25 Embedded Systems Programming

"'Unsafe' means 'Somebody may die.'"

—**Safety officer**

We present a view of embedded systems programming; that is, we discuss topics primarily related to writing programs for "gadgets" that do not look like conventional computers with screens and keyboards. We focus on the principles, programming techniques, language facilities, and coding standards needed to work "close to the hardware." The main language issues addressed are resource management, memory management, pointer and array use, and bit manipulation. The emphasis is on safe use and on alternatives to the use of the lowest-level features. We do not attempt to present specialized machine architectures or direct access to hardware devices; that is what specialized literature and manuals are for. As an example, we present the implementation of an encryption/decryption algorithm.

Stroustrup_book.indb 925 5/8/15 10:31 AM

25.1 Embedded systems

25.2 Basic concepts
- 25.2.1 Predictability
- 25.2.2 Ideals
- 25.2.3 Living with failure

25.3 Memory management
- 25.3.1 Free-store problems
- 25.3.2 Alternatives to the general free store
- 25.3.3 Pool example
- 25.3.4 Stack example

25.4 Addresses, pointers, and arrays
- 25.4.1 Unchecked conversions
- 25.4.2 A problem: dysfunctional interfaces
- 25.4.3 A solution: an interface class
- 25.4.4 Inheritance and containers

25.5 Bits, bytes, and words
- 25.5.1 Bits and bit operations
- 25.5.2 bitset
- 25.5.3 Signed and unsigned
- 25.5.4 Bit manipulation
- 25.5.5 Bitfields
- 25.5.6 An example: simple encryption

25.6 Coding standards
- 25.6.1 What should a coding standard be?
- 25.6.2 Sample rules
- 25.6.3 Real coding standards

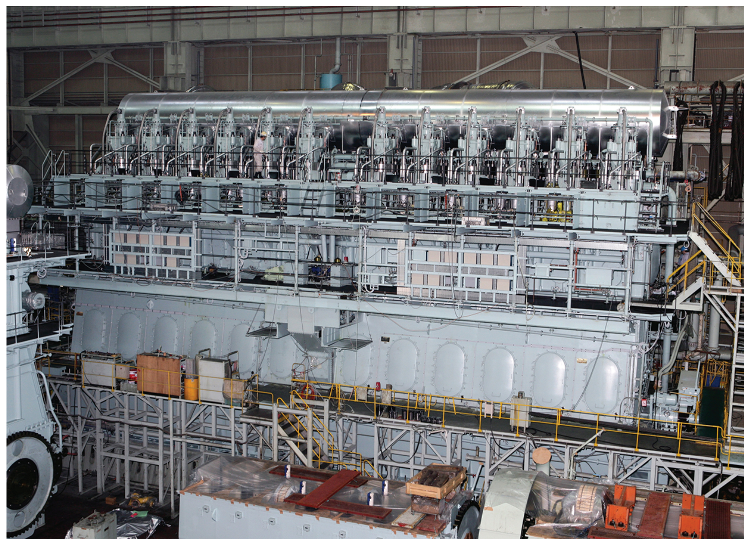
25.1 Embedded systems

Most computers in the world are not immediately recognizable as computers. They are simply a part of a larger system or "gadget." For example:

- *Cars:* A modern car may have many dozens of computers, controlling the fuel injection, monitoring engine performance, adjusting the radio, controlling the brakes, watching for underinflated tires, controlling the windshield wipers, etc.
- *Telephones:* A mobile telephone contains at least two computers; often one of those is specialized for signal processing.
- *Airplanes:* A modern airplane contains computers for everything from running the passenger entertainment system to wiggling the wing tips for optimal flight properties.
- *Cameras:* There are cameras with five processors and for which each lens even has its own separate processor.
- *Credit cards* (of the "smart card" variety)
- *Medical equipment monitors and controllers* (e.g., CAT scanners)
- *Elevators* (lifts)
- *PDAs* (Personal Digital Assistants)
- *Printer controllers*
- *Sound systems*
- *MP3 players*

- *Kitchen appliances* (such as rice cookers and bread machines)
- *Telephone switches* (typically consisting of thousands of specialized computers)
- *Pump controllers* (for water pumps and oil pumps, etc.)
- *Welding robots:* some for use in tight or dangerous places where a human welder cannot go
- *Wind turbines:* some capable of generating megawatts of power and 200m (650ft) tall
- *Sea-wall gate controllers*
- *Assembly-line quality monitors*
- *Bar code readers*
- *Car assembly robots*
- *Centrifuge controllers* (as used in many medical analysis processes)
- *Disk-drive controllers*

These computers are parts of larger systems. Such "large systems" usually don't look like computers and we don't usually think of them as computers. When we see a car coming down the street, we don't say, "Look, there's a distributed computer system!" Well, the car is *also* a distributed computer system, but its operation is so integrated with the mechanical, electronic, and electrical parts that we can't really consider the computers in isolation. The constraints on their computations (in time and space) and the very definition of program correctness cannot be separated from the larger system. Often, an embedded computer controls a physical device, and the correct behavior of the computer is defined as the correct operation of the physical device. Consider a large marine diesel engine:



Note the engineer at the head of cylinder number 5. This is a big engine, the kind of engine that powers the largest ships. If an engine like this fails, you'll read about it on the front page of your morning newspaper. On this engine, a cylinder control system, consisting of three computers, sits on each cylinder head. Each cylinder control system is connected to the engine control system (another three computers) through two independent networks. The engine control system is then connected to the control room where the engineers can communicate with it through a specialized GUI system. The complete system can also be remotely monitored via radio (through satellites) from a shipping-line control center. For more examples, see Chapter 1.

So, from a programmer's point of view, what's special about the programs running in the computers that are parts of that engine? More generally, what are examples of concerns that become prominent for various kinds of embedded systems that we don't typically have to worry too much about for "ordinary programs"?

- Often, *reliability is critical:* Failure can be spectacular, expensive (as in "billions of dollars"), and potentially lethal (for the people on board a wreck or the animals in its environment).
- Often, *resources (memory, processor cycles, power) are limited:* That's not likely to be a problem on the engine computer, but think of smartphones, sensors, computers on board space probes, etc. In a world where dual-processor 2GHz laptops with 8GB of memory are common, a critical computer in an airplane or a space probe may have just 60MHz and 256KB, and a small gadget just sub-1MHz and a few hundred words of RAM. Computers made resilient to environmental hazards (vibration, bumps, unstable electricity supplies, heat, cold, humidity, workers stepping on them, etc.) are typically far slower than what powers a student's laptop.
- Often, *real-time response is essential:* If the fuel injector misses an injection cycle, bad things can happen to a very complex system generating 100,000Hp; miss a few cycles – that is, fail to function correctly for a second or so – and strange things can start happening to propellers that can be up to 33ft (10m) in diameter and weigh up to 130 tons. You really don't want that to happen.
- Often, *a system must function uninterrupted for years:* Maybe the system is running in a communications satellite orbiting the earth, or maybe the system is just so cheap and exists in so many copies that any significant repair rate would ruin its maker (think of MP3 players, credit cards with embedded chips, and automobile fuel injectors). In the United States, the mandated reliability criterion for backbone telephone switches is 20 minutes of downtime in 20 years (don't even think of taking such a switch down each time you want to change its program).

- Often, *hands-on maintenance is infeasible or very rare:* You can take a large ship into a harbor to service the computers every second year or so when other parts of the ship require service and the necessary computer specialists are available in the right place at the right time. Unscheduled, hands-on maintenance is infeasible (no bugs are allowed while the ship is in a major storm in the middle of the Pacific). You simply can't send someone to repair a space probe in orbit around Mars.

Few systems suffer all of these constraints, and any system that suffers even one is the domain of experts. Our aim is not to make you an "instant expert"; attempting to do that would be quite silly and very irresponsible. Our aim is to acquaint you with the basic problems and the basic concepts involved in their solution so that you can appreciate some of the skills needed to build such systems. Maybe you could become interested in acquiring such valuable skills. People who design and implement embedded systems are critical to many aspects of our technological civilization. This is an area where a professional can do a lot of good.

Is this relevant to novices? To C++ programmers? Yes and yes. There are many more embedded systems processors than there are conventional PCs. A huge fraction of programming jobs relate to embedded systems programming, so your first real job may involve embedded systems programming. Furthermore, the list of examples of embedded systems that started this section is drawn from what I have personally seen done using C++.

25.2 Basic concepts

Much programming of computers that are part of an embedded system can be just like other programming, so most of the ideas presented in this book apply. However, the emphasis is often different: we must adjust our use of programming language facilities to the constraints of the task, and often we must manipulate our hardware at the lowest level:

- *Correctness:* This is even more important than usual. "Correctness" is not just an abstract concept. In the context of an embedded system, what it means for a program to be correct becomes not just a question of producing the correct results, but also producing them at the right time, in the right order, and using only an acceptable set of resources. Ideally, the details of what constitutes correctness are carefully specified, but often such a specification can be completed only after some experimentation. Often, critical experiments can be performed only after the complete system (of which the computer running the program is a part) has been built. Completely specifying correctness for an embedded system can at the same time be extremely difficult and extremely important. Here, "extremely difficult" can mean "impossible given the time and resources available";

we must try our best using all available tools and techniques. Fortunately, the range of specification, simulation, testing, and other techniques in a given area can be quite impressive. Here, "extremely important" can mean "failure leads to injury or ruin."

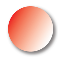
- *Fault tolerance:* We must be careful to specify the set of conditions that a program is supposed to handle. For example, for an ordinary student program, you might find it unfair if we kicked the cord out of the power supply during a demonstration. Losing power is not among the conditions an ordinary PC application is supposed to deal with. However, losing power is not uncommon for embedded systems, and some are expected to deal with that. For example, a critical part of a system may have dual power sources, backup batteries, etc. Worse, "But I assumed that the hardware worked correctly" is no excuse for some applications. Over a long time and over a large range of conditions, hardware simply doesn't work correctly. For example, some telephone switches and some aerospace applications are written based on the assumption that sooner or later some bit in the computer's memory will just "decide" to change its value (e.g., from 0 to 1). Alternatively, it may "decide" that it likes the value 1 and ignore attempts to change that 1 to a 0. Such erroneous behavior happens eventually if you have enough memory and use it for a long enough time. It happens sooner if you expose the memory to hard radiation, such as you find beyond the earth's atmosphere. When we work on a system (embedded or not), we have to decide what kind of tolerance to hardware failure we must provide. The usual default is to assume that hardware works as specified. As we deal with more critical systems, that assumption must be modified.
- *No downtime:* Embedded systems typically have to run for a long time without changes to the software or intervention by a skilled operator with knowledge of the implementation. "A long time" can be days, months, years, or the lifetime of the hardware. This is not unique for embedded systems, but it is a difference from the vast majority of "ordinary applications" and from all examples and exercises in this book (so far). This "must run forever" requirement implies an emphasis on error handling and resource management. What is a "resource"? A resource is something of which a machine has only a limited supply; from a program you acquire a resource through some explicit action ("acquire the resource," "allocate") and return it ("release," "free," "deallocate") to the system explicitly or implicitly. Examples of resources are memory, file handles, network connections (sockets), and locks. A program that is part of a long-running system must release every resource it requires except a few that it permanently owns. For example, a program that forgets to close a

file every day will on most operating systems not survive for more than about a month. A program that fails to deallocate 100 bytes every day will waste more than 32K a year – that's enough to crash a small gadget after a few months. The nasty thing about such resource "leaks" is that the program will work perfectly for months before it suddenly ceases to function. If a program will crash, we prefer it to crash as soon as possible so that we can remedy the problem. In particular, we prefer it to crash long before it is given to users.

- *Real-time constraints:* We can classify an embedded system as *hard real time* if a certain response must occur before a deadline. If a response must occur before a deadline most of the time, but we can afford an occasional time overrun, we classify the system as *soft real time*. Examples of soft real time are a controller for a car window and a stereo amplifier. A human will not notice a fraction of a second's delay in the movement of the window, and only a trained listener would be able to hear a millisecond's delay in a change of pitch. An example of hard real time is a fuel injector that has to "squirt" at exactly the right time relative to the movement of the piston. If the timing is off by even a fraction of a millisecond, performance suffers and the engine starts to deteriorate; a major timing problem could completely stop the engine, possibly leading to accident or disaster.
- *Predictability:* This is a key notion in embedded systems code. Obviously, the term has many intuitive meanings, but here – in the context of programming embedded systems – we will use a specialized technical meaning: an operation is *predictable* if it takes the same amount of time to execute every time it is executed on a given computer, and if all such operations take the same amount of time to execute. For example, when **x** and **y** are integers, **x+y** takes the same amount of time to execute every time and **xx+yy** takes the same amount of time when **xx** and **yy** are two other integers. Usually, we can ignore minor variations in execution speed related to machine architecture (e.g., differences caused by caching and pipelining) and simply rely on there being a fixed, constant upper limit on the time needed. Operations that are not predictable (in this sense of the word) can't be used in hard real-time systems and must be used with great care in all real-time systems. A classic example of an unpredictable operation is a linear search of a list (e.g., **find()**) where the number of elements is unknown and not easily bounded. Only if we can reliably predict the number of elements or at least the maximum number of elements does such a search become acceptable in a hard real-time system; that is, to *guarantee* a response within a given fixed time we must be able to – possibly aided by code analysis tools – calculate the time needed for every possible code sequence leading up to the deadline.

- *Concurrency:* An embedded system typically has to respond to events from the external world. This leads to programs where many things happen "at once" because they correspond to real events that really happen at once. A program that simultaneously deals with several actions is called *concurrent* or *parallel*. Unfortunately the fascinating, difficult, and important issue of concurrency is beyond the scope of this book.

25.2.1 Predictability



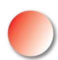
From the point of view of predictability, C++ is pretty good, but it isn't perfect. All facilities in the C++ language (including virtual function calls) are predictable, except

- Free-store allocation using **new** and **delete** (see §25.3)
- Exceptions (§19.5)
- **dynamic_cast** (§A.5.7)

These facilities must be avoided for hard real-time applications. The problems with **new** and **delete** are described in detail in §25.3; those are fundamental. Note that the standard library **string** and the standard containers (**vector**, **map**, etc.) indirectly use the free store, so they are not predictable either. The problem with **dynamic_cast** is a problem with current implementations but is not fundamental.

