `decltype` in the last statement of this example. The comma operator returns the last argument, so `var` will have the same base type as `i`, but the compiler will still need to instantiate the `Num` template to make sure the expression is valid. This template instantiation is a side effect, which you may sometimes prefer not to happen.

declval

The expression passed to `decltype` has to be valid, even though it isn't evaluated. This means I can have a problem when a class with a private constructor is involved:

```
class A
{
private:
    A();
};

cout << typeid(decltype(A())).name();  // doesn't compile:
                                       // A() is private
```

My class `A` has a private constructor, and I attempt to use the constructor call in `decltype`. Even though the type I want is clearly `A`, this doesn't compile, because the expression isn't valid.

This is where `declval` can come to my rescue. `declval` is a standard template which is provided for just such situations:

```
cout << typeid(decltype(declval<A>())).name();  // OK this time
```

With the addition of `declval`, my `decltype` expression now compiles. Keep in mind that `declval<A>` actually yields an *rvalue reference* to `A`, which is a new type of reference

added in C++11. I will discuss rvalue references in detail when I talk about *move semantics*, another new concept in the language.

One more thing to keep in mind is that `declval` can only be used in an unevaluated operand, for example with `decltype`. If you attempt to use it where it has to be evaluated, you will get a compilation error.

`auto`, `decltype` - how about both at once?

When `auto` and `decltype` are combined in a function definition, you end up with something altogether different. The keyword `auto` can be used in function declarations and definitions to indicate a trailing return type, that is, the new syntax variation where the return type is specified after the parameters. It looks like this:

```
template<typename X, typename Y>
auto multiply(X x, Y y) -> decltype(x * y)
{
    return x * y;
}
```

You've already seen this definition previously when I mentioned using `decltype` to determine the return type of a function.

So why is this a useful feature? The reason for introducing this syntax is that it allows you to declare templated functions where the return type depends on the type of the arguments. With the old syntax, there was a scoping problem. Consider this function template definition:

```
template<typename X, typename Y>
ReturnType multiply(X x, Y y)
{
    return x * y;
}
```

I want `ReturnType` to be the type of the result of `(x * y)` but I have no way of specifying it, because `x` and `y` aren't in scope yet. Because of this, I can't use `decltype` in this fashion:

```
template<typename X, typename Y>
decltype(x * y) multiply(X x, Y y) // x and y in decltype aren't
{                                  // in scope yet!
    return x * y;
}
```

This is where trailing return types are useful. Moving the return type *after* the parameter list allows me to use the parameters in the `decltype` expression, which gives me the return type.

Other than saying "use trailing return type syntax when needed", I won't give you a guideline on when to use it. Some people suggest using the new syntax everywhere because of the stylistic and naming advantages it offers: the function names align nicely, and function declarations are consistent regardless of whether the return type requires `decltype`. However, it's definitely not prevalent at this point, so your code may look odd to other programmers.

That's all I had to say about type inference, so let's move on to the next major feature, lambda expressions.

Lambda Expressions

Lambda expressions, or lambdas for short, were one of the most anticipated features of C++11. They provide a way to define anonymous functions in place, right where you need them. One situation where they are very handy is using STL algorithms:

```
for_each(v.begin(), v.end(),
    [](const JetPlane &jet) { cout << jet.model() << endl; })
```

A lambda expression starts with a pair of square brackets. The brackets are followed by the parameter list and the body of the lambda.

As you can see, lambda definitions look a lot like functions. In the past we had to define a function object and pass it to the algorithm, but now we are encouraged to use lambda expressions instead. In fact, internally lambdas are implemented as classes with `operator()`, i.e. as function objects, so my example will be turned into something like this behind the scenes:

```
class lambda0
{
public:
    void operator()(const JetPlane& jet) const
        { cout << jet.model() << endl; }
};
for_each(v.begin(), v.end(), lambda0());
```

The compiler will generate the appropriate function object and pass an instance of it in place of the lambda expression.

Each lambda expression results in a corresponding class called *closure type*, and its invocation is replaced with an invocation of the closure type's function call operator.

Why do we need this thing?

So what's the benefit of lambdas? Mainly, they allow us to do three things:

- Improve locality
- Reduce boilerplate
- In some cases, express our intentions better.

One example of lambdas leading to more expressive code was suggested by Herb Sutter:

```
auto const_val = some_default_value;
if (some_condition_is_true)
{
    // ... Do some operations and calculate the value
    // of const_val ...
    const_val = calculate();
}
const_val = 1000;  // oops, const_val can be modified later!
```

The idea is that if you have some fairly complex code with branches which initializes a constant variable, you can't really make it constant. However, if you wrap the initialization code in a lambda, it allows you to declare `const_val` a `const`:

```
const auto const_val = [&] {
    auto const_val = some_default_value;
    if(some_condition_is_true)
    {
        // ... Do some operations and calculate the value
        // of const_val ...
        const_val = calculate();
    }
    return const_val;
}(); // lambda is invoked immediately!
```

Note the parentheses following the lambda body. They are there to invoke the lambda immediately.

Return type

Let's get back to the syntax. Lambda expressions have a return type. As long as all the return statements in the lambda body return the same type, or the lambda body doesn't

contain any return statements, the compiler will figure out the return type. Otherwise, you'll get a compilation error.

You can also specify the return type explicitly. This is useful, for example if you return several expressions of different types which are convertible to one type. This is done using trailing return type syntax:

```
[](int i) -> double { if (i > 10) return 0.0; return double(i); }
```

However with lambdas, unlike regular functions, you don't use the `auto` keyword.

Note that the C++ standard actually says that the type should only be deduced if it's `void`, or if the body consists of a single `return` statement. This is now considered a defect, and the next version of the standard should match the current GCC behavior.

Lambda parameters

The next thing I'd like to bring your attention to is parameters.

You can pass parameters to a lambda much like a regular function, with several exceptions:

- you can't specify default values for parameters
- you can't use variable length argument lists
- you can't specify unnamed parameters.

For example, here is a lambda expression which takes two parameters by reference:

```
[](JetPlane& jet, const date_t& date)
{ jet.require_service(date); }
```

Lambdas without parameters may omit the parameter list (that is, the empty parentheses) altogether, unless they are declared `mutable`. I'll talk about mutable lambdas later on in the chapter.

Lambda body

Finally, the body of a lambda is just like a normal function, so it can contain arbitrary code with multiple statements, including `try/catch` blocks and multiple return statements.

Storing lambdas

Sometimes you can pass an anonymous lambda expression straight into a function call. But other times you might want to do something more complex, such as storing a lambda for later use in a variable, or returning it from a function. Let's look at how you can do that.

You can't know the type of the lambda expression, there's simply no way to get it:

```
??? f = [](int i) { return i > 10; };
```

So how can you declare `f`? As I mentioned earlier, this is one of the use cases for `auto`. You simply declare `f` with `auto`, and the compiler infers the type behind the scenes:

```
auto f = [](int i) { return i > 10; };
f(5); // returns false
```

So now you can store a lambda in a variable and invoke it at a later point. However, there are still some limitations. You won't be able to pass a lambda into a function this way, or store it as a class member, because in those cases you need to specify the type explicitly, and `auto` isn't a suitable tool.

`std::function` to the rescue

Fear not, there is a solution. C++11 includes a standard library template called `std::function`. This template can be bound to pretty much anything callable: lambdas, function objects, function pointers and member function pointers. It has a lot of uses in addition to storing lambdas, but that's outside the scope of this book. Let's look at a couple of examples of using `std::function` to store and pass lambdas:

```
#include <functional>

class LambdaStore
{
    function<bool(double)> _stored_lambda;
public:
    function<int(int)> get_abs() const
```

```
    {
        return [](int i) { return abs(i); };
    }

    void set_lambda(const function<bool(double)>& lambda)
    {
        _stored_lambda = lambda;
    }
};
```

In order to use `std::function`, you need to include the `<functional>` header.

In my class `LambdaStore`, I have a member `_stored_lambda` of `std::function` type. `std::function` requires that the type of the function it will be bound to is specified in the declaration. A function's type consists of its return type followed by a list of parameter types in parentheses. In this case, my member can refer to lambdas which take a `double` argument and return a `bool`.

This class also has an example of a function returning a lambda wrapped in `std::function` - `get_abs`.

Finally, I can pass a lambda expression which takes a `double` argument and returns a `bool` to `set_lambda`.

Here is how I could use this class:

```
LambdaStore ls;
ls.set_lambda([](double d) { return d > 0.0; }); // store lambda

auto abs_lambda = ls.get_abs();
abs_lambda(-10);                                 // returns 10
```

I can call `set_lambda` with an anonymous lambda expression. I can assign the return value of `get_abs` to a variable and then invoke the lambda via the variable.

References to outside context

So far we've been dealing with simple lambdas which take some arguments, perform operations on them and return the result.

However, the body of a lambda can also refer to variables from the scope it's defined in. An important consideration that goes along with this is that a lambda is also allowed to exist after the scope it's created in is gone. So what happens in this situation? Should the lambda still be able to refer to external variables then?

The solution offered by computer science is to *capture* variables from the scope where the lambda is defined, and thus form something called a *closure*.

Closures

A closure is a function combined with a referencing environment for the non-local variables of the function. The terminology for this is to say a function is *closed over* its *free* variables, hence the term closure. A free variable is a variable referred to in a function that isn't either local or an argument.

The referencing environment binds the non-local variables to corresponding local variables in the function when the closure is created. This is called *capturing*. When a function is executed later on, the non-local variables in it refer to the variables which were captured.

For full closure support, the referencing environment should extend the lifetime of the free variables to match the lifetime of the closure itself. This is why languages with closure support typically use garbage collection. Of course, garbage collection isn't a required part of C++, so the C++ solution to providing closures deviates from this, offering more control on one hand, and some possibilities for creating undefined behavior on the other.

Capturing in C++11

Let's dig into the details of the C++ solution to using external variables in lambda expressions.

In the simplest case, you do it by specifying the variables to capture in the *lambda introducer* which is the standard's name for the square brackets denoting the start of a lambda. For example:

```
[today](JetPlane& jet) { jet.require_service(today); }
```

The `today` variable will be copied for later use when the lambda expression is executed. This is called capturing *by value*. The compiler will generate a closure type similar to this:

```
class lambda1
{
    date_t _today;
public:
    lambda1(date_t today): _today(today) {}
    void operator()(JetPlane& jet) const
        { jet.require_service(_today); }
};
```

So a copy of `today` is stored in a member of this class (`_today` variable).

Note that you can always refer to *non-local* variables without capturing them:

```
function<bool()> g()
{
    static auto a = 5;
    static auto b = -3;
    return []() { return a + b > 0; };
}
```

In this example, `a` and `b` aren't local to the enclosing scope so it's fine to use them in the lambda.

However, *local* variables always need to be captured explicitly:

```
function<bool()> f()
{
    auto a = 5;
    auto b = -3;
    return [a, b]() { return a + b > 0; };  // won't compile if
                                            // a & b aren't
                                            // captured
}
```

Capturing by reference

Similarly to how you can pass parameters to functions by value or by reference, you can also capture variables either by value, or by reference.

To capture a variable by reference, you prefix its name in the lambda introducer with an ampersand:

```
vector<Person> passenger;
for_each(passengers.begin(), passengers.end(),
    [&jet](const Person& p) { jet.load_passenger(p); });
```

Under the hood, it will result in a class like this:

```
class lambda1
{
    JetPlane& _jet;
public:
    lambda1(JetPlane& jet): _jet(jet) {}
    void operator()(Person& p) const { _jet.load_passenger(p); }
};
```

Note that the member variable `_jet` in this case has a reference type.

You can't capture by `const` reference, but if you're capturing `const` locals by reference, they are effectively `const` references.

If you are capturing multiple variables, you can mix and match capturing by value and by reference in the lambda introducer:

```
int a, b, c, d;
[a, &b, c, &d]() {};
```

Here, `a` and `c` are captured by value, while `b` and `d` are captured by reference.

One caveat with capturing by reference is that it's up to you to to make sure that the references are still valid when you invoke the lambda. If they aren't, the result in undefined behavior.

Default capture modes

Sometimes, you might want to capture multiple variables in the same way, that is, either take a copy of each of them, or use references for all of them. In that case, you can

specify the default capture mode, and you won't need to write an explicit list of captured variables. All the free variables referred to in the lambda will be captured as necessary.

To capture everything by value, use the equals sign in the capture block:

```
int a, b, c, d;
[=]() { return (a > b) && (c < d); };
```

To capture everything by reference, use an ampersand in the capture block:

```
int a, b, c, d;
[&]() { a = b = c = d = 10; }();
```

Just to be clear, using a default mode doesn't mean that *all* the variables from the enclosing scope will be captured, but rather just the ones that are actually *used* within the body of the lambda expression.

C++11 has yet another convenient option. You can specify a default capture mode, and then override it for specific variables, for example like this:

```
int a, b, c, d;
// override default capture by value
[=, &a]() { a = 20; }();

// override default capture by reference
[&, d]() { d = 20; }();  // doesn't compile because d is
                         // explicitly captured by value
```

This is useful when you need to capture a lot of variables, with some captured by value and some by reference. For example, you could simplify the capture block `[&a, &b, &c, x, y, &z]` to a shorter equivalent `[&, x, y]`.

Capturing class members

When you need to capture class members, it's a bit different. Instead of capturing the members themselves, you have to capture `this`:

```
class JetPlane
{
    const int _min_fuel_level;
    vector<Tank> _tanks;
public:
    bool is_fuel_level_safe()
    {
        return all_of(_tanks.begin(), _tanks.end(),
            [this](Tank& t)
                { return t.fuel_level() > _min_fuel_level; });
    }
};
```

I've used a lambda which captures `this` in the `is_fuel_level_safe` method, and it refers to the `_min_fuel_level` member.

When lambdas are created within a member function, their closure types are defined within the member function, making them part of the class and allowing `operator()` to refer to all members of the class.

If you use a default capture mode, it automatically captures `this`, providing access to class members:

```
// another member function in the JetPlane class
bool is_fuel_level_critical()
{
    return any_of(_tanks.begin(), _tanks.end(), [=](Tank& t) {
        return t.fuel_level() <= _min_fuel_level;
    });
}
```

You need to remember this behavior, because if you are returning a lambda from a member function, the code may *look* like it's capturing member variables but in fact it's capturing `this`, and the object that returned the lambda may go out of scope, making `this` invalid. I'll show an example of this shortly, when I talk about avoiding undefined behavior.

Another thing to keep in mind is that `this` is *always* captured by value. So even if I used the ampersand instead of the equals sign in the last example, `this` would still be captured by value.

If you try to capture `this` by reference by explicitly writing `[&this]`, you'll get a compilation error.

Limitations of capturing

Capture lists are only allowed in lambdas declared within a *block scope*. Therefore, the following program is malformed and won't compile:

```
#include <iostream>
auto x = 12;
auto f = [&x]() { return x; };      // error

int main()
{
    std::cout << f() << std::endl;
}
```

The identifiers in a capture list are looked up using the rules for unqualified name lookup within the *reaching scope* of the lambda. The reaching scope is the set of enclosing scopes up to and including the innermost enclosing function and its parameters.

Another limitation is that you can't capture a *move-only type*. This is a new kind of type in C++11. I'll talk about these types as part of discussing move semantics later in the book. One alternative is to store your move-only object in a `shared_ptr` instead, and capture that. `shared_ptr` is a reference counted smart pointer in the C++11 standard library. Of course, you can also write a custom function object.

Mutable lambdas

As you've seen previously, by default the function call operator in the lambda's closure type is declared `const`. So when capturing by value you can't modify the captured variables. This can be limiting in situations where you are trying to write a stateful lambda.

Suppose that you're writing some airplane maintenance software, and you need to construct a `vector` of pairs for an airplane where each pair contains this week's flight hours together with the running total from the start of the year. You could write something like this:

```
vector<pair<int, int>> flight_hours;
// ... flight hours are populated with values ...

int running_total = 100;    // accumulated from previous month
for_each(flight_hours.begin(), flight_hours.end(),
    [running_total](pair<int, int>& x)
        { running_total += x.first; x.second = running_total; });
```

This example is a bit contrived perhaps, but the idea is to capture `running_total` to get an initial value, and then use it to add up the hours from the start of the year from subsequent calls to the lambda.

Trouble is, this code isn't going to compile as is because by default the lambda's `operator()` isn't allowed to modify its member variables. But C++ does provide a solution. You can add the `mutable` keyword to the lambda definition:

```
vector<pair<int, int>> flight_hours;
// ... flight hours are populated with values ...

int running_total = 100;    // accumulated from previous month
for_each(flight_hours.begin(), flight_hours.end(),
    [running_total](pair<int, int>& x) mutable
        { running_total += x.first; x.second = running_total; });
```

Then the function call `operator()` of the closure type will no longer be `const`, allowing you to maintain state within the lambda, and the above example will compile.

One caveat with mutable lambda expressions is that you can't pass them by `const` reference:

```
template <class Func>
void by_const_ref(const Func& f) { f(); }

by_const_ref([] {});                                    // OK

by_const_ref([]() mutable {});                          // OK

string s("executing mutable lambda");
by_const_ref([s]() mutable { cout << s << endl; });  // error
```

The function template `by_const_ref` simply invokes the callable entity passed to it. If I pass a non-mutable lambda, everything is fine.

GCC goes beyond the call of duty and actually allows passing a *mutable* lambda too, as long as it doesn't capture anything, which is why the second call works even though it should be an error according to the standard. But the third call definitely results in an error.

Conversion to function pointers, nested lambdas, recursion

There are a couple more little lambda features I'd like to point out before I talk about avoiding undefined behavior.

The first thing is that non-capturing lambdas can be converted into function pointers:

```
typedef int (*Func)();

Func f = [] { return 10; };

f();  // invoke lambda via function pointer
```

This can be handy if you want to pass a lambda to a function that takes a function pointer to a callback, for example.

However, in the case of lambdas which capture variables, you'll need to wrap them in `std::function`.

The second thing, which is perhaps obvious, is that unlike normal functions, lambdas can be nested. This is useful, for example, when you want to do something with containers of containers:

```
vector<Cabins> cabins;
// ... populate cabins ...
SeatScreenMode mode = public_announcement;

for_each(cabins.begin(), cabins.end(), [=](Cabin& cabin) {
    for_each(cabin.seats().begin(), cabin.seats().end(),
        [=](SeatScreen& seat_screen)
            { seat_screen.set_mode(mode); });
});
```

In the above example, I have a `vector` of airplane cabins, and each cabin in turn provides access to a `vector` of seat screens located inside it. The lambda expression passed to the outer `for_each` call contains another lambda expression which sets the mode of each seat's screen.

Like with any other form of nesting, it's a good idea not to get carried away, so that the code remains readable.

Finally, the third point is that while lambda expressions aren't normally recursive, you can emulate recursion within a lambda with a little help from `std::function`:

```
function<int(int)> fibonacci = [&](int n) -> int
{
    if (n < 1)
        return -1;
    else if (n == 1 || n == 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
};
```

This example implements a Fibonacci sequence. To achieve recursive behavior the lambda is bound to an `std::function` object, and the body of the lambda captures and refers to this object (which it's also bound to).

How not to shoot yourself in the foot

Now you know everything there is to know about writing lambda expressions, so let's discuss how to use them safely, without causing undefined behavior.

Because of the tradeoffs in the C++11 closure implementation, some uses of lambdas can result in undefined behavior. I mentioned a couple of potential traps when I talked about capturing by reference and capturing `this`. Undefined behavior can strike when captured variables go out of scope or get deallocated, and it's particularly easy to end up in a situation like that when you store a lambda to call it later.

Let's look at a few examples.

