C++11/14 Rocks

VS2013 Edition

Alex Korban

Contents

Introduction

C++11 is the latest version of the C++ standard which replaces C++03. It represents a significant set of changes, including additions to the core language as well as the standard library.

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. With features like lambda expressions and type inference you'll be able to write more expressive and concise code, while the introduction of move semantics will provide opportunities for improved performance. The standard library has been greatly extended and includes new containers and algorithms, as

well as powerful new concepts like smart pointers, regular expressions and a generalized function object class.

This book describes the subset of C++11 features supported by Visual Studio 2013. Many chapters include a section which describes instances of non-standard behavior or bugs in the Visual Studio implementation of the standard. I aimed to describe the standard behavior first, followed by any deviations at the end of the chapter. I recommend that you read both parts, otherwise you may be surprised to find that some aspects of C++11 features don't work for you.

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.

The example code has been compiled (or in case of examples of invalid code, failed to compile) using the 32-bit version of the Visual Studio 2013 compiler.

Type Inference

C++11 provides mechanisms for type inference which make the compiler deduce the types of expressions. I'm starting the book with type inference because it can make your code more concise and expressive. Also, a lot of examples in subsequent chapters rely on these features.

`auto`

I'm sure everyone has been annoyed at least once by having to type out types like `std::map<std::string, std::vector<int>>::iterator`. The new keyword `auto` solves this problem by inferring the type of a variable from the initializing expression:

```
auto i = 5;
auto plane = JetPlane("Boeing 737");
plane.model();
for (auto i = plane.engines().begin();
    i != plane.engines().end(); ++i)
    i->set_power_level(Engine::max_power_level);
```

> **Note:**
>
> The keyword `auto` has 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 been removed from the language.

> **Note:**
>
> 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.

`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
    // ...
}
```

You should be aware that variables declared with `auto` still have a static type just like any other variables, and the compiler needs to have enough information (in the form of an initializing expression) to figure the type out. This is why you can't declare arrays with `auto`, or use it for function parameters or member variables. None of the following will compile:

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

class A
{
    auto _m;              // error
};

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

Using the keyword has a lot of advantages over specifying types explicitly:

- it reduces verbosity of the code and makes it more DRY
- it promotes a higher level of abstraction and coding to interfaces
- it helps future-proof the code as it allows types to change without necessarily requiring code that uses them to change
- it allows for easier refactoring, e.g. changing class names or function return types

- it allows template code to be simpler because now you can avoid specifying some of the types for intermediate expressions by using `auto`
- it's much easier to declare variables of undocumented types
- it allows you to store lambdas for later use. I'll talk about lambdas in the next 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`.

  Note:

  DRY stands for Don't Repeat Yourself. This principle was defined in the book called The Pragmatic Programmer by Dave Thomas and Andy Hunt. It promotes reduced repetition of information.

As a rule of thumb, you should always use `auto` unless you require a type conversion, i.e. 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.

A common objection to `auto` is that it obscures the type information and makes code hard to comprehend for a new programmer. It is a valid consideration, but the benefits outlined above outweigh it. The fact that the types of variables are less obvious is mitigated to a large degree by IntelliSense and by the improved readability achieved by not having type name duplication all over the code.

Consider that similar objections have been made for a long time about other C++ features such as operator overloading, and yet they contribute to more powerful and expressive code. `auto` is in the same camp.

An interesting thing about `auto` is that the compilers have had the ability to infer types for a long time as they had to figure out the types of template arguments. The mechanism behind `auto` is the same as that used for templates:

```
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
```

`auto` can be used to declare multiple variables, 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 and
                                    // reference declarations
auto j = 10, str = "error";         // error: initializing
                                    // expressions of different types
```

You can add `const` or `volatile` qualifiers to your `auto` declarations, as well as turn them into a pointer, a reference or an rvalue reference (rvalue references will be discussed in a later chapter):

```
map<string, int> index;

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

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.

For example:

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

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

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

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

You can use either assignment or copy initialization syntax to initialize your `auto` variables, unless the inferred type has an explicit copy constructor:

```
int i = 10;

auto a = i;
auto b(i);

struct Expl
{
    Expl() {}
    explicit Expl(constExpl&) {}
};

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

decltype

Another 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:

```
int i = 10;
cout << typeid(decltype(i + 1.0)).name() << endl; // outputs "double"
```

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

Compared to `auto`, `decltype` is a more general purpose construct. While `auto` can only be used in variable 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.push_back(10);

decltype(a)::iterator iter = a.end();
```

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;
}
```

The example above uses the new syntax for declaring functions with a trailing return type. This is discussed in detail below.

Because `decltype` is a type specifier, it ignores `const` and `volatile` qualifiers. If you want to get a `const` reference from `decltype`, put the expression into parentheses:

```
is_reference<decltype((i))>::value;  // evaluates to true
```

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.

Trailing return types

The keyword `auto` has a different meaning when it's used in conjunction with function declarations. C++11 introduces an alternative syntax for declaring functions, where the return type is specified *after* the parameters (hence *trailing return type*). In order to use this syntax, you have to start your function declaration with `auto`:

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

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:

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

I want `ReturnType` to be the type of `(x * y)` but I have no way of specifying it, because `x` and `y` aren't in scope yet:

```
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 come to the rescue. Putting the return type after the parameter list allows the parameters to be used to specify the return type.

> **Note:**
>
> There is a hack which allows you to use type names instead of the parameter names, but it's ugly and error-prone:

```
template<typename X, typename Y>
decltype(*(X*)(0) * *(Y*)(0))
mul2(X x, Y y)
{
    return x * y;
}
```

Non-standard behavior and bugs in Visual Studio 2013

`const` qualifier retained incorrectly by `decltype`

In the following example, the compiler fails to remove the `const` qualifier:

```
template<typename T>
struct Point
{
    T x, y;
};

template<typename T1, typename T2>
auto operator+ (const Point<T1>& lhs, const Point<T2>& rhs)
    -> Point<decltype(lhs.x + rhs.x)>
{
    // ...
}

Point<int> x;
Point<double> y;
Point<double> pt = x + y;    // doesn't compile because operator+
                             // returned Point<const double>
                             // instead of Point<double>
```

Using class members and `this` in `decltype` expressions

While VS2013 can handle *some* cases of class members used to define other members with the help of `decltype`, this code doesn't compile:

```
struct NoCompile
{
    string _s;
    decltype(_s.begin()) begin()
    {
        return _s.begin();  // doesn't compile
    }
};
```

Similarly, you can't use `this` in a `decltype` expression:

```
struct A
{
    auto f() -> decltype(this) { return this; }  // doesn't compile
};
```

If you happen to define a `struct` like the one above inside a lambda, you'll get a barrage of additional, more confusing errors.

Referring to a member of a member object inside `decltype`

This will incorrectly fail to compile:

```
struct A
{
    int _val;
};

struct B
{
    A _a;
    decltype(_a._val) _b;  // compile error
};
```

The workaround is to use `declval` to declare `_b`:

```
decltype(declval<A>()._val) _b;  // OK
```

`decltype` doesn't respect access control

You can use private members in `decltype` expressions in Visual Studio:

```
class A
{
    int _val;
};

// this should not compile because member is private (but does)
decltype(A::_val) val;
```

This shouldn't compile as the expression isn't valid.

Using a `friend` function defined inside a class in a `decltype` expression

If a friend function is defined inside a class it's friends with, Visual Studio won't be able to handle it inside a `decltype` expression:

```
struct Friendly
{
    friend int friendly_func(const Friendly& a, const Friendly& b)
    {
        return 10;
    }
};

// the return type causes a compilation error
auto duplify(const Friendly& a) -> decltype(friendly_func(a, a))
{
    return friendly_func(a, a);
}
```

The workaround for successful compilation is to declare `duplify` outside `Friendly`.

There are more cases of more complex `decltype` expressions which can produce incorrect results, or even internal compiler errors. So keep in mind that this feature remains rather buggy, and you need to be careful when using it.

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;
}(); // lamdba 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 VS2013 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, which I'll discuss later in the book. Right now, 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*.

> **Note:** VS2013 allows conversion of non-capturing lambdas into function pointers and also supports arbitrary calling conventions. This is important e.g. when dealing with APIs that expect `__stdcall` function pointers.

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:** VS2013's naming scheme for closure types is `anonymous-namespace::<lambda0>`, `anonymous-namespace::<lambda1>` and so on.

Keep in mind 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.

> **Note:** If you want to watch a captured class member while stepping through lambda code in the debugger, you need to refer to it via `__this` (e.g. `__this->_tanks` in case of the example above).

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 {});                          // error
```

The function template `by_const_ref` simply invokes the callable entity passed to it via `const` reference. If I try to pass a mutable lambda, I will get an error, but a non-mutable lambda is fine.

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 behaviour.

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 behaviour 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. I haven't discussed this in more detail because dynamic exception specifications are now deprecated, and the new `noexcept` specifications introduced in C++11 aren't supported by VS2013.

`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.

Non-standard behavior and bugs in VS2013

Nested lambdas lead to slow compile times and huge object files

If you try nesting lambdas, and particularly if you try to do something non-trivial in your nested lambda expressions such as defining local classes, the compilation times will be

extremely long, and the resulting object files will be huge.

If you have enough nesting, you may also see a compilation error.

Using a lambda as a comparator for an STL container

If you try using a lambda function as a comparator for an STL container, you'll get an error:

```
auto custom_compare = [](const X& l, const X& r) { return false; };
set<X, decltype(custom_compare)> s(custom_compare);
s.insert(X());  // error
```

This is a bug in the empty comparator optimization code. One workaround is to wrap the lambda in `std::function`.

Using a lambda as a default function argument

This doesn't work in Visual Studio:

```
void f(void(*)() = [](){})  // compile error
{}
```

Using `const` from enclosing scope inside a lambda

You might run into problems if you try to use a `const` variable to declare an array inside a lambda:

```
const int N = 5;

auto m1 = [=]
{
    int a[N];    // error
    cout << N << endl;
};
```

The workaround is to either move the constant definition into the lambda, or to declare the array outside the lambda.

Invalid initialization of closure type variables

The compiler allows some really weird behavior when initializing a variable of a closure type:

```
const int x = 10;
auto lmb1 = [=] { return x; };

decltype(lmb1) lmb2 = 200;   // shouldn't compile but does

cout << lmb2();              // outputs 200
```

`lmb2` is supposed to have the same type as `lmb1` - that is, the closure type of the lambda used to initialize `lmb1`. It shouldn't be possible to initialize it with an `int` (and in fact, IntelliSense says so). However, this code compiles, and when `lmb2` is invoked, it outputs 200.

It's also possible to default-initialize a closure type variable in Visual Studio:

```
auto lmb1 = [] { cout << "Hi!"; };
decltype(lmb1) lmb2;  // compiles but shouldn't
lmb2();               // outputs "Hi!"
```

This code sample should produce a compilation error instead.

Lambda returning a capturing lambda

If you try to return a lambda which captures a variable from another lambda expression, it will not compile unless you specify the return type of the outer lambda explicitly (via `std::function`):

```
auto f = [] {           // compile error
    auto n = 10;
    return [=] {
        cout << n << endl;
    };
};
```

Lambdas as ternary operator operands

You can't use lambdas as operands in the conditional operator:

```
false ? []{} : []{};
```

This is because Visual Studio checks whether the operands can be converted to each other instead of checking that they are convertible to a common arithmetic pointer type as the standard requires. The workaround is to use an `if-else` statement or to use explicit casts.

Template Features

Visual Studio 2013 adds several convenient improvements 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.

Let's get started.

Variadic templates

In the previous module, 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)>
```

Naturally, this is because functions, lambda expressions and other callable objects can take an arbitrary number of arguments themselves in the general case.

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

To solve this problem, C++11 introduces variadic templates - that is, templates which can have an arbitrary number of 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. Because it's 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 has the ability to check at compile time that it's supplied with the correct data types in the right order.

What else are variadic templates good for?

Generally speaking, variadic templates simplify metaprogramming and increase 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 in the next chapter.

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 could 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.

Working with parameter packs

What can we do with a parameter pack? 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));
}
```

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`.

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 calling a method on the rest of it. This is how I can use this technique to implement `write_line`:

```
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 other 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. Its template arguments are the type of elements 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.

There aren't any other parameter pack operations available in C++11. 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. Let's look at each of these 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;      // 0
TupleSize<int, double, char>::value;  // 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.

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 show how this template can be used.

Constraining parameter packs to one type

You can also 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

Finally, I'd like to point out that pack expansion can appear in places other than class or function bodies. It can also appear in the member initialization list, lambda captures or 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 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.

One function template, two 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", 125000000);

// construct a tuple of (10, "B737", 125000000)
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 use a lot of templates, you'll probably find a few places in your code which could benefit from variadic templates.

Template aliases

Template aliases allow 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
    {
```

```
        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 case of significant whitespace in the C++ syntax is no longer present, and you can now terminate multiple template type specifications without spaces:

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

Local and unnamed types as template arguments

It's now possible to use local types as template arguments:

```
{
    struct A
    {
        int _a;
        int a()
        {
            return _a;
        }
    };
    vector<A> v;
}
```

Another new possibility is using 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 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 changes allow you to improve locality of your code, and to reduce the number of names floating around.

`extern` template qualifier

This feature isn't really new to Visual C++. It allowed preventing template instantiation via `extern` qualifier through a non-standard extension since C++03. However, it's new to C++11.

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

```
// file1.hpp
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);
}
```

Both `file1.obj` and `file2.obj` will contain an instantiation of `templated_func<int>`, and the compiler will 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);
}
```

This qualifier can also be used for classes (in which case it applies to all members), or 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, it will result in a linker 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.

This function can be used in this manner:

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

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

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

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

Non-standard behavior and bugs in Visual Studio 2013

Unfortunately, the new template features (particularly variadic templates and template aliases) have a significant number of bugs. You can expect to see compilation errors and internal compiler errors when writing non-trivial code which relies on these features. The rest of this section demonstrates some of the bugs.

Empty parameter pack not handled

In some situations, the compiler is unable to handle an empty parameter pack:

```
template<typename T>
size_t sz()
{
    return sizeof(T);
}

template<typename T, typename... Args>
array<size_t, sizeof...(Args) + 1> sizes()
{
    const auto N = sizeof...(Args) + 1;
    array<size_t, N> arr = { sz<T>(), (sz<Args>())... };
    return arr;
}

auto arr1 = sizes<int, float>();  // OK

auto arr2 = sizes<double>();      // compile error
```

In the second instantiation, the parameter pack is empty, but that's valid and shouldn't cause a compilation error.

Variadic template-template parameter troubles

VS2013 has trouble with the following code:

```
template<template<typename...> class T, typename... Args>
auto make_instance(Args... args) -> T<Args...>
{
    return T<Args...> {args...};
}

int main()
{
    make_instance<pair>(1, 2);  // compile error
}
```

This code is standard compliant and should compile.

Here is another setup which produces an unnecessary compiler error:

```
template<template<typename...> class Container,
    typename ValueType, typename... MoreTypes>
void mangle_container(Container<ValueType, MoreTypes...>)
{
    Container<int> c;  // compile error - too few template arguments
}

int main()
{
    vector<int> v{1, 2, 3, 4};
    mangle_container(v);
}
```

Here, the compiler fails to apply the default template arguments of the `vector` template when creating an instantiation of `Container` inside `mangle_container` which is different from the instantiation passed as the function's argument.

Can't use a template alias inside another template

Visual Studio sometimes gets confused if you attempt to instantiate a template alias inside another template:

```
template<typename T>
using MemberPtr = int (T::*)();

template<typename T>
class Cont
{
    MemberPtr<T> _p;  // compile error
};
```

Can't define a private static member using a template alias type

The following sample shows that template aliases are treated differently from ordinary member classes, even though they shouldn't be:

```
class Lookup
{
    class UniqueID {};
    static UniqueID _id;

    template <typename T>
    using Cont = vector<T>;

    static Cont<int> _indexes;
};

Lookup::UniqueID Lookup::_id;   // OK

Lookup::Cont<int> _indexes(0);  // error - Cont is inaccessible
                                // because it's private
```

Default template arguments not applied in class context

If a member function template uses a private member class as a default argument, Visual Studio is unable to use it:

```
class BatchHandler
{
    class BatchCont {};
public:
    template<typename T, typename Cont = BatchCont>
    Cont get_batch(T seed) const
    {
        Cont batch;
        // ... populate batch using seed value ...
        return batch;
    }

};

BatchHandler().get_batch(100);  // compile error: couldn't deduce
                                // argument for Cont
```

The workaround is to make the member class public instead of private.

`sizeof...` always returns 1 if used in a template alias

The following code sample illustrates the issue using `array` template from the standard library:

```
template <typename... T>
using array_alias = array<int, sizeof...(T)>;

static_assert(tuple_size<array_alias<int, int, int>>::value == 3,
    "array size should be 3");   // triggers but shouldn't
```

`tuple_size` returns the size of the `array`, which should be 3. However, in Visual Studio `sizeof...` returns 1 instead, triggering the static assert.

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
- 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, 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 use a data 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 some members can still be uninitialized. 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.

Delegating constructors

Constructors can be a source of duplication 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 ...
};
```

C++11 improves things 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.

However, if you declare your own constructor, it prevents the generation of the default version. If you declare move operations, then the compiler won't generate copy 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, for example, to reinstate the default versions of the constructors:

```
class JetPlane
{
public:
    JetPlane() = default;
    JetPlane(const JetPlane& other) = default;
    JetPlane(JetPlane&&) { /* move operations */ }
};
```

In this example, I'd like to have my own version of the move constructor. But when I declare it, it disables compiler-generated implementations of the default constructor and the copy 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 not-explicit, which isn't always what you want.

You can make the copy operations protected:

```
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&&) { /* move operations */ }
    JetPlane& operator=(JetPlane&&) { /* move operations */ }
};
```

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 marked as 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 a function marked with `delete`, 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 overriding 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.

> **Note:** Instead of `final`, Visual Studio 2010 and 2012 had non-standard keyword sealed with similar semantics. `sealed` is still allowed in VS2013.

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 between the two styles when 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.

Non-standard behavior and bugs in Visual Studio 2013

`default` can't be applied to move operations

Visual Studio compiler isn't able to generate move operations, so the `default` keyword can't be applied to the move constructor or the move assignment operator - you'll get a compilation error.

`final` isn't honored on class templates

If the `final` specifier is applied to a class template, it's ignored by the compiler (however, IntelliSense does pick up on it):

```
template<typename T>
struct Final final
{};

struct A : Final<int>  // compiles but shouldn't
{};
```

`friend` lookup goes too far

One aspect of the extended C++11 `friend` declarations that doesn't work correctly in VS2013 is the lookup outside the innermost enclosing namespace, which shouldn't happen when `class` is used:

```
struct T {};

namespace friendly
{
    struct Amigo
    {
        friend class T;  // should declare friendly::T but looks up
                         // ::T instead
    };
}
```

Nested class access fails with enough nesting

As you have seen, nested classes have gained member access rights. However, Visual Studio has trouble with more complex scenarios, so you may need to change your design to avoid them. For example, this causes a compilation error:

```
class Autopilot
{
protected:
    void adjust_flaps() {}
};

struct Airplane
{
    struct EnhancedAutopilot : public Autopilot
    {
        struct Controller
        {
            void adjust_speed()
            {
                // ...
                EnhancedAutopilot().adjust_flaps(); // error
            }
        };
    };
};
```

Move Semantics and Rvalue References

In some situations in C++98/03 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 would cause each call to `operator+` to construct and return a new temporary string, which would 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.

Classes can now have move constructors and move assignment operators. These 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 some cheap value (from a performance standpoint) to members, e.g. null for pointers.

This can provide significant performance improvements compared to copying. For example, some STL operations in VS2013 are several times faster (compared to VS2008, the last version without move semantics) as the STL implementation was changed to take advantage of move semantics.

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).

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.

Lvalue/rvalue revision

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.

> **Note:**
>
> The standard says that the operand of the address-of operator must be an lvalue, so taking the address to determine whether something is an lvalue is reversing the definition – but it works.

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.

Here are some examples of lvalues:

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

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

```
&(a * b)
&x++
&string("abc") // this is actually allowed in VS unless you compile
               // with /Za to disable language extensions
```

This time taking the address clearly *doesn't* make sense, so all the above are rvalues.

> **Note:**
>
> `this` pointer is a named expression but is an rvalue expression by definition.

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
```

`const` attribute

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

```
JetPlane jet;
jet;                // lvalue

const int max_power_level = 100;
max_power_level;    // const lvalue

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

```
f();                // rvalue

const string g() { return string("G"); }
g();                // const rvalue
```

Non-const rvalues might seem rather useless because 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:

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

Reference initialization

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

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

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
```

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();
```

> **Note:**
>
> There is a slightly mind-bending twist to keep in mind. When you bind a const reference to an rvalue, the reference variable you bound it to is a const lvalue (because it has a name!):

```
const JetPlane& jet_ref4 =
    make_const_jet();
jet_ref4; // jet_ref4 is a const
          // lvalue - it has a name!
```

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:

```
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.

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

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

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

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

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

Note:

A *named* rvalue reference is an *lvalue*, just like a named lvalue reference.

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

```
void f(string& s);
void f(const string& s);
void f(string&& s);
void f(const string&& s);
```

The overloads are resolved using the following rules:

1. the selected overload has to maintain const-correctness (i.e. no binding 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 above 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
```

If, however, the last overload `f(const string&&)` 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 T&&` overload.

xvalues

The C++11 standard talks about a new class of expressions called *xvalues*. Xvalues are lvalues that can be treated as rvalues, which happens when it's known that the lvalue will never be referenced again. It's useful to distinguish them from lvalues because they can be used in move operations.

One example of an xvalue is an object that is returned from a function. In C++11 a return value can be move-constructed from an lvalue return expression. Similarly, an lvalue `throw` expression can be used to move-construct an exception object.

Move semantics implementation

As I mentioned before, 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 are applied to these two constructors, it turns out that non-const rvalues are passed to the move constructor, while lvalues and const rvalues (thanks to overload resolution rule 3 in the previous chapter) are passed to the copy constructor – which is exactly what's required to be able to swap the ownership of the members.

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());
```

The output will be:

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

Note 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 add a move constructor:

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

When we re-run the same program after the addition of a move constructor, the operations change because `vector` will take advantage of move semantics:

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

This can make a significant performance difference in real-world code.

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.

The compiler doesn't provide a default implementation of move constructors and assignment operators. You have to implement them yourself in order to take advantage of move semantics. Note that *the standard* actually has some provisions for automatic generation of move operations - see the non-standard behavior section at the end of the chapter for details.

Moving members

So how do you actually move data? You need to do it member by member. For example, suppose `A` had the following members:

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

The move operations need to get rid of the object's current state (if it's initialized), move ownership, and leave the source object in a valid state (which is normally done in the cheapest possible way performance-wise). For pointers, it normally means assigning `nullptr` to prevent a double delete. In case of `A`, the operations 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._str is a named rvalue
                           // ref so it's an lvalue; use std::move
                           // to force a move operation

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

    return *this;
}
```

It is good practice to empty out the source objects, because if `rhs` happens to be an lvalue which is being explicitly moved from (see next section for details on that), it will continue to exist after the call which invokes the move operation. This can lead to subtle problems.

The move semantics are useful for more than just constructors. You can also provide move overloads for setters in a class:

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

void Plane::set_model(string&& model)
{
    _model = move(model); // careful: model is a named rvalue ref so
                          // it's an lvalue; 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
```

`std::move`

The `move` function in the examples above is necessary so that you can tell the compiler to choose a move overload when you have a named rvalue reference (as in the example), or an lvalue that you know isn't going to be used anymore. Without `move`, the compiler will choose the copy overload.

> **Note:**
>
> move is defined in the `<utility>` header.

If you want to move an lvalue (e.g. when 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));
```

Note that `move` 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 of a constructor or assignment operator. 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. The following sections highlight some things to be aware of.

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
```

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. If you do this:

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

it will result in the `Base`'s *copy constructor* being called (`rhs` is a *named* rvalue so it's in fact an lvalue). You need to strip the name from `rhs` to make the compiler invoke the `Base`'s move constructor:

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

Implementing the move constructor in terms of assignment

Be careful if you implement copy constructors in terms of assignment operators. If you do this:

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

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

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

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 (e.g. `swap`, `unique_ptr`, `sort`) require types to be move constructible, so a type with an explicit move constructor won't work with them.

Move-only types

At this point, with the understanding of move semantics that you've gained, I can introduce move-only types which I mentioned earlier in the book.

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(const MoveOnly& rhs);
    MoveOnly& operator=(const MoveOnly& rhs);
public:
    MoveOnly() : _p(new int(10))
    {}

    ~MoveOnly()
    {
        delete _p;
    }

    MoveOnly(MoveOnly&& rhs)
    {
        *this = move(rhs);
    }

    MoveOnly& operator=(MoveOnly&& rhs)
    {
        if (this == &rhs)
```

```
            return *this;
        _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 move instances of this class but not copy them:

```
MoveOnly a;

MoveOnly b(a);          // doesn't compile

MoveOnly b(move(a));    // OK
MoveOnly c;
c = 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.

Non-standard behavior and bugs in Visual Studio 2013

Move operations never generated by the compiler

The C++11 standard has provisions for the compiler to generate move operations in some situations:

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

However, VS2013 doesn't implement these rules, and never generates a move constructor. There are similar rules for a compiler generated move assignment operator. These aren't implemented in VS2013 either. One of the consequences is that `std::array` containing `unique_ptr` cannot be moved or used within other containers.

Move operations don't disable default copy operations

Once you define a move constructor or a move assignment operator in a class, the default copy operations should be disabled. However, it doesn't happen in Visual Studio:

```
struct A
{
    int _n;

    A(int n) : _n(n)
    {}

    A(A&&) {}

    A& operator=(A&&)
    {
        return *this;
    }
};

A a(1), b(2);

a = b;  // compiles but shouldn't
```

Rvalue reference not recognized correctly

Consider this code:

```
struct A
{
    int _x;
};
```

```
auto get_a = [] { A a = {10}; return a; };

void take_rvalue(int&&)
{}

take_rvalue(A()._x);          // error
```

The result of calling `get_a` should be recognized as an rvalue reference but isn't, and the compiler returns an error. The workaround is to add a `move`:

```
take_rvalue(move(get_a()._x));    // OK
```

Perfect Forwarding

In addition to enabling move semantics, the introduction of rvalue references also solved a long-standing problem of forwarding function arguments which was previously unsolvable in C++ (in the general case at least).

The forwarding problem and solution

The forwarding problem arises when forwarding the arguments to a function via a template. For example, consider the implementation of a `make_shared` template which produces `shared_ptr` instances (`shared_ptr` is a standard library smart pointer discussed in the Smart Pointers chapter).

The `shared_ptr` template takes the type of the object pointed to as its argument. The `shared_ptr` pointer constructor takes a raw pointer as its argument. Let's start with a really basic template which can only forward one argument:

```
template<typename T, typename Arg>
shared_ptr<T> make_shared(Arg arg)
{
    return shared_ptr<T>(new T(arg));
}
```

The issue with this template is that it makes a copy of `arg` so it's not really transparent. One solution is to make the template take the argument by reference instead:

```
template<typename T, typename Arg>
shared_ptr<T> make_shared(Arg& arg)
{
    return shared_ptr<T>(new T(arg));
}
```

However, now the problem is that it can't take an rvalue argument (because of the reference binding rules):

```
make_shared<vector<int>>(10);  // 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_shared2<vector<int>>(a);   // OK, Arg& overload
make_shared2<vector<int>>(10);  // OK, const Arg& overload
```

This works but doesn't scale, because handling all the `const`/non-`const` permutations of 2 arguments requires 4 overloads, 3 arguments requires 8 overloads and so on – 2^n overloads for each set of `n` arguments. You'd have to write 62 overloads to make this template handle between one and 5 arguments.

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>
shared_ptr<T> make_shared(T1&& arg1, T2&& arg2)
{
    return shared_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>
shared_ptr<T> make_shared(Args&&... args)
{
    return shared_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: template argument deduction rules for rvalue references, and reference collapsing. Let's look at how these things work.

Reference collapsing and templates involving rvalue reference arguments

C++98/03 didn't allow a reference to a reference, so the following wouldn't compile:

```
Point p1(10, 10);
typedef Point& PointRef;
PointRef& p2 = p1;
```

C++11 changes this behavior and introduces the following rules:

Reference combination	Transformed into...
`A& &`	`A&`
`A& &&`	`A&`
`A&& &`	`A&`
`A&& &&`	`A&&`

So, 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, all the other combinations turn into an lvalue reference.

In addition, rvalue references and templates interact in a particular way. If you have a template like this:

```
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 perfect forwarding works

As shown above, a perfect forwarding template relies on the `forward` template, which looks like this:

```
template<class T>
T&& forward(typename identity<T>::type& arg)
{
    // forward arg, given explicitly specified type parameter
    return (T&&)arg;
}
```

The `identity` template used for `forward`'s argument type is implemented this way:

```
template<class T>
struct identity
{
    typedef T type; // map T to type unchanged

    const T& operator()(const T& left) const
    {
        // apply identity operator to operand
        return left;
    }
};
```

Let's see what happens when `make_shared` gets called with an lvalue:

```
string model("Boeing 787");
auto sp = make_shared<JetPlane>(model);
```

Following the template instantiation rules above, it results in the following instantiations:

```
shared_ptr<JetPlane> make_shared(JetPlane& && arg1)
{
    return shared_ptr<JetPlane>(
        new JetPlane(forward<JetPlane&>(arg1)));
```

```
}

JetPlane& && forward(identity<JetPlane&>::type& arg)
{
    return (JetPlane& &&) arg;
}
```

And finally, after applying reference collapsing rules this turns into

```
shared_ptr<JetPlane> make_shared(JetPlane& arg1)
{
    return shared_ptr<JetPlane>(new JetPlane(forward<JetPlane&>(arg1)));
}

JetPlane& forward(JetPlane& arg)
{
    return (JetPlane&) arg;
}
```

As a result, the argument is passed through `make_shared` 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_shared<JetPlane>("Boeing 787");
```

The instantiations are now going to be

```
shared_ptr<JetPlane> make_shared(JetPlane&& && arg1)
{
    return shared_ptr<JetPlane>(
        new JetPlane(forward<JetPlane>(arg1)));
}

JetPlane&& forward(identity<JetPlane>::type& arg)
{
    return (JetPlane&&) arg;
}
```

The reason `forward` requires the use of the `identity` template and can't be declared as simply

```
template <typename T>
T&& forward(T&& arg)
{
    return arg;
}
```

is that then `forward` could be called without explicitly specifying the template argument, in which case template argument deduction would be used. This means that when `forward` is passed an lvalue, `T&&` would turn into an lvalue reference, and we don't want that to happen.

So, after collapsing references this code turns into

```
shared_ptr<JetPlane> make_shared(JetPlane&& arg1)
{
    return shared_ptr<JetPlane>(new JetPlane(forward<JetPlane>(arg1)));
}

JetPlane&& forward(JetPlane& arg)
{
    return (JetPlane&&) arg;
}
```

Thus, an rvalue is passed through `make_shared` 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: the implementation of `std::move`

It turns out that the implementation of `std::move` (which was introduced in the previous chapter) 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 the implementation of `move`:

```
template<class T>
typename remove_reference<T>::type&& move(T&& arg)
{
    // forward arg as movable
    return (typename remove_reference<T>::type&&)arg;
}
```

remove_reference is defined like this:

```
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;
};
```

The result of these definitions is that `remove_reference<int>::type`, `remove_reference<int&>::type` and `remove_reference<int&&>::type` all yield `int`.

Let's see how this template works with lvalues and rvalues. When `move` is called with an lvalue argument, e.g.

