Conversations With a Guru

Herb Sutter & Jim Hyslop

Contents

1 Remembering auto_ptr

— A class with an auto_ptr member: How can copying go wrong? Let us count the ways...

I had met Jeannine only the day before, in the crew rotation staging area now far below us. "I'll always remember my first job," I told her after the flight attendant walked past, checking our belts.

"What about it?"

"The senior programmer on my project," I smiled, remembering. "She was an odd duck. We called her the Guru. Management didn't like to assign new programmers to her team; I was the only one of four hires that year to last through the probation period."

Jeannine tilted her head and was about to ask a question when the last chime sounded and the rumbling acceleration took hold, cutting off conversation for a few minutes. At the end of the burn, as we left orbit, I told her the story of my second day at work.

- - - - -

We were programming in early C++. On the afternoon of my second day, tired of reading employee handbooks, I wrote a utility class that contained a raw pointer as one of its members:

```
#include "xStruct.h" // definition of struct X

class xWrapper
{
  X* xItem;
public:
  xWrapper() : xItem(new X) { }
  ~xWrapper() { delete xItem; }
  void dump() { /* dumps xItem to cout */ }
};
```

Of course, the program using this class kept crashing intermittently with memory corruption, because I'd violated the Law of the Big Three: Whenever you provide any one of a destructor, copy constructor or assignment operator, you will generally need to provide all three[1]. "So," said I to myself, said I, "I have to handle copying and assignment. Simple... auto_ptr already has a copy constructor and assignment operator, so I'll use that." (You've heard of the auto_ptr in the original C++ library, right?)

[1] M. Cline, G. Lomow, and M. Girou. C++ FAQs, 2nd ed. (Addison-Wesley, 1999).

Since the auto_ptr automatically deletes the object it points to, all I had to do was change the type of xItem and remove the delete statement in the destructor – auto_ptr would take care of the rest, right?

```
class xWrapper
{
  auto_ptr<X> xItem;
public:
  xWrapper() : xItem(new X) { }
  void dump() { /* dumps xItem to cout */ }
};
```

Unfortunately, my program was still crashing, this time because it was trying to dereference a null pointer. I had been trying to puzzle out the problem for about half an hour when the Guru, thin as a rail, decided to walk by, carrying a thick book open in one hand. She did that a lot. . . show up at propitious times, I mean; I think it was some sort of prescient thing. Downright spooky, actually.

"Uh, what're you reading?" I asked her, pointing at her book to deflect attention away from my screen and hoping she would go away.

The Guru blinked. "The writings of Josuttis," she said softly, marked her page, and closed the book. "What's that you have there, young one?"

"I'm having problems with this wrapper class I'm writing," I admitted. "I'm using an auto_ptr member, but in my test harness its pointer keeps getting reset to null for some reason."

"Show me your writings," the Guru said. I showed her the screen. "Ownership," she said immediately, after barely a glance.

It was my turn to blink.

"Ownership, child; your problem is ownership semantics. No person can serve two masters, and no pointer can serve two auto_ptrs."

Her words, although certifiably strange, made me realize my mistake. "Okay, right," I nodded. "When you copy an auto_ptr, the original one relinquishes ownership and gets reset to null. My xWrapper copy constructor reuses that default behavior, so the original xWrapper object's auto_ptr is being reset, and when I try to access it I'm dereferencing a null pointer."

"Correct," the Guru agreed. "You have done well to reuse the true tools of the Standard, but you must take care with them. For xWrapper, you must still manage xWrapper copying and assignment yourself."

"But I can't implement them in terms of auto_ptr's own versions because those won't do the ri— Oh. I get it. I'll use auto_ptr's dereference operator to access the owned objects." I quickly wrote out the two functions:

```
xWrapper::xWrapper(const xWrapper& other)
  : xItem(new X(*other.xItem))
{ }
```

```
xWrapper\& xWrapper::operator=(const xWrapper &other)
{
  *xItem = *other.xItem;
}
```

"Hey, cool." This, I liked. "I don't even need to check for self-assignment in the assignment operator."

"That is correct."

I should have stopped then and kept my mouth shut, but I wasn't that smart yet. "auto_ptr sure is easy to misuse. If only it could have told me I was trying to transfer ownership when I didn't expect that to happen. . ."

"Peace!" interrupted the Guru. "The fault is not with auto_ptr in this instance. You should have said that you did not want the auto_ptr to be copied, had that been your desire."

"But how? That's not possible."

"Ah, but it is. Remember the blessings of const-correctness. The way to state that an auto_ptr is immutable is to make it const. Had you made the member a const auto_ptr, the compiler would not have been able to silently perform the copy of the xWrapper object. Alternatively, had you used something like the strict_auto_ptr from the Second Revisionist rendering of Cline 30:12, the compiler would not have been able to mistakenly generate incorrect xWrapper copying and assignment. Of course, in this case making it const would have been simpler and sufficient."[1]

She reopened her copy of Josuttis and resumed reading as she started to walk away, still talking to me absently, both she and her voice gradually drifting away: "A word of caution, my child. . . auto_ptr is a useful tool, but as you have discovered, it is not a panacea. Read and meditate upon Josuttis chapter 4[2]. You must never instantiate a Standard container of auto_ptr, such as vector<auto_ptr>, because auto_ptr does not meet the copying and assignment requirements of the Standard. Furthermore, never attempt to use auto_ptr to point to an array of objects, for the auto_ptr's destructor uses non-array delete to delete the object it owns; for an array of objects, use a vector instead. The library. . ."

But then she turned a corner, and was gone. It was only my second day; I wondered, not idly, if I ought to update my résumé while I could still pretend this job had never happened.

- - - - -

"Weird lizard," Jeannine opined, sipping coffee as we exited the Terran local traffic control area, still gaining velocity. "So, did you leave?"

[1] ibid., FAQ 30.12, pages 426-8.

[2] N. Josuttis, The C++ Standard Library (Addison-Wesley, 1999).

"She was, but no. I'm not sure why," I added honestly. "A few run-ins like that, and I was ready to leave during probation like the rest. I guess he grew on me, though. Didn't you ever work with a quirk like that?"

"Mmm. Some. I suppose."

It was not the last time I would speak with Jeannine about the Guru, or about more pleasant things.

2 Null References

— Can references be null? Case in point: A factory that returns a pointer to the newly created object.

Aboard ship, the last thing you want to feel is wind. Jeannine and I happened to be unlucky enough to be the closest to the incident; straining together, we managed to wrestle the bulky door shut and seal it, isolating the breached compartment. As we leaned against the door, breathing deeply, the klaxons stopped sounding.

"What in tarnation was that?" the watch officer's voice demanded shrilly over the comlink.

"Just a small penetration, sir," Jeannine responded. "We sealed off the compartment. All clear, no casualties."

"You sealed off the compartment? Why didn't it seal automatically? Didn't we just refit that door?"

Jeannine and I glanced at each other. "Uh, sir, it seems that the maintenance crews assumed the new locking mechanism would be fine, because it was brand new from the factory, so they didn't check it. It was fine, only they didn't take off all the packing materials and it couldn't seal by itself."

There was a pause, then: "All right, good work. There's a damage crew on the way." Then in the background, just before the comlink was broken, we heard the XO's voice say: "Mr. Johnson, have the chief of engineering see me in my cabin. Now. Tell Reilly to–"

I grinned at Jeannine. "I'd give a week's pay to hear that conversation. This reminds me of a code safety problem we had once, a long time ago. . . "

- - - - -

I had been on my first programming job for a couple of weeks, and I'd met the other programmers on my project. One of them was Bob. He was the antithesis of the Guru in almost every way: Bob was smug, his code was difficult for others to maintain, and he had a tendency to violate programming rules.

I was well into my first project, and Bob had been assigned to perform a code review on some stuff I'd checked in. He came by my desk, holding a latte, and leaned against the partition. "Your code is crashing with an access violation inside your helper function," he said. "You better fix it before the Space Cadet hears about it."

"The Guru really is an excellent programmer, Bob," I bristled. True, the Guru was odd, and I was still uneasy around her. Every time I talked to her, I kept thinking about updating my rsum. But Bob's attitude just rubbed me the wrong way somehow.

"Yeah, whatever," Bob waved his cup dismissively, sloshing a little. "Anyway, your helper function is using a null. Let's take a look."

"But the helper function doesn't use any pointers," I frowned. "Just a reference to an xWrapper class." I called up the code in my text editor. It was something like this:

```
class xWrapper
{
  /* ... */
public:
  virtual void dump() { cout << name << endl; }
};
void helper( xWrapper& w )
{
  w.dump();
  // ... do other stuff with w ...
}
```

Bob leaned at the monitor. "Yeah, it's crashing at that dump() statement," he said, scratching his nose.

