Miro Samek

C/C++ Users Journal
Advanced Solutions for Professional Developers

The Embedded Mindset

Innovation in skinning the Heap Cat.

Some $45.7 billion worth of microprocessors, microcontrollers, and DSPs were sold in 2001. Only a tiny fraction of these chips (less than one percent by unit count, actually) were used in "real" computers such as desktops, laptops, servers, or mainframes. The vast majority ended up in embedded-systems—devices generally not perceived as computers. Sure, the multi-gigahertz clock speeds and the mind-boggling complexity of the high-end CPUs can easily steal the spotlight. However, the constantly dropping prices and power consumption of the low-end computer chips transform our way of life in far more profound ways. Since the invention of the first microprocessor (the 4004) three decades ago, Moore's Law has been opening up far more markets and applications for the low-end micros than for the high-end super CPUs. Indeed, our whole society has become vitally dependent on billions of tiny computers embedded in an amazing variety of products. We are way past the no return point in adoption of the technology. For better or worse, embedded systems are here to stay, and their importance is only going to grow in the future.

For the software industry, the proliferation of embedded systems means ever-increasing demand for firmware. Sensing the trend, more and more post-PC-era programmers try to break into embedded systems hoping for higher salaries, more interesting work, and greater job security. But, do desktop-style programmers have what it takes to become successful in the embedded business? This would assume that embedded-systems programming does not require a much different skill set than programming for the desktop. Actually, this is exactly what the aggressive marketing around Windows CE, or more recently around embedded Linux, tries to suggest. (Remember the rhetoric "easy portability of application software and programming skills from the desktop"?) Admittedly, embedded systems span such a broad spectrum that some of them can indeed be successfully programmed with desktop-style techniques. For example, hand-held computers, PDAs, or set-top boxes might approximately fit this category. Generally, however, the less a product's users perceive it as a computer the less applicable (if not out-right harmful) are desktop-style programming practices. In particular, embedded real-time systems, which are the overwhelming majority of all embedded systems shipped worldwide, require a fundamentally different programming approach.

From the dawn of programming, the software community has concentrated mostly on effective strategies to cope with the challenges of general-purpose computing. Perhaps because of this long tradition, the resulting strategies and rules of thumb have become so well established that programmers apply them without giving them a second thought. However, many of these time-honored techniques are not adequate (and often quite harmful) for embedded systems. In fact, embedded programming is so different that it has shaped its own distinct programming mindset. Embedded developers are used to accepting the whole responsibility for their software, but expect to have complete control over all aspects of their creations (including the freedom to shoot themselves in the foot). Consequently, they tend to favor simple, specific solutions, because they cannot take full responsibility for code that they don't understand. (That's partially why embedded folks tend to have an overgrown Not-Invented-Here syndrome.) In contrast, general-purpose programmers are used to relinquishing much of their freedoms in exchange for reduced responsibility for the minute or "boring" aspects of programming. Consequently, desktop developers tend to prefer generic, complex solutions (preferably built into the programming language or the OS) that free them from that responsibility even further. For example, virtual memory emulates a manifold increase in available storage and allows desktop programmers to be quite sloppy in budgeting memory (I mean, having a little better idea than "128 MB of RAM required, 256 MB recommended").

The distinct "embedded mindset" leads to deep differences in the ways embedded and general-purpose programmers design and write code. Even though embedded developers mostly program in C and occasionally in C++, just as their desktop counterparts do, they often intentionally avoid certain established design techniques or language features. I believe that concrete examples of specific C and C++ embedded programming techniques might be of interest to the readers of this magazine, and I hope to bring you some of them in every installment of this column. I would like to kick off with one of the biggest taboos: the use of the heap in embedded systems.

Embedded Real-Time Systems vs. General-Purpose Computers

Embedded real-time systems have two main characteristics:

1. They have a computer buried inside, but the users don't perceive them as computers.
2. They often must respond to external events in a timely fashion, which means that for all practical purposes, a late computation is just as bad as an outright wrong computation.