```
int a = 10;
move(a);
```

it results in the following template instantiation:

```
remove_reference<int&>::type&& move(int& && arg)
{
    return (remove_reference<int&>::type&&) arg;
}
```

which is equivalent to the following after substituting `remove_reference` and collapsing references:

```
int&& move(int& arg)
{
    return (int&&) arg;
}
```

For an rvalue argument, this instantiation is made:

```
remove_reference<int>::type&& move(int&& arg)
{
    return (remove_reference<int>::type&&) arg;
}
```

which is equivalent to

```
int&& move(int&& arg)
{
    return (int&&) arg;
}
```

In other words, `move` returns an rvalue reference when passed either an lvalue or an rvalue.

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;
```

`const` and `volatile` qualifiers are allowed for the iterating variable. You can also specify the type of the iterating variable explicitly:

```
for (int elem : v)
    cout << elem << endl;
```

> **Note:**
>
> Explicit conversions aren't allowed when initializing `elem`.

There is special handling for built-in arrays under the hood. Just like with a container, you can iterate over a C-style array using this syntax:

```
int arr[] = {10, 20, 30, 40};
for (auto elem : arr)
    cout << elem << endl;
```

The `initializer_list` template from the standard library (which I will discuss 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;
```

The `match_results` template from the regex library (discussed in the Regular Expressions chapter) isn't an STL container but it has `begin` and `end` members, so matches can be iterated over with `for` as well:

```
string s = "Arrival NZ1 ETA 00:15 Actual 01:17 Runway 1 Terminal 1";
smatch match_res;   // smatch is a typedef for match_results<string>
regex arrival_regex(
    "([A-Z][A-Z])(\\d+) ETA (\\d+:\\d+) Actual (\\d+:\\d+)");

if (regex_search(s, match_res, arrival_regex))
    // print the whole match and all submatches
    for (const auto& submatch : match_res)
        cout << submatch << endl;
```

When the compiler looks for `begin`/`end`, the member versions of `begin` and `end` take precedence. If they aren't available, then `begin(elem)` and `end(elem)` 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, I can then use the range-based `for` loop to iterate over this container:

```
MyContainer cont;
for (const auto& elem : cont)
    cout << elem << endl;
```

Generally, a range-based `for` loop statement

```
for (elem_decl : seq)
    statement;
```

is equivalent to

```
for (auto iter = seq.begin(), seq_end = seq.end(); iter != seq_end;
    ++iter)
{
    elem_decl = *iter;
    statement;
}
```

or, if member versions of `begin`/`end` aren't available:

```
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`.

Non-standard behavior and bugs in Visual Studio 2013

The compiler has trouble parsing a `do/while` loop nested inside a range-based `for` loop:

```
vector<int> v;
for (auto &elem : v)
    do { /* stuff */ } while (false);  // compile error
```

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. 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
```

nullptr_t is itself defined in terms of nullptr (and specified to behave as a fundamental type):

```
namespace std
{
    typedef decltype(nullptr) nullptr_t;
}
```

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 libraries have been updated so that, for example, a failed dynamic_cast returns nullptr rather than 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, ambiguous(int) overload chosen
ambiguous(nullptr);  // returns true, ambiguous(int*) overload chosen
```

Non-standard behavior and bugs in VS2013

Keep in mind that Visual Studio incorrectly allows `nullptr` to be used as a template argument in some situations:

```
template<int> void non_type_template()
{}

non_type_template<nullptr>();   // compiles but shouldn't
```

IntelliSense correctly highlights this as an error but it compiles nonetheless. However, if you actually attempt to use the template parameter in the above template, you *will* get compilation errors.

The compiler will also erroneously let you perform some arithmetic operations with `nullptr`:

```
auto x = nullptr;
x++;        // compiles but shouldn't
++x;        // compiles but shouldn't
x += 1;     // compiles but shouldn't
```

Once again, IntelliSense correctly highlights these operations on `x` as errors but the compiler disagrees.

`enum` Changes

C++11 introduced strongly typed `enums` in order to improve type safety, and forward declaration semantics for `enums` to reduce coupling and improve compilation times.

Scoped `enums`

If you use `enum class` instead of `enum`, the names in the `enum` will not be exported to the surrounding scope (which is why it's called a scoped `enum`):

```
enum class Proportion
{
    OneHalf,
    OneThird,
    OneQuarter
};

Proportion prop = OneThird;                 // error
Proportion prop2 = Proportion::OneThird;    // OK
```

The `enum` members of a scoped `enum` won't be implicitly converted to integer values either:

```
if (prop == 1)  // error
    // ...
```

Specifying the underlying type

Another addition to `enums` is the ability to specify the underlying type of the enumerants. The type is specified after the `enum` name and must be one of integral types:

```
enum Direction : unsigned short
{
    South,
    West,
    East,
    North
};

cout << sizeof(North) << endl;  // outputs 2

enum Color : double             // error
{
    Black
};
```

This allows you to improve cross-platform compatibility by guaranteeing that the `enum` uses the type you specified (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` behave as before.

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:

```
// 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);
    }
};
```

enums 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)
{}
```

This can help with data hiding, e.g. if you're only distributing the headers of a library.

There are several rules for forward declarations:

1. the forward declaration has to tell the compiler the underlying type
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:

```
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

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

enum class E : short; // error, can't change from unscoped to scoped
enum G : short;        // error, can't change from scoped to unscoped

enum E;                // error, underlying type is required
enum E : int;          // error, different underlying type
enum class G;          // error, different underlying type
enum class H : short; // error, different underlying type

enum class H {};       // OK, this redeclaration is a definition
```

Non-standard behavior and bugs in Visual Studio 2013

Visual Studio will complain incorrectly if an enumerant in a scoped `enum` has the same name as the class it's defined in:

```
struct North
{
    enum class Direction
    {
        South,
        North,    // compile error
        East,
        West
    };
};
```

This is a valid complaint for a regular `enum` but should work fine if it's scoped.

Additionally, it allows forward declaration of a non-scoped `enum`, which isn't allowed by the standard.

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 still check whether a `function` instance has a target before calling the function like this:

```
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.

Non-standard behavior and bugs in Visual Studio 2013

In contradiction with the standard, you won't be able to apply `virtual` or `override` keywords to your explicit conversion operators.

Additionally, Visual Studio sometimes treats explicit conversion operators as implicit. Consider this code:

```
struct A
{
    explicit operator int() const { return 10; }
};

struct B
{
    B(int) {}
};


A a;

B b1{a};    // compiles but shouldn't
B b2(a);    // compiles but shouldn't

B b3 = a;   // compile error - correct behavior
```

Only copy initialization is treated correctly in this example.

Raw String Literals

Raw string 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;
```

This is going to output the string literal verbatim, without any translation. So you can 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)"
```

Note that if you use the character type prefix `L`, 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 very hard to understand.

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 here, 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\nliteral"`, so an actual newline in the source file gets translated into a newline character in the string.

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

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

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

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

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

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

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

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});
```

With 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.

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 VS2013 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 the from a floating point type to an integer type. It also includes implicit any 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 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. 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>> v {unique_ptr<int>(new int(1))};  // error
v.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.

`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`.

<> + {} = ?

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
```

Once again, the solution is to be more explicit:

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

Both of these options provide the compiler with enough information.

Surprising consequences of narrowing

The restriction on narrowing can sometimes result in a somewhat 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 use it only where it's the only option for initialization.

Non-standard behavior and bugs in Visual Studio 2013

If this chapter hasn't convinced you that the new initialization syntax isn't such a good idea, then this section probably will. In addition to all the caveats of the new initialization syntax built into the standard, Visual Studio 2013 adds another layer of glitches on top.

New initialization syntax doesn't work for member arrays

Unfortunately, the example of initializing a member array from the beginning of the chapter doesn't actually work in VS2013:

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

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

You won't be able to work around this by providing an in-class member initializer for the array either.

Aggregate initialization doesn't work in the constructor initialization list

You won't be able to apply aggregate initialization in the constructor initalization list:

```
struct POD
{
    int _n;
};

struct A
{
    POD _pod;

    A() : _pod {10}   // compile error
    {}
};
```

The above should compile but doesn't, because instead of performing aggregate initialization, VS2013 attempts to call `POD`'s copy constructor with an `int`.

Nested initializer lists can cause crashes or memory leaks

If you try to initialize, say, a `map` of `maps` with the help of nested initializer lists, you will get a crash at runtime:

```
unordered_map<int, unordered_map<int, int>> map =
{
    { 10, { { 2, 3 } } },
    { 20, { { 4, 5 } } }
};

auto value = map[10][2];  // crash here
```

A slightly different arrangement results in memory leaks:

```
auto map = unordered_map<int, vector<int>>
{
    {1, {2, 3}},
    {2, {4, 5}},
};

// memory leak
```

Double `delete` of `initializer_list` elements

Perhaps to compensate for the memory leaks in nested lists, in some situations `initializer_list` elements are deleted *twice*. This simple example is enough to demonstrate the issue:

```
struct A
{
    char _data[256];
    A(const string& s)
    {
        cout << "A(): " << this << endl;
    }
    A(const A& rhs)
    {
        cout << "Copy of A: " << this << endl;
    }
    ~A()
    {
        cout << "~A(): " << this << endl;
    }
};

int main()
{
    vector<A> v {A("abc"), A("def") };
    return 0;
}
```

The output will show something like this:

```
A(): 003CFBBC
A(): 003CFCBC
Copy of A: 001F3640
Copy of A: 001F3740
~A(): 003CFCBC
~A(): 003CFBBC
~A(): 003CFBBC
```

```
~A(): 001F3640
~A(): 001F3740
```

First, the elements of the initializer list are constructed. They are then copied into the `vector`. As you can see afterwards, when the elements of the initializer list are subsequently destroyed, the destructor of the first element is called *twice*. The copies are destroyed last.

Nested initializer lists compile when they shouldn't

The following code compiles even though it shouldn't, as `A` in this example doesn't have a constructor taking `initializer_list`:

```
struct A
{
    A(int, int) {}
};

A a {1, 2};       // OK

A b { {1, 2} };  // shouldn't compile but does
```

In fact, you can add as many levels of nesting as you like, and it will still compile.

There are other situations where nested initialization lists will incorrectly *fail* to compile instead, for example when you have an STL container inside a class.

Can't combine `auto` with an initializer list of function pointers

If you try to declare an `auto` variable which refers to an initializer list of function pointers, you'll trigger an internal compiler error:

```
int sum(int a, int b) { return a + b; }
int diff(int a, int b) { return a - b; }

auto list = { sum, diff };  // internal compiler error
```

The workaround is to use an explicit type for the variable instead of `auto`.

Uniform initialization of a type with a user-defined destructor

Through an unfortunate combination of factors, you can trigger another internal compiler error:

```
void f(string, string) {}

f(string{}, string{});  // internal compiler error
```

You're not very likely to encounter this bug, as the prerequisites for this are:

- The function has to take at least two arguments
- The first parameter type has a user-defined destructor
- You use uniform initialization syntax to default-construct all the arguments of the function.

The workaround is not to use uniform initialization syntax, or to invoke something other than the default constructor for one or more arguments.

Keep in mind that there are other possible programs using uniform initialization syntax which will produce internal compiler errors.

Empty parameter pack expansion error when combined with uniform initialization syntax

This bit of code produces a compilation error:

```
struct A {};

template<typename... Args>
struct TypeSize
{
    static const int value = sizeof(decltype(A{Args(0)...}));
};

auto size = TypeSize<>::value;  // compile error
```

The problem is the uniform initialization syntax in the `decltype` expression used to initialize `TypeSize::value`. An empty parameter pack should reduce the `decltype` argument to `A()`. The workaround is to use old initialization syntax instead.

Smart Pointers

Smart pointers are templated classes which encapsulate raw pointers. The concept was present in the C++ library since the introduction of `auto_ptr`, however the new classes are far more useful. Smart pointers allow you to write code at a higher level of abstraction. They manage memory automatically, which helps e.g. to deal with exceptions:

```
Thrower* p = new Thrower;
p->execute();
delete p;
```

> **Note:**
>
> Smart pointers are defined in the `<memory>` header.

The other common problems with raw pointers are memory leaks, dangling pointers and double deletion. In addition to that, using raw pointers requires you to litter your code with `delete` statements (think about a function with multiple returns where you need to ensure memory is freed before every return).

Smart pointers overload the dereference (*) and arrow (->) operators, so using them is very similar to raw pointers.

The previous version of the standard library supplied only one kind of smart pointer, `auto_ptr`. However, it had its share of problems due to the way ownership was transferred when the pointer objects were copied. Because of this, it couldn't be used with STL containers. `auto_ptr` is deprecated in C++11.

There are 3 different types of new smart pointers included in the standard library. They have different behavior and are discussed in detail below.

unique_ptr

`unique_ptr` is very simple conceptually: it's an object that owns another object and manages it through a pointer. When a `unique_ptr` is destroyed, it will dispose of the object it owns:

```
{
    unique_ptr<JetPlane> p(new JetPlane("Boeing 737"));
    cout << p->model();
}   // the JetPlane object will be deallocated here

// the owned object can also be const
unique_ptr<const JetPlane> p_const(new JetPlane("Boeing 737"));
```

I'm only going to discuss the more interesting aspects of the `unique_ptr` interface. `unique_ptr` provides comparison operators and methods like `.get()` and `.reset()` which I'm not going to cover as they are very straightforward.

`unique_ptr` provides exception safety for dynamically allocated memory, and is useful for passing ownership of dynamically allocated memory to a function or returning dynamically allocated memory from a function. `unique_ptr`'s can also be stored in containers and arrays.

It's possible to transfer ownership of the object from one `unique_ptr` to another:

```
unique_ptr<JetPlane> p(new JetPlane("Boeing 737"));
unique_ptr<JetPlane> p_copy(p); // won't compile

// move instead of a copy; p will be set to nullptr afterwards
unique_ptr<JetPlane> new_owner(move(p));
```

Because only one `unique_ptr` can own a given object, it only supports move semantics but not copy.

> **Note:**
>
> `unique_ptr` allows you to use an incomplete type, however the type needs to be complete at the point when `delete[]` is called.

Custom deleters

Smart pointers can also be an application of the RAII (resource acquisition is initialization) pattern with the help of a custom deleter. By default `unique_ptr` uses `delete` to deallocate the object it owns. However, when you declare a `unique_ptr`, you can specify a custom deleter as a template argument, which allows you to use it with objects that need non-standard memory management or even to manage different kinds of resources:

```
struct StreamDeleter
{
    void operator()(ofstream* os) const
    {
        os->close();
    }
};

{
    ofstream log_file;
    log_file.open("log file");
    unique_ptr<ofstream, StreamDeleter> p_log(&log_file);

    *p_log << "Log statement";
}    // log_file.close() gets called
```

The deleter can be a function object, a function or even a lambda.

Array specialization

`unique_ptr` can be used to manage an array. In this case, its interface is different: you can't dereference it via operators `*` or `->`, but you can use the subscript operator (`[]`) instead. The array specialization also prohibits base/derived class conversions.

```
unique_ptr<JetPlane[]> p_fleet(new JetPlane[10]);

*p_fleet;                                    // doesn't compile
unique_ptr<Plane[]> p_base(move(p_fleet));   // doesn't compile

p_fleet[2].model(); // subscript operator returns a reference
```

`make_unique`

`make_unique` is a shortcut for creating a `unique_ptr`:

```
auto up = make_unique<JetPlane>("DC-10");
```

This function template wasn't available in the C++11 standard due to an oversight, but it has been added to C++14. It's one of a handful of C++14 features implemented by VS2013.

Combined with the `auto` keyword, it allows you to save a good number of keystrokes and avoid typing the name of the class twice.

If you need to pass parameters to the object's constructor, you can put them into the `make_unique` call.

Same as `unique_ptr`, it also has a specialization for arrays:

```
auto up = make_unique<JetPlane[]>(25);  // pointer to 25 JetPlanes
```

`make_unique` only performs value initialization of the members of the array. Note that the `unique_ptr<T[N]>` specialization is explicitly disallowed.

shared_ptr

Shared pointers manage memory via reference counting. Each `shared_ptr` object has a pointer to the reference counter. When a `shared_ptr` is constructed, it increments the reference count, and decrements it on destruction. When the reference count reaches zero, the managed object is deallocated. Shared pointers can be used in containers and arrays.

Here is how you use `shared_ptr`:

```
shared_ptr<JetPlane> p(new JetPlane("Airbus A320"));
shared_ptr<JetPlane> p2 = p;    // a copy increases reference count

// the owned object can also be const
shared_ptr<const JetPlane> p_const(new JetPlane("Airbus A320"));
```

I'm only going to discuss the more interesting aspects of the `shared_ptr` interface. `shared_ptr` provides comparison operators and methods like `get()`, `reset()`, `unique()`, `use_count()` which I'm not going to cover as they are very straightforward.

A shared_ptr can also be constructed from a unique_ptr or auto_ptr, provided that pointer is a non-const rvalue. If it's an lvalue, you have to use std::move.

Keep in mind that you should never create two shared_ptr's pointing to the same object, i.e.

```
auto p = new int;
shared_ptr<int> sp1(p);
shared_ptr<int> sp2(p);
```

The code above will lead to a double delete because each of the shared pointers will attempt to deallocate the object when it goes out of scope.

Custom deleters

Like unique_ptr, shared_ptr uses delete by default, but you can provide a different deleter on construction instead. Note that for shared_ptr the deleter isn't a part of its type. For example, if you have some sort of handles, you can automate their management like this:

```
typedef void* HANDLE;
HANDLE create_process() { return 0; }
void close_handle(HANDLE) {}

typedef shared_ptr<void> Handle;

Handle start_process()
{
    return Handle(create_process(), close_handle);
}
```

Thread safety

shared_ptr has the same level of thread safety as built-in types. You can perform read (const) operations from multiple threads simultaneously. You can also perform write operations (i.e. use mutable operations such as operator= or reset) on different shared_ptr instances simultaneously from multiple threads. This includes the case when these instances share the same reference count.

If you need any other type of access, you will need to synchronize it or otherwise you'll get undefined behavior. This code shows three different cases of undefined behavior:

```
shared_ptr<int> p(new int(10));
shared_ptr<int> p2(p);
shared_ptr<int> p3(p);

// thread A
p = p2;     // reads p2, writes p

// thread B
p2.reset(); // writes p2; undefined behavior, simultaneous read/write
```

```
// thread A
p2 = p3;    // reads p3, writes p2

// thread B
// p3 goes out of scope: undefined behavior, the destructor is
// a write operation
```

```
// thread A
p2.reset(new int(1));   // write access

// thread B
p2.reset(new int(2));   // undefined behavior, simultaneous writes
```

Under the hood, `shared_ptr` performs atomic increments/decrements of the reference count.

Performance

`shared_ptr` uses more memory and is slower than raw pointers, however the overhead is small and not significant in the vast majority of cases. Consider that to handle the complex cases of memory management you'll have to create a custom class in any case, and writing a bug-free, thread safe and efficient smart pointer is a non-trivial exercise.

Keep in mind that it's better to pass `shared_ptr` by reference when possible because passing it by value involves changing the reference count. This can affect performance in both single threaded and multi-threaded code.

`make_shared` and `allocate_shared`

`make_shared` is a shortcut for creating a `shared_ptr` which is analogous to `make_unique`:

```
auto sp = make_shared<JetPlane>("DC-10");
```

If you need to pass parameters to the object's constructor, you can put them into the `make_shared` call. `make_shared` has an additional benefit over creating a `shared_ptr` explicitly: the compiler can generate more efficient code when using `make_shared` compared to explicitly declaring a `shared_ptr`, because it doesn't do a separate allocation for the reference count.

On the other hand, `make_shared` doesn't allow the use of a custom deleter, as it can't be distinguished from the managed object's constructor arguments.

`allocate_shared` is analogous to `make_shared`, except it takes a custom allocator as its first argument.

> **Note:** According to Herb Sutter, the intention for C++11 was that you would never have to write `new`. However, because `make_unique` was overlooked in C++11, this vision is only realized in C++14 (and VS2013). Starting from VS2013, you can use references and smart pointers along with `make_shared()`, `make_unique()` and their cousins, and avoid `new` altogether unless you specifically need raw pointers to achieve a behavior not possible with these mechanisms.

`enable_shared_from_this`

Sometimes you have an object that is managed by a shared pointer, and you also need to return a pointer to this from one of the object's methods. The following will cause a double deletion:

```
struct A
{
    shared_ptr<A> get_sp()
    {
        return shared_ptr<A>(this);
    }
};
```

```
shared_ptr<A> sp1(new A);
shared_ptr<A> sp2(sp1->get_sp());
```

It will result in a double deletion because this code constructed two separate `shared_ptr`'s from the same raw pointer. One way to resolve this is to have a member `shared_ptr` in the class. Another solution is to make your class inherit from `enable_shared_from_this`:

```
struct B : public enable_shared_from_this<B>
{
    shared_ptr<B> get_sp()
    {
        return shared_from_this();
    }
};

shared_ptr<B> sp1(new B);
shared_ptr<B> sp2(sp1->get_sp());  // OK: sp1 and sp2 share the ref count
```

Note that a shared pointer that owns the object must already exist before you call `shared_from_this`.

shared_ptr<void>

`shared_ptr<void>` can be used as a generic pointer similarly to `void*`. A `shared_ptr<void>` is constructed in this manner:

```
shared_ptr<void> p_void(new JetPlane);
```

When this instance is destroyed, it will call `~JetPlane` as it should. A `shared_ptr<void>` can be cast back to the right type with `static_pointer_cast` (see next section).

However, be aware that if you pass a `void*` to `shared_ptr<void>`, it will result in undefined behavior:

```
void *p = new JetPlane("DC-9");
shared_ptr<void> p_bad(p);  // this causes undefined behavior
```

Class hierarchies and smart casts

Shared pointers allow you to manage instances of derived classes through pointers to base classes:

```
shared_ptr<Plane> p_base(new JetPlane);
```

> **Note:**
>
> Your classes will need to have virtual destructors to work correctly – same as in the case of raw pointers.

This is useful, e.g. for storing polymorphic objects in containers.

Sometimes you'll want to perform a cast to a derived class. When you have objects managed through a `shared_ptr` instead of a raw pointer, you can't use `static_cast` or `dynamic_cast` without getting a raw pointer first. However, if you attempt to combine shared_ptr with a regular cast, it will result in a double delete:

```
shared_ptr<JetPlane>(static_cast<JetPlane*>(p_base.get()));
```

To resolve this, the C++ standard provides special templates for `shared_ptr` casts:

```
static_pointer_cast<JetPlane>(p_base)->jet_plane_only_method();
dynamic_pointer_cast<JetPlane>(p_base)->jet_plane_only_method();

shared_ptr<const JetPlane> p_const(new JetPlane("Airbus A320"));
// require_service is a non-const method
const_pointer_cast<JetPlane>(p_const)->require_service(date_t(10));
```

> **Note:**
>
> There is a wealth of programming techniques for `shared_ptr` use on the Boost web site:
>
> http://www.boost.org/libs/smart_ptr/sp_techniques.html

weak_ptr

The third type of pointer in the standard library is `weak_ptr`. It points to memory owned by a `shared_ptr` and doesn't own memory itself, so it doesn't increase the reference count.

The primary reason for its existence is that it's still possible to leak memory when using shared pointers. If you have two objects that hold shared pointers to each other, you will end up with an uncollectable cycle. Consider this code:

```
struct B;
struct A
{
    shared_ptr<B> _pb;
};
struct B
{
    shared_ptr<A> _pa;
};

{
    shared_ptr<A> pa(new A);    // A ref count = 1
    shared_ptr<B> pb(new B);    // B ref count = 1
    pa->_pb = pb;               // B ref count = 2
    pb->_pa = pa;               // A ref count = 2
}
```

This is what 's going to happen (see Figure 1 for an illustration):

1. `pa` and `pb` are created. They own instances of `A` and `B`.
2. `_pa` and `_pb` are set so the objects point to each other via `shared_ptr`.
3. `pb` goes out of scope; however,`_pb` points to the B instance, so the reference count is 1, and the instance isn't deallocated.
4. `pa` goes out of scope; similarly, the `B` instance still holds a `shared_ptr` to the A instance, so the A instance can't be deleted.

At this point, the objects are still in memory but the code doesn't have access to them.

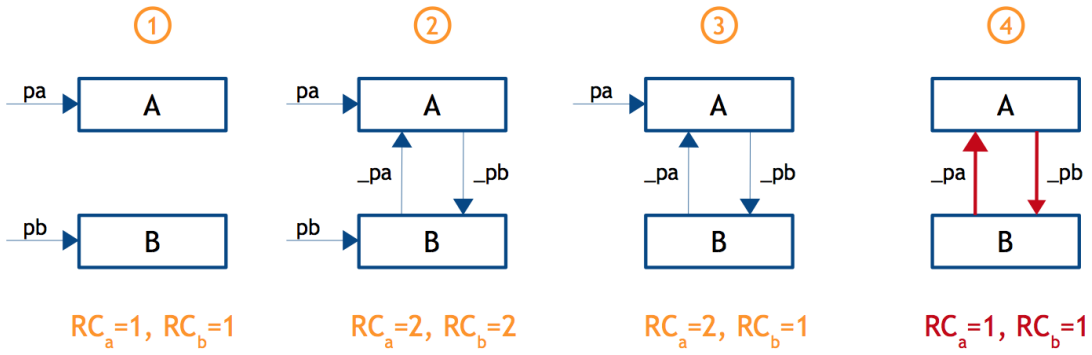

Figure 1: Circular reference memory leak

The cycle can be broken by using a `weak_ptr` instead of a `shared_ptr` in one of the classes.

As shown in Figure 2, in this case `_pa` is a `weak_ptr`. When `pb` goes out of scope in step 3, the reference count for `A` and `B` is 1. When `pa` goes out of scope in step 4, the reference count for `A` reduces to zero, and the object can be deallocated, subsequently triggering the deallocation of the `B` instance.

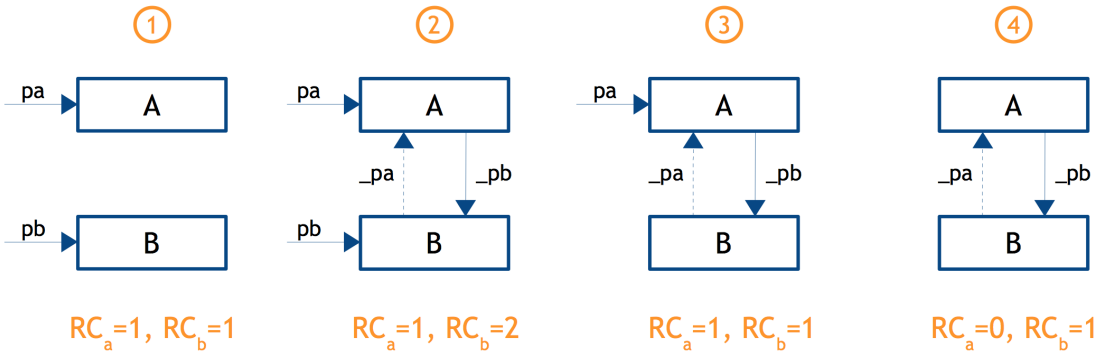

Figure 2: Breaking the cycle with weak_ptr

A `weak_ptr` can also be used as an alternative to `shared_ptr` or raw pointers for observing objects which are owned by a `shared_ptr` in some other part of the code. Its advantage over using raw pointers is that it allows you to check whether it's pointing to a valid object. When the `shared_ptr` releases its resource, `weak_ptrs` expire. You can check whether or not a `weak_ptr` has expired:

```
p_weak.expired()
```

> **Note:**
>
> You could also use `use_count()` to check for expiry but it can be less efficient as it may not take constant time.