"That's not possible."

"Sure it is, Junior." Bob was really starting to bug me. "All I do is get a brand new pointer from the factory, and dereference it when I pass it to you. If that pointer's null, 'course, you have a null reference. You should check for that."

"Uh, well," I faltered, less certain now. After all, I was just fresh out of school, and Bob had several years of experience. "I didn't think that was possible to see if a reference was null."

"All you do," Bob said, "is check the address of the reference. Like I said, you want to be safe, you should check for that. . . "

"No," the Guru's quiet voice startled us both. Once again, she had appeared at the right moment, and now stood behind us, an open tome in hand. "There is no need of such an abomination. The Standard teaches that it is not possible to have null references."

"But it is possible, dearie," Bob insisted. "We've just shown that. That's his problem."

I wanted to smack him. The Guru merely favored him with a chill glare. "No. You should check for that. Never dereference a null pointer. Change your code: If the pointer is null, do not call helper(). Show me the working code this afternoon. Begone, you instrument of malformed programs," she waved him away.

Bob blinked. But she was the senior programmer on the team, so he shut up and left.

"My apprentice," she continued to me alone, "one of the major reasons to use a reference instead of a pointer is to free you from the burden of having to test to see if it refers to a valid object. The only way to create a null reference would be to dereference a null pointer – an act clearly forbidden by the Holy Standard, well inside the realm of Undefined Behavior."

"Okay, but why tell only me? Why not tell Bob?"

She shook her head sadly. "This he knows. I cannot teach him; he is drawn to Undefined Behavior. My apprentice, beware the path to Undefined Behavior. Once you start down that path, it will dominate your life forever. It will consume your time, as you try to track down and fix problems."

I considered this. "Well," I asked, "what about accidentally creating an invalid reference? Say you pass an object to a function, and the object you pass goes out of scope before the function is finished with it. How do you guard against that?"

She shook her head. "You cannot. In this respect, references and pointers are very similar. Consider this parable." The Guru quickly scribbled out some code on my whiteboard:

```
T* f()
{
  T t;
  return &t;  // return a dangling pointer
}
void f1( T* t )
{
  if( t )
    t->doSomething();
}

int main()
{
  T *tPtr = f();
  f1( tPtr );   // evil
}
```

"This f() quite clearly lives in infamy," she said, pushing a graying lock behind one ear, "as it returns a pointer to a local object that oversoon goes out of scope. Thus tPtr in main() points to an object that has been destroyed. f1() does the best it can – it checks for a null pointer. Some platforms also provide compiler-specific functions that can test whether a pointer points to a valid region of memory. However, such functions would still not detect the case that, inside f1(), the variable t does not point to a valid T object. It is impossible for f1() to be certain that t is valid; therefore as the programmer writing f1(), you must have faith. Faith that your fellow programmers will do everything in their power

to provide you with a pointer to a valid object."

"Just as you must assume that a non-null pointer is valid, you must assume that a reference is valid. You must have faith in your fellow programmers."

"Even Bob?" I asked.

She nodded with an air of sadness, then drifted quietly away down the corridor, reading her tome.

- - - - -

"Did Bob last long?" Jeannine asked.

"Longer than he should have." I stopped smiling. "Our company also built software that got used in pacemakers. You know, the devices they used to put inside people's chests. One time he checked in a piece of code that didn't verify all its preconditions, and the code got out into production. And..."

Jeannine lifted an eyebrow.

I nodded soberly. "Bob finally did get another job, in retail sales. He can be glad it happened before the programmer liability laws." Then the damage control team relieved us, and we got ready to go on watch.

It was not the last time I would speak with Jeannine about the Guru, or about more pleasant things.

3 Genesis

— How we got here from there: A colorfully rendered history of the C family of languages.

I burst into the compartment. "Hey, have you heard the rumors?"

Jeannine looked up from her work. "Hay is for horses. What rumors?"

"About Ganymede," I explained. "I hear they found something under the ice. No sign of recent melting, either; it's something old, and definitely not natural."

That got her attention. She blinked and stretched. "Really? And not ours?"

"That's the buzz. Your neurons tingling yet?"

"Sure, sure. Wow. What else?"

"No more information yet," I admitted, letting myself drop down onto the couch. "And it's just a rumor. But, man, wouldn't that be exciting, if something big happened on our tour? We'll be there on station in less than two months."

"In the ice. . . ?" Jeannine considered this. "Whether it's an artifact or not, it must be ancient. Makes you think about all the big questions, doesn't it? Our own race's history, where we came from, where we're going."

"Roots and directions, right. Y'know," I said, "this reminds me of something that happened back when I was on my first job. . . "

- - - - -

It was a slow day, and I was talking with Wendy. She was the girl next door. . . literally, I mean. Her cubicle was next to mine.

"Y'know," I said, "I heard that Stroustrup chose the name C++ because it was an incremental change to the C language – C incremented, right?"

"Right," Wendy said. "But if you think about the semantics of the name, what does it mean?"

I puzzled for a moment, then shook my head. "I dunno."

"Well, it uses the post-increment, so it means 'take the C language, increment it, and use the original.' " We both laughed.

"A common jest." The Guru's voice startled us both. We turned, and I tensed, but then I saw an unusual thing: A twinkle in the Guru's blue eyes. She added: "But how well do you know the history of the language you are abusing?"

I relaxed again, quickly at ease. "How do you mean?"

The Guru closed the book in her hand – I recognized the cover of the D&E[1]. She raised her head slightly, her eyes focusing off in the distance. Then she began to recite, in almost a sing-song way, and her thin body seemed to resonate with the chanted words:

[1] B. Stroustrup, The Design and Evolution of C++ (Addison-Wesley, 1994) ISBN 0201543303.

"In the beginning, computer languages were in chaos, and high-level computer languages were void, and without form. And from his desk at Bell Labs, Richard Martin did look upon the computer languages, and saw that a high-level language was needed. And he went forth, and developed BCPL, and he saw that it was good.

"And Ken Thompson did look upon BCPL, and saw that it was good, but it had to be smaller to run on a PDP-7. And Thompson went forth, and did produce a new language, and he did call it B, which is a contraction of BCPL[1]. And he saw that it was good.

"But B was without types. And Dennis Ritchie saw that it was without types, and he too went forth, and extended the B language. Thus Ritchie added structs, and added types to the language, and he called it C, for C is the next letter after B, both in the alphabet and in 'BCPL.' And Ritchie saw that it was almost good, but he was not satisfied, and he continued to toil in the sweat of his brow and to refine the language. And in 1978 did Brian Kernighan and Dennis Ritchie proclaim 'The C Programming Language[2].' And there was much rejoicing among the multitudes, and they saw that C was good. 'Yea, verily, this language is good,' proclaimed the multitudes.

"And it came to pass that the multitudes began to use C. And new features were added, but not by all compiler vendors. And the multitudes were dismayed, and cried out. 'Yea, verily, we need a Standard C,' they did cry. And ANSI did hear their cry. And in 1989 did ANSI say, 'Behold, I bring you good tidings of great joy, which shall be unto all programmers. For unto you this day is adopted a Standard for C, which is X3.159-1989.' And then did ISO take up the cry, and did proclaim ISO/IEC 9899-1990. And there was much rejoicing among the multitudes.

"Now it also came to pass, before the C Standard was adopted, that Bjarne Stroustrup did also labor to perfect the C language. And Stroustrup labored to add Classes to C, and function argument type checking, and other features possessed of niftiness. And so he went forth, and in 1980 did he proclaim 'C With Classes.' And there was much rejoicing and excitement among the multitudes.

"But Stroustrup was not satisfied. And he made further changes to

[1] see http://cm.bell-labs.com/cm/cs/who/dmr/chist.html for alternate etymologies of "B".

[2] B. Kernighan and D. Ritchie, The C Programming Language, 2nd edition (Prentice Hall, 1998) ISBN 0131103709.

the language, and he did call it C++, which is to say, the C language incremented. And so he went forth, and in 1986 proclaimed 'The C++ Programming Language,' and there was much rejoicing among the multitudes[1].

"But as with all living things, the C++ language evolved. Templates, exception handling, and other features were added to the language. And there was rejoicing among the multitudes.