Vague as it is, this definition can gain the most strength by contrasting real-time embedded systems with general-purpose computers (such as desktop PCs), in which the two main characteristics are either nonexistent or far less important. So, you can read embedded to mean "not for general-purpose computing" and real-time to mean "dedicated to an application with timeliness requirements." Either way, the definition emphasizes that embedded systems pose different challenges and require different programming strategies than general-purpose computers. I strongly disagree with the opinion that embedded real-time developers face all the challenges of "regular" software development plus the complexities inherent in embedded real-time systems. Although each domain has its fair share of difficulties, each also offers unique opportunities for simplification, so embedded-systems programmers specifically do not have to cope with many problems encountered in programming general-purpose computers.

Consider for example the challenges of programming a desktop PC. As far as hardware is concerned, no desktop application can rely on a specific amount of memory available to it or on how many and what kind of disk drives, network cards, graphics adapters, and other peripherals are present and available at the moment. The software environment is even less predictable. Users frequently install and remove applications and application components from all possible sources (remember the Windows DLL Hell?). All the time, users launch, close, or crash their applications — drastically changing the CPU load and availability of memory and other resources. The desktop operating system has the tough job of allocating CPU cycles, memory, and other resources among constantly changing tasks in such a way that each receives a fair share of the resources and no single task can hog the CPU. To succeed in this harsh environment, the desktop OS has no other option but to drastically limit the applications. All applications must strictly comply with a specific API (such as Win32 or a Unix API). Interrupt handling is black magic reserved for device drivers that common mortals (application programmers) better not touch. Fiddling directly with external hardware is prohibited. This scheme is diametrically opposed to the needs of embedded real-time systems, in which a specific task must gain control right now and run until it produces the appropriate output. Fairness isn't part of real-time programming — meeting the dead-lines is. To achieve this, however, embedded software must have full control over the CPU, memory and all the external hardware. Restricted to a desktop-style API, an embedded developer not only loses control that he so badly needs, but must bend back-wards just to flash an LED, let alone to service an interrupt. The increased security of a desktop API in the embedded domain is bogus too. In an embedded system, the specific application code is at least as critical as the generic OS (many embedded systems don't use an OS at all), so a failure in the application renders the system useless regardless of the security mechanisms built into the OS. ❑

A Heap of Problems

General-purpose C and C++ programmers are used to taking heap-based memory allocation for granted. Indeed, most textbook techniques and standard libraries heavily rely on the availability of the heap (or free store, as it is alternatively called). However, if you have been in the embedded real-time software business for a while, you must have learned to be wary of `malloc` and `free` (or their C++ counterparts `new` and `delete`), because embedded real-time systems are particularly intolerant of heap problems. Here are some of the main reasons:

- Dynamically allocating and freeing memory can fragment the heap over time to the point that the program crashes because of an inability to allocate more RAM. The total remaining heap storage might be more than adequate, but no single piece satisfies a specific `malloc` request.
- Heap-based memory management is wasteful. All heap management algorithms must maintain some form of header information for each block allocated. At the very least, this information includes the size of the block. For example, if the header causes a four-byte overhead, then a four-byte allocation requires at least eight bytes, so only 50 percent of the allocated memory is usable to the application. Because of these overheads and the aforementioned fragmentation, determining the minimum size of the heap is difficult. Even if you were to know the worst-case mix of objects simultaneously allocated on the heap (which you typically don't), the required heap storage is much more than a simple sum of the object sizes. As a result, the only practical way to make the heap more reliable is to massively oversize it.
- Both `malloc` and `free` can be (and often are) non-deterministic, meaning that they potentially can take a long (hard to quantify) time to execute, which conflicts squarely with real-time constraints. Although many RTOSs (real-time operating systems) have heap management algorithms with bounded, or even deterministic, performance, they don't necessarily handle multiple small allocations efficiently.

Unfortunately, the list of heap problems doesn't stop there. A new class of problems appears when you use heap in a multithreaded environment. The heap becomes a shared resource and consequently causes all the headaches associated with resource sharing, so the list goes on.

- Both `malloc` and `free` can be (and often are) non-reentrant; that is, they cannot be safely called simultaneously from multiple threads of execution.
- The reentrancy problem can be remedied by protecting malloc, free, realloc, and so on internally with a mutex semaphore (for instance, the VxWorks RTOS from Wind River Systems does just that), which lets only one thread at a time access the shared heap. However, this scheme could cause excessive blocking of threads (especially if memory management is non-deterministic) and can significantly reduce parallelism. Mutexes are also subject to priority inversion. Naturally, the heap management functions protected by a mutex are not available to ISRs (interrupt service routines) because ISRs cannot block.