In order to access the object pointed to by a `weak_ptr` you need to construct a `shared_ptr`:

```
shared_ptr<A> sp(p_weak);
```

> **Note:**
>
> `shared_ptr` always stores a `weak_ptr` count, even if you aren't using weak pointers.

If the weak pointer is expired, the `shared_ptr` constructor will throw a `bad_weak_ptr` exception. To avoid the exception, you can use `lock()` instead:

```
shared_ptr<A> sp(p_weak.lock());
```

In this case, the `shared_ptr` will be null if the `weak_ptr` is expired.

Non-standard behavior and bugs in Visual Studio 2013

A `bool` can be assigned to `unique_ptr`

`unique_ptr` implementation incorrectly allows a boolean value to be assigned to a `unique_ptr`. This value is then converted into `nullptr` which replaces the object previously pointed to:

```
auto p = make_unique<int>(10);
p = false;           // compiles but shouldn't, replacing
                     // the object with nullptr
cout << *p << endl;  // crash!
```

An `std::array` of `unique_ptr`'s cannot be moved or used within other containers

If you would like to use an `std::array` containing `unique_ptr` objects inside another container, or to be able to move it, you're out of luck for now:

```
typedef array<unique_ptr<int>, 10> PtrArray;

void add_array(PtrArray &&x)
{
    vector<PtrArray> v;
    v.push_back(move(x));   // error
}
```

The reason for this is that the compiler doesn't automatically generate move constructors and move assignment operators, which should happen in some situations.

`shared_ptr` debugging

There is a slight hiccup in the debugger. If you try to look at the value of a `shared_ptr` object where another object of the same name has previously gone out of scope in the same block, the `shared_ptr` will be incorrectly shown as empty:

```
for (auto p = shared_ptr<int>(); p != nullptr;);

shared_ptr<int> tmp(new int(10));
auto p = tmp;

// on the next line, the debugger shows
// p as empty, but 10 prints as expected
cout << *p << endl;
```

Once you get to the next line after `cout`, the debugger will show the correct value.

Tuple Types

Tuple types generalize the concept of `std::pair` for more than two elements to provide you with heterogeneous fixed size collections of values. Tuple size is determined at compile time. You can think of them as classes with members of the specified element tuple types:

```
tuple<int, string, double> t(56, "fifty six", 5.6);
```

> **Note:**
>
> `tuple` is defined in the `<tuple>` header.

This can be useful when you want to deal with aggregate data but don't want to define a named class for it, for example, when returning aggregate information from a function:

```
tuple<string, double, double, int> radar_data =
    get_radar_data(plane_id);
```

Of course, you need to strike a balance between convenience and the readability of your code. Using tuples instead of classes means you don't have member variable names.

Tuples can be used to store not only values but also references:

```
auto a = 10;
double d;
string str;
int arr[10];
tuple<int&, double&, const string&, int(&)[10]> t(a, d, str, arr);
get<0>(t) = 12;
```

> **Note:**
>
> `tuple<>` is valid but there isn't much you can do with it. Similarly, you probably won't be using `tuple<int>` in your code. However, the fact that these types are valid is useful in template metaprogramming, e.g. in the tuple iteration example at the end of the chapter.

Tuple elements need to be initialized. In case of non-reference elements it means that either the default constructor or a copy constructor has to be available. Reference elements also have to be initialized. Arrays can't be copied, so they can only be included in a tuple by reference.

make_tuple

Just like `tuple` is a generalization of pair, `make_tuple` is a generalization of `make_pair`:

```
auto t = make_tuple(10, "ten");
```

`make_tuple` deduces the types of tuple elements from its arguments. If you want to create a tuple with `make_tuple` which has references as its elements, you need to use `ref` and `cref` helpers:

```
int a, b;
auto t2 = make_tuple(ref(a), cref(b), 10);
```

> **Note:**
>
> `ref` and `cref` helpers are declared in the `<functional>` header.

Array arguments result in `const` reference elements by default, so it's not necessary to wrap them with `cref`.

It's possible to store function pointers in a tuple, but it isn't possible to use `make_tuple` to store a reference to a function. You need to construct a tuple manually to store a function reference:

```
make_tuple(&get_radar_data);  // returns tuple<RadarData (*)(int)>

tuple<void(&)(int)> tuple_func_ref(f);  // should compile but doesn't
```

> **Note:**
>
> `ref` and `cref` are used often in combination with tuples, but you can use them anywhere you need to obtain a reference, e.g. with bind, which will be discussed later.

Accessing tuple elements

Once you've constructed a tuple, you'll want to access the values in it. Elements of a tuple can be retrieved by their index using `get`:

```
auto t = make_tuple(10, "ten", 10.0);
auto d = get<2>(t);  // gets 10.0
```

Note that the index passed to `get` is a template parameter, so it has to be a value known at compile time:

```
int var_index = 2;
auto d2 = get<var_index>(t);    // doesn't compile, var_index isn't
                                // const

const int const_index = 2;
auto d3 = get<const_index>(t);  // this works
```

Using `get` with an out of bounds index will result in a "base class undefined" error from the compiler.

`get` can also be used with `std::pair` and `std::array`:

```
pair<string, int> p("ten", 10);
get<1>(p);    // returns 10

array<string, 3> arr;
arr.fill("one");
get<2>(arr);  // returns "one"
```

Consider using `enum` values with `get` instead of numeric indexes to make your code more readable.

Multiple assignment

You can perform multiple assignment of tuple values with `tie`. `tie` creates a tuple of references, so it allows you to assign the values of multiple tuple elements to variables in one line:

```
tuple<int, string, double> return_tuple()
{
    return make_tuple(10, string("ten"), 10.0);
}

int i;
string s;
double d;
tie(i, s, d) = return_tuple();
```

> **Note:**
>
> `tie` works with `std::pair` as well.

You'll need to pass variables of types matching those in the tuple (or constructible from tuple values), otherwise you'll get a compiler error.

If you only need to get some of the elements, you can use the ignore placeholder with `tie`:

```
tie(ignore, ignore, d) = return_tuple();
```

Type information

The `tuple` template makes some type information available at compile time via the `tuple_size` and `tuple_element` templates. For example, here is how you can get some information about the tuple returned by `return_tuple()` from the previous section:

```
tuple_size<decltype(return_tuple())>::value;
is_same<string, tuple_element<1, decltype(return_tuple())>::type>::value;
```

> **Note:**
>
> `tuple_size` and `tuple_element` can be used on `std::pair` and `std::array` as well.

This can be useful, e.g. when using template metaprogramming to work with tuples (for example, see the code to print out tuple elements in the "Advanced use" section below).

If you try to use `tuple_size` on a class derived from `tuple`, you will get a compilation error:

```
class TupleClass : public tuple<int, int>
{};
tuple_size<TupleClass>::value;          // triggers error
```

To get the size, you need to explicitly request the size of the base class:

```
tuple_size<TupleClass::tuple>::value;   // 2
```

> **Note:**
>
> The same idea applies to classes derived from `std::pair` and `std::array`.

Comparison operators

Tuples can be compared using <, <=, ==, !=, >= and >. == and != operators use per element comparisons (all elements have to be equal for the tuples to be equal). The other operators perform lexicographical comparisons.

> **Note:**
>
> Lexicographical order on the Cartesian product AxB of partially ordered sets A and B is defined as (a,b) $\leq$ (a',b') if and only if a $<$ a' or (a $=$ a' and b $\leq$ b')

More advanced uses of `tuple`

Iterating over a tuple with template metaprogramming

Because tuple elements have to be accessed using a compile time index, it isn't possible to iterate over tuple elements using a loop. However, there are two possible solutions for this. One is to use template recursion. First, you'll need a helper template:

```
template <typename Tuple, size_t size>
struct print_tuple_helper
{
    static ostream& print(ostream& s, const string& separator,
        const Tuple& t)
    {
        return
        // print the first (size - 1) elements
        print_tuple_helper<Tuple, size - 1>::print(s, separator, t)
            // print the separator (starting after the 1st element)
            << (size > 1 ? separator : "")
            // print the last element
            << get<size - 1>(t);
    }
};

// this template specialization stops the recursion
template <typename Tuple>
struct print_tuple_helper<Tuple, 0>
{
    static ostream& print(ostream& s, const string&, const Tuple&)
    {
        return s;
    }
};
```

You can then use the helper template to implement the `print` function:

```
template <typename Tuple>
ostream& print_tuple(ostream& s, const string& separator,
    const Tuple& t)
{
    return print_tuple_helper<Tuple,
        tuple_size<Tuple>::value>::print(s, separator, t);
}

// output 5 | literal | abc | 1.25
```

```
print_tuple(cout, " | ",
    make_tuple(5, "literal", string("abc"), 1.25));
```

Another possible approach would be to use variadic templates.

Nested tuples

Tuples can be nested, but be aware that nesting isn't going to be good for the readability of your code, so it should be used sparingly:

```
tuple<int, tuple<int, tuple<int, int>>> t(
    1, make_tuple(2, make_tuple(3, 4)));

get<1>(get<1>(get<1>(t)));  // returns 4
```

Tuple concatenation

The standard defines a variadic template called `tuple_cat` which allows you to concatenate multiple tuples into one:

```
auto t = tuple_cat(make_tuple(1, string("two")),
    make_tuple(3.0, string("four")));
get<3>(t);  // returns the string "four"
```

Non-standard behavior and bugs in Visual Studio 2013

`tuple_size` appears to have been broken in the transition from VS2012 to VS2013. It always returns 0 instead of the correct value.

Binding Arguments with `std::bind`

`std::bind` allows you to create a new function object from a callable object and a mapping of its parameters. Its purpose is to allow you to substitute some or all parameters with fixed values, as well as duplicate and reorder parameters. This can be more convenient than writing a new class or function to achieve the same effect.

> **Note:**
>
> `bind` is made available by including `<functional>`.

The simplest example is binding all parameters to values:

```
long square(int x)
{
    return x * x;
}

auto f = bind(square, 10);  // binds 10 to square()'s parameter
                            // and returns new function object f

auto s = f();               // the value of s is 100
```

> **Note:**
>
> `bind` is a generalization of the functors available in the previous standard, `bind1st` and `bind2nd`, which are now deprecated.

`bind` also provides placeholders which allow you to bind only some of the parameters, and construct a function object which will require the rest of the parameters to be passed to it:

```
long subtract(int x, int y)
{
    return x - y;
```

```
}

auto f = bind(subtract, _1, 30);  // binds 30 to subtract()'s 2nd
                                  // parameter and puts a
                                  // placeholder in place of the 1st

auto s = f(20);          // f requires a value for the placeholder;
                         // the value of s is -10
```

The arguments specified with placeholders are passed by reference (using perfect forwarding which is discussed in chapter 12).

Note that placeholders are in the `std::placeholders` namespace. The common practice is to bring them into the global namespace with `using namespace std::placeholders`. This is assumed in all the examples so I don't have to write `placeholders::_1` etc.

Rearranging and duplicating parameters

Placeholders allow you not only to leave some parameters unbound, but also to change their order:

```
auto f = bind(subtract, _2, _1);
auto s = f(30, 20);  // same as calling subtract(20, 30);
                     // the value of s is -10
```

> **Note:**
>
> **Any** function object produced by `bind` will allow you to pass any number of arguments to it; however, it will only use the arguments specified by placeholders.

You can also duplicate parameters:

```
auto f = bind(subtract, _1, _1);    // duplicate first parameter
auto s = f(10);     // equivalent to subtract(10, 10), so s is 0
```

Passing bound values by reference

By default, if you bind a value to a parameter when constructing a `bind` expression, the value is copied. This is in contrast with placeholder arguments, which are passed by reference. If you need to bind a reference, you have to use the `ref` or `cref` wrapper (discussed previously in the tuple chapter):

```
void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}

int x = 10, y = 20;

auto f = bind(swap, x, y);  // here, x and y are passed by value
f();
cout << "x = " << x << ", y = " << y << endl;  // x = 10, y = 20

auto g = bind(swap, ref(x), ref(y));  // note the ref wrapper
g();
cout << "x = " << x << " y = " << y << endl;   // x = 20, y = 10
```

If you want to pass an argument as a `const` reference, you need to use `cref`:

```
auto h = bind(swap, cref(x), ref(y));  // note the cref wrapper
h();    // error: cannot convert parameter 1 from const int to int&
        // so x is indeed passed by const reference
```

Using `bind` with overloaded functions

When you have overloaded functions, `bind` cannot tell which one you want because you're only giving it the function name, so you have to tell it explicitly. If the functions have different return types, then it's enough to specify the return type of the resulting function object:

```
int max(int x, int y)
{
    return x > y ? x : y;
}

double max(double x, double y)
{
    return x > y ? x : y;
}

auto err = bind(max, _1, _2);  // doesn't compile as max is ambiguous

auto f = bind<int>(max, _1, _2);  // int max(int, int) gets bound
int n = f(10, 20);
```

> **Note:** If you try to pass an overloaded function to `bind`, in some cases you may get an enormous amount of error messages, and unfortunately they won't tell you what's wrong. In other cases, the error clearly tells you that the compiler couldn't disambiguate an overloaded function.

If both overloads have the same return type, then you'll need to cast the function pointer to the exact type you need:

```
double square_root(int x);
double square_root(double x);

auto f = bind((double(*)(double))square_root, 100.0);
```

Using `bind` with member functions

`bind` can be used with member functions. When you do that, you need to bind the object to call the member function on as the first argument:

```
Person p;
JetPlane jet;
```

```
auto f = bind(&JetPlane::load_person, jet, _1);
f(p);   // equivalent to jet.load_person(p)
```

As with other parameters, you can use a placeholder in place of the object, which allows you, for example, to call a member function on every object in a container:

```
// ground the fleet
for_each(fleet.begin(), fleet.end(),
    bind(&JetPlane::require_service, _1, today));
```

bind supports different ways of binding the object. You can pass an object itself or a pointer to object:

```
JetPlane jet;
auto f = bind(&JetPlane::load_person, jet, _1);

// pointers work too
JetPlane* p_jet = new JetPlane();
auto g = bind(&JetPlane::load_person, p_jet, _1);

function<void(const Person&)> JetPlane::get_add_passenger_functor()
{
    // pass *this* when binding a member function from another
    // member function
    return bind(&JetPlane::load_person, this, _1);
}
```

You can even pass a shared_ptr instead of a raw pointer. In fact, you can pass any custom smart pointer type which overloads operator* to return a reference to the object it points to:

```
shared_ptr<JetPlane> sp(new JetPlane);
// binding with shared_ptr
auto f = bind(&JetPlane::load_person, sp, _1);

template<typename T>
class CustomPtr
```

```
{
public:
    T& operator*() { return *_ptr; }
    // ... the rest of implementation
};

CustomPtr<JetPlane> cp(new JetPlane);
// binding with a custom pointer
auto f = bind(&JetPlane::load_person, cp, _1);
```

Since STL iterators have `operator*`, they can also be passed to `bind`.

> **Note:**
>
> As `weak_ptr` doesn't overload `operator*`, so you'll need to construct a `shared_ptr` from it before you can use the object it points to with `bind`.

There is a quirk in using `unique_ptr` with `bind`. Because it only supports move operations but not copy, and `bind` makes a copy of the pointer object, it has to be passed to `bind` by reference:

```
unique_ptr<JetPlane> up(new JetPlane);
auto f = bind(&JetPlane::load_person, ref(up), _1);
```

When using static member functions, the only difference is that you don't pass an object to `bind`:

```
auto f = bind(&JetPlane::is_jet);
f();
```

Binding data members

Object data members are included in the definition of a callable object, so it's possible to use `bind` with them as well:

```
pair<int, double> p(10, 10.0);
auto f = bind(&pair<int, double>::first, _1);
f(p);
```

This could be useful, for example, when you want to perform an operation on a container of pairs (which could be a `vector` of pairs or a `map`):

```
map<string, bool> switches;

// get the number of switches set to true
size_t count = count_if(switches.begin(), switches.end(),
    bind(&map<string, bool>::value_type::second, _1));
```

Using `bind` with function objects

In addition to functions, you can use `bind` with function objects. Just pass an instance of a function object in instead of a function pointer:

```
auto mult_by_2 = bind(multiplies<int>(), 2, _1);
mult_by_2(3);   // returns 6
```

Using `bind` with lambdas

Since lambdas are essentially function objects, it's possible to use them with `bind`:

```
auto lambda = [](int n) -> bool { return n > 10; };
auto f = bind(lambda, _1);
f(20);
```

Function composition with nested `bind` expressions

`bind` expressions can be nested, allowing you to compose functions. A composition of functions `f(x)` and `g(x)` is a function `F` such that `F(x) = f(g(x))`. For instance, you could do this:

```
auto it = find_if(jet.engines().begin(), jet.engines().end(),
    bind(less<int>(), bind(&Engine::get_power_level, _1), 50));
```

In this example the inner bind expression is evaluated first, and the engine power level is passed to the outer `bind` expression, where it's compared to the target level of 50. So the whole expression finds an engine with a power level less than 50.

> **Note:** The Boost version of the bind library implements a number of overloaded operators which allow very convenient code like
>
> ```
> find_if(v.begin(), v.end(), bind(&A::rank, _1) > 10);
> ```
>
> Unfortunately, this functionality isn't available in C++11.

`bind` vs. lambda expressions

At first glance it might seem that `bind` doesn't provide anything that can't be done just as easily with lambda expressions. On one hand, `bind` expressions can be more succinct because they don't require enumerating the types of the arguments, but on the other hand lambda expressions can be easier to read because they look more or less like normal functions and don't need placeholders (`_1`, `_2` etc.). Lambdas also provide more flexibility as they allow multiple statements.

In terms of performance, lambdas are going to be compiled into more efficient code in most situations. `bind` uses function pointers, which means that the compiler is unable to perform inlining, whereas it's possible with lambdas.

There are a couple of things, however, which are possible with `bind` but not with lambdas. `bind` can be used to bind move-only types:

```
unique_ptr<string> up(new string("abc"));
auto bound = bind(&string::size, move(up));
bound();
```

Another situation where `bind` is more powerful is when it's applied to templated functions:

```
struct A
{
    typedef void result_type; // required for bind

    template <typename A, typename B>
    void operator()(const A& a, const B& b) const
    {
        cout << a << ", " << b << endl;
    }
};

auto f = bind(A(), _1, _2);
f("five", 5.0);      // will print "five, 5.0"
f(5, 10);            // will print "5, 10"
```

This essentially means that `bind` allows you to implement compile-time polymorphic behavior, which isn't possible with lambda expressions.

Non-standard behavior and bugs in Visual Studio 2013

Can't pass a reference to `this` as an argument to `bind`

VS2012 introduced a bug which means that you can't pass a reference to `this` as an argument to the functor returned by `bind`. It carried over to VS2013. This is best illustrated by an example. When you pass `ref(this)` as an argument to `bind`, it works as expected:

```
JetPlane jet("Boeing 737");
Person p;

auto add_passenger = bind(&JetPlane::load_person, ref(jet), _1);
add_passenger(p);  // OK
```

But if you try to pass `ref(this)` to the object *returned* by `bind` instead, you'll get a compiler error:

```
auto add_passenger = bind(&JetPlane::load_person, _1, p);
add_passenger(ref(jet));  // error
```

Can't use `bind` with member functions taking rvalue references

Another problem is that `bind` can't handle member functions taking rvalue references as arguments:

```
JetPlane jet;

// load_luggage takes a Luggage&& parameter
auto b = bind(&JetPlane::load_luggage, jet, _1);
b(Luggage());  // error
```

Can't always use `bind` with data members

You can write code like this, which uses `bind` with data members:

```
map<string, bool> switches;

// get the number of switches set to true
size_t count = count_if(switches.begin(), switches.end(),
    bind(&map<string, bool>::value_type::second, _1));
```

This works because the functor returned by `bind` receives each element of the container as its argument. However, due to a bug in the `bind` implementation, it isn't possible to bind a data member through a pointer, smart pointer or iterator pointing to an object. So a simple example like this wouldn't compile:

```
A* pa = new A;
auto f = bind(&A::_a, pa);
f();
delete pa;
```

Generalized Function Objects

Having functions as first class objects is a very powerful feature that allows your code to be flexible and generic. C++11 makes a few steps in this direction with lambda expressions and `std::function`.

> **Note:**
>
> `function` is made available by including the `<functional>` header.

You can pass function pointers or function objects as arguments, however there is no *generic* way to do it. In addition, now we have lambdas, and they can be passed around as well. You've already encountered the `function` template when learning about lambda expressions. It allows you to store lambdas for subsequent use and to pass them as arguments.

However, `function` can be used not only for lambdas, it can also wrap member and non-member function pointers, function objects (functors), the objects returned by `bind` – basically, anything callable. This is great, for example, when you want to implement a generic callback interface. With `function`, you can pass in anything callable as a callback.

Internally, `function` is a templated function object with various specializations for the different callable objects it can store.

Using the `function` template

When constructing a `function` object, its type needs to be specified in the declaration. A function's type consists of its return type followed by a list of parameter types in parentheses:

```
function<float(int)> f1;            // takes an int, returns a float
function<string(string, int)> f2;   // takes a string and an int,
                                    // returns a string

long square(int x)
```

```
{
    return x * x;
}

long (*fp)(int) = square;
function<long(int)> f = fp;  // initialize with function pointer
```

When you are simply wrapping a function pointer in a local `function` object, you can take a shortcut by using `auto`:

```
auto f = square;
```

If you're using something else to initialize your `function` object, you need to be explicit about what you're constructing because `auto` will infer the expression type in those cases:

```
// initialize with a static member function
function<bool()> always_true = JetPlane::is_jet;

// initialize with a functor
function<int (int, int)> mult = multiplies<int>();

// initialize with bind
function<int(int)> mult_2 = bind(multiplies<int>(), 2, _1);

// initialize with lambda
function<bool(int)> is_zero = [](int x) { return x == 0; };
```

> **Note:**
>
> When passing non-member functions or static member functions to `function`, it's optional to use `&`.

Once constructed, the `function` object can be used just like a regular function:

```
always_true();

mult(10, 5);    // returns 50
```

```
mult_2(100);    // returns 200

is_zero(10);    // returns false
```

> **Note:**
>
> When a `function` object is called and it doesn't have a target, its `operator()` throws a `bad_function_call` exception.

Parameter and return type conversions

Another aspect of `function` objects being more flexible when used as function parameters is that the function signature of the callable object passed to `function` doesn't have to match the function signature given as the template argument to `function` *exactly*. It's enough for the parameter types and the return type of the callable object to be convertible to the corresponding types in the `function` object declaration:

```
double square_root(int x);

function<float(int)> f = square_root; // compiles (with a "loss of
                                      // data" warning)
```

Checks and comparisons

It isn't generally possible to compare two `function` objects with each other (e.g. to determine whether they point to the same callable object). However, the `function` template defines the `bool()` conversion operator as well as `==` and `!=` operators for comparisons with `nullptr`, which you can use to check whether a function object contains a target to call:

```
function<long (int)> f;  // initialized to nullptr

if (f == nullptr)        // compares with nullptr
    f = square;

if (f)                   // evaluates to bool
```

```
{
    auto s = f(2);
    cout << s << endl;   // outputs 4
}
```

Performance and code size

In terms of performance and size, `function` inevitably introduces some overhead. `function` objects are going to be larger than a function pointer (because they are functors). Invoking a `function` object will require one call through a function pointer, or two in case of calling a free function.

On the plus side, `function` can limit the number of template instantiations from your code. When you have a template with a `function` parameter, it only needs to be instantiated once for each particular calling interface. Of course, `function` is a template itself, so it will need to be instantiated for each variation of its contents (function pointer, functor etc.), however this may result in fewer template instantiations and less generated code overall.

Non-standard behavior and bugs in Visual Studio 2013

No support for move-only types

`function` doesn't support move only types in VS2013, so you'll need to make any function objects you plan to use with it copyable.

Can't use `std::function` with member functions

It should be possible to use `function` with member functions, with various ways of passing `this`. However, this functionality was broken in the transition from VS2010 to VS2012 and remains broken since then. All the following code should compile but doesn't:

```
JetPlane jet;
Person p;

function<void(JetPlane&, const Person&)> f1 = &JetPlane::load_person;
```

```
f1(jet, p);

function<void(JetPlane*, const Person&)> f2 = &JetPlane::load_person;
f2(&jet, p);

shared_ptr<JetPlane> sp(new JetPlane);
function<void(shared_ptr<JetPlane>, const Person&)> f3 =
    &JetPlane::load_person;
f3(sp, p);
```

The workaround in all three cases is to wrap the function pointer with `mem_fn`:

```
function<void(JetPlane&, const Person&)> f1 =
    mem_fn(&JetPlane::load_person);
f1(jet, p);
```

`void`-returning `std::function` can't swallow return type

The C++ standard allows you to pass a callable object with *any* return type to an `std::function` returning `void`. That is, a `void`-returning `function` swallows the return type of its callable object. However, this behavior isn't supported in VS2013 and causes a compilation error.

Regular Expressions

Regular expressions (or regexes for short) are a powerful mini-language for analyzing and manipulating strings. They are a part of many programming languages, particularly dynamic ones. However, until the latest C++ standard came out, you had to use a third party library to get regex support in C++.

> **Note:**
>
> Regular expression functionality is made available by including `<regex>`.

Regexes can be used for a variety of tasks like input validation, parsing and extracting information from strings, search & replace, and tokenizing. You can check that the input from a user is an integer or a phone number, or replace passwords in a body of text with asterisks, or split program options into an array, all with just a few lines of code.

> **Note:**
>
> Boost has had a regular expression library for a long time, and it was adopted into C++11 with minor changes.

A regular expression is a *match pattern* expressed as a string. For example:

```
\d+:\d+(am|pm)
```

A regular expression can *match* the string it's applied to. The above pattern will match strings like `12:01am` or `1:20pm`.

There are different grammars for writing regexes, and the C++11 standard declares support for a number of them: modified ECMAScript, POSIX basic, POSIX extended, awk, grep and egrep. ECMAScript is the default, and it's the standardized version of the regex syntax found in Perl. If you've worked with Unix/Linux you might be familiar with the other grammars and might find it easier to work with them. If not, ECMAScript grammar is versatile and powerful, and you should probably keep the default.

The rest of the chapter deals with the ECMAScript flavor of the grammar.

Syntax

This section is an overview of the ECMAScript regular expression syntax. It's not exhaustive, my intention is to show the main regex capabilities and to make the following examples understandable.

Wildcards

The full stop can be used to match any single character, e.g. `x.z` will match `xyz` or `xaz` but not `xaaz`.

Repetition

Repeats are applied to the preceding group of symbols and are used to specify the number of times the group should be matched.

Symbol	What it does	Example	Will match	Won't match
?	match 0 or 1 times	x?z	z or xz	az or xy
*	match 0 or more times	x*z	z, xz, xxz etc.	xy
+	match 1 or more times	x+z	xz, xxz etc.	z or xy
{n}	match exactly n times	x{2}z	xxz	xz or xxxz
{n, }	match at least n times	x{2,}z	xxz, xxxz etc.	xz
{n, m}	match n to m times	x{1,2}z	xz or xxz	z or xxxz

Note:

The closest thing to a C++11 regex language reference that I know of is Pete Becker's book "The C++ Standard Library Extensions". It's based on TR1 libraries but to my knowledge the regex library has been incorporated into C++11 unchanged from TR1.

Another option is the Boost regex documentation (http://www.boost.org/libs/regex), since C++11 functionality is based on it.

You can use online tools to help you write and quickly try out your regular

expressions. A couple of examples that support the ECMAScript grammar are http://regexpal.com http://gskinner.com/RegExr/

Character sets

Character sets are used to specify a match which can be any of the characters in the set. They can be specified either by enumerating the symbols, e.g. `[xyz]` or by specifying a range, e.g. `[a-z]`. It's also possible to specify several ranges at once, e.g. `[a-zA-Z]`.

`^` can be used to match characters which are not in the set: `[^xyz]` will match any character which isn't `x`, `y` or `z`.

Character classes

Character classes are named predefined character sets. They are an extension of the ECMAScript grammar added to C++11. They are specified as `[:name:]`. Here are the available classes:

Class	What it matches
`alpha`	lowercase and uppercase letters
`digit` or `d`	digits
`alnum` or `w`	characters from either alpha or digit classes
`space` or `s`	whitespace characters
`blank`	space or tab
`cntrl`	file format escape characters (`\n`, `\r` etc.)
`punct`	punctuation characters
`lower`	lowercase letters
`upper`	uppercase letters
`graph`	characters from lower, upper, digit or punct
`print`	characters from either graph or space
`xdigit`	hexadecimal digits (including both lowercase and uppercase a-f)

Classes are used inside range specifications, so for example an expression to match digits is `[[:digit:]]`.

There is a shorthand notation for some ranges using backslash:

Range	Shorthand
`[[:digit:]]`	`\d`
`[[:space:]]`	`\s`
`[_[:alnum:]]`	`\w`

Using uppercase letters instead of lowercase letters matches characters which are not in the class instead, e.g. `\D` is equivalent to `[^[:digit:]]`.

Anchors

Anchors are used to match the beginning or the end of a string. Use `^` to match the beginning of a string, and `$` to match the end. These anchors can only appear at the start or the end of the regex, respectively. For example, `xyz$` will match `vwxyz` but won't match `xyzzz`.

Or

The symbol `|` can be used as an or operator. For example, `x|y` will match either `x` or `y`.

Word boundaries

`\b` can be used to match a word boundary (and `\B` for anything other than a word boundary). `\b` will match either the first character of a word, or the first character after a word. It can also match the beginning of the input if the first character is from the `[\w]` class, or the end of the input if the last character is from `[\w]`.

Capture groups and back references

You can use parentheses to mark parts of a regex. The marked parts are called *capture groups*, and they can be used by the *regex_search* algorithm to match substrings of a

match string (details below).

