C++ Concurrency IN ACTION

Practical Multithreading

Anthony Williams

MANNING

Table of Contents

appendix A
Brief reference for some C++11 language features

The new C++ Standard brings more than just concurrency support; there are a whole host of other language features and new libraries as well. In this appendix I give a brief overview of the new language features that are used in the Thread Library and the rest of the book. Aside from `thread_local` (which is covered in section A.8), none of them are directly related to concurrency, though they are important and/or useful for multithreaded code. I've limited this list to those that are either necessary (such as rvalue references) or serve to make the code simpler or easier to understand. Code that uses these features may be difficult to understand at first because of lack of familiarity, but as you become familiar with them, they should generally make code easier to understand rather than harder. As the use of C++11 becomes more widespread, code making use of these features will become more common.

Without further ado, let's start by looking at *rvalue references*, which are used extensively by the Thread Library to facilitate transfer of ownership (of threads, locks, or whatever) between objects.

A.1 Rvalue references

If you've been doing C++ programming for any time, you'll be familiar with references; C++ references allow you to create a new name for an existing object. All accesses and modifications done through the new reference affect the original, for example:

```
int var=42;
int& ref=var;
ref=99;
assert(var==99);
```

Create a reference to var

Original updated because of assignment to reference

The references that we've all been using up to now are *lvalue references*—references to lvalues. The term *lvalue* comes from C and refers to things that can be on the left side of an assignment expression—named objects, objects allocated on the stack or heap, or members of other objects—things with a defined storage location. The term *rvalue* also comes from C and refers to things that can occur only on the right side of an assignment expression—literals and temporaries, for example. Lvalue references can only be bound to lvalues, not rvalues. You can't write

```
int& i=42;      <-- Won't compile
```

for example, because `42` is an rvalue. OK, that's not quite true; you've always been able to bind an rvalue to a `const` lvalue reference:

```
int const& i=42;
```


But this is a deliberate exception on the part of the standard, introduced before we had rvalue references in order to allow you to pass temporaries to functions taking references. This allows implicit conversions, so you can write things like this:

```
void print(std::string const& s);
print("hello");      <-- Create temporary std::string object
```

Anyway, the C++11 Standard introduces *rvalue references*, which bind *only* to rvalues, not to lvalues, and are declared with two ampersands rather than one:

```
int&& i=42;
int j=42;
int&& k=j;      <-- Won't compile
```

You can thus use function overloading to determine whether function parameters are lvalues or rvalues by having one overload take an lvalue reference and another take an rvalue reference. This is the cornerstone of *move semantics*.

A.1.1 Move semantics

Rvalues are typically temporary and so can be freely modified; if you know that your function parameter is an rvalue, you can use it as temporary storage, or "steal" its contents without affecting program correctness. This means that rather than *copying* the contents of an rvalue parameter, you can just *move* the contents. For large dynamic structures, this saves a lot of memory allocation and provides a lot of scope for optimization. Consider a function that takes a `std::vector<int>` as a parameter and needs to have an internal copy for modification, without touching the original. The old way of doing this would be to take the parameter as a `const` lvalue reference and make the copy internally:

```
void process_copy(std::vector<int> const& vec_)
{
     std::vector<int> vec(vec_);
     vec.push_back(42);
}
```

This allows the function to take both lvalues and rvalues but forces the copy in every case. If you overload the function with a version that takes an rvalue reference, you can avoid the copy in the rvalue case, because you know you can freely modify the original:

```
void process_copy(std::vector<int> && vec)
{
     vec.push_back(42);
}
```

Now, if the function in question is the constructor of your class, you can pilfer the innards of the rvalue and use them for your new instance. Consider the class in the following listing. In the default constructor it allocates a large chunk of memory, which is freed in the destructor.

Listing A.1 A class with a move constructor

```
class X
{
private:
    int* data;
public:
    X():
        data(new int[1000000])
    {}
    ~X()
    {
        delete [] data;
    }
    X(const X& other):                    <-- 1
        data(new int[1000000])
    {
        std::copy(other.data,other.data+1000000,data);
    }
    X(X&& other):                         <-- 2
        data(other.data)
    {
        other.data=nullptr;
    }
};
```

The *copy constructor* ❶ is defined just as you might expect: allocate a new block of memory and copy the data across. However, you also have a new constructor that takes the old value by rvalue reference ❷. This is the *move constructor*. In this case you just copy the *pointer to* the data and leave the `other` instance with a null pointer, saving yourself a huge chunk of memory and time when creating variables from rvalues.

For class `X` the move constructor is just an optimization, but in some cases it makes sense to provide a move constructor even when it doesn't make sense to provide a copy constructor. For example, the whole point of `std::unique_ptr<>` is that each non-null instance is the one and only pointer to its object, so a copy constructor makes no sense. However, a move constructor allows ownership of the pointer to be

transferred between instances and permits `std::unique_ptr<>` to be used as a function return value—the pointer is *moved* rather than *copied*.

If you wish to explicitly move from a named object that you know you'll no longer use, you can cast it to an rvalue either by using `static_cast<X&&>` or by calling `std::move()`:

```
X x1;
X x2=std::move(x1);
X x3=static_cast<X&&>(x2);
```

This can be beneficial when you wish to move the parameter value into a local or member variable without copying, because although an rvalue reference parameter can bind to rvalues, within the function itself it is treated as an lvalue:

```
void do_stuff(X&& x_)
{
    X a(x_);                  <-- Copies
    X b(std::move(x_));       <-- Moves
}
do_stuff(X());                <-- OK, rvalue binds to rvalue reference
X x;
do_stuff(x);                  <-- Error, lvalue can't bind to rvalue reference
```

Move semantics are used extensively in the Thread Library, both where copies make no semantic sense but resources can be transferred, and as an optimization to avoid expensive copies where the source is going to be destroyed anyway. You saw an example of this in section 2.2 where we used `std::move()` to transfer a `std::unique_ptr<>` instance into a newly constructed thread, and then again in section 2.3 where we looked at transferring ownership of threads between `std::thread` instances.