The problem with exceptions is that when looking at a particular **throw**, the programmer cannot – without looking at large sections of code – know how long it will take to find a matching **catch** or even if there is such a **catch**. In an embedded systems program, there had better be a **catch** because we can't rely on a C++ programmer sitting ready to use the debugger. The problems with exceptions can in principle be dealt with by a tool that for each **throw** tells you exactly which **catch** will be invoked and how long it will take the **throw** to get there, but currently, that's a research problem, so if you need predictability, you'll have to make do with error handling based on return codes and other old-fashioned and tedious, but predictable, techniques.

25.2.2 Ideals



When writing an embedded systems program there is a danger that the quest for performance and reliability will lead the programmer to regress to exclusively using low-level language facilities. That strategy is workable for individual small pieces of code. However, it can easily leave the overall design a mess, make it difficult to be confident about correctness, and increase the time and money needed to build a system.

As ever, our ideal is to work at the highest level of abstraction that is feasible given the constraints on our problem. Don't get reduced to writing glorified

assembler code! As ever, represent your ideas as directly in code as you can (given all constraints). As ever, try hard to write the clearest, cleanest, most maintainable code. Don't optimize until you have to. Performance (in time or space) is often essential for an embedded system, but trying to squeeze performance out of every little piece of code is misguided. Also, for many embedded systems the key is to be correct and fast enough; beyond "fast enough" the system simply idles until another action is needed. Trying to write every few lines of code to be as efficient as possible takes a lot of time, causes a lot of bugs, and often leads to missed opportunities for optimization as algorithms and data structures get hard to understand and hard to change. For example, that "low-level optimization" approach often leads to missed opportunities for memory optimization because almost similar code appears in many places and can't be shared because of incidental differences.

John Bentley – famous for his highly efficient code – offers two "laws of optimization":

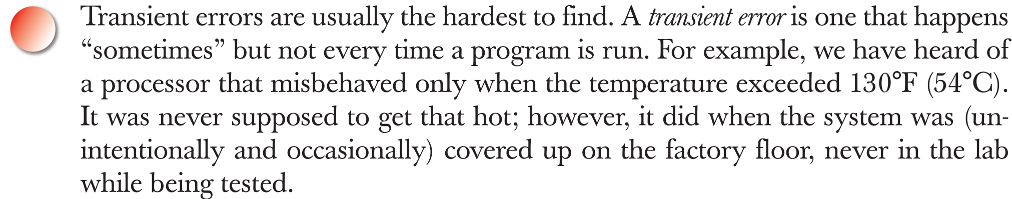
- First law: Don't do it.
- Second law (for experts only): Don't do it yet.

Before optimizing, make sure that you understand the system. Only then can you be confident that it is – or can become – correct and reliable. Focus on algorithms and data structures. Once an early version of the system runs, carefully measure and tune it as needed. Fortunately, pleasant surprises are not uncommon: clean code sometimes runs fast enough and doesn't take up excessive memory space. Don't count on that, though; measure. Unpleasant surprises are not uncommon either.

25.2.3 Living with failure

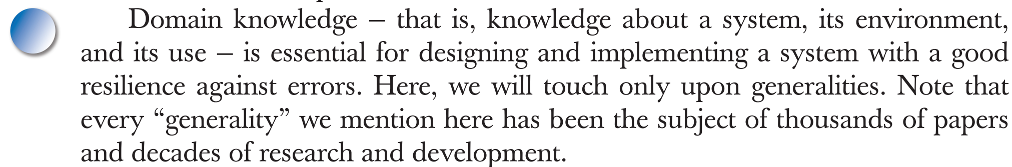
Imagine that we are to design and implement a system that may not fail. By "not fail" let's say that we mean "will run without human intervention for a month." What kind of failures must we protect against? We can exclude dealing with the sun going nova and probably also with the system being trampled by an elephant. However, in general we cannot know what might go wrong. For a specific system, we can and must make assumptions about what kinds of errors are more common than others. Examples:

- Power surges/failure
- Connector vibrating out of its socket
- System hit by falling debris crushing a processor
- Falling system (disk might be destroyed by impact)
- X-rays causing some memory bits to change value in ways impossible according to the language definition



Transient errors are usually the hardest to find. A *transient error* is one that happens "sometimes" but not every time a program is run. For example, we have heard of a processor that misbehaved only when the temperature exceeded 130°F (54°C). It was never supposed to get that hot; however, it did when the system was (unintentionally and occasionally) covered up on the factory floor, never in the lab while being tested.

Errors that occur away from the lab are the hardest to fix. You will have a hard time imagining the design and implementation effort involved in letting the JPL engineers diagnose software and hardware failures on the Mars Rovers (20 minutes away from the lab for a signal traveling at the speed of light) and update the software to fix a problem once understood.



Domain knowledge – that is, knowledge about a system, its environment, and its use – is essential for designing and implementing a system with a good resilience against errors. Here, we will touch only upon generalities. Note that every "generality" we mention here has been the subject of thousands of papers and decades of research and development.

- *Prevent resource leaks:* Don't leak. Be specific about what resources your program uses and be sure you conserve them (perfectly). Any leak will kill your system or subsystem eventually. The most fundamental resources are time and memory. Typically, a program will also use other resources, such as locks, communication channels, and files.
- *Replicate:* If a system critically needs a hardware resource (e.g., a computer, an output device, a wheel) to function, then the designer is faced with a basic choice: should the system contain several copies of the critical resource? We can either accept failure if the hardware breaks or provide a spare and let the software switch to using the spare. For example, the fuel injector controllers for the marine diesel engine are triplicate computers connected by duplicate networks. Note that "the spare" need not be identical to the original (e.g., a space probe may have a primary strong antenna and a weaker backup). Note also that "the spare" can typically be used to boost performance when the system works without a problem.
- *Self-check:* Know when the program (or hardware) is misbehaving. Hardware components (e.g., storage devices) can be very helpful in this respect, monitoring themselves for errors, correcting minor errors, and reporting major failures. Software can check for consistency of its data structures, check invariants (§9.4.3), and rely on internal "sanity checks" (assertions). Unfortunately, self-checking can itself be unreliable, and care must be taken that reporting an error doesn't itself cause an error – it is really hard to completely check error checking.
- *Have a quick way out of misbehaving code:* Make systems modular. Base error handling on modules: each module has a specific task to do. If a module

Stroustrup_book.indb 934 5/8/15 10:31 AM

decides it can't do its task, it can report that to some other module. Keep the error handling within a module simple (so that it is more likely to be correct and efficient), and have some other module responsible for serious errors. A good reliable system is modular and multi-level. At each level, serious errors are reported to a module at the next level – in the end, maybe to a person. A module that has been notified of a serious error (one that another module couldn't handle itself) can then take appropriate action – maybe involving a restart of the module that detected the error or running with a less sophisticated (but more robust) "backup" module. Defining exactly what "a module" is for a given system is part of the overall system design, but you can think of it as a class, a library, a program, or all the programs on a computer.

- *Monitor subsystems* in case they can't or don't notice a problem themselves. In a multi-level system higher levels can monitor lower levels. Many systems that really aren't allowed to fail (e.g., the marine engines or space station controllers) have three copies of critical subsystems. This triplication is not done just to have two spares, but also so that disagreements about which subsystem is misbehaving can be settled by 2-to-1 votes. Triplication is especially useful where a multi-level organization is difficult (i.e., at the highest level of a system or subsystem that may not fail).

We can design as much as we like and be as careful with the implementation as we know how to, but the system will still misbehave. Before delivering a system to users, it must be systematically and thoroughly tested; see Chapter 26.

25.3 Memory management

The two most fundamental resources in a computer are time (to execute instructions) and space (memory to hold data and code). In C++, there are three ways to allocate memory to hold data (§17.4, §A.4.2):

- *Static memory:* allocated by the linker and persisting as long as the program runs
- *Stack (automatic) memory:* allocated when we call a function and freed when we return from the function
- *Dynamic (heap) memory:* allocated by **new** and freed for possible reuse by **delete**

Let's consider these from the perspective of embedded systems programming. In particular, we will consider memory management from the perspective of tasks where predictability (§25.2.1) is considered essential, such as hard real-time programming and safety-critical programming.

Static memory poses no special problem in embedded systems programming: all is taken care of before the program starts to run and long before a system is deployed.

Stack memory can be a problem because it is possible to use too much of it, but this is not hard to take care of. The designers of a system must determine that for no execution of the program will the stack grow over an acceptable limit. This usually means that the maximum nesting of function calls must be limited; that is, we must be able to demonstrate that a chain of calls (e.g., **f1** calls **f2** calls . . . calls **fn**) will never be too long. In some systems, that has caused a ban on recursive calls. Such a ban can be reasonable for some systems and for some recursive functions, but it is not fundamental. For example, I *know* that **factorial(10)** will call **factorial** at most ten times. However, an embedded systems programmer might very well prefer an iterative implementation of **factorial** (§15.5) to avoid any doubt or accident.

Dynamic memory allocation is usually banned or severely restricted; that is, **new** is either banned or its use restricted to a startup period, and **delete** is banned. The basic reasons are

- *Predictability:* Free-store allocation is not predictable; that is, it is not guaranteed to be a constant time operation. Usually, it is not: in many implementations of **new**, the time needed to allocate a new object can increase dramatically after many objects have been allocated and deallocated.

- *Fragmentation:* The free store may fragment; that is, after allocating and deallocating objects the remaining unused memory may be "fragmented" into a lot of little "holes" of unused space that are useless because each hole is too small to hold an object of the kind used by the application. Thus, the size of the useful free store can be far less than the size of the initial free store minus the size of the allocated objects.

The next section explains how this unacceptable state of affairs can arise. The bottom line is that we must avoid programming techniques that use both **new** and **delete** for hard real-time or safety-critical systems. The following sections explain how we can systematically avoid problems with the free store using stacks and pools.

25.3.1 Free-store problems

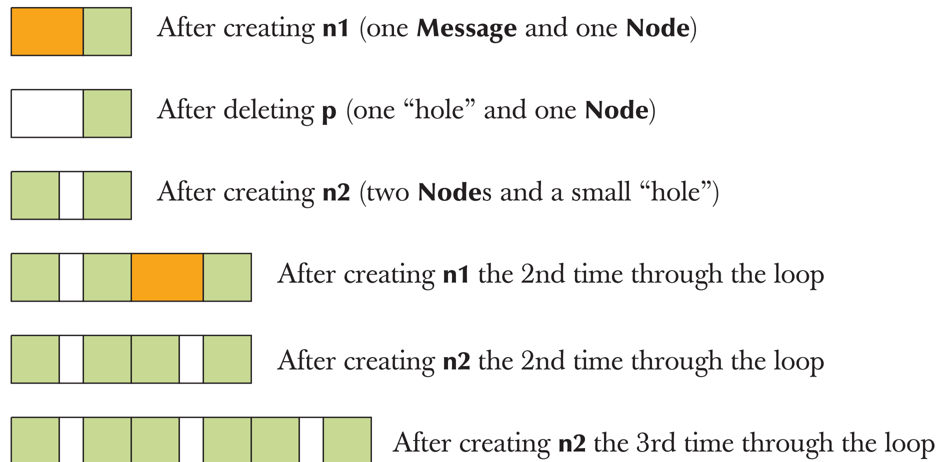
What's the problem with **new**? Well, really it's a problem with **new** and **delete** used together. Consider the result of this sequence of allocations and deallocations:

```
Message* get_input(Device&);          // make a Message on the free store

while(/* . . . */) {
      Message* p = get_input(dev);
      // . . .
```

```
        Node* n1 = new Node(arg1,arg2);
        // . . .
        delete p;
        Node* n2 = new Node (arg3,arg4);
        // . . .
}
```

Each time around the loop we create two **Node**s, and in the process of doing so we create a **Message** and delete it again. Such code would not be unusual as part of building a data structure based on input from some "device." Looking at this code, we might expect to "consume" **2*sizeof(Node)** bytes of memory (plus free-store overhead) each time around the loop. Unfortunately, it is not guaranteed that the "consumption" of memory is restricted to the expected and desired **2*sizeof(Node)** bytes. In fact, it is unlikely to be the case.

Assume a simple (though not unrealistic) memory manager. Assume also that a **Message** is a bit larger than a **Node**. We can visualize the use of free space like this, using orange for the **Message**, green for the **Node**s, and plain white for "a hole" (that is, "unused space"):

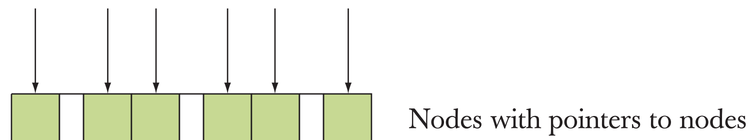


So, we are leaving behind some unused space ("a hole") on the free store each time we execute the loop. That may be just a few bytes, but if we can't use those holes it will be as bad as a memory leak – and even a small leak will eventually kill a long-running program. Having the free space in our memory scattered in many "holes" too small for allocating new objects is called *memory fragmentation*. Basically, the free-store manager will eventually use up all "holes" that are big enough to hold the kind of objects that the program uses, leaving only holes that are too small to be useful. This is a serious problem for essentially all

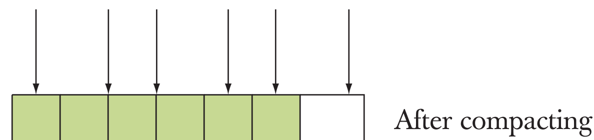
long-running programs that use **new** and **delete** extensively; it is not uncommon to find unusable fragments taking up most of the memory. That usually dramatically increases the time needed to execute **new** as it has to search through lots of objects and fragments for a suitably sized chunk of memory. Clearly this is not the kind of behavior we can accept for an embedded system. This can also be a serious problem in naively designed non-embedded systems.

Why can't "the language" or "the system" deal with this? Alternatively, can't we just write our program to not create such "holes"? Let's first examine the most obvious solution to having all those little useless "holes" in our memory: let's move the **Node**s so that all the free space gets compacted into one contiguous area that we can use to allocate more objects.

