5

A Tour of C++: Concurrency and Utilities

Programming is like sex:
It may give some concrete results,
but that is not why we do it.
– apologies to Richard Feynman

- Introduction
- Resource Management
  unique_ptr and shared_ptr
- Concurrency
  Tasks and threads; Passing Arguments; Returning Results; Sharing Data; Communicating Tasks
- Small Utility Components
  Time; Type Functions; pair and tuple
- Regular Expressions
- Math
  Mathematical Functions and Algorithms; Complex Numbers; Random Numbers; Vector Arithmetic; Numeric Limits
- Advice

5.1 Introduction [tour4.intro]

From an end-user's perspective, the ideal standard library would provide components directly supporting essentially every need. For a given application domain, a huge commercial library can come close to that ideal. However, that is not what the C++ standard library is trying to do. A manageable, universally available, library cannot be everything to everybody. Instead, the C++ standard library aims to provide components that are

useful to most people in most application areas. That is, it aims to serve the intersection of all needs rather than their union. In addition, support for a few widely important application areas, such as mathematical computation and text manipulation, have crept in.

5.2 Resource Management [tour4.resources]

One of the key tasks of any nontrivial program is to manage resources. A resource is something that must be acquired and later (explicitly or implicitly) released. Examples are memory, locks, sockets, thread handles, and file handles. For a long-running program, failing to release a resource in a timely manner (''a leak'') can cause serious performance degradation and possibly even a miserable crash. Even for short programs, a leak can become an embarrassment, say by a resource shortage increasing the run time by orders of magnitude.

The standard library components are designed not to leak resources. To do this, they rely on the basic language support for resource management using constructor/destructor pairs to ensure that a resource doesn't outlive an object responsible for it. The use of a constructor/destructor pair in Vector to manage the lifetime of its elements is an example (§3.2.1.2) and all standard-library containers are implemented in similar ways. Importantly, this approach interacts correctly with error handling using exceptions. For example, the technique is used for the standard-library lock classes:

```
mutex m;  // used to protect access to shared data
// ...
void f()
{
        lock_guard<mutex> lck {m};      // acquire the mutex m
        // ... manipulate shared data ...
}
```

A thread will not proceed until lck's constructor has acquired its mutex, m (§5.3.4). The corresponding destructor releases the resource. So, in this example, lock_guard's destructor releases the mutex when the thread of control leaves f() (through a return, by ''falling off the end of the function,'' or through an exception throw).

This is an application of the ''Resource Acquisition Is Initialization'' technique (RAII; §3.2.1.2, §13.3). This technique is fundamental to the idiomatic handling of resources in C++. Containers (such as vector and map), string, and iostream manage their resources (such as file handles and buffers) similarly.

5.2.1 unique_ptr and shared_ptr [tour4.smart]

The examples so far take care of objects defined in a scope, releasing the resources they acquire at the exit from the scope, but what about objects allocated on the free store? In <memory>, the standard library provides two ''smart pointers'' to help manage objects on the free store:

[1] unique_ptr to represent unique ownership (§34.3.1)

[2] shared_ptr to represent shared ownership (§34.3.2)

The most basic use of one of these ''smart pointers'' is to prevent memory leaks caused by careless programming:

```
void f(int i, int j)  // X* vs. unique_ptr<X>
{
        X* p = new X;                 // allocate a new X
        unique_ptr<X> sp {new X};// allocate a new X and give its pointer to unique_ptr
        // ...
        if (i<99) throw Z{};          // may throw an exception
        if (j<77) return;             // may return "early"
        p->do_something();            // may throw an exception
        sp->do_something();           // may throw an exception
        // ...
        delete p;                     // destroy *p
}
```

Here, we ''forgot'' to delete p if i<99 or if j<77. On the other hand, unique_ptr ensures that its object is properly destroyed whichever way we exit f() (by throwing an exception, by executing return, or by ''falling off the end'').

In this simple case, we could have solved the problem simply by *not* using a pointer and *not* using new:

```
void f(int i, int j)  // use a local variable
{
        X x;
        // ...
}
```

Unfortunately, overuse of new (and of pointers and references) seems to be an increasing problem.

However, when you really need the semantics of pointers, unique_ptr is a very light-weight mechanism with no space or time overhead compared to correct use of a built-in pointer. Its further uses include passing free-store allocated objects in and out of functions:

```
unique_ptr<X> make_X(int i)
        // make an X and immediately give it to a unique_ptr
{
        // check i, etc.
        return unique_ptr<X>{new X{i}};
}
```

A unique_ptr is a handle to an individual object (or an array) in much the same way that a vector is a handle to a sequence of objects. Both control the lifetime of other objects (using RAII) and both rely on move semantics to make return simple and efficient.