Finally, all the problems listed previously are in addition to the usual pitfalls associated with dynamic memory allocation. For completeness, I'll mention them here as well.

- If you destroy all pointers to an object and fail to free it, or you simply leave objects lying about well past their useful lifetimes, you create a memory leak. If you leak enough memory, your storage allocation eventually fails.
- Conversely, if you free a heap object but the rest of the program still believes that pointers to the object remain valid, you have created dangling pointers. If you dereference such a dangling pointer to access the recycled object (which by that time might be already allocated to somebody else), your application can crash.
- Most of the heap-related problems are notoriously difficult to test. For example, a brief bout of testing often fails to uncover a storage leak that kills a program after a few hours, or weeks, of operation. Similarly, exceeding a real-time deadline because of non-determinism can show up only when the heap reaches a certain fragmentation pattern. These types of problems are extremely difficult to reproduce.

This is quite a litany, although I didn't even touch the more subtle problems yet, for example, an exception-handling mechanism can cause memory leaks when a thrown exception "bypasses" memory de-allocation. (Here is an idea: why don't you use this list when interviewing for an embedded-systems programmer position? Awareness of heap problems is like a litmus test for a good embedded-systems programmer.) So why use the heap at all? Well, because the heap is so convenient, especially in general-purpose computing. Dynamic memory management perfectly addresses the general problem of not knowing how much memory you'll need in advance. (That's why it's called general-purpose computing.)

However, if you go down the lists again, you will notice that most of the heap problems are much less severe in the desktop environment than they are in embedded applications. For a desktop application, with only "soft" real-time requirements, all issues (except dangling pointers, perhaps) boil down to inefficiencies in RAM and CPU use. The obvious technique that cures many issues is to massively oversize the heap. To this end, all desktop operating systems these days support virtual memory, which is a mechanism that effects a manifold increase in the size of available RAM by spilling less frequently used sections of RAM onto disk. Another interesting approach, brought recently to the forefront by Java (not that it wasn't known before), is automatic garbage collection. You can view garbage collection as a software mechanism to simulate an infinite heap. It addresses the problems of dangling pointers, storage leaks, and heap fragmentation (albeit at a hefty performance price tag). The other issues, especially deterministic execution, remain unsolved and actually aggravated in the presence of garbage collection.

For decades, heap problems have been addressed in desktop environments with ever more powerful hardware. Fast CPUs, cheap dynamic RAM, and massive virtual memory disk buffers can mask heap-management inefficiencies and tolerate memory-hungry applications (some of them leaking like a sieve) long enough to allow them to finish their job. (That's why it's a good idea to reboot your PC every once in awhile.)

Not so in the embedded real-time business! Most embedded systems must run for weeks, months, or years without rebooting, not to mention the small amount of RAM they typically must get by with. Under these circumstances, the numerous problems caused by heap-based memory allocation can easily outweigh the benefits. You should very seriously consider not using the heap, especially in smaller embedded designs.

Even though the free store is definitely not a "free lunch," getting by without it is certainly easier said than done. In C, you will have to rethink implementations that use DLLs, trees, and other dynamic data structures. You'll also have to severely limit your choice of the third-party libraries and legacy code you want to reuse (especially if you borrow code designed for the desktop). In C++, the implications are even more serious because the object-oriented nature of C++ applications results in much more intensive dynamic-memory use than in applications using procedural techniques. For example, a lack of dynamic storage implies that you will lose many benefits of constructors, because you'll often be forced to instantiate objects at times when you don't have enough initialization information.