Unfortunately, "the system" can't do that. The reason is that C++ code refers directly to objects in memory. For example, the pointers **n1** and **n2** contain real memory addresses. If we moved the objects pointed to, those addresses would no longer point to the right objects. Assume that we (somewhere) keep pointers to the nodes we created. We could represent the relevant part of our data structure like this:


Nodes with pointers to nodes

Now we compact memory by moving an object so that all the unused memory is in one place:


After compacting

Unfortunately, we now have made a mess of those pointers by moving the objects they pointed to without updating the pointers. Why don't we just update the pointers when we move the objects? We could write a program to do that, but only if we knew the details of the data structure. In general, "the system" (the C++ run-time support system) has no idea where the pointers are; that is, given an object, the question "Which pointers in the program point to this object right now?" has no good answer. Even if that problem could be easily solved, this approach (known as *compacting garbage collection*) is not always the right one. For example, to work well, it typically requires more than twice the memory that the program ever needs to be able to keep track of pointers and to move objects

around in. That extra memory may not be available on an embedded system. In addition, an efficient compacting garbage collector is hard to make predictable.

We could of course answer that "Where are the pointers?" question for our own data structures and compact those. That would work, but a simpler approach is to avoid fragmentation in the first place. In the example here, we could simply have allocated both **Node**s before allocating the message:

```
while( . . . ) {
        Node* n1 = new Node;
        Node* n2 = new Node;
        Message* p = get_input(dev);
        // . . . store information in nodes . . .
        delete p;
        // . . .
}
```

However, rearranging code to avoid fragmentation isn't easy in general. Doing so reliably is at best very difficult and often incompatible with other rules for good code. Consequently, we prefer to restrict the use of the free store to ways that don't cause fragmentation in the first place. Often, preventing a problem is better than solving it.

TRY THIS

Complete the program above and print out the addresses and sizes of the objects created to see if and how "holes" appear on your machine. If you have time, you might draw memory layouts like the ones above to better visualize what's going on.

25.3.2 Alternatives to the general free store

So, we mustn't cause fragmentation. What do we do then? The first simple observation is that **new** cannot by itself cause fragmentation; it needs **delete** to create the holes. So we start by banning **delete**. That implies that once an object is allocated, it will stay part of the program forever.

In the absence of **delete**, is **new** predictable; that is, do all **new** operations take the same amount of time? Yes, in all common implementations, but it is not actually guaranteed by the standard. Usually, an embedded system has a startup sequence of code that establishes the system as "ready to run" after initial power-up or restart. During that period, we can allocate memory any way we like

up to an allowed maximum. We could decide to use **new** during startup. Alternatively (or additionally) we could set aside global (static) memory for future use. For reasons of program structure, global data is often best avoided, but it can be sensible to use that language mechanism to pre-allocate memory. The exact rules for this should be laid down in a coding standard for a system (see §25.6).

There are two data structures that are particularly useful for predictable memory allocation:

- *Stacks:* A stack is a data structure where you can allocate an arbitrary amount of memory (up to a given maximum size) and deallocate the last allocation (only); that is, a stack can grow and shrink only at the top. There can be no fragmentation, because there can be no "hole" between two allocations.
- *Pools:* A pool is a collection of objects of the same size. We can allocate and deallocate objects as long as we don't allocate more objects than the pool can hold. There can be no fragmentation because all objects are of the same size.

For both stacks and pools, both allocation and deallocation are predictable and fast.

So, for a hard real-time or critical system we can define stacks and pools as needed. Better yet, we ought to be able to use stacks and pools as specified, implemented, and tested by someone else (as long as the specification meets our needs).

Note that the C++ standard containers (**vector**, **map**, etc.) and the standard **string** are not to be used because they indirectly use **new**. You can build (buy or borrow) "standard-like" containers to be predictable, but the default ones that come with your implementation are not constrained for embedded systems use.

Note that embedded systems typically have very stringent reliability requirements, so whatever solution we choose, we must make sure not to compromise our programming style by regressing into using lots of low-level facilities directly. Code that is full of pointers, explicit conversions, etc. is unreasonably hard to guarantee as correct.

25.3.3 Pool example

A *pool* is a data structure from which we can allocate objects of a given type and later deallocate (free) such objects. A pool contains a maximum number of objects; that number is specified when the pool is created. Using green for "allocated object" and blue for "space ready for allocation as an object," we can visualize a pool like this:

A **Pool** can be defined like this:

```
template<typename T, int N>
class Pool {                    // Pool of N objects of type T
public:
        Pool();                 // make pool of N Ts
        T* get();               // get a T from the pool; return 0 if no free Ts
        void free(T*);          // return a T given out by get() to the pool
        int available() const;  // number of free Ts
private:
        // space for T[N] and data to keep track of which Ts are allocated
        // and which are not (e.g., a list of free objects)
};
```

Each **Pool** object has a type of elements and a maximum number of objects. We can use a **Pool** like this:

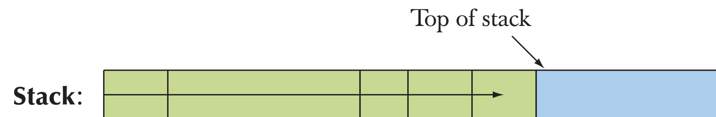
```
Pool<Small_buffer,10> sb_pool;
Pool<Status_indicator,200> indicator_pool;

Small_buffer* p = sb_pool.get();
// . . .
sb_pool.free(p);
```

It is the job of the programmer to make sure that a pool is never exhausted. The exact meaning of "make sure" depends on the application. For some systems, the programmer must write the code such that **get()** is never called unless there is an object to allocate. On other systems, a programmer can test the result of **get()** and take some remedial action if that result is **0**. A characteristic example of the latter is a telephone system engineered to handle at most 100,000 calls at a time. For each call, some resource, such as a dial buffer, is allocated. If the system runs out of dial buffers (e.g., **dial_buffer_pool.get()** returns **0**), the system refuses to set up new connections (and may "kill" a few existing calls to create capacity). The would-be caller can try again later.

Naturally, our **Pool** template is only one variation of the general idea of a pool. For example, where the restraints on memory allocation are less Draconian, we can define pools where the number of elements is specified in the constructor or even pools where the number of elements can be changed later if we need more objects than initially specified.

25.3.4 Stack example

A *stack* is a data structure from which we can allocate chunks of memory and deallocate the last allocated chunk. Using green for "allocated memory" and blue for "space ready for allocation," we can visualize a stack like this:



As indicated, this stack "grows" toward the right.

We could define a stack of objects, just as we defined a pool of objects:

```
template<typename T, int N>
class Stack {                    // stack of N objects of type T
        // . . .
};
```

However, most systems have a need for allocation of objects of varying sizes. A stack can do that whereas a pool cannot, so we'll show how to define a stack from which we allocate "raw" memory of varying sizes rather than fixed-size objects:

```
template<int N>
class Stack {                    // stack of N bytes
public:
        Stack();                 // make an N-byte stack
        void* get(int n);        // allocate n bytes from the stack;
                                 // return 0 if no free space
        void free();             // return the last value returned by get() to the stack
        int available() const;   // number of available bytes
private:
        // space for char[N] and data to keep track of what is allocated
        // and what is not (e.g., a top-of-stack pointer)
};
```

Since **get()** returns a **void*** pointing to the required number of bytes, it is our job to convert that memory to the kinds of objects we want. We can use such a stack like this:

```
Stack<50*1024> my_free_store;     // 50K worth of storage to be used as a stack

void* pv1 = my_free_store.get(1024);
int* buffer = static_cast<int*>(pv1);
```

```
void* pv2 = my_free_store.get(sizeof(Connection));
Connection* pconn = new(pv2) Connection(incoming,outgoing,buffer);
```

The use of **static_cast** is described in §17.8. The **new(pv2)** construct is a "placement **new**." It means "Construct an object in the space pointed to by **pv2**." It doesn't allocate anything. The assumption here is that the type **Connection** has a constructor that will accept the argument list **(incoming,outgoing,buffer)**. If that's not the case, the program won't compile.

Naturally, our **Stack** template is only one variation of the general idea of a stack. For example, where the restraints on memory allocation are less Draconian, we can define stacks where the number of bytes available for allocation is specified in the constructor.

25.4 Addresses, pointers, and arrays

Predictability is a need of some embedded systems; reliability is a concern of all. This leads to attempts to avoid language features and programming techniques that have proved error-prone (in the context of embedded systems programming, if not necessarily everywhere). Careless use of pointers is the main suspect here. Two problem areas stand out:

- Explicit (unchecked and unsafe) conversions
- Passing pointers to array elements

The former problem can typically be handled simply by severely restricting the use of explicit type conversions (casts). The pointer/array problems are more subtle, require understanding, and are best dealt with using (simple) classes or library facilities (such as **array**, §20.9). Consequently, this section focuses on how to address the latter problems.

25.4.1 Unchecked conversions

Physical resources (e.g., control registers for external devices) and their most basic software controls typically exist at specific addresses in a low-level system. We have to enter such addresses into our programs and give a type to such data. For example:

```
Device_driver* p = reinterpret_cast<Device_driver*>(0xffb8);
```

See also §17.8. This is the kind of programming you do with a manual or online documentation open. The correspondence between a hardware resource – the address of the resource's register(s) (expressed as an integer, often a hexadecimal integer) – and pointers to the software that manipulates the hardware resource

is brittle. You have to get it right without much help from the compiler (because it is not a programming language issue). Usually, a simple (nasty, completely unchecked) **reinterpret_cast** from an **int** to a pointer type is the essential link in the chain of connections from an application to its nontrivial hardware resources.

Where explicit conversions (**reinterpret_cast**, **static_cast**, etc.; see §A.5.7) are not essential, avoid them. Such conversions (casts) are necessary far less frequently than is typically assumed by programmers whose primary experience is with C and C-style C++.

25.4.2 A problem: dysfunctional interfaces

As mentioned (§18.6.1), an array is often passed to a function as a pointer to an element (often, a pointer to the first element). Thereby, they "lose" their size, so that the receiving function cannot directly tell how many elements are pointed to, if any. This is a cause of many subtle and hard-to-fix bugs. Here, we examine examples of those array/pointer problems and present an alternative. We start with an example of a very poor (but unfortunately not rare) interface and proceed to improve it. Consider:

```
void poor(Shape* p, int sz)              // poor interface design
{
      for (int i = 0; i<sz; ++i) p[i].draw();
}

void f(Shape* q, vector<Circle>& s0)     // very bad code
{
      Polygon s1[10];
      Shape s2[10];
      // initialize
      Shape* p1 = new Rectangle{Point{0,0},Point{10,20}};
      poor(&s0[0],s0.size());            // #1 (pass the array from the vector)
      poor(s1,10);                       // #2
      poor(s2,20);                       // #3
      poor(p1,1);                        // #4
      delete p1;
      p1 = 0;
      poor(p1,1);                        // #5
      poor(q,max);                       // #6
}
```

The function **poor()** is an example of poor interface design: it provides an interface that provides the caller ample opportunity for mistakes but offers the implementer essentially no opportunity to defend against such mistakes.

TRY THIS

Before reading further, try to see how many errors you can find in **f()**. Specifically, which of the calls of **poor()** could cause the program to crash?

At first glance, the calls look fine, but this is the kind of code that costs a programmer long nights of debugging and gives a quality engineer nightmares.

1. Passing the wrong element type, e.g., **poor(&s0[0],s0.size())**. Also, **s0** might be empty, in which case **&s0[0]** is wrong.
2. Use of a "magic constant" (here, correct): **poor(s1,10)**. Also, wrong element type.
3. Use of a "magic constant" (here, incorrect): **poor(s2,20)**.
4. Correct (easily verified): first call **poor(p1,1)**.
5. Passing a null pointer: second call **poor(p1,1)**.
6. May be correct: **poor(q,max)**. We can't be sure from looking at this code fragment. To see if **q** points to an array with at least **max** elements, we have to find the definitions of **q** and **max** and determine their values at our point of use.

In each case, the errors are simple. We are not dealing with some subtle algorithmic or data structure problem. The problem is that **poor()**'s interface, involving an array passed as a pointer, opens the possibility of a collection of problems. You may appreciate how the problems were obscured by our use of "technical" unhelpful names, such as **p1** and **s0**. However, mnemonic, but misleading, names can make such problems even harder to spot.

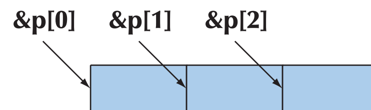
In theory, a compiler could catch a few of these errors (such as the second call of **poor(p1,1)** where **p1==0**), but realistically we are saved from disaster for this particular example only because the compiler catches the attempt to define objects of the abstract class **Shape**. However, that is unrelated to **poor()**'s interface problems, so we should not take too much comfort from that. In the following, we use a variant of **Shape** that is not abstract so as not to get distracted from the interface problems.

How come the **poor(&s0[0],s0.size())** call is an error? The **&s0[0]** refers to the first element of an array of **Circle**s; it is a **Circle***. We expect a **Shape*** and we pass a pointer to an object of a class derived from **Shape** (here, a **Circle***). That's obviously acceptable: we need that conversion so that we can do object-oriented programming, accessing objects of a variety of types through their common interface

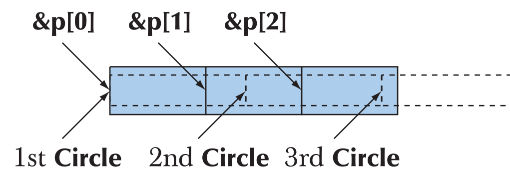
(here, **Shape**) (§14.2). However, **poor()** doesn't just use that **Shape*** as a pointer; it uses it as an array, subscripting its way through that array:

```
for (int i = 0; i<sz; ++i) p[i].draw();
```

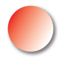
That is, it looks at the objects starting at memory locations **&p[0]**, **&p[1]**, **&p[2]**, etc.:



In terms of memory addresses, these pointers are **sizeof(Shape)** apart (§17.3.1). Unfortunately for **poor()**'s caller, **sizeof(Circle)** is larger than **sizeof(Shape)**, so that the memory layout can be visualized like this:



That is, **poor()** is calling **draw()** with a pointer into the middle of the **Circle**s! This is likely to lead to immediate disaster (crash).



The call **poor(s1,10)** is sneakier. It relies on a "magic constant" so it is immediately suspect as a maintenance hazard, but there is a deeper problem. The only reason the use of an array of **Polygon**s doesn't immediately suffer the problem we saw for **Circle**s is that a **Polygon** didn't add data members to its base class **Shape** (whereas **Circle** did; see §13.8 and §13.12); that is, **sizeof(Shape)==sizeof(Polygon)** and – more generally – a **Polygon** has the same memory layout as a **Shape**. In other words, we were "just lucky"; a slight change in the definition of **Polygon** will cause a crash. So **poor(s1,10)** works, but it is a bug waiting to happen. This is emphatically not quality code.