The first situation is when a variable is captured by reference and goes out of scope by the time the lambda is invoked:

```
function<int()> f;

{
    auto i = 5;
    f = [&i] { return i; };
}

f();   // undefined because i is out of scope
```

Here, I'm assigning a lambda to `f` within a block. My lambda captures `i`, but `i` goes out of scope when execution goes past the block. So when `f` is executed, it results in undefined behavior - because `i` is out of scope. It's up to you to ensure that variables captured by reference have a sufficiently long lifetime.

The second situation is similar, but it involves pointers. When a pointer is captured, perhaps by value, it can be deallocated before the lambda is invoked:

```
function<int()> f;

{
    auto p = new int(10);
    f = [=] { return *p; };
    delete p;
}

f();  // undefined behavior because p has been deleted
```

By the time `f` is called, `p` has been deleted, and dereferencing it in the lambda results in undefined behavior. Similarly to the previous situation, you need to ensure that dynamically allocated variables exist long enough if they are used within lambdas.

The third situation involves capturing `this`:

```
{
    class Plane
    {
        int _capacity;
    public:
```

```
        Plane(int capacity): _capacity(capacity) {}

        function<int()> get_lambda() const
        {
            return [=] { return _capacity; };
        }
    };

    Plane plane(10);
    f = plane.get_lambda();
}

f();  // undefined behavior because *plane* is out of scope now
```

After a cursory glance at this bit of code you might think that `_capacity` is captured by value. However, remember that using a default capture mode implicitly captures `this`, and besides, you can only access class members by capturing `this`.

So in this example, calling `f` once again causes undefined behavior, because the body of the lambda actually refers to `plane._capacity`, and the `plane` object is out of scope by then.

Rules of thumb for lambdas

What else should you keep in mind when writing lambda expressions? When is it a good idea to use lambdas rather than regular functions or old school function objects?

The general guidelines are quite simple:

- Write short and clear lambdas.
- If it's becoming long - that is, more than a few statements - you might need a named function object instead.
- If you need to use the same code in multiple lambdas, don't duplicate it in anonymous lambdas. Instead, factor it out into a stored lambda, a function, or a function object.
- If your lambdas are nested more than two levels deep, once again, you may need to refactor.

Lambda syntax in all its glory

We've looked at all the different facets of lambda expressions, so let's summarize by reviewing the complete syntax:

```
[capture_block](parameter_list)
    mutable exception_spec -> return_type
    { body }
```

The `capture block` can be empty, or contain a default mode specifier and an optional list of captured variables.

The `parameter_list` is much like a normal function parameter list (with a few restrictions). If it's empty, the parentheses can be omitted, unless you're also using the `mutable` specifier.

`mutable` is an optional specifier which allows you to modify variables captured by value.

`exception_spec` is another optional clause which allows you to describe the exception throwing behavior of the lambda. Later in the book, I'll discuss the new `noexcept` specifications introduced in C++11 to replace dynamic exception specifications.

`return_type` is the return type of the lambda (specified using the new trailing syntax). This can be omitted in the cases where the compiler is able to infer the return type.

The `body` is like a normal function body with multiple statements and so on.

Template Features

C++11 introduced a number of changes to the template functionality. The two most significant additions are variadic templates and template aliases. There is also a number of other, smaller changes which can make your life easier if you write template code.

Here is the list of things I'm going to discuss in this chapter:

- Variadic templates, which allow templates to take an arbitrary number of arguments
- Template aliases, which are like `typedefs` with partially bound template parameters
- Proper parsing of multiple closing angle brackets
- Local and unnamed types as template arguments
- Extern templates
- Default values for function template parameters
- Arbitrary expressions in template deduction contexts.

Let's get started.

Variadic templates

In the previous chapter, I mentioned `std::function`, a template which can be used to wrap callable entities, including lambda expressions. This template has to support an arbitrary number of template parameters if it's going to work with *any* callable entity:

```
function<bool(int, double)>
function<int(double, double, double)>
function<void(string, int, string, int, string, int)>
```

This is because generally speaking, functions, lambda expressions and other callable objects can take an arbitrary number of arguments themselves.

Before C++11, you had no choice but to limit a template like this to a *fixed* maximum number of parameters, and its implementation usually involved fairly nasty preprocessor use or a lot of duplicate code.

To solve this problem, C++11 introduces variadic templates - that is, templates which can have an arbitrary number of type or non-type parameters:

```
template<typename Stream, typename... Columns>
class CSVPrinter
{
public:
    void output_line(const Columns&... columns);
    // other methods, constructors etc. not shown
};
```

The purpose of this class template is to output arbitrary CSV files. As it is a variadic template, it can handle any number of columns and an arbitrary mix of integers, floating point numbers, strings, dates and any other types it's able to serialize. It's capable of checking at compile time that it's supplied with the correct data types in the right order.

What else are variadic templates good for?

Broadly speaking, variadic templates simplify metaprogramming and increase the genericity of templates.

More specifically, they can be useful in performing type computation at compile time. They can also be used to generate type structure for classes like `tuple`. `tuple` is a template in the standard library which generalizes the concept of `std::pair` to an arbitrary number of types.

Additionally, variadic templates allow you to implement functions with a variable number of parameters in a type safe manner. A popular example of this is a typesafe version of `printf` which is capable of checking the types of its arguments against the format specifiers at compile time.

Finally, they are needed to perform argument forwarding between functions, which is a very useful thing even if it doesn't come up very often. In particular, they are used to implement perfect forwarding, which I'll talk about later in the book.

Back to the example

Let's have a closer look at the previous example:

```
template<typename Stream, typename... Columns>
class CSVPrinter
{
public:
    void output_line(const Columns&... columns);
    // other methods, constructors etc. not shown
};
```

The main thing to notice is the ellipsis. It will make an appearance in a few places as we examine the syntax and operations on variadic templates.

An ellipsis is used to indicate zero or more occurrences of something.

The first ellipsis in the `CSVPrinter` template (which appears after the `typename` keyword) indicates that `Columns` is a template parameter pack. By the way, the ellipsis is a separate token so you can have whitespace between it and the `typename` keyword. You can also use `class` instead of `typename` - it doesn't matter.

The ellipsis also appears in the declaration of member function `output_line`. This time it indicates a function parameter pack. In both cases, the pack has to appear last in the list.

Working with parameter packs

What can we do with a parameter pack? Not that much - there are only two operations supported: pack expansion and getting the type count. You can also recursively iterate over the list by using templates to separate the first element from the rest.

Let's look at pack expansion first:

```
void output_line(const Columns&... columns)
{
    write_line(validate(columns)...);
}
```

The expansion happens in the call to `write_line`.

The `validate(columns)` expression followed by an ellipsis tells the compiler to expand the function parameter pack given to `output_line` into its elements here. The interesting twist with parameter packs is that you can apply *patterns* when expanding them. This is

best explained with an example. Suppose you instantiate the `CSVPrinter` class (which contains `output_line`) with these types:

```
CSVPrinter<decltype(stream), int, double, string> printer;
```

Its `output_line` method after parameter pack expansion is equivalent to this:

```
void output_line(
    const int& col1,
    const double& col2,
    const string& col3)
{
    write_line(validate(col1), validate(col2), validate(col3));
}
```

So, each argument type is adorned with `const` and a reference qualifier, and each argument passed to `write_line` is wrapped in a call to `validate`. This is the result of the compiler expanding the patterns it found preceding the ellipsis in the template definition.

If you've got a variadic template, you'll probably need to perform recursive traversal of the parameter pack. This is done by splitting off the *head* of the pack and recursively instantiating the template with the rest of it. I can use this technique to implement `write_line` in this manner:

```
template<typename Value, typename... Values>
void write_line(const Value& val, const Values&... values) const
{
    write_column(val, _sep);
    write_line(values...);
}

template<typename Value>
void write_line(const Value &val) const
{
    write_column(val, "\n");
}
```

The first template splits off the head of the parameter list, performs an output operation on it, and then recursively calls itself with the rest of the parameters.

The second template is used to terminate the recursion. In my case, it was convenient to terminate it when one element is left in the type list, because I want to output a newline after the last element. In other situations, you might find it more convenient to finish the recursion on a call with no arguments. Both options work.

By the way, I couldn't apply this recursive traversal directly to `output_line` because otherwise I would lose the static enforcement of the class interface. If you look at the definition of `write_line`, you can see that it can be called with *any* kind of parameter pack. `output_line`, on the other hand, only allows one specific sequence of parameters determined by the types `CSVPrinter` was instantiated with.

The last operation you can perform on a parameter pack is obtaining the number of types in it. For example, I can declare an array of column headers in my `CSVPrinter` class like this:

```
template<typename Stream, typename... Columns>
class CSVPrinter
{
    array<string, sizeof...(Columns)> _headers;
    // ... rest of implementation ...
};
```

`_headers` is the data member which will store column headers. Its type is `array` which is a template from the C++11 standard library representing fixed size arrays. It requires two template arguments: the type of elements it stores, and their count. I've provided the count by using a `sizeof...` expression on the template parameter pack `CSVPrinter` is instantiated with. This expression returns the number of types in the parameter pack.

C++11 doesn't support any other parameter pack operations. You can't directly transform parameter packs into data member declarations, for example, without resorting to the usual metaprogramming techniques for type lists.

There are a few more things which are useful to know with regards to parameter packs: how to traverse them, how to constrain parameter pack elements to one type, and some other places in the code where parameter packs can be expanded. It's possible to nest variadic templates, and function templates can have more than one parameter pack. Let's look at each of these items in turn.

Traversing template parameter packs

Parameter pack expansion works very similarly for both template parameter packs and function parameter packs. For example, you could determine the memory requirements for a set of types like this:

```
template<typename... Types>
struct TupleSize;

template<typename Head, typename... Tail>  // traverse types
struct TupleSize<Head, Tail...>
{
    static const size_t value =
        sizeof(Head) + TupleSize<Tail...>::value;
};

template<> struct TupleSize<>              // end recursion
{
    static const size_t value = 0;
};

const size_t TupleSize<>::value;      // yields 0
TupleSize<int, double, char>::value;  // yields 13
                                      // (on a 32-bit platform)
```

Let's examine what's going on here. The first template declaration is needed to allow template instantiation with zero arguments, which may be handy if you use this template in other templates.

The second template splits off the head of the list, gets its size and recursively instantiates itself with the rest of the list.

The third template ends the recursion when there are no more types left.

The last two lines of the example show how this template can be used.

Constraining parameter packs to one type

You can write a function template which takes a variable number of arguments of the *same* type. For example, if you wanted to create CSV files consisting entirely of strings,

you could add something like this to the `CSVPrinter` class:

```
template<typename... Strings>
void output_strings(const string& s,
    const Strings&... strings) const
{
    write_column(s, _sep);
    output_strings(strings...);
}

void output_strings(const string& s) const
{
    write_column(s, "\n");
}
```

This is very similar to the `write_line` implementation I showed before, which worked with *any* combination of types. But here, the type of the first argument is fixed as `const string&` rather than templated. This ensures that the first parameter - and consequently all the following parameters - are instances of `string`.

More places to expand a parameter pack

Pack expansion can appear in places other than class or function bodies. It can also appear in a member initialization list, lambda captures or a base class list:

```
template<typename... Bases>
class Derived : public Bases...
{};
```

Nested variadic templates

Variadic templates can even be nested. Here is a slightly modified example from the standard:

```
template<typename...> struct Tuple {};
```

```
template<typename... Args1>
struct Zip
{
  template<typename... Args2>
  struct With
  {
    typedef Tuple<pair<Args1, Args2>...> type;
  };
};

typedef Zip<short, int>::With<unsigned short, unsigned>::type T1;
// T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>

typedef Zip<short>::With<unsigned short, unsigned>::type T2;
// error: different number of arguments specified
// for Args1 and Args2
```

As you can see, both `Zip` and `With` structs are variadic templates, and `With` is nested inside `Zip`. The rule is that both parameter packs have to be part of a single expansion, and they have to have the same number of types. Therefore, `T1` in the above example is a valid `typedef`, but `T2` isn't.

Multiple parameter packs in function templates, non-type parameter packs

Last but not least, I want to mention that while class templates can have at most one parameter pack, function templates are allowed more than one. Also, parameter packs aren't limited to types. You can also use them for non-type template parameters. You can see both of these features in the following example:

```
template <size_t... Ns>
struct Indexes
{};

template<typename... Ts, size_t... Ns>
auto cherry_pick(const tuple<Ts...>& t, Indexes<Ns...>) ->
    decltype(make_tuple(get<Ns>(t)...))
{
```

```
    return make_tuple(get<Ns>(t)...);
}

// construct tuple<int, int, const char*, const char*, int, int>
auto data = make_tuple(10, 12012013, "B737", "Boeing 737", 2,
    125000);

// construct a tuple of (10, "B737", 2)
auto even_index_data = cherry_pick(data, Indexes<0, 2, 4>());
```

> **Note:** This example is adapted from Paul Keir's blog post on applying variadic templates to transforming tuples: http://paulkeir.wordpress.com/2012/12/03/building-tuple-indices.

The intention of this code is to extract a subset of values at arbitrary indexes from a standard library `tuple`.

As you can see, `struct Indexes` has a non-type parameter pack. It will be used to specify indexes.

The function `cherry_pick` has two parameter packs: one for types stored in its `tuple` parameter, and the other one for the indexes in its second parameter.

`make_tuple` is a standard library variadic function template which constructs a `tuple` from its arguments. `get` is another template which extracts a value from a `tuple` object at the index specified by the template argument. The end result is that `cherry_pick` returns a `tuple` constructed from a subset of values in the input `tuple`.

That's all I have to tell you about variadic templates! If you rely on metaprogramming a lot, you'll probably find a few places in your code which could benefit from this language feature.

Template aliases

The `using` keyword now allows you to create an alias for a template, but unlike a `typedef`, you only have to provide *some* of the template arguments:

```
template<typename T>
using StrKeyMap = map<string, T>;

StrKeyMap<int> counters;
```

StrKeyMap is an alias for the map template with the key type set to string. Here is a slightly more elaborate example:

```
template<typename Stream>
struct StreamDeleter
{
    void operator()(Stream* os) const
    {
        cout << "Deleter called" << endl;
        os->close();
        delete os;
    }
};

template<typename Stream>
using StreamPtr = unique_ptr<Stream, StreamDeleter<Stream>>;

{
    StreamPtr<ofstream> p_log(new ofstream("log.log"));
    *p_log << "Log statement";

}  // stream gets closed and deleted here
```

The standard library has a simple smart pointer called unique_ptr which owns an object and deletes it when it goes out of scope. This smart pointer template has support for custom deleters, so I can provide it with a deleter which closes stream objects before deleting them.

The interesting thing is the template alias called StreamPtr. This alias sets the custom deleter type to StreamDeleter, but leaves the object type and the StreamDeleter's template parameter unspecified.

Therefore, the alias is itself a template, but with fewer parameters. You can use aliases to make type names in your code more obvious.

Using an alias is exactly equivalent to using the type it's replacing. If you specialize a template, the specializations are correctly used when you use this template through an alias.

On the other hand, template aliases can't be specialized, there is simply no syntax for it. If you require specialization, then you should combine your alias with a traits class.

Using `using` instead of `typedef`

The `using` keyword can also be used with non-templated types as an alternative syntax for `typedef`:

```
using PlaneID = int;
```

In some cases, it may be more readable:

```
using Func = int(*)(double, double);
```

You may find that when this syntax is used to create aliases for function pointers or references, it's easier to see the alias name compared to `typedef`.

Closing angle brackets are officially allowed to tail-gate

This is a simple but very convenient change, which applies when you use a template type as a template argument, like this for example:

```
vector<vector<int>> v;
map<int, vector<vector<int>>> m;
```

You no longer need to insert a space between the angle brackets to provide a well-mannered distance between them. The compiler can parse these bracket sequences correctly without spaces in complete agreement with the standard.

Local and unnamed types as template arguments

This change is also in the "simple but convenient" bucket. C++11 allows you to use local types as template arguments:

```
{
    struct A
    {
        string name() { return "I'm A!"; }
    };

    vector<A> v(10);
    cout << v[0].name() << endl;
}
```

Here, `struct A` is a local type, and I'm now able to create a `vector` of `A` instances.

Another new possibility is to use unnamed types as template arguments:

```
template <typename T>
void print(const T& t)
{
    t.print();
}

struct
{
    int x;
    void print() const
    {
        cout << x;
    }
} a;

print(a);
```

The variable `a` which I pass to function template `print` is an instance of an unnamed `struct`.

The definition of unnamed types includes `enums` in addition to classes/`structs`. Note that if you have two unnamed types with the same definition in different translation units, it results in two different template instantiations.

These two changes allow you to improve locality of your code, and to reduce the potential for name clashes.

`extern` templates

The way `extern` works for templates is similar to how it works for ordinary declarations. Its purpose is to speed up compilation time and to reduce object file sizes by preventing redundant template instantiations. Consider what happens if you have the following code:

```
// file1.h
template<typename T>
T templated_func(const T& t)
{
    return t;
}
```

```
// file1.cpp
#include "file1.hpp"
void f()
{
    cout << templated_func(10);
}
```

```
// file2.cpp
#include "file1.hpp"
void g()
{
    cout << templated_func(0);
}
```

The header file contains a function template, and the two `.cpp` files instantiate it. Both object files the compiler generates will contain an instantiation of `templated_func<int>`, and the compiler would have spent the time to generate each instantiation. However, one of them will be discarded by the linker.

You can use `extern` to prevent template instantiation in one of the files:

```
// file1.cpp
#include "file1.hpp"
void f()
{
    cout << templated_func(10);
}
```

```
// file2.cpp
#include "file1.hpp"

extern template int templated_func(const int&);

void g()
{
    cout << templated_func(0);
}
```

Now the template will only be instantiated in `file1.cpp`. This qualifier can also be used for classes (in which case it applies to all members), or even for individual members:

```
extern template vector<int>;

extern template vector<int>::size_type vector<int>::size() const;
```

If you don't provide an instantiation in any of the translation units, the linker will produce an error.

Default values for function template parameters

Before C++11, you could provide default values for class template parameters, but not for function templates. This has been rectified, so you can create templates like this one:

```
template<typename T, typename Cont = vector<T>>
Cont get_batch(T seed)
```

```
{
    Cont batch;
    // ... populate batch using seed value ...
    return batch;
}
```

The second template parameter provides a default value for the return type.

Here is how this function can be used:

```
vector<int> batch_vector = get_batch(1234);

list<double> batch_list = get_batch<double, list<double>>(50.37);
```

On the first line, the first template parameter is deduced from the function argument, and the default value is used for the second parameter.

On the second line, the return type is provided explicitly, so the function returns a different type of a container.

Arbitrary expressions in template deduction contexts

Now that sounds like a bit of a mouthful! What it means is that the standard has been generalized to allow code like the following to compile:

```
template <int I> struct A
{
    static int size() { return I; }
};

int f(int);
double f(double);

template <typename T>
A<sizeof(f((T)0))> calc_size(T)
{
    return A<sizeof(f((T)0))>();
```

```
}

calc_size(1)::size();
```

Note the template argument given to `A`. Overload resolution between the two different versions of `f` is required to figure out the return type of `calc_size`.

The idea is that only specific conditions produce hard errors in template deduction contexts, while everything else is treated as a case of *substitution failure isn't an error* (SFINAE). This allows some more complex use cases for templates to work.

The specific situations which cause an error are:

- errors that occur while processing some entity external to the expression, e.g. an instantiation of a template or the generation of the definition of an implicitly-declared copy constructor
- errors due to implementation limits
- errors due to access violations.

This covers all of the template related changes. In the next chapter, I'm going to talk about additions and changes to classes.