The shared_ptr is similar to unique_ptr except that shared_ptrs are copied rather than moved. The shared_ptrs for an object share ownership of an object and that object is destroyed when the last of its shared_ptrs is destroyed. For example:

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);
void h(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
{
        shared_ptr<fstream> fp {new fstream(name,mode)};
        if (!*fp) throw No_file{};      // make sure the file was properly opened

        f(fp);
        g(fp);
        h(fp);
        // ...
}
```

Now, the file opened by `fp`'s constructor will be closed by the last function to (explicitly or implicitly) destroy a copy of `fp`. Note that `f()`, `g()`, or `h()` may spawn a task holding a copy of `fp` or in some other way store a copy that outlives `user()`. Thus, `shared_ptr` provides a form of garbage collection that respects the destructor-based resource management of the memory-managed objects. This is neither cost free nor exorbitantly expensive, but does make the lifetime of the shared object hard to predict. Use `shared_ptr` only if you actually need shared ownership.

Given `unique_ptr` and `shared_ptr`, we can implement a complete ''no naked `new`'' policy (§3.2.1.2) for many programs. However, these ''smart pointers'' are still conceptually pointers and therefore only my second choice for resource management – after containers and other types that manage their resources at a higher conceptual level. In particular, `shared_ptr`s do not in themselves provide any rules for which of their owners can read and/or write the shared object. Data races (§41.2.4) and other forms of confusion are not addressed simply by eliminating the resource management issues.

Where do we use ''smart pointers'' (such as `unique_ptr`) rather than resource handles with operations designed specifically for the resource (such as `vector` or `thread`)? Unsurprisingly, the answer is ''when we need pointer semantics.''

- When we share an object, we need pointers (or references) to refer to the shared object, so `shared_ptr` becomes the obvious choice (unless there is an obvious single owner).
- When we refer to a polymorphic object, we need a pointer (or a reference) because we don't know the exact type of the object or even its size), so `unique_ptr` becomes the obvious choice.
- A shared polymorphic object typically requires `shared_ptr`s.

We do *not* need to use a pointer to return a collection of objects from a function; a container that is a resource handle will do that simply and efficiently (§3.3.2).

5.3 Concurrency [tour4.concurrency]

Concurrency – the execution of several tasks simultaneously – is widely used to improve throughput (by using several processors for a single computation) or to improve responsiveness (by allowing one part of a program to progress while another is waiting for a response). All modern programming languages provide support for this. The support provided by the C++ standard library is a portable and type-safe variant of what has been used in C++ for more than 20 years and is almost universally supported by modern hardware. The standard-library support is primarily aimed at supporting systems-level concurrency rather than directly providing sophisticated higher-level concurrency models; those can be supplied as libraries built using the standard-library facilities.

The standard library directly supports concurrent execution of multiple threads in a single address space. To allow that, C++ provides a suitable memory model (§41.2) and a set of atomic operations (§41.3). However, most users will see concurrency only in terms of the standard library and libraries built on top of that. This section briefly gives examples of the main standard-library concurrency support facilities: threads, mutexes, lock() operations, packaged_tasks, and futures. These features are built directly upon what operating systems offer and do not incur performance penalties compared with those.

5.3.1 Tasks and threads [tour4.thread]

We call a computation that can potentially be executed concurrently with other computations a *task*. A *thread* is the system-level representation of a task in a program. A task to be executed concurrently with other tasks is launched by constructing a std::thread (found in <thread>) with the task as its argument. A task is a function or a function object:

```
void f();                   // function

struct F {                  // function object
    void operator()();      // F's call operator (§3.4.3)
};

void user()
{
    thread t1 {f};          // f() executes in separate thread
    thread t2 {F()};        // F()() executes in separate thread

    t1.join();              // wait for t1
    t2.join();              // wait for t2
}
```

The join()s ensure that we don't exit user() until the threads have completed. To ''join'' means to ''wait for the thread to terminate.''

Threads of a program share a single address space. In this, threads differ from processes, which generally do not directly share data. Since threads share an address space, they can communicate through shared objects (§5.3.4). Such communication is typically controlled by locks or other mechanisms to prevent data races (uncontrolled concurrent

access to a variable).

Programming concurrent tasks can be *very* tricky. Consider possible implementations of f and F:

```
void f() { cout << "Hello "; }

struct F {
        void operator()() { cout << "Parallel World!\n"; }
};
```

This is an example of a bad error: Here, f and F() each use the object cout without any form of synchronization. The resulting output would be unpredictable and could vary between different executions of the program because the order of execution of the individual operations in the two tasks is not defined. The program may crash because cout was corrupted or produce ‘‘odd’’ output, such as

```
PaHeralllel o World!
```

When defining tasks of a concurrent program, our aim is to keep tasks completely separate except where they communicate in simple and obvious ways. The simplest way of thinking of a concurrent task is as a function that happens to run concurrently with its caller. For that to work, we just have to pass arguments, get a result back, and make sure that there is no use of shared data in between (no data races).

5.3.2 Passing Arguments [tour4.passing]

Typically, a task needs data to work upon. We can easily pass data (or pointers or references to the data) as arguments. Consider:

```
void f(vector<double>& v);          // function do something with v

struct F {                          // function object: do something with v
        vector<double>& v;
        F(vector<double>& vv) :v{vv} { }
        void operator()();          // application operator; §3.4.3
};

int main()
{
        vector<double> some_vec {1,2,3,4,5,6,7,8,9};
        vector<double> vec2 {10,11,12,13,14};

        thread t1 {f,some_vec};     // f(some_vec) executes in a separate thread
        thread t2 {F{vec2}};        // F(vec2)() executes in a separate thread

        t1.join();
        t2.join();
}
```

Obviously, F{vec2} saves a reference to the argument vector in F. F can now use that array

and hopefully no other task accesses vec2 while F is executing. Passing vec2 by value would eliminate that risk.

The initialization with {f,some_vec} uses a thread variadic template constructor that can accept an arbitrary sequence of arguments (§28.6). The compiler checks that the first argument can be invoked given the following arguments and builds the necessary function object to pass to the thread. Thus, if F::operator()() and f() perform the same algorithm, the handling of the two tasks are roughly equivalent: in both cases, a function object is constructed for the thread to execute.

5.3.3 Returning Results [tour4.results]

In the example in §5.3.2, I pass the arguments by non-const reference. I only do that if I expect the task to modify the value of the data referred to (§7.7). That's a somewhat sneaky, but not uncommon, way of returning a result. A less obscure technique is to pass the input data by const reference and to pass the location of a place to deposit the result as a separate argument:

```
void f(const vector<double>& v, double* res);     // take input from v;  place result in *res

class F {
public:
        F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
        void operator()();                  // place result in *res
private:
        const vector<double>& v;            // source of input
        double* res;                        // target for output
};

int main()
{
        vector<double> some_vec;
        vector<double> vec2;
        // ...

        double res1;
        double res2;

        thread t1 {f,some_vec,&res1};   // f(some_vec,&res1) executes in a separate thread
        thread t2 {F{vec2,&res2}};      // F{vec2,&res2}() executes in a separate thread

        t1.join();
        t2.join();

        cout << res1 << ' ' << res2 << '\n';
}
```

I don't consider returning results through arguments particularly elegant, so I return to this topic in §5.3.5.1.

5.3.4 Sharing Data [tour4.sharing]

Sometimes tasks need to share data. In that case, the access has to be synchronized so that at most one task at a time has access. Experienced programmers will recognize this as a simplification (e.g., there is no problem with many tasks simultaneously reading immutable data), but consider how to ensure that at most one task at a time has access to a given set of objects.

The fundamental element of the solution is a mutex, a "mutual exclusion object." A thread acquires a mutex using a lock() operation:

```
mutex m;  // controlling mutex
int sh;   // shared data

void f()
{
        lock_guard<mutex> lck {m};      // acquire mutex
        sh += 7;                        // manipulate shared data
}       // release mutex implicitly
```

The lock_guard's constructor acquires the mutex (through a call m.lock()). If another thread has already acquired the mutex, the thread waits ("blocks") until the other thread completes its access. Once a thread has completed its access to the shared data, the lock_guard releases the mutex (with a call m.unlock()). The mutual exclusion and locking facilities are found in <mutex>.

The correspondence between the shared data and a mutex is conventional: The programmer simply has to know which mutex is supposed to correspond to which data. Obviously, this is error-prone, and equally obviously we try to make the correspondence clear through various language means. For example:

```
class Record {
public:
        mutex rm;
        // ...
};
```

It doesn't take a genius to guess that for a Record called rec, rec.rm is a mutex that you are supposed to acquire before accessing the other data of rec, though a comment or a better name might have helped a reader.

It is not uncommon to need to simultaneously access several resources to perform some action. This can lead to deadlock. For example, if thread1 acquires mutex1 and then tries to acquire mutex2 while thread2 acquires mutex2 and then tries to acquire mutex1, then neither task will ever proceed further. The standard library offers help in the form of an operation for acquiring several locks simultaneously:

```
void f()
{
    // ...
    lock_guard<mutex> lck1 {m1,defer_lock};   // defer_lock: don't yet try to acquire the mutex
    lock_guard<mutex> lck2 {m2,defer_lock};
    lock_guard<mutex> lck3 {m3,defer_lock};
    // ...
    lock(lck1,lck2,lck3);                     // acquire all three locks
    // ... manipulate shared data ...
} // implicitly release all mutexes
```

This lock() will only proceed after acquiring all its mutex arguments and will never block (''go to sleep'') while holding a mutex. The destructors for the individual lock_guards ensure that the mutexes are released when a thread leaves the scope.

Communicating through shared data is pretty low level. In particular, the programmer has to devise ways of knowing what work has and has not been done by various tasks. In that regard, use of shared data is inferior to the notion of call and return. On the other hand, some people are convinced that sharing must be more efficient than copying arguments and returns. That can indeed be so when large amounts of data are involved, but locking and unlocking are relatively expensive operations. On the other hand, modern machines are very good at copying data, especially compact data, such as vector elements. So don't choose shared data for communication because of ''efficiency'' without thought and preferably not without measurement.

5.3.4.1 Waiting for Events [tour4.condition]

Sometimes, a thread needs to wait for some kind of external event, such as another thread completing a task or a certain amount of time having passed. The simplest ''event'' is simply time passing. Consider:

```
using namespace std::chrono;   // see §35.2

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();
cout << nanoseconds(t1−t0).count() << " nanoseconds passed\n";
```

Note that I didn't even have to launch a thread; by default, this_thread refers to the one and only thread (§42.2.6).

See _tour4.time_ and §35.2 before trying anything more complicated than this with time. The time facilities are found in <chrono>.

The basic support for communicating using external events is provided by condition_variables found in <condition_variable> (§42.3.4). A condition_variable is a mechanism allowing one thread to wait for another. In particular, it allows a thread to wait for some *condition* (often called an *event*) to occur as the result of work done by other threads.

Consider the classical example of two threads communicating by passing messages through a queue. For simplicity, I declare the queue and the mechanism for avoiding race

conditions on that queue global to the producer and consumer:

```
class Message {        // object to be communicated
      // ...
};

queue<Message> mqueue;      // the queue of messages
condition_variable mcond;   // the variable communicating events
mutex mmutex;               // the locking mechanism
```

The types queue, condition_variable, and mutex are provided by the standard library.

The consumer() reads and processes Messages:

```
void consumer()
{
      while(true) {
            unique_lock<mutex> lck{mmutex};    // acquire mmutex
            mcond.wait(lck);                   // release lck and wait;
                                               // re-acquire lck upon wakeup
            auto m = mqueue.top();             // get the message
            mqueue.pop();
            lck.unlock();                      // release lck
            // ... process m ...
      }
}
```

Here, I explicitly protect the operations on the queue and on the condition_variable with a unique_lock on the mutex. Waiting on condition_variable releases its lock argument until the wait is over (so that the queue is non-empty) and then reacquires it.

The corresponding producer looks like this:

```
void producer()
{
      while(true) {
            Message m;
            // ... fill the message ...
            unique_lock<mutex> lck {mmutex};   // protect operations
            mqueue.push(m);
            mcond.notify_one();                // notify
      }                                        // release lock (at end of scope)
}
```

Using condition_variables supports many forms of elegant and efficient sharing, but can be rather tricky (§42.3.4).

5.3.5 Communicating Tasks [tour4.task]

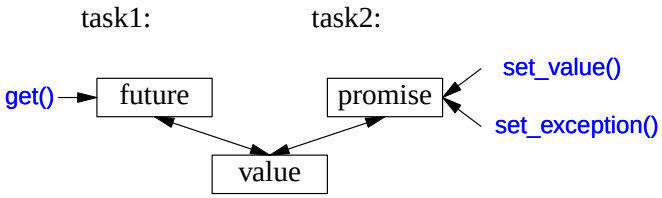
The standard library provides a few facilities to allow programmers to operate at the conceptual level of tasks (work to potentially be done concurrently) rather than directly at the lower level of threads and locks:

[1] future and promise for returning a value from a task spawned on a separate thread
[2] packaged_task to help launch tasks and connect up the mechanisms for returning a result
[3] async() for launching of a task in a manner very similar to calling a function.

These facilities are found in <future>.

5.3.5.1 future and promise [tour4.future]

The important point about future and promise is that they enable a transfer of a value between two tasks without explicit use of a lock; "the system" implements the transfer efficiently. The basic idea is simple: When a task wants to pass a value to another, it puts the value into a promise. Somehow, the implementation makes that value appear in the corresponding future, from which it can be read (typically by the launcher of the task). We can represent this graphically:



If we have a future<X> called fx, we can get() a value of type X from it:

```
X v = fx.get();    // if necessary, wait for the value to get computed
```

If the value isn't there yet, our thread is blocked until it arrives. If the value couldn't be computed, get() might throw an exception (from the system or transmitted from the task from which we were trying to get() the value).

The main purpose of a promise is to provide simple "put" operations (called set_value() and set_exception()) to match future's get(). The names "future" and "promise" are historical; please don't blame me. They are yet another fertile source of puns.

If you have a promise and need to send a result of type X to a future, you can do one of two things: pass a value or pass an exception. For example:

```
void f(promise<X>& px)     // a task: place the result in px
{
        // ...
        try {
                X res;
                // ... compute a value for res ...
                px.set_value(res);
        }
```

```
        catch (...) {       // oops: couldn't compute res
            // pass the exception to the future's thread:
            px.set_exception(current_exception());
        }
    }
```

The current_exception() refers to the caught exception (§30.4.1.2).

To deal with an exception transmitted through a future, the caller of get() must be prepared to catch it somewhere. For example:

```
void g(future<X>& fx)       // a task: get the result from fx
{
    // ...
    try {
        X v = fx.get();    // if necessary, wait for the value to get computed
        // ... use v ...
    }
    catch (...) {       // oops: someone couldn't compute v
        // ... handle error ...
    }
}
```

5.3.5.2 packaged_task [tour4.packaged]

How do we get a future into the task that needs a result and the corresponding promise into the thread that should produce that result? The packaged_task type is provided to simplify setting up tasks connected with futures and promises to be run on threads. A packaged_task provides wrapper code to put the return value or exception from the task into a promise (like the code shown in §5.3.5.1). If you ask it, the packaged_task will give you the corresponding future. For example, we can set up two tasks to each add half of the elements of a vector<double> using the standard-library accumulate() (§3.4.2, §40.6):

```
double accum(double* beg, double * end, double init)
    // compute the sum of [beg:end) starting with the initial value init;
{
    return accumulate(beg,end,init);
}

double comp2(vector<double>& v)
{
    using Task_type = double(double*,double*,double);          // type of task

    packaged_task<Task_type> pt0 {accum};   // package the task (i.e., accum)
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};   // get hold of pt0's future
    future<double> f1 {pt1.get_future()};   // get hold of pt1's future
```