What we see here is the implementation reason for the general language rule that "a **D** is a **B**" does not imply "a **Container<D>** is a **Container<B>**" (§19.3.3). For example:

```
class Circle : public Shape { /* . . . */ };

void fv(vector<Shape>&);
void f(Shape &);
```

```
void g(vector<Circle>& vd, Circle & d)
{
	f(d);	// OK: implicit conversion from Circle to Shape
	fv(vd);	// error: no conversion from vector<Circle> to vector<Shape>
}
```

OK, so the use of **poor()** is very bad code, but can such code be considered embedded systems code; that is, should this kind of problem concern us in areas where safety or performance matters? Can we dismiss it as a hazard for programmers of non-critical systems and just tell them, "Don't do that"? Well, many modern embedded systems rely critically on a GUI, which is almost always organized in the object-oriented manner of our example. Examples include the iPod user interface, the interfaces of some cell phones, and operator's displays on "gadgets" up to and including airplanes. Another example is that controllers of similar gadgets (such as a variety of electric motors) can constitute a classic class hierarchy. In other words, this kind of code – and in particular, this kind of function declaration – is exactly the kind of code we should worry about. We need a safer way of passing information about collections of data without causing other significant problems.

So, we don't want to pass a built-in array to a function as a pointer plus a size. What do we do instead? The simplest solution is to pass a reference to a container, such as a **vector**. The problems we saw for

```
void poor(Shape* p, int sz);
```

simply cannot occur for

```
void general(vector<Shape>&);
```

If you are programming where **std::vector** (or the equivalent) is acceptable, simply use **vector** (or the equivalent) consistently in interfaces; never pass a built-in array as a pointer plus a size.

If you can't restrict yourself to **vector** or equivalents, you enter a territory that is more difficult and the solutions there involve techniques and language features that are not simple – even though the use of the class (**Array_ref**) we provide is straightforward.

25.4.3 A solution: an interface class

Unfortunately, we cannot use **std::vector** in many embedded systems because it relies on the free store. We can solve that problem either by having a special implementation of **vector** or (more easily) by using a container that behaves like

a **vector** but doesn't do memory management. Before outlining such an interface class, let's consider what we want from it:

- It is a reference to objects in memory (it does not own objects, allocate objects, delete objects, etc.).
- It "knows" its size (so that it is potentially range checked).
- It "knows" the exact type of its elements (so that it cannot be the source of type errors).
- It is as cheap to pass (copy) as a (pointer,count) pair.
- It does *not* implicitly convert to a pointer.
- It is easy to express a subrange of the range of elements described by an interface object.
- It is as easy to use as built-in arrays.

We will only be able to approximate "as easy to use as built-in arrays." We don't want it to be so easy to use that errors start to become likely.

Here is one such class:

```
template<typename T>
class Array_ref {
public:
        Array_ref(T* pp, int s) :p{pp}, sz{s} { }

        T& operator[ ](int n) { return p[n]; }
        const T& operator[ ](int n) const { return p[n]; }

        bool assign(Array_ref a)
        {
                if (a.sz!=sz) return false;
                for (int i=0; i<sz; ++i) { p[i]=a.p[i]; }
                return true;
        }

        void reset(Array_ref a) { reset(a.p,a.sz); }
        void reset(T* pp, int s) { p=pp; sz=s; }

        int size() const { return sz; }

        // default copy operations:
        // Array_ref doesn't own any resources
        // Array_ref has reference semantics
```

```
private:
        T* p;
        int sz;
};
```

Array_ref is close to minimal:

- No **push_back()** (that would require the free store) and no **at()** (that would require exceptions).
- **Array_ref** is a form of reference, so copying simply copies (**p**,**sz**).
- By initializing with different arrays, we can have **Array_ref**s that are of the same type but have different sizes.
- By updating (**p**,**size**) using **reset()**, we can change the size of an existing **Array_ref** (many algorithms require specification of subranges).
- No iterator interface (but that could be easily added if we needed it). In fact, an **Array_ref** is in concept very close to a range described by two iterators.

An **Array_ref** does not own its elements; it does no memory management; it is simply a mechanism for accessing and passing a sequence of elements. In that, it differs from the standard library **array** (§20.9).

To ease the creation of **Array_ref**s, we supply a few useful helper functions:

```
template<typename T> Array_ref<T> make_ref(T* pp, int s)
{
        return (pp) ? Array_ref<T>{pp,s} : Array_ref<T>{nullptr,0};
}
```

If we initialize an **Array_ref** with a pointer, we have to explicitly supply a size. That's an obvious weakness because it provides us with an opportunity to give the wrong size. It also gives us an opportunity to use a pointer that is a result of an implicit conversion of an array of a derived class to a pointer to a base class, such as **Polygon[10]** to **Shape*** (the original horrible problem from §25.4.2), but sometimes we simply have to trust the programmer.

We decided to be careful about null pointers (because they are a common source of problems), and we took a similar precaution for empty **vector**s:

```
template<typename T> Array_ref<T> make_ref(vector<T>& v)
{
        return (v.size()) ? Array_ref<T>{&v[0],v.size()} : Array_ref<T>{nullptr,0};
}
```

The idea is to pass the **vector**'s array of elements. We concern ourselves with **vector** here even though it is often not suitable in the kind of system where **Array_ref** can be useful. The reason is that it shares key properties with containers that can be used there (e.g., pool-based containers; see §25.3.3).

Finally, we deal with built-in arrays where the compiler knows the size:

```
template <typename T, int s> Array_ref<T> make_ref(T (&pp)[s])
{
	return Array_ref<T>{pp,s};
}
```

The curious **T(&pp)[s]** notation declares the argument **pp** to be a reference to an array of **s** elements of type **T**. That allows us to initialize an **Array_ref** with an array, remembering its size. We can't declare an empty array, so we don't have to test for zero elements:

```
Polygon ar[0];		// error: no elements
```

Given **Array_ref**, we can try to rewrite our example:

```
void better(Array_ref<Shape> a)
{
	for (int i = 0; i<a.size(); ++i) a[i].draw();
}

void f(Shape* q, vector<Circle>& s0)
{
	Polygon s1[10];
	Shape s2[20];
	// initialize
	Shape* p1 = new Rectangle{Point{0,0},Point{10,20}};
	better(make_ref(s0));		// error: Array_ref<Shape> required
	better(make_ref(s1));		// error: Array_ref<Shape> required
	better(make_ref(s2));		// OK (no conversion required)
	better(make_ref(p1,1));		// OK: one element
	delete p1;
	p1 = 0;
	better(make_ref(p1,1));		// OK: no elements
	better(make_ref(q,max));	// OK (if max is OK)
}
```

We see improvements:

- The code is simpler. The programmer rarely has to think about sizes, but when necessary they are in a specific place (the creation of an **Array_ref**), rather than scattered throughout the code.
- The type problem with the **Circle[]**-to-**Shape[]** and **Polygon[]**-to-**Shape[]** conversions is caught.
- The problems with the wrong number of elements for **s1** and **s2** are implicitly dealt with.
- The potential problem with **max** (and other element counts for pointers) becomes more visible – it's the only place we have to be explicit about size.
- We deal implicitly and systematically with null pointers and empty **vector**s.

25.4.4 Inheritance and containers

But what if we wanted to treat a collection of **Circle**s as a collection of **Shape**s, that is, if we really wanted **better()** (which is a variant of our old friend **draw_all()**; see §19.3.2, §22.1.3) to handle polymorphism? Well, basically, we can't. In §19.3.3 and §25.4.2, we saw that the type system has very good reasons for refusing to accept a **vector<Circle>** as a **vector<Shape>**. For the same reason, it refuses to accept an **Array_ref<Circle>** as an **Array_ref<Shape>**. If you have a problem remembering why, it might be a good idea to reread §19.3.3, because the point is pretty fundamental even though it can be inconvenient.

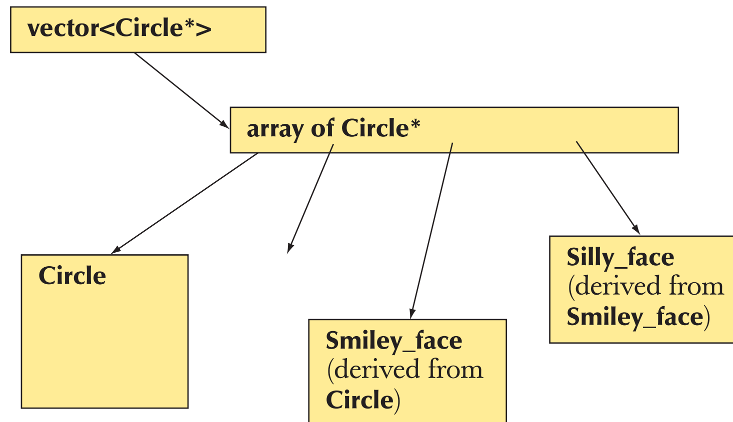
Furthermore, to preserve run-time polymorphic behavior, we have to manipulate our polymorphic objects through pointers (or references): the dot in **a[i].draw()** in **better()** was a giveaway. We should have expected problems with polymorphism the second we saw that dot rather than an arrow (**–>**).

So what can we do? First we *must* use pointers (or references) rather than objects directly, so we'll try to use **Array_ref<Circle*>**, **Array_ref<Shape*>**, etc. rather than **Array_ref<Circle>**, **Array_ref<Shape>**, etc.

However, we still cannot convert an **Array_ref<Circle*>** to an **Array_ref<Shape*>** because we might then proceed to put elements into the **Array_ref<Shape*>** that are not **Circle***s. But there is a loophole:

- Here, we don't want to modify our **Array_ref<Shape*>**; we just want to draw the **Shape**s! This is an interesting and useful special case: our argument against the **Array_ref<Circle*>**-to-**Array_ref<Shape*>** conversion doesn't apply to a case where we don't modify the **Array_ref<Shape*>**.
- All arrays of pointers have the same layout (independently of what kinds of objects they point to), so we don't get into the layout problem from §25.4.2.

That is, there would be nothing wrong with treating an **Array_ref<Circle*>** as an *immutable* **Array_ref<Shape*>**. So, we "just" have to find a way to treat an **Array_ref<Circle*>** as an immutable **Array_ref<Shape*>**. Consider:



There is no logical problem treating that array of **Circle*** as an immutable array of **Shape*** (from an **Array_ref**).

We seem to have strayed into expert territory. In fact, this problem is genuinely tricky and is unsolvable with the tools supplied so far. However, let's see what it takes to produce a close-to-perfect alternative to our dysfunctional – but all too popular – interface style (pointer plus element count; see §25.4.2). Please remember: Don't go into "expert territory" just to prove how clever you are. Most often, it is a better strategy to find a library where some experts have done the design, implementation, and testing for you.

First, we rework **better()** to something that uses pointers and guarantees that we don't "mess with" the argument container:

```
void better2(const Array_ref<Shape*const> a)
{
	for (int i = 0; i<a.size(); ++i)
		if (a[i])
			a[i]–>draw();
}
```

We are now dealing with pointers, so we should check for null pointers. To make sure that **better2()** doesn't modify our arrays and vectors in unsafe ways through **Array_ref**, we added a couple of **const**s. The first **const** ensures that we do not apply modifying (mutating) operations, such as **assign()** and **reset()**, on our

Array_ref. The second **const** is placed after the * to indicate that we want a constant pointer (rather than a pointer to constants); that is, we don't want to modify the element pointers even if we have operations available for that.

Next, we have to solve the central problem: how do we express the idea that **Array_ref<Circle*>** can be converted

- To something like **Array_ref<Shape*>** (that we can use in **better2()**)
- But only to an immutable version of **Array_ref<Shape*>**

We can do that by adding a conversion operator to **Array_ref**:

```
template<typename T>
class Array_ref {
public:
        // as before

        template<typename Q>
        operator const Array_ref<const Q>()
        {
                // check implicit conversion of elements:
                static_cast<Q>(*static_cast<T*>(nullptr));      // check element
                                                                // conversion
                return Array_ref<const Q>{reinterpret_cast<Q*>(p),sz}; // convert
                                                                       // Array_ref
        }

        // as before
};
```

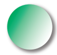
This is headache-inducing, but basically:

- The operator casts to **Array_ref<const Q>** for every type **Q** provided we can cast an element of **Array_ref<T>** to an element of **Array_ref<Q>** (we don't use the result of that cast; we just check that we can cast the element types).
- We construct a new **Array_ref<const Q>** by using brute force (**reinterpret_cast**) to get a pointer to the desired element type. Brute-force solutions often come at a cost; in this case, never use an **Array_ref** conversion from a class using multiple inheritance (§A.12.4).
- Note that **const** in **Array_ref<const Q>**: that's what ensures that we cannot copy an **Array_ref<const Q>** into a plain old mutable **Array_ref<Q>**.

We did warn you that this was "expert territory" and "headache-inducing." However, this version of **Array_ref** is easy to use (it's only the definition/implementation that is tricky):

```
void f(Shape* q, vector<Circle*>& s0)
{
        Polygon* s1[10];
        Shape* s2[20];
        // initialize
        Shape* p1 = new Rectangle{Point{0,0},10};
        better2(make_ref(s0));          // OK: converts to Array_ref<Shape*const>
        better2(make_ref(s1));          // OK: converts to Array_ref<Shape*const>
        better2(make_ref(s2));          // OK (no conversion needed)
        better2(make_ref(p1,1));        // error
        better2(make_ref(q,max));       // error
}
```

The attempts to use pointers result in errors because they are **Shape***s whereas **better2()** expects an **Array_ref<Shape*>**; that is, **better2()** expects something that holds pointers rather than a pointer. If we want to pass pointers to **better2()**, we have to put them into a container (e.g., a built-in array or a **vector**) and pass that. For an individual pointer, we could use the awkward **make_ref(&p1,1)**. However, there is no solution for arrays (with more than one element) that doesn't involve creating a container of pointers to objects.