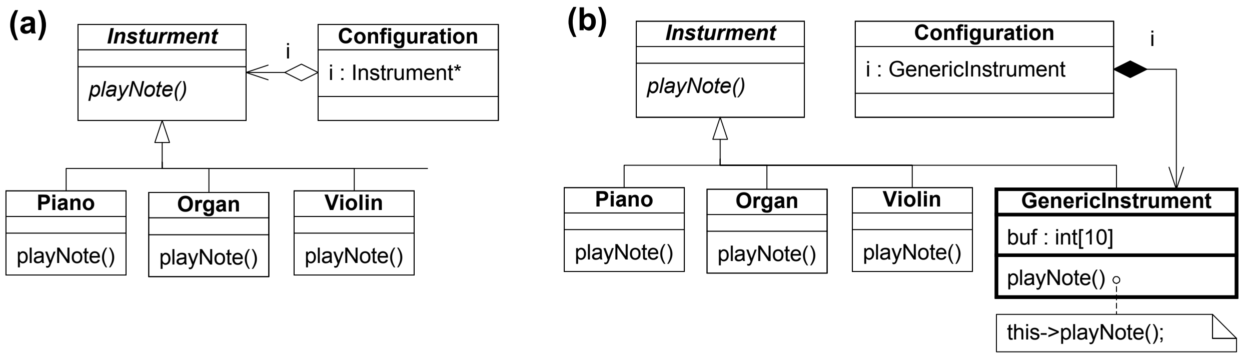

Figure 1 Standard heap-based design for "MySongMaker" keyboard (a), and a solution without the heap (b)

Polymorphism without the Heap?

However, perhaps the biggest problem is that heap-based memory allocation is at the heart of so many basic programming techniques, and that most programmers are trained to use them without giving it a second thought. Consider for example one of the most typical (textbook) use cases of polymorphism. Suppose you were to develop a C++ program for a $9.99 “MySongMaker” toy keyboard (such as my six-year-old daughter uses to torture the family from time to time). “MySongMaker” can imitate many instruments such as the piano, violin, organ, etc., which are selectable at run time by a button press. Figure 1a shows a standard design with an `Instrument` base class defining an abstract operation `playNote` that concrete subclasses (such as `Piano`) override. In this design, selecting the instrument corresponds to instantiating the concrete `Instrument` subclass, and playing the instrument boils down to invoking the virtual `playNote` function for every note (key press). The standard (heap-based) way of coding the design would look approximately as follows:

```
Instrument *i;
...
i =new Piano;       //select instrument Piano
...
i->playNote(key);   //play a note
...
```

Obviously, this snippet of code is very incomplete. At the very least, you must delete the previous instrument before reassigning the pointer `i`, and you must explicitly handle the situation when new fails. (How do you explain to a kid that her keyboard broke because it ran out of heap?) But this is not all. Things get more complicated if you need to store the current instrument as part of another object. (Indeed, “MySongMaker” can record simple “compositions” and thus must store the configuration.) In the standard design, you’d invent a class `Configuration`, which would contain a pointer (or reference) to the `Instrument` class, as shown in Figure 1a. Since the `Configuration` class contains a pointer, the standard rule requires providing a destructor, overloading the assignment operator, and defining the appropriate copy constructor. Although the code for these elements isn’t terribly complicated, the overall price for this little bit of polymorphism seems high. However, perhaps the most disturbing aspect of this traditional design is that once you start using the heap you are stuck for good and you need to keep churning the heap forever (e.g., inside the assignment operator and the copy constructor). Nevertheless, you probably employ such designs every day without thinking twice, because that’s the standard way of implementing polymorphism on the desktop.

An embedded developer, however, would probably think twice before accepting such a design. Figure 1b shows an example of a solution that doesn’t require the heap. As you can see, the Configuration class in this design aggregates by composition the whole instance of a `GenericInstrument` subclass of Instrument, instead of the Instrument pointer, as in the traditional design. The `GenericInstrument` class is never intended to be used as such, but rather just to hold an instance of any other Instrument subclass (such as Piano or Violin). Obviously, `GenericInstrument` must be big enough, which you can achieve by appropriately sizing its attribute `buf[]` (see Listing 1, line 9). While a desktop programmer would probably frown at such hard coding of the upper size limit for all (perhaps unknown yet) subclasses of Instrument , this is a perfectly reasonable thing to do for an embedded programmer. You see, an embedded developer is used to budgeting memory anyway, and the size of `GenericInstrument` provides just this: the worst-case size of any `Instrument` subclass. However, it is always a good idea to assert that the concrete subclass of `Instrument` (such as `Piano` or `Violin`) would indeed fit (see Listing 2, lines 10 and 20). Listing 2 also demonstrates how you can instantiate an arbitrary subclass of `Instrument` inside a `GenericInstrument` object by using placement new (lines 36 and 41). Please note that you need to include the `<new.h>`header file in order to use placement `new`.

```
 1: typedef char Note;
 2:
 3: class Instrument {
 4: public:
 5:   virtual void playNote(Note key) = 0;
```

```
};

class GenericInstrument : public Instrument {
   char buf[40]; // maximum size of an Instrument
public:
   virtual void playNote(Note key) {
      this->playNote(key); // enforce late binding
   }
};

class Configuration {
public:
   //...
   GenericInstrument i;
   //...
};
```