You can also refer to capture groups in the regex itself using `\1`, `\2` etc. (these are called back references). For example, `(\d+)-\1` will match groups of same digits separated by a dash, such as `11-11` or `999-999`, but not `11-12` or `899-999`.

Capture groups can be nested. Nested capture groups are counted in the order of the opening parentheses:

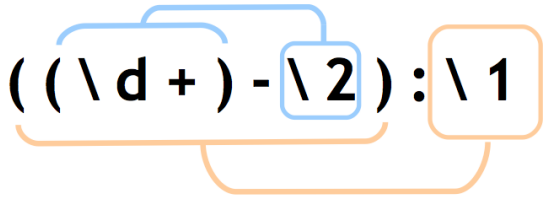

Figure 3: Nested capture group order

Escaping

When you want to match any characters used in the regular expression language itself (like `.`, `?`, or `$`), they need to be escaped with a backslash. The backslash itself needs to be escaped in C++ string literals, meaning that you always end up using a double backslash. For example, `\\?` will match a question mark, and `\\\\` will match a single backslash. Alternatively, if you use raw string literals, then you can write `\?` or `\\` to match a question mark or a backslash, respectively.

Analyzing strings and extracting information

Let's look at an example. Suppose that you have a log file with the scheduled and actual airplane landing times, and you want to analyze how closely different airlines stick to the schedule.

The log contains a lot of different types of records, including arrival records. As an added complication, the arrival log lines can be in slightly different formats:

```
Flight arrival: SQ521 ETA 12:35 Actual 12:33, Runway 2, Terminal 3
Arrival NZ1 ETA 00:15 Actual 01:17 Runway 1 Terminal 1
```

You want to extract and analyze the components of the substrings highlighted in blue, and regular expressions are a great tool for it.

The two main regex algorithms are `regex_match` and `regex_search`. `regex_match` checks whether the whole string matches the given regex, and `regex_search` checks whether there is a substring within a string that matches the regex:

```
regex_match("Flight landed at 12:32", regex("landed"));   // false

regex_search("Flight landed at 12:32", regex("landed"));  // true
```

For the purposes of extracting arrival times, `regex_search` is a good tool:

```
ifstream f("airport-log.txt");
string s;
regex arrival_regex(R"([A-Z][A-Z]\d+ ETA \d+:\d+ Actual \d+:\d+)");

while (getline(f, s))
    if (regex_search(s, arrival_regex))
        cout << s << endl;

f.close();
```

Note:

Longer regexes can easily become very hard to decipher. It helps to break them up into segments and add some comments:

```
regex arrival_regex(
    R"(([A-Z][A-Z])(\d+) " // Flight no.
    "ETA "
    "(\d+:\d+) "          // ETA
    "Actual "
    "(\d+:\d+))");         // Actual time
```

The code above allows us to find (and print out) the log lines containing arrival times. The next step is to extract the individual bits of data from these lines, which can be done by using parentheses to mark *capture groups* in the regex:

```
regex arrival_regex(R"(([A-Z][A-Z])(\d+) ETA (\d+:\d+) "
    "Actual (\d+:\d+))");
```

Running `regex_search` with this regex will populate the match result with *submatches* based on capture groups, so now we can output each of the elements identified with brackets in the regex separately:

```
ifstream f("airport-log.txt");
string s;
regex arrival_regex(R"(([A-Z][A-Z])(\d+) ETA (\d+:\d+) Actual +:\d+))");

while (getline(f, s))
{
    smatch match_res;
    if (regex_search(s, match_res, arrival_regex))
    {
        for_each(match_res.begin() + 1, match_res.end(),
            [](const string& submatch) { cout << submatch << endl; });
    }
}
```

> **Note:**
>
> when `regex_match` or `regex_search` returns `false`, only `empty()` and `size()` methods of `match_results` can be called, anything else is undefined.

`regex_search` populates the `match_res` object. Its type `smatch` is a `typedef` for `match_results<string::const_iterator>`. There is also `cmatch`, which is a `typedef` for `match_results<const char*>`.

Keep in mind that `match_res[0]` returns a string matching the whole regex, and capture group submatches begin from index 1.

> **Note:**

> Don't use `regex_search` in a loop (with iterators into the source string) to find all occurrences of a pattern in the input. It can produce incorrect results when you're using anchors, word boundaries etc. in your regex. It can also result in an infinite loop if your regex matches empty string. Use regex iterators when you have more than one match instead (see next section).

Flags

When constructing a regex, you can pass in a number of flags:

Flag	Effect
`icase`	Perform case-insensitive matching
`nosubs`	Don't store sub-matches in the *match_results* object
`optimize`	Pay more attention to matching speed instead of the speed of constructing a regex object. Note that it may take much longer to construct a regex object with this flag, so use it only when you really need to speed up the matching
`collate`	Make character ranges locale sensitive

For example, if you specify the `icase` flag, the following will return `true`:

```
regex_search("Flight LANDED at 12:32",
    regex("landed", regex_constants::icase));
```

The following flags modify the behavior of `regex_match` and are useful, for example, when you match substrings of the input strings against the regex:

Flag	Effect
`match_not_bol`	Don't treat the first position in the input as the beginning of line
`match_not_eol`	Don't treat the past-the-end position in the input as the end of a line
`match_not_bow`	Don't treat the first position in the input as the beginning of a word

`match_not_eow`	Don't treat the past-the-end position in the input as the end of a word
`match_any`	Any match is acceptable when more than one match is possible
`match_not_null`	Don't match an empty input
`match_continuous`	Don't search for matches other than at the beginning of the input
`match_prev_avail`	`--first` is a valid iterator; if set, ignore `match_not_bol` and `match_not_bow`

For example, this bit of code will return `false`:

```
regex_match("landed",
    regex("landed$", regex_constants::match_not_eol));
```

Handling multiple matches with iterators

So far we've had only one match (with multiple sub-matches) in the input string. But what if you want to extract multiple matches? For example, let's say you had a log containing flights and gate numbers they used:

```
NZ1 - G1; SQ520 - G17; UA345 - G33; UA432 - G1; SQ127 - G33
```

If you want to count the number of times each gate has been used, you need to match multiple gate numbers in this string. You can employ the *regex_iterator* template for this purpose:

```
string input("NZ1 - G1; SQ520 - G17; UA345 - G33; "
    "UA432 - G1; SQ127 - G33");
regex gate_regex(R"(- G(\d+))");

map<string, int> counts;

// count how many times gates have been used
for (sregex_iterator it(input.begin(), input.end(), gate_regex);
    it != sregex_iterator(); ++it)
```

```
    ++counts[(*it)[1]];

// output the counts
for(auto i = counts.begin(); i != counts.end(); ++i)
    cout << i->first << ": " << i->second << endl;
```

> **Note:**
>
> Don't construct a regex iterator with a temporary regex object, because the iterator stores a pointer to the regex internally.

`sregex_iterator` is a `typedef` for `regex_iterator<string::const_iterator>`, while `cregex_iterator` is a `typedef` for `regex_iterator<const char*>`. Dereferencing an `sregex_iterator` gives you an *smatch* object.

The regex library also provides `regex_token_iterator`, which can be used to iterate over capture groups across all the matches in the input string.

`regex_token_iterator` can iterate over capture groups with a particular index, so counts in the previous example could be populated like this instead:

```
for (sregex_token_iterator it(input.begin(), input.end(),
        gate_regex, 1);
    it != sregex_token_iterator(); ++it)
    ++counts[*it];
```

> **Note:**
>
> `sregex_token_iterator` and `cregex_token_iterator` are `typedefs` analogous to those for the `regex_iterator` template.

The default value of `index` is 0, which makes `regex_token_iterator` iterate over whole matches:

```
string input("NZ1 - G1; SQ520 - G17; UA345 - G33");
regex gate_regex(R"(([A-Z]{2}\d+) - G(\d+))");

// the default value for index is 0, which means iterating over
// whole matches
```

```
for (sregex_token_iterator it(input.begin(), input.end(), gate_regex);
    it != sregex_token_iterator(); ++it)
    cout << *it << endl;

// Outputs:
// NZ1 - G1
// SQ520 - G17
// UA345 - G33
```

If you supply a list of capture group indexes in a vector (or an array), the iterator will only iterate over the capture groups with those indexes:

```
string input("NZ1 - G1; SQ520 - G17; UA345 - G33");
regex gate_regex(R"(([A-Z]{2})(\d+) - G(\d+))");
int indexes[] = {1, 3};

for (sregex_token_iterator it(input.begin(), input.end(),
        gate_regex, indexes);
    it != sregex_token_iterator(); ++it)
    cout << *it << " - ";

// Outputs: NZ - 1 - SQ - 17 - UA - 33 -
```

Tokenization

When the `regex_token_iterator` is constructed with -1 as the capture group index, it will be in tokenization mode. This means that it will iterate over all the sub-strings in the source string which don't match the regex, allowing you to tokenize the input:

```
string s = "This is a sample string";
regex white_space("\\s"); // the regex for the separator

sregex_token_iterator token_it(s.begin(), s.end(), white_space, -1);

// output each word in the string on a new line
for (auto it = token_it; it != sregex_token_iterator(); ++it)
    cout << *it << endl;
```

As the regex describes the separator, `token_it` iterates over everything that isn't a separator, i.e. each word in the input string.

Searching and replacing

The regex library also allows you to replace portions of a string using `regex_replace`. For example, you could replace passwords in a log file with asterisks:

```
string credentials = "Username: cpprocks Password: <abc123>";
regex rx(R"(<[\w]+>)");

auto scrubbed = regex_replace(credentials, rx, string("****"));
cout << scrubbed << endl;
// Outputs: "Username: cpprocks Password: ****"
```

By default this algorithm will replace all occurrences of the pattern it finds, but it can be configured to replace only the first occurrence:

```
string str = "12:19.57";
regex rx("\\d+");

// global replace, res is 0:0.0
auto res = regex_replace(str, rx, string("0"));

// first occurrence only, res is 0:19.57
res = regex_replace(str, rx, string("0"),
    regex_constants::format_first_only);
```

If you need to perform a replace with more complicated conditions, you'll have to process the matches and construct the result string yourself.

You aren't limited to replacing matches with fixed text. The first parameter of `regex_replace` is actually a *formatting string* which can refer to sub-matches in the source string. The syntax for references is as follows:

Reference	What it refers to

$0 or $&	the string matching the whole regex
$n	the string matching the n-th capture group, where n >= 1
$‘	the part of the source string that comes before the substring in $0
$’	the part of the source string that comes after the substring in $0

For example, given the regex (c+)(d+)ef and the input abccddefgg, the format specifiers will denote the following parts of the input:

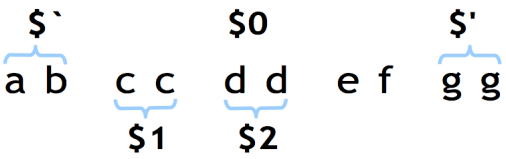

Figure 4: References to parts of the input string

Using formatting strings gives you a lot of flexibility. For example, if you needed to convert the arrival log from previous examples into HTML, you could add a bit of markup to the records easily:

```
regex arrival_regex(R"(([A-Z][A-Z]\d+) ETA (\d+:\d+) "
    "Actual (\d+:\d+))");
string arrival_record("Arrival NZ1 ETA 00:15 Actual 01:17 Runway 1 "
    "Terminal 1");

auto res = regex_replace(arrival_record, arrival_regex,
    string("<strong>$1</strong> scheduled for <em>$2</em> "
           "landed at <em>$3</em>"));

cout << res << endl;
// Outputs: "Arrival <strong>NZ1</strong> scheduled for
//     <em>00:15</em> landed at <em>01:17</em> Runway 1 Terminal 1"
```

regex_replace can also be configured not to output anything that doesn't match the regex pattern. For example, if you were extracting flight airlines, numbers and times from the arrival log into a CSV file, you could try something like this:

```
regex arrival_regex(R"(([A-Z][A-Z])(\d+) ETA (\d+:\d+) "
    "Actual (\d+:\d+))");
string arrival_record("Arrival NZ1 ETA 00:15 Actual 01:17 Runway 1 "
    "Terminal 1");

auto csv_output = regex_replace(arrival_record, arrival_regex,
    string("$1,$2,$3,$4"));

cout << csv_output << endl;
// Outputs: "Arrival NZ,1,00:15,01:17 Runway 1 Terminal 1"
```

This output isn't quite right because it still has parts of the input string at the start and end of it. The `format_no_copy` flag will help in this case. When you pass it to `regex_replace`, you'll get the desired output:

```
csv_output = regex_replace(arrival_record, arrival_regex,
     string("$1,$2,$3,$4"), regex_constants::format_no_copy);

cout << csv_output << endl;
// Outputs: "NZ,1,00:15,01:17"
```

Unicode support and localization

In addition to the functionality discussed above, the regex library has support for wide characters via `wregex`, `wsmatch`, `wsregex_iterator` and so on – every `typedef` has a corresponding wide character version, and the algorithms support wide characters as well.

There are two features supported by the ECMAScript regex grammar which can be helpful when dealing with localized input: equivalence classes and collating elements.

An equivalence class consists of characters (and collating elements) that are the same when case, accents or local-specific tailorings are ignored. An equivalence class can be specified in a regular expression as `[[=c=]]` where c is a sample character or collating element from that class.

A collating element is a single character or a sequence of characters that collates as a single character. For example, ae could be treated as a single collating element in some locales, and would be specified in a regex as `[.ae.]`. Collating elements can be used like single characters, e.g. they can be used as the start or the end of a range:

```
[[.ae.]-z]
```

Attaching a locale to a regex object

The regex object can be imbued with a locale, which will cause locale specific matching behavior when your regex uses collation and equivalence classes.

For example, the regex `[a-f]` imbued with a French locale should match é. If you are using the Finnish locale, `\w` should match ä but `[a-z]` shouldn't, because ä, å and ö collate after z in Finnish.

Note that a call to *imbue* invalidates the regular expression (because the regex doesn't store a copy of the regex string) so you'll need to assign a new regular expression to the regex object:

```
my_regex.imbue(locale(""));
my_regex.assign(regex_str, regex_flags);
```

Compile Time Assertions and Type Traits

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 they don't incur a runtime overhead.

Many compile time checks are implemented with 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.

In order to apply compile time assertions to type checking, it's necessary to be able to retrieve type properties and to manipulate types. The `type_traits` library provides a variety of templates to allow you to do this.

`static_assert`

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");
    // ...
}
```

Note that `static_assert` allows you to provide your own message which 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
```

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
```

A `static_assert` can be in any place where a declaration is allowed, not just inside a function body – in the above example it's in the class definition.

Type traits

Here is how you can query the type properties:

```
cout << is_integral<int>::value << endl;      // outputs 1
cout << is_integral<double>::value << endl;   // outputs 0
```

There are several categories of templates in the `type_traits` library. Primary traits allow you to find out whether a type belongs to one of the main categories. Composite traits are convenient combinations of the primary traits. Type properties let you query more fine grained attributes of types.

Primary traits

Template	**`::value` is `true` if the argument is...**
`is_void`	`void`
`is_integral`	an integral type
`is_floating_point`	a floating-point type
`is_array`	an array type
`is_enum`	an enumeration type
`is_union`	a `union` type
`is_class`	a class type (but not `union` type)
`is_function`	a function type
`is_lvalue_reference`	an lvalue reference
`is_rvalue_reference`	an rvalue reference
`is_member_object_pointer`	a pointer to a non-static member object
`is_member_function_pointer`	a pointer to a non-static member function
`is_pointer`	a pointer type (but not a pointer to a non-static member object or function)

`bool` is considered an integral type:

```
is_integral<bool>::value;      // true
```

Note that if you pass a reference to an array to `is_array`, it will return `false`:

```
is_array<int(&)[2]>::value;    // false
is_array<int(&&)[2]>::value;   // false
```

Similarly, `is_class`, `is_floating_point` and other non-pointer and non-reference related primary traits will return `false` when passed a reference or pointer instead of the actual type.

Composite traits

These are convenient combinations of the primary traits.

Template	`::value` is `true` if the argument is...
`is_reference`	`is_lvalue_reference || is_rvalue_reference`
`is_arithmetic`	`is_integral || is_floating_point`
`is_fundamental`	`is_arithmetic || is_void`
`is_object`	`!is_function && !is_reference && !is_void`
`is_scalar`	`is_arithmetic || is_enum || is_pointer || is_member_pointer`
`is_compound`	`!is_fundamental`
`is_member_pointer`	`is_member_object_pointer || is_member_function_pointer`

Type properties

You can also query more fined grained properties of types.

Template	`::value` is `true` if the argument is...
`is_const`	`const` qualified
`is_volatile`	`volatile` qualified
`is_trivial`	`is_trivially_copyable && is_trivially_default_constructible`
`is_trivially_copyable`	trivially copyable
`is_standard_layout`	a standard layout type
`is_pod`	a POD type
`is_literal_type`	a literal type
`is_empty`	a class type, but not a `union` type, with no non-static data members other than bit-fields of length 0, no virtual member functions, no virtual base classes, and no base class `B` such that `is_empty::value` is `false`
`is_polymorphic`	a class that declares or inherits a virtual function
`is_abstract`	an abstract class
`is_signed`	`is_arithmetic && T(-1) < T(0)`
`is_unsigned`	`is_arithmetic && T(0) < T(-1)`
`is_constructible<T, T1, ...>`	a variable of type `T` could be constructed using a constructor that takes arguments of types `T1`, ..., `Tn`
`is_default_constructible`	`is_constructible` (with no constructor argument types passed)
`is_copy_constructible`	`is_constructible<T, const T&>`
`is_move_constructible`	`is_constructible<T, T&&>`
`is_assignable<T, U>`	`declval(T) = declval(U)` is a well formed expression when treated as an unevaluated operand
`is_copy_assignable`	`is_assignable<T, const T&>`

`is_move_assignable`	`is_assignable<T, T&&>`
`is_destructible`	`T` is complete, and `D<T>::~D` in `template<typename U> struct D { U u;}` isn't deleted
`is_trivially_constructible`	`is_constructible<T, T1, ...>` and the corresponding variable definition wouldn't call a non-trivial operation
`is_trivially_default_constructible`	`is_trivially_constructible<T>`
`is_trivially_copy_constructible`	`is_trivially_constructible<T, const T&>`
`is_trivially_move_constructible`	`is_trivially_constructible<T, T&&>`
`is_trivially_assignable`	`is_assignable<T, U>` and the assignment wouldn't call a non-trivial operation
`is_trivially_copy_assignable`	`is_trivially_assignable<T, const T&>`
`is_trivially_move_assignable`	`is_trivially_assignable<T, T&&>`
`is_trivially_destructible`	`is_destructible<T>` and the destructor is trivial
`is_nothrow_constructible`	`is_constructible<T, T1, ...>` and the corresponding variable definition wouldn't throw an exception
`is_nothrow_default_constructible`	`is_nothrow_constructible<T>`
`is_nothrow_copy_constructible`	`is_nothrow_constructible<T, const T&>`
`is_nothrow_move_constructible`	`is_nothrow_constructible<T, T&&>`
`is_nothrow_assignable`	`is_assignable<T, U>` and the assignment wouldn't throw an exception
`is_nothrow_copy_assignable`	`is_nothrow_assignable<T, const T&>`
`is_nothrow_move_assignable`	`is_nothrow_assignable<T, T&&>`
`is_nothrow_destructible`	`is_destructible<T>` and the destructor doesn't throw exceptions
`has_virtual_destructor`	has a virtual destructor

The definition of POD types given in the standard is:

> "A POD `struct` is a class 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). 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). A POD class is a class that is either a POD `struct` or a POD `union`."

This means that `is_pod` is equivalent to `is_trivial && is_standard_layout`.

A trivial type is a scalar type, or a trivially copyable class with a trivial default constructor (or an array of such type/class), possibly cv-qualified. It's useful to distinguish such types because objects of trivial types may be created via `malloc`, which can be an opportunity for optimization.

A trivially copyable type is a scalar type, or a trivially copyable class (or an array of such type/class), possibly cv-qualified. A trivially copyable class is defined by the following:

- no virtual functions or virtual base classes
- trivial copy & move constructors
- trivial copy & move assignment operators
- trivial destructor.

It's useful to distinguish trivially copyable types because objects of trivially copyable types can be copied with `memcpy` (which could be done as an optimization) and serialized/deserialized with `offstream::write()/ifstream::read()`.

A standard layout type is a scalar type, standard layout class, an array of such type/class, or a cv-qualified version of it. A standard layout class is defined by the following:

- no non-static data members of type non-standard-layout class (or array of such types) or reference,
- no virtual functions or virtual base classes
- the same access control for all non-static data members,
- has no non-standard-layout base classes
- either has no non-static data members in the most derived class and at most one base class with non-static data members, or has no base classes with non-static data members

- no base classes of the same type as the first non-static data member Standard layout types are meant to be compatible with C.

Note that a class can be trivial but not standard-layout, and vice versa:

```
struct Trivial          // trivial but not standard-layout
{
    int a;
private:
    int b;
};

struct StandardLayout   // standard-layout but not trivial
{
    int a;
    int b;
    ~StandardLayout();
};
```

Array-specific traits

In addition to the type property templates above, there are also two array-specific templates:

Template	**`::value` is...**
`rank`	0 if the argument isn't an array, otherwise the number of dimensions
`extent <typename T, unsigned I = 0>`	If `T` is not an array type, or if it has rank less than or equal to `I`, or if `I` is 0 and `T` has type "array of unknown bound of `U`", then 0; otherwise, the bound of the `I`'th dimension of `T`, where indexing of `I` is zero-based the alignment requirement for `T`

`rank` is used like this:

```
rank<int>::value;          // 0
rank<int[2]>::value;       // 1
rank<int[][4]>::value;     // 2
```

`extent` works as follows:

```
extent<int>::value;               // 0
extent<int[2]>::value;            // 2
extent<int[2][4]>::value;         // 2
extent<int[][4]>::value;          // 0
extent<int, 1>::value;            // 0
extent<int[2], 1>::value;         // 0
extent<int[2][4], 1>::value;      // 4
extent<int[][4], 1>::value;       // 4
```

`alignment_of` calculates the alignment requirement of a type:

```
struct A
{
    char _a;
    double _d;
};

alignment_of<int>::value;       // 4
alignment_of<double>::value;    // 8
alignment_of<A>::value;         // 8
```

Type relationships

These templates let you query what the relationship is between two types.

Template	**`::value` is `true` when...**
`is_same<T, U>`	`T` and `U` are the same, including `const` and `volatile` qualifiers

`is_base_of<Base, Derived>`	`Base` is a base class of `Derived`, or `Base` and `Derived` aren't unions and are the same class type; `const` and `volatile` qualifiers are ignored by this template
`is_convertible<From, To>`	if an rvalue of type `From` can be used as a return expression in a function returning `To`, i.e. if `From` can be implicitly converted into `To`

`is_convertible` works with reference types, `void` types, array types, and function types.

Type manipulation

The type manipulation templates allow you to add or remove `const` and/or `volatile` qualifiers, references or pointers, make signed/unsigned conversions, and remove some or all dimensions from array types. These templates contain a `typedef type` for the result type.

`const` and `volatile` modifications

Template	**`::type` is...**
`remove_const<T>`	same as `T` but without any top-level `const` qualifier
`remove_volatile<T>`	same as `T` but without any top-level `volatile` qualifier
`remove_cv<T>`	same as `T` but without any top-level `const` or `volatile` qualifiers
`add_const<T>`	`T` if `T` is a reference, function, or top-level `const`-qualified type, otherwise `T const`
`add_volatile<T>`	`T` if `T` is a reference, function, or top-level `volatile`-qualified type, otherwise `T volatile`
`add_cv<T>`	`add_const<typename add_volatile<T>::type>::type`

Keep in mind that `remove_const<const int*>::type` evaluates to `const int*` (same as `remove_const<const int* const>::type`), because the `const` isn't top level. Similarly, `remove_volatile<volatile int*>` evaluates to `volatile int*`.

References

Template	`::type` is...
`remove_reference<T>`	If `T` is a reference type, then `T` without the reference, or `T` otherwise
`add_lvalue_reference<T>`	equivalent to `remove_reference<T>::type&`
`add_rvalue_reference<T>`	equivalent to `remove_reference<T>::type&&`

Sign conversions

Template	`::type` is...
`make_signed<T>`	If `T` is a signed integer type, then `T`, otherwise the corresponding unsigned type; cv-qualifiers are preserved. For `enum` type `T`, the signed integer type with the smallest rank for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`
`make_unsigned<T>`	If `T` is an unsigned integer type, then `T`, otherwise the corresponding signed type; cv-qualifiers are preserved. For `enum` type `T`, the unsigned integer type with the smallest rank for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`

The standard defines rank for integer types as follows:

- No two signed integer types other than `char` and `signed char` (if `char` is signed) shall have the same rank, even if they have the same representation.
- The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.

- The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.
- The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.
- The rank of any standard integer type shall be greater than the rank of any extended integer type with the same size.
- The rank of `char` shall equal the rank of signed char and unsigned char.
- The rank of `bool` shall be less than the rank of all other standard integer types.
- The ranks of `char16_t`, `char32_t`, and `wchar_t` shall equal the ranks of their underlying types.
- The rank of any extended signed integer type relative to another extended signed integer type with the same size is implementation-defined, but still subject to the other rules for determining the integer conversion rank.
- For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 shall have greater rank than T3.

Array modifications

Template	**`::type` is...**
`remove_extent<T>`	If `T` is an array type, then `T` with the first dimension removed, otherwise `T`; if `T` is an array of `const`, then `const` is preserved
`remove_all_extents<T>`	if `T` is a multi-dimensional array type, then `T` with all dimensions removed, otherwise `T`

Some examples:

```
remove_extent<int>::type;               // yields int
remove_extent<int[2]>::type;            // yields int
remove_extent<int[2][3]>::type;         // yields int[3]
remove_extent<int[][3]>::type;          // yields int[3]

remove_all_extents<int[2][3]>::type;    // yields int
```

Pointer modifications

Template	::type is...
remove_pointer<T>	If T is a possibly cv-qualified pointer type, then T without the pointer, otherwise T
add_pointer<T>	remove_reference<T>::type*

Other transformations

Finally, there are several more transformations which don't fit into the categories above.

Template	::type is...
aligned_storage<Len, Align>	a POD type suitable for use as unitialized storage for any object whose size is at most Len and whose alignment is a divisor of Align. Align has a default value which is the most stringent alignment requirement for any C++ object type whose size is no greater than Len
aligned_union<Len, T1, ...>	a POD type of size $\geq$ Len suitable for use as unitialized storage for any object whose type is listed in T1, ..., Tn; the alignment of this type (which is the strictest alignment of the type arguments) can be found in the ::alignment_type member
decay<T>	Let U be remove_reference<T>. If is_array<U>::value, then remove_extent<U>::type*. If is_function<U>::value, then add_pointer<U>::type. In all other cases remove_cv<U>
enable_if<bool B, typename T = void>	If B is true then T, otherwise type isn't present. B is a a compile time boolean expression

`conditional<bool B, T, U>`	If `B` is `true` then `T`, otherwise `U`. `B` is a compile time boolean expression
`common_type<T, U>`	The common arithmetic type of arithmetic `T` and `U` which can accommodate values of either `T` or `U`
`underlying_type<T>`	underlying type of `T` (`T` must be an `enum` type)
`result_of<T>`	return type of `T`, where `T` describes a function type

`decay` The result of applying decay to a type is similar to the conversions that happen when lvalues are used as rvalues: arrays and functions are converted to pointers, and lvalues are converted to rvalues. It also removes `const` and `volatile` qualifiers to model passing by value more closely.

`enable_if` The purpose of `enable_if` is to allow a class template or a function template specialization to be included or excluded from consideration by the compiler based on the properties of the template arguments. One possible use for `enable_if` is to enable template specializations for specific types:

```
template <typename T, typename Enable = void>
struct A
{
    // ...
};

template<typename T>
struct A<T, typename enable_if<is_integral<T>::value>::type>
{
    // ...
};

template<typename T>
struct A<T, typename enable_if<is_floating_point<T>::value>::type>
{
    // ...
```

```
};

A<string> a;    // instantiates generic template
A<int> b;       // instantiates integral type specialization
A<float> c;     // instantiates floating point type specialization
```

When the compiler chooses what to instantiate for `A<int>`, it will consider `enable_if<is_integral<T>::value>::type`, which is `int`, and `enable_if <is_floating_point<T>::value>::type` which isn't a valid expression because the condition is `false` and the `type` typedef isn't present. Based on that, it will select the first specialization.

When using `enable_if` for function specialization, there are two options: either using `enable_if` in the return type, or in a dummy argument. The dummy argument is the only option for constructors (as they don't have a return type). The SFINAE rule enables the function specialization to work. For example, consider this code:

```
long square(int n)
{
    return n * n;
}

template<typename T>
typename enable_if<is_floating_point<T>::value, T>::type
square(const T& n)
{
    return n * n;
}

square(10);             // calls square(int)
square(10.0);           // calls square(double)
square(string("A"));    // doesn't compile,
                        // "can't convert string to int"
```

The compiler has to consider both versions of `square` when choosing the overload to call. In case of an `int` argument the return type of the templated function is invalid, so the long `square(int)` version is selected (and there is no compilation error). In the case of a `double` argument, the template instantiation `double square(const double&)` can

be chosen. In the case of a `string` argument none of the overloads are suitable, which results in a compiler error.

If all overloads have the same return type (or no return type), then you can use a dummy argument instead:

```
double f(double n)
{
    // ...
}

template<typename T>
double f(T n,
    typename enable_if<is_integral<T>::value>::type* dummy = nullptr)
{
    // ...
}

f(10.0);                // calls f(double)
f(1);                   // calls f(int)
f(string("A"));         // doesn't compile,
                        // "can't convert string to double"
```

In the template definition above, the function gets a second `void*` parameter (with a default value of `nullptr`) when the first parameter is an integral type. If the type is not integral, then the function's parameter list is malformed, and the compiler will disregard it in choosing an overload to call.

`common_type` `common_type` doesn't support references or pointers but it's easy to add a specialization for that purpose if you need it:

```
template <typename A, typename B>
struct common_type<A*, B*> {
    typedef typename common_type<A, B>::type* type;
};
```

`result_of` The template argument must be of the form `F(T1, T2, ..., Tn)`, where `F` is a callable type. The return type is determined according to the following rules:

- if `F` is a pointer to a function of the form `R(*)(U1, U2, ..., Un)`, the return type is `R`
- if `F` is a reference to a function of the form `R(&)(U1, U2, ..., Un)`, the return type is `R`
- if `F` is a pointer to a member function `R(U1::*)(U2, ..., Un)` the return type is `R`
- if `F` is a pointer to a data member of the form `R U1::*`, the return type is `R`
- if `F` is a class containing a member `typedef result_type`, the return type is `F::result_type`
- if the template argument is of the form `F()`, the return type is `void`
- if `F` is a class containing a member template named `result` the return type is `F::result<T1, T2, ..., Tn>::type`

All other cases result in a compiler error.

Additional template aliases in C++14

VS2013 provides a set of additional template aliases for the type traits listed in the previous two sections ("Type manipulation" and "Other transformations"). The purpose of these aliases is to make the code more readable.

If you use the type traits to construct non-trivial expressions, they quickly become unwieldy. Here is an example from the C++14 proposal:

```
template<typename T> using reference_t =
    typename conditional<is_reference<T>::value, T,
        typename add_lvalue_reference<T>::type>::type;
```

Using the `::type` member of type trait templates is the predominant case, so in C++14 you can instead write a more compact equivalent statement:

```
template<typename T> using reference_t
    = conditional_t<is_reference<T>::value, T,
        add_lvalue_reference_t<T>>;
```

Note the `_t` suffix in the alias names `conditional_t` and `add_lvalue_reference_t`. The aliases are defined like this:

```
template <typename T> using add_cv_t = typename add_cv<T>::type;
```

Non-standard behavior and bugs in Visual Studio 2013

`static_assert` in a class definition

Sometimes `static_assert` fails to work in the class definition:

```
template <typename T>
struct UnsignedMangler
{
    static_assert(is_integral<T>::value, "T must be integral"); // OK
    static_assert(T(-1) > T(0), "type is not unsigned"); // error

    UnsignedMangler()
    {
        static_assert(T(-1) > T(0), "type is not unsigned"); // OK
    }
};
```

In this example, even though the first assert works fine, the second one always fails – even without instantiating the template.

On the other hand, the same `static_assert` placed into the constructor works as expected. This is a workaround in case you encounter a similar issue.

Compiling with code analysis may wrongly trigger a `static_assert`

If you attempt to compile something like the following with the `/analyze` flag, the static assert will be triggered incorrectly (`Val` *is* actually zero):

```
template <typename T>
struct A
{
    T _val;
    enum
```

```
    {
        Val = 0
    };
    static_assert(Val == 0, "Val should be 0");
};
```

When code analysis is enabled, Visual Studio compiles the code twice. The second "analysis" pass is what triggers the error.

`is_function` fails to detect a static member function

The following incorrectly triggers the static assert in Visual Studio:

```
struct A
{
    static void f() {}
};

static_assert(is_function<decltype(A::f)>::value, "not a function");
```

This is a bug in `is_function`.

`remove_pointer` fails with a `const` pointer to a function

`remove_pointer` doesn't work correctly when passed a `const` pointer to a function, but works with a non-`const` pointer to function:

```
// yields void(*)(long) instead of void(long)
remove_pointer<void(*const)(long)>::type;

// OK, yields void(long)
remove_pointer<void(*)(long)>::type;
```

`is_pod` provides wrong results for types with user-defined constructors

Even though C++11 allows POD types to have user-defined constructors (as long as there is a trivial default constructor), Visual Studio's `is_pod` fails on such types:

```
struct A
{
    A() = default;

    A(int a) : _a(a)
    {}

    int _a;
};

static_assert(std::is_trivial<A>::value,
    "A has to be trivial");                                  // OK
static_assert(std::is_pod<A>::value, "A has to be a POD");   // error
```

Neither of the static asserts in this example should trigger, but the second one does.

`is_trivially_copyable` fails on arrays of scalar types

Arrays of scalar types are defined to be trivially copyable by the standard. However, `is_trivially_copyable` in Visual Studio disagrees:

```
const int n = 100;
static_assert(is_trivially_copyable<int[n]>::value,
    "shouldn't trigger");        // error
static_assert(is_trivially_copyable<const int[n]>::value,
    "shouldn't trigger");        // error
```

Construction-related type traits fail on abstract types

In this example, all of the static asserts should trigger, but none do:

```
struct A
{
    A() = default;
    A(int) {}
    virtual void f() = 0;
};

// none of the asserts trigger
static_assert(is_constructible<A>::value, "oops");
static_assert(is_constructible<A, int>::value, "oops");
static_assert(is_default_constructible<A>::value, "oops");
static_assert(is_copy_constructible<A>::value, "oops");
static_assert(is_move_constructible<A>::value, "oops");
```

Visual Studio fails to take into account that `A` is an abstract type.

`is_assignable` provides wrong results in some cases

`is_assignable` incorrectly reports that rvalues can be assigned to in some situations:

```
// outputs true instead of false
cout << is_assignable<double, int>::value << endl;

// outputs true instead of false
cout << is_assignable<double, int&>::value << endl;
```

`is_destructible` doesn't work

`is_destructible` doesn't seem to do much of anything. Whether you mark a destructor with `delete` or make it private, the trait always reports that the type is destructible (which is of course different from what the standard prescribes).

`is_convertible` gives wrong results

`is_convertible` fails in situations where protected or private inheritance is involved:

```
class A {};

class B : protected A {};

class C : private B {};

is_convertible<B, A>::value;  // yields true but should be false
is_convertible<C, A>::value;  // yields true but should be false
```

`alignment_of` implementation deviates from the standard

`alignment_of<T>` is defined as `alignof(T)` in the standard but `alignof` isn't yet supported by Visual Studio, so its implementation is different.

`enable_if` combined with two type parameters compile error

If you use `enable_if` in the return type of a function template which has two type parameters, you may get a compilation error:

```
template <typename T, typename U>
typename enable_if<is_integral<T>::value &&
    is_integral<U>::value, void>::type
f(T t, U u) {}

template <typename T, typename U>
typename enable_if<is_integral<T>::value &&
    is_floating_point<U>::value, void>::type
f(T t, U u) {}  // compile error - function template already defined
```

This occurs even without instantiating either of these templates. The workaround is to swap the types in the `enable_if` conditional expression:

```
template <typename T, typename U>
typename enable_if<is_integral<U>::value &&
    is_integral<T>::value, void>::type
f(T t, U u) {}
```

```
template <typename T, typename U>
typename enable_if<is_integral<U>::value &&
    is_floating_point<T>::value, void>::type
f(T t, U u) {}
```

New STL Containers

The C++11 STL has been expanded to include several new containers. The first two are `forward_list` and `array`. There is also a group of hash table based containers: `unordered_map`, `unordered_multimap`, `unordered_set` and `unordered_multiset`.

`forward_list`

`forward_list` provides a single-linked list with constant time insert and erase operations. The primary goal in defining this container was to have no overhead over a hand-coded list, so some operations have been intentionally omitted, and the interface doesn't fully follow the STL conventions.

> **Note:**
>
> `forward_list` is defined in the `<forward_list>` header.

It only supports forward iterators, and doesn't have `size()`, `rbegin()/rend()` or `push_back()` methods. In contrast to other containers, it also requires that ranges which are modified, such as ranges supplied to `erase` and `splice`, to be open at the beginning (i.e. `(first, last)` instead of `[first, last)`).

This is because modifying any list requires access to the element preceding the start of the range to update the link, and `forward_list` doesn't have a constant time method of accessing the preceding element. For the same reason, `forward_list` has `before_begin`, returning an iterator preceding `*begin()`.

Instead of insert, `erase`, `emplace` and `splice` operations, `forward_list` has `insert_after`, `erase_after`, `emplace_after` and `splice_after`:

```
forward_list<int> list;

auto it1 = list.insert_after(list.before_begin(), 10);
for (int i = 20; i < 50; i += 10)
    it1 = list.insert_after(it1, i);
```

```
// list now contains 10, 20, 30, 40

list.insert_after(list.begin(), 15);            // 10, 15, 20, 30, 40

auto it2 = find(list.cbegin(), list.cend(), 15);
list.erase_after(it2);                          // 10, 15, 30, 40
list.erase_after(list.before_begin());          // 15, 30, 40

forward_list<int> list2;
auto it3 = list2.before_begin();
for (int i = 100; i < 400; i += 100)
    it3 = list2.insert_after(it3, i);

// list2 contains 100, 200, 300

list.splice_after(list.begin(), list2);         // 15, 100, 200, 300,
                                                // 30, 40
```

`emplace_after` takes arguments for the element's constructor and constructs an element in place, so no copying or moving is involved:

```
forward_list<JetPlane> planes;
planes.emplace_after(planes.before_begin(), "Boeing 737");
```

The iterator invalidation rules for `forward_list` are the same as those for `list`. Inserting an element doesn't invalidate iterators, and erasing only invalidates iterators to the erased elements.

array

`std::array` is a container similar to `vector`, however it has a fixed size. The advantage of that is that its storage can be allocated statically instead of requiring dynamic allocation like `vector`, which brings its performance closer to built-in C-style arrays.

> **Note:**
>
> `array` is defined in the `<array>` header.

Like the `vector`, the elements of an array are stored in contiguous memory locations. It doesn't have any memory overhead over a built-in array. It also doesn't get converted into a pointer unless explicitly requested. It doesn't allow derived-to-base conversions, which have the potential to cause problems.

With the introduction of this container in addition to `vector`, the reasons to use a C-style array are becoming really few and far between.

Because of the fixed size, the template requires two arguments: the type of the elements and their number:

```
array<int, 10> arr;
```

> **Note:**
>
> The size of an `array` is allowed to be zero.

`array` is a good choice of container when you know that you're going to have a fixed number of objects. For example, planes have a fixed number of engines, so you could use an array to store `Engine` objects in a `Plane` template:

```
template<size_t engine_count>
class Plane
{
    typedef array<Engine, engine_count> Engines;
    Engines _engines;
public:
    const Engines& engines() const
    {
        return _engines;
    }
};

Plane<4> boeing747;
boeing747.engines().size();     // 4
```

As it uses statically allocated memory, `array` doesn't support custom allocators.

Element operations

`array` supports random access iterators:

```
array<int, 5> arr = { 1, 2, 3, 4, 5 };
cout << "3rd element: " << *(arr.begin() + 2) << endl;
cout << "Last element: " << *(--arr.end()) << endl;
```

Elements can be accessed using `operator[]`, `front()`, `back()` and `at()`:

```
for (int i = 0; i < 5; ++i)     // 1, 2, 3, 4, 5
    arr[i] = i * 10;

arr.at(2) = 100;
arr.front() = arr.back();
```

Container operations

Arrays can be assigned and swapped:

```
array<char, 6> temp = { 'h', 'e', 'l', 'l', 'o', 0 };

array<char, 6> hello = temp;
array<char, 6> world = { 'w', 'o', 'r', 'l', 'd', 0 };

hello.swap(world);

cout << world.data() << " " << hello.data() << endl;
// outputs "hello world"
```

The swap operation takes linear time and may throw exceptions, in contrast to the `vector` swap which takes constant time and doesn't throw.

`data()` provides access to the underlying built-in array:

```
char arr[6];
array<char, 6> arr2 = { 'h', 'e', 'l', 'l', 'o', 0 };

strcpy(arr, arr2.data());

cout << arr << endl;    // outputs "hello"
```

> **Note:**
>
> The result of calling `data()` on a zero sized `array` is unspecified.

You can fill the whole array with a value/object using `fill()`:

```
array<int, 100> arr;
arr.fill(1);

array<string, 100> strs;
strs.fill("<default>");
```

Initialization

Array is an aggregate type, so it can be initialized with an initializer list:

```
array<int, 5> arr = {1, 2, 3, 4, 5};
```

> **Note:**
>
> An aggregate type has the following properties: only default constructors, all its non-static members are public, no base classes, no virtual functions.

The rules for initializing elements of an `array` are the same as for built-in arrays.

If an `array` stores elements of a built-in type (e.g. `int`), then the elements will be default-initialized in the absence of an initializer list. If the `array` is in static storage, the elements will be set to zeros, otherwise they'll have random values.

The situation is different if you provide an initializer list that is shorter than the size of the `array`. In this case, the remaining objects will be *value initialized*:

```
array<int, 4> arr = { 1, 2 };   // arr contains 1, 2, 0, 0

// planes[2] and planes[3] are initialized with the default
// constructor
array<JetPlane, 4> planes =
    {JetPlane("Boeing 737"), JetPlane("Boeing 757")};

struct A
{
    int _n;
    double _d;
};

// arr2[0]._n is 10 and arr2[0]._d is 10.0, while
// arr2[1]._n and arr2[1]._d are set to zeros
array<A, 2> arr2 = { {10, 10.0} };
```

An array can't be initialized with an iterator range.

Array as tuple

Because of its fixed size, array is conceptually similar to tuple, and you can sometimes treat it as a tuple. For example, you can use get to access elements:

```
array<int, 5> arr = {1, 2, 3, 4, 5};
get<1>(arr);    // returns the second element
```

You can also use tuple_element and tuple_size templates to query array properties:

```
array<double, 5> arr = {1.1, 2.2, 3.3, 4.4, 5.5};

// elem_count will be 5
size_t elem_count = tuple_size<decltype(arr)>::value;

// this will be true
is_floating_point<tuple_element<0, decltype(arr)>::type>::value;
```

Hash tables

Hash tables are counterparts to `map`, `multimap`, `set` and `multiset`. They do not order their elements, which allows them to provide faster insertion, deletion and lookup compared to their ordered counterparts. The interface is similar between both types of containers for the most part. I'll highlight the differences in the following sections.

In order to understand the behavior and the interface of the unordered containers, it's useful to have a look at how hash tables are implemented. The hash function partitions a sequence into ordered sub-sequences called *buckets*. Each key is passed through a hash function, which produces the bucket id for that key.

When two different keys result in the same bucket id, it's a *hash collision*. For a container, a collision means that some bucket is going to contain more than one item.

> **Note:**
>
> A *perfect hash* has no collisions.

In Visual Studio, all unordered containers inherit from the hash table class which combines a `list` of elements with a `vector` containing pairs of iterators into the `list`, where the iterators delimit bucket boundaries.

The lookup consists of

- calculating the hash value (i.e. the bucket index) for a key
- looking up the bucket boundaries in the `vector` of `list` iterators
- iterating through the section of the list corresponding to the bucket in order to get the element with the matching key

`unordered_map`

This container is similar to `map`. The keys have to be unique, it provides `operator[]` and so on. In a simple case, the template parameters specify the key and value types:

```
unordered_map<string, Part> parts_register;
parts_register["screw"] = Part();
```

> **Note:**
>
> `unordered_map` is defined in the `<unordered_map>` header.

Additionally, you can specify a custom hash function, comparison function and an allocator:

```
template<class Key,
    class Ty,
    class Hash = std::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<std::pair<const Key, Ty> > >
    class unordered_map;
```

The hash and comparison functions can be retrieved with `hash_function()` and `key_eq()` member functions. I'll talk about custom hash functions in more detail below.

Lookup, insertion and removal of an element is constant time when all buckets contain approximately the same number of elements. In the worst case, when all elements are in one bucket, performance becomes linear.

Inserting an element doesn't invalidate any iterators. Removing an element only invalidates iterators pointing to that element.

The standard specifies that `unordered_map` supports forward iterators (i.e. not reverse iterators), however Visual Studio provides reverse iterators and associated functions (`rbegin()`, `rend()` etc.). `lower_bound()` and `upper_bound()` aren't included in the standard, but are also provided in the Microsoft implementation.

Similarly to map, `equal_range()` is present, as well as `find()` and `count()`. `equal_range()` returns (`map.end()`, `map.end()`) when no elements are found.

For the purposes of comparing containers, only == and != are available. These are based on comparing elements rather than their ordering in the container. As a result, the performance is O(n) on average and $O(n^2)$ in the worst case.

The differences from map are related to the implementation of the container as a hash table. There are several functions that deal with buckets:

```
unordered_map<string, Part>::size_type bucket =
    parts_register.bucket("screw");
cout << "Bucket size: " << parts_register.bucket_size(bucket) << endl
     << "Bucket count: " << parts_register.bucket_count() << endl
     << "Max bucket count: " << parts_register.max_bucket_count()
     << endl;
```

`local_iterator` and `const_local_iterator` allow you to iterate over the elements in a particular bucket. You can get iterators into a bucket by passing the bucket id to `begin()`/`end()`:

```
for_each(parts_register.begin(bucket), parts_register.end(bucket),
    [](const pair<string, Part>& p) { cout << p.first << endl; });
```

The `const` versions of `begin(size_type bucket)` and `end(size_type bucket)` return `const_local_iterators`.

As more elements are inserted, the map may need to add more buckets and rebuild the hash. Using `max_load_factor()`, you can get or set the maximum number of elements per bucket:

```
parts_register.max_load_factor();
parts_register.max_load_factor(0.5);
```

`load_factor()` returns the average number of elements per bucket:

```
parts_register.load_factor();
```

`load_factor` can be higher than `max_load_factor` because `load_factor` won't come down until more elements are inserted (which can trigger a rehash) or the container is explicitly rehashed with `rehash()`. `rehash()` allows you to set the minimum number of buckets, and also triggers a rebuild of the hash table:

```
parts_register.rehash(100);
```

Implicit or explicit rehashing invalidates iterators.

The minimum number of buckets can also be specified during construction:

```
unordered_map<string, Part> parts_register(10);
cout << "Initial buckets: "
     << parts_register.bucket_count() << endl; // outputs 16
```

Instead of specifying the number of buckets, you can specify the expected number of elements with `reserve()`:

```
known_size.reserve(100);
```

This makes the container choose a number of buckets such that the expected number of elements will result in a load factor less than `max_load_factor`.

Custom hash

Hash functions are provided by default for built-in types, `enums` (as of C++14 but available in Visual Studio since the 2012 version), `strings`, `bitset`, smart pointers and several other classes. If you need to use some other type as a key, you'll need to implement your own hash function. You can do that in two ways. One is to provide a custom functor:

```
struct HashInt
{
    size_t operator()(int n) const
    {
        return n % 10;
    }
};
unordered_map<int, int, HashInt> custom;
```

The other is to specialize `hash<T>` for your type (note that there is no need to specify the hash type explicitly in this case):

```
template<>
struct std::hash<Part>
{
    size_t operator()(const Part& p) const
    { /* ... */ }
};
unordered_map<Part, int> parts_index;    // will use custom hash
                                         // for Part
```

If you have a composite key, a simple solution for building a hash function with acceptable statistical properties is using Boost's `hash_combine` function. For example, a `pair` could be hashed like this:

```
template <class T>
void hash_combine(size_t &res, const T& t)    // this is borrowed
{                                             // from Boost
    hash<T> hash_func;
    res ^= hash_func(t) + 0x9e3779b9 + (res << 6) + (res >> 2);
}

template<typename T, typename U>
struct std::hash<pair<T, U>>
{
    size_t operator()(const pair<T, U>& p) const
    {
        size_t res = 0;
        ::hash_combine(res, p.first);
        ::hash_combine(res, p.second);
        return res;
    }
};
```

(The order of the calls matters when using `hash_combine`.)

> **Note:**
>
> The purpose of the magic constant in `hash_combine` is to avoid mapping all zeros to all zeros, and to reduce collisions. It should be a random string of 0 and 1. One technique to get such a string is to take the binary representation of an irrational number. This particular number happens to be the reciprocal of the golden ratio (232/(1 + sqrt(5)).

This could easily be extended to tuples, or implemented for your own class.

When implementing your own hash function, you need to make sure that equal objects produce the same hash value.

`unordered_multimap`, `unordered_set`, `unordered_multiset`

These containers are also mostly similar to their ordered counterparts. Like `unordered_map`, they provide hash table specific operations, and have similar limitations, so I won't discuss them in more detail.

`unordered_multimap` is defined in `<unordered_map>`, while `unordered_set` and `unordered_multiset` are defined in `<unordered_multiset>`.

Other STL Improvements

In addition to introducing new containers, the C++ committee has also made a number of improvements to the existing containers, and introduced some new algorithms. Visual Studio 2013 supports most of these changes.

Container improvements

Support for move semantics

The containers now support move semantics, both by implementing move constructors/move assignment operators, and by providing move versions of operations such as `push_back`. So, for example, you could insert temporary objects and return by value without incurring the performance overhead of copying:

```
vector<string> preload()
{
    vector<string> &strings;
    for (int i = 0; i < 100; ++i)
        strings.push_back("Some long string");
    return strings;
}

vector<string> strings = preload();
```

> **Note:**
>
> The compiler could actually use the NRVO (named return value optimization) in this particular case, but it's not always applicable, unlike the move assignment.

The return value will be move constructed, and the strings will be moved into the vector instead of being copied.

Better `const_iterator` support

The containers now have better `const_iterator` support by providing `cbegin()`/`cend()` and `crbegin()`/`crend()` where possible. The new methods are useful when you explicitly want to get a `const_iterator`, e.g. in combination with the `auto` keyword:

```
template<typename Cont>
void print(const Cont& cont)
{
    for (auto it = cont.cbegin(); it != cont.cend(); ++it)
        cout << *it << endl;
}
```

In addition, C++14 introduces free versions of `cbegin()`, `cend()`, `crbegin()` and `crend()` which are implemented by VS2013:

```
for (auto it = crbegin(cont); it != crend(cont); ++it)
    cout << *it << endl;
```

`emplace` methods

`emplace()`/`emplace_back()`/`emplace_front()` are intended to construct elements in place instead of moving/copying an object into the container, so they take arguments which they pass on to the element's constructor:

```
vector<pair<int, string>> pairs;
pairs.emplace_back(10, "ten");
pairs.emplace(pairs.begin(), 0, "zero");

unordered_map<int, string> hash;
hash.emplace(10, string("ten"));
```

> **Note:**
>
> `emplace_front()` is only available for deque.

Associative containers provide `emplace_hint()` in addition to `emplace()`, which takes an iterator as a hint:

```
unordered_map<int, string> hash;
// hint is ignored
hash.emplace_hint(hash.begin(), 0, "zero");

map<string, string> m;
m.emplace_hint(m.begin(), "B", "Y");
```

The unordered versions of the containers ignore the hint, as permitted by the standard.

Reducing container capacity

`vector`, `deque` and `string` have acquired a `shrink_to_fit()` operation, which reduces their `capacity()` to `size()`:

```
vector<int> v;
v.shrink_to_fit();

deque<string> d;
d.shrink_to_fit();

string s("abc");
s.shrink_to_fit();
```

The standard defines `shrink_to_fit` as a non-binding request, so you can't really rely on it to perform any particular operation in a cross-platform code base.

Immutable set elements

`set` and `multiset` elements are now required to be immutable, so `begin()`/`end()`, `find()`, `operator[]` and so on return `const_iterators`:

```
set<int> st;
st.insert(1);
*st.begin() = 3;    // compile error
```

String improvements

The <string> header now provides to_string() and to_wstring() functions for various numeric types:

```
cout << "int: " << to_string(10) << endl
     << "long: " << to_string((long long)10L) << endl
     << "float: " << to_string(long double(10.0f)) << endl
     << "double: " << to_string(10.0) << endl;
```

Additionally, a number of functions have been added for converting strings into numbers of different types: stoi(), stol(), stoll() (long long), stoul(), stoull() (unsigned long long), stof(), stod(), stold() (long double).

Miscellaneous

vectors now have a vector.data() function which returns a pointer to the first element, similarly to string, and strings now have front() and back() analogous to those of vector.

map has obtained at(), which is similar to operator[] but throws an out_of_range exception if the key is not present instead of inserting a new element.

Iterator improvements

begin() and end() have become available as free functions:

```
for (auto it = begin(v); it != end(v); ++it)
    cout << *it << endl;
```

Additionally, as of C++14 (and VS2013), rbegin() and rend() are also available:

```
for (auto it = rbegin(v); it != rend(v); ++it)
    cout << *it << endl;
```

> **Note:**
>
> These functions are available by including <iterator> or any of the container headers, or <regex> or <string>.

These functions work on all STL containers, arrays, and can be specialized for classes which don't provide `begin()`/`end()` as members but allow iteration.

There are two versions of each function: the ones that take a `const` container reference return a `const_iterator`, and the ones that take a non-`const` reference return an `iterator`.

Iterator adapters to support move operations

`move_iterator` is an adapter which behaves like the underlying iterator, with the only difference that it returns an rvalue upon dereferencing, which means that the value it points to can be moved from. For example, you could move elements of a `vector` into two sub-ranges:

```
vector<string> v, v1, v2;
// ... insert elements into v ...
typedef vector<string>::iterator Iter;
Iter halfway = v.begin() + v.size() / 2;
v1.insert(v1.end(), move_iterator<Iter>(v.begin()),
    move_iterator<Iter>(halfway));
v2.insert(v2.end(), move_iterator<Iter>(halfway),
    move_iterator<Iter>(v.end()));
```

Alternatively, you could make this code a bit simpler by using `make_move_iterator()`:

```
vector<string> v, v1, v2;
// ... insert elements into v ...
auto halfway = v.begin() + v.size() / 2;
v1.insert(v1.end(), make_move_iterator(v.begin()),
    make_move_iterator(halfway));
v2.insert(v2.end(), make_move_iterator(halfway),
    make_move_iterator(v.end()));
```

After these operations, `v` is left with empty elements, and `v1` and `v2` contain one half of its elements each.

`prev()`/`next()` functions

`next()` and `prev()` functions return a new iterator that's `n` elements ahead or behind:

```
vector<int> v;
for (int i = 0; i < 4; ++i)
    v.push_back(i + 1);

auto it = v.begin();
auto last = next(it, 3);        // last points to 4
auto second = prev(last, 2);    // second points to 2
```

C++14 specializations for operator functors in `functional`

C++14 introduces extra specializations for the operator functors in the `functional` header.

As an example, instead of writing this:

```
vector<double> v;
sort(v.begin(), v.end(), greater<double>());
```

you can now use `greater` without specifying element type (but note the empty angle brackets):

```
sort(v.begin(), v.end(), greater<>());
```

The second version is not only shorter, but also reduces duplication, and thus the potential for errors to creep in (e.g. if you change the type of elements in the container).

There are a couple more benefits these specializations enable. Consider the implementation of `greater<>`:

```
template <typename T = void> struct greater;
// ...
template <> struct greater<void>
{
```

```
    template <typename T, typename U>
    auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) > std::forward<U>(u));
};
```

As you can see, `greater<>` is a perfectly forwarding function which also allows its arguments to have different types, unlike the previously existing `greater<T>`. Independent argument types allow more flexibility, and more efficient code generation in those cases where the compiler can avoid type conversions, or is able to use move operations when one of the arguments of the functor is an rvalue.

Here is the full list of functors with new specializations:

Category	Specialized templates
Arithmetic operations	`plus`, `minus`, `multiplies`, `divides`, `modulus`, `negate`
Comparisons	`equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`
Logical operators	`logical_and`, `logical_or`, `logical_not`
Bitwise operations	`bit_and`, `bit_or`, `bit_xor`, `bit_not`

Note that `bit_not` is another C++14 addition.

New algorithms

The C++11 standard has introduced 23 new algorithms, and they have been implemented by Visual Studio. Additionally, algorithms now also support move operations and lambdas.

`bool all_of(Iter first, Iter last, Pred pred)`

`true` if all the values in `[first, last)` satisfy the predicate (or the range is empty), `false` otherwise.

`bool any_of(Iter first, Iter last, Pred pred)`

`true` if at least one of the values in `[first, last)` satisfies the predicate, `false` otherwise (or if the range is empty)

`bool none_of(Iter first, Iter last, Pred pred)`

`true` if no values in `[first, last)` satisfy the predicate (or if the range is empty), `false` otherwise:

```
vector<int> v;
// v contains 1, 2, 3, 4
all_of(v.begin(), v.end(), [](int n) { return n < 10; });      // true
any_of(v.begin(), v.end(), [](int n) { return n % 2 == 0; });  // true
none_of(v.begin(), v.end(), [](int n) { return n > 10; });     // true
```

`Iter find_if_not(Iter first, Iter last, Pred pred)`

Returns the first iterator `i` in the range where `pred(*i) == false` or `last` if no such iterator found:

```
// find an element that's not equal to 3
auto it = find_if_not(v.begin(), v.end(), [](int n) { return n == 3; });
```

`OutIter copy_if(InIter first, InIter last, OutIter result, Pred pred)`

Copies all elements in `[first, last)` which satisfy a predicate into a range starting from result (the opposite of `remove_copy_if`).

`OutIter copy_n(InIter first, Size n, OutIter result)`

Copies `n` elements starting from `first` into a range starting from `result`

uninitialized_copy_n(InIter first, Size n, OutIter result)

Invokes uninitialized_copy for n elements:

```
vector<int> v_even;
// copy only even numbers
copy_if(v.begin(), v.end(), back_inserter(v_even),
    [](int n) { return n % 2 == 0; });

vector<int> sub_range;
// copy the first 3 elements from v
copy_n(v.begin(), 3, back_inserter(sub_range));

vector<int> sub_range2(3);
// copy the first 3 elements from v using uninitialized_copy
uninitialized_copy_n(v.begin(), 3, sub_range2.begin());
```

OutIter move(InIter first, InIter last, OutIter result)

Moves elements from [first, last) into a range starting from result.

OutIter move_backward(InIter first, InIter last, OutIter result)

Moves elements in the range [first, last) into the range [result - (last - first), result) starting from last - 1 and proceeding to first:

```
vector<string> v, v2, v3;
// v contains A, B, C

move(v.begin(), v.end(), back_inserter(v2));
// v2 now contains A, B, C

v3.resize(v.size());
move_backward(v2.begin(), v2.end(), v3.end());
// v3 now contains A, B, C
```

`is_partitioned(InIter first, InIter last, Pred pred)`

`true` if `[first, last)` is empty or if `[first, last)` is partitioned by `pred`, i.e. if all elements that satisfy `pred` appear before those that don't.

`pair<OutIter1, OutIter2>`

`partition_copy(InIter first, InIter last, OutIter1 out_true, OutIter2 out_false, Pred pred)`

Copies elements that satisfy `pred` from `[first, last)` into the range starting with `out_true`, and other elements into the range starting with `out_false`.

`Iter partition_point(Iter first, Iter last, Pred pred)`

Returns an iterator to the first element in `[first, last)` that doesn't satisfy `pred`:

```
// v contains 1, 3, 5, 2, 4, 6

// is v partitioned into odd and even numbers?
is_partitioned(v.begin(), v.end(),
    [](int n) { return n % 2 != 0; });  // true

// what's the first element of the 2nd partition?
*partition_point(v.begin(), v.end(),
    [](int n) { return n % 2 != 0; });   // 2

// v contains 1, 2, 3, 4, 5, 6
vector<int> v_odd, v_even;
partition_copy(v.begin(), v.end(),
    back_inserter(v_odd), back_inserter(v_even),
    [](int n) { return n % 2 != 0; });
// v_odd now contains 1, 3, 5; v_even now contains 2, 4, 6
```

`RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last)`

`RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last, Compare comp)`

Copies sorted elements from `[first, last)` into the result range (in terms of `comp` If supplied); the number of elements copied is determined by the size of the smaller of input and result ranges.

`bool is_sorted(Iter first, Iter last)`

`bool is_sorted(Iter first, Iter last, Compare comp)`

`true` if `[first, last)` is sorted (in terms of `comp` if supplied), `false` otherwise.

`Iter is_sorted_until(Iter first, Iter last)`

`Iter is_sorted_until(Iter first, Iter last, Compare comp)`

Returns the last iterator `i` in `[first, last]` for which the range `[first, i)` is sorted (in terms of `comp` if supplied):

```
vector<int> v;
// v contains 1, 2, 3, 3, 2, 1

vector<int> sorted(5);
partial_sort_copy(v.begin(), v.end(), sorted.begin(), sorted.end());
// sorted now contains 1, 1, 2, 2, 3;
// the last "3" didn't make it in because sorted only has space for
// 5 elements

is_sorted(v.begin(), v.end());       // false
is_sorted_until(v.begin(), v.end()); // v.begin() + 4
```

bool is_heap(Iter first, Iter last)

bool is_heap(Iter first, Iter last, Compare comp)

`true` if `[first, last)` is a heap (in terms of `comp` if supplied), i.e. the first element is the largest.

Iter is_heap_until(Iter first, Iter last)

Iter is_heap_until(Iter first, Iter last, Compare comp)

Returns the last iterator `i` in `[first, last]` for which the range `[first, i)` is a heap (in terms of `comp` if supplied):

```
vector<int> v;
// v contains 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

make_heap(v.begin(), v.end());
// v now contains 10, 9, 7, 8, 5, 6, 3, 1, 4, 2

is_heap(v.begin(), v.end()) == true);   // true

v[7] = 100;
is_heap_until(v.begin(), v.end()); // v.end() - 3
```

pair<const T&, const T&> minmax(const T& a, const T& b)

pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp)

Returns `(b, a)` pair if `b < a` (in terms of `comp` if supplied), and `(a, b)` pair otherwise.

pair<const T&, const T&> minmax(initializer_list<T> lst)

pair<const T&, const T&> minmax(initializer_list<T> lst, Compare comp)

Returns a pair containing min and max elements in the list (in terms of `comp` if supplied):

```
minmax(3, 2).first;   // 2
minmax({3, 2, 1});    // returns a (1, 3) pair
```

const T& min(initializer_list<T> lst)

const T& min(initializer_list<T> lst, Compare comp)

Returns the smallest element in the list (in terms of comp if supplied).

const T& max(initializer_list<T> lst)

const T& max(initializer_list<T> lst, Compare comp)

Returns the largest element in the list (in terms of comp if supplied).

pair<Iter, Iter> minmax_element(Iter first, Iter last)

pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp)

Returns the *first* iterator in [first, last) pointing to the smallest element, and the *last* iterator pointing to the largest element (in terms of comp if supplied), or make_pair(first, first) if the range is empty:

```
// v contains 10, 9, 7, 8, 5, 6, 3, 1, 4, 2, 10
auto res = minmax_element(begin(v), end(v));
// res.first points to v[7]
// res.second points to v[10]
```

void iota(Iter first, Iter last, T value)

Creates a range of sequentially increasing values; assigns *i = value to each element in [first, last) and increments value as if by ++value:

```
vector<int> v(10);
iota(v.begin(), v.end(), 0);
// v now contains 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Note:

All the algorithms listed here with the exception of iota are defined in `<algorithm>`. `iota` is defined in `<numeric>`.

Random Number Facility

The random number generation functionality has been greatly expanded in C++11. Instead of just `srand()` and `rand()` you now have the ability to generate random numbers of varying quality, within a defined range, and with particular statistical properties.

The new functionality consists of two categories of classes: engines and distributions. In the simplest case, you can just use an engine to generate random numbers:

```
default_random_engine eng;
eng.seed(default_random_engine::result_type(time(NULL)));
for (int i = 0; i < 10; ++i)
    cout << eng() << endl;
```

> **Note:**
>
> Random number generation functionality is defined in `<random>`.

The random number engine has to be seeded. If you want numbers in a particular range, you can use the modulo operator:

```
// produce numbers in the range [1, 100]
for (int i = 0; i < 10; ++i)
     cout << (1 + eng() % 100) << endl;
```

Alternatively, you can use a uniform distribution to enforce a range:

```
uniform_int_distribution<int> distr(1, 100);
for (int i = 0; i < 10; ++i)
     cout << distr(eng) << endl;
```

Distributions not only produce numbers in a particular range, they also produce a sequence with particular statistical properties, as you'll see below.

Engines

Several engines are provided by the library:

- `mersenne_twister_engine`
- `linear_congruential_engine`
- `subtract_with_carry_engine`
- `random_device`

All engines, with the possible exception of `random_device`, produce (mostly) uniformly distributed pseudo-random numbers based on some algorithm. The sequence eventually wraps around. They have to be seeded, otherwise they produce the same sequence on each run.

In order to produce the number sequence, they store internal state which can be substantial in size. There is a tradeoff between memory and required speed, and the quality of the produced sequence.

There is also a `typedef` called `default_random_engine` which is an implementation defined typedef for one of the engines. In VS2013, it's based on the Mersenne twister.

All engines provide the following methods:

Method	Effects
`seed`	sets the current state
`discard(n)`	equivalent to calling `operator()` `n` times and discarding results
`operator()`	returns the next number in the sequence
`min`	returns the minimum possible value
`max`	returns the maximum possible value

There are `typedefs` available for engines with predefined parameters. It's also possible to instantiate an engine template with the particular parameters you need, but it requires understanding the algorithm that's used to generate the value sequence.

`mersenne_twister_engine`

This engine generates the best quality values, however it requires a large state (624 values in the case of the predefined *mt19937*) and is slower than the other engines.

Predefined configurations supplied: *mt19937*, *mt19937-64*.

`linear_congruential_engine`

This engine is fairly fast, and its state consists of just one value (it only relies on the previous value to produce the next one). However, its period is less than 264 which is quite small.

Predefined configurations supplied: `minstd_rand`, `minstd_rand0`.

`subtract_with_carry_engine`

This engine also has a small state (e.g. 25 values for the predefined `ranlux24_base`) and is very fast.

Predefined configurations supplied: `ranlux24_base`, `ranlux48_base`.

`random_device`

`random_device` is a special kind of engine. Its intention is to interface with a hardware device in order to produce truly random numbers. However, if no such device is present, the engine is allowed to generate numbers in software.

`random_device` has a method called `entropy` which is supposed to return zero when a software generator is used (but in fact returns 32.0 in VS2013), and a non-zero estimate of the entropy if numbers are coming from external hardware.

Engine adapters

Engine adapters modify the values returned or the sequence they are returned in from the underlying engine. There are three of them available:

- `discard_block_engine<Engine, p, r>`

- `independent_bits_engine<Engine, w, IntType>`
- `shuffle_order_engine<Engine, k>`

`discard_block_engine` takes `p` values produced by the base engine, discards `(p - r)` of them (so `r` must be less than `p`), and returns the remaining `r` values. Its state consists of the state of the base engine, plus an integer for the number of `operator()` calls that have occurred in the current cycle. Predefined configurations supplied: `ranlux24`, `ranlux48`.

`independent_bits_engine` repacks bits from several values returned by the base engine to produce new `w`-bit values. Its state is the state of the base engine.

`shuffle_order_engine` changes the order of values returned from the base engine. It stores `k` values returned by the base engine in an array, and uses them for reshuffling subsequent output. Its state consists of the state of the base engine, plus `(k + 1)` values. Predefined configurations supplied: `knuth_b`.

`seed_seq`

This class is a seed value generator. Given a sequence of inputs, it produces a sequence of values distributed over [0, 232), even if values in the input sequence are bunched together:

```
array<int, 5> arr = {1, 2, 3, 4, 5};
seed_seq seq(arr.begin(), arr.end());
vector<seed_seq::result_type> seeds(10);
seq.generate(seeds.begin(), seeds.end());
// seeds now contains values such as 3904109078
```

The output can be used to populate multiple engines, or an engine that requires a lot of entropy.

Distributions

Distributions provide random numbers in a defined range with particular statistical properties.

All the distributions provide the following methods:

Method	Effects
`operator()`	given an engine, returns a random value
`reset`	resets the distribution state
`min`	returns the bottom end of the range of values
`max`	returns the top end of the range of values

In addition, individual distributions provide specific methods to retrieve their parameters. There are several groups of distributions with different properties, which are listed below.

Uniform distributions

Distribution	Sequence produced
`uniform_int_distribution`	uniformly distributed integer values
`uniform_real_distribution`	uniformly distributed real values
`generate_canonical`	uniformly distributed real values in `[0, 1)` with `n` bits of randomness

Note:

Uniform distribution on Wikipedia

Normal distributions

Distribution	Sequence produced
`normal_distribution`	real values with a standard normal distribution
`lognormal_distribution`	real values with a log-normal distribution
`cauchy_distribution`	real values with a Cauchy distribution
`chi_squared_distribution`	real values with a chi-squared distribution

`fisher_f_distribution`	real values with a Fisher's F-distribution
`student_t_distribution`	real values with a Student's t-distribution

Note:

Normal distribution on Wikipedia

Log-normal distribuion on Wikipedia

Cauchy distribution on Wikipedia

Chi-squared distribution on Wikipedia

Fisher's F-distribution on Wikipedia

[Student's t-distribution on Wikipedia](http://en.wikipedia.org/wiki/Student's_t-distribution)

Bernoulli distributions

Distribution	Sequence produced
`bernoulli_distribution`	boolean values with a Bernoulli distribution
`binomial_distribution`	integer values with a binomial distribution
`negative_binomial_distribution`	integer values with a negative binomial distribution
`geometric_distribution`	integer values with a geometric distribution

Note:

Bernoulli distribution on Wikipedia

Binomial distribution on Wikipedia

Negative binomial distribution on Wikipedia

Geometric distribution on Wikipedia

Poisson distributions

Distribution	Sequence produced
`poisson_distribution`	integer values with a Poisson distribution
`exponential_distribution`	real values with an exponential distribution
`extreme_value_distribution`	real values with an extreme value distribution
`gamma_distribution`	real values with a gamma distribution
`weibull_distribution`	real values with a Weibull distribution

Note:

Poisson distribution on Wikipedia

Exponential distribution on Wikipedia

Extreme value distribution on Wikipedia

Gamma distribution on Wikipedia

Weibull distribution on Wikipedia

Sampling distributions

Distribution	Sequence produced
`discrete_distribution`	integer values with a discrete distribution
`piecewise_constant_distribution`	real values distributed on constant subintervals
`piecewise_linear_distribution`	real values distributed on defined subintervals

Note:

Sampling distribution on Wikipedia

Non-standard behavior and bugs in Visual Studio 2013

As I mentioned above, `random_device` has a method called `entropy` which is supposed to return zero when a software generator is used, but in fact returns 32.0 in VS2013.

Rational Arithmetic and Time Support Libraries

C++11 provides new standard libraries for compile time rational number representation and manipulation, and runtime time utilities. The time utilities library relies on the rational number support, so I discuss both libraries in this chapter.

Rational number representation and manipulation

The numbers are represented using the `ratio` template which takes two arguments, the numerator and the denominator:

```
typedef ratio<2, 60> r2_60;
typedef ratio<-1, 3> r1_3;
```

> **Note:**
>
> Include `<ratio>` to use the rational arithmetic functionality.

The numerator and denominator have to be representable by constants of type `intmax_t`, and exclude the largest negative integer value.

> **Note:**
>
> The denominator argument is equal to 1 by default.

The representation is normalized internally. For an object `ratio<N, D>`, the numerator and denominator will be divided by the greatest common divisor, and the static data members `num` and `den` will have the following values:

```
num = sign(N) * sign(D) * abs(N) / gcd
den = abs(D) / gcd
```

You can perform arithmetic operations on `ratio` objects using templates:

```
cout << "1/2 + 1/3 = "
    << to_s<ratio_add<ratio<1, 2>, ratio<1, 3>>::type>()
    << endl;      // 5/6

cout << "1/2 - 1/3 = "
    << to_s<ratio_subtract<ratio<1, 2>, ratio<1, 3>>::type>()
    << endl; // 1/6

cout << "1/2 * 1/3 = "
    << to_s<ratio_multiply<ratio<1, 2>, ratio<1, 3>>::type>()
    << endl; // 1/6

cout << "1/2 / 1/3 = "
    << to_s<ratio_divide<ratio<1, 2>, ratio<1, 3>>::type>()
    << endl;   // 3/2
```

Similarly, templates are provided for comparing rational numbers:

```
cout << boolalpha;
cout << "1/2 < 1/3 is "
    << ratio_less<ratio<1, 2>, ratio<1, 3>>::value
    << endl;  // false

cout << "1/2 = 1/3 is "
    << ratio_equal<ratio<1, 2>, ratio<1, 3>>::value
    << endl;  // false

cout << "1/2 > 1/3 is "
    << ratio_greater<ratio<1, 2>, ratio<1, 3>>::value
    << endl;    // true
```

The other available comparison operations are `ratio_not_equal`, `ratio_less_equal` and `ratio_greater_equal`.

Finally, the library includes definitions of SI prefixes:

```
typedef ratio<1,         1000000000000000000> atto;
typedef ratio<1,            1000000000000000> femto;
typedef ratio<1,               1000000000000> pico;
typedef ratio<1,                  1000000000> nano;
typedef ratio<1,                     1000000> micro;
typedef ratio<1,                        1000> milli;
typedef ratio<1,                         100> centi;
typedef ratio<1,                          10> deci;
typedef ratio<                        10, 1> deca;
typedef ratio<                       100, 1> hecto;
typedef ratio<                      1000, 1> kilo;
typedef ratio<                   1000000, 1> mega;
typedef ratio<                1000000000, 1> giga;
typedef ratio<             1000000000000, 1> tera;
typedef ratio<          1000000000000000, 1> peta;
typedef ratio<       1000000000000000000, 1> exa;
```

> **Note:**
>
> The standard defines 4 more prefixes: `yocto`, `zepto`, `zetta` and `yotta`, which aren't provided by Visual Studio. They only need to be defined if the constants used in their definition can be represented by values of `intmax_t`.

These prefixes are useful, for example, when working with the `chrono` library (discussed later in this chapter).

Time utilities

There are three main concepts in the time support library: time durations, clocks and time points. I'll discuss each in turn.

> **Note:**
>
> Include `<chrono>` to use the standard library time utilities.

Time durations

A duration is defined by a *number of ticks* and the *length of a tick* represented as a rational number (expressed using the `ratio` template discussed above):

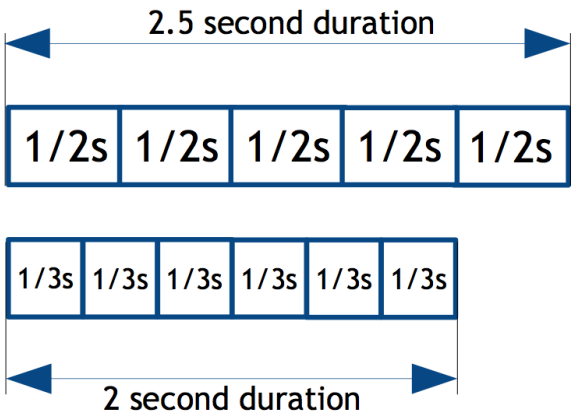

Figure 5: Time durations and tick lengths

Note:

All of the time utilities are defined in the namespace `chrono`. For brevity, the subsequent examples in this chapter assume `using namespace chrono`.

To construct a duration object, you need to specify the type for tick count and the type expressing tick length:

```
// 10 ticks of 1 second each = 10 seconds
duration<int> d(10);
// 10 ticks of 1/10 seconds each = 1 second
duration<int, ratio<1, 10>> deciseconds(10);
// 10.5 ticks of 60 seconds each = 630 seconds
duration<double, ratio<60>> duration_minutes(10.5);
```

Note:

The number of ticks is allowed to be negative but you'll get compilation errors if you use a negative tick length.

The library provides a number of predefined duration types for convenience:

```
hours route_time(5);
minutes descent_time(25);
nanoseconds polling_frequency(500);
```

> **Note:**
>
> In Visual Studio, minutes and hours use `int` and the rest of the convenience types are defined using `long long`.

The other predefined types are `seconds`, `milliseconds` and `microseconds`. The standard defines the following requirements for the underlying types of these `typedefs`:

```
typedef
    duration<signed integer type of >= 64 bits, nano> nanoseconds;
typedef
    duration<signed integer type of >= 55 bits, micro> microseconds;
typedef
    duration<signed integer type of >= 45 bits, milli> milliseconds;
typedef
    duration<signed integer type of >= 35 bits> seconds;
typedef
    duration<signed integer type of >= 29 bits, ratio< 60>> minutes;
typedef
    duration<signed integer type of >= 23 bits, ratio<3600>> hours;
```

A duration can be implicitly converted to a duration with a different tick length, as long as its tick length is an integer multiple of the new duration's tick length:

```
duration<int, ratio<1, 2>> half_seconds(1);
duration<int, ratio<1, 8>> eigth_seconds(half_seconds);  // OK

duration<int, ratio<2, 3>> two_thirds(1);
duration<int, ratio<1, 3>> one_thirds(two_thirds);       // OK
```

> **Note:**
>
> These conversions are facilitated by `common_type` overloads provided in the `chrono` library.

Another way to look at it is that as long the conversion doesn't cause loss of precision, it will happen automatically. This means that hours can be converted into minutes, minutes into seconds, and so on. However, there are no implicit conversions in the other direction.

Similarly, durations with different tick count types can be converted implicitly as long as there is no loss of data:

```
duration<int, ratio<1, 3>> one_thirds(two_thirds);
duration<double, ratio<1, 7>> fuzzy(one_thirds);  // OK, 4.667 1/7's
                                                  // of a second

typedef duration<int, ratio<1, 7>> Sevenths;
Sevenths precise(fuzzy);                          // error
```

If you need to, you can force lossy conversions with the help of `duration_cast`:

```
seconds s(duration_cast<seconds>(one_thirds));
```

> **Note:**
>
> The template argument for `duration_cast` is a `duration` type.

Durations can be compared, even when they have different tick lengths:

```
duration<int, ratio<1, 2>> d1(1);
duration<int, ratio<1, 3>> d2(2);
d1 < d2;   // true
d1 <= d2;  // true
d1 > d2;   // false
d1 >= d2;  // false
d1 == d2;  // false
d1 != d2;  // true
```

You can also perform arithmetic on durations. For operations involving two durations, the tick length of the result will be determined by the least common multiple of the tick lengths of the operands:

```
duration<int, ratio<1, 2>> d1(4);
duration<int, ratio<1, 3>> d2(1);
auto sum = d1 + d2;  // sum is 14 ticks of 1/6s
```

Operation	Description
`d1 + d2`	Add `d1` and `d2` to produce a new duration
`++d and d++`	Increment `d` by 1 tick
`d1 += d2`	Add `d2` to `d1`
`d1 - d2`	Subtract `d2` from `d1` to produce a new duration
`--d and d--`	Decrement `d` by 1 tick
`d1 -= d2`	Subtract `d2` from `d1`
`d * num and num * d`	Produce a new duration that's num times the length of `d`

Operation	Description
`d *= num`	Multiply `d`'s tick count by `num`
`d / num`	Produce a new duration that's `num` times shorter than `d`
`d1 / d2`	Return the number of times `d2` can fully fit into `d1`; broken in VS2013
`d /= num`	Divide `d`'s tick count by `num`
`d % num`	Produce a new duration with a tick count equal to the remainder from dividing `d`'s tick count by `num`
`d1 % d2`	Returns the remainder from dividing `d1`'s tick count by `d2`'s tick count (after converting to common denominator); broken in VS2013
`d %= num`	Make `d`'s tick count equal to the remainder from dividing the original tick count by `num`
`d1 %= d2`	Make `d1`'s tick count equal to the remainder from dividing `d1`'s original tick count by `d2`'s tick count; this operator requires that `d1` and `d2` are of the same type

In addition, the `duration` class provides several methods:

```
minutes d(15);

d.count();  // the number of ticks (15)
d.min();    // returns a duration with the same tick length as d and
            // the smallest possible tick count (-2147483648)
d.max();    // returns a duration with the same tick length as d and
            // the largest possible tick count (2147483647)
d.zero();   // returns a duration with the same tick length as d and
            // a tick count of zero
```

Finally, you can use two member typedefs `rep` and `period` to get the underlying types:

```
milliseconds::rep;    // yields long long
milliseconds::period; // yields ratio<1, 1000>
```

Clocks and time points

A `clock` is defined by its `epoch` (i.e. its starting point) and its `resolution` (i.e. the length of one tick). A clock provides a method called *now* which returns a *time point* object representing the current time as a duration relative to the clock's epoch:

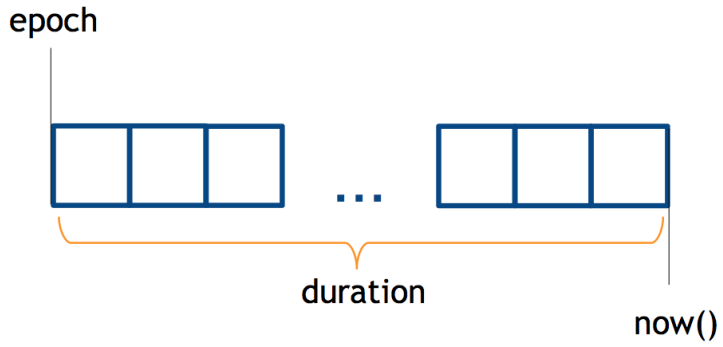

Figure 6: The relationship between a clock, epoch and time point

The time point object is in turn associated with a clock.

Each clock type provides several `typedefs`:

typedef	Description
duration	duration type of the clock reflecting the clocks resolution
rep	equivalent to duration::rep
period	equivalent to duration::period
time_point	time point type for the clock

The clock interface also includes the is_steady attribute which allows you to query whether the clock is guaranteed to be steady, i.e. that for any two current time values t1 and t2 obtained in succession, t2 is greater than t1. This means, for example, that a steady clock isn't allowed to go backwards due to daylight savings time changes.

> **Note:**
>
> Visual Studio also retains is_monotonic attribute which was present in the earlier drafts of the standard and was subsequently removed. It also has monotonic_clock which is a typedef for steady_clock.

There are three clocks provided by the library: system_clock, steady_clock and high_resolution_clock.

system_clock This clock provides a mapping to the C API via the to_time_t and from_time_t methods. The time_t object returned by to_time_t and the time_point object returned by from_time_t will both have the coarser of the precisions between time_point and time_t.

system_clock isn't guaranteed to be steady, but in practice it should be in the Visual Studio implementation.

steady_clock steady_clock derives from system_clock in Visual Studio, so it also has methods for conversion to and from time_t, although that's not required by the standard. The only difference from system_clock is that it reports that it's steady (its is_steady flag is set to true).

However, in reality the clock isn't steady and can jump backwards when system time is adjusted. If you don't require cross-platform compatibility, then a workaround is to use

`GetTickCount`/`GetTickCount64`/`QueryPerformanceCounter` Windows API functions instead of relying on this clock.

`high_resolution_clock` The intention of the C++11 standard seems to be for this clock to have the maximum resolution that can be provided by the system. However, in VS2013 `high_resolution_clock` is simply a synonym for `system_clock`, and it doesn't currently use the `QueryPerformanceCounter` function from the Windows API. This is considered to be a bug by Microsoft and will be fixed in a later version.

The workaround is either to use `QueryPerformanceCounter` API function directly, or to use the Boost version of the `chrono` library which implements `high_resolution_clock` using this API function.

Time points

As I mentioned above, a time point combines a clock with a duration to express a point in time as a time interval from the start of the clock's epoch.

Time points can be compared, and you can perform arithmetic operations on them:

```
auto start_time = steady_clock::now();
// ...
if (steady_clock::now() > start_time + hours(1))
{
    auto run_time = duration_cast<minutes>(
        steady_clock::now() - start_time);
    cout << "Time since startup: " << run_time.count() << " min"
        << endl;
}
```

In addition to `operator>`, other comparisons provided for time points are `>=`, `<`, `<=`, `==` and `!=`. The full list of arithmetic operations is below:

Operation	Description
`t + dur` and `dur + t`	Produce a new time point by adding `dur` to `t`'s duration
`t += dur`	Add `dur` to `t`'s duration

`t - dur`	Produce a new time point by subtracting `dur` from `t`'s duration
`t1 - t2`	Produce a duration representing the difference between `t1` and `t2`
`t -= dur`	Subtract `dur` from `t`'s duration

Time points can be constructed in several ways:

```
// default constructor produces a time point at the start of the
// clock's epoch
time_point<system_clock> epoch;

// a time_point object can be constructed from another time_point
// (which could have a larger tick length)
time_point<system_clock, nanoseconds> t1(epoch);

// a time point can be constructed from a duration, which is
// the same as time_point() + dur
time_point<system_clock> t2(epoch + minutes(1));
```

Similarly to `duration_cast`, you can employ `time_point_cast` to obtain a time point with a different representation of duration:

```
// day_two will have its duration in minutes
auto day_two = time_point_cast<minutes>(epoch + hours(24));
```

Finally, the `time_point` class provides several methods. Static methods `min` and `max` return minimum and maximum possible time points, similarly to the `duration` methods. `time_since_epoch` returns the time point's duration object:

```
auto dur = day_two.time_since_epoch(); // dur is a duration in minutes
```

Non-standard behavior and bugs in Visual Studio 2013

The implementation of `d1 % d2` for durations is non-standard. It's supposed to produce a new duration object with a tick period determined by the common denominator of the arguments. Instead, it only returns the remainder which isn't very useful.

In addition, both the / and % operators have been broken in the transition from VS2012 to VS2013, and appear to be completely non-functional.

Concurrency Support: High Level

The concurrency support in the C++11 standard is extensive enough to require a separate book, and Visual Studio 2013 implements a significant portion of it. I will provide an overview of these features in this and the following chapters, starting with the high level concurrency concepts and progressing towards finer grained features for controlling concurrent code execution.

Memory model overview

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. The initialization of objects of static storage duration is now guaranteed to be thread-safe by the standard, but this isn't implemented by Visual Studio 2013. In any case, access to such objects still needs to be synchronized.

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.

Asynchronous code execution

The high level concurrency features allow you to execute code asynchronously without worrying about creating threads or locks. You can use `async` to achieve this:

```
// calculate_route calculates the route and returns its length
auto distance = async(launch::async, &Autopilot::calculate_route,
    autopilot);
```

```
// ... other code ...
auto time_to_destination = distance.get() / nav.get_ground_speed();
```

> **Note:**
>
> Include <future> to use high level concurrency features.

This code will cause calculate_route to run in a separate thread behind the scenes:

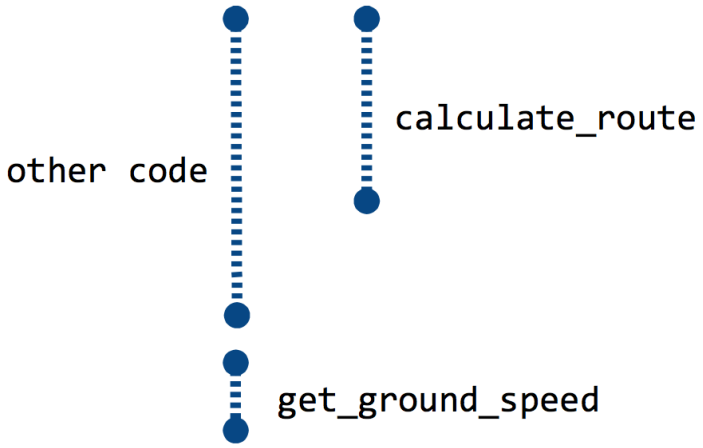

Figure 7: calculate_route running on a separate thread

> **Note:**
>
> The async implementation is powered by Microsoft's Concurrency Runtime, which will reuse threads if possible.