"But also did the multitudes complain. For in those days, did various compiler vendors implement different solutions to templates and exceptions, and other features. Even did some compiler vendors refuse to implement them. And so the ISO went forth, and in 1998 – which is to say, the sixth year of Clinton, while Chrtien was prime minister of all Canada with the possible exception of Qubec, and Lewinsky was celebrity of choice in the media, for Simpson was no more and there was no news that year – in the ninth month, on the first day of the month, did ISO proclaim, 'Behold, I bring you good tidings of great joy, which shall be unto all programmers. For unto you this day is adopted a Standard for C++, which is ISO/IEC 14882:1998(E).' And ANSI did take up the cry, and proclaim the very same Standard, on the seventh month, on the twenty-seventh day of the month, which was technically before the day on which ISO did proclaim it because that's just how these things work out sometimes. And there was much rejoicing among the multitudes. 'Yea, verily, now maybe we shall get some work done,' the multitudes did proclaim.

"Now it came to pass, during those days, that Patrick Naughton did work for Sun Microsystems. And Naughton was sore tried, and sought to leave Sun. But the Company counter-offered, and did say unto him, 'Thou shalt have a team of developers at thy disposal, if thou wilt but give us something cool.' And so was born the team known as Green.

"And Green team did seek solitude, and went into exile in the wilderness. And they did seek an Object Oriented Language to be used in embedded devices, and they did start to modify C++. But C++ was too cumbersome for their needs, and so did they create a new language based on C++, and they called it Oak, for James Gosling did look out his window and see an oak tree. And the team looked upon Oak, and saw that it was good.

[1] B. Stroustrup, The C++ Programming Language, Special Edition (Addison-Wesley, 2000) ISBN 0201700735.

> "Now in those days it also came to pass that the National Center for Supercomputing Applications did beget Mosaic, and thus did they bring forth the World Wide Web. And there was much rejoicing among the multitudes. And then did Bill Joy seek to give away the source code to Oak, and to make Oak useable in web browsers. And Sun looked upon the idea, and saw that it was good. But the trademark Oak was already taken, and therefore did Sun call the new language Java. And so it came to pass that Sun did proclaim the Java Programming Language. And there was much rejoicing among the multitudes. 'Yea, verily, now we have another truly platform-independent language,' they did proclaim, 'we think, and it does really cool stuff.' "

The Guru finished her recitation, and focused on me. "And that, my young apprentice, is the story of the C family of languages." She pushed a gray lock behind her ear, stood still for a moment in a respectful silence, then bowed her head and silently walked away, reopening her D&E.

I sat there, stunned, and turned to Wendy.

"Hey, don't look at me," Wendy shrugged. "You'll get used to her. She really is the best programmer I've ever worked with."

I barely heard her. I was trying to remember where I'd put the floppy disk with my résumé.

"What did you spend more time at – doing work, or thinking about how to get away from her?" Jeannine asked, smiling facetiously.

"Yes," I smirked back. It was not the last time I would speak with Jeannine about the Guru, or about more pleasant things.

4 So Who's the Portable Coder?

–Writing portable code. Case in point: Scope of variables declared in a for-init-statement.

"Hi, buster." Jeannine came over and sat down across from me in the mess. "Hear any more rumors?"

I glanced at the time just to make sure, but I wasn't due to go on for another half hour. "Hi, busted. No, not yet. They're keeping a lid on it, whatever it is. You're off watch early?"

"Brzenkowski couldn't sleep, so he relieved me. I owe him one."

"Thoughtful of him," I agreed around another mouthful of lunch. "I really like him. He always does good work, makes sure it won't affect other people, understands how he fits into the big picture. I wish we could keep working with him after planetfall next month. Everyone should be that professional."

"Most people are," Jeannine smiled. "You're no slouch yourself."

"Actually," I chuckled, "it reminds me of Bob. You know, I've told you about him before, the other weird from my first job. Man, sometimes he'd do things that'd just make you fry a lobe. It got to the point where if someone did something dumb we'd say he 'pulled a Bob.' "

I proceeded to explain...

- - - - -

A voice beside me said: "Hey, Junior, you broke the build."

I sighed silently and closed my eyes. I'd been on the job nearly two months and already decided I hated the Guru's calling me "my apprentice" less than anything this rotten programmer, Bob, might call me – especially "Junior." I counted to ten, then answered without turning: "What are you talking about, Bob?"

"Your latest code checkin," Bob responded amiably, and sipped at his latté. "My compiler barfs on it. It complains that an iterator is already defined. You broke the build, man. Better fix it before Her Holiness finds out." I winced; as inexperienced as I was, I knew that breaking the build was A Bad Thing. I could just imagine the Guru's reaction: Remember, my young apprentice, the second commandment of team membership: Thou shalt not break the build.

"Okay, fine," I sighed again, audibly this time. "What part of the code are you talking about?"

Bob leaned over, hit a few keys, and pulled up a function that looked something like this:

```
// OBJMAP is a typedef for a standard container
void f( OBJMAP &theMap )
{
    for( OBJMAP::iterator iter = theMap.begin();
         iter != theMap.end();
         ++iter )
    {
        // do something
    }
    // other code in here
    for( OBJMAP::iterator iter = theMap.begin();
         iter != theMap.end();
         ++iter )
    {
        // do something else
    }
}
```

"What's wrong with that?" I asked. "I checked it out in Stroustrup, and the iterator should go out of scope at the end of each loop [1]." "Yeah, well, not with my compiler." Bob replied, picking at his teeth. "It complains that iter is already defined. Easy fix, though. . . just get rid of the second declaration of iter. In fact, that was how my code originally read when I checked it in. Then you went and broke the build."

Then I understood. I had to be diplomatic because I was still on probation, though. "Bob," I said reasonably, "this has to compile on all the platforms we're targeting. I put the second declaration in because my compiler complained about the second iterator being undefined, and that's how it's supposed to work, according to Stroustrup." Bob seemed a little put off. "I don't care about your newfangled compiler. I've been using mine for years and it's never let me down. Your code doesn't work." "Bob. The code's right. It compiles fine for me." "So you got a buggy compiler. Not my fault, I'm just doing my work, I don't care about other compilers," Bob shrugged again, more defensively.

A quiet voice behind us said: "I sense much fear in you." I jumped a little; Bob jumped more, sloshing his latté. The Guru was far too good at gliding up silently behind us, a tome in one hand as she always had. The Guru shook her greying head slowly, frowning. "Fear leads to anger," she continued. "Aw, cut it out. I'm not afraid of other compilers!" Bob said hotly, sucking the spilled latté from his wrist. "I don't care about them at all." "Anger leads to hate. Hate leads to suffering." "I'm suffering, all right," I muttered under my breath. "I'm just trying to write portable code. —All right, Bob," I capitulated, anxious to be

[1] Bjarne Stroustrup. The C++ Programming Language, 3rd Edition (Addison-Wesley, 1997).

rid of him, "I'll fix it so it works. Okay?" "Like I said, Junior, just get rid of the second declaration." He glared once more, then turned and left. Wendy gophered up from the next cubicle, watching him go. "So what's Bab complaining about this time?" she asked us. While the Guru stood nearby, having resumed reading in her tome, I showed Wendy the code and explained what Bob wanted to do. "Typical," Wendy snorted. "Give him a choice between implementing a portable solution, and a solution geared towards his compiler, he'll unerringly choose the latter. His code is not portable, just like you said. Y'know, in this case, you don't really need to declare a new iterator for each loop anyway. Just declare it once, before the first loop, and it'll work on both compilers." She made some quick changes to the code

```
// OBJMAP is a typedef for a standard container
void f(OBJMAP &theMap)
{
    OBJMAP::iterator iter;
    for( iter = theMap.begin();
         iter != theMap.end();
         ++iter ) {
        // do something
    }
    // other code in here
    for( iter = theMap.begin();
         iter != theMap.end();
         ++iter ) {
    // do something else
    }
}
```

The Guru looked up from her reading and glanced at the solution. "Very wise, daughter." Wendy raised an eyebrow, but only a little; she was used to it. "And, as you implied, that is not the only solution. Another elegant one might be to avoid the iterator declarations entirely, and perhaps improve readability, by using for_each and expression templates, where possible [1]. My apprentice," the Guru continued to me, "while it is important to know, understand, and obey the Holy Standard, it is also important to know your tools, even those that are Deviant. Bob is using a compiler that does not obey the scope of a variable declared in a for-init-statement. He cares not about portable code, as long as his own compiles."

"Hmm..." I said, rubbing my chin. "Writing portable code is harder than I thought. So, we have to wait for a new release of Bob's compiler before we can take advantage of the code I originally had?"

The Guru looked off in the distance for a moment. "I am not a prophet, I

[1] See the feature article on expression templates in the May 2000 issue of C++ Report.

cannot see the future. However, this I do foresee: Portability issues will never truly disappear. Compilers will not all become Standards-conforming at the same rate. Some vendors may even leave outdated behavior in their compilers, in order not to break legacy code.