Listing 1 `instrument.h` declaring abstract class `Instrument`, `GenericInstrument` subclass, and `Configuration` class.

```
#include "instrument.h"
#include <new.h>  // for placement new
#include <iostream.h>
#include <assert.h>

class Piano : public Instrument {
   int attr; // an attribute
public:
   Piano() {
      assert(sizeof(*this) <= sizeof(GenericInstrument));
   }
   virtual void playNote(Note key);
};

class Violin : public Instrument {
   int attr1; // an attribute
   int attr2; // another attribute
public:
   Violin() {
      assert(sizeof(*this) <= sizeof(GenericInstrument));
   }
   virtual void playNote(Note key);
};

void Piano::playNote(Note key) {
    cout << "Piano " << key << endl;
}

void Violin::playNote(Note key) {
    cout << "Violin " << key << endl;
}

static Configuration c1, c2;

int main() {
   new(&c1.i) Piano; // instantiate Piano
   c1.i.playNote('c');
38:    memcpy(&c2, &c1, sizeof(Configuration));
39:    static_cast<Instrument *>(&c2.i)->playNote('d');
40:
41:    new(&c1.i) Violin; // instantiate Violin
42:    c1.i.playNote('e');
43:    c1 = c2; // Oops, doesn’t copy the v-pointer!
44:    c1.i.playNote('f');
45:
46:    return 0;
47: }
```

Listing 2 Test case for `GenericInstrument` and `Configuration` classes.

Another point of interest in this design is arranging for late binding while invoking virtual method(s) defined in the `Instrument` superclass, such as `playNote` . Because now we use an instance of the `GenericInstrument` class (rather than a pointer to the `Instrument` class, as in the standard design), a C++ compiler would synthesize a regular (early binding) call. One way of enforcing the late binding would be to replace every such call (say, `i.playNote`) with a call via a pointer, which then needs to be explicitly upcast to `Instrument*`, as demonstrated in line 39 of Listing 2. However, such an approach is ugly and requires clients to remember which methods are virtual (and require late binding) and which are not. A better way is to inline a pointer-based call inside the `GenericInstrument` class, as shown in Listing 1, line 12. At first, the `GenericInstrument::playNote` implementation might look like a classic example of endless recursion. However, as stated earlier, the `GenericInstrument` subclass should never hold an instance of itself (in other words, its virtual pointer will never point to `GenericInstrument` ’s virtual table), so the call always resolves through a virtual table of some other Instrument subclass. `GenericInstrument::playNote` is defined inline, so the indirect virtual call via this method should be optimized away. Finally, I would like to point out a potential pitfall of this approach. You might have noticed that in line 38 of Listing 2 the `Configuration` object is copied byte-for-byte using `memcpy`, rather than the assignment operator, as in line 43. The reason for avoiding the default assignment operator is that it does not copy the virtual pointer. (So in spite that `c2.i` is always a Piano , the method invoked in line 44 is `Violin::playNote`.) Obviously, you could remedy the problem by explicitly overloading the assignment operator in class `GenericInstrument`, for instance, as follows:

```
GenericInstrument &operator=(const GenericInstrument &x){
   if (&x !=this)
      memcpy(this, &x, sizeof(*this));
   return *this;
}
```

I hope you find this little technique useful, or at least thought provoking. However, I don’t want to pretend that the design fits well within the C++ fabric. In fact, it must fight the C++ compiler in too many places, because polymorphism in C++ seems to be really designed with heap-allocated objects in mind. And this is exactly the problem I wanted to illustrate: using C or especially C++ in an embedded domain requires sometimes going off the beaten path, because the vast majority of the coding techniques have been designed for programming general-purpose computers. This doesn’t mean, however, that in embedded systems you must abandon such powerful techniques as polymorphism. The main point is that you usually can find a suitable workaround, but this requires an embedded-system-specific attitude to programming — and that’s what I mean by the “embedded mindset.”