None of `std::thread`, `std::unique_lock<>`, `std::future<>`, `std::promise<>`, or `std::packaged_task<>` can be copied, but they all have move constructors to allow the associated resource to be transferred between instances and support their use as function return values. `std::string` and `std::vector<>` both can be copied as always, but they also have move constructors and move-assignment operators to avoid copying large quantities of data from an rvalue.

The C++ Standard Library never does anything with an object that has been explicitly moved into another object, except destroy it or assign *to* it (either with a copy or, more likely, a move). However, it's good practice to ensure that the invariant of the class encompasses the moved-from state. A `std::thread` instance that has been used as the source of a move is equivalent to a default-constructed `std::thread` instance, for example, and an instance of `std::string` that has been used as the source of a move will still have a valid state, although no guarantees are made as to what that state is (in terms of how long the string is or what characters it contains).

A.1.2 Rvalue references and function templates

There's a final nuance when you use rvalue references for parameters to a function template: if the function parameter is an rvalue reference to a template parameter,

automatic template argument type deduction deduces the type to be an lvalue reference if an lvalue is supplied or a plain unadorned type if an rvalue is supplied. That's a bit of a mouthful, so let's look at an example. Consider the following function:

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

If you call it with an rvalue as follows, then `T` is deduced to be the type of the value:

```
foo(42);                  <-- Calls foo<int>(42)
foo(3.14159);             <-- Calls foo<double>(3.14159)
foo(std::string());       <-- Calls foo<std::string>(std::string())
```

However, if you call `foo` with an *lvalue*, `T` is deduced to be an lvalue reference:

```
int i=42;
foo(i);                   <-- Calls foo<int&>(i)
```

Because the function parameter is declared `T&&`, this is therefore a reference to a reference, which is treated as just the original reference type. The signature of `foo<int&>()` is thus

```
void foo<int&>(int& t);
```

This allows a single function template to accept both lvalue and rvalue parameters and is used by the `std::thread` constructor (sections 2.1 and 2.2) so that the supplied callable object can be moved into internal storage rather than copied if the parameter is an rvalue.

A.2 Deleted functions

Sometimes it doesn't make sense to allow a class to be copied. `std::mutex` is a prime example of this—what would it mean if you did copy a mutex? `std::unique_lock<>` is another—an instance is the one and only owner of the lock it holds. To truly copy it would mean that the copy also held the lock, which doesn't make sense. Moving ownership between instances, as described in section A.1.2, makes sense, but that's not copying. I'm sure you've met other examples.

The standard idiom for preventing copies of a class used to be to declare the copy constructor and copy assignment operator private and then not provide an implementation. This would cause a compile error if any code outside the class in question tried to copy an instance and a link-time error (due to lack of an implementation) if any of the class's member functions or friends tried to copy an instance:

```
class no_copies
{
public:
    no_copies(){}
private:
    no_copies(no_copies const&);
    no_copies& operator=(no_copies const&);
};
```

❶ No implementation

```
no_copies a;
no_copies b(a);
```

❷ Won't compile

With C++11, the committee realized that this was a common idiom but also realized that it's a bit of a hack. The committee therefore provided a more general mechanism that can be applied in other cases too: you can declare a function as *deleted* by adding `= delete` to the function declaration. `no_copies` can thus be written as

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

This is much more descriptive than the original code and clearly expresses the intent. It also allows the compiler to give more descriptive error messages and moves the error from link time to compile time if you try to perform the copy within a member function of your class.

If, as well as deleting the copy constructor and copy-assignment operator, you also explicitly write a move constructor and move-assignment operator, your class becomes move-only, the same as `std::thread` and `std::unique_lock<>`. The following listing shows an example of such a move-only type.

Listing A.2 A simple move-only type

```
class move_only
{
    std::unique_ptr<my_class> data;
public:
    move_only(const move_only&) = delete;
    move_only(move_only&& other):
        data(std::move(other.data))
    {}
    move_only& operator=(const move_only&) = delete;
    move_only& operator=(move_only&& other)
    {
        data=std::move(other.data);
        return *this;
    }
};
move_only m1;
move_only m2(m1);
move_only m3(std::move(m1));
```

Error, copy constructor is declared deleted

OK, move constructor found

Move-only objects can be passed as function parameters and returned from functions, but if you wish to move from an lvalue, you always have to be explicit and use `std::move()` or a `static_cast<T&&>`.

You can apply the `= delete` specifier to any function, not just copy constructors and assignment operators. This makes it clear that the function isn't available. It does

a bit more than that too, though; a deleted function participates in overload resolution in the normal way and only causes a compilation error if it's selected. This can be used to remove specific overloads. For example, if your function takes a `short` parameter, you can prevent narrowing of `int` values by writing an overload that takes an `int` and declaring it deleted:

```
void foo(short);
void foo(int) = delete;
```

Any attempts to call `foo` with an `int` will now be met with a compilation error, and the caller will have to explicitly cast supplied values to `short`:

```
foo(42);                    <-- Error, int overload declared deleted
foo((short)42);             <-- OK
```

A.3 Defaulted functions

Whereas deleted functions allow you to explicitly declare that a function isn't implemented, *defaulted* functions are the opposite extreme: they allow you to specify that the compiler should write the function for you, with its "default" implementation. Of course, you can only do this for functions that the compiler can autogenerate anyway: default constructors, destructors, copy constructors, move constructors, copy-assignment operators, and move-assignment operators.