"Also, some compilers will introduce new features to extend the Standard. In and of itself, this is good, else would the language become stale, inflexible, and eventually die stagnant. Programs destined for only one platform may benefit from the new features. Using those non-Standard features in portable code, however, would be disastrous – and some novitiates, such as yourself, may inadvertently use them, without being aware of their non-portability. Do not come to love one compiler so much as to fear others. Fear is the path to the unportable."

She paused a moment, then turned and walked off. As she left, we could hear her soft voice saying, "I wish I could remember the name of the wise one who said, 'The difference between theory and practice is greater in practice than it is in theory . . . ' "

- - - - -

"Bob just wasn't a big-picture fellow," I finished.

"Nobody had better pull a Bob on Ganymede," Jeannine agreed. "It's not quite that forgiving a place, just yet. Hey, you'd better go. Anderson's waiting."

I looked at the time and stood up. "Duty calls. What are you and Laura doing after third watch?" I asked, since I was discovering that there were, of course, more pleasant things to talk about than Bob, or even the Guru . . .

5 Virtually Yours

–An application of the Template Method pattern, and a discussion of when to use public vs. protected vs. private virtual functions.

I gave the wrench another twist, but it was useless. I could have tried pounding the wrench with the hammer to try to force it over, but that didn't seem like a good idea. "It won't budge," I said. "There's almost enough room for the new module, but what's in there already is all just wedged in too tight."

"Kee-ristopher," Jeannine complained, watching because there was only room for one of us to work on the machine. "It can't be that hard to extend the hardware."

"It sure can be when it's not put together right," I sighed. "Whoever slapped this together must've figured that as long as it worked it was good enough. Everything's been put together too tightly coupled. We can extend it, all right, but first we'll have to take it all the way apart and rebuild it."

"Welcome to Europa, the pull-your-own-weight capital of the system. Back home we'd have techs doing this for us."

I mopped my brow. "You know," I said thoughtfully, "this reminds me..." Jeannine smiled; that was the signal for a caffeine break, and I began to tell her another story from my first job.

- - - - -

I was working on some code that Wendy, the programmer in the next cubicle, had designed. She had created an abstract base class for the project, and my job was to derive a concrete class from it.

As usual when studying a new class, I began with the public interface:

```
class Mountie
{
public:
    void read( std::istream & );
    void write( std::ostream & ) const;
    virtual ~Mountie();
}
```

No surprises here a virtual destructor, implying that this is intended to be used as a base class. Non-virtual read and write members, though I wanted to look into that one. Then I moved on to the protected interface:

```
protected:
    virtual void do_read( std::istream & );
    virtual void do_write( std::ostream & ) const;
```

After a moment, I realized that Wendy was using the Template Method pattern: the public, non-virtual functions clearly would invoke the protected virtual functions [1]. Pleased with my own quick understanding, I was feeling pretty smug by this point ready to handle anything that would be thrown at me. Until I looked at the private interface:

```
private:
    virtual std::string classID() const = 0;
```

I stopped cold. A pure virtual, private function? How could it possibly work? I gophered up. "Um, Wendy," I said, "your **Mountie** class can't work. It's got a private virtual function." "Have you tried it?" Wendy asked, without looking up from her keyboard. "Uh, well, no," I admitted. "But my derived class can't possibly override the **classID** function." "So certain are you?" The Guru's quiet voice startled us both. "Always with you it cannot be done. Have you learned nothing from me in these months you have been my apprentice?" Despite the calm quiet of the Guru's voice, I was taken aback by the force of her words. "My child," she continued, "you have forgotten that access privilege and virtualness are independent of each other. Determining whether the function is to be statically or dynamically bound is the last step in resolving a function call. You need to read and meditate upon the Holy Standard, chapter 3.4 verse 1, and chapter 5.2.2 verse 1 [2]."

I decided it was time to demonstrate my brilliance. I opened my mouth... "Oh, yeah. Uh, right." So much for brilliance. I decided to take another tack distraction. "But I still don't understand why it's private. It just doesn't make sense to me." "Meditate upon this parable, my apprentice. You are creating a class, and after reflecting on the design, you decide the class requires a member function that should not be called by any other classes, not even its derived classes. What do you do?" "You make it private, of course," I replied. The Guru looked at me, her eyebrows raised expectantly. My brain kicked into overdrive trying to figure out how to link together private and virtual functions. Wendy spoke up. "Have you examined the implementation of the **Mountie** class, especially the **write** function?" I quickly turned to my keyboard, glad to briefly escape the Guru's unrelenting gaze. I found the function fairly quickly:

```
void Mountie::write(std::ostream &Dudley) const
{
    Dudley << classID() << std::endl;
    do_write(Dudley);
}
```

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995).

[2] ISO/IEC 14882:1998, "Programming Languages C++," clauses 3.4 and 5.2.2.

I could tell Wendy had spent way too much time watching cartoons as a child. "I see now," I said. "**classID** is an implementation detail, used to signal the concrete type of the class that is being saved. Derived classes must override the function, but since it is an implementation detail, it's kept private." "Good, my child," the Guru agreed. "Now explain to me why **do_write** and **do_read** are not private." That stopped me for a moment. But then the penny dropped: "Because the derived class has to call its parent class's implementation of those functions so that the parent class has a chance to read and write its data." "Very good, my apprentice," the Guru was almost beaming. "But why not make them public?" "Because," I was really warming up to this now, "they must be called in a controlled fashion, especially the **do_write** function. The object type has to be written to the stream first so that when the object is read in, the factory knows which type of object to instantiate and then load from the stream. Allowing public access to the functions would cause chaos." "Very wise, my apprentice." The Guru paused a moment. "As with spoken languages, there is more to knowing the C++ language than just knowing the grammar and the rules. You must know the idioms." "Yes, Coplien's book was the next one on my [1]" The Guru held up her hand. "Peace, child. I do not refer to the prophet Coplien. I refer to the linguistic sense of the word idiom the implied meaning behind certain constructs. You know, for example, that a virtual destructor tells you 'I am meant to be used as a polymorphic base class, you may create from me little ones as you need' whereas a non-virtual destructor tells you 'I am not meant to be used as a polymorphic base class, do not derive from me for that reason, I pray.' "In a similar manner, the access privilege given a virtual function conveys an idiomatic meaning to the initiated. A protected virtual member tells you 'my little ones should or perhaps even must invoke my implementation of this function.' A private virtual function tells the initiated 'my little ones may or may not override me as they choose, however they may not invoke my implementation.' " I nodded as I absorbed this. "What about public virtual functions, though?" "Avoid them where possible, and prefer to use the Template Method. Consider this parable." She picked up the dry erase marker, and began writing in her delicate script:

```
class HardToExtend
{
public:
    virtual void f();
};
void HardToExtend::f()
{
   // Perform a specific action
```

[1] Coplien. Advanced C++ Programming Styles and Idioms (Addison-Wesley, 1992).

```
}
```

"Consider that you have released this class, and your requirements have changed," she continued. "In the second release of this class, you discover you need to implement the Template Method on the function **f()**. It will be nigh on impossible to accomplish. Can you see why?" "Uh, um. Yes, well... I think... No, I can't." "There are two possible ways to convert this class to the Template Method. The first way would be to move the implementation code of **f()** into a new function and make **f()** non-virtual, like this:

```
class HardToExtend
{
// possibly protected
    virtual void do_f();
public:
    void f();
};
void HardToExtend::f()
{
    // pre-processing
    do_f();
    // post-processing
}
void HardToExtend::do_f()
{
    // Perform a specific action
}
```

"However, **HardToExtend**'s little ones will expect to override **f()**, not **do_f()**. You must now change all classes derived from **HardToExtend**. If you miss just one class, that class will attempt to override a non-virtual function. This could introduce, in the words of the prophet Meyers, 'schizophrenic behavior' into your class hierarchy [1]. "The other solution is to introduce a new, non-virtual function and move f() to the private section, like this:"

```
class HardToExtend
{
// possibly protected
    virtual void f();
public:
    void call_f();
};
```

[1] S.Meyers. Effective C++: 50 Specific Ways to Improve Your Programs and Design, 2nd edition (Addison-Wesley, 1998); Item 37: "Never redefine an inherited non-virtual function."

"This will cause users of the class no end of headaches. All the clients of **HardToExtend** will attempt to invoke **f()**, not **call_f()**. Those clients will no longer compile. Furthermore, derived classes will most likely leave **f()** in the public interface, and clients that use those derived classes directly, rather than using a pointer or reference to **HardToExtend**, will still have direct access to the function you want to protect.

"Virtual functions should be treated very much like data members make them private, until design needs indicate a less restricted approach is indicated. It is much easier to promote them to a more accessible level, than it is to demote them to a more private level."

- - - - -

"When did all this happen?" Jeannine asked.