In conclusion, we can create simple, safe, easy-to-use, and efficient interfaces to compensate for the weaknesses of arrays. That was the major aim of this section. "Every problem is solved by another indirection" (quote by David Wheeler) has been proposed as "the first law of computer science." That was the way we solved this interface problem.

25.5 Bits, bytes, and words

We have talked about hardware memory concepts, such as bits, bytes, and words, before, but in general programming those are not the ones we think much about. Instead we think in terms of objects of specific types, such as **double**, **string**, **Matrix**, and **Simple_window**. Here, we will look at a level of programming where we have to be more aware of the realities of the underlying memory.

If you are uncertain about your knowledge of binary and hexadecimal representations of integers, this may be a good time to review §A.2.1.1.

25.5.1 Bits and bit operations

Think of a byte as a sequence of 8 bits:

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

Note the convention of numbering bits in a byte from the right (the least significant bit) to the left (the most significant bit). Now think of a word as a sequence of 4 bytes:

3:	2:	1:	0:
0xff	0x10	0xde	0xad

Again, we number right to left, that is, least significant byte to most significant byte. These pictures oversimplify what is found in the real world: there have been computers where a byte was 9 bits (but we haven't seen one for a decade), and machines where a word is 2 bytes are not rare. However, as long as you remember to check your system's manual before taking advantage of "8 bits" and "4 bytes," you should be fine.

In code meant to be portable, use **<limits>** (§24.2.1) to make sure your assumptions about sizes are correct. It is possible to place assertions in the code for the compiler to check:

```
static_assert(4<=sizeof(int),"ints are too small");
static_assert(!numeric_limits<char>::is_signed,"char is signed");
```

The first argument of a **static_assert** is a constant expression assumed to be true. If it is not true, that is, the assertion failed, the compiler writes the second argument, a string, as part of an error message.

How do we represent a set of bits in C++? The answer depends on how many bits we need and what kinds of operations we want to be convenient and efficient. We can use the integer types as sets of bits:

- **bool** – 1 bit, but takes up a whole byte of space
- **char** – 8 bits
- **short** – 16 bits
- **int** – typically 32 bits, but many embedded systems have 16-bit **int**s
- **long int** – 32 bits or 64 bits (but at least as many bits as **int**)
- **long long int** – 32 bits or 64 bits (but at least as many bits as **long**)

The sizes quoted are typical, but different implementations may have different sizes, so if you need to know, test. In addition, the standard library provides ways of dealing with bits:

- **std::vector<bool>** – when we need more than 8***sizeof(long)** bits
- **std::bitset** – when we need more than 8***sizeof(long)** bits
- **std::set** – an unordered collection of named bits (see §21.6.5)
- A file: lots of bits (see §25.5.6)

Furthermore, we can use two language features to represent bits:

- Enumerations (**enum**s); see §9.5
- Bitfields; see §25.5.5

This variety of ways to represent "bits" reflects the fact that ultimately everything in computer memory is a set of bits, so people have felt the urge to provide a variety of ways of looking at bits, naming bits, and doing operations on bits. Note that the built-in facilities deal with a set of a fixed number of bits (e.g., 8, 16, 32, and 64) so that the computer can do logical operations on them at optimal speed using operations provided directly by hardware. In contrast, the standard library facilities provide an arbitrary number of bits. This may limit performance, but don't prejudge efficiency issues: the library facilities can be – and often are – optimized to run well if you pick a number of bits that maps well to the underlying hardware.

Let's first look at the integers. For these, C++ basically provides the bitwise logical operations that the hardware directly implements. These operations apply to each bit of their operands:

Bitwise operations		
|	or	Bit **n** of **x|y** is 1 if bit **n** of **x** or bit **n** of **y** is 1.
&	and	Bit **n** of **x&y** is 1 if bit **n** of **x** and bit **n** of **y** is 1.
^	exclusive or	Bit **n** of **x^y** is 1 if bit **n** of **x** or bit **n** of **y** is 1 but not if both are 1.
<<	left shift	Bit **n** of **x<<s** is bit **n+s** of **x**.
>>	right shift	Bit **n** of **x>>s** is bit **n–s** of **x**.
~	complement	Bit **n** of **~x** is the opposite of bit **n** of **x**.

You might find the inclusion of "exclusive or" (**^**, sometimes called "xor") as a fundamental operation odd. However, that's the essential operation in much graphics and encryption code.

Stroustrup_book.indb 956 5/8/15 10:31 AM

The compiler won't confuse a bitwise logical **<<** for an output operator, but you might. To avoid confusion, remember that an output operator takes an **ostream** as its left-hand operand, whereas a bitwise logical operator takes an integer as its left-hand operand.

Note that **&** differs from **&&** and **|** differs from **||** by operating individually on every bit of its operands (§A.5.5), producing a result with as many bits as its operands. In contrast, **&&** and **||** just return **true** or **false**.

Let's try a couple of examples. We usually express bit patterns using hexadecimal notation. For a half byte (4 bits) we have

Hex	Bits
0x0	0000
0x1	0001
0x2	0010
0x3	0011
0x4	0100
0x5	0101
0x6	0110
0x7	0111

Hex	Bits
0x8	1000
0x9	1001
0xa	1010
0xb	1011
0xc	1100
0xd	1101
0xe	1110
0xf	1111

For numbers up to 9 we could have used decimal, but using hexadecimal helps us to remember that we are thinking about bit patterns. For bytes and words, hexadecimal becomes really useful. The bits in a byte can be expressed as two hexadecimal digits. For example:

Hex byte	Bits
0x00	0000 0000
0x0f	0000 1111
0xf0	1111 0000
0xff	1111 1111
0xaa	1010 1010
0x55	0101 0101

So, using **unsigned** (§25.5.3) to keep things as simple as possible, we can write

```
unsigned char a = 0xaa;
unsigned char x0 = ~a;        // complement of a
```

```
 a:  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |  0xaa
~a:  | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |  0x55
```

```
unsigned char b = 0x0f;
unsigned char x1 = a&b;       // a and b
```

```
  a:  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |  0xaa
  b:  | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |  0xf
a&b:  | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |  0xa
```

```
unsigned char x2 = a^b;       // exclusive or: a xor b
```

```
  a:  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |  0xaa
  b:  | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |  0xf
a^b:  | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |  0xa5
```

```
unsigned char x3 = a<<1;      // left shift 1
```

```
    a:  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |  0xaa
 a<<1:  | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |  0x54
```

Note that a 0 is "shifted in" from beyond bit 0 (the least significant bit) to fill up the byte. The leftmost bit (bit 7) simply disappears.

```
unsigned char x4 == a>>2;       // right shift 2
```

```
    a:  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |  0xaa
 a>>2:  | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |  0x2a
```

Note that two 0s are "shifted in" from beyond bit 7 (the most significant bit) to fill up the byte. The rightmost 2 bits (bit 1 and bit 0) simply disappear.

Stroustrup_book.indb 958 5/8/15 10:31 AM

We can draw bit patterns like this and it is good to get a feel for bit patterns, but it soon becomes tedious. Here is a little program that converts integers to their bit representation:

```
int main()
{
        for (unsigned i; cin>>i; )
                cout << dec << i << "=="
                        << hex << "0x" << i << "=="
                        << bitset<8*sizeof(unsigned)>{i} << '\n';
}
```

To print the individual bits of the integer, we use a standard library **bitset**:

```
bitset<8*sizeof(unsigned)>{i}
```

A **bitset** is a fixed number of bits. In this case, we use the number of bits in an **int** – **8*sizeof(unsigned)** – and initialize that **bitset** with our unsigned integer **i**.

TRY THIS

Get the bits example to work and try out a few values to develop a feel for binary and hexadecimal representations. If you get confused about the representation of negative values, just try again after reading §25.5.3.

25.5.2 bitset

The standard library template class **bitset** from **<bitset>** is used to represent and manipulate sets of bits. Each **bitset** is of a fixed size, specified at construction:

```
bitset<4> flags;
bitset<128> dword_bits;
bitset<12345> lots;
```

A **bitset** is by default initialized to "all zeros" but is typically given an initializer; **bitset** initializers can be unsigned integers or strings of zeros and ones. For example:

```
bitset<4> flags = 0xb;
bitset<128> dword_bits {string{"1010101010101010"}};
bitset<12345> lots;
```

Here **lots** will be all zeros, and **dword_bits** will have 112 zeros followed by the 16 bits we explicitly specified. If you try to initialize with a string that has characters different from **'0'** and **'1'**, a **std::invalid_argument** exception is thrown:

```
string s;
cin>>s;
bitset<12345> my_bits{s};     // may throw std::invalid_argument
```

We can use the usual bit manipulation operators for **bitset**s. Assume that **b1**, **b2**, and **b3** are **bitset**s:

```
b1 = b2&b3;          // and
b1 = b2|b3;          // or
b1 = b2^b3;          // xor
b1 = ~b2;            // complement
b1 = b2<<2;          // shift left
b1 = b2>>3;          // shift right
```

Basically, for bit operations (bitwise logical operations), a **bitset** acts like an **unsigned int** (§25.5.3) of an arbitrary, user-specified size. What you can do to an **unsigned int** (with the exception of arithmetic operations), you can do to a **bitset**. In particular, **bitset**s are useful for I/O:

```
cin>>b;                      // read a bitset from input
cout<<bitset<8>{'c'};        // output the bit pattern for the character 'c'
```

When reading into a **bitset**, an input stream looks for zeros and ones. Consider:

```
10121
```

This is read as **101**, leaving **21** unread in the stream.

As for a byte and a word, the bits of a **bitset** are numbered right to left (from the least significant bit toward the most significant), so that, for example, the numerical value of bit 7 is 2^7:

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

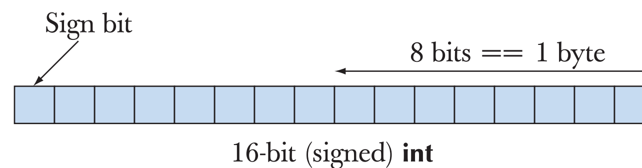
For **bitset**s, the numbering is not just a convention because a **bitset** supports subscripting of bits. For example:

```
int main()
{
	constexpr int max = 10;
	for (bitset<max> b; cin>>b; ) {
		cout << b << '\n';
		for (int i =0; i<max; ++i) cout << b[i];  // reverse order
		cout << '\n';
	}
}
```

If you need a more complete picture of **bitset**s, look them up in your online documentation, a manual, or an expert-level textbook.

25.5.3 Signed and unsigned

Like most languages, C++ supports both signed and unsigned integers. Unsigned integers are trivial to represent in memory: bit0 means 1, bit1 means 2, bit2 means 4, and so on. However, signed integers pose a problem: how do we distinguish between positive and negative numbers? C++ gives the hardware designers some freedom of choice, but almost all implementations use the two's complement representation. The leftmost (most significant bit) is taken as the "sign bit":



16-bit (signed) **int**

If the sign bit is 1, the number is negative. Almost universally, the two's complement representation is used. To save paper, we consider how we would represent signed numbers in a 4-bit integer:

Positive:	0	1	2	4	7
	0000	**0001**	**0010**	**0100**	**0111**
Negative:	**1111**	**1110**	**1101**	**1011**	**1000**
	–1	–2	–3	–5	–8

The bit pattern for **–(x+1)** can be described as the complement of the bits in **x** (also known as **~x**; see §25.5.1).

So far, we have just used signed integers (e.g., **int**). A slightly better set of rules would be:

- Use signed integers (e.g., **int**) for numbers.
- Use unsigned integers (e.g., **unsigned int**) for sets of bits.

That's not a bad rule of thumb, but it's hard to stick to because some people prefer unsigned integers for some forms of arithmetic and we sometimes need to use their code. In particular, for historical reasons going back to the early days of C when **int**s were 16 bits and every bit mattered, **v.size()** for a **vector** is an unsigned integer. For example:

```
vector<int> v;
// . . .
for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';
```

A "helpful" compiler may warn us that we are mixing signed (i.e., **i**) and unsigned (i.e., **v.size()**) values. Mixing signed and unsigned variables could lead to disaster. For example, the loop variable **i** might overflow; that is, **v.size()** might be larger than the largest signed **int**. Then, **i** would reach the highest value that could represent a positive integer in a signed **int** (the number of bits in an **int** minus 1 to the power of two, minus 1, e.g., $2^{15}-1$). Then, the next **++** couldn't yield the next-highest integer and would instead result in a negative value. The loop would never terminate! Each time we reached the largest integer, we would start again from the smallest negative **int** value. So for 16-bit **int**s that loop is a (probably very serious) bug if **v.size()** is 32*1024 or larger; for 32-bit **int**s the problem occurs if **i** reaches 2*1024*1024*1024.

So, technically, most of the loops in this book have been sloppy and could have caused problems. In other words, for an embedded system, we should either have verified that the loop could never reach the critical point or replaced it with a different form of loop. To avoid this problem we can use the **size_type** provided by **vector**, iterators, or a range-**for**-statement:

```
for (vector<int>::size_type i = 0; i<v.size(); ++i) cout << v[i] << '\n';

for (auto p = v.begin(); p!=v.end(); ++p) cout << *p << '\n';

for (int x : v) cout << x << '\n';
```

The **size_type** is guaranteed to be unsigned, so the first (unsigned integer) form has one more bit to play with than the **int** version above. That can be significant, but it still gives only a single bit of range (doubling the number of iterations that can be done). The loop using iterators has no such limitation.

TRY THIS

The following example may look innocent, but it is an infinite loop:

```
void infinite()
{
        unsigned char max = 160;      // very large
        for (signed char i=0; i<max; ++i) cout << int(i) << '\n';
}
```

Run it and explain why.

Basically, there are two reasons for using unsigned integers as integers, as opposed to using them simply as sets of bits (i.e., not using **+**, **–**, *****, and **/**):

- To gain that extra bit of precision
- To express the logical property that the integer can't be negative

The former is what programmers get out of using an unsigned loop variable.