Class Features

For classes, C++11 makes a number of changes related to constructors and initialization, adds keywords for additional control over inheritance and compiler-generated functions, and incorporates a few smaller changes such as access rights for nested classes.

To be more specific, this is the list of features:

- In-class initializers for non-static data members
- Delegating constructors which call other constructors
- Inheriting constructors from a base class
- Control over default methods
- Marking methods as uncallable with the `delete` keyword
- `override` and `final` specifiers
- Extended `friend` declarations
- Nested class access rights.

The rest of the chapter explains these features.

In-class initializers for non-static data members

C++11 gives you a new option for initializing non-static data members:

```
class JetPlane
{
public:
    string _model = "Unknown";
};
```

Instead of initializing members in constructors, you can add an in-class initializer to the member declaration.

You can use either the assignment form or curly braces (which in this context denote the new uniform initialization syntax), but not parentheses:

```
class JetPlane
{
public:
    string _model = "Unknown";
    vector<Engine> _engines {2};
};
```

This is useful when you have several constructors and they initialize some of the members with the same expressions:

```
class JetPlane
{
    vector<Engine> _engines;
    string _manufacturer;
    string _model;
public:
    JetPlane() : _engines(2), _manufacturer("Unknown"),
        _model("Unknown")
    {}

    JetPlane(const string& manufacturer) : _engines(2),
        _manufacturer(manufacturer), _model("Unknown")
    {}
};
```

Previously you had to duplicate initialization code between these constructors. Now you can tidy things up with a member initializer instead:

```
class JetPlane
{
    vector<Engine> _engines {2};
    string _manufacturer {"Unknown"};
    string _model {"Unknown"};
public:
    JetPlane()
    {}
```

```
    JetPlane(const string& manufacturer) :
        _manufacturer(manufacturer)
    {}
};
```

You can also use other members or member function calls in the initializing expression:

```
class JetPlane
{
public:
    string _manufacturer = "Unknown";
    string _model = "Unknown";
    vector<Engine> _engines {get_engine_count(_manufacturer,
        _model)};

    static size_t get_engine_count(
        const string& manufacturer,
        const string& model);
};
```

Note the initializer for the `vector`. It calls the static member function `get_engine_count` with the other two data members as arguments.

Generally, the initializing expression can contain any names which are in scope at that point. However, when using member functions, bear in mind that the members they use can still be uninitialized at that point. The initialization is done in declaration order.

Another thing to remember is that the type is no longer an aggregate if it has in-class initializers, so you can't do this:

```
struct Counter
{
    int _count = 1;
};

Counter c = {10};  // error
```

What happens if you use an in-class initializer, and then initialize the member with some other value in a constructor? Here is a bit of code to illustrate this:

```
class JetPlane
{
public:
    vector<Engine> _engines {2};

    JetPlane() : _engines(4)
    {}
};
```

In this situation, the initializer in a constructor overrides the in-class initializer, so the number of engines will be set to 4.

Inheriting constructors

You know how you have to reimplement constructors in derived classes? Suppose you have a class `Plane` which looks like this:

```
class Plane
{
    vector<Engine> _engines;
    string _manufacturer;
    string _model;
public:

    Plane(const string& manufacturer) :
        _engines(2),
        _manufacturer(manufacturer),
        _model("Unknown")
    {}

    Plane(const PlaneID& tail_number) :
        _engines(Lookup::engine_count(tail_number)),
        _manufacturer(Lookup::manufacturer(tail_number)),
        _model(Lookup::model(tail_number))
    {}
};
```

You could have a derived class `JetPlane` which is constructed in the same way but behaves differently in some respects:

```
class JetPlane : public Plane
{
public:
    JetPlane(const string& manufacturer) : Plane(manufacturer)
    {}

    JetPlane(const PlaneID& tail_number) : Plane(tail_number)
    {}
};

JetPlane plane("Boeing");
```

You have to do quite a bit of typing to define `JetPlane` constructors, even though they don't do anything other than passing their parameters to the base class constructors. Obviously, that's not the best use of your time, so C++11 makes an improvement in this department. In addition to regular functions, you can now apply the `using` directive to base class constructors:

```
class JetPlane : public Plane
{
    using Plane::Plane;
};
```

This `using` statement means that `JetPlane` now has constructors from `Plane` available to it, and they are referred to as inherited constructors.

Inherited constructors keep attributes such as `explicit`, and exception specifications.

If you bring in base class constructors via `using` and also declare an additional constructor in the derived class, it overrides the base class constructor with the same signature:

```
class PropPlane : public Plane
{
public:
    using Plane::Plane;
```

```
    // overrides Plane constructor with the same parameters
    PropPlane(const string& manufacturer) : Plane(manufacturer)
    {
        cout << "In PropPlane()" << endl;
    }
};

PropPlane prop_plane("ATR");  // outputs "In PropPlane()"
```

This behavior can be used to resolve conflicts arising from multiple inheritance. Consider this example:

```
class Boat
{
    string _boat_manufacturer;
public:
    Boat(const string& manufacturer) :
        _boat_manufacturer(manufacturer)
    {}
};

class Plane
{
    string _manufacturer;
public:
    Plane(const string& manufacturer) :
        _manufacturer(manufacturer)
    {}
};

class FloatPlane : public Plane, public Boat
{
    using Plane::Plane;
    using Boat::Boat;
};
```

Given these definitions of `Boat` and `Plane`, the definition of `FloatPlane` isn't going to

compile because both `Boat` and `Plane` classes have a constructor which takes a `string` argument.

But you can override that particular constructor to resolve the ambiguity:

```
class FloatPlane : public Plane, public Boat
{
    using Plane::Plane;
    using Boat::Boat;
    FloatPlane(const string& manufacturer) :
        Plane(manufacturer), Boat("n/a")
    {}
};
```

The rest of the constructors from `Boat` and `Plane` will still be available in the `FloatPlane` class.

It's probably not a good idea to use inherited constructors if you add data members to your derived class, because it's easy to forget to initialize them:

```
class PropPlane : public Plane
{
    size_t _prop_count;
public:
    using Plane::Plane;
};

// oops, _prop_count is not initialized
PropPlane prop_plane("ATR");
```

The effect of this code is the same as if you omitted the derived class data members from the constructor initialization list. You can mitigate the issue by using in-class initializers for the data members in the derived class.

Delegating constructors

Another source of duplication in constructors is when they have to repeat the same initialization code:

```
class JetPlane
{
    JetPlane() :
        _engines(2), _manufacturer("Unknown"), _model("Unknown")
    {
        configure_engines();
    }

    JetPlane(const string& manufacturer, const string& model) :
        _engines(Lookup::engine_count(manufacturer, model)),
        _manufacturer(manufacturer), _model(model)
    {
        configure_engines();
        assign_tail_number();
    }

    // ... rest of the class ...
};
```

And again, C++11 makes things better by allowing a constructor to call another constructor from the same class in the initialization list:

```
class JetPlane
{
public:
    JetPlane() : JetPlane(2, "Unknown", "Unknown")
    {}

    JetPlane(const string& manufacturer, const string& model) :
        JetPlane(Lookup::engine_count(manufacturer, model),
            manufacturer, model)
    {
        assign_tail_number();
    }
private:
    JetPlane(
        size_t engine_count,
        const string& manufacturer,
```

```
        const string& model) :
        _engines(engine_count), _manufacturer(manufacturer),
        _model(model)
    {
        configure_engines();
    }
};
```

Both of the public constructors now call another constructor which takes 3 parameters. Note that the new constructor is private because I don't want the users of the class to create instances that way. The call to `configure_engines` is now only in one place.

A constructor which invokes another constructor like this is called a delegating constructor. If you employ delegation, you can't do anything else in the member initialization list.

The constructor which is delegated to executes first, followed by the body of the delegating constructor.

Be careful not to cause a recursive invocation of a constructor, as that makes the program ill-formed.

Next, let's look at the new keywords to control methods.

Default methods

The compiler can generate a default constructor, destructor, copy constructor and copy assignment operator, and move operations.

However, if you declare your own constructor, it prevents the generation of the default version. If you declare your own copy operations, then the compiler won't generate the move operations, and so on.

But sometimes, it's convenient to preserve some of these default operations, so C++11 provides a way to explicitly request default implementations from the compiler:

```
class JetPlane
{
public:
    JetPlane() = default;
};
```

You can use this to reinstate the default versions of the constructors:

```
class JetPlane
{
public:
    JetPlane() = default;
    JetPlane(const JetPlane& other);
    JetPlane(JetPlane&&) = default;
};
```

In this class definition, I'd like to have my own version of the copy constructor. But when I declare it, it disables compiler-generated implementations of the default constructor and the move constructor. Using the `default` keyword, I can reinstate them.

Not only that, this keyword also allows you to change the declaration at the same time. This is convenient, because the compiler-generated versions are `public`, `inline` and non-`explicit`, which isn't always what you want.

You can make the copy operations `protected`, for example:

```
class JetPlane
{
public:
    JetPlane() = default;
protected:
    JetPlane(const JetPlane&) = default;
    JetPlane& operator=(const JetPlane&) = default;
};
```

Or you can make the default destructor `virtual`:

```
class JetPlane
{
public:
    virtual ~JetPlane() = default;
};
```

Generally, it's better to use the default implementation when suitable rather than providing your own.

Deleted methods

You can also apply the `delete` keyword to methods, which is sort of the opposite to `default`. `delete` means that the function doesn't have an implementation, it's uncallable and can't be used in any way. The first use for it is to disable compiler-generated methods:

```
class JetPlane
{
public:
    JetPlane() = default;
    JetPlane(const JetPlane&) = delete;
    JetPlane& operator=(const JetPlane&) = delete;
    JetPlane(JetPlane&&) = default;
    JetPlane& operator=(JetPlane&&) = default;
};
```

This example disables copying for `JetPlane`, while still allowing moving.

An interesting thing about `delete` is that it can be applied to any function, not just to those generated by the compiler and not just to class methods. This gives it a range of additional uses:

- disabling particular instantiations for a template
- disabling unwanted conversion
- disabling heap allocation.

Let's look at examples.

If you have a template, you can disable instantiations for particular types:

```
template<typename T>
void serialize(const T& obj)
{
    cout << obj.to_string();
};

// PasswordStore is not allowed to be serialized
void serialize(const PasswordStore&) = delete;
```

The result of this code is that `serialize` can't be called on instances of `PasswordStore`.

Similarly, you can disable unwanted conversions:

```
class Altimeter
{
public:
    Altimeter(double);
    Altimeter(int) = delete;
};
```

You'll be able to initialize `Altimeter` instances with `doubles` but not with `ints`.

To disable heap allocation, you need to mark `operator new` deleted:

```
class StackOnly
{
public:
    void* operator new(size_t) = delete;
};

StackOnly inst;             // OK
auto* p = new StackOnly();  // error
```

With this definition of `StackOnly`, you'll be able to create instances on the stack, but you won't be able to allocate it with `new`.

You can mark virtual functions with `delete` too, but bear in mind that any other declarations of that function in the class hierarchy will also need to be deleted.

The key aspect of `delete` is that a deleted function still participates in name lookup. So the difference between not declaring a function and marking it with `delete`, is that in the case of undeclared function, the compiler will continue looking for alternatives. Whereas if it finds an explicitly deleted function, it will stop with an error message. The end result is that `delete` gives you additional control over function lookup.

Next, let's examine the new keywords which control inheritance.

`override` and `final`

There are two new keywords which allow you to have stricter rules around inheritance. Both of these keywords are context sensitive, so they can be used as identifiers outside the context of class or member function declarations:

```
int override = 5;       // OK
int final = 10;         // OK
```

The `override` keyword is applied to a virtual function to tell the compiler that it must be overriding a function in a base class. It helps to prevent errors caused by trying to override a virtual function with a function that has a different parameter list:

```
struct Base
{
    virtual void f(int)
    {}
};

struct Derived : public Base
{
    virtual void f(int) override          // OK
    {}
    virtual void f(double) override       // error
    {}
};
```

The first definition of `f` in the `Derived` class is OK because it overrides the base class definition. But the second definition is going to cause a compilation error, because it's got the `override` keyword and it doesn't override any base class method. Without `override`, it would have simply created a new method.

The `final` keyword can be applied to classes or methods. When it's applied to a class, it disallows inheritance from that class:

```
struct Base final
{};
```

```
struct Derived : public Base  // compile error, can't inherit
{};                           // from a final class
```

Because `Base` is declared `final`, the `Derived` definition will fail to compile.

When `final` is applied to a method, it means that the method can't be overridden in a derived class:

```
struct Interface
{
    virtual void f()
    {}
};

struct Base : public Interface
{
    virtual void f() final
    {}
};

struct Derived : public Base
{
    virtual void f()          // compile error, can't override
    {}                        // a final method
};
```

The definition of `f` in `Derived` will cause a compilation error because `f` was declared `final` in Base.

Extended friend declarations

The rules for non-function friend declarations have been made more flexible, so you have a wider range of options for your classes and templates to make friends with other parts of the code. You can now make a friend declaration without a `class` keyword:

```
class A;
class B;

class Friend
{
    friend class A;   // old declarations are still OK
    friend B;         // you can also do this now
};
```

As you can see in class `Friend`, old style declarations continue to work as well, but there's a difference in behavior between the two styles when they are applied to an undeclared class:

```
class Amigo
{
    friend D;         // error: undeclared class D
    friend class D;   // OK: declares new class D
};
```

In the class `Amigo`, the declaration without the `class` keyword will cause a compilation error, whereas the second declaration *with* the `class` keyword declares a new class called `D`.

In addition, `typedefs` can be declared as friends as long as the `class` keyword is omitted:

```
class B;
typedef B B2;

class Amigo
{
    friend B2;        // OK
};
```

The new style of friend declarations also allows template parameters to be declared as friends:

```
template <typename T, typename U>
class Ami
{
    friend T;             // OK
    friend class U;       // still an error, can't use an
                          // elaborate specifier in a template
};
```

If the template parameter for this template happens to be a built-in type, the friend declaration is simply ignored:

```
Ami<string, string> rc;   // OK
Ami<char, string> f;      // OK, "friend char" has no effect in
                          // the template
```

Nested class access rights

In C++03, members of a nested class didn't have special access to members of an enclosing class. This was a defect in the standard which was rectified in C++11:

```
class JetPlane
{
    // ...
private:
    int _flap_angle;
    class GPSNavigator {};
    class Autopilot
    {
        GPSNavigator _gps_navigator;  // OK, JetPlane::Autopilot
                                      // can access
                                      // JetPlane::GPSNavigator
        void adjust_flaps(JetPlane& plane, int flap_angle)
        {
            // OK, JetPlane::Autopilot can access
            // JetPlane::_flap_angle
            plane._flap_angle = flap_angle;
```

```
        }
    };
};
```

The current rule is that a nested class is a member, and therefore has the same access rights as other members. In the above example, I'm using this rule to declare a `GPSNavigator` member in the nested `Autopilot` class.

I'm also using it to set `_flap_angle` from the `Autopilot` class. Note that `_flap_angle` is a private member of the outer class.

This change allows you to improve encapsulation and locality in your code.

The Dream of Uniform Initialization

The C++ committee made an attempt to make initialization of various types more uniform. Prepare for an onslaught of curly braces as we explore how it works.

First of all, let's consider where initialization was not as smooth or consistent as desired before C++11 came along:

```
class Point
{
public:
    int _x, _y;
};

Point p = {10, 20};
```

If you had a class `Point` without a user defined constructor, you could initialize it as an aggregate. However, as soon as you added a constructor, you had to switch from curly braces to parentheses:

```
class Point
{
public:
    int _x, _y;
    Point(int x, int y) : _x(x), _y(y)
    {}
};

Point p = {10, 20}; // error
Point p(10, 20);    // OK
```

While you could initialize an array on the stack with a list of values, the same courtesy wasn't extended to a dynamically allocated array:

```
int values[] = {1, 2, 3};                // OK
int* p_values = new int[3] {1, 2, 3};    // not going to happen
```

You couldn't initialize a member array in the constructor initialization list either:

```
class Hexagon
{
    int _points[6];

    Hexagon()  // no way to initialize _points
               // in initialization list
    {}
};
```

In addition to these cases, C++03 had a few different variations of syntax for things you *could* initialize.

There is direct initialization, there is copy initialization, and there is brace initialization:

```
int x(10);
int y = 20;
int values[] = {1, 2, 3};
```

The committee's solution to these problems was to permit brace initialization in a lot more situations. It relies on language support, and on the concept of `initializer_list` which allows things like brace initialization of containers.

I'll talk about brace initialization first, and then explain how `initializer_list` fits into it.

Embrace the braces

So, at first glance the picture is a lot more uniform in C++11. All of the examples I've shown you, both working and not working, can now be written using brace initialization:

It can be used to initialize variables of built-in types, and dynamically allocated arrays:

```
int x{5};
int* p_values = new int[3] {1, 2, 3};
```

It can be used for classes with user defined constructors. Note that it works with both direct initialization and copy initialization:

```
class Point
{
public:
    int _x, _y;
    Point(int x, int y) : _x{x}, _y{y}
    {}
};

Point p1{10, 20};
Point p2 = {10, 20};
```

For aggregates, the compiler will perform per-element initialization using values from the list. If there aren't enough elements in the brace list, the rest of the elements are value initialized. If there are too many, then the compiler complains about it.

For non-aggregate classes, for example those with a user defined constructor, the compiler will invoke a constructor with brace list elements as its arguments, as long as it can find a constructor with matching types and number of parameters.

It will also work for member arrays:

```
class Hexagon
{
    int _points[6];

    Hexagon() : _points{1, 2, 3, 4, 5, 6}
    {}
};
```

This syntax extends even further now. C++11 generalizes the concept of aggregate initialization to all user defined types. You can initialize a `vector` or another container with the same syntax you use for arrays:

```
vector<int> v{1, 2, 3, 4};
```

You can even use a brace list in place of a `vector` when returning from a function or passing a `vector` parameter:

```
vector<int> extract_core_points(const vector<int>& v)
{
    return {v.front(), v[v.size() / 2], v.back()};
}

vector<int> core_points = extract_core_points({1, 2, 3, 4, 5});
```

For both of the brace lists in this example, the compiler will figure out that it needs to construct a `vector`.

It turns out that initializing containers like this is done with the help of a standard library template called `initializer_list`, so uniform initialization isn't purely a language feature, it actually extends into library code.

When a user defined type has a constructor which takes an `initializer_list` parameter, the compiler converts the brace delimited list into an instance of `initializer_list` and passes it to that constructor.

`initializer_list`

Most STL headers already include the `initializer_list` header because the standard containers provide `initializer_list` constructors, so you may not need to do it explicitly. But if you do, just include the header:

```
#include <initializer_list>
```

An `initializer_list` instance contains an underlying array of values and provides `begin`, `end` and `size` methods. You can see all of these methods in action in this polygon constructor:

```
Polygon(initializer_list<int> point_indexes)
{
    if (point_indexes.size() < 3)
        throw Error("Polygons require 3 or more points");

    for_each(point_indexes.begin(), point_indexes.end(),
        [=](int index) {
            _points.push_back(Lookup::point(index));
        });
}
```

The standard library code always passes `initializer_list` by value as it's a small object.

There are also overloads of freestanding `begin` and `end`, so you could write the `for_each` call in a slightly different way:

```
Polygon(initializer_list<int> point_indexes)
{
    for_each(begin(point_indexes), end(point_indexes),
        [=](int index) {
            _points.push_back(Lookup::point(index));
        });
}
```

The underlying array is read-only, and `begin` and `end` return pointers to `const`.