Why would you want to do that? There are several reasons why you might:

- *In order to change the accessibility of the function*—By default, the compiler-generated functions are `public`. If you wish to make them `protected` or even `private`, you must write them yourself. By declaring them as defaulted, you can get the compiler to write the function *and* change the access level.
- *As documentation*—If the compiler-generated version is sufficient, it might be worth explicitly declaring it as such so that when you or someone else looks at the code later, it's clear that this was intended.
- *In order to force the compiler to generate the function when it would not otherwise have done so*—This is typically done with default constructors, which are only normally compiler generated if there are no user-defined constructors. If you need to define a custom copy constructor (for example), you can still get a compiler-generated default constructor by declaring it as defaulted.
- *In order to make a destructor virtual while leaving it as compiler generated.*
- *To force a particular declaration of the copy constructor, such as having it take the source parameter by a non-*`const` *reference rather than by a* `const` *reference.*
- *To take advantage of the special properties of the compiler-generated function, which are lost if you provide an implementation*—More on this in a moment.

Just as deleted functions are declared by following the declaration with `= delete`, defaulted functions are declared by following the declaration by `= default`, for example:

```
class Y
{
```

```
private:
    Y() = default;
public:
    Y(Y&) = default;
    T& operator=(const Y&) = default;
protected:
    virtual ~Y() = default;
};
```

Change access

Take a non-const reference

Declare as defaulted for documentation

Change access and add virtual

I mentioned previously that compiler-generated functions can have special properties that you can't get from a user-defined version. The biggest difference is that a compiler-generated function can be *trivial*. This has a few consequences, including the following:

- Objects with trivial copy constructors, trivial copy assignment operators, and trivial destructors can be copied with `memcpy` or `memmove`.
- Literal types used for `constexpr` functions (see section A.4) must have a trivial constructor, copy constructor, and destructor.
- Classes with a trivial default constructor, copy constructor, copy assignment operator, and destructor can be used in a union with a user-defined constructor and destructor.
- Classes with trivial copy assignment operators can be used with the `std::atomic<>` class template (see section 5.2.6) in order to provide a value of that type with atomic operations.

Just declaring the function as `= default` doesn't make it trivial—it will only be trivial if the class also supports all the other criteria for the corresponding function to be trivial—but explicitly writing the function in user code does *prevent* it from being trivial.

The second difference between classes with compiler-generated functions and user-supplied equivalents is that a class with no user-supplied constructors can be an *aggregate* and thus can be initialized with an aggregate initializer:

```
struct aggregate
{
    aggregate() = default;
    aggregate(aggregate const&) = default;

    int a;
    double b;
};
aggregate x={42,3.141};
```

In this case, `x.a` is initialized to `42` and `x.b` is initialized to `3.141`.

The third difference between a compiler-generated function and a user-supplied equivalent is quite esoteric and applies only to the default constructor and only to the default constructor of classes that meet certain criteria. Consider the following class:

```
struct X
{
    int a;
};
```

If you create an instance of class `X` without an initializer, the contained `int` (`a`) is *default initialized*. If the object has static storage duration, it's initialized to zero; otherwise, it has an indeterminate value that can potentially cause undefined behavior if it's accessed before being assigned a new value:

```
X x1;                                   <-- x1.a has an indeterminate value
```

If, on the other hand, you initialize your instance of `X` by explicitly invoking the default constructor, then `a` is initialized to zero:

```
X x2=X();                               <-- x2.a==0
```

This bizarre property also extends to base classes and members. If your class has a compiler-generated default constructor and any of your data members and base classes also have a compiler-generated default constructor, data members of those bases and members that are built-in types are also either left with an indeterminate value or initialized to zero, depending on whether or not the outer class has its default constructor explicitly invoked.

Although this rule is confusing and potentially error prone, it does have its uses, and if you write the default constructor yourself, you lose this property; either data members like `a` are always initialized (because you specify a value or explicitly default construct) or always uninitialized (because you don't):

```
X::X():a(){}          <-- a==0 always
X::X():a(42){}                          <-- a==42 always
X::X(){}              <-- (1)
```

If you omit the initialization of `a` from the constructor of `X` as in the third example (1), then `a` is left uninitialized for nonstatic instances of `X` and initialized to zero for instances of `X` with static storage duration.

Under normal circumstances, if you write any other constructor manually, the compiler will no longer generate the default constructor for you, so if you want one you have to write it, which means you lose this bizarre initialization property. However, by explicitly declaring the constructor as defaulted, you can force the compiler to generate the default constructor for you, and this property is retained:

```
X::X() = default;                       <-- Default initialization rules for a apply
```

This property is used for the atomic types (see section 5.2), which have their default constructor explicitly defaulted. Their initial value is always undefined unless either (a) they have static storage duration (and thus are statically initialized to zero), or (b) you explicitly invoke the default constructor to request zero initialization, or (c) you explicitly specify a value. Note that in the case of the atomic types, the constructor for initialization with a value is declared `constexpr` (see section A.4) in order to allow static initialization.

A.4 constexpr functions

Integer literals such as `42` are *constant expressions*. So are simple arithmetic expressions such as `23*2-4`. You can even use `const` variables of integral type that are themselves initialized with constant expressions as part of a new constant expression:

```
const int i=23;
const int two_i=i*2;
const int four=4;
const int forty_two=two_i-four;
```

Aside from using constant expressions to create variables that can be used in other constant expressions, there are a few things you can *only* do with constant expressions:

- Specify the bounds of an array:

```
int bounds=99;
int array[bounds];
const int bounds2=99;
int array2[bounds2];
```

Error bounds is not a constant expression

OK, bounds2 is a constant expression

- Specify the value of a nontype template parameter:

```
template<unsigned size>
struct test
{};
test<bounds> ia;
test<bounds2> ia2;
```

Error bounds is not a constant expression

OK, bounds2 is a constant expression

- Provide an initializer for a `static const` class data member of integral type in the class definition:

```
class X
{
    static const int the_answer=forty_two;
};
```

- Provide an initializer for a built-in type or aggregate that can be used for static initialization:

```
struct my_aggregate
{
    int a;
    int b;
};
static my_aggregate ma1={forty_two,123};
int dummy=257;
static my_aggregate ma2={dummy,dummy};
```

Static initialization

Dynamic initialization

- Static initialization like this can be used to avoid order-of-initialization problems and race conditions.