"Just before the turn of the millennium the real turn, that is. It was at the end of 2000, I think. We were just about to celebrate New Year's Day, and we were really looking forward to the special year of 2001, because of all the literature, you know. . . "

Oddly, it was shortly after that conversation that we heard about the discovery of the obelisk under the surface of Europa.

6 Obelisk

–Perils of writing "monolithic" code.

"and parties will start going down next week. I think we'll get to go the week after," Jeannine finished. "They're still excavating around the top of the crystal obelisk, but no further news yet. It's hard going, and we didn't plan to do this kind of exploration. This goes way beyond our research and mining mandates. Frenell still thinks it's a hoax."

I laughed. "Frenell's a genuine space oddity and needs to get out more. It's the wrong color; it's in the wrong place; it's"

"Sure, we've known for months now that it was deep under the ice."

"it's the wrong shape, and if someone was playing a practical joke, they'd do better than that. Anyway, no joker could ever embed it that deeply."

"I've heard a better one," Jeannine interrupted. "Aubrey believes it's real, all right, but he thinks it's a conspiracy. Decades too late for 2001, but 'still suspiciously similar,"' she imitated the junior officer, exaggerating his characteristic sibilance.

"Aw, Aubrey sees government cover-ups everywhere. Remember when we were kids, and cover-ups about magic ricocheting bullets were all the rage?" I paused, and smiled thoughtfully. "Speaking of monolithic, actually, did I tell you about my bizarre discovery just after New Year's Day, back in the real 2001 . . . ? No? Well, it was while I was still at my first job . . ."

- - - - -

It was January 2, a Tuesday, when I discovered Bob's monolithic code. Happy and with loosened belts, we were all just back from the holidays unfortunately, delivery deadlines wouldn't wait and we were slowly getting ourselves back into motivated mode. I was mainly happy that I'd managed to survive my job's probation period; I was now a full employee, although some days I still wondered whether that was entirely a good thing, given that the job often had me working in close proximity with the Guru. She was smart, all right, but more than a tad strange.

It was getting close to lunch time when Wendy interrupted me. "You," she called.

I gophered up. "Yeah? What?" I'd been in the middle of a debugging session trying to find an error in my code and was just about ready for a break. Anything had to be better than stepping through the same function for the two dozenth time.

Wendy appeared. "Bab's still out this week, and I need a quick fix to the utility module he took over from you a month ago; I'm getting a buffer overflow trap at line 510. Can you look at it and fix it by this afternoon?"

"Sure." She gave me the filename, we both ungophered, and I pulled it up in my editor.

Half an hour later, I had to admit that I'd been wrong after all: There were indeed some things that were worse than stepping through my own code in the debugger. I finally gave up and walked over to Wendy's cubicle. "Sorry," I said merrily.

Wendy started; I guess she hadn't heard me come up behind her. I suppressed a smile . . . usually it was the Guru who did that to both of us. "What do you mean, sorry?" she asked.

"You wanted it fixed quickly. I'm sorry, Wendy; I'm afraid I can't do that."

"What's the problem?"

"I think you know what the problem is just as well as I do."

"What are you talking about . . . ?"

I sighed and stopped playing. "I can't fix it quickly. Have you seen what Bob's done to the code since I handed it off to him? I swear this bears almost no resemblance; its origin and purpose are a total mystery to me. It'll take me half a day to understand what he's done to it." I reached over her and pulled up the file on her screen, went to the top which is where the function began, and then held down the scroll-down key. Nearly 2,000 lines later, we reached the matching close-brace at the end of the file. "The function is now 1,908 lines long roughly, a 1,200-line if block, followed by a 700-line else block. Most of the rest can be fixed with a source code beautifier, I think."

Wendy was shaking her head in disbelief, but nodded. "I've seen him do that before, but I really thought we'd broken him of the random tabbing and long code lines, and especially the random vertical whitespace."

"There's at least one spot where you have to page down past a full screen of empty lines before you get to the next code line! And did you notice how he sometimes tabs out when entering a for or while block [1]? The Guru usually won't stand for this . . . doesn't she enforce the coding standards any more?"

"Indeed we do, my child," sighed a voice from close behind us. We both jumped. The Guru continued sadly, "Bob has been warned before. He shall be warned again, and this time be docked of pay. But . . ." she trailed off thoughtfully.

Neither Wendy nor I wanted to speak immediately. The silence settled. Finally, I quietly prompted: "But what?"

[1] Note from the authors: Believe it or not, this is an actual example of code written by a programmer the authors encountered some years ago.

The Guru blinked carefully. "How long is the longest line in the file?"

Wendy did a quick check. "Several are exactly 848 characters long. That's the longest."

"And what is the shortest?"

This took Wendy a little longer. "The shortest nonblank line is 212 characters. There are a few that length."

The Guru seemed to do some quick mental arithmetic, and sighed again. "I see. Perhaps he was making a statement."

I thought for a moment, then: "Oh. I get it: One to four to nine. That's the ratio of line lengths to total function size."

Wendy groaned.

The Guru pushed her hair back behind her ear and adjusted her glasses. "Avoid monolithic code," she said. "Bob is a structured programmer, but even he should know well the merits of functional decomposition. Prefer 'one class, one responsibility,' and 'one function, one responsibility.' This function is obviously decomposable into at least two parts, and yet even those parts would be too complex and should be decomposed further. Wendy, instruct this young one: What are the benefits of good code decomposition?"

"Clarity," Wendy said, ticking off each point on a finger. "Reusability. Encapsulation. Maintainability . . . "

And it was at that moment, at just that unfortunate moment, that we heard the front door opening, the whistling wind outside, and the front door closing. And Bob walked in.

"Hey dudes," Bob said, taking off his mittens and knocking snow off his boots all over the carpet. He scratched himself absently. "Whassup?"

"I thought you were off this week," Wendy grumbled.

"Yeah, sure, babe, I just had to swing by to get some files that I'd downloaded here . . . surfing the net . . . er, ah, well, that's not important. Hey," he squinted at Wendy's monitor, the centerpiece to our little gathering. "That's my code."

"That was my code. It used to be good code. Before you turned it into a joke," I said hotly, gathering steam. Bob had more years of experience than I did, but my probation period was over, and I was angry; and I guess I might still have been a little hung over, as I'd had a pretty lively last several nights. I felt like risking it.

Bob shrugged. "Hey, look. I can see you're really upset about this."

"Upset? Bob," I retorted, "you've made a travesty of my code!"

"It can only be attributable to human error," the Guru agreed quietly.

"Hey," Bob said defensively, "I honestly think you ought to calm down. Take a stress pill and think things over."

"Bob," said the Guru, "begone."

"Hey, I didn't ask to be" Then Bob caught himself, and calmed down. "Hey, look. Okay. I know I've made some very poor decisions recently, but I can give you my complete assurance that my work will be back to normal . . . "

"Bob, this conversation can serve no purpose anymore. Goodbye. Take your vacation. We will discuss it privately on Monday."

Bob grumbled, but seemed to think better of it and left. So did the Guru, after smiling. Once both were out of earshot, Wendy and I dissolved into a fit of the giggles. When we recovered, Wendy ran the file through a source code beautifier as a start, and I spent the rest of the day rewriting Bob's code. Along the way, I discovered that some of the resulting smaller functions could be reused nearby, now that their functionality was better isolated instead of being buried inside a monolithic block. The smaller chunks were also simpler to grasp, and soon I found Wendy's problem and solved it. I added a unit test for Wendy's problem, reran all the unit tests to be sure everything still passed, and checked the module back in.

- - - - -

"You sure picked one hoot of a first job," Jeannine shook her head.

Before I could answer, the rec lounge door opened and Brzenkowski walked in. He started walking over to the far side of the lounge without looking up, which was unusual for him.

"Hey," I called, "you don't say hi any more?"

Brzenkowski looked up. "Hmm? Oh, sorry. Say, have you heard the news yet?"

Jeannine frowned. "What news? From the excavators?"

"Yes. It appears that the obelisk," Brzenkowski said quietly, "isn't just an obelisk. It seems to be the top of a building."

7 Access Restrictions

Stirring the debate about public data vs. accessor methods: the latter are safer, just as fast when inlined, and can have nicer syntax if operator overloading is appropriate.

Boom.

The distant thundering quickly passed. It was faint, felt more than heard. The excavation teams were at work again.

"I wish..." Jeannine started, then trailed off.

"I know." I buried my face in my work. After a long time, I added: "I wish I could send mail home too. But the comms should be working again soon."

"They do work," Jeannine said.

Boom. Another distant explosion.

"Yeah, but while the dome's systems are damaged, it makes sense to reserve them for only emergency traffic. We just aren't being given access, for now. That's reasonable." My heart wasn't in the words.

