APPENDIX

C++11

D

This appendix explains new features in the C++ 2011 standard, informally called C++11, that are used in this book. These features are available in several widely used C++ compilers. This is not intended to be a full tutorial on these features, but should provide enough information to enable understanding of the examples. It also explains suitable substitutes for use with C++ 1998 compilers.

D.1 DECLARING WITH `auto`

C++11 permits the `auto` keyword to be used in place of a type in some contexts where the type can be deduced by the compiler. Here is an example:

```
std::vector<double> v;
...
for (auto i = v.begin(); i != v.end(); ++i) {
   auto& x = *i;
   ...
}
```

The C++98 equivalent would be:

```
std::vector<double> v;
...
for (std::vector<double>::iterator i = v.begin(); i != v.end(); ++i) {
   double& x = *i;
   ...
}
```

C++11 introduces a range-based `for` statement that can make this example even shorter, but it was not widely available at the time of this writing.

D.2 LAMBDA EXPRESSIONS

Lambda expressions are a C++11 feature already supported by several compilers, such as Intel C++ 12.0, GCC 4.5, and Microsoft Visual Studio 2010. Lambda expressions are important because they greatly simplify using templates to implement control structures, including parallel ones. A key point

```
#include <algorithm>

class comparator {
   const float key;
public:
   comparator( float key_ ) : key(key_) { }
   bool operator()( const float& x ) const {
      return (x < key);
   }
};

// Return number of keys in [ first , last ) that are less than given key
size_t count_less_than_key(
   float* first,
   float* last,
   float key
) {
   return std::count_if(first, last, comparator(key));
}
```

LISTING D.1

Using a manually written functor `comparator`.

to understand is there is nothing magic about lambda expressions. A lambda expression is simply a concise way to create a function object that could otherwise be written manually.

For example, consider the standard library template `std::count`(*first*,*last*,*pred*). It counts the number of elements in the half-open interval [*first*,*last*) for which predicate *pred* holds. Listing D.1 shows an example that counts how many elements are less than a `key` of type `float`.

Class `comparator` is called a *function object*, or **functor**. It has three parts:

- A field for storing the key
- A constructor that captures the key
- A definition of what to do when the class is applied like a function

For example, the following fragment constructs a `comparator` that compares against `0` and applies it as a function to `−1`.

```
comparator c(0);            // Calls constructor for comparator
bool is_negative = c(-1); // Calls comparator::operator()
```

As the counting example shows, writing functor objects can be a lot of work. Lambda expressions make the compiler do this work for you. Listing D.2 shows the counting example rewritten with a lambda expression.

The explicit class definition has completely disappeared. The lambda expression builds the equivalent for you.

```
1  // Return number of keys in [ first , last ) that are less than given key
2  size_t count_less_than_key(
3     float* first,
4     float* last,
5     float key
6  ) {
7     return std::count_if(first, last,
8        [=]( const float& x ) {
9           return x < key;
10       }
11    );
12 }
```

LISTING D.2

Using a lambda expression lets Listing D.1 be rewritten more concisely.

The lambda expression here has three parts:

- A part `[=]` that describes how to capture local variables outside the lambda
- A parameter list `(const float& x)`, which can be omitted if there are no parameters
- A definition {`return key;`}

For the sake of exposition, these parts will be discussed in reverse order, from simplest to most subtle.

The definition is a compound statement that becomes the body of the `operator()` that the compiler will generate. The argument list for a lambda expression becomes the argument list of the `operator()`. A lambda expression can optionally specify a return type after the argument list. The notation is `->`*`return-type`*. Here is the lambda expression from the example, rewritten with an explicit return type:

```
[=]( const float& x )->bool {return x < key;}
```

Often the return type does not need to be specified, because if omitted it is inferred by the following rules:

- If the definition is a single statement `return` *expr*, the return type is the type of *expr*.
- Otherwise, the return type is `void`.