None of this is new—you could do all that with the 1998 edition of the C++ Standard. However, with the new Standard what constitutes a *constant expression* has been extended with the introduction of the `constexpr` keyword.

The `constexpr` keyword is primarily a function modifier. If the parameter and return type of a function meet certain requirements and the body is sufficiently simple,

a function can be declared `constexpr`, in which case it can be used in constant expressions, for example:

```
constexpr int square(int x)
{
    return x*x;
}
int array[square(5)];
```

In this case, `array` will have 25 entries, because `square` is declared `constexpr`. Of course, just because the function *can* be used in a constant expression doesn't mean that all uses are automatically constant expressions:

```
int dummy=4;
int array[square(dummy)];
```

❶ **Error, dummy is not a constant expression**

In this example, `dummy` is not a constant expression ❶, so `square(dummy)` isn't either—it's just a normal function call—and thus can't be used to specify the bounds of `array`.

A.4.1 constexpr and user-defined types

Up to now, all the examples have been with built-in types such as `int`. However, the new C++ Standard allows constant expressions to be of any type that satisfies the requirements for a *literal type*. For a class type to be classified as a literal type, the following must all be true:

- It must have a trivial copy constructor.
- It must have a trivial destructor.
- All non-`static` data members and base classes must be trivial types.
- It must have either a trivial default constructor or a `constexpr` constructor other than the copy constructor.

We'll look at `constexpr` constructors shortly. For now we'll focus on classes with a trivial default constructor, such as class `CX` in the following listing.

Listing A.3 A class with a trivial default constructor

```
class CX
{
private:
    int a;
    int b;
public:
    CX() = default;                    <-- 1
    CX(int a_, int b_):                <-- 2
        a(a_),b(b_)
    {}
    int get_a() const
    {
        return a;
    }
    int get_b() const
```

```
    {
        return b;
    }
    int foo() const
    {
        return a+b;
    }
};
```

Note that we've explicitly declared the default constructor ❶ as *defaulted* (see section A.3) in order to preserve it as trivial in the face of the user-defined constructor ❷. This type therefore fits all the qualifications for being a literal type, and you can use it in constant expressions. You can, for example, provide a `constexpr` function that creates new instances:

```
constexpr CX create_cx()
{
    return CX();
}
```

You can also create a simple `constexpr` function that copies its parameter:

```
constexpr CX clone(CX val)
{
    return val;
}
```

But that's about all you can do—a `constexpr` function can only call other `constexpr` functions. What you *can* do, though, is apply `constexpr` to the member functions and constructor of `CX`:

```
class CX
{
private:
    int a;
    int b;
public:
    CX() = default;
    constexpr CX(int a_, int b_):
        a(a_),b(b_)
    {}
    constexpr int get_a() const      <-- 1
    {
        return a;
    }
    constexpr int get_b()            <-- 2
    {
        return b;
    }
    constexpr int foo()
    {
        return a+b;
    }
};
```

Note that the `const` qualification on `get_a()` ❶ is now superfluous, because it's implied by the use of `constexpr`. `get_b()` is thus `const` even though the `const` qualification is omitted ❷. This now allows more complex `constexpr` functions such as the following:

```
constexpr CX make_cx(int a)
{
    return CX(a,1);
}
constexpr CX half_double(CX old)
{
    return CX(old.get_a()/2,old.get_b()*2);
}
constexpr int foo_squared(CX val)
{
    return square(val.foo());
}
int array[foo_squared(half_double(make_cx(10)))];    <-- 49 elements
```

Interesting though this is, it's a lot of effort to go to if all you get is a fancy way of computing some array bounds or an integral constant. The key benefit of constant expressions and `constexpr` functions involving user-defined types is that objects of a literal type initialized with a constant expression are statically initialized, and so their initialization is free from race conditions and initialization order issues:

```
CX si=half_double(CX(42,19));          <-- Statically initialized
```

This covers constructors too. If the constructor is declared `constexpr` and the constructor parameters are constant expressions, the initialization is *constant initialization* and happens as part of the static initialization phase. *This is one of the most important changes in C++11 as far as concurrency goes:* by allowing user-defined constructors that can still undergo static initialization, you can avoid any race conditions over their initialization, because they're guaranteed to be initialized before any code is run.

This is particularly relevant for things like `std::mutex` (see section 3.2.1) or `std::atomic<>` (see section 5.2.6) where you might want to use a global instance to synchronize access to other variables and avoid race conditions in *that* access. This wouldn't be possible if the constructor of the mutex was subject to race conditions, so the default constructor of `std::mutex` is declared `constexpr` to ensure that mutex initialization is always done as part of the static initialization phase.

A.4.2 constexpr objects

So far we've looked at `constexpr` as applied to functions. `constexpr` can also be applied to objects. This is primarily for diagnostic purposes; it verifies that the object is initialized with a constant expression, `constexpr` constructor, or aggregate initializer made of constant expressions. It also declares the object as `const`:

```
constexpr int i=45;                    <-- OK
constexpr std::string s("hello");      <-- Error, std::string isn't a literal type
```

```
int foo();
constexpr int j=foo();                    ⏎ Error, foo() isn't declared constexpr
```

A.4.3 constexpr function requirements

In order to declare a function as `constexpr` it must meet a few requirements; if it doesn't meet these requirements, declaring it `constexpr` is a compilation error. The requirements for a `constexpr` function are as follows:

- All parameters must be of a literal type.
- The return type must be a literal type.
- The function body must consist of a single `return` statement.
- The expression in the `return` statement must qualify as a constant expression.
- Any constructor or conversion operator used to construct the return value from the expression must be `constexpr`.