End-around Check

If you are not already subscribing to Embedded Systems Programming, you should. You can probably qualify for a free subscription — check out <www.embedded.com/subscribe.htm>. Jim Turley’s Embedded Systems Programming articles (“Embedded Processors by the Numbers,” May 1999, <www.embedded.com/-1999/9905/9905turley.htm>, and “The Death of ASICs,” November 2002, <www.embedded.com/story/-

OEG20021028S0039>) offer an interesting perspective on the embedded microprocessor market. The Semiconductor Industry Association website (<www.sia.org>) publishes a free breakdown of worldwide semiconductor market by products. If you are interested in more detailed business reports, you can buy them from the MicroDesign Resources website at <www.microdesign.com>.

The 4004 ur-microprocessor recently celebrated its 30th birthday. It all started in 1971, when Busicom, a Japanese company, wanted a chip for a new calculator. With incredible overkill, Intel built the world's first general-purpose microprocessor. Then it bought back the rights for $60,000. The four-bit 4004 ran at 108 kHz and contained 2,300 transistors. Its speed is estimated at 0.06 MIPS (<www.dotpoint.com/xnumber/-intel_4004.htm>).

Evans Data Corporation (<www.evansdata.com>) sells the very comprehensive "Embedded Systems Developer Survey 2002," which provides information on demographics of the embedded community, salaries, what programming languages are in vogue (C and C++ clearly lead the pack), what RTOSs dominate, and much more. Speaking of embedded marketplace surveys, though, in "Surveys Lie" Jack Ganssle advises to take them with the grain of salt (<www.embedded.com/story/OEG20020715S0058>).

Once upon a time, computer memory was one of the most expensive commodities on earth, and large amounts of human ingenuity were spent to invent memory-efficient programming techniques. These days, the developers who know how to write memory-efficient code are rapidly becoming an endangered species. However, as James Noble and Charles Weir point out in their book Small Memory Software: Patterns for Systems with Limited Memory (Addison-Wesley, 2000), a more thoughtful usage of memory has many unexpected benefits that go far beyond the dollar cost of RAM chips. Although aimed at a general-purpose programmer, this book is especially relevant to an embedded developer.

Priority inversion is the deadly condition in which a low-priority thread blocks a ready and willing high-priority thread indefinitely. Priority inversion can be a show stopper in any real-time system; yet, most less sophisticated RTOSs don't support any mechanisms to prevent it. Even the high-end systems, which can detect and remedy the condition (by applying priority-inheritance or priority-ceiling protocols), don't enable these mechanisms by default because of the high overhead involved. A highly publicized example of a system failure caused by priority inversion is the Mars Pathfinder mission in July 1997. The mission (eventually very successful) was saved by remotely enabling priority inheritance in the VxWorks mutex implementation that was originally not activated because of its high overhead (e.g., see <www.windriver.com/customer/html/-jpl.html>).

Obviously, this little technique of changing an object's class at run time without using the heap described earlier is just one way of skinning the cat. In the source code accompanying this column (available for download at <www.cuj.com/code>, you can also find an alternative design, for which I'd like to thank Jeff Claar. Jeff's method replaces inheritance with object composition, so `GenericInstrument` does not inherit from `Instrument` , but rather contains a concrete `Instrument` subclass. Further, overloading the `operator->` in `GenericInstrument` enables convenient and intuitive access to virtual functions of the Instrument superclass. Jeff's technique seems to blend much better with C++, at only a tiny performance cost. ❑

Miro Samek is the author of *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, CMP Books, 2002. He is the lead software architect at IntegriNautics Corporation (Menlo Park, CA) and a consultant to industry. Miro previously worked at GE Medical Systems, where he has developed safety-critical, real-time software for diagnostic imaging X-ray machines. He earned his Ph. D. in nuclear physics at GSI (Darmstadt, Germany) where he conducted heavy-ion experiments. Miro welcomes feedback and can be reached at `miro@quantum-leaps.com`.