`initializer_list` doesn't have a subscript operator, but you can use a pointer returned by `begin` to simulate it:

```
const int* p = point_indexes.begin();
cout << p[1] << endl;
```

Because `initializer_list` is more or less a regular template class, it's not limited to use in constructors. Any function can take an `initializer_list` parameter, and for example standard library containers take advantage of this:

```
vector<int> core_points;
core_points.insert(core_points.end(), {7, 9, 11});
```

You can also see that it doesn't have to be the only parameter. It can happily coexist with other parameters.

Narrowing conversions

I'd like to go back to brace initialization in general. An important property of brace initialization is that it doesn't allow narrowing conversions.

This is a breaking change from the previous version of the standard, although GCC only issues a warning when narrowing happens. For example, this code was fine in C++03 but results in a warning when compiled in C++11 mode:

```
int x[] = {1, 2.5, 3};
```

Basically, narrowing is defined as any kind of implicit lossy conversion, such as going from a floating point type to an integer type. It also includes any implicit conversions where the target type isn't able to represent all values of the source type. The exception to this is if the source expression is constant and the value after conversion can be represented by the target type.

Note that these rules can even prohibit conversions between different floating point types, as well as conversions from integers to floating point types.

Distortions of the uniformity continuum

Unfortunately, the initialization picture isn't entirely rosy. There are several caveats to uniform initialization I have to tell you about. First, consider these two lines of code:

```
vector<int> v1(10);
vector<int> v2{10};
```

What do you expect them to do? `vector` has a number of constructors. One of them takes a count and default-constructs the specified number of elements. Another one

takes an `initializer_list` and initializes the `vector` by copying elements from the initializer list.

When brace initialization is used, the compiler will always pick a constructor taking an `initializer_list` over other constructors. So these two lines of code do different things. Unfortunately, the difference isn't particularly obvious from looking at the code. The first creates a `vector` of 10 default initialized elements. The second creates a `vector` with one element set to 10.

This means that you can't always use brace initialization with a `vector`. Sometimes you'll have to revert to the old syntax.

Because of this, some developers advise avoiding `initializer_list` constructors altogether in your own classes, and relying on variadic templates instead.

Want a move-only type in your `vector`?

The second problem concerns move-only types. I'll explain what they are in the next chapter, so please bear with me for now. I'll just say that for classes which own a resource exclusively it's common to implement move operations but disable copying, thus making them move-only.

The issue is that even though you can store objects of such types in a container, you won't be able to use brace initialization for them:

```
vector<unique_ptr<int>>
    pointers{unique_ptr<int>(new int(1))};          // error

pointers.push_back(unique_ptr<int>(new int(1)));    // OK
```

I mentioned `unique_ptr` before. It's a move-only type and it owns its underlying pointer exclusively. While you can store unique pointers in a container, you won't be able to supply a list of them via a constructor. The old syntax with parentheses is fine, so the `push_back` call works.

`auto` + `{}`

Another hitch is the result of interaction between `auto` and brace delimited lists. Consider these two lines of code:

```
int x = 5;
auto x{5};
```

You might expect them to do the same thing, but they don't. The first obviously defines an `int`. The second, however, defines an instance of `initializer_list` *containing* one `int`. Same as with constructors, the compiler prefers to bind brace lists to `initializer_list`.

For this reason, Bjarne Stroustrup recommends sticking with the first variation of the syntax when using `auto`. This is why I made the same recommendation in the Type Inference chapter.

<> + {} = ?

There are limits on what you can do with initializer lists in a template deduction context.

The type of a brace delimited list can't be deduced for a plain template argument:

```
template<typename T>
void f(T);

f({1});      // error
f({1, 2});   // error
```

You have to specify the type you're passing explicitly.

Similarly, type isn't deduced for a templated container:

```
template<typename T>
void f(const vector<T>&);

f({1, 2, 3});                  // error
f({"Template", "Trouble"});    // error
```

The compiler doesn't guess that you want to create a `vector` containing the elements in braces. Once again, the solution is to be more explicit:

```
f(vector<int>{1, 2, 3});
f<int>({1, 2, 3});
```

Both of these calls provide the compiler with enough information.

This is not the end of the world, of course, but nonetheless it chips away at the idea of uniformity.

Surprising consequences of narrowing

The restriction on narrowing can sometimes result in a surprising behavior compared to old initialization syntax.

For example, this isn't going to compile:

```
uint16_t x{0};
uint16_t y{x + 1};   // error
```

The reason is that the expression `y` is initialized with could potentially result in narrowing.

A similar situation where this can be surprising is this:

```
unsigned int x{true ? 1 : 2};  // OK

bool b{true};
unsigned int y{b ? 1 : 2};     // error
```

The definition of `x` works because it's initialized with a constant expression.

The definition of `y`, on the other hand, will not compile, because the initializing expression isn't constant.

What's the verdict?

The last thing I want to mention is that you can only initialize the first member of a `union` using the uniform initialization syntax. This is unchanged from C++03.

So what's the verdict on uniform initialization? Should it be favored over the previously available syntax? I've seen different opinions from C++ experts, and my personal view is

that unfortunately it still isn't uniform enough to recommend it as the primary option for initialization.

So in my code and in the examples in this book I'll use it only where there is no other option (e.g. initialization of member arrays), or where it obviously saves keystrokes.

Move Semantics

In some situations in C++03 copying was obviously inefficient, but there was no way around it. For example, it's quite typical to insert temporary objects into an STL container:

```
vector<string> v;
v.push_back(string("a"));
v.push_back(string("b"));
```

This results in temporary objects being constructed, copied and destroyed. A `vector` also has to perform reallocation as it grows, which results in copying all the elements each time it happens.

Another example is constructing a `string`:

```
string s = string("Boeing") + "737" + "-" + "300";
```

This statement will cause each call to `operator+` to construct and return a new temporary `string`, which will finally be copied over into `s`.

In order to eliminate such unnecessary copying, C++11 introduces a new mechanism called *move semantics*, which is an addition to copying via copy constructor or assignment operator. Explaining move semantics also requires talking about rvalue references and how they work.

Classes can now have move constructors and move assignment operators. The move operations normally perform a shallow copy of the members and switch ownership to the new members. The original object is required to be left in some valid state, which usually means assigning a value to members which is cheap from a performance standpoint, for example a null value for pointers.

I think this chapter is quite exciting, because the main advantage of this feature is improved performance.

What are the benefits?

Being able to perform a shallow copy when appropriate can result in significant performance improvements compared to copying.

Move operations are particularly beneficial where objects have dynamically allocated members (or some other type of external resources), and where deep copying is involved. While a copy operation will copy the object's memory as well as any additional memory/resources, a move operation only copies the object's memory (because that's all that's necessary).

The second benefit is related to performance as well as clarity of the code. With move semantics, it's possible to return large objects by value without incurring a performance penalty. Consider this code:

```
Surface3D get_surface(const Latitude& lat, const Longitude& lon)
{
    Surface3D surface;
    // ... load up millions of points making up the surface ...
    return surface;
}
```

As long as `Surface3D` provides move operations, this function isn't going to be a performance nightmare.

The last benefit is that move semantics also allow more straightforward implementation of classes which own resources exclusively. Thanks to move semantics, it's now easier to store such classes in standard library containers, for example.

How does this stuff work?

So how do you implement move semantics in your classes? The compiler is able to distinguish between move operations and copy operations through the use of *rvalue references* in move operations. Rvalue references are a new construct in C++11.

Before I start talking about rvalue references, it's a good idea to revise the C++ concept of lvalues and rvalues, as it will help with understanding the behavior of rvalue references later on.

Revision part 1: lvalue vs. rvalue

Every expression in C++ falls into one of two categories: it's either an lvalue or an rvalue. There are two important things about lvalues/rvalues:

- they are attributes of *expressions* (not variables)
- *l* and *r* don't stand for anything in particular (thinking about them as left and right isn't helpful).

There are a couple of rules of thumb to determine whether something is an lvalue. If it has a name (i.e. it's a variable) then it's an lvalue. Alternatively, if you can take its address, then it's an lvalue, otherwise it's an rvalue.

The C++ standard actually says that the operand of the address-of operator must be an lvalue, so taking the address to determine whether something is an lvalue means reversing the definition – but it works.

Here are some examples of lvalues:

```
variable
*ptr
arr[n]
++x
```

For each of these expressions, taking the address makes sense. Now let's try this with some rvalues:

```
&(a * b)
&x++
&string("abc")
```

This time taking the address clearly *doesn't* make sense, which means that all of the above are rvalues.

By the way, `this` pointer is a named expression but it's an rvalue expression by definition.

So, rvalues are temporary values that only exist for the duration of one expression. Lvalues, on the other hand, persist beyond the boundaries of a single expression.

When function or operator calls are part of an expression, you need to look at the return type to figure out whether the expression is an lvalue or an rvalue. If the return type is a reference, then the standard defines it as an lvalue, otherwise it's an rvalue:

```
vector<int> v;
v[0];      // lvalue because vector<int>::operator[] returns int&
v.size();  // rvalue because because vector<int>::size() returns
           // size_t

string s;
s + "abc"; // rvalue because string::operator+ returns string
```

Here, `v[0]` is an lvalue because the index operator returns an `int` reference. `v.size` is an rvalue because the call to size returns `size_t`. Now consider the `s + "abc"` expression. `s + "abc"` is an rvalue because `operator+` returns `string`, which isn't a reference type.

Hopefully this is pretty straightforward, so let's now throw the `const` attribute into the mix.

Revision part 2: const attribute + lvalue/rvalue

Both lvalues and rvalues can have a `const` attribute, which results in 4 types of expressions. Consider these definitions:

```
string f() { return string("F"); }

const string g() { return string("G"); }

JetPlane jet;

const auto max_power_level = 100;
```

The difference between `f` and `g` is that `g` returns a `const` instance. `jet` is not `const` but `max_power_level` is. Let's look at some expressions and classify them:

```
jet;                // lvalue
max_power_level;    // const lvalue
f();                // rvalue
g();                // const rvalue
```

`jet` is an lvalue, `max_power_level` is a `const` lvalue, the call to `f` is an rvalue, and the call to `g` is a `const` rvalue. Thus, we have 4 different types of expressions.

Non-`const` rvalues might seem rather useless as they are going to disappear at the end of the expression, but the reason for their existence is to allow you to call non-`const` member functions on them:

```
// append call modifies the string
cout << jet.model().append("_RR").size() << endl;
```

Here the `model` method returns a temporary `string` object, and I can chain a couple of method calls together to produce the value I need without creating a variable.

Revision part 3: reference initialization

Finally, let's look at initializing references in the context of lvalue- or rvalue-ness. A reference to a non-`const` type can only be initialized with a non-`const` lvalue:

```
JetPlane jet;
JetPlane& jet_ref = jet;                 // OK

const JetPlane grounded_jet;
JetPlane& jet_ref2 = grounded_jet;     // doesn't compile

JetPlane& jet_ref3 = JetPlane();       // doesn't compile

const JetPlane make_const_jet() { return JetPlane(); }
JetPlane& jet_ref4 = make_const_jet(); // doesn't compile
```

My second, third and fourth attempts at initializing a variable in the above example aren't going to compile. `jet_ref2` is initialized with a `const` lvalue. `jet_ref3` is initialized with a non-`const` rvalue. `jet_ref4` is initialized with a `const` rvalue.

A reference to `const`, on the other hand, can be initialized with any of the 4 types of values:

```
JetPlane jet;
const JetPlane& jet_ref = jet;

const JetPlane grounded_jet;
const JetPlane& jet_ref2 = grounded_jet;

const JetPlane& jet_ref3 = JetPlane();

const JetPlane& jet_ref4 = make_const_jet();
```

I've simply added `const` to my variable declarations, and now all 4 examples will compile.

There is just one slightly mind-bending twist you should keep in mind. When you bind a `const` reference to an rvalue, the reference variable you bound it to is in fact an *lvalue expression*:

```
const JetPlane& jet_ref4 = make_const_jet();

jet_ref4;  // jet_ref4 is a const lvalue - it has a name!
```

So here, even though the `make_const_jet` call is an rvalue, the `jet_ref4` expression is an lvalue. That's because it consist of a variable name.

Let's summarize the review:

- Expressions can be categorized as lvalues and rvalues.
- They can also have a `const` attribute, which gives us 4 different types of expressions.
- Non-`const` references can only be initialized with one type of expression - a non-`const` lvalue, while `const` references can be initialized with any of the 4 types of expressions.

With all of these pre-C++11 concepts in mind, we can now look at rvalue references.

Rvalue references

Rvalue references are declared with a double ampersand instead of a single ampersand:

```
JetPlane&& rvalue_ref
```

The regular `T&` references are now called *lvalue references*. Lvalue and rvalue references are distinct types, but they behave similarly: they must be initialized, and they cannot be reassigned.

By the way, a named rvalue reference is an *lvalue expression*, just like a named lvalue reference. As you'll see a bit later, it's an important distinction to keep in mind when implementing move operations.

The difference in initialization from lvalue references is that an rvalue reference can only be initialized with a non-`const` rvalue:

```
JetPlane&& jet_ref = JetPlane();         // OK

JetPlane jet;
JetPlane&& jet_ref2 = jet;               // doesn't compile to
                                         // prevent accidental
                                         // change of an lvalue

const JetPlane grounded_jet;
JetPlane&& jet_ref3 = grounded_jet;      // doesn't compile

JetPlane&& jet_ref4 = make_const_jet(); // doesn't compile
```

Therefore, the first variable definition will compile, but the rest of them will cause the compiler to admonish you with error messages, because the initializer expressions are either lvalues, or a `const` rvalue in the case of `jet_ref4`.

Overload resolution

There are further differences in overload resolution. In C++11, you can have 4 reference-based overloads instead of two:

```
void f(JetPlane& plane);
void f(const JetPlane& plane);
void f(JetPlane&& plane);
void f(const JetPlane&& plane);
```

There are two possible overloads for lvalue references and two more overloads for rvalue references. The overloads are resolved using the following rules:

1. The selected overload has to maintain `const`-correctness (so it's not allowed to bind a `const` value to a non-`const` reference).
2. Lvalues are always bound to lvalue references, and rvalues are bound to rvalue references if possible.
3. Finally, if the second rule wasn't enough to choose one overload, the compiler chooses an overload which preserves `const`-ness.

If you have all 4 overloads, then the resolution is simple:

```
string a = "a";
f(a);           // f(string&) called

const string b = "b";
f(b);           // f(const string&) called

string str() { return string("G"); }
f(str());       // f(string&&) called

const string const_str() { return string("const G"); }
f(const_str()); // f(const string&&) called
```

For the first call, the compiler will select the non-`const` lvalue reference version of `f`.

The second call will be to `f` which takes a `const JetPlane` reference.

The third call will be to the version which takes a non-`const` rvalue reference argument, because the temporary instance of `JetPlane` is an rvalue.

And finally, the fourth call will be to the `const` rvalue reference version because the call to `const_str` is a `const` rvalue as it returns `const JetPlane`.

If, however, the last overload for a `const` rvalue reference wasn't present, then the third overload resolution rule would make a `const` rvalue bind to the `const` *lvalue* overload instead.

This is quite important when the compiler has to choose between a copy and a move operation, as I'll show later, and for this reason you wouldn't normally implement a `const` rvalue reference overload.

These rules may seem a bit arbitrary, but they are crafted to allow the compiler to distinguish between copying and moving by selecting an appropriate function overload based on the type of expression passed as the argument.

Implementing move semantics

Let's have a look at adding move semantics to a type. There are three elements which make it possible.

The first piece of the puzzle is that the compiler is able to distinguish move operations because they have rvalue reference parameters, so a class can have both a copy and a move constructor, which have these signatures:

```
A(const A& rhs);
A(A&& rhs);
```

When overload rules listed in the previous section are applied to these two constructors, it turns out that non-`const` rvalues are passed to the move constructor, while lvalues and `const` rvalues are passed to the copy constructor - thanks to the rule about preserving `const`-ness. This happens to be exactly what's required to be able to swap the ownership of the members.

For an illustration of how this works in practice, let's see what happens when rvalues are pushed into a `vector`. First, let's try a class that doesn't have a move constructor:

```
struct A
{
    A()
    {
        cout << "A's constructor" << endl;
    }
    A(const A& rhs)
    {
        cout << "A's copy constructor" << endl;
    }
};

vector<A> v;
```

```
cout << "==> push_back A():" << endl;
v.push_back(A());
cout << "==> push_back A():" << endl;
v.push_back(A());
```

A only has a default constructor and a copy constructor. If I ran this code, the output would be this:

```
==> push_back A():
A's constructor
A's copy constructor
==> push_back A():
A's constructor
A's copy constructor
A's copy constructor
```

The important thing to note is that in order to insert the second object the copy constructor had to be invoked twice. This is due to the re-allocation performed as the `vector` grows.

Now let's give A a move constructor:

```
A(A&& rhs) noexcept
{
    cout << "A's move constructor" << endl;
}
```

With this in place, I'm going to get different output from running the code:

```
==> push_back A():
A's constructor
A's move constructor
==> push_back A():
A's constructor
A's move constructor
A's move constructor
```

The operations change because `vector` will take advantage of move semantics provided by `A`. The copy constructor is no longer executed. This can make a significant performance difference in real-world code.

Note that I marked the move constructor `noexcept` to tell the compiler it won't throw any exceptions. That's required for GCC's implementation of `vector` to always select the move constructor rather than the copy constructor. I'll talk about `noexcept` later in the book.

You should implement move constructors and move assignment operators in your classes whenever you can. You'll get performance benefits from move semantics not only when your objects are used in your own code, but also when using your classes with STL containers and algorithms, and any other third party libraries that support move semantics.

Compiler generated move operations

The C++11 standard makes provisions for the compiler to generate move operations automatically in some situations. The following conditions have to be fulfilled for this to happen in your class:

- the class doesn't have a user-declared copy constructor or copy assignment operator
- the class doesn't have a user-declared move assignment operator
- the class doesn't have a user-declared destructor
- the move constructor would not be implicitly defined as deleted.

Implementing your own move operations

What we know so far is that the compiler can choose to use move operations when they're available, and it makes code faster. But what do you need to write in the bodies of your move operations? This is the second piece of the puzzle.

The moving is performed member by member. Suppose our class `A` has these members:

```
double _d;
int* _p;
string _str;
```

The move operations need to get rid of the object's current state (if it's initialized), transfer the ownership, and leave the source object in a valid state, which is normally done in the cheapest possible way from a performance standpoint.

For pointers, it normally means assigning `nullptr` to prevent a double `delete`. For our class `A`, the operations should look like this:

```
A(A&& rhs) : _d(rhs._d), _p(rhs._p), _str(move(rhs._str))
{
    rhs._p = nullptr;
    rhs._str.clear();
}

A& operator=(A&& rhs)
{
    delete _p;

    _d = rhs._d;
    _p = rhs._p;
    _str = move(rhs._str);  // careful!

    rhs._p = nullptr;
    rhs._str.clear();

    return *this;
}
```

For built-in types, the process is simple. Just reassign the source value member by member. If it's a pointer, set the source value to `nullptr`. `nullptr` is the C++11 replacement for `NULL`, and I'll explain how it works later in the book.

The move constructor and the move assignment operator implementations are similar, except in the assignment operator I need to `delete _p` before assigning the new value to it.

These operations also show the third piece of the move semantics puzzle, which is dealing with objects of non-built-in types. The `_str` member is an object rather than an instance of a built-in type, which means it's got its own copy and move operations.

But `rhs._str` is a named rvalue reference, and as I mentioned before, that makes it

an lvalue expression. If you simply assigned `rhs._str` to the target `str` member, the overload resolution rules would result in the copy assignment operator being called.

Something needs to be done to inform the compiler that what we want here is a move operation. This is exactly why `rhs._str` is wrapped in a `move` call. I'll talk more about what `move` is and how it works in a minute, but first I want to mention two other things.