This is straightforward; you must be able to inline the function into a constant expression and it will still be a constant expression, and you must not modify anything. `constexpr` functions are *pure functions* with no side effects.

For `constexpr` class member functions there are additional requirements:

- `constexpr` member functions can't be virtual.
- The class for which the function is a member must be a literal type.

The rules are different for `constexpr` constructors:

- The constructor body must be empty.
- Every base class must be initialized.
- Every non-`static` data member must be initialized.
- Any expressions used in the member initialization list must qualify as constant expressions.
- The constructors chosen for the initialization of the data members and base classes must be `constexpr` constructors.
- Any constructor or conversion operator used to construct the data members and base classes from their corresponding initialization expression must be `constexpr`.

This is the same set of rules as for functions, except that there's no return value, so no `return` statement. Instead, the constructor initializes all the bases and data members in the member initialization list. Trivial copy constructors are implicitly `constexpr`.

A.4.4 constexpr and templates

When `constexpr` is applied to a function template, or to a member function of a class template, it's ignored if the parameters and return types of a particular instantiation of the template aren't literal types. This allows you to write function templates that are `constexpr` if the type of the template parameters is appropriate and just plain `inline` functions otherwise, for example:

```
template<typename T>
constexpr T sum(T a,T b)
```

```
{
    return a+b;
}
constexpr int i=sum(3,42);
std::string s=
    sum(std::string("hello"),
        std::string(" world"));
```

OK, sum<int> is constexpr

OK, but sum<std::string> isn't constexpr

The function must satisfy all the other requirements for a `constexpr` function. You can't declare a function with multiple statements `constexpr` just because it's a function template; that's still a compilation error.

A.5 Lambda functions

Lambda functions are one of the most exciting features of the C++11 Standard, because they have the potential to greatly simplify code and eliminate much of the boilerplate associated with writing callable objects. The C++11 lambda function syntax allows a function to be defined at the point where it's needed in another expression. This works well for things like predicates provided to the wait functions of `std::condition_variable` (as in the example in section 4.1.1), because it allows the semantics to be quickly expressed in terms of the accessible variables rather than capturing the necessary state in the member variables of a class with a function call operator.

At its simplest, a *lambda expression* defines a self-contained function that takes no parameters and relies only on global variables and functions. It doesn't even have to return a value. Such a lambda expression is a series of statements enclosed in braces, prefixed with square brackets (the *lambda introducer*):

```
[]{                          <-- Start the lambda expression with []
    do_stuff();
    do_more_stuff();
}();                         <-- Finish the lambda, and call it
```

In this example, the lambda expression is called by following it with parentheses, but this is unusual. For one thing, if you're going to call it directly, you could usually do away with the lambda and write the statements directly in the source. It's more common to pass it as a parameter to a function template that takes a callable object as one of its parameters, in which case it likely needs to take parameters or return a value or both. If you need to take parameters, you can do this by following the lambda introducer with a parameter list just like for a normal function. For example, the following code writes all the elements of the vector to `std::cout` separated by newlines:

```
std::vector<int> data=make_data();
std::for_each(data.begin(),data.end(),[](int i){std::cout<<i<<"\n";});
```

Return values are almost as easy. If your lambda function body consists of a single `return` statement, the return type of the lambda is the type of the expression being returned. For example, you might use a simple lambda like this to wait for a flag to be set with a `std::condition_variable` (see section 4.1.1) as in the following listing.

Listing A.4 A simple lambda with a deduced return type

```
std::condition_variable cond;
bool data_ready;
std::mutex m;

void wait_for_data()
{
    std::unique_lock<std::mutex> lk(m);
    cond.wait(lk,[]{return data_ready;});      <-- 1
}
```

The return type of the lambda passed to `cond.wait()` ❶ is deduced from the type of `data_ready` and is thus `bool`. Whenever the condition variable wakes from waiting, it then calls the lambda with the mutex locked and only returns from the call to `wait()` once `data_ready` is `true`.

What if you can't write your lambda body as a single `return` statement? In that case you have to specify the return type explicitly. You can do this even if your body is a single `return` statement, but you *have* to do it if your lambda body is more complex. The return type is specified by following the lambda parameter list with an arrow (`->`) and the return type. If your lambda doesn't take any parameters, you must still include the (empty) parameter list in order to specify the return value explicitly. Your condition variable predicate can thus be written

```
cond.wait(lk,[]()->bool{return data_ready;});
```

By specifying the return type, you can expand the lambda to log messages or do some more complex processing:

```
cond.wait(lk,[]()->bool{
    if(data_ready)
    {
        std::cout<<"Data ready"<<std::endl;
        return true;
    }
    else
    {
        std::cout<<"Data not ready, resuming wait"<<std::endl;
        return false;
    }
});
```

Although simple lambdas like this are powerful and can simplify code quite a lot, the real power of lambdas comes when they capture local variables.

A.5.1 Lambda functions that reference local variables

Lambda functions with a *lambda introducer* of `[]` can't reference any local variables from the containing scope; they can only use global variables and anything passed in as a parameter. If you wish to access a local variable, you need to *capture* it. The simplest way to do this is to capture the entire set of variables within the local scope by

using a lambda introducer of `[=]`. That's all there is to it—your lambda can now access *copies* of the local variables at the time the lambda was created.