There was silence again, for a time. Finally, Jeannine whispered: "Yeah. Well, that's what they say, now isn't it?"

"That's right. It's just temporary access restrictions. I don't like it, either, but surely you don't think they're...." It was my turn to trail off.

Boom.

- - - - -

That was all the sticky note on my monitor said. Well, that and a filename. I recognized the file it was a utility class I had added to the project a while back. The note was written in the Guru's distinctive handwriting, but I had no clue what it meant. The Guru wasn't in yet, so I couldn't ask her.

Just then, Wendy, the programmer in the cubicle next to mine, came in and flopped down in her chair. I gophered up, hoping she'd be able to make sense of it. She looked haggard, resting her head on her hands. "You look like you had a rough weekend," I ventured.

She nodded without looking up. "When I got home Friday, the carbon monoxide detectors were beeping away turns out the furnace was clogged up. We had to spend the weekend at a friend's house, until it was fixed. I couldn't sleep a wink I kept wondering what would've happened if we hadn't installed the detectors [1]."

[1]Note from the authors: this situation actually happened to me (Jim) recently. I urge all readers to install carbon monoxide and smoke detectors and ensure they are well-maintained. I know it's the best investment I've ever made! Contact your local fire prevention office for information and assistance.

"Wow, that was a close call," I commiserated, making a mental note to double-check my detectors. "Is everyone OK?" Wendy nodded. I decided she needed a distraction, so I mentioned the note to her and asked her what it might mean.

"Wait a minute," she said. "The Guru left that note on your monitor?" I nodded. "My friend," she said, "it sounds like you may have just pulled a Bob."

I was aghast, speechless, at such strong language. "You're joking!" I blanched. Bob was the worst programmer ever, and I had made it my personal mission to thwart him and his poor programming style every opportunity I got.

Wendy shook her head, dead serious. "Nope. He's usually the only one who gets those cryptic notes, and if you got one, then you must've been a very bad boy. Let's have a look at that source file." She turned to her computer, logged in, and called up the file in question.

I dragged over a chair. "Everything in here looks good to me," I said, peering over her shoulder.

Wendy jumped slightly she must have been more stressed than I realized. "It seems OK on a quick glance," Wendy agreed. "Maybe a diff against the previous revision will give us some clues." I watched as she used our revision control system to show the difference between the two versions of the file. I was impressed I was still trying to master the basic checkin/checkout syntax, and here she called up a diff between two revisions in no time at all. I was still trying to sort out the commands Wendy had used when she spoke up.

"Well, here's something, but it's really minor," she said, pointing to the screen. "You've changed some of the class data members."

I peered closer at the screen. "Yes, I reworked the implementation slightly. I changed the three separate doubles, representing the object's location, into an array of three doubles. It made some member functions much more efficient. But the members are all private, see?"

```
class Point
{
private:
    double location[3]; // x, y, z coordinates.

public:
    void setLocation(double x, double y, double z) {
        location[0]=x;
        location[1]=y;
        location[2]=z;
   }

    void getLocation(double &x, double& y, double &z) {
        x=location[0];
```

```
        y=location[1];
        z=location[2];
    }
};
```

Then I noticed a blank line at the top of the file that the diff viewer indicated had changed. "Now, why would the viewer highlight that line?" I wondered out loud. "Probably just white-space changes," Wendy surmised. "Here, let me set the viewer to ignore white space." A few keystrokes later, and... the line was still highlighted. I placed the cursor on the line, hit the "End" key and froze in horror at the sight in front of me. Indented way off to the right, out of normal view, was this statement:

```
                                #define private public
```

Wendy and I looked at each other. "Bob!" we chorused. Bob, as it turned out, was already walking toward us. The Guru wasn't the only one who could show up at the most appropriate moment. "Hey, Junior," he said, "I just came by to see if you've fixed the problem yet." I closed my eyes and counted to ten. Slowly. Then I counted back down to zero. I held up the sticky note: "Do you mean this problem?" I asked sweetly. "Heh. Yeah, that's the one. Her Weirdness left it on my monitor. When you changed the x, y, and z members to that array, it broke a raft and a half of my code, so I passed it on to you. Fix it, and quit messing things up from now on, will ya." "Bob, what's the idea of this #define statement?" I pointed to the offending line. "Oh, that," Bob chuckled. "Pretty cool, eh? My code needed direct access to the coordinates. Efficiency, y'know. At first, I just had the #define statement in my own code, but I was using it a lot, so it was easier to add it to your header. I'm doing a lot of number crunching, and I can't afford the overhead of the accessor functions you provided. But then you got rid of the individual doubles, for that array. And my code went 'Boom!"' "Boom, indeed." As if on cue, we heard the Guru's voice. We all jumped a little. She used a finger to pointedly move her glasses back up the bridge of her nose. "And that, you purveyor of programming perversions," she rebuked, "is why I left the sticky note on your monitor. You violated the class's encapsulation by attempting to subvert the access privileges. You violated the One Definition Rule by changing private to public in your code. You must fix your code to use the accessor functions." "I already said, I can't afford the overhead of the accessors, dearie," bristled Bob. "I have to-" "Silence!" the Guru interrupted. "The accessors are inline and should cause no overhead." Bob started to protest again, but the Guru headed him off. "However, in order to avoid the root of all evil [1], we will create a test harness to demonstrate the

[1] "Premature optimization is the root of all evil" Donald Knuth. Depending whom you

actual costs of the accessor functions. Meanwhile, clean up that unseemly code. Then read and meditate on 1 Meyers 20 [1]." Bob grumbled as he left for his cubicle. The Guru simply stood silently until he was gone and then led me back to my own cubicle. "My apprentice," she said, her voice soft again, "please write a test harness that will time the cost of direct access, against inline accessors and non-inline accessors. Be sure that the non-inline accessors are in a separate translation unit. While you are at it, an inline operator [] would be useful." I thought about that for a moment. "You mean like this?" I began writing on my whiteboard:

```
class Point
{
public:
    inline double operator[] (int index)  const {
        return location[index];
     }
};
```

"Very good, my apprentice. You have remembered the blessings of const-correctness. But now, write it so that it can serve not only as an accessor, but as a mutator as well. What other benefits can you derive from this function?" I thought about it for a moment. "You can do bounds checking on the index." I revised the scribbling on the whiteboard:

```
class Point
{
public:
    inline double &operator[] (int index) {
        assert(index >=0 && index <= 2);
        return location[index];
     }
};
```

"Hey, cool!" This, I liked. "The bounds checking has no performance penalty in release mode."

The Guru nodded. "And," she added, "with a good optimizing compiler, the compiler should be able to generate code that is as efficient as accessing the members directly, with the added blessings and safety of encapsulation."

"So," I mused, "there really is never any reason to have a public data member, is there?"

---

believe, he said it in one or more of: "Structured Programming with go to Statements" (Computing Surveys, Vol. 6, No. 4, December, 1974, page 268), Literate Programming, or Computer Programming As an Art (1974).

[1]Scott Meyers, Effective C++, 2nd edition, Item 20: "Avoid data members in the public interface" (Addison-Wesley, 1997).

The Guru paused a moment before answering. "In most cases, no. However, if the class is merely a convenient aggregation of data, with no object being modeled that is to say, a C-style struct of data with no additional behavior then it would be reasonable to have all data members public [1].

"You can expose private members through accessor and mutator functions but only if it is really necessary. An excessive amount of Get and Set member functions including ones that are disguised as operator [] indicates that you are not thinking of the blessings of encapsulation. If your tests indicate that the accessors are providing a substantial run-time penalty, then you can improve the speed by making the accessors inline. Of course, the savings and penalties will vary from compiler to compiler hence the need for the test harness."

"I'll get right on it, boss," I replied cheerfully. She smiled, folded her hands and silently walked away. I sat down to write the test harness. The results, at least on the compiler I was then using, were quite interesting [2]:

Table 1: Approximate run-time in milliseconds

Implementation	Optimizations off	Optimizations on
1: direct access	1061	251
2: inline accessor function	1673	240
3: out-of-line accessor function	1662	1432

- - - - -

Boom.

"I don't like it," Jeannine repeated. "They say the dome is damaged because of an accidental internal explosion, so we don't have access to go there, but work goes on in there, and people with higher clearances have been allowed in and they never come out again. They say the long-range comms systems are damaged and are restricted for emergencies only, but the only real effect I can see is that we can't communicate with anybody outside near-Jupiter space."

"So?"

"I don't like it."

Boom.

We said no more for some time, and eventually the distant thundering stopped, but the resulting silence was worse than the faint noise had been. A few minutes later, Jeannine frowned, got up, and left.