```
        double* first = &v[0];
        thread t1 {move(pt0),first,first+v.size()/2,0};          // start a thread for pt0
        thread t2 {move(pt1),first+v.size()/2,first+v.size(),0}; // start a thread for pt1

        // ...

        return f0.get()+f1.get();          // get the results
}
```

The packaged_task template takes the type of the task as its template argument (here Task_type, an alias for double(double*,double*,double)) and the task as its constructor argument (here, accum). The move() operations are needed because a packaged_task cannot be copied.

Please note the absence of explicit mention of locks in this code: we are able to concentrate on tasks to be done, rather than on the mechanisms used to manage their communication. The two tasks will be run on separate threads and thus potentially in parallel.

5.3.5.3 async() [tour4.async]

The line of thinking I have pursued in this chapter is the one I believe to be the simplest yet still among the most powerful: Treat a task as a function that may happen to run concurrently with other tasks. It is far from the only model supported by the C++ standard library, but it serves well for a wide range of needs. More subtle and tricky models, e.g., styles of programming relying on shared memory, can be used as needed.

The standard-library function async() provides a very simple way of executing a task asynchronously:

```
double comp4(vector<double>& v)
        // spawn many tasks if v is large enough
{
        if (v.size()<10000) return accum(v.begin(),v.end(),0.0);

        auto v0 = &v[0];
        auto sz = v.size();

        auto f0 = async(accum,v0,v0+sz/4,0.0);          // first quarter
        auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0);     // second quarter
        auto f2 = async(accum,v0+sz/2,v0+sz*3/4,0.0);   // third quarter
        auto f3 = async(accum,v0+sz*3/4,v0+sz,0.0);     // fourth quarter

        return f0.get()+f1.get()+f2.get()+f3.get();     // collect and combine the results
}
```

Basically, async() separates the ''call part'' of a function call from the ''get the result part,'' and separates both from the actual execution of the task. Using async(), you don't have to think about threads and locks. Instead, you think just in terms of tasks that potentially compute their results asynchronously. There is an obvious limitation: Don't even think of using async() for tasks that share resources needing locking – with async() you don't even

know how many threads will be used because that's up to async() to decide based on what it knows about the system resources available at the time of a call. For example, async() may check whether any idle cores (processors) are available before deciding how many threads to use.

Please note that async() is not just a mechanism specialized for parallel computation for increased performance. For example, it can also be used to spawn a task for getting information from a user, leaving the ''main program'' active with something else (§42.4.6).

5.4 Small Utility Components [tour4.utilities]

Not all standard-library components come as part of obviously labeled facilities, such as ''containers'' or ''I/O.'' This section gives a few examples of small, widely useful components:

- clock and duration for measuring time.
- Type functions, such as iterator_traits and is_arithmetic, for gaining information about types.
- pair and tuple for representing small potentially heterogeneous sets of values.

The point here is that a function or a type need not be complicated or closely tied to a mass of other functions and types to be useful. Such library components mostly act as building blocks for more powerful library facilities, including other components of the standard library.

5.4.1 Time [tour4.clock]

The standard library provides facilities for dealing with time. For example, here is the basic way of timing something:

```
using namespace std::chrono;    // see §35.2

auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1−t0).count() << "msec\n";
```

The clock returns a time_point (a point in time). Subtracting two time_points gives a duration (a period of time). Various clocks give their results in various units of time (the clock I used measures nanoseconds), so it is usually a good idea to convert a duration into a known unit. That's what duration_cast does.

The standard-library facilities for dealing with time are found in the subnamespace std::chrono in <chrono> (§35.2).

Don't make statements about ''efficiency'' of code without first doing time measurements. Guesses about performance are most unreliable.

5.4.2 Type Functions [tour4.typetraits]

A *type function* is a function that is evaluated at compile-time given a type as its argument or returning a type. The standard library provides a variety of type functions to help library implementers and programmers in general to write code that take advantage of aspects of the language, the standard library, and code in general.

For numerical types, numeric_limits from <limits> presents useful information (§5.6.5). For example:

```
constexpr float min = numeric_limits<float>::min();      // smallest positive float (§40.2)
```

Similarly, information about sizes can be extracted by the built-in sizeof operator (§2.2.2). For example:

```
constexpr int szi = sizeof(int);     // the number of bytes in an int
```

Such type functions are part of C++'s mechanisms for compile-time computation that allow tighter type checking and better performance than would otherwise have been possible. Use of such features is often called *metaprogramming* or (when templates are involved) *template metaprogramming* (Chapter 28). Here, I just present two facilities provided by the standard library: iterator_traits (§5.4.2.1) and type predicates (§5.4.2.2).

5.4.2.1 iterator_traits [tour4.iteratortraits]

The standard-library sort() takes a pair of iterators supposed to define a sequence (§4.5). Furthermore, those iterators must offer random access to that sequence, that is, they must be *random-access iterators*. Some containers, such as forward_list, do not offer that. In particular, a forward_list is a singly-linked list so subscripting would be expensive and there is no reasonable way to refer back to a previous element. However, like most containers, forward_list offers *forward iterators* that can be used to traverse the sequence by algorithms and for-statements (§33.1.1).

The standard library provides a mechanism, iterator_traits that allows us to check which kind of iterator is supported. Given that, we can improve the range sort() from §4.5.6 to accept either a vector or a forward_list. For example:

```
void test(vector<string>& v, forward_list<int>& lst)
{
        sort(v);     // sort the vector
        sort(lst);   // sort the singly-linked list
}
```

The techniques needed to make that work are generally useful.

First, I write two helper functions that take an extra argument indicating whether they are to be used for random-access iterators or forward iterators. The version for random-access iterators is trivial:

```
template<typename Ran>          // for random-access iterators
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)
        // we can subscript into [beg:end)
{
        sort(beg,end);   // just sort it

}
```

The version for forward iterators is almost as simple; just copy the list into a `vector`, sort, and copy back again:

```
template<typename For>          // for forward iterators
void sort_helper(For beg, For end, forward_iterator_tag)
        // we can traverse [beg:end)
{
        vector<decltype(*beg)> v {beg,end};   // initialize a vector from [beg:end)
        sort(v.begin(),v.end());
        copy(v.begin(),v.end(),beg);              // copy the elements back
}
```

The `decltype()` is a built-in type function that returns the declared type of its argument (§6.3.6.3). Thus, `v` is a `vector<X>` where `X` is the element type of the input sequence.

The real ''type magic'' is in the selection of helper functions:

```
template<class C)
void sort(C& c)
{
        using Iter = Iterator_type<C>;
        sort_helper(c.begin(),c.end(),Iterator_category<Iter>{});
}
```

Here, I use two type functions: `Iterator_type<C>` returns the iterator type of `C` (that is, `C::iterator`) and then `Iterator_category<Iter>{}` constructs a ''tag'' value indicating the kind of iterator provided:

- `std::random_access_iterator_tag` if `C`'s iterator supports random access.
- `std::forward_iterator_tag` if `C`'s iterator supports forward iteration.

Given that, we can select between the two sorting algorithms at compile time. This technique, called *tag dispatch* is one of several used in the standard library and elsewhere to improve flexibility and performance.

The standard-library support for techniques for using iterators, such as tag dispatch, comes in the form of a simple class template `iterator_traits` from `<iterator>` (§33.1.3). This allows simple definitions of the type functions used in `sort()`:

```
template<typename C>
        using Iterator_type = typename C::iterator; // C's iterator type

template<typename Iter>
        using Iterator_category = typename std::iterator_traits<Iter>::iterator_category;    // Iter's category
```

If you don't want to know what kind of ''compile-time type magic'' is used to provide the

standard-library features, you are free to ignore facilities such as iterator_traits. But then you can't use the techniques they support to improve your own code.

5.4.2.2 Type Predicates [tour4.typepredicates]

A standard-library type predicate is a simple type function that answers a fundamental question about types. For example:

```
bool b1 = Is_arithmetic<int>();          // yes, int is an arithmetic type
bool b2 = Is_arithmetic<string>();       // no, std::string is not an arithmetic type
```

These predicates are found in <type_traits> and described in §35.4.1. Other examples are is_class, is_pod, is_literal_type, has_virtual_destructor, and is_base_of. They are most useful when we write templates. For example:

```
template<typename Scalar>
class complex {
        Scalar re, im;
public:
        static_assert(Is_arithmetic<Scalar>(), "Sorry, I only support complex of arithmetic types");
        // ...
};
```

To improve readability compared to using the standard library directly, I defined a type function:

```
template<typename T>
constexpr bool Is_arithmetic()
{
        return std::is_arithmetic<T>::value ;
}
```

Older programs use ::value directly instead of (), but I consider that quite ugly and it exposes implementation details.

5.4.3 pair and tuple [tour4.pair]

Often, we need some data that is just data; that is, a collection of values, rather than an object of a class with a well-defined semantics and an invariant for its value (§2.4.3.2, §13.4). In such cases, we could define a simple struct with an appropriate set of appropriately named members. Alternatively, we could let the standard library write the definition for us. For example, the standard-library algorithm equal_range (§32.6.1) returns a pair of iterators specifying a sub-sequence meeting a predicate:

```
template<typename Forward_iterator, typename T, typename Compare>
        pair<Forward_iterator,Forward_iterator>
        equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare cmp);
```

Given a sorted sequence [first:last), equal_range() will return the pair representing the subsequence that matches the predicate cmp. We can use that to search in a sorted sequence of Records:

```
void f(const vector<Record>& v)
{
        // assume that v is sorted on its "name" field
        auto er = equal_range(v.begin(),v.end(), "Reg",
                                    [](const Record& r1, const Record& r2) { return r1.name==r2.name;}
                                   );
        for (auto p = er.first; p!=er.second; ++p)          // print all equal records
              cout << *p;                                   // assume that << is defined for Record
}
```

The first member of a `pair` is called `first` and the second member is called `second`. This naming is not particularly creative and may look a bit odd at first, but such consistent naming is a boon when we want to write generic code.

The standard-library `pair` (from `<utilty>`) is quite frequently used in the standard library and elsewhere. A `pair` provides operators, such as `=`, `==`, and `<`, if its elements do. The `make_pair()` function makes it easy to create a `pair` without explicitly mentioning its type (§34.2.4.1). For example:

```
void f(vector<string>& v)
{
        auto pp = make_pair(v.begin(),2);       // pp is a pair<vector<string>::iterator,int>
        // ...
}
```

If you need more than two elements (or less), you can use `tuple` (from `<utility>`; §34.2.4.2). A `tuple` is a heterogeneous sequence of elements; for example:

```
tuple<string,int,double> t2("Sild",123, 3.14);      // the type is explicitly specified

auto t = make_tuple(string("Herring"),10, 1.23);  // the type is deduced
                                                  // t is a tuple<string,int,double>

string s = get<0>(t);   // get first element of tuple
int x = get<1>(t);
double d = get<2>(t);
```

The elements of a `tuple` are numbered (starting with zero), rather than named the way elements of `pair`s are (`first` and `second`). To get compile-time selection of elements, I must unfortunately use the ugly `get<1>(t)`, rather than `get(t,1)` or `t[1]` (§28.5.2).

Like `pair`s, `tuple`s can be assigned and compared if their elements can be.

A `pair` is common in interfaces because often we want to return more than one value, such as a result and an indicator of the quality of that result. It is less common to need three or more parts to a result, so `tuple`s are more often found in the implementations of generic algorithms.

5.5 Regular Expressions [tour4.regex]

Regular expressions are a powerful tool for text processing. They provide a way to simply and tersely describe patterns in text (e.g., a U.S. ZIP code such as TX 77845, or an ISO-style date, such as 2009−06−07) and to efficiently find such patterns in text. In <regex>, the standard library provides support for regular expressions in the form of the std::regex class and its supporting functions. To give a taste of the style of the regex library, let us define and print a pattern:

```
regex pat (R"(\w{2}\s*\d{5}(−\d{4})?)");      // ZIP code pattern: XXddddd-dddd and variants
cout << "pattern: " << pat << '\n';
```

People who have used regular expressions in just about any language will find \w{2}\s*\d{5}(−\d{4})? familiar. It specifies a pattern starting with two letters \w{2} optionally followed by some space \s* followed by five digits \d{5} and optionally followed by a dash and four digits −\d{4}. If you are not familiar with regular expressions, this may be a good time to learn about them ([Stroustrup 2009], [Maddock,2009], [Friedl,1997]). Regular expressions are summarized in §37.1.1.

To express the pattern, I used a *raw string literal* (§7.3.2.1) starting with a R"(and terminated by)" . This allows backslashes and quotes to be represented in the string without the use of special notation.

The simplest way of using a pattern is to search for it in a stream:

```
int lineno = 0;
for (string line; getline(cin,line);) {          // read into line buffer
        ++lineno;
        smatch matches;                          // matched strings go here
        if (regex_search(line,matches,pat))      // search for pat in line
                cout << lineno << ": " << matches[0] << '\n';
}
```

The regex_search(line,matches,pat) searches the line for anything that matches the regular expression stored in pat and if it finds any matches, it stores them in matches. If no match was found, regex_search(line,matches,pat) returns false. The matches variable is of type smatch. The ‘‘s’’ stands for ‘‘sub’’ and an smatch is a vector of sub-matches. The first element, here matches[0], is the complete match.

For a more complete description see Chapter 37.

5.6 Math [tour4.math]

C++ wasn’t designed primarily with numerical computation in mind. However, C++ is heavily used for numerical computation and the standard library reflects that.

5.6.1 Mathematical Functions and Algorithms [tour4.stdmath]

In <cmath>, we find the ‘‘usual mathematical functions,’’ such as sqrt(), log(), and sin() for arguments of type float, double, and long double (§40.3). Their complex number versions are found in <complex> (§40.4).

In <numeric> we find a small set of generalized numerical algorithms, such as accumulate(). For example:

```
list<double> lst {1, 2, 3, 4, 5, 6 , 9999.99999};
auto s = accumulate(lst.begin(),lst.end(),0.0);
cout << s << '\n';
```

These algorithms work for every standard-library sequence and can have operations supplied as arguments (§40.6).

5.6.2 Complex Numbers [tour4.complex]

The standard library supports a family of complex number types along the lines of the complex class described in §2.3. To support complex numbers where the scalars are single-precision floating-point numbers (floats), double-precision floating-point numbers (doubles), etc., the standard library complex is a template:

```
template<typename Scalar>
class complex {
public:
        complex(const Scalar& re ={}, const Scalar& im ={});
        // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers. For example:

```
void f(complex<float> fl, complex<double> db)
{
        complex<long double> ld {fl+sqrt(db)};
        db += fl*3;
        fl = pow(1/fl,2);
        // ...
}
```

The sqrt() and pow() (exponentiation) functions are among the usual mathematical functions defined in <complex>. For more details, see §40.4.

5.6.3 Random Numbers [tour4.random]

Random numbers are useful in many contexts, such as testing, games, simulation, and security. The diversity of application areas is reflected in the wide selection of random number generators provided by the standard library in <random>. A random number generator consists of two parts:

[1] an *engine* that produces a sequence of random or pseudo-random values.
[2] a *distribution* that maps those values into a mathematical distribution in a range.