To see this in action, consider the following simple function:

```
std::function<int(int)> make_offseter(int offset)
{
   return [=](int j){return offset+j;};
}
```

Every call to `make_offseter` returns a new lambda function object through the `std::function<>` function wrapper. This returned function adds the supplied offset to any parameter supplied. For example,

```
int main()
{
    std::function<int(int)> offset_42=make_offseter(42);
    std::function<int(int)> offset_123=make_offseter(123);
    std::cout<<offset_42(12)<<",“<<offset_123(12)<<std::endl;
    std::cout<<offset_42(12)<<",“<<offset_123(12)<<std::endl;
}
```

will write out `54,135` twice because the function returned from the first call to `make_offseter` always adds 42 to the supplied argument, whereas the function returned from the second call to `make_offseter` always adds 123 to the supplied argument.

This is the safest form of local variable capture; everything is copied, so you can return the lambda and call it outside the scope of the original function. It's not the only choice though; you can choose to capture everything by reference instead. In this case it's undefined behavior to call the lambda once the variables it references have been destroyed by exiting the function or block scope to which they belong, just as it's undefined behavior to reference a variable that has already been destroyed in any other circumstance.

A lambda function that captures all the local variables by reference is introduced using `[&]`, as in the following example:

```
int main()
{
    int offset=42;                                                    ❶
    std::function<int(int)> offset_a=[&](int j){return offset+j;};    ❷
    offset=123;                                                       ❸
    std::function<int(int)> offset_b=[&](int j){return offset+j;};    ❹
    std::cout<<offset_a(12)<<",”<<offset_b(12)<<std::endl;            ❺
    offset=99;                                                        ❻
    std::cout<<offset_a(12)<<",”<<offset_b(12)<<std::endl;            ❼
}
```

Whereas in the `make_offseter` function from the previous example we used the `[=]` lambda introducer to capture a copy of the offset, the `offset_a` function in this example uses the `[&]` lambda introducer to capture `offset` by reference ❷. It doesn't matter that the initial value of `offset` is 42 ❶; the result of calling `offset_a(12)` will always depend on the current value of `offset`. Even though the value of `offset` is

then changed to 123 ❸ before we produce the second (identical) lambda function `offset_b` ❹, this second lambda again captures by reference, so the result depends on the current value of `offset`.

Now, when we print the first line of output ❺, `offset` is still 123, so the output is `135,135`. However, at the second line of output ❼, `offset` has been changed to 99 ❻, so this time the output is `111,111`. Both `offset_a` and `offset_b` add the current value of `offset` (99) to the supplied argument (12).

Now, C++ being C++, you're not stuck with these all-or-nothing options; you can choose to capture some variables by copy and some by reference, and you can choose to capture only those variables you have explicitly chosen just by tweaking the lambda introducer. If you wish to *copy* all the used variables except for one or two, you can use the `[=]` form of the lambda introducer but follow the equals sign with a list of variables to capture by reference preceded with ampersands. The following example will thus print `1239`, because `i` is copied into the lambda, but `j` and `k` are captured by reference:

```
int main()
{
    int i=1234,j=5678,k=9;
    std::function<int()> f=[=,&j,&k]{return i+j+k;};
    i=1;
    j=2;
    k=3;
    std::cout<<f()<<std::endl;
}
```

Alternatively, you can capture by reference by default but capture a specific subset of variables by copying. In this case you use the `[&]` form of the lambda introducer but follow the ampersand with a list of variables to capture by copy. The following example thus prints `5688` because `i` is captured by reference, but `j` and `k` are copied:

```
int main()
{
    int i=1234,j=5678,k=9;
    std::function<int()> f=[&,j,k]{return i+j+k;};
    i=1;
    j=2;
    k=3;
    std::cout<<f()<<std::endl;
}
```

If you only want to capture the named variables, then you can omit the leading `=` or `&` and just list the variables to be captured, prefixing them with an ampersand to capture by reference rather than copy. The following code will thus print `5682` because `i` and `k` are captured by reference, but `j` is copied:

```
int main()
{
    int i=1234,j=5678,k=9;
    std::function<int()> f=[&i,j,&k]{return i+j+k;};
```

```
    i=1;
    j=2;
    k=3;
    std::cout<<f()<<std::endl;
}
```

This final variant allows you to ensure that only the intended variables are being captured, because any reference to a local variable not in the capture list will cause a compilation error. If you choose this option, you have to be careful when accessing class members if the function containing the lambda is a member function. Class members can't be captured directly; if you wish to access class members from your lambda, you have to capture the `this` pointer by adding it to the capture list. In the following example, the lambda captures `this` to allow access to the `some_data` class member:

```
struct X
{
    int some_data;
    void foo(std::vector<int>& vec)
    {
        std::for_each(vec.begin(),vec.end(),
            [this](int& i){i+=some_data;});
    }
};
```

In the context of concurrency, lambdas are most useful as predicates for `std::condition_variable::wait()` (section 4.1.1) and with `std::packaged_task<>` (section 4.2.1) or thread pools for packaging small tasks. They can also be passed to the `std::thread` constructor as a thread function (section 2.1.1) and as the function when using parallel algorithms such as `parallel_for_each()` (from section 8.5.1).

A.6 Variadic templates

Variadic templates are templates with a variable number of parameters. Just as you've always been able to have variadic functions such as `printf` that take a variable number of parameters, you can now have variadic templates that have a variable number of *template* parameters. Variadic templates are used throughout the C++ Thread Library. For example, the `std::thread` constructor for starting a thread (section 2.1.1) is a variadic function template, and `std::packaged_task<>` (section 4.2.2) is a variadic class template. From a user's point of view, it's enough to know that the template takes an unbounded number of parameters, but if you want to write such a template, or if you're just interested in how it all works, you need to know the details.