The problem with using both signed and unsigned types is that in C++ (as in C) they convert to each other in surprising and hard-to-remember ways. Consider:

```
unsigned int ui = –1;

int si = ui;
int si2 = ui+2;
unsigned ui2 = ui+2;
```

Surprisingly, the first initialization succeeds and **ui** gets the value 4294967295, which is the unsigned 32-bit integer with the same representation (bit pattern) as the signed integer –1 ("all ones"). Some people consider that neat and use –1 as shorthand for "all ones"; others consider that a problem. The same conversion rule applies from unsigned to signed, so **si** gets the value –1. As we would expect, **si2** becomes 1 (–1+2 == 1), and so does **ui2**. The result for **ui2** ought to surprise you for a second: why should 4294967295+2 be 1? Look at 4294967295 as a hexadecimal number (**0xffffffff**) and things become clearer: 4294967295 is the largest unsigned 32-bit integer, so 4294967297 cannot be represented as a 32-bit integer – unsigned or not. So we say either that 4294967295+2 overflowed or (more precisely) that unsigned integers support modular arithmetic; that is, arithmetic on 32-bit integers is modulo-32 arithmetic.

Is everything clear so far? Even if it is, we hope we have convinced you that playing with that extra bit of precision in an unsigned integer is playing with fire. It can be confusing and is therefore a potential source of errors.

What happens if an integer overflows? Consider:

```
Int i = 0;
while (++i) print(i);      // print i as an integer followed by a space
```

What sequence of values will be printed? Obviously, this depends on the definition of **Int** (no, for once, the use of the capital *I* isn't a typo). For an integer type with a limited number of bits, we will eventually overflow. If **Int** is unsigned (e.g., **unsigned char**, **unsigned int**, or **unsigned long long**), the **++** is modulo arithmetic, so after the largest number that can be represented we get 0 (and the loop terminates). If **Int** is a signed integer (e.g., **signed char**), the numbers will suddenly turn negative and start working their way back up to 0 (where the loop will terminate). For example, for a **signed char**, we will see 1 2 . . . 126 127 –128 –127 . . . –2 –1.

What happens if an integer overflows? The answer is that we proceed as if we had enough bits, but throw away whichever part of the result doesn't fit in the integer into which we store our result. That strategy will lose us the leftmost (most significant) bits. That's the same effect we see when we assign:

```
int si = 257;          // doesn't fit into a char
char c = si;           // implicit conversion to char
unsigned char uc = si;
signed char sc = si;
print(si); print(c); print(uc); print(sc); cout << '\n';

si = 129;              // doesn't fit into a signed char
c = si;
uc = si;
sc = si;
print(si); print(c); print(uc); print(sc);
```

We get

```
257    1      1      1
129    –127   129    –127
```

The explanation of this result is that 257 is two more than will fit into 8 bits (255 is "8 ones") and 129 is two more than can fit into 7 bits (127 is "7 ones") so the sign bit gets set. Aside: This program shows that **char**s on our machine are signed (**c** behaves as **sc** and differs from **uc**).

TRY THIS

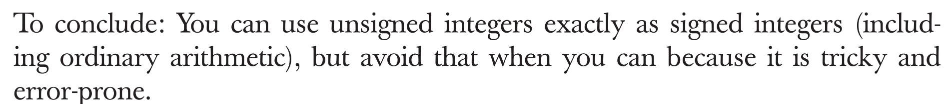
Draw out the bit patterns on a piece of paper. Using paper, then figure out what the answer would be for **si**=128. Then run the program to see if your machine agrees.

An aside: Why did we introduce that **print()** function? We could try

```
cout << i << ' ';
```

However, if **i** was a **char**, we would then output it as a character rather than an integer value. So, to treat all integer types uniformly, we defined

```
template<typename T> void print(T i) { cout << i << '\t'; }

void print(char i) { cout << int(i) << '\t'; }

void print(signed char i) { cout << int(i) << '\t'; }

void print(unsigned char i) { cout << int(i) << '\t'; }
```



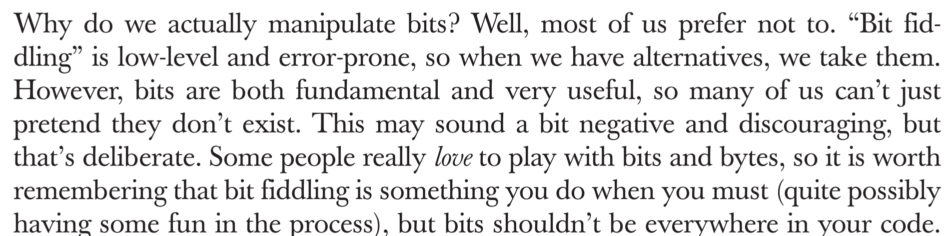
To conclude: You can use unsigned integers exactly as signed integers (including ordinary arithmetic), but avoid that when you can because it is tricky and error-prone.

- Try never to use unsigned just to get another bit of precision.
- If you need one extra bit, you'll soon need another.

Unfortunately, you can't completely avoid unsigned arithmetic:

- Subscripting for standard library containers uses unsigned.
- Some people like unsigned arithmetic.

25.5.4 Bit manipulation



Why do we actually manipulate bits? Well, most of us prefer not to. "Bit fiddling" is low-level and error-prone, so when we have alternatives, we take them. However, bits are both fundamental and very useful, so many of us can't just pretend they don't exist. This may sound a bit negative and discouraging, but that's deliberate. Some people really *love* to play with bits and bytes, so it is worth remembering that bit fiddling is something you do when you must (quite possibly having some fun in the process), but bits shouldn't be everywhere in your code.

To quote John Bentley: "People who play with bits will be bitten" and "People who play with bytes will be bytten."

So, when do we manipulate bits? Sometimes the natural objects of our application simply are bits, so that some of the natural operations in our application domain are bit operations. Examples of such domains are hardware indicators ("flags"), low-level communications (where we have to extract values of various types out of byte streams), graphics (where we have to compose pictures out of several levels of images), and encryption (see the next section).

For example, consider how to extract (low-level) information from an integer (maybe because we wanted to transmit it as bytes, the way binary I/O does):

```
void f(short val)                        // assume 16-bit, 2-byte short integer
{
        unsigned char right = val&0xff;  // rightmost (least significant) byte
        unsigned char left = val>>8;     // leftmost (most significant) byte
        // . . .
        bool negative = val&0x8000;      // sign bit
        // . . .
}
```

Such operations are common. They are known as "shift and mask." We "shift" (using **<<** or **>>**) to place the bits we want to consider to the rightmost (least significant) part of the word where they are easy to manipulate. We "mask" using and (**&**) together with a bit pattern (here **0xff**) to eliminate (set to zero) the bits we do not want in the result.

When we want to name bits, we often use enumerations. For example:

```
enum Printer_flags {
        acknowledge=1,
        paper_empty=1<<1,
        busy=1<<2,
        out_of_black=1<<3,
        out_of_color=1<<4,
        // . . .
};
```

This defines each enumerator to have exactly the value that its name indicates:

```
out_of_color     16    0x10    0001 0000
out_of_black     8     0x8     0000 1000
busy             4     0x4     0000 0100
paper_empty      2     0x2     0000 0010
acknowledge      1     0x1     0000 0001
```

Such values are useful because they can be combined independently:

```
unsigned char x = out_of_color | out_of_black;    // x becomes 24 (16+8)
x |= paper_empty;                                  // x becomes 26 (24+2)
```

Note how **|=** can be read as "set a bit" (or as "set some bits"). Similarly, **&** can be read as "Is a bit set?" For example:

```
if (x& out_of_color) {      // is out_of_color set? (yes, it is)
    // . . .
}
```

We can still use **&** to mask:

```
unsigned char y = x &(out_of_color | out_of_black);   // y becomes 24
```

Now **y** has a copy of the bits from **x**'s positions 4 and 3 (**out_of_color** and **out_of_black**).

It is very common to use an **enum** as a set of bits. When doing that, we need a conversion to get the result of a bitwise logical operation "back into" the **enum**. For example:

```
Flags z = Printer_flags(out_of_color | out_of_black);   // the cast is necessary
```

The reason that the cast is needed is that the compiler cannot know that the result of **out_of_color | out_of_black** is a valid value for a **Flags** variable. The compiler's skepticism is warranted: after all, no enumerator has a value 24 **(out_of_color | out_of_black)**, but in this case, we know the assignment to be reasonable (but the compiler does not).

25.5.5 Bitfields

As mentioned, the hardware interface is one area where bits occur frequently. Typically, an interface is defined as a mixture of bits and numbers of various sizes. These "bits and numbers" are typically named and occur in specific positions of a word, often called a *device register*. C++ has a specific language facility to deal with such fixed layouts: *bitfields*. Consider a page number as used in the page manager deep in an operating system. Here is a diagram from an operating system manual:

position:	31:	9:	6:	3:	2:	1:	0:
PPN:	22	3	3	1	1	1	1
name:	PFN	unused	CCA	nonreachable	dirty	valid	global

Stroustrup_book.indb 967 5/8/15 10:31 AM

The 32-bit word is used as two numeric fields (one of 22 bits and one of 3 bits) and four flags (1 bit each). The sizes and positions of these pieces of data are fixed. There is even an unused (and unnamed) "field" in the middle. We can express this as a **struct**:

```
struct PPN {                        // R6000 Physical Page Number
        unsigned int PFN : 22 ;     // Page Frame Number
        int : 3 ;                   // unused
        unsigned int CCA : 3 ;      // Cache Coherency Algorithm
        bool nonreachable : 1 ;
        bool dirty : 1 ;
        bool valid : 1 ;
        bool global : 1 ;
};
```

We had to read the manual to see that **PFN** and **CCA** should be interpreted as unsigned integers, but otherwise we could write out that **struct** directly from the diagram. Bitfields fill a word left to right. You give the number of bits as an integer value after a colon. You can't specify an absolute position (e.g., bit 8). If you "consume" more bits with bitfields than a word can hold, the fields that don't fit are put into the next word. Hopefully, that's what you want. Once defined, a bitfield is used exactly like other variables:

```
void part_of_VM_system(PPN * p )
{
        // . . .
        if (p->dirty) { // contents changed
                // copy to disk
                p->dirty = 0 ;
        }
        // . . .
}
```

Bitfields primarily save you the bother of shifting and masking to get to information placed in the middle of a word. For example, given a **PPN** called **pn** we could extract **CCA** like this:

```
unsigned int x = pn.CCA;            // extract CCA
```

Had we used an **int** called **pni** to represent the same bits, we could instead have written

```
unsigned int y = (pni>>4)&0x7;      // extract CCA
```

That is, shift **pn** right so that **CCA** is the leftmost bit, then mask all other bits off with **0x7** (i.e., last three bits set). If you look at the machine code, you'll most likely find that the generated code is identical for those two lines.

The "acronym soup" (**CCA**, **PPN**, **PFN**) is typical of code at this level and makes little sense out of context.

25.5.6 An example: simple encryption

As an example of manipulation of data at the level of the data's representation as bits and bytes, let us consider a simple encryption algorithm: the Tiny Encryption Algorithm (TEA). It was originally written by David Wheeler of Cambridge University (§22.2.1). It is small but the protection against undesired decryption is excellent.

Don't look too hard at the code (unless you really want to and are willing to risk a headache). We present the code simply to give you the flavor of some real-world and useful bit manipulation code. If you want to make a study of encryption, you need a separate textbook for that. For more information and variants of the algorithm in other languages, see http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm and the TEA website of Professor Simon Shepherd, Bradford University, England. The code is not meant to be self-explanatory (no comments!).

The basic idea of enciphering/deciphering (also known as encryption/decryption) is simple. I want to send you some text, but I don't want others to read it. Therefore, I transform the text in a way that renders it unreadable to people who don't know exactly how I modified it – but in such a way that you can reverse my transformation and read the text. That's called enciphering. To encipher I use an algorithm (which we must assume an uninvited listener knows) and a string called the "key." Both you and I have the key (and we hope that the uninvited listener does not). When you get the enciphered text, you decipher it using the "key"; that is, you reconstitute the "clear text" that I sent.

TEA takes as argument an array of two unsigned **long**s (**v[0]**,**v[1]**) representing eight characters to be enciphered, an array of two unsigned **long**s (**w[0]**,**w[1]**) into which the enciphered output is written, and an array of four unsigned **long**s (**k[0]..k[3]**), which is the key:

```
void encipher(
        const unsigned long *const v,
        unsigned long *const w,
        const unsigned long * const k)
{
        static_assert(sizeof(long)==4,"size of long wrong for TEA");

        unsigned long y = v[0];
        unsigned long z = v[1];
```

```
        unsigned long sum = 0;
        const unsigned long delta = 0x9E3779B9;

        for (unsigned long n = 32; n-->0; ) {
                y += (z<<4 ^ z>>5) + z^sum + k[sum&3];
                sum += delta;
                z += (y<<4 ^ y>>5) + y^sum + k[sum>>11 & 3];
        }
        w[0]=y;
        w[1]=z;
}
```

Note how all data is unsigned so that we can perform bitwise operations on it without fear of surprises caused by special treatment related to negative numbers. Shifts (**<<** and **>>**), exclusive or (**^**), and bitwise and (**&**) do the essential work with an ordinary (unsigned) addition thrown in for good measure. This code is specifically written for a machine where there are 4 bytes in a **long**. The code is littered with "magic" constants (e.g., it assumes that **sizeof(long)** is 4). That's generally not a good idea, but this particular piece of software fits on a single sheet of paper. As a mathematical formula, it fits on the back of an envelope or – as originally intended – in the head of a programmer with a good memory. David Wheeler wanted to be able to encipher things while he was traveling without bringing notes, a laptop, etc. In addition to being small, this code is also fast. The variable **n** determines the number of iterations: the higher the number of iterations, the stronger the encryption. To the best of our knowledge, for **n==32** TEA has never been broken.