One, it is good practice to empty out the source objects, because if `rhs` happens to be an lvalue which is being explicitly moved from, it will continue to exist after the call which invokes the move operation. As `rhs` can be holding some kind of resources, this can lead to subtle problems with resource management.

The second thing is that move semantics aren't reserved to constructors and assignment operators. You can also provide move overloads for setters in a class (or any methods, really). For example, you could write two versions of the `set_model` method like this:

```
void JetPlane::set_model(const string& model)
{
    _model = model;
}

void JetPlane::set_model(string&& model)
{
    _model = move(model); // careful: model is a named rvalue ref
                          // so it's an lvalue expression; use
                          // std::move to force a move operation
    model.clear();
}

string model("Airbus 320");
JetPlane jet;

jet.set_model(model);                   // copy overload used
jet.set_model(string("Airbus 320"));    // move overload used
```

Once again, you need to wrap the `model` variable in a `move` call when assigning it to the `_model` member so the compiler knows you want a move assignment.

If you call `set_model` with an lvalue, then the copy overload will be used. If you call it with an rvalue, for example a temporary object, then the compiler will select the move overload.

`std::move`

So what is the `move` call and how does it force a move operation?

The `move` function is used to tell the compiler to choose a move overload when you have a named rvalue reference, or an lvalue that you know isn't going to be used anymore. Without `move`, the compiler would choose the copy overload.

No lvalue is special, `move` can be applied to any of them. So if you want to move from a regular variable because you know it's going out of scope straight after you use it, you just have to explicitly convert it into an rvalue with `move`:

```
A a;
v.push_back(move(a));
```

That's all `move` does - it converts the lvalue into an rvalue. It doesn't perform any moving, it just converts its argument into an unnamed rvalue reference, which then allows the compiler to choose the correct (move) overload. Instead of `move`, you could use `static_cast<T&&>` but `move` is shorter and clearer.

Moving it right

In general, you have to ensure that your code isn't written in a way that precludes the move operations from being selected by the compiler. I'm going to offer several guidelines for that in this section.

Rvalue references to `const` values

While rvalue references to `const` are legal, they are not useful in the context of move semantics.

Because of the new overload resolution rules, returning a `const` value from a function (which was useless but harmless before) can result in a copy operation being selected instead of a move operation, as a `const` rvalue can't bind to a non-`const` rvalue reference:

```
const JetPlane make_const_jet() { return JetPlane(); }

JetPlane jet(make_const_jet()); // invokes copy constructor even
                                // if JetPlane has move semantics
```

`jet` will be constructed using the copy constructor even if `JetPlane` has a move constructor. It's easy to see why. It's given a `const` rvalue which can't be modified, so consequently it can't be moved from - as that would require modifying the source object. Therefore, don't return `const` objects.

For the same reason, avoid declaring `const T&&` parameters, because you won't be able to move from them.

Derived class construction

If you have a class with move semantics, and you would also like to provide move semantics in the class that derives from it, you need to be careful when implementing the move constructor. Suppose you do this:

```
Derived(Derived&& rhs) : Base(rhs) {}
```

It will result in the `Base`'s *copy constructor* being called. We've seen similar situations before. The reason is that `rhs` is an rvalue with a name, which makes it an lvalue expression.

You need to help the compiler make the right choice by wrapping `rhs` in `move`:

```
Derived(Derived&& rhs) : Base(move(rhs)) {}
```

Now `Base`'s move constructor will be called.

Move construction in terms of assignment

Be careful if you implement move constructors in terms of assignment operators. Don't do this:

```
B(B&& rhs)
{
    *this = rhs; // WRONG: invokes the copy assignment operator
}
```

This approach will result in the copy assignment operator being called, and defeating the purpose of a move constructor. Instead, make sure that the source of the assignment is an rvalue:

```
B(B&& rhs)
{
    *this = move(rhs);
}
```

Basically, this is the same issue as when you invoke a base class constructor.

Check for self-assignment

Suppose that your class `B` has an `int* _p` member and a move assignment operator that looks like this:

```
B& operator=(B&& rhs)
{
    delete _p;
    _p = rhs._p;
    rhs._p = nullptr;
    return *this;
}
```

Moving an instance of `B` into itself will cause `_p` to become null unexpectedly:

```
B b;
b = move(b);
*b._p;        // undefined behavior, _p is null
```

This is what happens when `b` is moved into `b`:

```
B& operator=(B&& rhs)
{
    delete _p;          // b._p is deleted
    _p = rhs._p;        // b._p gets assigned to itself
    rhs._p = nullptr;   // b._p is now set to null!
    return *this;
}
```

First, the object _p points to is deleted. Then the assignment operator assigns b's _p to itself. The third line sets b's _p to `nullptr`. In the end, object state is lost as a result of what looks like a harmless operation.

To prevent such situations, always include a check for self-assignment in move-assignment operators:

```
B& operator=(B&& rhs)
{
    if (this == &rhs)
        return *this;

    delete _p;
    _p = rhs._p;
    rhs._p = nullptr;
    return *this;
}
```

I should point out that not checking for self-assignment in a copy assignment operator doesn't have similarly drastic consequences because the source is left intact.

Don't make move constructors `explicit`

You shouldn't declare the move constructor `explicit`, because it will result in a copy constructor being chosen instead. In addition, some parts of the STL such as the `swap` method, `unique_ptr` smart pointer class, and the `sort` algorithm require types to be move constructible, which means a type with an `explicit` move constructor can't be used with them.

This is the last guideline I had for you. If you follow these guidelines, you will avoid bugs and your classes will take advantage of move semantics whenever possible.

Reference qualifiers for member functions

I'd like to explain one more language feature which allows you to provide different method overloads based on whether they are called on an lvalue or an rvalue.

So far, we've seen rvalue method overloads based on the parameter types. But it's also possible for non-static member functions to differentiate whether they are called on an

lvalue or an rvalue object. These overloads are distinguished by a reference qualifier in the function declaration:

```
struct A
{
    void run() const &;
    void run() &&;
};

A a;
a.run();    // calls run() const &
A().run();  // calls run() &&
```

So when you call `run` on an lvalue, the first overload is executed. When you call it on an rvalue, the second version of `run` is invoked.

It's conceptually similar to lvalue/rvalue overloads we've looked at before:

```
void f(const T&);
void f(T&&);
```

However, in this case the overloads are selected based on the implicit `*this` parameter instead of a regular parameter.

With this feature, you get the ability to optimize operators, for example:

```
struct Counter
{
    Counter operator+(const Counter& other) const &
    {
        Counter c(*this); // take a copy of this
        c += other;
        return c;
    }

    C operator+(const C& other) &&
    {
        (*this) += other;
        return move(*this);
```

```
    }
};
```

The first version of the plus operator creates a new object and returns it. The second version can modify `this` and return it because the object is an rvalue.

An additional benefit is that it allows you to prevent some ill-formed code and make your types behave more like built-in types. Consider this `struct`:

```
struct Curious
{
    int _count = 10;
    Curious& operator ++() { ++_count; return *this; }
    Curious* operator &() { return this; }
};
```

With this definition, you can write interesting code like this:

```
Curious() = Curious();      // assign to rvalue
Curious& c = ++Curious();   // c is a dangling reference
&Curious();                 // address of rvalue
```

All of the above is going to compile despite its questionable nature.

But if you add reference qualifiers to your operators, all of these cases will result in compilation errors as the operators won't be selected by the compiler anymore:

```
struct Curious
{
    int _count = 10;
    Curious& operator ++() & { ++_count; return *this; }
    Curious* operator &() & { return this; }
};
```

Thus, reference qualifiers can help you craft better types.

One rule you have to adhere to is not to mix non-reference-qualified overloads with reference-qualified overloads:

```
struct A
{
    bool run() const { return false; }
    bool run() && { return true; }       // error
};
```

The compiler will complain about this `struct` definition.

Move-only types

With the understanding of move semantics that you've gained, I can now introduce move-only types.

It turns out that move semantics are useful for more than just improving performance. They can also help you express the semantics of ownership. For some types, copying doesn't make sense but they can still benefit from being movable. One example from the standard library is `unique_ptr` which has exclusive ownership of a dynamically allocated object. Other examples are threads, locks, and streams which also have exclusive ownership of their resources.

Creating a move-only type means simply implementing move constructors and/or move assignment operators, but not copy constructors or copy assignment operators. Here is some sample code to illustrate this:

```
class MoveOnly
{
    int* _p;

    MoveOnly() : _p(new int(10))
    {}
    ~MoveOnly()
    {
        delete _p;
    }
    MoveOnly(const MoveOnly& rhs) = delete;
    MoveOnly& operator=(const MoveOnly& rhs) = delete;

    MoveOnly(MoveOnly&& rhs)
```

```
    {
        *this = move(rhs);
    }
    MoveOnly& operator=(MoveOnly&& rhs)
    {
        if (this == &rhs)
            return *this;

        delete _p;
        _p = rhs._p;
        rhs._p = nullptr;

        return *this;
    }
};
```

The main thing to note is that the copy operations are marked as deleted and the move operations are implemented.

You'll be able to use move instances of this class but not copy:

```
MoveOnly a;

MoveOnly b(a);          // copying, doesn't compile

MoveOnly c(move(a));    // OK
MoveOnly d;
d = move(b);            // OK
```

The definition of `b` requires copying so it won't compile. The definition of `c` properly invokes the move constructor by casting `a` to an rvalue reference. Similarly, the assignment to `d` is going to compile as well.

Perfect Forwarding Problem and Solution

As you've seen so far, rvalue references enable move semantics and potential performance gains. But there is another interesting consequence. The introduction of rvalue references along with variadic templates also solved the long-standing problem of transparently forwarding function arguments. This problem was previously unsolvable in C++, at least in the general case.

The forwarding problem arises when forwarding the arguments to a function via a template. To examine the problem and the solution, I'm going to use `unique_ptr` as an example. If you recall, it's a simple smart pointer which has exclusive ownership of a dynamically allocated object, and deletes that object when going out of scope.

If you create a `unique_ptr` instance, you end up writing the type name of the owned object twice:

```
unique_ptr<vector<Point>> p_points(new vector<Point>(10));
```

This isn't very convenient. Let's try to implement a `make_unique` function template to tidy this up. Obviously, it'll need to transparently forward the arguments it receives to the object it constructs. Let's start with a really basic template which can only forward one argument:

```
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg arg)
{
    return unique_ptr<T>(new T(arg));
}
```

The issue with this template is that it makes a copy of `arg` - which doesn't fulfill the goal of transparently passing it through. I can try to make the template take the argument by reference instead:

```
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg& arg)
```

```
{
    return unique_ptr<T>(new T(arg));
}
```

Because of the reference binding rules, I'll run into another problem - it can't take an rvalue argument:

```
make_unique<vector<int>>(10);  // error, can't convert argument
                               // from int to int&
```

In order to handle both lvalues and rvalues it's necessary to provide an additional overload which takes a reference to `const`:

```
int a = 10;
make_unique<vector<int>>(a);   // OK, Arg& overload
make_unique<vector<int>>(10);  // OK, const Arg& overload
```

This works for one argument but doesn't scale, because for multiple arguments, it's necessary to handle all the `const`/non-`const` permutations, and variadic templates can't help with that. Handling 2 arguments requires 4 overloads, 3 arguments requires 8 overloads and so on.

C++11 solves the forwarding problem with the help of rvalue references, and provides a mechanism for *perfect forwarding*:

```
template <typename T, typename T1, typename T2>
unique_ptr<T> make_unique(T1&& arg1, T2&& arg2)
{
    return unique_ptr<T>(
        new T(forward<T1>(arg1), forward<T2>(arg2)));
}
```

This can be easily extended to an arbitrary number of arguments by means of a variadic template:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    return unique_ptr<T>(new T(forward<Args>(args)...));
}
```

So one part of the solution is to take rvalue reference arguments. The other part is the `forward` template, so what is it?

The `forward` template is part of the standard library and it's defined in the `<utility>` header. The implementation of `forward` relies on two new concepts: reference collapsing, and template argument deduction rules for rvalue references. Let's look at how these things work.

Reference collapsing and rvalues in templates

C++03 didn't allow a reference to a reference, so this wouldn't compile:

```
Point p1(10, 10);
using PointRef = Point&;
PointRef& p2 = p1;
```

C++11 changes this behavior and introduces new rules:

Double reference	Collapses into...
A& &	A&
A& &&	A&
A&& &	A&
A&& &&	A&&

Double references collapse into single references, and all the combinations of lvalue and rvalue references have a rule for what they collapse into. Only a double rvalue reference collapses into an rvalue reference, while all the other combinations turn into an lvalue reference.

In addition, rvalue references and templates interact in a particular way. For a template `template<typename T> void f(T&&)`, the following rules apply:

1. If `f` is passed an lvalue of type `A`, then `T` is resolved to `A&`. After that, reference collapsing kicks in, making the final type of the argument `A&`.
2. If `f` is passed an rvalue of type `A`, then `T` is resolved to `A`, and the final type of the argument becomes `A&&`.

In other words, lvalues are passed to an lvalue reference overload, and rvalues are passed to an rvalue reference overload.

How the `forward` template works

As shown above, a perfect forwarding template relies on the `forward` template, which looks like this:

```
template<typename T>
T&& forward(typename remove_reference<T>::type& arg)
{
    // forward arg, given explicitly specified type parameter
    return static_cast<T&&>(arg);
}
```

The `remove_reference` template used for `forward`'s argument type is implemented this way:

```
template<typename T>
struct remove_reference
{ typedef T   type; };

template<typename T>
struct remove_reference<T&>
{ typedef T   type; };

template<typename T>
struct remove_reference<T&&>
{ typedef T   type; };
```

It's specialized for both lvalue and rvalue references, and its member `typedef` is always the type without the reference qualifier.

`forward` does require the use of the `remove_reference` template and can't be simplified to this definition:

```
template <typename T>
T&& forward(T&& arg)
{
    return arg;
}
```

The reason is that with the above definition `forward` could be called without explicitly specifying the template argument, in which case template argument deduction would be used. The result would be that when `forward` was passed an lvalue, `T&&` would turn into an lvalue reference, and we don't want that to happen.

Let's now go back to our `make_unique` template and see what happens when `make_unique` gets called with an lvalue:

```
string model("Boeing 787");
auto sp = make_unique<JetPlane>(model);
```

Following the template instantiation rules, it results in the following instantiations:

```
unique_ptr<JetPlane> make_unique(JetPlane& && arg1)
{
    return unique_ptr<JetPlane>(
        new JetPlane(forward<JetPlane&>(arg1)));
}

JetPlane& && forward(remove_reference<JetPlane&>::type& arg)
{
    return (JetPlane& &&) arg;
}
```

Note that I'm showing uncollapsed references here, so it's C++ pseudocode. Finally, after applying reference collapsing rules this turns into the following:

```
unique_ptr<JetPlane> make_unique(JetPlane& arg1)
{
    return unique_ptr<JetPlane>(
        new JetPlane(forward<JetPlane&>(arg1)));
}

JetPlane& forward(JetPlane& arg)
{
    return (JetPlane&) arg;
}
```

As a result, the *lvalue* argument is passed through `make_unique` and `forward` by *lvalue reference*, which is what we need.

Now let's look what happens when the argument is an rvalue:

```
auto sp = make_unique<JetPlane>("Boeing 787");
```

The instantiations are now going to look like this:

```
unique_ptr<JetPlane> make_unique(JetPlane&& && arg1)
{
    return unique_ptr<JetPlane>(
        new JetPlane(forward<JetPlane>(arg1)));
}

JetPlane&& forward(remove_reference<JetPlane>::type& arg)
{
    return (JetPlane&&) arg;
}
```

After collapsing double rvalue references this code turns into the following:

```
unique_ptr<JetPlane> make_unique(JetPlane&& arg1)
{
    return unique_ptr<JetPlane>(
        new JetPlane(forward<JetPlane>(arg1)));
```

```
}

JetPlane&& forward(JetPlane& arg)
{
    return (JetPlane&&) arg;
}
```

So, an *rvalue* is passed through `make_unique` and `forward` by *rvalue reference*, which again is exactly what's required.

The end result is that a single template can be used for both lvalues and rvalues, and passes the arguments through exactly as they are.

Bonus: implementation of `std::move`

Let's double back and have another look at the `move` function. It turns out that the implementation of `std::move` relies on the same rules as perfect forwarding. Now that you've got an understanding of rvalue references and template deduction rules, it's also easy to understand how `move` is implemented.

This is what `move` looks like:

```
template<typename T>
typename remove_reference<T>::type&& move(T&& arg)
{
    // forward arg as movable
    return
        static_cast<typename remove_reference<T>::type&&>(arg);
}
```

Let's see how this template works with lvalues and rvalues. `move` can be called with an lvalue argument, for example:

```
int a = 10;
move(a);
```

It results in the following template instantiation:

```
remove_reference<int&>::type&& move(int& && arg)
{
    return static_cast<remove_reference<int&>::type&&>(arg);
}
```

It's equivalent to the following after substituting `remove_reference` and collapsing references:

```
int&& move(int& arg)
{
    return (int&&) arg;
}
```

So you can see that in the end it's a simple cast to an rvalue reference.

For an rvalue argument, this instantiation is made:

```
remove_reference<int>::type&& move(int&& arg)
{
    return static_cast<remove_reference<int>::type&&>(arg);
}
```

It's equivalent to the following:

```
int&& move(int&& arg)
{
    return (int&&) arg;
}
```

In other words, `move` returns an rvalue reference when passed either an lvalue or an rvalue.

`constexpr` Mechanism

`constexpr` is a new keyword in C++11. It's an abbreviation of *constant expression*. It can be applied to variables and functions to tell the compiler that they may be evaluated at compile time. This is of course good for performance, as it allows you to move some of the computation from runtime to compile time.

What else is it good for?

Depending on the situation, regular `const` variables can be initialized dynamically, for example if the initializing expression is a function call. By contrast, `constexpr` lets you ensure that a constant is initialized at compile time.

This gives you a couple of advantages. One is that you can use these constants in places where a constant expression is required, such as case labels, `enum` values, or template arguments.

The other is that you are guaranteed that the initialization of these constants will not cause race conditions in a multi-threaded context.

What's in a constant expression?

A basic constant expression is a literal of an integer type, floating point type or an enumerator. In some instances, you can use addresses too. But you can also build more complex expressions by creating your own classes with instantiations usable in constant expressions, and by calling `constexpr` functions. The results can be assigned to `constexpr` variables.

`constexpr` variables

Let's look at `constexpr` variables first. They must be initialized when declared:

```
constexpr auto c_dimensions = 3;
constexpr auto c_threshold = 42.5;
constexpr auto c_name = "constexpr evaluator";
```

They can only be initialized with a literal value, `constexpr` value, or the return value of a `constexpr` function.

`const` and `constexpr`

`constexpr` implies `const` but not the other way around. For this reason, a `constexpr` variable can't always be initialized with a `const`. It can only be initialized with a `const` variable if that variable was itself initialized with a constant expression.

For this reason, the snippet below wouldn't compile:

```
auto a = 10;
const auto b = a;       // a is not a constant expression
constexpr auto d = b;
```

This, on the other hand, compiles fine:

```
const auto c = 10;      // c is a constant expression
constexpr auto e = c;
```

This is because `c` refers to a constant expression.

`constexpr` functions

If a function is marked with `constexpr`, it means that it can be evaluated at compile time as long as all its arguments are constant expressions. If they are not, then it will be evaluated at runtime.

Marking a function with `constexpr` requires that its body consist of a single `return` statement. As a consequence, you'll need to rely on the ternary operator for conditional code, and recursion for any kind of sequence handling.

In addition to the `return` statement, you can only use things like `using` statements and `typedefs`, as well as `static_assert` which I'll talk about in the Compile Time Assertions chapter. These are all evaluated at compile time, which is why they are allowed.

Additionally, if you're marking a member function with `constexpr`, it can't be `virtual`.

Here is an example of a `constexpr` function:

```
constexpr long fibonacci(int n)
{
    return n < 1 ? -1 :
        (n == 1 || n == 2 ?
            1 : fibonacci(n - 1) + fibonacci(n - 2));
};
```

This function returns members of the Fibonacci sequence. As you can see, I've used both ternary operators and recursion to implement it. Since it's `constexpr`, you could use the result of this function in defining an `enum`, for example:

```
enum Fibonacci
{
    Ninth = fibonacci(9),
    Tenth = fibonacci(10)
};
```

But you can also call this function at runtime:

```
auto a = 4, b = 6;
cout << fibonacci(a + b) << endl;  // outputs 55
```

`constexpr` functions can be templated as well:

```
template<typename T>
constexpr auto square(T v) -> decltype(v * v)
{
    return v * v;
}
```

This function returns its argument squared.

`constexpr` functions can't modify their arguments, so passing arguments by reference is off limits. However, they can still take `const` reference arguments.

The return type and the parameters must conform to the requirements for literal types.

Literal types

A literal type includes `void`, scalar types, reference types which refer to literal types, as well as arrays of literal types.

Additionally, it can be a class type if it fulfills the following requirements:

- It has a trivial destructor.
- All of its non-static data members and base classes are also literal types.
- Finally, it needs to be either an aggregate type, or have at least one `constexpr` constructor which isn't a move or copy constructor.

The `constexpr` constructor also has to follow some rules. `constexpr` constructors have very similar constraints to `constexpr` functions, except for the `return` statement of course. Instead, they have an empty body, and they must initialize all base classes and non-static data members. The initialization must be done with constant expressions, both for members and for base class constructors. It implies that all the other constructors involved must also be `constexpr`.

So if you want your classes to be usable in constant expressions, you need to make them literal types. I'll borrow an example from the standard library:

```
class Complex
{
    double _real, _imaginary;
public:
    constexpr Complex(double real, double imaginary) :
        _real(real), _imaginary(imaginary)
    {}

    constexpr double real() const { return _real; }
    constexpr double imaginary() const { return _imaginary; }
};
```

This is a very basic implementation of a class for storing complex numbers. The standard library provides a much more elaborate version - also with `constexpr` constructors and operations.

This class conforms to literal type requirements, and so it can be used to initialize `constexpr` variables:

```
constexpr Complex c1(1, 2);
```

I can also add a `constexpr operator+` which looks like this:

```
constexpr Complex operator+(const Complex& lhs,
    const Complex& rhs)
{
    return Complex(lhs.real() + rhs.real(),
        lhs.imaginary() + rhs.imaginary());
}
```

And with this, I can start performing operations on instances of `Complex` at compile time:

```
constexpr Complex c1(1, 2);
constexpr Complex c2(3, 4);

constexpr Complex c3 = c1 + c2;
```

Note that `c3` is still a constant expression. This concept of compile time computation can be extended further, and if you look online, you can find examples of `constexpr` classes supporting simple operations on arrays and strings.

A couple more notes on `constexpr` functions

There are a couple more things I should mention.

In addition to being implicitly `const`, `constexpr` functions and constructors are implicitly `inline`.

When functions are evaluated at compile time, the compiler doesn't evaluate the conditional branches which aren't taken. These branches may still be taken if the function is evaluated at runtime. It means that you could write code like this:

```
constexpr int percentage(int n)
{
    return (n >= 0 && n <= 100) ? n : throw "out of range";
}

constexpr int c_threshold = percentage(101);
```

If the argument falls outside the bounds at compile time, the compiler will tell you that `throw` isn't a constant expression. If the function is evaluated at runtime, it will throw an exception.

Finally, you might be wondering what `constexpr` does to build times. As `constexpr` can cause a lot of compile time evaluation, for example if there is deep recursion involved, it has the potential to increase build times. This is potentially offset by the compiler memoizing results of compile time function evaluation. Another consideration is that `constexpr` is likely to be replacing template metaprograms which have build overhead of their own, so the overall effect on compile times will depend on your codebase.

Range-based `for` Loop

In addition to the usual `for` loop, you can now use a new form of it to iterate over a whole STL container, or anything which supports the concept of a range defined via `begin` and `end` functions:

```
vector<int> v;
// ... populate the vector ...
for (auto elem : v)
    cout << elem << endl;

for (auto& elem : v)
    elem *= 2;
```

The first loop is simply going to print every element of the `vector`. The second loop multiplies every element by 2. Note that the loop variable is a reference in this case. As you can see, the syntax is very succinct.

`const` and `volatile` qualifiers are also allowed for the iterating variable. Additionally, if you want, you can specify the type of the iterating variable explicitly:

```
for (int elem : v)
    cout << elem << endl;
```

Note that explicit conversions aren't allowed when initializing the loop variable.

There is special handling for built-in arrays under the hood:

```
int arr[] = {10, 20, 30, 40};
for (auto elem : arr)
    cout << elem << endl;
```

Just like with a container, you can iterate over a C-style array using this syntax.

The `initializer_list` template from the standard library (which I discussed in the chapter about uniform initialization) isn't an STL container, but it happens to have `begin` and `end` members, so initializer lists can be iterated over with `for` as well:

```
auto list = {100, 200, 300, 400};
for (auto elem : list)
    cout << elem << endl;
```

When the compiler looks for `begin`/`end`, the member versions of `begin` and `end` take precedence. If they aren't available, then non-member versions which return appropriate iterators are looked up:

```
class MyContainer
{
    list<int> _values;
public:
    // ...
    friend list<int>::iterator begin(MyContainer&);
    friend list<int>::iterator end(MyContainer&);
};

list<int>::iterator begin(MyContainer& cont)
{
    return cont._values.begin();
}

list<int>::iterator end(MyContainer& cont)
{
    return cont._values.end();
}
```

So if I have a container class, and I add non-member `begin` and `end` functions like I did for `MyContainer`, I can then use the range-based `for` loop to iterate over this container:

```
MyContainer cont;
for (const auto& elem : cont)
    cout << elem << endl;
```

Consider the elements of a range-based `for` loop statement:

```
for (elem_decl : seq)
    statement;
```

It consists of `elem_decl` for the loop variable, `seq` for the sequence iterated over, and `statement` which is executed for each element in the sequence. This is equivalent to a traditional `for` loop which looks like this:

```
for (auto iter = seq.begin(), seq_end = seq.end();
    iter != seq_end; ++iter)
{
    elem_decl = *iter;
    statement;
}
```

If member versions of `begin/end` aren't available, then it's slightly different:

```
for (auto iter = begin(seq), seq_end = end(seq);
    iter != seq_end; ++iter)
{
    elem_decl = *iter;
    statement;
}
```

> **Note:**
>
> Note that this implies there will be a copy of each element made if you don't specify that you want a reference in `elem_decl`.

That's all there is to `for` loops.

nullptr

`nullptr` is a new keyword for null pointers which is aimed at replacing the use of `NULL` and 0:

```
int* p = nullptr;
```

`nullptr` is an rvalue which has the type `nullptr_t`. `nullptr_t` is in turn defined in terms of `nullptr`:

```
namespace std
{
   typedef decltype(nullptr) nullptr_t;
}
```

It's specified to behave as a fundamental type.

`NULL` and 0 can still be used the same as before, and interoperate with `nullptr`:

```
int* p = nullptr;
int* p1 = NULL;
int* p2 = 0;
p1 == p;    // true
p2 == p;    // true
```

Comparing `nullptr` with either `NULL` or zero works as expected. `nullptr` is convertible to any pointer type or member pointer type, and also to `bool`, but nothing else. Applying `delete` to the `nullptr` has no effect.

The language and the standard library have been updated to use `nullptr`, so for example a failed `dynamic_cast` returns `nullptr` rather than `NULL`.

By the way, the default value for pointers is `nullptr`, so you can initialize pointers with `nullptr` using uniform initialization syntax:

```
int* p {};
```

However, for readability I prefer explicitly assigning `nullptr`.

What's the advantage over `NULL`?

The advantage of `nullptr` is that it helps avoid ambiguity in situations like this:

```
bool ambiguous(int)
{
    return false;
}

bool ambiguous(int*)
{
    return true;
}

ambiguous(NULL);    // returns false, (int) overload chosen
ambiguous(nullptr); // returns true, (int*) overload chosen
```

I've got two overloads of a function called `ambiguous`. The first one takes an `int`, and the second one takes an `int*`.

Depending on the platform, if you call `ambiguous` with `NULL`, you may get a compilation error due to ambiguity. If the code compiles, passing `NULL` results in a call to the `int` overload. This is because `NULL` is really zero.

`nullptr` resolves the ambiguity in both situations, and causes the compiler to call the second overload for `int*`.

Generally speaking, `nullptr` removes potential confusion between integers and null pointers.

`enum` Changes

C++11 introduced several changes to `enum`s. One of them is strongly typed `enum`s in order to improve type safety, another is forward declaration semantics for `enum`s to reduce coupling and improve compilation times. The third change is the so called scoped `enum`s.

I'll start by talking about the third change - scoped `enum`s.

Scoped `enum`s

If you use `enum class` instead of `enum`, the names in the `enum` will not be exported to the surrounding scope. This is why it's called *scoped*:

```
enum class Proportion
{
    OneHalf,
    OneThird,
    OneQuarter
};

Proportion prop = OneThird;                  // error
Proportion prop2 = Proportion::OneThird;     // OK
```

If you refer to an `enum` value without qualifying it with the `enum` name, the compiler will report an error.

The `enum` members of a scoped `enum` won't be implicitly converted to integer values either:

```
if (prop == 1)  // error
    // ...
```

You need to perform explicit conversion if you want to work with `enum` values.

Specifying the underlying type

Another addition to enums is the ability to specify the underlying type of the enumerants. The type is specified following the enum name and must be one of integral types:

```
enum Direction : unsigned short
{
    South,
    West,
    East,
    North
};

cout << sizeof(North) << endl;  // outputs 2
```

Here, the enumerants of the Direction enum are of type unsigned short. If you try to use a non-integral type, you'll get an error:

```
enum Color : double             // error
{
    Black
};
```

Specifying the type allows you to improve cross-platform compatibility by guaranteeing that the enum uses that type, rather than a type chosen by the compiler.

Scoped enums have an underlying type of int by default, unless you specify a type explicitly. Traditional enums still behave the same way they did before C++11.

Forward declaration

Specifying the underlying type also paves the way for forward declaration of enums. Forward declaration semantics for enums help reduce coupling and can cut down compilation times.

With traditional enums, the compiler needed to know the number of enumerants to select the appropriate underlying type. If the type is specified explicitly, then having a list of enumerants is no longer a requirement. It allows me to write headers like this:

```
// flight_board.h
enum class AirportCode; // forward declared enum

struct FlightBoard
{
    void print_airport_name(AirportCode code)
    {}

    void print_flight(AirportCode code, const string& flight)
    {
        // ...
        print_airport_name(code);
    }
};
```

I've got a forward declaration of the `AirportCode` `enum` preceding the class definition, and I can use it in member function definitions. Both of the functions in the `FlightBoard` class take an `AirportCode` parameter. The definition of the `enum` can be in the corresponding `.cpp` file.

`enum`s which are declared at namespace or class scope can be defined in a surrounding scope:

```
// navigator.h
struct Navigator
{
    Navigator();
private:
    enum CompassPoint : int;      // forward declaration
    CompassPoint _compass_point;
};
```

```
// navigator.cpp
enum Navigator::CompassPoint : int
{
    North,
    South,
    East,
```

```
    West
};

Navigator::Navigator() : _compass_point(North)
{}
```

In the above code sample, I have a forward declaration for `CompassPoint` in the `Navigator` `struct` declaration, which is in a header file. The definition of the `enum` is in the companion `.cpp` file. This can help with data hiding, e.g. if you're only distributing the headers of a library.

Forward declarations have to follow three rules:

1. The forward declaration has to tell the compiler the underlying type, either explicitly or implicitly.
2. The underlying type has to match between declarations and the definition.
3. You can't change from scoped to unscoped `enum` or vice versa.

This is best illustrated with examples from the forward declaration proposal for C++11. Let's go over these declarations line by line:

```
enum E : short;       // OK
enum F;               // error, underlying type is required
enum class G : short; // OK
enum class H;         // OK, underlying type for scoped enums
                      // is int by default
```

The first declaration is fine as it specifies the type. The second declaration without a type is problematic. The third declaration is fine because it also has a type. The fourth declaration relies on the default type for scoped `enums`, which is `int`.

The next 4 declarations are all fine because they re-declare `enums` without changing the underlying type:

```
enum E : short;       // OK, redeclaration
enum class G : short; // OK, redeclaration
enum class H;         // OK, redeclaration
enum class H : int;   // OK, redeclaration with the same
                      // underlying type
```

These two declarations attempt to change the `enum` from unscoped to scoped, and vice versa. This kind of change isn't allowed, so both of these are errors:

```
enum class E : short; // error, can't change unscoped to scoped
enum G : short;       // error, can't change scoped to unscoped
```

Let's look at the next group:

```
enum E : int;         // error, different underlying type
enum class G;         // error, different underlying type
enum class H : short; // error, different underlying type
```

These declarations attempt to modify the underlying type, either explicitly or by omitting type and thus relying on the default type of `int`. All of these are errors as well.

Finally, this redeclaration is actually a definition, so it's fine:

```
enum class H {};      // OK, this redeclaration is a definition
```

Et voilà, now you can start using the new `enums` in your code.

Compile Time Assertions

C++11 introduces the concept of compile time assertions. Unlike the familiar runtime assertions typically done to check pre- or post-conditions for a function, these assertions are statically checked at compile time. The advantage of this approach is that some errors can be detected very early, before you even run the software, and the checking doesn't incur a runtime overhead.

Many compile time checks are implemented with the help of template metaprogramming and work on expressions which can be evaluated at compile time. They can be used, for example, to narrow down the definitions of acceptable types in your templates, which can improve the type safety of your code. You can use them to validate non-type template parameters. There are other possible uses outside of templates, such as ensuring sufficient sizes of integral types.

Compile time assertions are performed with `static_assert`. For example, in cross-platform code it might be useful to enforce assumptions about built-in types:

```
int int_magic(int a, int b)
{
    static_assert(sizeof(int) <= 4,
        "int must be no more than 4 bytes");
    // ... do things with a and b ...
}
```

Note that `static_assert` allows you to provide your own message. It will be printed by the compiler if the assertion fails.

Another example is checking preconditions on template parameters:

```
template<unsigned int dimensions>
struct Matrix
{
    Matrix()
    {
        static_assert(dimensions <= 3,
```

```
            "dimensions must not exceed 3");
    }
};

Matrix<4> m;    // generates a compile error
```

As you can see, it allows me to easily enforce the valid range for a non-type template parameter, which in this case is the number of dimensions in the matrix.

You can also use it to enforce the behavior of type template parameters:

```
struct Base
{
    virtual ~Base() {}
};

template<typename T>
class Derived : public T
{
    static_assert(has_virtual_destructor<T>::value,
        "the base class must have a virtual destructor");
};

Derived<Base> d;        // OK
Derived<string> s;      // triggers static_assert
```

Here, I've got a `static_assert` in the constructor of the derived class. It allows me to ensure that the base class has a virtual destructor. The `has_virtual_destructor` template I used in the static assert is one of the many templates for querying and manipulating type properties in the standard library called `type_traits`.

A `static_assert` can be placed anywhere where a declaration is allowed, not just inside a function body - in this example it's in the class definition.

Literals

Some changes have been made to literals. C++11 acquired a number of literals to represent Unicode literals in different encodings. It also has raw literals which don't allow special characters. The third change is that it allows you to define your own types of literals with custom suffixes.

Unicode support and literals

C++11 finally added some support for Unicode into the language and the libraries. In order to support Unicode strings, there are three new string literal prefixes:

```
u8"UTF-8: \u00BD"
u"UTF-16: \uA654"
U"UTF-32: \U0002387F"
```

Unlike wide string literals which are arrays of type `wchar_t` and have an implementation dependent size, the standard defines types of specific size for these new literals.

Literals prefixed with `u8` are UTF-8 encoded strings. Lowercase `u` prefix is for UTF-16 encoded strings, and capital `U` is for UTF-32 encoded strings.

Different literals are stored in arrays of different character types:

Prefix	Character type	String type
`u8`	`char`	`string`
`u`	`char16_t`	`u16string`
`U`	`char32_t`	`u32string`

There is also a `string` class `typedef` in the standard library for each of these character types. A useful thing about character types is they have defined sizes which are not implementation dependent.

You can specify Unicode symbols by their code in string literals by prefixing the code with backslash lowercase u or backslash uppercase u:

```
string s(u8"\u00BD \u00B5s");
```

The string s will contain $^1/_2\mu$s.

The mapping between these codes and symbols is defined by a standard called ISO/IEC 10646. The form with the lowercase u prefix is a shorthand for the uppercase U prefix, so u'\uA654' is equivalent to U'\U0000A654'.

Unicode character literals

In addition to Unicode string literals, there are two Unicode character literals:

Prefix	Character type	Example literal
u	char16_t	u'\u160E'
U	char32_t	U'\U0000160E'

They start with a lowercase or uppercase u and contain a Unicode code point. Note that you can't use the u8 prefix with a character literal.

Please keep in mind that while there is now language support for Unicode, the standard library support is quite limited. iostreams are able to handle Unicode with appropriately set up locales, but the string class, for example, treats its contents simply as a sequence of values of a particular type and isn't aware of Unicode code points and so on.

Raw literals

Raw literals have a simple purpose. They can't contain any special characters, allowing you to have things like backslashes, quotes or what would normally be escape sequences like \n in your string literals:

```
cout << R"(use "\n" to represent newline)" << endl;
```

The above is going to output the string literal verbatim, without any translation. You'll be able to see the quotes and `\n` in the output.

Raw literals start with a capital `R` prefix. They are delimited with parentheses enclosed in double quotes.

The `R` prefix can be used with all types of string literals:

```
R"(No newline \n)"
LR"(No newline \n)"
u8R"(No newline \n)"
uR"(No newline \n)"
UR"(No newline \n)"
```

Note that if you use a character type prefix like `L` or `u8`, the `R` has to come after it.

Raw literals are convenient when you define regular expressions for example:

```
R"("\w+\\w+")"
```

This regular expression would match strings like `"ab\cd"` (including the quotes). Only regular expression escaping is necessary here. A *regular* literal representing the same regular expression would look like this due to extra escaping required for C++ string literals:

```
"\"\\w+\\\\w+\""
```

This is much messier and becomes very hard to understand quickly.

Raw literals can be handy for command strings too:

```
R"(grep -r "\.js" *)"
```

Raw string literals still contain one special character sequence, which is the closing parenthesis followed by a quote. This is a problem if your string happens to contain this sequence of characters too. To get around this, you can customize the delimiter:

```
cout << R"!!(A raw literal is delimited with "( )")!!" << endl;
```

I've added a couple of exclamation marks between the quote and the parenthesis, but I could have also used stars or letters, for example. Whitespace isn't allowed, however. The custom delimiter characters at the start and at the end of the literal have to match. The delimiter sequence can be up to 16 characters long.

Lastly, raw string literals are allowed to contain newlines:

```
R"(multiline
string)"
```

The above raw literal is equivalent to the regular literal `"multiline\nstring"`, so an actual newline in the source file gets translated into a newline character in the string.

User defined literals

Another interesting feature is user defined literals. These are marked with suffixes instead of prefixes, by analogy with suffixes for number literals:

```
1.2_i;  // express complex numbers
10_km;  // express units
```

`_i` and `_km` are not part of the language, they are something you can define yourself. This allows you to attach things like units or type information to literals. User defined literals can also be used to make your code more readable:

```
widget.set_height(150_px);
widget.set_width(80_percent);
```

The code can be more self-documenting when written in this manner, but like with many other features, it's probably best not to take this approach too far.

This kind of literals is made possible by literal operators:

```
// create a 'complex' instance from an imaginary literal
constexpr complex<double> operator "" _i(long double d)
{
    return {0, d};
}
```

When the compiler encounters a literal with a non-standard suffix, it looks for a corresponding operator to pass the literal to.

The operator name is two double quotes followed by the literal suffix. There has to be a space between quotes and the suffix, but there can't be any spaces between the quotes.

The operator you see here transforms the literal suffixed with `_i` into an instance of the standard `complex` class. It sets the real part to zero, and imaginary part to the argument.

User defined literal identifiers have to start with an underscore. All the names which don't start with an underscore are reserved for future standardization. The reason for this is that this mechanism will be used to add standard literals in the future.

Please note that built-in literals cannot be redefined.

Types of literals

A user defined suffix can be added to 4 kinds of literals:

- integer literals
- floating point literals
- character literals
- string literals.

For each of these, the compiler will look for particular operator signatures.

By the way, literal operators and literal operator templates behave the same as any regular functions, except they don't have C linkage. They are looked up like any other functions, and they can be called explicitly.

For character and string literals, there is one possible operator signature for each character type:

```
operator "" _suffix(char)
operator "" _suffix(wchar_t)
operator "" _suffix(char16_t)
operator "" _suffix(char32_t)

operator "" _suffix(const char*)
operator "" _suffix(const wchar_t*)
operator "" _suffix(const char16_t*)
operator "" _suffix(const char32_t*)
```

For integer and floating point literals, there are 4 possible operator signatures, with different intents:

```
operator "" _suffix(unsigned long long)
operator "" _suffix(long double)

operator "" _suffix(const char*)

template<char... Digits>
operator "" _suffix()
```

The first two versions of the operator receive the *value* of the literal which is parsed by the compiler. There are two different signatures, one for integer literals, and another one for floating point literals.

The other two versions receive the *characters* comprising the literal, either in the form of a `char` array, or as a template parameter pack. The literal could be either an integer or a floating point number.

Let's look at some sample code for integer literals to see how the different versions of the operator can be used in practice.

Literal operators for integers

The operator with the first version of the signature could look like this:

```
// create a distance
constexpr Distance operator "" _au(unsigned long long int n)
{
    return {n, Unit::astronomical_unit};
}

cout << "Neptune orbit radius: "
    << (30_au).to_light_years() << endl;
```

The compiler parses the literal and passes the numeric value to the operator. I had to wrap the literal along with its suffix in parentheses, as otherwise the compiler is unable to parse a literal directly followed by a method call. This appears to be a limitation of the standard rather than GCC.

Note that besides a possibly more natural flow for reading the code, this kind of operator offers little advantage over a regular conversion function.

The other two possible signatures allow you to do something more interesting.

In some cases, you might want access to the characters comprising the literal instead of its value. This allows you to implement things like binary literals (as an example). The implementation would look something like this:

```
unsigned long long operator"" _b(const char* digits)
{
    if (strlen(digits) >
        numeric_limits<unsigned long long>::digits)
        throw runtime_error("Too many digits in binary literal");

    unsigned long long res = 0;
    auto digit = digits;
    while (*digit != '\0')
    {
        if (*digit != '0' && *digit != '1')
            throw runtime_error("Only 1 and 0 allowed in binary "
                "literals");

        res = (*digit - '0') + (res << 1);
        ++digit;
```

```
    }
    return res;
}
```

This kind of literal operator is called a *raw* literal operator. The implementation is pretty straightforward. First it checks that the number of digits in the literal isn't more than can be represented by the return type. Then it loops over the argument and adds up the number bit by bit. It also makes sure that the digits are only ones or zeroes, and throws exceptions in case of errors.

With this operator available, you can write things like this:

```
101_b;    // equals 5
-1011_b;  // equals -11
```

Note that I don't have to handle minus in my literal operator as minus is a unary operator applied to the integer value returned by my operator.

If I write an invalid binary literal, it will compile, but I will get an exception at runtime:

```
123_b;    // throws an exception
```

It's also possible to write a `constexpr` version of this operator to get compile time enforcement of these literals. Alternatively, you could use the *third* form of the numeric literal operator to process literals at compile time.

The third version of the literal operator allows you to use template metaprogramming to handle the characters in the literal. The handling of binary literals could look like this:

```
template<char... Digits>
constexpr unsigned long long operator "" _b()
{
    return Binary<Digits...>::value;
}
```

This relies on a helper variadic template called `Binary`, which is going to recursively process the digits. It looks like this:

```
template<char... Digits>
struct Binary;

template<char digit, char... Digits>
struct Binary<digit, Digits...>
{
    static_assert(
        sizeof...(Digits) + 1 <=
        numeric_limits<unsigned long long>::digits,
        "Too many digits in binary literal");

    static_assert(digit == '1' || digit == '0',
        "Only 1 and 0 allowed in binary literals");

    static constexpr unsigned long long value =
        ((digit - '0') << sizeof...(Digits)) +
        Binary<Digits...>::value;
};

template<>
struct Binary<>
{
    static constexpr unsigned long long value = 0;
};
```

This is quite straightforward as long as you've already read the chapter on variadic templates. The main template which splits off the head of the parameter pack has a couple of `static_asserts` to make sure there aren't too many digits, and that all the digits are either 0 or 1.

It then calculates the value represented by the literal. To do that, it puts the first bit into its correct place with the shift-left operator, and adds in the other bits by recursively instantiating itself with the rest of the parameter pack.

There's also a specialization of the `Binary` template with no parameters. This is necessary to stop the recursion when there are no more digits left to process.

As all of this happens at compile time, the binary literals can be used in constant expressions:

```
constexpr auto value = 101_b;
```

Character and string literal operators

Let's now give some attention to character and string literals.

For character and string literals, there is a literal operator signature corresponding to each type of the literal.

To give an example of an operator for string literals, it could do something like this:

```
u16string operator "" _reverse(const char16_t* str, size_t len)
{
    u16string s(str);
    reverse(s.begin(), s.end());
    return s;
}

u"palindrome"_reverse;  // yields "emordnilap"
```

This operator handles 16-bit character literals, and it reverses the string that's given to it. The `size_t` argument conveniently allows the compiler to distinguish a string literal operator from a number literal operator which receives the digits as a string.

Raw string literals are passed to the operator as is:

```
uR"(two\nlines)"_reverse;   // yields "seniln\owt"
```

There is no special handling required for raw literals.

If string literals are concatenated, you only need to use the user defined literal suffix on one of them:

```
u"The brown fox "
"jumps over a lazy dog "
"and back again"_reverse
```

The above is the same as

```
u"The brown fox jumps over a lazy dog and back again"_reverse
```

This covers all of the functionality related to user defined literals.

noexcept

The introduction of the noexcept specifier is a byproduct of adding move semantics to the language. When move semantics were added, it turned out that some classes which were fine in C++03 would now automatically acquire a move constructor which can throw, and as a result some operations on STL containers which provided the strong guarantee would no longer provide it.

noexcept is a specifier used in function declarations or definitions. It tells the compiler that a function should not throw exceptions.

In addition to the introduction of noexcept, C++11 also deprecates dynamic exception specifications which were added using the throw specifier. This mechanism didn't work too well, and noexcept is intended to be a simpler and more practical replacement.

So, all the compiler generated operations such as default constructors, destructors and so on are noexcept if every function they directly invoke is also noexcept. Among other things, it allows STL containers to choose move operations over copy and therefore have better performance.

Also, the delete operators and user defined destructors are noexcept by default, unless explicitly declared otherwise.

noexcept for your own functions

You can also add noexcept to any of the functions you write:

```
constexpr long fibonacci(int n) noexcept
{
    return n < 1 ? -1 :
        (n == 1 || n == 2 ?
            1 : fibonacci(n - 1) + fibonacci(n - 2));
};
```

Keep in mind that this isn't a compile time check. Rather, you're telling the compiler that this function *shouldn't* throw any exceptions.

If it does throw, then it will result in an immediate call to `std::terminate`. Stack unwinding doesn't have to happen in this situation. Unlike the dynamic exception specifications, `std::unexpected` isn't called either.

This is an advantage of `noexcept` because it means that it can be implemented without the runtime overhead that the `throw` specifier introduced.

The implication is that functions declared `noexcept` have to catch all exceptions internally.

In addition to simply adding `noexcept` to a declaration, you can use `noexcept` with a constant expression which evaluates to `true` or `false`:

```
template<typename T>
auto square(const T& v) noexcept(is_fundamental<T>::value)
    -> decltype(v * v)
{
    return v * v;
}
```

Here I'm using a template called `is_fundamental` from the standard `type_traits` library to establish whether my function can throw exceptions or not. If `T` is a fundamental type, then it won't throw anything. `noexcept` without a constant expression is equivalent to `noexcept(true)`.

You can also use this functionality to override the default exception specification of a destructor:

```
~A() noexcept(false);
```

Of course, the general rule is not to throw any exceptions from the destructor, so you probably won't use this often.

`noexcept` operator

A useful constant expression to use with the `noexcept` specifier is a call to the `noexcept` operator. I could rewrite the `noexcept` condition for the `square` function from the preceding example in this fashion:

```
template<typename T>
auto square(const T& v) noexcept(noexcept(v * v))
    -> decltype(v * v)
{
    return v * v;
}
```

The double `noexcept` is not a mistake. Aesthetically, it isn't great but I guess the committee deemed minimizing the number of keywords more important than how the code looks.

The fact that I had to repeat the same expression three times to define this function isn't great either, but we've got to work with what we have.

So anyway, the first `noexcept` is the exception specifier, and the second `noexcept` is the operator keyword.

The operator analyzes the expression and returns `true` if it shouldn't throw exceptions, and `false` otherwise. So, if the multiplication operation doesn't throw exceptions, then this function is also `noexcept(true)`. Otherwise, it's `noexcept(false)`.

The `noexcept` operator is similar to `decltype` in that its operand is an expression, and the expression isn't evaluated.

Note that the analysis it performs isn't exhaustive. The compiler looks at all the operations in the expression, and if all of their exception specifications are `noexcept(true)`, then it yields `true`. It doesn't analyze the actual definitions of the operations though.

When to use `noexcept`

Here are some rough guidelines on when to use `noexcept`.

First of all, it isn't likely to be a feature that you use a lot. The default should be to allow functions to throw. It's difficult to make sure that a non-trivial function never throws.

So if you use it, preferably apply it to small functions only, where you can easily convince yourself that no exceptions can be thrown.

You should probably avoid using `noexcept` if your function has preconditions, because preconditions are likely to change over time. This has the potential to introduce exception throwing behavior.

Declaring move constructors and move assignment operators, as well as the `swap` method `noexcept` is a good idea, so do it whenever you can. As I mentioned, it allows better performance as STL containers can choose to use move operations. Some of the STL internals use `move_if_noexcept` instead of simple `move` calls in order to provide the strong exception guarantee.

A couple more notes on `noexcept`

A pointer to a `noexcept` function can be declared `noexcept`. For example:

```
long fibonacci(int) noexcept;

long (*p_fib)(int) = fibonacci;  // OK, but noexcept is lost

long (*p_fib2)(int) noexcept = fibonacci;  // instead, noexcept
                                           // can be preserved
```

Note that you shouldn't be able to add a `noexcept` specifier to a pointer if the function itself isn't `noexcept`:

```
double square(double);
double (*p_square)(double) noexcept = square;  // error
```

However, GCC doesn't currently enforce this.

Additionally, `noexcept` can't appear in type aliases:

```
using Func = int(double) noexcept;  // error
```

But once again, this is ignored by GCC.

That's all I've got to tell you about `noexcept`.

Explicit Conversion Operators

In addition to `explicit` constructors, you can now declare conversion operators `explicit` too. It allows you to prevent unwanted implicit conversion. For example, the standard template `std::function` has an explicit `operator bool`:

```
explicit operator bool() const noexcept;
```

This operator returns `true` if the function object has a target. So things like this will not compile without an explicit cast:

```
function<void(int, int)> f1, f2;

auto sum = f1 + f2;         // error
bool flag = f;              // error
```

The attempts to define both `sum` and `flag` result in compilation errors because implicit conversion to `bool` is disallowed by the operator.

However, you can check whether a `function` instance has a target before calling the function:

```
if (f)
    f(a, b);
```

This is because `operator bool` gets special treatment from the compiler, and its `explicit` specifier is ignored where it's considered safe to do so. Examples of safe contexts are conditional statements and the ternary operator.

The `explicit` specifier will be in force when checking for equality or inequality though.

Inline Namespaces

The `inline` keyword acquired a new meaning which can be applied to namespaces. When a namespace is marked `inline`, its contents become available in the enclosing namespace.

The primary purpose of inline namespaces is to provide a way to implement versioning for libraries. Consider this example:

```
namespace API
{
    inline namespace v2
    {
        // v2 allows processing doubles instead of ints
        void process(vector<double>);
    }

    namespace v1
    {
        void process(vector<int>);
    }
}
```

The idea is that a library provides some API which may change from version to version. So each version of the functionality is wrapped in a corresponding namespace. The latest namespace is additionally declared `inline`. This allows me to use the library in this manner:

```
vector<double> doubles;
API::process(doubles);

vector<int> ints;
API::v1::process(ints);
```

As `v2` namespace is inline, it means its contents are available in the outer namespace `API`, so I can call `process` on `doubles` without mentioning `v2`.

However, if I want to call a function from version 1 of the API, I need to qualify it with the `v1` namespace.

What happens if you need to create a new version of the API? When version 3 of the API is created, the latest declarations are placed into a namespace called `v3` which becomes the inline namespace, while `v2` loses its `inline` qualifier. Clients using `v3` will be able to use functions as if they were in the API namespace, whereas those who want to use `v2` or `v1` will have to qualify their calls with the version.

Why not just add a `using` statement?

You might be thinking that you could achieve the same thing by adding a `using` directive to the `API` namespace:

```
namespace API
{
    using namespace v2;
    namespace v2
    {
        // v2 allows processing doubles instead of ints
        void process(vector<double>);
    }
    namespace v1
    {
        void process(vector<int>);
    }
}
```

This *mostly* works, but there are some issues. There is a problem if you want the users of your library to be able to provide template specializations for templates in your API. These specializations need to be in the namespace where the template is defined. Your users would logically expect to provide specializations in the `API` namespace, because that's what they normally work with. They may not even know about the `v*` namespaces if they only read the documentation but don't look at the library headers.

For example, imagine that the library requires users to provide hash template specializations for their own types which they want to use as hash keys. They might try something like this:

```
namespace API
{
    template<>
    class Hash<Part>
    {
        size_t operator()(const Part& p) const
        { /* ... */ }
    };
}
```

But specializing in the `API` namespace isn't going to work because the template is actually defined in the nested `v2` namespace rather than the `API` namespace.

There are additional issues related to name lookup. Inline namespaces take care of these problems, which is why they are a better alternative.

Alignment

C++11 introduced two new keywords related to alignment - `alignas` and `alignof`.

`alignof`

`alignof` keyword returns the alignment requirements of its argument, which is a type:

```
alignof(double);  // yields 8
```

The type used with `alignof` must be complete.

If it's applied to a reference type, then it's the same as applying it to the type which is referred to. If it's applied to an array type, it returns the alignment requirement of the element type.

`alignas`

`alignas` specifies how a type or an object should be aligned.

It can be applied to variables, data members, class declarations and enumeration types:

```
alignas(32) int arr[10];

struct S
{
    alignas(32) Buffer _buf;
};

struct alignas(2 * alignof(double)) Doubled
{}
```

For classes, `alignas` has to come after the `class` or `struct` keyword. The requested alignment has to be a power of two.

In addition to passing an integral constant expression to `alignas`, you can pass a type. In that case, the `alignas` expression is equivalent to `alignas(alignof(T))`:

```
alignas(double) unsigned char double_buf[256];
// alignas(double) is the same as alignas(alignof(double))
```

This allows you to match alignment requirements of another type.

It's also possible to apply more than one `alignas` specifier:

```
alignas(T) alignas(A) T buffer[N];
```

In this case, the strictest alignment requirement (that is, the requirement with the highest numeric value) will be used. You can use this, for example, to ensure that alignment requirement for an array isn't weaker than the requirement for the element type. If it was, the program would be ill-formed.

An alignment requirement which would weaken the implicit requirement for a type is ignored.

`sizeof` Applied to Non-static Data Members

The next change I'd like to mention is very simple, so this is the shortest chapter in the book. `sizeof` can now be applied to non-static data members without providing an object:

```
class A
{
public:
    int _a;
};

sizeof(A::_a);      // yields sizeof(int)
```

This is a bit tidier than the workarounds previously employed for this, such as casting the null pointer to the type and referencing the member via that.

Memory Model

Let's talk about the C++ memory model now. I'm only going to provide a short overview of it, as it isn't visible in the language syntax. Also, a more in-depth discussion makes more sense in conjunction with talking about the standard library facilities for atomic types, which is outside the scope of this book.

The main thing to know is that the C++11 standard has a different definition of the abstract machine which now allows the possibility of multi-threaded execution, unlike the previous versions of the standard. The result is that it's now possible to write multi-threaded code in a non-platform specific way.

As part of this, the memory model in the standard defines what the compiler is allowed to do when it comes to memory access. For instance, it guarantees that threads can read and write separate memory locations without interfering with each other.

On the library side, there are provisions for the ordering of memory operations and control over it. The standard also has a few things to say about data races in the context of the standard library functionality.

Multi-threading related semantics

There are a couple of things worth mentioning with regards to semantics.

The initialization of objects of static storage duration is now guaranteed to be thread-safe, however access still needs to be synchronized.

In C++11, the standard library requires `const` objects to be thread-safe, that is, all the operations on `const` objects have to be either read only or otherwise internally synchronized. This isn't enforced by the compiler, however.

As a corollary, this also means that if you have a `mutable` member in your class, and you want to use `const` objects of this class with any standard library functionality, then the `mutable` member must be internally synchronized. Otherwise your `const` object may not provide the required thread safety when it modifies the `mutable` member, and there is potential for race conditions.

Thread local storage

Now that concurrency is part of the picture, C++11 introduces the concept of thread-local variables. Variables with thread local storage exist for the duration of the thread in which they are created, and each thread gets its own copy of the variable. To use thread local storage, you need to mark a variable with the `thread_local` storage specifier.

GCC also has a non-standard `__thread` keyword intended to designate thread local storage, but `thread_local` is different because it allows dynamic initialization and destruction semantics. This requires a runtime overhead for references to `thread_local` variables which are not function-local and defined in a different translation unit. The overhead is present regardless of whether the variables use dynamic initialization.

The `thread_local` attribute cannot be applied to functions. It can only be applied to data declarations with static storage duration. This encompasses namespace scope variables (including those in the global namespace), static function variables, and static class members. Here is some sample code:

```
thread_local B b1;

namespace
{
    thread_local B b2;
}

class C
{
    // each thread gets a separate instance counter
    static thread_local int _instance_counter;
};

void f()
{
    // each thread invoking f() has a copy of run_count
    static thread_local int run_count;
}
```

b1 is a global variable, and b2 is a namespace scope variable.

Class C contains a static member called instance_counter which is marked with thread_local. As a result, each thread using this class will get a separate instance counter. For block scope variables, static is implied when using thread_local, however you might prefer to specify it explicitly for clarity.

thread_local can also be combined with extern.

In a similar fashion, the function f contains a static variable called run_count. Each of the threads which call this functions will have its own copy of run_count.

All the declarations and the definition of a thread local object must have the thread_local specifier. For example, the following will produce an error:

```
extern int i;
int thread_local i;           // error
```

thread_local must be specified in all declarations:

```
extern int thread_local i;
int thread_local i;           // OK
```

Thread local initialization

thread_local objects can be of types with arbitrarily complex constructors and destructors.

By default, thread_local variables are zero initialized. They can be initialized dynamically.

thread_local variables at namespace scope and thread_local static class members are constructed before first use, but the standard doesn't specify exactly when. Depending on the compiler, they might be initialized when the thread is started, or just before the first use in a thread, or somewhere in between.

thread_local variables declared in a function are initialized similarly to local static variables. Initialization happens when the execution goes through the declaration on the given thread.

If an exception is thrown while constructing a thread_local variable, it results in a call to std::terminate.

Thread local destruction

As usual, the destructors are called in the reverse order of construction, but the order of initialization is unspecified, so you can't rely on destructors being called in a particular order.

If a thread calls `std::exit` or exits from `main`, its thread local variables are destroyed. However, you need to keep in mind that they're only destroyed on that thread, and if any other threads are still running at the time, *their* thread local variables will not be destroyed.

Attributes

The next feature I'd like to show you is generalized attributes. Attributes provide a standard mechanism for annotating parts of the program such as functions with vendor specific or optional information. They are meant to unify and standardize compiler specific extension syntax like GCC's `__attribute` or Microsoft's `__declspec`.

In the standard syntax, attributes are denoted with double square brackets:

```
[[attribute_name]]
```

Note that you can't use two opening square brackets in any other context. For example, this code combining a lambda expression with an `operator[]` call is illegal:

```
vector<int> v;
// ...
int a = v[[] { return 10; }()];
```

Even if you add a space between the brackets, it won't compile:

```
vector<int> v;
// ...
int a = v[ [] { return 10; }()];
```

Instead, you'll need to take the lambda out:

```
auto lambda = [] { return 2; };
int c = v[lambda()];              // OK
```

Without the double square brackets, there is no problem.

Standard attributes

There are only two standard attributes defined in C++11. The first one is `noreturn` which is used to indicate that control flow will not return to the caller:

```
[[noreturn]]
void throw_error(const string& error) noexcept(false)
{
    throw runtime_error(error);
}
```

A function that loops indefinitely, or always throws an exception, or exits the program could be marked with `noreturn`. This gives the compiler an opportunity to give warnings about unreachable code, or to perform optimizations.

If a function is marked with `noreturn` but returns nonetheless, it's undefined behavior. The compiler will give a warning about it.

The second standard attribute is called `carries_dependency`:

```
void f(int* p_handle [[carries_dependency]]);
```

It can be applied either to functions or to function parameters, and allows the compiler to perform optimizations when using a relaxed memory model. Currently, GCC ignores this attribute (which is compliant with the standard).

You can also use GCC-specific attributes with this syntax. Specifying an attribute is done like this:

```
struct [[gnu::aligned (32)]] S {};
```

The above is equivalent to this GCC-specific definition:

```
struct __attribute((aligned (32))) S {};
```

The standard provides a namespacing mechanism for attributes, and recommends that vendors choose a distinctive namespace name. Attribute names can be prefixed with a namespace name followed by two colons. The names of GCC specific attributes have to be prefixed with `gnu`.

POD Types, Trivial Types, and Standard Layout Types

The definition of Plain Old Data classes and unions has changed. In C++11, the definition goes like this:

1. A POD class is a class that is either a POD `struct` or a POD `union`.
2. A POD `struct` is a non-`union` class that is both a trivial class and a standard-layout class, and all of its non-static data members are POD `structs`, POD `unions`, or arrays of such types.
3. Similarly, a POD `union` is a `union` that is both a trivial class and a standard layout class, and has no non-static data members of type non-POD `struct`, non-POD `union`, or array of such types.

Let's look into what makes a class trivial or standard layout.

Trivial classes

A trivial class has to fulfill the following requirements: it is either a scalar type, or a trivially copyable class with a trivial default constructor, or an array of such type/class. It can optionally be `const` or `volatile` qualified.

It's useful to distinguish such types because objects of trivial types may be created via `malloc` and copied with `memcpy`, which can be an opportunity for optimization.

Trivially copyable classes

A part of the trivial class definition is the notion of a trivially copyable class. A trivially copyable class is defined by the following conditions:

- no virtual functions or virtual base classes

- trivial copy and move constructors
- trivial copy and move assignment operators
- a trivial destructor.

The big change to note here is that POD types in C++11 are allowed to have user defined constructors, as long as they provide a trivial default constructor.

Trivial operations

Next we need to dig into what makes an operation trivial. A copy or move constructor is trivial if, first of all, it is not user-provided, and as long as

- the class doesn't have virtual functions or virtual base classes
- all the base class copy and move constructors are also trivial, and
- copy and move constructors are trivial for all the non-static data members of the class which are of class types or arrays of class types.

Standard layout types and classes

The second part of the new POD definition is that the class conforms to standard layout class requirements, which are part of the definition of a standard layout type.

A standard layout type is either a scalar type or a standard layout class. It can also be an array of such type or class, and possibly `const` or `volatile` qualified.

A standard layout class has to fulfill these requirements:

- all non-static data members are standard layout or arrays of standard layout classes
- no virtual functions or virtual base classes
- the same access control for all non-static data members (this rule is relaxed - before C++11 all members had to be public)
- all base classes are standard layout too
- at most one class in the inheritance hierarchy has non-static data members (this can be the most derived class or one of the base classes)
- no base classes of the same type as the first non-static data member.

The idea is that such a type will have a memory layout which is the same as the layout of a C `struct`.

These requirements are quite specific and can be hard to remember, so it's best to look them up when you need to create one of these classes.

The reason that POD types are defined in terms of trivial and standard layout types is that a class can be trivial but not standard-layout, and vice versa, and sometimes it's useful to be able to make the distinction. Consider an example:

```
struct Trivial          // trivial but not standard-layout
{
    int _a;
private:
    int _b;
};

struct StandardLayout   // standard-layout but not trivial
{
    int _a;
    int _b;
    ~StandardLayout();
};
```

The `Trivial` class isn't standard layout because it has both a private and a public member. Both members would have to be either public or private for it to be standard layout.

The `StandardLayout` class isn't trivial because it doesn't have a trivial destructor.

Changed restrictions on `unions`

The restrictions on `unions` have changed in C++11 to make more member types feasible. Members of types with user defined constructors, destructors and assignment operations are now allowed.

As before, member types aren't allowed to have virtual functions, or to be reference types.

At the same time, restrictions have been added to encourage *discriminated unions* which are less prone to causing errors. There is a new restriction which says that if a non-static

data member of a union has a non-trivial default constructor, destructor, or a copy or move operation, then the same function of the union itself is implicitly deleted.

You have to write your own function to override the implicit behavior. Let's look at some sample code to see what this rule means.

So, you can now write a union like this:

```
union Compact
{
    int _i;
    string _s;
};
```

But you won't be able to do much with it:

```
Compact c1;    // error
```

This is because the string class overrides all the compiler-generated operations with non-trivial versions, and they are disabled in the union as a result. You need to define your own constructor and a destructor instead:

```
union Compact
{
    int _i;
    string _s;

    Compact() : _i(100)
    {}
    ~Compact()
    {}
};
```

Note that I chose to initialize the int member in the constructor. This kind of works, you can now create an instance:

```
Compact c1;
cout << c1._i << endl;   // outputs 100
```

_i is set to 100 by default, but you can make either member of this union active, which requires the use of placement new to initialize the string member and an explicit destructor call to clean it up later:

```
new (&c1._s) string("ABC");     //switch the active member to _s

// ... some time later ...
c1._s.~string();    // destroy the string before making _i active
c1._i = 0;          // switch the active member back to _i
```

However, this implementation still has an unresolved question of destroying the string when the union instance is destroyed. Somehow you need to keep track of whether it's the active member or not.

Discriminated unions

What you really need for an all-in-one solution is a so called *discriminated* or *tagged union* which is in fact a struct incorporating an anonymous union together with a data member to keep track of which member of the union is active:

```
class Compact
{
    enum
    {
        Int,
        String
    } _active_type;

    union
    {
        int _i;
        string _s;
    };
};
```

The _active_type member is what I'll use to track which member is active.

This class needs constructors and a destructor:

```
Compact(int i) : _active_type(Int), _i(i)
{}

Compact(const string& s) : _active_type(String)
{
    new (&_s) string(s);
}

~Compact()
{
    if (_active_type == String)
        _s.~string();
}
```

I've created two different constructors: one that initializes the `int` member, and another to initialize the `string` member using placement `new`. Now that I can determine whether the `string` member of the `union` is active, I can destroy it when appropriate.

By the way, because I used an anonymous `union`, I can access its members as if they were members of the outer class, as you can see in the above code sample.

I also need to add some functions to get and set the members. First for the `int` member:

```
int get_int() const
{
    if (_active_type != Int)
        throw runtime_error("Inactive type requested");

    return _i;
}

void set(int i)
{
    if (_active_type == String)
        _s.~string();

    _i = i;
    _active_type = Int;
}
```

Things to note here are that in the getter I check the active type and throw an exception if it isn't `int`. In the setter, I destroy the `string` with an explicit destructor call if necessary.

I need similar functions for the `string` member:

```
const string& get_string() const
{
    if (_active_type != String)
        throw runtime_error("Inactive type requested");

    return _s;
}

void set(const string& s)
{
    if (_active_type == String)
        _s = s;
    else
        new(&_s) string(s);
    _active_type = String;
}
```

Once again, I check the active type in the getter. In the setter, if the active member is a `string`, I can simply perform an assignment. If not, then I construct a `string` using placement `new`.

I could implement copy and move operations for the `Compact` class in a similar fashion.

As you can see, there's a fair amount of code required to implement this class but with the exception of placement `new` and explicit destructor calls, it's pretty much trivial.

So, with all of this machinery in place, I can instantiate and use the class conveniently:

```
Compact c1(100);                // _i is made the active member

cout << c1.get_int() << endl;   // outputs 100

c1.get_string();                // throws because _s is inactive

string s("ABC");
```

```
c1.set(s);                          // makes _s the active member

cout << c1.get_string() << endl; // outputs ABC
```

As the discriminated union keeps track of the active member and throws exceptions in case of invalid usage, it's quite safe to use.

C99 Compatibility Features

C++11 incorporated most of the changes from C99 with the exception of variable length arrays and designated initializers. Variable length arrays weren't included because STL provides better functionality. Designated initializers are redundant because C++ has constructors.

These features aren't C++ features as such and are included primarily for compatibility with C, so I'm only going to list them and will not go into detail about how they work:

- C++11 has a `long long` type which has a size equal or greater than `long`.
- Standard C macros such as `__func__` and `__STDC_HOSTED__` are present. `__func__` provides the name of the current function as a C-style string. `__STDC_HOSTED__` tells you whether the implementation is hosted or freestanding.
- `_Pragma(X)` preprocessor operator is available.
- vararg macros and empty macro arguments have been included.
- Concatenation of wide and narrow strings is supported.

Following on from C99, there is actually a new version of the C standard called C11. However, the C11 features were too new to be considered for inclusion in C++11.

Deprecated and Removed Features

For language features, there isn't a lot to talk about. It makes sense as C++ is big on backward compatibility. Here is the list:

- `auto` can no longer be used as a storage class specifier. It's now used either to declare variables with inferred types or in functions with trailing return types.
- `export` specifier for templates is no longer supported. This feature was only ever implemented in one compiler so it didn't gain acceptance. `export` still remains a keyword though.
- Dynamic exception specifications with the use of the `throw` specifier are deprecated, with `noexcept` taking their place.
- The use of the `register` storage class specifier is deprecated.
- The compiler is no longer supposed to generate a default copy constructor if a class has a user declared copy assignment operator or destructor.
- Similarly, it's not supposed to generate a default copy assignment operator if a class has a user declared copy constructor or destructor.

There are also a few deprecations in the standard library but that's outside the scope of this book.

Writing Cross-Platform Code

If you write code which has to compile on compilers other than GCC, it's useful to know the level of C++11 support they have. You may need to limit the features that you use to be able to compile your project on all relevant platforms.

Not all compilers released around the same time as GCC 4.8 series have full support for the C++11 standard.

Clang has full support both for the language and library features starting from version 3.3.

Intel's C++ compiler doesn't support some of the C++11 language features as of version 14. It doesn't implement revised restrictions on unions or the new rules for initializing objects of static duration on Windows. It also doesn't support some features at all:

- Alignment
- Thread local storage
- Using `sizeof` on non-static data members without an instance
- Inheriting constructors
- User defined literals.

If you have to target Windows and the Visual Studio compiler, then you have to deal with even more constraints.

As of Visual Studio 2013 release, Microsoft's compiler trails behind, although it does implement the major C++11 features.

First let me list the features where the compiler implementation isn't complete:

- It has partial support for move semantics. The compiler doesn't automatically generate move operations, and reference qualifiers for member functions are not implemented.
- Defaulted functions are also not fully supported.
- Instead of the `thread_local` keyword, you have to use `__declspec(thread)` which has similar semantics.
- C99 compatibility is also partial.

There is unfortunately a longer list of features which aren't supported at all. This includes:

- `constexpr`
- Unicode support and literals (although raw literals are supported)
- user defined literals
- `noexcept`
- inline namespaces
- inheriting constructors
- generalized attributes
- `alignas` and `alignof` keywords
- use of `sizeof` on data members without providing an object
- use of arbitrary expressions in template deduction contexts
- and finally, revised restrictions on unions.

If you need to compile code with Visual Studio, you'll have to be mindful of these limitations.

C++14

After the 4.8 release, GCC started introducing C++ features from the upcoming version of the standard, which should be finalized in 2014.

While this update to the standard is intended primarily to fix problems with the C++11 standard, it nonetheless introduces a number of new language features and changes a few others.

To compile code with experimental support for these features enabled, you should use this flag:

```
-std=c++1y
```

I'm going to highlight some of the new features in the draft of the standard. As GCC is a moving target, they may or may not work when you try them. To stay on topic for this book, I will only talk about the language features, but keep in mind that the standard will include library changes too.

Return type deduction for functions

The compiler will be able to deduce the return type of functions:

```
auto square(int n)
{
    return n * n;
}
```

To get the compiler to figure out the return type, you start the declaration with `auto` but don't specify a trailing return type. This is basically extending the return type deduction GCC does for lambdas to regular functions.

Generic lambdas

When specifying lambda parameters, you'll be able to use `auto` instead of a type:

```
auto lambda = [](auto a, auto b) { return a * b; };
```

In C++ pseudocode, this definition is equivalent to the following:

```
struct lambda1
{
    template<typename A, typename B>
    auto operator()(A a, B b) const -> decltype(a * b)
    {
        return a * b;
    }
};

auto lambda = lambda1();
```

So using `auto` for parameters effectively makes the function call operator templated. The types of the parameters will be determined using the rules for template argument deduction rather than the type inference that happens with regular `auto` variables.

Extended capturing in lambdas

Another change to lambdas involves capturing variables. In C++14, captured variables can have an initializing expression:

```
auto timer = [val = system_clock::now()]
    { return system_clock::now() - val; };
// ... do stuff ...
timer();   // returns time since timer creation
```

In this example, `val` is assigned the current time which is then used in the lambda expression. A `timer()` call then returns the time interval since the lambda expression was created. `val` doesn't need to be an existing variable, so this is in effect a mechanism

for adding data members to the lambda. The types of these members are inferred by the compiler.

This allows capturing move-only variables - something that isn't possible in C++11:

```
auto p = make_unique<int>(10);
auto lmb = [p = move(p)] { return *p; };
```

It is equivalent to capturing a variable via move, although what's happening under the hood is a bit more tricky. In reality, the capture block declares a new data member in the lambda called `p`. This member is initialized with `p` from outside the lambda which is cast to an rvalue reference.

Revised restrictions on `constexpr` functions

The proposal lists the following things which are going to be allowed in `constexpr` functions:

- declarations, with the exception of `static`, `thread_local` or uninitialized variables
- `if` and `switch` statements
- looping constructs, including range-based `for`
- modification of objects whose lifetime began within the constant expression evaluation.

This should make `constexpr` functions a lot more versatile.

`constexpr` variable templates

In addition to type and function templates, C++14 will allow `constexpr` variables to be templated. Here is how it will work:

```
template<typename T>
constexpr T pi = T(3.1415926535897932385);
```

There is no new syntax, the already existing template syntax is simply applied to a variable here. This variable can then be used in generic functions:

```
template<typename T>
T area_of_circle_with_radius(T r)
{
    return pi<T> * r * r;
}
```

The idea is that despite having a single initializing expression, it's sometimes useful to vary the type of a variable used in a generic function.

More language changes

There are more language changes on the way. They include:

- Another alternative for type inference which will allow you to use `decltype` inference rules for `auto` variables.
- Aggregate initialization will work for classes with in-class member initializers.
- The ability to use a runtime size for the last dimension of an array allocated on the stack.
- Built-in binary literals prefixed with `0b`.
- It'll be possible to separate digits in a numeric literal with a single quote.
- Another standard attribute called `[[deprecated]]` will be added with the purpose of marking deprecated parts of the code.
- There will be more standard literal suffixes, for example to express time durations.

This isn't an exhaustive list, and it may change before the next version of the standard is finalized.

Beyond C++14

After C++14, the next version of the standard is expected around 2017. Starting from 2012, the committee also decided to do some of the work in parallel with the main standard,

and deliver it in the form of Technical Specifications which are released separately. They can then be incorporated into the standard later.

The idea is that it allows a more predictable schedule for the main standard, while new features can be introduced into the C++ community more often via Technical Specifications.

Currently, these specifications are primarily focused on libraries, with work being done on things like file system manipulation and networking, as well as concepts which are a way of describing type requirements for template parameters.

Conclusion

This is the end of the book. I've shown you all of the features added to the C++ language in the 2011 version of the standard.

You've also found out which features have been deprecated or removed from the language. I talked about the C++11 support in other compilers which you have to consider if you target multiple platforms.

You also got a glimpse of the new features planned for C++14. These features should start appearing in GCC starting from the 4.9 release.

I hope this book has convinced you of the value of the new features in C++. If you incorporate them into your projects, you'll take advantage of the improved abstractions to write more expressive and better performing code.

C++ has become a much more modern language, and there are many new interesting features which will be added to it in the future.

Happy coding!

Contact Information and License Agreement

If you have any comments or feedback, feel free to drop me a line at alex@cpprocks.com. For more information about this and other C++ books, as well as C++ articles, please visit cpprocks.com.

License agreement

4. Address: by supplying a non-New Zealand address during checkout, you confirm that you are presently located outside of New Zealand and will not be making use of the Products in New Zealand.

5. Email: we will send requests for feedback to improve the quality of our products and customer service, and occasional promotional emails to the email address you supply as part of the purchase. You will be able to unsubscribe from these emails at any time.

6. Limitation of liability: every precaution was taken in the preparation of the Products. However, Aotea Studios Ltd assumes no responsibility for errors or omissions, or for loss or damages that may result from the use of information contained in the Products, including but not limited to any indirect, punitive, special, incidental or consequential damages. The Customer is solely responsible for ensuring that their use of the Products is in accordance with the law of their jurisdiction.

7. Prohibition of illegal use: use of the Products which is contrary to the law is prohibited, and immediately terminates the Customer's license to use the Products.