Just as variadic functions are declared with an ellipsis (`...`) in the function parameter list, variadic templates are declared with an ellipsis in the template parameter list:

```
template<typename ... ParameterPack>
class my_template
{};
```

You can use variadic templates for a partial specialization of a template too, even if the primary template isn't variadic. For example, the primary template for `std::packaged_task<>` (section 4.2.1) is just a simple template with a single template parameter:

```
template<typename FunctionType>
class packaged_task;
```

However, this primary template is never defined anywhere; it's just a placeholder for the partial specialization:

```
template<typename ReturnType,typename ... Args>
class packaged_task<ReturnType(Args...)>;
```

It's this partial specialization that contains the real definition of the class; you saw in chapter 4 that you can write `std::packaged_task<int(std::string,double)>` to declare a task that takes a `std::string` and a `double` as parameters when you call it and that provides the result through a `std::future<int>`.

This declaration shows two additional features of variadic templates. The first feature is relatively simple: you can have normal template parameters (such as `ReturnType`) as well as variadic ones (`Args`) in the same declaration. The second feature demonstrated is the use of `Args...` in the template argument list of the specialization to show that the types that make up `Args` when the template is instantiated are to be listed here. Actually, because this is a partial specialization, it works as a pattern match; the types that occur in this context in the actual instantiation are captured as `Args`. The variadic parameter `Args` is called a *parameter pack*, and the use of `Args...` is called a *pack expansion*.

Just like with variadic functions, the variadic part may be an empty list or may have many entries. For example, with `std::packaged_task<my_class()>` the `ReturnType` parameter is `my_class`, and the `Args` parameter pack is empty, whereas with `std::packaged_task<void(int,double,my_class&,std::string*)>` the `ReturnType` is `void`, and `Args` is the list `int`, `double`, `my_class&`, `std::string*`.

A.6.1 Expanding the parameter pack

The power of variadic templates comes from what you can do with that pack expansion: you aren't limited to just expanding the list of types as is. First off, you can use a pack expansion directly anywhere a list of types is required, such as in the argument list for another template:

```
template<typename ... Params>
struct dummy
{
    std::tuple<Params...> data;
};
```

In this case the single member variable `data` is an instantiation of `std::tuple<>` containing all the types specified, so `dummy<int,double,char>` has a member of type `std::tuple<int,double,char>`. You can combine pack expansions with normal types:

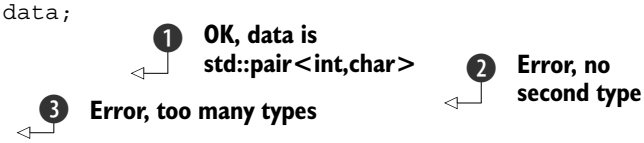
```
template<typename ... Params>
struct dummy2
{
    std::tuple<std::string,Params...> data;
};
```

This time, the tuple has an additional (first) member of type `std::string`. The nifty part is that you can create a pattern with the pack expansion, which is then copied for each element in the expansion. You do this by putting the ... that marks the pack expansion at the end of the pattern. For example, rather than just creating a tuple of the elements supplied in your parameter pack, you can create a tuple of pointers to the elements or even a tuple of `std::unique_ptr<>`s to your elements:

```
template<typename ... Params>
struct dummy3
{
    std::tuple<Params* ...> pointers;
    std::tuple<std::unique_ptr<Params> ...> unique_pointers;
};
```

The type expression can be as complex as you like, provided the parameter pack occurs in the type expression, and provided the expression is followed by the ... that marks the expansion. When the parameter pack is expanded, for each entry in the pack that type is substituted into the type expression to generate the corresponding entry in the resulting list. Thus, if your parameter pack `Params` contains the types `int,int,char`, then the expansion of `std::tuple<std::pair<std::unique_ptr<Params>,double> ... >` is `std::tuple<std::pair<std::unique_ptr<int>,double>, std::pair<std::unique_ptr<int>,double>, std::pair<std::unique_ptr<char>, double> >`. If the pack expansion is used as a template argument list, that template doesn't have to have variadic parameters, but if it doesn't, the size of the pack must exactly match the number of template parameters required:

```
template<typename ... Types>
struct dummy4
{
    std::pair<Types...> data;
};
dummy4<int,char> a;
dummy4<int> b;
dummy4<int,int,int> c;
```



The second thing you can do with a pack expansion is use it to declare a list of function parameters:

```
template<typename ... Args>
void foo(Args ... args);
```

This creates a new parameter pack `args`, which is a list of the function parameters rather than a list of types, which you can expand with ... as before. Now, you can use a pattern with the pack expansion for declaring the function parameters, just as you can use a pattern when you expand the pack elsewhere. For example, this is used by

the `std::thread` constructor to take all the function arguments by rvalue reference (see section A.1):

```
template<typename CallableType,typename ... Args>
thread::thread(CallableType&& func,Args&& ... args);
```

The function parameter pack can then be used to call another function, by specifying the pack expansion in the argument list of the called function. Just as with the type expansions, you can use a pattern for each expression in the resulting argument list. For example, one common idiom with rvalue references is to use `std::forward<>` to preserve the rvalue-ness of the supplied function arguments:

```
template<typename ... ArgTypes>
void bar(ArgTypes&& ... args)
{
    foo(std::forward<ArgTypes>(args)...);
}
```

Note that in this case, the pack expansion contains both the type pack `ArgTypes` and the function parameter pack `args`, and the ellipsis follows the whole expression. If you call `bar` like this,

```
int i;
bar(i,3.141,std::string("hello "));
```

then the expansion becomes

```
template<>
void bar<int&,double,std::string>(
    int& args_1,
    double&& args_2,
    std::string&& args_3)
{
    foo(std::forward<int&>(args_1),
        std::forward<double>(args_2),
        std::forward<std::string>(args_3));
}
```

which correctly passes the first argument on to `foo` as an lvalue reference, while passing the others as rvalue references.