Here is the corresponding deciphering function:

```
void decipher(
        const unsigned long *const v,
        unsigned long *const w,
        const unsigned long * const k)
{
        static_assert(sizeof(long)==4,"size of long wrong for TEA");

        unsigned long y = v[0];
        unsigned long z = v[1];
        unsigned long sum = 0xC6EF3720;
        const unsigned long delta = 0x9E3779B9;
```

```
        // sum = delta<<5, in general sum = delta * n
        for (unsigned long n = 32; n-- > 0; ) {
                z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
                sum -= delta;
                y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
        }
        w[0]=y;
        w[1]=z;
}
```

We can use TEA like this to produce a file to be sent over an unsafe connection:

```
int main()      // sender
{
        const int nchar = 2*sizeof(long);       // 64 bits
        const int kchar = 2*nchar;              // 128 bits

        string op;
        string key;
        string infile;
        string outfile;
        cout << "please enter input file name, output file name, and key:\n";
        cin >> infile >> outfile >> key;
        while (key.size()<kchar) key += '0';   // pad key
        ifstream inf(infile);
        ofstream outf(outfile);
        if (!inf || !outf) error("bad file name");

        const unsigned long* k =
                reinterpret_cast<const unsigned long*>(key.data());

        unsigned long outptr[2];
        char inbuf[nchar];
        unsigned long* inptr = reinterpret_cast<unsigned long*>(inbuf);
        int count = 0;

        while (inf.get(inbuf[count])) {
                outf << hex;                    // use hexadecimal output
                if (++count == nchar) {
                        encipher(inptr,outptr,k);
                        // pad with leading zeros:
```

```
                    outf << setw(8) << setfill('0') << outptr[0] << ' '
                         << setw(8) << setfill('0') << outptr[1] << ' ';
                    count = 0;
               }
          }

          if (count) { // pad
               while(count != nchar) inbuf[count++] = '0';
               encipher(inptr,outptr,k);
               outf << outptr[0] << ' ' << outptr[1] << ' ';
          }
     }
```

The essential piece of code is the **while**-loop; the rest is just support. The **while**-loop reads characters into the input buffer, **inbuf**, and every time it has eight characters as needed by TEA it passes them to **encipher()**. TEA doesn't care about characters; in fact, it has no idea what it is enciphering. For example, you could encipher a photo or a phone conversation. All TEA cares about is that it is given 64 bits (two unsigned **long**s) so that it can produce a corresponding 64 bits. So, we take a pointer to the **inbuf** and cast it to an unsigned **long*** and pass that to TEA. We do the same for the key; TEA will use the first 128 bits (four unsigned **long**s) of the key, so we "pad" the user's input to be sure that there are 128 bits. The last statement pads the text with zeros to make up the multiple of 64 bits (8 bytes) required by TEA.

How do we transmit the enciphered text? We have a free choice, but since it is "just bits" rather than ASCII or Unicode characters, we can't really treat it as ordinary text. Binary I/O (see §11.3.2) would be an option, but here we decided to output the output words as hexadecimal numbers:

```
5b8fb57c   806fbcce   2db72335   23989d1d   991206bc   0363a308
8f8111ac   38f3f2f3   9110a4bb   c5e1389f   64d7efe8   ba133559
4cc00fa0   6f77e537   bde7925f   f87045f0   472bad6e   dd228bc3
a5686903   51cc9a61   fc19144e   d3bcde62   4fdb7dc8   43d565e5
f1d3f026   b2887412   97580690   d2ea4f8b   2d8fb3b7   936cfa6d
6a13ef90   fd036721   b80035e1   7467d8d8   d32bb67e   29923fde
197d4cd6   76874951   418e8a43   e9644c2a   eb10e848   ba67dcd8
7115211f   dbe32069   e4e92f87   8bf3e33e   b18f942c   c965b87a
44489114   18d4f2bc   256da1bf   c57b1788   9113c372   12662c23
eeb63c45   82499657   a8265f44   7c866aae   7c80a631   e91475e1
5991ab8b   6aedbb73   71b642c4   8d78f68b   d602bfe4   d1eadde7
55f20835   1a6d3a4b   202c36b8   66a1e0f2   771993f3   11d1d0ab
```

Stroustrup_book.indb 972 5/8/15 10:31 AM

```
74a8cfd4  4ce54f5a  e5fda09d  acbdf110  259a1a19  b964a3a9
456fd8a3  1e78591b  07c8f5a2  101641ec  d0c9d7e1  60dbeb11
b9ad8e72  ad30b839  201fc553  a34a79c4  217ca84d  30f666c6
d018e61c  d1c94ea6  6ca73314  cd60def1  6e16870e  45b94dc0
d7b44fcd  96e0425a  72839f71  d5b6427c  214340f9  8745882f
0602c1a2  b437c759  ca0e3903  bd4d8460  edd0551e  31d34dd3
c3f943ed  d2cae477  4d9d0b61  f647c377  0d9d303a  ce1de974
f9449784  df460350  5d42b06c  d4dedb54  17811b5f  4f723692
14d67edb  11da5447  67bc059a  4600f047  63e439e3  2e9d15f7
4f21bbbe  3d7c5e9b  433564f5  c3ff2597  3a1ea1df  305e2713
9421d209  2b52384f  f78fbae7  d03c1f58  6832680a  207609f3
9f2c5a59  ee31f147  2ebc3651  e017d9d6  d6d60ce2  2be1f2f9
eb9de5a8  95657e30  cad37fda  7bce06f4  457daf44  eb257206
418c24a5  de687477  5c1b3155  f744fbff  26800820  92224e9d
43c03a51  d168f2d1  624c54fe  73c99473  1bce8fbb  62452495
5de382c1  1a789445  aa00178a  3e583446  dcbd64c5  ddda1e73
fa168da2  60bc109e  7102ce40  9fed3a0b  44245e5d  f612ed4c
b5c161f8  97ff2fc0  1dbf5674  45965600  b04c0afa  b537a770
9ab9bee7  1624516c  0d3e556b  6de6eda7  d159b10e  71d5c1a6
b8bb87de  316a0fc9  62c01a3d  0a24a51f  86365842  52dabf4d
372ac18b  9a5df281  35c9f8d7  07c8f9b4  36b6d9a5  a08ae934
239efba5  5fe3fa6f  659df805  faf4c378  4c2048d6  e8bf4939
31167a93  43d17818  998ba244  55dba8ee  799e07e7  43d26aef
d5682864  05e641dc  b5948ec8  03457e3f  80c934fe  cc5ad4f9
0dc16bb2  a50aa1ef  d62ef1cd  f8fbbf67  30c17f12  718f4d9a
43295fed  561de2a0
```

TRY THIS

The key was **bs**; what was the text?

Any security expert will tell you that it is a dumb idea to store clear text and enciphered files together and also express an opinion about padding, about using a two-letter key, etc., but this is a programming book, rather than a book on computer security.

We tested the programs by reading the enciphered text and getting the original back. When writing a program, it is always nice to be able to conduct a simple test of correctness.

Here is the central part of the deciphering program:

```
unsigned long inptr[2];
char outbuf[nchar+1];
outbuf[nchar]=0;    // terminator
unsigned long* outptr = reinterpret_cast<unsigned long*>(outbuf);
inf.setf(ios_base::hex ,ios_base::basefield);    // use hexadecimal input

while (inf>>inptr[0]>>inptr[1]) {
      decipher(inptr,outptr,k);
      outf<<outbuf;
}
```

Note the use of

```
inf.setf(ios_base::hex ,ios_base::basefield);
```

to read the hexadecimal numbers. For decryption, it's the output buffer, **outbuf**, that we treat as bits using a cast.

Is TEA an example of embedded systems programming? Not specifically, but you can imagine it being used wherever privacy is needed or financial transactions are conducted – that could include many "gadgets." Anyway, TEA demonstrates many of the characteristics of good embedded systems code: it is based on a well-understood (mathematical) model that makes us confident about its correctness, it's small, it's fast, and it relies directly on hardware properties. The interface style of **encipher()** and **decipher()** is not quite to our taste. However, **encipher()** and **decipher()** were designed to be C as well as C++ functions, so no C++ facilities that are not also supported by C could be used. In addition, the many "magic constants" came from direct hand translation from the math.

25.6 Coding standards

There are many sources of errors. The most serious and hardest to remedy relate to high-level design decisions, such as overall error-handling strategies, conformance to certain standards (or lack thereof), algorithms, the representation of data, etc. These problems are *not* the ones we address here. Instead, we focus on errors that arise from code that is poorly written, that is, code that uses programming language facilities in unnecessarily error-prone ways or expresses ideas in ways that obscure their meaning.

Coding standards try to address the latter kinds of problems by defining a "house style" that guides programmers to a subset of the C++ language that is deemed appropriate for a given application. For example, a coding standard for

an embedded system involving hard real-time constraints or for a system needing to run "forever" may prohibit the use of **new**. Typically a coding standard also tries to ensure that code written by two programmers is more similar than if they had chosen freely from all possible styles. For example, a coding standard may require that **for**-statements be used for loops (thereby banning **while**-statements). This can make code more uniform, and in large projects that can be important for maintenance. Please note that a coding standard is aimed at improving code for a specific kind of programming given a specific kind of programmer. There is no one coding standard suitable for all C++ applications and all C++ programmers.

So, the problems that a coding standard tries to address are problems that arise from the way we express our solutions rather than the problems that arise from inherent complexities of the problem we are trying to solve with our application. We could say that coding standards are trying to address incidental complexities rather that inherent complexities.

The major sources of such incidental complexities are

- *Overly clever programmers,* who use features they don't understand or delight in complicated solutions
- *Undereducated programmers,* who don't use the most appropriate language and library features
- *Unnecessary variations in programming style,* causing code performing similar tasks to look different and confuse maintainers
- *Inappropriate programming language,* leading to use of language features that are poorly adapted to a particular application area or to a particular group of programmers
- *Insufficient library use,* leading to lots of ad hoc manipulation of low-level resources
- *Inappropriate coding standards,* causing extra work or prohibiting the best solution to some classes of problems, thus becoming a source of the kind of problems that the standards were introduced to solve

25.6.1 What should a coding standard be?

A good coding standard should help a programmer write good code; that is, it should help the programmer by giving answers to lots of little questions that each programmer would otherwise have to spend time deciding on a case-by-case basis. There is an old engineer's proverb that says, "Form is liberating." Ideally, a coding standard should be prescriptive, stating what should be done. That seems obvious, but many coding standards are simply a list of prohibitions, with no guidance about what to do after having obeyed a long list of *don't*s. Just being told what not to do is rarely helpful and often annoying.

The rules of a good coding standard should be verifiable, preferably by a program; that is, once we have written the code, we should be able to look at it and easily answer the question, "Have I broken any rule of my coding standard?"

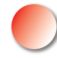
A good coding standard should present a rationale for the rules. Programmers should not just be told, "Because that's the way we do it!" When they are, they resent it. Worse, programmers invariably try to subvert parts of a coding standard that they see as pointless and as preventing them from doing a good job. Don't expect to like everything about a coding standard. Even the best coding standard is a compromise, and most prohibit certain practices assumed to cause problems – even if they never caused you a problem. For example, inconsistent naming rules are a source of confusion, but different people have strong attachments to some naming conventions and strong dislikes of others. For example, I consider the **CamelCodingStyle** of identifiers "pug ugly" and strongly prefer **underscore_style** as cleaner and inherently more readable, and many people agree. On the other hand, many reasonable people disagree. Obviously, no naming standard can please everyone, but in this case, as in many others, a consistent style is definitely better than the lack of a standard.

To summarize:

- A good coding standard is designed for a specific application domain and a specific group of programmers.
- A good coding standard is prescriptive as well as restrictive.
  - Recommending some "foundation" library facilities is often the most effective use of prescriptive rules.
- A coding standard is a set of rules for what code should look like,
  - Typically specifying naming and indentation rules; e.g., "Use 'Stroustrup layout.'"
  - Typically specifying a subset of a language; e.g., "Don't use **new** or **throw**."
  - Typically specifying rules for commenting; e.g., "Every function must have a comment explaining what it does."
  - Often requiring the use of certain libraries; e.g., "Use **<iostream>** rather than **<stdio.h>**" or "Use **vector** and **string** rather than built-in arrays and C-style strings."
- Common aims of most coding standards are to improve
  - Reliability
  - Portability
  - Maintainability
  - Testability

  - Reusability
  - Extensibility
  - Readability

- A good coding standard is better than no standard. We wouldn't start a major (multi-person, multi-year) industrial project without one.



- A poor coding standard can be worse than no standard. For example, C++ coding standards that restrict programming to something like the C subset do harm. Unfortunately, poor coding standards are not uncommon.
- All coding standards are disliked by programmers, even the good ones. Most programmers want to write their code exactly the way they like it.

25.6.2 Sample rules

Here, we would like to give you a flavor of a coding standard by listing some rules. Naturally, we pick rules that we hope will be useful to you. However, we have never seen a real-world coding standard that could be described in fewer than 35 pages, and most are much longer. So, we don't try to give you a complete set of rules here. Furthermore, every good coding standard is designed for a particular application area and for a particular set of programmers. So, we don't make any pretenses of universality.

The rules are numbered and contain a (brief) rationale. Many rules contain examples for easier comprehension. We distinguish between *recommendations*, which a programmer may occasionally decide to ignore, and *firm rules*, which must be followed. In a real set of rules, a firm rule can usually be broken (only) with written permission from a supervisor. Each violation of a recommendation or a firm rule requires a comment in the code. Any exceptions to a rule can be listed in the rule. A firm rule is identified by a capital *R* in its number. A recommendation is identified by a lowercase *r* in its number.

The rules are classified as

- General
- Preprocessor
- Naming and layout
- Class rules
- Function and expression rules
- Hard real time
- Critical systems

The "hard real-time" and "critical systems" rules apply only to projects classified as such.

Compared to a good real-world coding standard, our terminology is underspecified (e.g., what does "critical" really mean?) and the rules overly terse. Similarities between these rules and the JSF++ rules (see §25.6.3) are not accidental; I helped formulate the JSF++ rules. However, the code examples in this book do not conform to the rules below – after all, the book code is not critical embedded systems code.

General rules

R100: Any one function or class shall contain no more than 200 logical source lines of code (non-comments).

Reason: Long functions and long classes tend to be complex and therefore difficult to comprehend and test.

r101: Any one function or class should fit on a screen and serve a single logical purpose.

Reason: A programmer looking at only part of a function or class is more likely to overlook a problem. A function that tries to perform several logical functions is likely to be longer and more complex than one that doesn't.

R102: All code shall conform to ISO/IEC 14882:2011(E) standard C++.

Reason: Language extensions or variations from ISO/IEC 14882 are likely to be less stable, to be less well specified, and to limit portability.

Preprocessor rules

R200: No macros shall be used except for source control using **#ifdef** and **#ifndef**.

Reason: Macros don't obey scope and type rules. Macro use is not obvious when visually examining source text.

R201: **#include** shall be used only to include header (***.h**) files.

Reason: **#include** is used to access interface declarations – not implementation details.

R202: All **#include** directives shall precede all non-preprocessor declarations.

Reason: An **#include** in the middle of a file is more likely to be overlooked by a reader and to cause inconsistencies from a name resolved differently in different places.