You can pass anything callable to async: function pointers, lambdas or function objects. You can also pass arguments:

```
auto distance = async(launch::async, &Autopilot::calculate_route,
    autopilot, start_location, end_location);
```

By default, the arguments are passed by value. To pass by reference, use the ref or cref wrappers (same as with bind). When passing arguments by reference or by pointer,

remember to make sure that they exist throughout the lifetime of the asynchronous operation. You also need to synchronize access to data that's shared between threads. The tools for that are discussed in the following chapters.

What happens if the function passed to `async` throws an exception? For example, if you want to execute the following on another thread:

```
void Autopilot::engage()
{
    // ... check preconditions ...
    if (!control_system_available)
    {
        throw runtime_error("Failed to engage autopilot: "
            "Can't access control system.");
    }
    // ...
}
```

If engage throws an unhandled exception, it will be transported to the thread where you launched `async` so you can handle it there:

```
try
{
    Autopilot autopilot;
    auto engage_task = async(launch::async, &Autopilot::engage,
        autopilot);
    // ...
    engage_task.get();  // throws an exception
}
catch(const runtime_error& e)
{
    cout << e.what() << endl;
}
```

Note:

If you called `wait` instead of `get`, the exception would simply get swallowed. So if you want to handle exceptions on the calling thread, take care to call get even if you aren't interested in the result of the asynchronous operation.

Launch policies and lazy evaluation

The first (optional) argument to `async` specifies the launch policy:

Policy	Effect
`launch::async`	Code will run on a separate thread; Concurrency Runtime will reuse threads if possible, to improve efficiency
`launch::deferred`	Code will not run on a separate thread, instead it will just be deferred until `.get()` is called
`launch::async | launch::deferred`	Code may or may not run on a different thread, and Concurrency Runtime will only create enough threads to saturate the hardware

> **Note:**
>
> `async` throws a `system_error` exception with error code `resource_unavailable_try_again` if it's unable to launch a new thread.

If you omit the policy argument altogether, it's equivalent to using `launch::async | launch::deferred`.

`launch::deferred` policy allows you to perform lazy evaluation, as your code will only be executed if you call `.get()` on the `async` return value:

```
auto this_res = async(launch::deferred, do_this);
auto that_res = async(launch::deferred, do_that);

// only one function gets executed, depending on user input
auto res = user_wants_this ? this_res.get() : that_res.get();
```

`future`

`async` returns an object of type `future`. A `future` provides access to the result of the asynchronous computation (via the `get` method). The good thing about it is that it provides synchronized access to the result – you don't need to worry about it.

Note:

As `get` blocks, you want to call `async` as early as possible in your code, to maximize the chances that by the time you need the result, the calculation is done (in which case `get` doesn't need to block) or at least the blocking time is minimized.

Be aware that `get` can only be called once. You can check whether it's already been called using the `valid` method:

```
future<int> task = async(launch::async, [] { return 10; });
task.valid();  // true
auto result = task.get();
task.valid();  // false
```

After a call to `get`, calling any method of `future` other than `valid` is undefined behavior. Compilers are encouraged by the standard to detect this situation and throw a `future_error` with an error code of `future_errc::no_state`. Visual Studio does this.

There are specializations of `get` for references and for callable objects that don't return anything:

```
R future::get();
R& future<R&>::get();
void future<void>::get();
```

Polling and waiting for task completion

`future` also has a number of blocking methods to wait for the result. This can be useful, for example, to update the UI while waiting for a background task to complete:

```
cout << "Indexing airport database";
auto res = async(launch::async, [] {
    /* call a lengthy indexing method */ });

while (res.wait_for(milliseconds(500)) == future_status::timeout)
    cout << ".";
cout << endl << "Indexing complete." << endl;
```

Note:

For brevity, the examples assume `using namespace chrono` in addition to `using namespace std`.

Note:

When you use `cout` in a multi-threaded application, you need to call `cout.sync_with_stdio(true)` to prevent race conditions. Note that you'll still need to synchronize access to `cout` to prevent interleaving of the output from different threads.

`wait_for` blocks until the specified timeout has elapsed, or until the result has become available.

`wait_until` can be used to wait until a specific point in time:

```
if (res.wait_until(midnight) == future_status::timeout)
{
    cout << "Ran out of time before completing indexing." << endl;
    stop_indexing();  // indexing not done by midnight,
                      // so stop until tomorrow
}
```

When the result is ready, `wait_for` and `wait_until` return `future_status::ready`. If the code passed to `async` is deferred instead of being run on another thread (e.g. if you used the `launch::deferred` policy), then `wait_for` and `wait_until` will immediately return `future_status::deferred`.

Finally, if you just need to wait until the result is available, you can use `wait`:

```
res.wait();
```

Sometimes you may want a way of checking whether the result is ready without blocking. The way to achieve it is to simply use a zero timeout:

```
while (res.wait_for(seconds(0)) == future_status::timeout)
{
    cout << ".";
    this_thread::sleep_for(milliseconds(500));
}
```

If you need to cancel a task, you'll need to implement your own mechanism for it using the concurrency support primitives discussed in the following chapters.

Getting multiple threads to wait for one result

`future` allows you to wait for a result of an asynchronous operation on one thread. However, sometimes you need multiple threads to wait for a result computed on another thread:

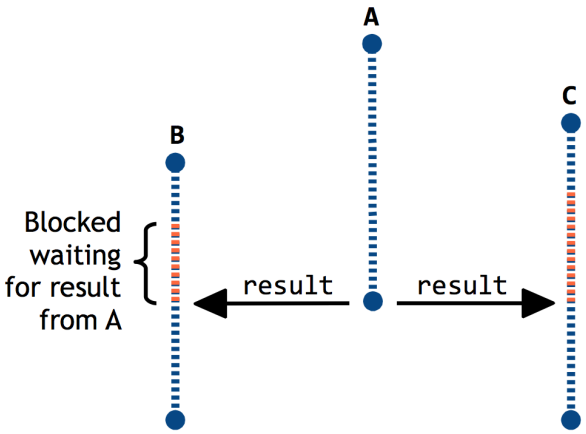

Figure 8: Blocked threads waiting for a result

In this situation, you can use `shared_future`, which allows multiple threads to wait for and obtain the result from another thread. All the waiting threads get access to the same result. Here is an example:

```
auto diagnostics_task = async(launch::async,
    run_startup_diagnostics).share();

auto autopilot_task = async(launch::async, &Autopilot::init,
    autopilot, diagnostics_task);
auto controls_task = async(launch::async, &ControlSystem::init,
    control_system, diagnostics_task);

while (
```

```
    diagnostics_task.wait_for(seconds(0)) == future_status::timeout ||
    autopilot_task.wait_for(seconds(0)) == future_status::timeout ||
    controls_task.wait_for(seconds(0)) == future_status::timeout)
{
    cout << ".";
    this_thread::sleep_for(milliseconds(500));
}

cout << endl << "System ready." << endl;
/* continue with system operation */
```

Note the `share()` call on the result of the first `async`. It converts the `future` object returned by `async` into a `shared_future` object. `shared_future` can also be constructed from `future`:

```
shared_future<DiagnosticsResults> diagnostics_task =
    async(launch::async, run_startup_diagnostics);
```

The `init` methods of `Autopilot` and `ControlSystems` can then use this object to wait for the diagnostics results:

```
void Autopilot::init(
    shared_future<DiagnosticsResults> diagnostics_task)
{
    /* ... do startup stuff ... */
    diagnostics_task.wait(); // don't care about the results
    /* ... continue with post-diagnostics init ... */
    cout << "Autopilot initialized." << endl;
}
void ControlSystem::init(
    shared_future<DiagnosticsResults> diagnostics_task)
{
    /* ... do startup stuff ... */
    const auto& diagnostics_res = diagnostics_task.get();
    /* ... continue with post-diagnostics init ... */
    cout << "Control system initialized." << endl;
}
```

`shared_future` has a similar interface to `future`, with the exception that `get` returns the result by `const` reference, not by value:

```
const R& shared_future::get() const;
R& shared_future<R&>::get() const;
void shared_future<void>::get() const;
```

If you access the result from multiple threads through references returned by `get`, you need to synchronize access to it unless you only use read-only operations. The synchronization provided by `shared_future` doesn't extend to this situation.

Non-standard behavior and bugs in Visual Studio 2013

`async` can't handle move-only arguments

You won't be able to pass arguments of move-only types to `async`.

Initialization of statics not thread-safe

As I mentioned at the start of the chapter, the initialization of objects of static storage duration is not guaranteed to be thread-safe in Visual Studio.

Non-blocking `future` destructor

A potential gotcha with `future` is that the standard currently requires the destructor of a `future` object returned by `async` to block until the corresponding asynchronous operation is finished (this doesn't apply to `futures` used otherwise). So code like this should *not* run in parallel when compiled with a compliant compiler, contrary to what you might expect:

```
async(launch::async, [] { f(); }); // future's destructor should
                                   // block till f is done
async(launch::async, [] { g(); }); // so g shouldn't run until f()
                                   // is done
```

Note:

This issue is explained in more detail in Herb Sutter's short paper, along with a proposal to remove the requirement for ~future to block: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3451.pdf

Consequently, you need to defer the destruction of the `future` objects to ensure parallel execution:

```
auto future1 = async(launch::async, [] { f(); });
auto future2 = async(launch::async, [] { g(); });
```

Whether intentionally or not, VS2013 doesn't implement this blocking behavior, so both of the above examples will produce parallel execution.

If Visual Studio's implementation fully conformed to the standard, `shared_future` returned by `async` would have the same caveat as `future`, that its destructor blocks until the asynchronous operation is complete. In that case, the thread that happened to run last would block because of `shared_future`'s destructor.

Concurrency Support: Building Blocks

The high level features are sufficient in a lot of situations. However, this is C++ so you may well need more control. In this chapter we'll look at creating your own threads and the template `promise` for inter-thread communication of computed results.

Rolling your own threads

Creating and launching a thread is simple:

```
Autopilot autopilot;
thread t1([=] { autopilot.calculate_route(); });

thread t2(&Navigator::get_distance, navigator,
    start_location, end_location);
```

> **Note:**
>
> Include `<thread>` to use the features in this chapter.

> **Note:**
>
> If the system is unable to create a new thread, the constructor throws a `system_error` with error code `resource_unavailable_try_again`.

You can pass in any callable object, followed by its arguments if neeeded. The arguments are passed by value, so you need to use `ref` or `cref` wrappers if you want to pass them by reference instead (with obvious race condition implications to be aware of).

You can create a thread object without launching a thread (which can be useful when storing thread objects in an STL container, for example). At a later point, you can launch an actual thread and associate it with the object using the assignment operator:

```
thread t3;
// ...
t3 = thread([=] { autopilot.calculate_route(); });
```

Joining threads

Member function `join` allows a thread to block until another thread is terminated:

```
// running code in thread a --->

Autopilot autopilot;
thread b([=] { autopilot.calculate_route(); });

b.join();
```

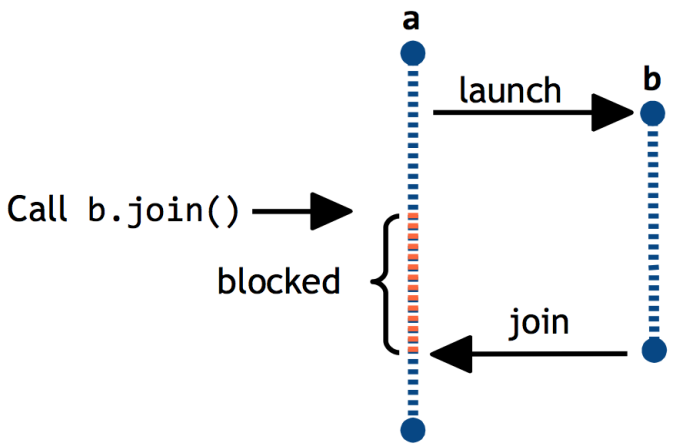

Figure 9: Blocked thread waiting in join()

`join` can throw system_error exceptions with the following error codes:

- `resource_deadlock_will_occur` if you're trying to join a thread with itself or if deadlock is detected
- `no_such_process` if the thread isn't valid
- `invalid_argument` if the thread isn't joinable.

It's important to be aware that if a thread object goes out of scope before its associated thread completes its work, then the whole program aborts and calls `std::terminate`. This is one of the reasons you may want to use `join`, in addition to the requirements your application logic places on threads.

You can check if a thread object is joinable, i.e. that your thread object has an actual thread associated with it:

```
b.joinable();      // true for the above example

thread blank;
blank.joinable();  // false
```

> **Note:**
>
> For brevity, the examples assume `using namespace chrono` in addition to `using namespace std`.

Detaching threads

The opposite of `join` is the `detach` method. When you call it, the thread continues to execute in the background but you no longer have any access to it. The operating system is responsible for releasing thread resources once the thread is finished.

```
Autopilot autopilot;
thread b([=] { autopilot.calculate_route(); });
b.detach();     // the thread associated with b continues to run
                // however b is disconnected from it

b.joinable();   // false - b is no longer associated with the thread
```

> **Note:**
>
> `detach` throws a `system_error` exception with error code `no_such_process` if the thread isn't valid, or `invalid_argument` if the thread isn't joinable.

You need to be aware that if `main` finishes while detached threads are running in the background, they are terminated abruptly.

> **Note:**
>
> This and other related issues are discussed in more detail in this paper: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2880.html

Detached threads can continue running during and after the destruction of static/global objects, so you need to ensure that you've resolved these lifetime issues in your code. One solution for this is to have the main thread wait for signals from detached threads that they have finished before it terminates itself.

The destruction of thread local objects (explained below) can also happen concurrently with static destruction when the process shuts down, potentially creating complex lifetime issues. For now, this issue is largely mitigated by Visual Studio's non-standard implementation of thread local variables which doesn't allow them to have non-trivial destructors.

However, once Visual Studio fully conforms to the standard, combining thread local variables with detached threads will have potential to create very subtle bugs.

Thread IDs and native handles

Each thread has a unique ID represented by an object of class `thread::id`:

```
auto id = t.get_id();
```

> **Note:**
>
> The implementation may reuse the ID of a terminated thread if it can no longer be joined.

Thread objects that don't represent a thread of execution (i.e. default constructed objects) all have the same ID which is, however, different from all the IDs of actual threads.

`thread::id` objects have relational operators so they can be used as keys in associative containers, as well as support for being used as `unordered_*` container keys. You can also put a `thread::id` object into an output stream to see its integer representation.

The standard also defines a method for obtaining a native handle:

```
auto handle = t.native_handle();
```

The type returned by `native_handle` is implementation defined, and its use is platform specific. In Visual Studio, this method returns a `HANDLE` which is cast to `void*`.

promise

The thread class doesn't provide any way to retrieve the result of the code it executes. The template promise is the counterpart to future which provides a way to obtain a result from another thread. A result of asynchronous operation is communicated between threads via a promise/future pair. The future allows you to get a value (or an exception), and a promise allows you to set a value or an exception on another thread.

> **Note:**
>
> You can think of promise as a communication channel which sends a value or an exception to a future across threads.

async uses a similar mechanism behind the scenes to provide the future it returns with the result. If you create your threads manually, then you need to use a promise object to make the result available:

```
promise<double> res_promise;
auto res = res_promise.get_future();

thread t([&]()
{
    // ...
    res_promise.set_value(navigator.get_distance(start_location,
        end_location));
});

// ...

auto distance = res.get();
t.join();
```

Instead of a result, you can communicate an exception:

```
promise<double> res_promise;
auto res = res_promise.get_future();

thread t([&]()
```

```
{
    Navigator navigator;
    // ...

    if (!navigator.is_ready())
        res_promise.set_exception(
            make_exception_ptr(
                runtime_error("Navigator isn't ready")));
});
```

`make_exception_ptr` converts the exception into an `exception_ptr` required by `set_exception`. It's discussed in more detail in the next chapter. The exception can then be handled on the receiving thread:

```
try
{
    auto distance = res.get();
}
catch (const exception& e)
{
    // handle exception set by promise
}

t.join();
```

`future` and `promise` synchronize their operations for setting and obtaining a value or exception so you don't need to do it manually. However, in other situations you need to ensure correct behavior yourself. For example, if the promise object is being destroyed in one thread while its `set_value` method executes in another thread, it can cause deadlocks and other issues.

The lifecycle of `promise` and its interaction with `future` is strictly regulated. Once you've used `set_value` or `set_exception` on the promise, it's considered satisfied. A promise can also be *broken* if it goes out of scope before the result is obtained by its `future` counterpart:

```
promise<int> p;
auto res = p.get_future();

{
    promise<int> p2(move(p));
}                       // promise is destroyed
res.get();              // get() throws a "broken promise" exception,
                        // an instance of future_error with error code
                        // future_errc::broken_promise
```

However, the `promise` object can go out of scope without generating an exception as long as `future`'s `get` method isn't called.

`get_future`, `set_value` and `set_exception` can only be called once, otherwise an exception will be thrown. You can't call *both* `set_value` and `set_exception` either:

```
promise<int> p;
auto res = p.get_future();
auto res2 = p.get_future(); // throws future_error with error code
                            // future_errc::future_already_retrieved

// ---------------------------------------------------------------------

t = thread([&]
{
    try
    {
        p.set_exception(make_exception_ptr(runtime_error("Exc 1")));
        p.set_value(11);    // throws future_error with error code
                            // future_errc::promise_already_satisfied
    }
    catch (future_error& e)
    {
        // ...
    }
});
```

Once you set the value or exception, the thread continues executing. In some cases, you may need to notify the `future` when the thread has finished its work instead of

immediately. The promise class provides methods to set the value or exception such that notification is delayed until thread exit (after thread local variables have been destroyed):

```
p.set_value_at_thread_exit(10);

p.set_exception_at_thread_exit(
    make_exception_ptr(runtime_error("Exc 1")));
```

These methods belong to the same category as set_value and set_exception. You can only call *one* of these four methods, otherwise you'll get a future_error exception.

> **Note:**
>
> The standard specifies that "set_value, set_exception, set_value_at_thread_exit, and set_exception_at_thread_exit member functions behave as though they acquire a single mutex associated with the promise object while updating the promise object."

Helper functions for use with threads

There are several helper functions which are provided in the namespace this_thread.

get_id returns the thread ID object for the current thread:

```
cout << "Current thread ID: " << this_thread::get_id() << endl;
```

yield gives the system an opportunity to reschedule (which means it may or may not happen), which can be useful in situations where busy-waiting is involved:

```
this_thread::yield();
```

sleep_for and sleep_until block the current thread for a period of time:

```
this_thread::sleep_for(milliseconds(500));
this_thread::sleep_until(midnight);
```

The thread class contains a static function which returns the number of hardware thread contexts – but it should only be considered an estimate:

```
thread::hardware_concurrency();   // returns 2 on my system
```

> **Note:**
>
> If `hardware_concurrency` returns 0, it means the value couldn't be computed.

packaged_task

The `packaged_task` template allows you to combine a piece of code with a future for execution at a later point:

```
packaged_task<void(int)> task(lengthy_func);
auto res = task.get_future();

task(5);   // execute lengthy_func(5) - another thread would
           // normally do this

res.get(); // obtain the result (void in this case)
```

> **Note:**
>
> Include `<future>` to use `packaged_task`.

Like `async`, it hides the details of dealing with a `promise` from you. However, in contrast with `async` which launches a thread immediately, `packaged_task` gives you control over when to launch a thread:

```
packaged_task<int(int, int)> task(juggle_numbers);
auto res = task.get_future();
thread t(move(task), 10, 20);
// ...
auto value = res.get();
```

It also gives you the opportunity to implement other ways of handling tasks, e.g. via a pool of threads servicing the task queue.

Additionally, you can instruct `packaged_task` to make the result ready when the current thread exits, after all of its thread local variables have been destroyed. This is similar to the delayed notification provided by `promise`:

```
packaged_task<void(int)> task(lengthy_func);
auto res = task.get_future();

thread t([&] { task.make_ready_at_thread_exit(5); });
res.get();
```

Note that `get_future` can only be called once, otherwise you'll get a `future_error` exception (same as with `promise`). Executing the task more than once causes the same exceptions as calling `promise::set_value` more than once. Finally, calling `reset` on the task object results in a broken promise, so the `future` will receive a `future_error` exception with error code `future_errc::broken_promise`.

Non-standard behavior and bugs in Visual Studio 2013

`thread` constructor doesn't take rvalue reference arguments

Due to a bug in Visual Studio, you won't be able to pass rvalue references via the `thread` constructor:

```
void unique_f(unique_ptr<int> p);

unique_ptr<int> p(new int(10));
thread t(unique_f, move(p));   // compile error
```

Calling `join` after `main` exits causes the program to hang

Visual Studio has a bug which causes a call to `join` after `main` exits to hang. The following program never completes:

```
#include <thread>
#include <chrono>

using namespace std;
using namespace chrono;

struct Hang
```

```
{
    Hang() : _thread([] { this_thread::sleep_for(seconds(1)); })
    {}
    ~Hang()
    {
        _thread.join();
    }
    thread _thread;
};

int main()
{
    static Hang hang;  // destroyed after main exits
    return 0;
}
```

Wrong exception type thrown by `promise`

If you call both `set_value_at_thread_exit` and `set_exception_at_thread_exit` on a `promise` object, you won't get a `future_error` as you should but instead Visual Studio will throw a different exception when attempting to lock a mutex in the `promise` class.

`future::get` doesn't throw when shared state in `packaged_task` is abandoned

Consider this code:

```
packaged_task<void()> task([](){});
future<void> f = task.get_future();
task.reset();
f.get();  // should throw but doesn't
```

The `f.get()` call should throw a `future_error` with a `broken_promise` error code because the shared state has been abandoned. However, instead it just blocks indefinitely.

`packaged_task` wrapping a `void` or reference-returning function isn't movable

If `packaged_task` wraps a function returning either `void` or a reference type (e.g. `packaged_task<void()>` or `packaged_task<int&()>`), it isn't movable.

This bug prevents passing a `packaged_task` which wraps a function returning `void` to `thread`, for example:

```
packaged_task<void(int)> task(lengthy_func);
thread t(move(task), 5);
t.join();
```

Incorrect handle value in a default-constructed `thread` object

The `assert` in this code sample will be triggered as the value returned by `native_handle` isn't equal to `INVALID_HANDLE_VALUE`:

```
thread thread;
HANDLE handle = thread.native_handle();
assert(handle == INVALID_HANDLE_VALUE);
```

Concurrency Support: Exceptions, Thread Local Storage

Manual exception handling in threads

If an unhandled exception occurs in a thread, the program terminates calling `std::terminate`, so exceptions must be handled. `future/promise` provide one mechanism for it, as you saw above. If it doesn't suit your needs, you have two other options.

> **Note:**
>
> Include `<thread>` to use the features in this chapter.

One option is to catch all exceptions inside each thread, which is sub-optimal as it leads to code duplication. The other is to provide a mechanism for transporting exceptions between threads, for example by serializing exceptions in one thread and deserializing in another. Fortunately, you don't need to implement it yourself as the standard library provides `exception_ptr` to transport exceptions to another thread.

`exception_ptr`

`exception_ptr` is a pointer-like object which refers to an exception. While it doesn't have `operator*`, `operator->` or any other public members, it has some of the semantics of a shared ownership smart pointer:

- a default constructed `exception_ptr` produces the null value of the type
- there can be multiple `exception_ptr` objects pointing to the same exception object
- the exception object referred to by `exception_ptr` remains valid while there is at least one `exception_ptr` instance referring to it
- changes in the number of `exception_ptr` objects pointing to the same exception objects don't introduce race conditions
- multiple threads can perform read-only operations on one `exception_ptr` object simultaneously

- multiple threads can modify *different* instances of `exception_ptr` simultaneously.

Note that the level of thread safety of `exception_ptr` is the same as that of built-in types, so you may need to synchronize access to `exception_ptr` instances, for example if you have multiple threads performing assignment on one `exception_ptr` variable.

Functions for working with `exception_ptr`

Suppose you have a master thread that kicks off a number of worker threads, and the worker thread functions can throw exceptions. The worker thread functions can simply put `exception_ptr` objects into an exception queue:

```
synchronized_queue<exception_ptr> exception_queue;
unsigned int WINAPI thread_func(void*)
{
    try
    {
        // ...
        // create an exception_ptr object without throwing
        // an exception
        exception_queue.push_back(
            make_exception_ptr(CustomException()));
        // ...
        throw runtime_error("Error");
        // ...
    }
    catch (...)
    {
        exception_queue.push_back(current_exception());
    }
    return 0;
}
```

Access to the exception queue needs to be synchronized as it's accessed by multiple threads (hence the custom `synchronized_queue` in the example).

`make_exception_ptr()` creates an `exception_ptr` that refers to a copy of the function's argument. It's more efficient than throwing and catching an exception and returning `current_exception()`.

`current_exception()` returns an `exception_ptr` object that refers to a copy of the currently handled exception. If no exception is being handled, then it returns a null `exception_ptr` object. If there isn't enough memory to copy the exception object, then the `exception_ptr` points to an instance of `bad_alloc`. If the function can't copy the exception object for any other reason, it calls `terminate()`.

Bear in mind that successive calls to `current_exception()` return `exception_ptr` objects that refer to different copies of the exception. As a result, these `exception_ptr` objects aren't equal.

Once the worker threads have terminated, the master thread can process the queue, re-throw the exceptions and handle them (e.g. by writing out information to a log file):

```
// handle accumulated exceptions
while (!exception_queue.empty())
{
    try
    {
        exception_ptr e = exception_queue.front();
        exception_queue.pop_front();
        rethrow_exception(e);
    }
    catch (const exception& e)
    {
        cout << "Caught an exception: " << e.what() << endl;
    }
    catch (const CustomException& e)
    {
        cout << "Caught a custom exception: " << e.message() << endl;
    }
}
```

Note:

`exception_ptr` and related functions can also be useful for crossing another kind of boundary. If you need to pass C++ callbacks which can throw exceptions to a C API (which isn't able to handle exceptions), you can use `exception_ptr` to streamline exception handling. Similarly to the example above, you can catch exceptions within your C++ callbacks, store `exception_ptr` objects, and re-throw exceptions once execution is back in C++ code and they can be handled.

In this bit of code, `rethrow_exception()` throws an exception referred to by its `exception_ptr` argument. Thus, exceptions that are thrown from worker threads are transported across thread boundaries, and handled in catch blocks of another thread.

Thread local storage

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. VS2013 allows you to specify thread local storage using the non-standard `__declspec(thread)` specifier. The semantics of this specifier are similar to those of the standard `thread_local` storage class.

This storage attribute can only be applied to data declarations (not functions), and only those with static storage duration, i.e. global variables, file or namespace scope variables, static function variables and static class members:

```
__declspec(thread) B b1;

namespace {
    __declspec(thread) B b2;
}

class C
{
    // each thread gets a separate instance counter
    static __declspec(thread) int _instance_counter;
};

void f()
{
    // each thread invoking f() has a copy of run_count
    static __declspec(thread) int run_count;
}
```

For block scope variables, `static` is implied when using `thread_local`, however Visual Studio requires `static` to be specified explicitly.

Also, unlike `thread_local`, `__declspec(thread)` doesn't allow objects with constructors or destructors.

All the declarations and the definition of a thread local object must have the `__declspec(thread)` specifier. For example, the following will produce an error:

```
extern int i;
int __declspec(thread) i;           // error
```

`__declspec(thread)` must be specified in all declarations:

```
extern int __declspec(thread) i;
int __declspec(thread) i;           // OK
```

Non-standard behavior and bugs in Visual Studio 2013

If you combine thread local storage with custom alignment, e.g.

```
__declspec(thread) __declspec(align(16)) int aligned_val;
```

the specified alignment may not be honored.

Concurrency Support: Synchronization

When you use threads, in most cases they share some data, so you need to ensure that access to this data is synchronized. C++11 defines several types of synchronization primitives. Mutexes allow you to implement exclusive access to shared data from multiple threads. Locks help with locking and unlocking mutexes. Condition variables provide an efficient mechanism for making threads wait for events from other threads. Finally, the "call once" facility helps with the common situation where code needs to be executed only once in a multi-threaded context.

Mutexes

To prevent race conditions when accessing data shared by multiple threads, you need to provide each thread with access to a mutex object and ensure that it's locked and unlocked around the critical section:

```
mutex _speed_mutex;
// ...
_speed_mutex.lock();     // blocks if mutex is already locked
_speed = latest_speed;
_speed_mutex.unlock();
```

> **Note:**
>
> Include <mutex> to use mutexes and locks.

If you call `lock` on a mutex that's already locked by another thread, your thread will block until the mutex is unlocked. If you try to call `lock` twice on the same thread, Visual Studio will throw a `system_error` with error code `errc::device_or_resource_busy`. If you need a mutex that allows this behavior without causing exceptions, you can employ `recursive_mutex` instead:

```
void Logger::log_key_params()
{
```

```
    string s = get_key_params();
    _log_mutex.lock();
    _log << s;
    _log_mutex.unlock();
}

void Logger::log(const string& s)
{
    _log_mutex.lock();
    if (_always_log_key_params)
        log_key_params();
    _log << s;
    _log_mutex.unlock();
}
```

> **Note:**
>
> Destroying a mutex locked by any thread, or allowing a thread to terminate without unlocking mutexes it locked is undefined behavior.

In addition to `lock` and `unlock`, both `mutex` and `recursive_mutex` provide two other methods, `try_lock` and `native_handle`.