The final thing you can do with a parameter pack is find its size with the `sizeof...` operator. This is quite simple: `sizeof...(p)` is the number of elements in the parameter pack `p`. It doesn't matter whether this is a type parameter pack or a function argument parameter pack; the result is the same. This is probably the only case where you can use a parameter pack and not follow it with an ellipsis; the ellipsis is already part of the `sizeof...` operator. The following function returns the number of arguments supplied to it:

```
template<typename ... Args>
unsigned count_args(Args ... args)
{
    return sizeof... (Args);
}
```

Just as with the normal `sizeof` operator, the result of `sizeof...` is a constant expression, so it can be used for specifying array bounds and so forth.

A.7 Automatically deducing the type of a variable

C++ is a statically typed language: the type of every variable is known at compile time. Not only that, but as a programmer you have to actually specify the type of each variable. In some cases this can lead to quite unwieldy names, for example:

```
std::map<std::string,std::unique_ptr<some_data>> m;
std::map<std::string,std::unique_ptr<some_data>>::iterator
    iter=m.find("my key");
```

Traditionally, the solution has been to use `typedefs` to reduce the length of a type identifier and potentially eliminate problems due to inconsistent types. This still works in C++11, but there's now a new way: if a variable is initialized in its declaration from a value of the same type, then you can specify the type as `auto`. In this case, the compiler will automatically deduce the type of the variable to be the same as the initializer. Thus, the iterator example can be written as

```
auto iter=m.find("my key");
```

Now, you're not restricted to just plain `auto`; you can embellish it to declare `const` variables or pointer or reference variables too. Here are a few variable declarations using `auto` and the corresponding type of the variable:

```
auto i=42;        // int
auto& j=i;        // int&
auto const k=i;   // int const
auto* const p=&i; // int * const
```

The rules for deducing the type of the variable are based on the rules for the only other place in the language where types are deduced: parameters of function templates. In a declaration of the form

```
some-type-expression-involving-auto var=some-expression;
```

the type of `var` is the same as the type deduced for the parameter of a function template declared with the same type expression, except replacing `auto` with the name of a template type parameter:

```
template<typename T>
void f(type-expression var);
f(some-expression);
```

This means that array types decay to pointers, and references are dropped unless the type expression explicitly declares the variable as a reference, for example:

```
int some_array[45];
auto p=some_array;   // int*
int& r=*p;
auto x=r;            // int
auto& y=r;           // int&
```

This can greatly simplify the declaration of variables, particularly where the full type identifier is long or possibly not even known (for example, the type of the result of a function call in a template).

A.8 Thread-local variables

Thread-local variables allow you to have a separate instance of a variable for each thread in your program. You mark a variable as being thread-local by declaring it with the `thread_local` keyword. Variables at namespace scope, static data members of classes, and local variables can be declared thread-local, and are said to have *thread storage duration*:

```
thread_local int x;
```

A thread-local variable at namespace scope

```
class X
{
    static thread_local std::string s;
};
static thread_local std::string X::s;

void foo()
{
    thread_local std::vector<int> v;
}
```

A thread-local static class data member

The definition of X::s is required

A thread-local local variable

Thread-local variables at namespace scope and thread-local static class data members are constructed before the first use of a thread-local variable from the same translation unit, but it isn't specified *how much* before. Some implementations may construct thread-local variables when the thread is started; others may construct them immediately before their first use on each thread, and others may construct them at other times, or in some combination depending on their usage context. Indeed, if none of the thread-local variables from a given translation unit is used, there's no guarantee that they will be constructed at all. This allows for the dynamic loading of modules containing thread-local variables—these variables can be constructed on a given thread the first time that thread references a thread-local variable from the dynamically-loaded module.

Thread-local variables declared inside a function are initialized the first time the flow of control passes through their declaration on a given thread. If the function is not called by a given thread, any thread-local variables declared in that function are not constructed. This is just the same as the behaviour for local static variables, except it applies separately to each thread.

Thread-local variables share other properties with static variables—they're zero-initialized prior to any further initialization (such as dynamic initialization), and if the construction of a thread-local variable throws an exception, `std::terminate()` is called to abort the application.

The destructors for all thread-local variables that have been constructed on a given thread are run when the thread function returns, in the reverse order of construction. Since the order of initialization is unspecified, it's important to ensure that there are

no interdependencies between the destructors of such variables. If the destructor of a thread-local variable exits with an exception, `std::terminate()` is called, just as for construction.

Thread-local variables are also destroyed for a thread if *that thread* calls `std::exit()` or returns from main() (which is equivalent to calling `std::exit()` with the return value of `main()`). If any other threads are still running when the application exits, the destructors of thread-local variables on those threads are *not* called.

Though thread-local variables have a different address on each thread, you can still obtain a normal pointer to such a variable. The pointer then references the object in the thread that took the address, and can be used to allow other threads to access that object. It's undefined behaviour to access an object after it's been destroyed (as always), so if you pass a pointer to a thread-local variable to another thread, you need to ensure it's not dereferenced once the owning thread has finished.

A.9 Summary

This appendix has only scratched the surface of the new language features introduced with the C++11 Standard, because we've only looked at those features that actively affect the usage of the Thread Library. Other new language features include static assertions, strongly typed enumerations, delegating constructors, Unicode support, template aliases, and a new uniform initialization sequence, along with a host of smaller changes. Describing all the new features in detail is outside the scope of this book; it would probably require a book in itself. The best overview of the entire set of changes to the standard at the time of writing is probably Bjarne Stroustrup's C++11 FAQ,[1] though popular C++ reference books will be revised to cover it in due course.

Hopefully the brief introduction to the new features covered in this appendix has provided enough depth to show how they relate to the Thread Library and to enable you to write and understand multithreaded code that uses these new features. Although this appendix should provide enough depth for simple uses of the features covered, this is still only a brief introduction and not a complete reference or tutorial for the use of these features. If you intend to make extensive use of them, I recommend acquiring such a reference or tutorial in order to gain the most benefit from them.

[1] http://www.research.att.com/~bs/C++0xFAQ.html