R203: Header files (***.h**) shall not contain non-**const** variable definitions or non-inline, non-template function definitions.

Reason: Header files should contain interface declarations – not implementation details. However, constants are often seen as part of the interface, some very simple functions need to be inline (and therefore in headers) for performance, and current template implementations require complete template definitions in headers.

Naming and layout

R300: Indentations shall be used and be consistent within the same source file.

Reason: Readability and style.

R301: Each new statement starts on a new line.

Reason: Readability.

Example:

```
int a = 7; x = a+7; f(x,9);    // violation
int a = 7;                     // OK
x = a+7;                       // OK
f(x,9);                        // OK
```

Example:

```
if (p<q) cout << *p;           // violation
```

Example:

```
if (p<q)
    cout << *p;                // OK
```

R302: Identifiers should be given descriptive names.

Identifiers may contain common abbreviations and acronyms.

When used conventionally, **x**, **y**, **i**, **j**, etc. are descriptive.

Use the **number_of_elements** style rather than the **numberOfElements** style.

Hungarian notation shall not be used.

Type, template, and namespace names (only) start with a capital letter.

Avoid excessively long names.

Example: **Device_driver** and **Buffer_pool**.

Reason: Readability.

Note: Identifiers starting with an underscore are reserved to the language implementation by the C++ standard and thus banned.

Exception: When calling an approved library, the names from that library may be used.

R303: Identifiers shall not differ only by

- A mixture of case
- The presence/absence of the underscore character
- The interchange of the letter *O* with the number 0 or the letter *D*
- The interchange of the letter *I* with the number 1 or the letter *l*
- The interchange of the letter *S* with the number 5
- The interchange of the letter *Z* with the number 2
- The interchange of the letter *n* with the letter *h*

Example: **Head** and **head** *// violation*

Reason: Readability.

R304: No identifier shall be in all capital letters and underscores.

Example: **BLUE** and **BLUE_CHEESE** *// violation*

Reason: All capital letters are widely used for macros that may be used in **#include** files for approved libraries.

Exception: Macro names used for **#include** guards.

Function and expression rules

r400: Identifiers in an inner scope should not be identical to identifiers in an outer scope.

Example:

```
int var = 9; { int var = 7; ++var; }     // violation: var hides var
```

Reason: Readability.

R401: Declarations shall be declared in the smallest possible scope.

Reason: Keeping initialization and use close minimizes chances of confusion; letting a variable go out of scope releases its resources.

R402: Variables shall be initialized.

Example:

```
int var;                                 // violation: var is not initialized
```

Reason: Uninitialized variables are a common source of errors.

Exception: A variable that is immediately filled from input need not be initialized.

Note: Many types, such as **vector** and **string**, have a default constructor to guarantee initialization.

R403: Casts shall not be used.

Reason: Casts are a common source of errors.

Exception: **dynamic_cast** may be used.

Exception: Named casts may be used to convert hardware addresses into pointers and **void*** received from sources external to a program (e.g., a GUI library) into pointers of a proper type.

R404: Built-in arrays shall not be used in interfaces; that is, a pointer as function argument shall be assumed to point to a single element. Use **Array_ref** to pass arrays.

Reason: An array is passed as a pointer and its number of elements is not carried along to the called function. Also, the combination of implicit array-to-pointer conversion and implicit derived-to-base conversion can lead to memory corruption.

Class rules

R500: Use **class** for classes with no public data members. Use **struct** for classes with no private data members. Don't use classes with both public and private data members.

Reason: Clarity.

r501: If a class has a destructor or a member of pointer or reference type, it must have a copy constructor and a copy assignment defined or prohibited.

Reason: A destructor usually releases a resource. The default copy semantics rarely does "the right thing" for pointer and reference members or for a class with a destructor.

R502: If a class has a virtual function it must have a virtual destructor.

Reason: A class has a virtual function so that it can be used through a base class interface. A function that knows an object only through that base class may delete it and derived classes need a chance to clean up (in their destructors).

r503: A constructor that accepts a single argument must be declared **explicit**.

Reason: To avoid surprising implicit conversions.

Hard real-time rules

R800: Exceptions shall not be used.

Reason: Not predictable.

R801: **new** shall be used only during startup.

Reason: Not predictable.

Exception: Placement-**new** (with the standard meaning) may be used for memory allocated from stacks.

R802: **delete** shall not be used.

Reason: Not predictable; can cause fragmentation.

R803: **dynamic_cast** shall not be used.

Reason: Not predictable (assuming common implementation technique).

R804: The standard library containers, except **std::array**, shall not be used.

Reason: Not predictable (assuming common implementation technique).

Critical systems rules

R900: Increment and decrement operations shall not be used as sub-expressions.

Example:

```
int x = v[++i];      // violation
```

Example:

```
++i;
int x = v[i];        // OK
```

Reason: Such an increment might be overlooked.

R901: Code should not depend on precedence rules below the level of arithmetic expressions.

Example:

```
x = a*b+c;           // OK
```

Example:

```
if ( a<b || c<=d)    // violation: parenthesize(a<b) and (c<=d)
```

Reason: Confusion about precedence has been repeatedly found in code written by programmers with a weak C/C++ background.

We left gaps in the numbering so that we could add new rules without changing the numbering of existing ones and still have the general classification recognized

through the numbering. It is very common for rules to become known by their number, so that renumbering would be resisted by the users.

25.6.3 Real coding standards

There are lots of C++ coding standards. Most are corporate and not widely available. In many cases, that's probably a good thing except possibly for the programmers of those corporations. Here is a list of standards that – when used appropriately in areas to which they apply – can do some good:

Google C++ Style Guide: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml. A rather old-style and restrictive but evolving style guide.

Lockheed Martin Corporation. *Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program*. Document Number 2RDU00001 Rev C. December 2005. Colloquially known as "JSF++"; a set of rules written at Lockheed-Martin Aero for air vehicle (read "airplane") software. These rules really were written by and for programmers who produce software upon which human lives depend. www.stroustrup.com/JSF-AV-rules.pdf.

Programming Research. High-integrity C++ Coding Standard Manual Version 2.4. www.programmingresearch.com.

Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586. This is more of a "meta coding standard"; that is, instead of specific rules it has guidance on which rules are good and why.

Note that there is no substitute for knowing your application area, your programming language, and the relevant programming technique. For most applications – and certainly for most embedded systems programming – you also need to know your operating system and/or hardware architecture. If you need to use C++ for low-level coding, have a look at the ISO C++ committee's report on performance (ISO/IEC TR 18015, www.stroustrup.com/performanceTR.pdf); by "performance" they/we primarily mean "embedded systems programming."

Language dialects and proprietary languages abound in the embedded systems world, but whenever you can, use standardized language (such as ISO C++), tools, and libraries. That will minimize your learning curve and increase the likelihood that your work will last.

Stroustrup_book.indb 983 5/8/15 10:31 AM

Drill

1. Run this:

```
int v = 1; for (int i = 0; i<sizeof(v)*8; ++i) { cout << v << ' '; v <<=1; }
```

2. Run that again with **v** declared to be an **unsigned int**.
3. Using hexadecimal literals, define **short unsigned int**s with:
   a. Every bit set
   b. The lowest (least significant bit) set
   c. The highest (most significant bit) set
   d. The lowest byte set
   e. The highest byte set
   f. Every second bit set (and the lowest bit 1)
   g. Every second bit set (and the lowest bit 0)
4. Print each as a decimal and as a hexidecimal.
5. Do 3 and 4 using bit manipulation operations (**|**, **&**, **<<**) and (only) the literals 1 and 0.

Review

1. What is an embedded system? Give ten examples, out of which at least three should not be among those mentioned in this chapter.
2. What is special about embedded systems? Give five concerns that are common.
3. Define predictability in the context of embedded systems.
4. Why can it be hard to maintain and repair an embedded system?
5. Why can it be a poor idea to optimize a system for performance?
6. Why do we prefer higher levels of abstraction to low-level code?
7. What are transient errors? Why do we particularly fear them?
8. How can we design a system to survive failure?
9. Why can't we prevent every failure?
10. What is domain knowledge? Give examples of application domains.
11. Why do we need domain knowledge to program embedded systems?
12. What is a subsystem? Give examples.
13. From a C++ language point of view, what are the three kinds of storage?
14. When would you like to use the free store?
15. Why is it often infeasible to use the free store in an embedded system?
16. When can you safely use **new** in an embedded system?
17. What is the potential problem with **std::vector** in the context of embedded systems?

18. What is the potential problem with exceptions in the context of embedded systems?
19. What is a recursive function call? Why do some embedded systems programmers avoid them? What do they use instead?
20. What is memory fragmentation?
21. What is a garbage collector (in the context of programming)?
22. What is a memory leak? Why can it be a problem?
23. What is a resource? Give examples.
24. What is a resource leak and how can we systematically prevent it?
25. Why can't we easily move objects from one place in memory to another?
26. What is a stack?
27. What is a pool?
28. Why doesn't the use of stacks and pools lead to memory fragmentation?
29. Why is **reinterpret_cast** necessary? Why is it nasty?
30. Why are pointers dangerous as function arguments? Give examples.
31. What problems can arise from using pointers and arrays? Give examples.
32. What are alternatives to using pointers (to arrays) in interfaces?
33. What is "the first law of computer science"?
34. What is a bit?
35. What is a byte?
36. What is the usual number of bits in a byte?
37. What operations do we have on sets of bits?
38. What is an "exclusive or" and why is it useful?
39. How can we represent a set (sequence, whatever) of bits?
40. How do we conventionally number bits in a word?
41. How do we conventionally number bytes in a word?
42. What is a word?
43. What is the usual number of bits in a word?
44. What is the decimal value of **0xf7**?
45. What sequence of bits is **0xab**?
46. What is a **bitset** and when would you need one?
47. How does an **unsigned int** differ from a **signed int**?
48. When would you prefer an **unsigned int** to a **signed int**?
49. How would you write a loop if the number of elements to be looped over was very high?
50. What is the value of an **unsigned int** after you assign **–3** to it?
51. Why would we want to manipulate bits and bytes (rather than higher-level types)?
52. What is a bitfield?
53. For what are bitfields used?
54. What is encryption (enciphering)? Why do we use it?
55. Can you encrypt a photo?

56. What does TEA stand for?
57. How do you write a number to output in hexadecimal notation?
58. What is the purpose of coding standards? List reasons for having them.
59. Why can't we have a universal coding standard?
60. List some properties of a good coding standard.
61. How can a coding standard do harm?
62. Make a list of at least ten coding rules that you like (have found useful). Why are they useful?
63. Why do we avoid ALL_CAPITAL identifiers?

Terms

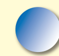
address	encryption	pool
bit	exclusive or	predictability
bitfield	gadget	real time
bitset	garbage collector	resource
coding standard	hard real time	soft real time
embedded system	leak	**unsigned**

Exercises

1. If you haven't already, do the **Try this** exercises in this chapter.
2. Make a list of words that can be spelled with hexadecimal notation. Read 0 as *o*, read 1 as *l*, read 2 as *to*, etc.; for example, Foo1 and Beef. Kindly eliminate vulgarities from the list before submitting it for grading.
3. Initialize a 32-bit signed integer with the bit patterns and print the result: all zeros, all ones, alternating ones and zeros (starting with a leftmost one), alternating zeros and ones (starting with a leftmost zero), the 110011001100 . . . pattern, the 001100110011 . . . pattern, the pattern of all-one bytes and all-zero bytes starting with an all-one byte, the pattern of all-one bytes and all-zero bytes starting with an all-zero byte. Repeat that exercise with a 32-bit unsigned integer.
4. Add the bitwise logical operators **&**, **|**, **^**, and **~** to the calculator from Chapter 7.
5. Write an infinite loop. Execute it.
6. Write an infinite loop that is hard to recognize as an infinite loop. A loop that isn't really infinite because it terminates after completely consuming some resource is acceptable.
7. Write out the hexadecimal values from 0 to 400; write out the hexadecimal values from –200 to 200.
8. Write out the numerical values of each character on your keyboard.
9. Without using any standard headers (such as **<limits>**) or documentation, compute the number of bits in an **int** and determine whether **char** is signed or unsigned on your implementation.

10. Look at the bitfield example from §25.5.5. Write an example that initializes a **PPN**, then reads and prints each field value, then changes each field value (by assigning to the field) and prints the result. Repeat this exercise, but store the **PPN** information in a 32-bit unsigned integer and use bit manipulation operators (§25.5.4) to access the bits in the word.
11. Repeat the previous exercise, but keep the bits in a **bitset<32>**.
12. Write out the clear text of the example from §25.5.6.
13. Use TEA (§25.5.6) to communicate "securely" between two computers. Email is minimally acceptable.
14. Implement a simple **vector** that can hold at most N elements allocated from a pool. Test it for N==1000 and integer elements.
15. Measure the time (§26.6.1) it takes to allocate 10,000 objects of random sizes in the [1000:0)-byte range using **new**; then measure the time it takes to deallocate them using **delete**. Do this twice, once deallocating in the reverse order of allocation and once deallocating in random order. Then, do the equivalent for allocating 10,000 objects of size 500 bytes from a pool and freeing them. Then, do the equivalent of allocating 10,000 objects of random sizes in the [1000:0)-byte range on a stack and then free them (in reverse order). Compare the measurements. Do each measurement at least three times to make sure the results are consistent.
16. Formulate 20 coding style rules (don't just copy those in §25.6). Apply them to a program of more than 300 lines that you recently wrote. Write a short (a page or two) comment on the experience of applying those rules. Did you find errors in the code? Did the code get clearer? Did some code get less clear? Now modify the set of rules based on this experience.
17. In §25.4.3–4 we provided a class **Array_ref** claimed to make access to elements of an array simpler and safer. In particular, we claimed to handle inheritance correctly. Try a variety of ways to get a **Rectangle*** into a **vector<Circle*>** using an **Array_ref<Shape*>** but no casts or other operations involving undefined behavior. This ought to be impossible.

Postscript



So, is embedded systems programming basically "bit fiddling"? Not at all, especially if you deliberately try to minimize bit fiddling as a potential problem with correctness. However, somewhere in a system bits and bytes have "to be fiddled"; the question is just where and how. In most systems, the low-level code can and should be localized. Many of the most interesting systems we deal with are embedded, and some of the most interesting and challenging programming tasks are in this field.