[1] C++ FAQ Lite, Marshall Cline, http://www.parashift.com/c++-faq-lite/classes-and-objects.htm#[7.8].

[2] The source code and full details of this test are available on the CUJ website.

8 Redirections

How to go about redirecting output to cout/cerr using streambufs.

I wasn't too concerned until the third day passed without seeing Jeannine. After some hesitation and mental handwringing, I went to see Frenell about it. She was the obvious choice; both Jeannine and I reported to her just then.

Her stateroom door slid open on the second knock. "Oh, it's you," she said. "What's on?"

"Just looking for Jeannine. Haven't seen her for a few days," I explained.

Frenell regarded me briefly, then said: "Oh, yes. She's been temporarily reassigned."

"She has? Where?"

"Over to the dome. They've been requesting some extra manpower there and at the excavation site. While she's there she has to stay there; access is restricted because the locks and tunnels aren't in great shape, and it's a pain to get people to suit up to get in and out just to shuttle back here."

"Are a lot of people being reassigned? It's been feeling a bit empty out here lately." All I knew was that repairs were apparently still going on in the main dome, since the explosion a few weeks before. Those of us on the surface kept to the outlying buildings and continued with our local work while we were out of contact with the inner system; long-range communications, which went through the dome's systems, were still crippled and under repair.

"A few," she said, and leaned on the side of the doorway. "Why? Want me to put in your name if there's an opening for someone with your skills?"

"Sure," I said without thinking and then wondered at my own quick reaction. Was I that concerned about Jeannine? Or was I just anxious to see her again? Either way, my feelings surprised me.

If Jeannine had been redirected to the dome, I figured I might as well go too if the chance came up. Briefly, I recalled a day, many years ago, when I had learned to deal with another kind of redirection. Oddly enough, it also had a bit to do with restricted access problems.

- - - - -

It was just as Wendy was passing my cubicle that I screamed: "Aaagh! I hate access violations!"

The frustration was too much, and I finally let it boil over into what I hoped was a primal roar. The effect was somewhat diminished in that my voice was still maturing, and it came out more like a mouse's roar.

It was enough to make Wendy stop and look in on me. "Sure and what's the matter, me young friend?" she said with a fake Irish accent. With St. Patrick's Day fast approaching, she was playing up her Irish heritage to the. . . lilt.

"Wendy, do me a favor," I sighed, "don't you act weird, too. One basket case around here is enough."

"Hey, you okay?" she said quietly, kindly. "What brought this on, buddy?"

"Just right now it's this code, but I guess it's also a bit to do with the Guru," I admitted. "At first, I was scared witless by her. I was sure she was an escapee from the local asylum. Then I kind of got to like her calling me 'my apprentice.' Yeah, it's a bit strange, but essentially harmless. But when I told my friends, they thought it was really condescending, and once I got to thinking about it, I realized they were right."

Wendy smiled. "Ah," she said, "I see. Look, you just haven't seen her from the right perspective yet. You really should try to get to know her outside work." I tried to imagine her home: shelves filled with dusty computer books and magazines; incense burning on altars dedicated to various computer scientists. . . . My face must have betrayed my thoughts Wendy burst out laughing. "She does the Guru shtick mainly to annoy Bob and scare interns. I go along with it because it bugs the Hades out of him."

"Oh." I remained unconvinced. The act was just a little too convincing for my liking. But I didn't want to argue, so I sighed and composed myself, and came back to the point: "I've been wrestling with this problem for a few hours. Simple task, really just reassign **cout** and **cerr** so that their output goes to a file, instead."

"Sorry, run that by me again?"

"I'm integrating a library developed by another group. The library was written with a command-line interface in mind, so all the debugging and diagnostic messages get sent to **cout** and **cerr**. What's worse, I think Bob was on that team, because there's no rhyme or reason to when messages go to **cout** and when they go to **cerr**. In fact, some messages are split between the two streams! So I have to capture and integrate both streams into a single log.

"Anyway, as I said, I'm integrating this library into our GUI application. Unfortunately, the GUI just tosses any output to **cout** and **cerr** into the bit bucket. So, what I'm trying to do is redirect the output into a file, instead. Simple job. So simple I can't figure out how to do it. I've boiled the problem down to this small program." I dragged the guest chair over to the monitor for Wendy to sit down.

```
#include <iostream>
#include <fstream>
int main()
{
```

```
    std::ofstream logFile("out.txt");
    std::cout = logFile;
    std::cerr = logFile;
    std::cout << "This goes to cout\n";
    std::cerr << "This goes to cerr\n";
}
```

"It compiles and runs OK on one compiler, but it gives me an access violation when I use another compiler."

Wendy studied the screen for a moment. "Hmmm...I'm not sure you're allowed to reassign streams like that."

"Indeed, child, it is an unnatural act clearly forbidden by the Holy Standard."

We jumped. But only slightly. Act or no act, I had to admit that the Guru was uncannily good at appearing at exactly the right moment. "So, what I'm trying to do is impossible?" I sighed.

"No," the Guru smiled. "It is quite simple, actually. My apprentice, which compiler did you use in your studies at university?"

"Well, it was actually an older compiler and didn't have very many standard features."

The Guru nodded sagely. "I see. Your compiler would be using what the prophets Langer and Kreft refer to as 'classic IOStreams' [1]. You must become familiar with standard IOStreams in particular the stream buffer classes. No longer is the stream buffer a lowly implementation detail; it is a blessed class in its own right. The designer of the IOStreams classes, Jerry Schwarz, foresaw that the stream buffer class would be an extremely useful feature. Yet its value and power are often overlooked."

"OK. So what do I do?"

The Guru picked up the dry-erase marker and quickly wrote out some code on my white board:

```
#include <iostream>
#include <fstream>
int main()
{
    std::ofstream logFile("out.txt");
    std::streambuf *outbuf = std::cout.rdbuf(logFile.rdbuf())
        ;
    std::streambuf *errbuf = std::cerr.rdbuf(logFile.rdbuf())
        ;

    // do the actual work of the program;
    // GUI code and event loop would go here
```

[1] Angelika Langer and Klaus Kreft. Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference (Addison-Wesley, 2000).

```
    std::cout << "This would normally go to cout but goes to
       the log file\n";
    std::cerr << "This would normally go to cerr but goes to
       the log file \n";
    logFile << "This goes to the log file\n";
    // end of program body

    // restore the buffers
    std::cout.rdbuf(outbuf);
    std::cerr.rdbuf(errbuf);
}
```

"The **rdbuf** function returns a pointer to the stream buffer managed by the base class, **basic_ios**. The overloaded version allows you to replace the stream buffer, and the return value is the original stream buffer. The solution is simple you replace the stream buffers for **cout** and **cerr** with the stream buffer of your log file. At the end of your program, you restore the original stream buffers. As you can see, you can still use **logFile** as a regular output file."

"Cool!" I was beginning to see the power of the stream buffers. "Hey, if I only wanted to redirect one stream, say **cerr**, could I just simply swap the two buffers and not have to worry about restoring them, like this:"

```
int main()
{
    std::ofstream logFile("out.txt");
    std::streambuf *saveBuf=cerr.rdbuf(logFile.rdbuf());
    logFile.rdbuf(saveBuf);
    // remainder of program...
}
```

"Alas, it is not that simple," the Guru sighed. "Your code will not compile. The **basic_ios::rdbuf** functions are not virtual. **ofstream** provides only one **rdbuf** function, which takes no parameters and returns a pointer to the file stream buffer embedded in the **ofstream** object. This means that **std::ofstream::rdbuf(void)** hides **std::ostream::rdbuf(std::streambuf *)**.

"You could manipulate your log file through a pointer to **std::ostream**, like this:"

```
int main()
{
    std::ofstream logFile("err.log");
    std::ostream * baseManipulator = &logFile;
    baseManipulator->rdbuf(std::cerr.rdbuf(baseManipulator->
       rdbuf()));
}
```

"This code will compile, but will not necessarily work properly. The Holy Standard does not specify how **ofstream** is to be implemented. It is possible that compiler implementers will implement **ofstream**'s functionality such that it ignores the base ostream::rdbuf function and instead works directly on the file stream buffer. The Standard does require **ofstream::close** to call **ofstream::rdbuf** to determine which buffer to close. Since this function always returns a pointer to the **filebuf** embedded in the **ofstream** object, **ofstream** will always close its embedded **filebuf** object, leaving **cerr** without a valid buffer. Thus, you still need to restore **cerr**'s stream buffer, by inserting this line before **main()** exits:

```
std::cerr.rdbuf(baseManipulator->rdbuf());
```

"If you ever want to redirect **cin**'s input, you must take similar precautions, as **ifstream** behaves analogously."