Examples of distributions are uniform_int_distribution (where all integers produced are equally likely), normal_distribution (“the bell curve”), and exponential_distribution (exponential growth); each for some specified range. For example:

```
using my_engine = default_random_engine;            // type of engine
using my_distribution = uniform_int_distribution<>; // type of distribution

my_engine re {};                                    // the default engine
my_distribution one_to_six {1,6};                   // distribution that maps to the ints 1..6
auto dice = bind(one_to_six,re);                    // make a generator

int x = dice();    // roll the dice: x becomes a value in [1:6]
```

The standard-library function `bind()` makes a function object that will invoke its first argument (here, `one_to_six`) given its second argument (here, `re`) as its argument (§33.5.1). Thus a call `dice()` is equivalent to a call `one_to_six(re)`.

Thanks to its uncompromising attention to generality and performance one expert has deemed the standard-library random number component ''what every random number library wants to be when it grows up.'' However, it can hardly be deemed ''novice friendly.'' The `using` statements makes what is being done a bit more obvious. Instead, I could just have written:

```
auto dice = bind(uniform_int_distribution<>{1,6}, default_random_engine{});
```

Which version is the more readable depends entirely on the context and the reader.

For novices (of any background) the fully general interface to the random number library can be a serious obstacle. A simple uniform random number generator is often sufficient to get started. For example:

```
Rand_int rnd {1,10};      // make a random number generator for [1:10]
int x = rnd();            // x is a number in [1:10]
```

So, how could we get that? We have to get something like `dice()` inside a class `Rand_int`:

```
class Rand_int {
public:
    Rand_int(int low, int high) :dist{low,high} { }
    int operator()() { return r(); }
private:
    default_random_engine re;
    uniform_int_distribution<> dist;
    auto r = bind(dist,re);
};
```

That definition is still ''expert level,'' but the *use* of `Rand_int()` is manageable in the first week of a C++ course for novices. For example:

```
int main()
{
    Rand_int rnd {0,9};         // make a uniform random number generator

    vector<int> mn(10);         // make a vector of size 10
    for (int i=0; i!=500; ++i)
        ++mn[rnd()];            // fill mn with the frequencies of numbers [0:9]
```

```
        for (int i = 0; i!=mn.size(); ++i) {   // write out a bar graph
                cout << i << '\t';
                for (int j=0; j!=mn[i]; ++j) cout << '*';
                cout << endl;
        }
}
```

The output is a (reassuringly boring) uniform distribution (with reasonable statistical variation):

```
0    **************************************************
1    **************************************************
2    **************************************************
3    **************************************************
4    **************************************************
5    **************************************************
6    **************************************************
7    **************************************************
8    **************************************************
9    **************************************************
```

There is no standard graphics library for C++, so I use "ASCII graphics." Obviously, there are lots of open source and commercial graphics and GUI libraries for C++, but in this book I'll restrict myself to ISO standard facilities.

For more information about random numbers, see §40.7.

5.6.4 Vector Arithmetic [tour4.valarray]

The vector described in §4.4.1 was designed to be a general mechanism for holding values, to be flexible, and to fit into the architecture of containers, iterators, and algorithms. However, it does not support mathematical vector operations. Adding such operations to vector would be easy, but its generality and flexibility precludes optimizations that are often considered essential for serious numerical work. Consequently, the standard library provides (in <valarray>) a vector-like template, called valarray, that is less general and more amenable to optimization for numerical computation:

```
template<typename T>
class valarray {
        // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for valarrays. For example:

```
void f(valarray<double>& a1, valarray<double>& a2)
{
        valarray<double> a = a1*3.14+a2/a1;        // numeric array operators *, +, /, and =
        a2 += a1*3.14;
        a = abs(a);
        double d = a2[7];
        // ...
}
```

For more details, see §40.5. In particular, valarray offers stride access to help implement multidimensional computations.

5.6.5 Numeric Limits [tour4.limits]

In <limits>, the standard library provides classes that describe the properties of built-in types – such as the maximum exponent of a float or the number of bytes in an int; see §40.2. For example, we can assert that a char is signed:

```
static_assert(numeric_limits<char>::is_signed,"unsigned characters!");
static_assert(100000<numeric_limits<int>::max(),"small ints!");
```

Note that the second assert (only) works because numeric_limits<int>::max() is a constexpr function (§2.2.3, §10.4).

5.7 Advice [tour4.advice]

[1] Use resource handles to manage resources (RAII); §5.2.
[1] Use unique_ptr to refer to objects of polymorphic type;§5.2.1.
[2] Use shared_ptr to refer to shared objects;§5.2.1.
[3] Use type-safe mechanisms for concurrency; §5.3.
[4] Minimize the use of shared data; §5.3.4.
[5] Don’t choose shared data for communication because of ‘‘efficiency’’ without thought and preferably not without measurement; §5.3.4.
[6] Think in terms of concurrent tasks, rather than threads; §5.3.5.
[7] A library doesn’t have to be large or complicated to be useful; §5.4.
[8] Time your programs before making claims about efficiency; §5.4.1.
[9] You can write code to explicitly depend of properties of types; §5.4.2.
[10] Use regular expressions for simple pattern matching §5.5.
[11] Don’t try to do serious numeric computation using only the bare language; use libraries; §5.6.
[12] Properties of numeric types are accessible through numeric_limits; §5.6.5.

blank page