The capture part describes how to capture local variables. Global variables are never captured since they can be globally referenced. A local variable can be captured **by value** or **by reference**. Capture by value copies the variable into the function object. Capture by reference creates a reference in the function object back to the local variable; the corresponding field in the function object becomes a reference. A lambda expression can specify the kind of capture or none at all:

`[=]` capture by value

`[&]` capture by reference

`[]` capture nothing

```
class assign {
   float &x, &y;
   float a, b;
public:
   assign( float& x_, float& y_, float a_, float b_ ) :
      x(x_), y(y_), a(a_), b(b_) {}
   void operator()() {x=a; y=b;}
};

float foo( float& x, float b ) {
   float a=2;
   float y;
   auto op = assign(x,y,a,b);
   op();
   return y;
}
```

LISTING D.3

Mixed capture with handwritten functor. The code is equivalent to Listing D.4.

```
float foo( float& x, float b ) {
   float a=2;
   float y;
   auto op = [=,&x,&y] {x = a; y = b;};
   op();
   return y;
}
```

LISTING D.4

Mixed capture modes. The lambda expression captures `x` and `y` by reference and `a` and `b` by value.

The same lambda can capture different variables differently. The initial = or & specifies a default, which can be overridden for specific variables. Here are some examples:

`[=,&x,&y]`	capture by value, except that `x` and `y` are captured by reference
`[&,x,y]`	capture by reference, except that `x` and `y` are captured by value

Listing D.4 shows mixed capture modes. The `auto` in the listing is more than a convenience—lambda expressions have anonymous type and hence it is impossible to name that type in the declaration. Listing D.4 shows that an equivalent functor can be written by hand.

For parallel programming, choosing the proper capture mode can be critical. Capture by value incurs the cost of making a private copy of the captured object. Capture by reference incurs the cost of an extra level of indirection when the object is accessed. In general, use capture by value if the object is small and quick to copy and does not need to be modified. Doing so avoids the extra level of indirection incurred for capture by reference and avoids potential for races. In particular, pointers are small objects. They should be captured by value if they are not going to be modified, even if they point to objects that

are going to be modified. Otherwise, use capture by reference, and think carefully about whether there could be races on the local variable being referenced. And, remember, a lambda expression is just a convenience. The object it generates can be reasoned about like a handcrafted function object.

D.3 std::move

The C++11 notation $y =$ `std::move(x)` is similar to $y = x$, except that it gives license to change the value of x. For example, if x is an instance of `std::vector`, the move may set x to empty. The advantage is that the license permits significant optimizations in some cases. For example, suppose y is an empty vector and x is a vector with N elements. Assignment takes $\Theta(N)$ time, because each element must be copied. Moving can be done in $\Theta(1)$ time, because it only needs to move internal pointers and size information from x to y, and not copy the vector's contents. In other words, moving is allowed to transfer resources from x to y.

C++11 also adds a "move" counterpart of `std::copy`. Given iterators `first`, `last`, and `result`, a call

```
std::move(first,last,result)
```

moves items in the range `[first,last)` to `[result,result+(last-first))`. Afterward, items in the range `[first,last)` have unspecified values.

For old compilers that do not support `std::move`, you can use one of two replacements for `y = std::move(x)`:

- **assignment**: The assignment $y = x$ is a valid replacement and, for C-like types, has performance similar to that of `y=std::move(x)`. However, for more complicated types, assignment may introduce additional copying overhead that moving can avoid. For example, if y and x are instances of `std::vector`, assignment copies the vector elements, whereas moving does not.
- **swap**: The expression `swap(y,x)` is a valid replacement and, for some object types such as STL containers, may be faster than `y=x`. The reason is that STL containers often implement `swap` in a way that swaps a few internal pointers instead of copying full state. However, `swap` introduces an unnecessary update of x and thus may be slower for C-like types.

You can replace `std::move(first,last,result)` with `std::copy(first,last,result)`, possibly adding additional copying overhead that moving avoids.

The `std::move` feature is based on a language feature called *rvalue references* [HSK08].