`try_lock` attempts to lock the mutex without blocking and immediately returns `false` if it can't:

```
while (!_speed_mutex.try_lock())  // note: busy waiting is mostly
                                  // a bad idea
    this_thread::sleep_for(milliseconds(50));

// ... do stuff in the critical section ...

_speed_mutex.unlock();
```

The standard allows spurious failures for this method, i.e. there is no guarantee it will lock the mutex even when it's possible. However, the implementation is supposed to make a strong effort to lock it. The standard doesn't provide this guarantee on purpose, as without it implementations based on compare-and-exchange are possible.

In the case of recursive mutexes, each call to `lock` or `try_lock` increases the level of ownership of the mutex by the thread. Unlocking the mutex requires calling `unlock` the same number of times as `lock`/`try_lock` to reduce the level of ownership back to zero.

The last method I'd like to mention - `native_handle` - simply returns the platform specific handle associated with the mutex. The type of the handle is implementation defined. In Visual Studio, it's `Concurrency::critical_section*` cast to `void*`.

Timed mutexes

Timed mutexes allow you to try obtaining the lock for a specified period of time. There are two versions: non-recursive (`timed_mutex`) and recursive (`recursive_timed_mutex`). There are two ways of specifying a time period:

```
if (!_speed_mutex.try_lock_for(milliseconds(50)))
    speed = estimate_speed();
else
    speed = _current_speed;
```

```
if (!_speed_mutex.try_lock_until(cutoff_point))
    speed = estimate_speed();
else
    speed = _current_speed;
```

Specifying a zero duration as the argument for `try_lock_for` or a time point in the past for `try_lock_until` will make them attempt to obtain ownership without blocking, same as `try_lock`. Spurious failures are also allowed for these functions.

> **Note:**
>
> The standard encourages implementations to detect deadlock and throw a `system_error` with error code `errc::resource_deadlock_would_occur` but that's not required.

Locking multiple mutexes

When you lock more than one mutex at a time, it creates a possibility of deadlock. To help prevent it, generic `lock` and `try_lock` functions are provided. `lock` uses a deadlock

avoidance algorithm to prevent deadlock. If it encounters a locked mutex or one of the `lock` calls in the sequence throws an exception, it unlocks all the previously locked mutexes and returns.

```
mutex m1, m2;

lock(m1, m2);
// ... do something ...
m1.unlock();
m2.unlock();
```

`try_lock` is the non-blocking alternative to `lock`. It calls `try_lock` on each argument in order (unlike `lock`, which can lock arguments in any order, as long as it doesn't cause deadlock). If any of the arguments can't be locked, it unlocks all the previous arguments and returns the index of the one that failed:

```
try_lock(m1, m2, m3); // m1, m2 & m3 are locked, returns -1

m1.unlock();
m3.unlock();

try_lock(m1, m2, m3); // m2 is still locked, so returns 1
```

Locks

`lock_guard` provides a simple application of the RAII pattern for mutexes:

```
{
    lock_guard<mutex> lock(_speed_mutex);

    // ... do work in the critical section ...

}   // the mutex is unlocked by ~lock_guard here
```

If you need to use mutex's `try_lock` method, you can still use a `lock_guard` to ensure it's unlocked later on:

```
if (!_speed_mutex.try_lock())
    return;
// _speed_mutex is now locked
lock_guard<mutex> lock(_speed_mutex, adopt_lock);
// ... do work in the critical section ...
```

This argument is also useful to combine locking multiple mutexes with RAII:

```
{
    lock(m1, m2);
    lock_guard<mutex> lock1(m1, adopt_lock);
    lock_guard<mutex> lock2(m2, adopt_lock);

    // ... do work in the critical section ...

}   // m1 & m2 get unlocked by ~lock_guard
```

> **Note:**
>
> If the mutex wrapped by `unique_lock` is destroyed before the lock object, it's undefined behavior.

`unique_lock` is a more flexible option provided by the standard. It has a variety of constructors which let you lock a mutex on construction or after construction:

```
// calls _speed_mutex.lock()
unique_lock<mutex> ul(_speed_mutex);

// calls _speed_mutex.try_to_lock()
unique_lock<mutex> ul(_speed_mutex, try_to_lock);

// doesn't lock the mutex
unique_lock<mutex> ul(_speed_mutex, defer_lock);

// adopts locked mutex
unique_lock<mutex> ul(_speed_mutex, adopt_lock);
```

For timed mutexes, you can also specify a timeout for trying to obtain a lock:

```
// calls _speed_mutex.try_to_lock_for(milliseconds(50))
unique_lock<timed_mutex> ul(_speed_mutex, milliseconds(50));

// calls _speed_mutex.try_to_lock_until(cutoff_point)
unique_lock<timed_mutex> ul(_speed_mutex, cutoff_point);
```

> **Note:**
>
> If you pass `m1` and `m2` to `try_lock` instead of `ul1` and `ul2`, it will cause a problem because `ul1` and `ul2`'s destructors will not try to unlock the mutexes in this situation.

Not locking the mutex in the constructor is useful if you want to combine `unique_lock` with locking multiple mutexes. As unique_lock provides `lock`, `try_lock` and `unlock` methods, it can be used as an argument to generic `lock` and `try_lock` functions:

```
unique_lock<mutex> ul1(m1, defer_lock);
unique_lock<mutex> ul2(m2, defer_lock);
try_lock(ul1, ul2);
```

When wrapping a timed mutex, `unique_lock` additionally allows `try_lock_for` and `try_lock_until` method calls.

The ownership of the mutex can be transferred to another `unique_lock` object (this behavior is similar to `unique_ptr`):

```
unique_lock<mutex> ul2 = move(ul);
```

There are several other operations available:

```
unique_lock<mutex> ul(_speed_mutex);

if (ul)
    cout << "Locked" << endl;
ul.owns_lock(); // returns true
ul.mutex();     // returns &_speed_mutex
ul.release();   // returns &_speed_mutex; ul no longer wraps
                // _speed_mutex
```

After calling `release` it's your responsibility to unlock `_speed_mutex`.

call_once

In some situations you need to make sure that a piece of code is only executed once in a multi-threaded context (e.g. some initialization code). To make it easier, the standard provides the `call_once` function which can perform this task without creating race conditions:

```
class Logger
{
    once_flag _log_file_flag;

    // ...

    void log(const string& s)
    {
        call_once(_log_file_flag, &Logger::prepare_log_file, this);
        // ... write to log ...
    }
};
```

`call_once` propagates any exception thrown by the function it executes. If this happens, the call is unsuccessful and the next invocation of `call_once` may complete instead.

> **Note:**
>
> Include `<condition_variable>` to use condition variables.

`call_once` uses the `once_flag` structure to prevent race conditions. If more than one thread calls `call_once` simultaneously, only one of them will proceed and the other threads will block. Note that you can pass different functions to `call_once` with the same `once_flag` instance. Only one of them will be executed.

Condition variables

Condition variables provide a way for threads to wait efficiently for an event on another thread by blocking until a condition is satisfied. There are two kinds of notification possible: notifying one of the waiting threads selected at random, or notifying all of them. Here is how you would notify one thread:

```
bool data_chunk_ready(false);
mutex cond_mutex;
condition_variable cond_var;

thread t([&]
{
    // ... work long hours to produce data chunk ...
    {
        lock_guard<mutex> lock(cond_mutex);
        data_chunk_ready = true;
    }
    cond_var.notify_one();
    // ... continue working on the next data chunk ...
});

// ... wait for worker thread to signal that the first data chunk
// is ready ...
{
    unique_lock<mutex> ul(cond_mutex);
    cond_var.wait(ul, [&] { return data_chunk_ready; });
}
// ... do things with data chunk ...
```

> **Note:**
>
> `futures` solve one specific use case that overlaps with condition variables: notifying a thread or threads *once* that another thread has produced a result. Condition variables give you a lot more flexibility.

There are a few things about the above example that require explanation.

The condition variable doesn't itself hold the condition, it merely signals the change. As you can see, condition variables are designed to work in tandem with a mutex (which you need to synchronize access to shared data anyway). The `wait` method unlocks the mutex and blocks the thread until the condition variable is signalled. On receiving the signal, it wakes the thread and re-locks the mutex.

Sometimes the signal can be a false positive. This is called a *spurious wakeup* and is allowed by the standard in order to provide more freedom for the underlying system

implementation. The predicate passed to `wait` ensures that the execution only continues when the condition is actually satisfied. It's equivalent to

```
while (!pred())
    cond_var.wait(ul);
```

There is in fact a `wait` method overload which doesn't take a predicate argument, so you could write this loop explicitly if you wanted.

C++11 additionally requires the mutex used with a condition variable to be wrapped in a `unique_lock`. This is done to ensure exception safety and simplify the code you need to write to use a condition variable correctly.

If you want to broadcast a notification to all waiting threads instead of one of them, use `notify_all` instead of `notify_one`:

```
cond_var.notify_all();
```

If you need to, you can use multiple condition variables with one mutex. For example, consider an application with a queue of tasks served by a pool of worker threads. You could have one condition variable to notify threads that a new task is available, and another one to notify a server managing thread when the queue is getting full so that it can spin up another server to help out:

```
{
    lock_guard<mutex> lock(queue_mutex);
    queue.push_back(task);
    if (queue.size() > full_threshold)
        full_queue_cond.notify_one();
}
new_task_cond.notify_all();

// ------------------------------------------------------------------
// worker thread
{
    unique_lock<mutex> ul(queue_mutex);
    new_task_cond.wait(ul, [&] { return !queue.empty(); });
}
```

```
// ... handle the task ...

// ------------------------------------------------------------------
// server manager thread
{
    unique_lock<mutex> ul(queue_mutex);
    full_queue_cond.wait(ul,
        [&] { return queue.size() > full_threshold; });
}
// ... spin up another server ...
```

Limiting wait time

You have the ability to limit the time a thread waits to be signalled, either by specifying a timeout or a time point to wait until:

```
auto status = cond_var.wait_for(ul, seconds(1));

cond_var.wait_until(ul, cutoff_point);
```

These functions return `cv_status::timeout` if the specified time has elapsed or `cv_status::no_timeout` otherwise. There are also overloads that take a condition predicate as an additional argument. They return the current value of the predicate instead. `wait_until` with a predicate is equivalent to

```
while (!pred())
    if (wait_until(lock, abs_time) == cv_status::timeout)
        return pred();
return true;
```

> **Note:**
>
> `condition_variable` also provides the `native_handle` method which returns a pointer to one of the Concurrency Runtime internal data structures.

`wait_for` with a predicate is equivalent to

```
wait_until(lock, chrono::steady_clock::now() + period, move(pred));
```

Other lockable types

If you need to combine a condition variable with something other than a mutex wrapped in a `unique_lock` (which could be your own lock type), you can use the `condition_variable_any` adaptor:

```
condition_variable_any cond_var;
mutex cond_mutex;

cond_mutex.lock();
cond_var.wait(cond_mutex, [&] { return cond_flag; });
// ... do stuff ...
cond_mutex.unlock();
```

> **Note:**
>
> This example has issues with exception safety. It's only shown here to illustrate that `condition_variable_any` doesn't require `unique_lock`.

`condition_variable` is limited to working with `unique_lock<mutex>` to ensure maximum performance, while `condition_variable_any` trades some of that performance for flexibility.

If you use a lock type which isn't `unique_lock` or one of the standard mutex types, you need to provide synchronization for the predicate associated with `condition_variable_any`. Your lock type must provide synchronized `lock` and `unlock` methods.

`notify_all_at_thread_exit`

This function lets you notify all threads waiting on a condition variable upon thread exit:

```
thread t([&] {
    // ...
    unique_lock<mutex> ul(cond_mutex);
```

```
    is_ready = true;
    notify_all_at_thread_exit(cond_var, move(ul));
});

// wait for notification that t has exited
{
    unique_lock<mutex> ul(cond_mutex);
    cond_var.wait(ul, [&] { return is_ready; });
}
```

The calling thread has to lock the lock before calling `notify_all_at_thread_exit`. Calling the function results in the thread unlocking the lock and calling `notify_all` on the condition variable once thread-local variables have been destroyed.

As the lock is held until thread exit, you need to be careful to make sure it doesn't result in deadlock. The standard recommends that after calling this function no blocking or time-consuming tasks are executed on the thread, and that the thread exits as soon as possible. Also remember that the waiting threads can wake up spuriously, just like with other uses of condition variables, so you still need a way of checking (e.g. a flag) that the signalling thread has indeed terminated.

A note on `const` and `mutable`

It's worth noting that in C++11, the definition of `const` has thread safety implications. 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 a potential for race conditions.

Non-standard behavior and bugs in Visual Studio 2013

The `unique_lock` implementation in Visual Studio isn't fully standard compliant. Its methods (`lock`, `try_lock` etc.) don't check whether its internal mutex

pointer is null and consequently don't throw a `system_error` with error code `errc::operation_not_permitted`. Neither do they throw a `system_error` with error code `errc::resource_deadlock_would_occur` when the mutex is already locked. Instead, they propagate the exception thrown by the mutex's `lock` method.

`notify_one` has a bug in Visual Studio. If you get an access violation in `Concurrency::DeleteAsyncTimerAndUnloadLibrary` during a call to `notify_one`, then you've hit this bug. It will be fixed in the next release of the standard library.

Concurrency Support: Atomic Data Types and Operations

The standard includes a number of atomic types and operations which don't require extra synchronization in a multi-threaded context. The operations on these types are sequentially consistent by default, i.e. they are guaranteed to be executed in the order written. C++11 also provides a way to relax the memory ordering constraints.

> **Note:**
>
> Include `<atomic>` to use atomic types and operations.

Atomics can be cheaper than mutexes but the tradeoff is that they are harder to get right. You have to be really clear about the benefits if you decide to use them. However, there are also a few situations where a simple use of atomics allows you to dispense with explicit synchronization and make your code clearer (e.g. a counter or a flag used by multiple threads).

Atomic data types

The types are defined via the `atomic` template. This template is fully specialized for integral types and `bool`, and partially specialized for pointer types. Any other type arguments have to be trivially copyable.

Using atomic data types can be as easy as wrapping your type declaration in `atomic<>`:

```
atomic<int> cnt(0);
vector<thread> v(1000);

for (auto& t: v)
    t = thread([&] { ++cnt; });

for (auto& t: v)
    t.join();

cout << "Count: " << cnt << endl;  // outputs 1000
```

The construction of an atomic type variable isn't thread-safe.

In addition to automatic conversion to the corresponding non-atomic type (seen in the output operation in the previous example), the `atomic` template provides several synchronized operations:

```
cnt.load();           // returns a copy of the value
cnt.store(10);        // stores the supplied value in cnt
cnt.exchange(1000);   // stores 1000 and returns the previous value (10)
cnt.is_lock_free();   // true if the object's operations are lock-free
cnt = 20;             // assigns new value and returns a copy of it
```

> **Note:**
>
> All of the methods with the exception of constructors are overloaded for both `volatile` and non-`volatile` objects.

It also provides a compare-and-exchange operation which only performs the exchange if the `atomic` value matches the `expected` argument:

```
atomic<int> cnt(20);
int expected = 10;
cnt.compare_exchange_strong(expected, 30);    // returns false,
                                              // cnt is still 20

expected = 20;
cnt.compare_exchange_strong(expected, 30);    // returns true,
                                              // cnt is now 30
```

Additionally, the standard includes a weak version of compare-and-exchange called `compare_exchange_weak`. This version may fail spuriously so it almost always needs to be used in a loop. It's included for broader compatibility with hardware and may provide better performance than the strong version when used in a loop.

For integral types, the `atomic` template additionally provides some convenient operators:

```
++cnt;
cnt++;
--cnt;
```

```
cnt--;

cnt += 2;   // same as cnt.fetch_add(2)
cnt -= 2;   // same as cnt.fetch_sub(2)

cnt &= 1;   // same as cnt.fetch_and(1)
cnt |= 1;   // same as cnt.fetch_or(1)
cnt ^= 1;   // same as cnt.fetch_xor(1)
```

> **Note:**
>
> The standard notes that the arithmetic operations have no undefined behavior: *"For signed integer types, arithmetic is defined to use two's complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior."*

These operations return a copy of the value immediately before the operation.

For pointer types, only a subset of arithmetic operations is possible: `fetch_add`, `fetch_sub` and the operators implemented in terms of those (++, --, +=, -=). These operations act on pointers, not on values they point to.

Alternative C-style interface

The standard types have non-templated equivalents for compatibility with C:

```
atomic_bool flag;
atomic_int cnt;
```

In addition to `atomic_bool`, the supported integral types are:

Atomic type	Corresponding non-atomic type
`atomic_char`	`char`
`atomic_schar`	`signed char`
`atomic_uchar`	`unsigned char`
`atomic_short`	`short`
`atomic_ushort`	`unsigned short`
`atomic_int`	`int`

Atomic type	Corresponding non-atomic type
atomic_uint	unsigned int
atomic_long	long
atomic_ulong	unsigned long
atomic_llong	long long
atomic_ullong	unsigned long long
atomic_char16_t	char16_t
atomic_char32_t	char32_t
atomic_wchar_t	wchar_t

Additionally, there are a number of types corresponding to the <inttypes.h> typedefs:

Atomic type	Corresponding <inttypes.h> typedef
atomic_int_least8_t	int_least8_t
atomic_uint_least8_t	uint_least8_t
atomic_int_least16_t	int_least16_t
atomic_uint_least16_t	uint_least16_t
atomic_int_least32_t	int_least32_t
atomic_uint_least32_t	uint_least32_t
atomic_int_least64_t	int_least64_t
atomic_uint_least64_t	uint_least64_t
atomic_int_fast8_t	int_fast8_t
atomic_uint_fast8_t	uint_fast8_t
atomic_int_fast16_t	int_fast16_t
atomic_uint_fast16_t	uint_fast16_t
atomic_int_fast32_t	int_fast32_t
atomic_uint_fast32_t	uint_fast32_t

Atomic type	Corresponding `<inttypes.h>` typedef
`atomic_int_fast64_t`	`int_fast64_t`
`atomic_uint_fast64_t`	`uint_fast64_t`
`atomic_intptr_t`	`intptr_t`
`atomic_uintptr_t`	`uintptr_t`
`atomic_size_t`	`size_t`
`atomic_ptrdiff_t`	`ptrdiff_t`
`atomic_intmax_t`	‘intmax_t“
`atomic_uintmax_t`	`uintmax_t`

For compatibility with C, atomic types have a default constructor, which is supposed to be followed by `atomic_init` (also not thread-safe, same as the non-default constructor):

```
atomic_int cnt;
atomic_init(&cnt, 0);
```

> **Note:**
>
> Non-member atomic functions are overloaded for `shared_ptr` as it can guarantee atomicity. This is the only standard type other than the atomic family which can be used with these functions.

The C-style types should behave the same as the corresponding atomic template instantiations. However, in Visual Studio they are missing non-default constructors, so code like this fails to compile:

```
atomic_bool flag(false);  // compile error
```

The methods of the `atomic` template have their non-member equivalents with names prefixed by `atomic_`:

```
atomic_store(&cnt, 10);
atomic_fetch_add(&cnt, 100);
```

Atomic flag

`atomic_flag` is a special atomic boolean type which is guaranteed to have lock-free operations. It has two states, *set* and *clear*:

```
atomic_flag flag = ATOMIC_FLAG_INIT;
flag.test_and_set();  // returns false, flag is now set
flag.test_and_set();  // returns true
flag.clear();         // flag is now clear
```

> **Note:**
>
> Member functions have overloads that take a `memory_order` argument, and non-member functions have versions with `_explicit` suffix.

The default constructor initializes the flag to an unspecified state. You can use the `ATOMIC_FLAG_INIT` macro to initialize the flag to clear state. Corresponding C-style operations are provided, with names prefixed with `atomic_flag_`.

Low level atomic interface

Most of the operations on atomic types have additional versions which allow you to specify a memory synchronization order:

```
atomic<int> cnt(0);
auto val = cnt.load(memory_order_relaxed);
atomic_store_explicit(&cnt, 10, memory_order_relaxed);
```

The default memory order is `memory_order_seq_cst`, which guarantees the operations to be sequentially consistent. Other orders provide ways of relaxing the sequential guarantees and potentially getting extra performance. This is how the standard defines them in the `memory_order enum`:

Memory order	Description
`memory_order_seq_cst`	Default; guarantees sequential consistency; a store operation performs a release operation on the affected memory location; a load operation performs an acquire operation on the affected memory location
`memory_order_relaxed`	No operation orders memory
`memory_order_consume`	A load operation performs a consume operation on the affected memory location
`memory_order_acquire`	A load operation performs an acquire operation on the affected memory location
`memory_order_release`	A store operation performs a release operation on the affected memory location
`memory_order_acq_rel`	A store operation performs a release operation on the affected memory location; a load operation performs an acquire operation on the affected memory location

The operations remain atomic regardless of the memory order, but their order may not be guaranteed when you use an order value other than `memory_order_seq_cst`. It's challenging to avoid race conditions in code with relaxed memory ordering so you should really know what you are doing. It's beyond the scope of this book to cover this subject in detail.

> **Note:**
>
> Herb Sutter has given an in-depth talk about the C++ memory model, synchronization and relaxed atomics: http://isocpp.org/blog/2013/02/atomic-weapons-the-c-memory-model-and-modern-hardware-herb-sutter

Fences

A fence is a memory synchronization primitive which enforces the order of load and store operations before and after the fence. Fences can have acquire or release semantics. C++11 provides two functions to establish memory fences without an associated atomic operation:

```
atomic_thread_fence(memory_order_acquire);

atomic_signal_fence(memory_order_release);
```

The `memory_order` argument of these functions allows you to specify fence semantics:

Memory order	Effects
`memory_order_seq_cst`	Sequentially consistent acquire and release fence
`memory_order_relaxed`	No effect
`memory_order_consume`	An acquire fence
`memory_order_acquire`	An acquire fence
`memory_order_release`	A release fence
`memory_order_acq_rel`	Both an acquire and a relase fence

The difference between these functions is that `atomic_signal_fence` doesn't result in hardware fence instructions. However, compiler optimizations and reordering of load and store operation is constrained in the same way as with `atomic_thread_fence`.

`atomic_signal_fence` establishes ordering constraints only between a thread and a signal handler which is executed in the same thread.

Non-standard behavior and bugs in Visual Studio 2013

The C-style atomic types should behave the same as the corresponding atomic template instantiations. However, in Visual Studio they are missing non-default constructors, so code like this fails to compile:

```
atomic_bool flag(false);  // compile error
```

The `atomic` template is unable to handle `const` pointers:

```
auto n = 10;
atomic<int const *> atomic_p;
atomic_p.store(&n);     // compile error
```

Miscellaneous Features

There are a few new features left to talk about which don't fit into a particular category. They are discussed in this chapter.

Alignment

VS2013 doesn't implement the `alignas` and `alignof` keywords, however it has `aligned_storage`, `aligned_union` and `alignment_of` type traits (which are described in the Compile Time Assertions and Type Traits chapter). It also provides `std::align` to compute a pointer with a particular alignment at runtime. Note that the function also updates the last two parameters:

```
void* p = (void*)0x1;

// try to align 200 bytes to the 32 byte boundary
// with 230 bytes of storage space available
size_t space = 230;
void* res = align(32, 200, (void*&)p, space);
// res is null as 31 bytes are needed for alignment
// but at most 30 are available

space = 256;  // increase available space to 256 bytes
res = align(32, 200, (void*&)p, space);
// res is now 0x20 (aligned to 32 byte boundary)
// p is now also 0x20
// space is now 225 to reflect the available space after alignment
```

After the call to `align`, the pointer passed to the function points to the first available aligned address. The last argument is updated to reflect the amount of space available after alignment.

`addressof` template

This template allows you to get the address of an object even if it has an overloaded `operator&`. For example, consider a class that tries to provide transparent access to a member pointer:

```
template<typename T>
struct SmartPtr
{
    T* _ptr;
    /* ... */
    T** operator&()
    {
        return &_ptr;
    }
};
```

If you need to get the address of a `SmartPtr<T>` instance `p`, `&p` isn't going to work as it will return the address of `_ptr`. However, `addressof(p)` is going to return the address of `p`.

> **Note:**
>
> `addressof` is defined in `<memory>`.

Using `type_info` in containers

As `type_info` doesn't have a public copy constructor or assignment operator, it can't be used in STL containers. The new `type_index` class is a thin wrapper around a `type_info` pointer which solves this problem (it can be copied and assigned). There is also a `std::hash` specialization for `type_index` to allow it to be used as a key in hash table based containers.

`bitset` and `valarray` improvements

`bitset` has two new methods: `all()`, which is the opposite of `none()`, and `to_ullong()` for conversion to an `unsigned long long` value.

valarray now has an std::swap() specialization, as well as specializations for non-member begin() and end().

New stream functionality

There are two new manipulators for floating point values. hexfloat is equivalent to applying fixed and scientific together. defaultfloat resets floating point value formatting to the default.

In addition, two sets of extended manipulators have been added. get_money/put_money for parsing and outputting currency values, and get_time/put_time for parsing and outputting time values. They use previously existing facets (e.g. money_get and money_put).

system_error header

This header provides the system_error exception class (derived from runtime_error) which is used by the standard library to report errors from the OS or other low level APIs.

It also includes the errc enum with symbolic names for the POSIX error codes defined in <errno.h>. The class error_code is used to hold implementation specific error codes, and the class error_condition is used to hold portable abstractions describing error conditions. Objects of these classes can be generated by passing values from the errc enum to make_error_code() and make_error_condition() functions.

C99 compatibility

VS2013 supports some of the C99 features:

- includes the typedefs and #define's from <stdint.h> (via <cstdint>)
- __func__ isn't available but there is __FUNCTION__ instead
- variadic macros are supported
- long long int (>= 64 bit) is available
- __pragma and __restrict are supported

Deprecated and Future Features

In this chapter I'd like to list pre-C++11 features that are now deprecated, and give you a glimpse of what to expect in the future as more of the new standard is implemented.

Deprecated features

The deprecated features are still supported but may disappear completely from the next version of the standard.

`auto_ptr` has been deprecated, with `unique_ptr` provided as an alternative.

`bind1st`, `binder1st`, `bind2nd`, `binder2nd` are deprecated (they are superseded by the more general bind).

`unary_function`, `binary_function`, as well as `ptr_fun`, `mem_fun`, `mem_fun_ref` and their corresponding return types are all deprecated. `bind` and `function` provide better alternatives.

`auto` can no longer be used as a storage class specifier. It's now used either to specify type inference or in functions with trailing return types.

`export` specifier for templates is no longer supported. `export` is still defined as a keyword though.

Dynamic exception specifications (via `throw()`) are deprecated, with `noexcept` taking their place. `noexcept(const_bool_expr)` specifies whether a function does or doesn't throw exceptions. `terminate()` is called if an exception propagates outside a `noexcept(true)` function. However, `noexcept` isn't yet supported by Visual Studio.

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.

Beyond Visual Studio 2013

There is a number of other changes to the C++ language which won't be available until some time in the future. First, let me list the C++11 features.

The *constexpr mechanism* will generalize constant expressions to include functions and user defined literals.

The language will support UTF-8, UTF-16 and UTF-32 encodings. You'll be able to create Unicode string literals for each of these encodings using new literal prefixes. New types `char16_t` and `char32_t` will provide a standard way to store Unicode characters.

User defined literals will allow you to use literals of user defined types. They will be similar to literals of built-in types:

```
1.2_i;  // express complex numbers
10_km;  // express units
```

They will be supported via *literal operators*:

```
// create a 'complex' instance from an imaginary literal
constexpr complex<double> operator "" _i(long double d)
{
    return complex<double>(0, d);
}
```

Automatically generated move constructors and move assignment operators will simplify the implementation of move semantics in certain cases.

Inheriting constructors will allow you to make base class constructors available in a derived class (the same way you could already make member functions available):

```
class Derived : public Base
{
public:
    // C++98/03: make all Base::f() overloads available in Derived's
    // scope
```

```
    using Base::f;

    // C++11: make Base constructors available in Derived's scope
    using Base::Base;
};
```

Member function reference qualifiers will allow overloading member functions based on whether `*this` as an lvalue or an rvalue.

Arbitrary expressions will be allowed in template deduction contexts, and the compiler will treat most of them as SFINAE failures, which is useful because of the introduction of `decltype` and `constexpr`, which can appear in template deduction context.

Attributes will provide a standard mechanism for including vendor specific or optional information (e.g. `__declspec` could become an attribute). They are denoted with double square brackets. One example of an attribute included in the standard is `[[noreturn]]` used to indicate that a function never returns.

`sizeof` will be extended so it can be applied to non-static data members without an object.

Inline namespaces will provide a way to implement versioning.

The restrictions on unions will change to make more member types feasible. Members of types with constructors and destructors will be allowed. At the same time, restrictions will be added to encourage discriminated unions.

Finally, you'll be able to specify whether a function throws exceptions with the use of `noexcept`.

Beyond that, work is under way on the next version of the standard, tentatively referred to as C++14. The next section provides an overview of the features expected to make their way into that version of the standard.

C++14

As you've learned from the previous chapters, Visual Studio already implements a few of the C++14 changes. 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.

I'm going to highlight some of the new language features in the draft of the standard.

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 VS2013 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 which 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 in `constexpr` functions 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 Technical Specifications currently under way include Filesystem TS, Networking TS and Concepts TS.

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 the 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. From the previous 29 chapters, you've learned all of the C++11 and C++14 features added to Visual Studio 2013. You've also learned which features have been deprecated or removed from the language. Finally, you got a glimpse of the new features planned for C++14 which are not available in Visual Studio so far.

Even though VS2013 has a number of deviations from the C++ standard, it still provides a large number of ways to improve the readability, performance and quality of your code, and I hope that you will incorporate your new knowledge into your projects.

I hope this book has given you confidence to use the new features of 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.