The Guru turned to leave, then paused. "I am inviting some people to my home this Saturday. Nothing special, just a social evening. You and Wendy, and your respective significant others, are welcome to attend."

I opened my mouth to gracefully decline. "Gaaah!"

"He means he'll be glad to come," Wendy smiled sweetly. My shin was in too much pain from where she had kicked me, so all I could do was nod helplessly through gathering tears. The Guru smiled and glided away.

I waited until both the Guru and the pain were gone, then growled at Wendy: "I'll get you for this."

- - - - -

"Okay," I told Frenell, "just let me know if you do put my name in."

"Sure will."

I was about to go, but asked: "Say, I wonder if Ling didn't get reassigned too, along with Jeannine I mean. They were watch partners."

"No," Frenell said. "We need her here."

"Thanks. See you tomorrow."

During my next watch, I checked some of the duty rosters that I had access to view. The names made a long and varied list - Jupiter et al. was UN space, so this was a UN mission, with personnel from many zones, predominantly from the American Union and the Asian Federation due to their heavy science budgets. None of the names was overtly marked as reassigned, but I could tell which names were no longer on regular rotation in the duty roster; those had to be the ones working elsewhere.

I paged through idly. It was a quiet watch.

In time, I noticed an odd pattern: No Asians were being reassigned to the excavation site or to the dome.

9 Manipulations

How to write a simple IOStream manipulator to quote and unquote strings.

That day began much like all other days.

I got up early to catch up on some technical reading, ate in the mess hall, and was five minutes early for my watch. The watch itself was routine and uneventful until nearly the end. Then, about ten minutes ahead of time, Brzenkowski wandered in for the next watch.

I suppose that was really the first odd thing, because Brzenkowski was never early for anything. I didn't think anything of it at the time, though. He looked at a few things and then walked over and said: "Hey. Frenell wants a minute of your time. You should drop by her stateroom on the way down."

"Oh? What's on?"

"Dunno," Brzenkowski shrugged. Then he smiled a little and clapped me on the back. "You might as well get out of here," he added. "I relieve you."

"Your watch," I agreed, signed off, and headed out and down.

When I got to Frenell's stateroom, the door was already open. She waved me inside, and the door slid shut behind us. There was another man sitting in her visitor's chair. I didn't remember seeing his face before.

"Hi, hi," he said in a friendly voice, standing to take my hand.

"This is Major Gilb," Frenell introduced us, and we sat.

"In understand you're available to help us out in the dome?" Gilb asked directly.

I glanced at Frenell. "Sure. If my skills fit."

"Oh, I think they will," Gilb assured me. "You worked with Jeannine Carruthers, didn't you? Yes, she works in my department now, and she speaks highly of you. Yes, quite highly. Now, I know you're doing more day-to-day work at the moment," he continued, "but this would be a bit more in the research area. Interested? Yes?"

"Sure. That would be nice. I've asked for some more research-oriented work anyway."

"Good. Good, I think that will fit the bill nicely. Yes, nicely. You understand of course that you'll need to stay for a time, at least until we get things well enough repaired to resume regular travel between these buildings and the dome? Don't want to use the transit tunnel more than we need to."

He began to ask me many things. I don't remember everything he said. My mind drifted, and among other things, I was thinking that it would be good to see Jeannine again. Funny, I thought, it's not like I'm a young buck with a new girlfriend. Not like it had been, on my first job, many years ago...

- - - - -

"There has to be a better way," I muttered.

"Hey, pardner, how's it going?" Wendy's cheerful voice floated over the cubicle wall. I stretched and decided it was a good time to take a break from my code, so I gophered up.

"Pretty good," I replied. "You back already? I thought you were going to be out for two weeks?"

That earned me a sideways glance. "Where've you been, on Jupiter? I was on training for two weeks and then took another week holiday!"

"Time flies when you're having fun," I allowed a devilish smile to sneak its way onto my mouth.

"Hmmph," she hmmphed. "You've been hanging out with Anna! See? I told you going to the Guru's party would be fun. Not that I expected you to go out with her daughter."

"Hey, I didn't know she was the Guru's daughter for a few weeks," I protested. "I thought she was just another party guest."

"You can tell me all about it over lunch." She tucked her rain boots under her desk and slipped on a pair of shoes. "So, how's it been going around here? You were looking a little intense when I came in."

"Yeah, pull up a chair and I'll fill you in on the gory details. You might have some fresh ideas." I dropped back into my seat as Wendy came into my cubicle.

"I implemented some of the communications functionality on our new system. It's a pretty simple protocol. Basically the server just responds with a series of comma-delimited, quoted strings. The strings can contain quotation marks, so I wrote a simple function that takes a string, wraps it with quotation marks, and escapes any embedded quotation marks, like this," I demonstrated on the whiteboard:

Table 2:

Input	Output
field 1	"field 1"
field "with quotes"	"field "" with quotes""

"I wrote two functions, one to add the quotes, and another to strip them:"

```
std::string quote(const std::string &);
void f(std::ostream &o)
{
    o << quote("whatever");
}

std::string unQuote(std::string &);
```

```
void g(std::istream &i)
{
   std::string input;
   i >> input;
   std::string original=unQuote(input);

   // use string 'original' ...
}
```

"OK, that sounds straightforward," Wendy replied. "Your typical text file parser stuff."

"Right, that was pretty easy. But if the original string is large, then I end up making a copy of the entire string. Our server software has to be pretty tight, running on those boxes we're making. Also, some of the other project members complained that it was a little awkward to use, and they wanted to be able to use it with stream insertion and extraction, like this," I squeezed some more code onto my whiteboard:

```
std::string message;
// aStream is an istream subclass
aStream >> message;
```

"Somehow I have to take the text stream in **aStream** and extract one field into **message**. I've been playing with the stream buffers to see if that would work, but it's looking pretty ugly."

"Manipulators." Wendy and I jumped yet again at the Guru's soft voice.

"Beg pardon?" I asked.

"Write your own **IOStream** manipulators, as Langer and Kreft described in C++ Report a while back [1]."

"Uh, don't you mean the prophets Langer and Kreft?" I asked.

"Bob's in a meeting, and I don't have the energy for the Guru thing right now," she shrugged. Even though she wasn't in her Guru act, I still couldn't help but think of her as 'The Guru'. And she still managed to sneak up at the right moment! I was beginning to wonder if she had hidden a spycam somewhere.

"Ah," I said, wondering how come my replies always seemed so lame. "So how do you write an **IOStream** manipulator?"

"There are two approaches, depending whether you're implementing a manipulator with arguments or without. You need to implement one with arguments, so that's all I'm going to describe for you. If you want to know how to implement a manipulator without arguments, you'll have to read the back issue.

"Implementing an **IOStream** manipulator that takes arguments is actually pretty easy. You simply create a class that takes the appropriate arguments in

[1] Klaus Kreft and Angelika Langer. "Effective Standard C++ Library", C++ Report, April 2000.

its constructor, and provide an insertion or extraction operator that calls member functions on the manipulator. For example, suppose you want to write an **ostream** manipulator that manipulates an integer, you'd do this." The Guru erased my chicken scratches and wrote neatly in her fine, spidery script:

```
class myManipulator
{
    int manipValue;
public:
    explicit myManipulator(int i) : manipValue(i) {} // Note
       it's 'explicit'!
    friend ostream & operator <<(ostream &os, const
       myManipulator &mm);
};
```

"Using the manipulator is quite simple, as simple as using **std::setfill** or **std::width**:"

```
std::cout << myManipulator(42);
```

The compiler will create a temporary, unnamed object of type **myManipulator**, and initialize it with the value 42. The compiler then invokes the overloaded insertion operator. The insertion operator performs the necessary manipulations on the stream, say something like this:"

```
ostream & operator <<(ostream &os, const myManipulator &mm)
{
    if (mm.manipValue == 42) {
        os << "Life, the universe, and everything";
    } else {
        os << "What was the question, again?";
    }
    return os;
}
```

I thought about it for a moment. "So..." I began cautiously, "my ostream manipulator would perform the character escaping and unescaping?"

"You could do that, or you could implement it as two separate classes, escape and unescape [1]." She paused suddenly, and a subtle change came over her. Just then, Bob walked past, on his way to the coffee station. "Remember, my child," the Guru continued, "to read and meditate on the writings of the prophets Kreft and Langer."

"Yes, my master, I will show you my writings this afternoon" I replied. Bob stopped in his tracks, dropping his latté cup on the carpeted floor. He picked

[1] A complete, working example of the narrator's manipulators is available on the CUJ website at hyslop.zip.

up his cup and, without looking back, continued on. After he was out of sight, Wendy, the Guru, and I exchanged a three-way high five. Well, we tried to, that is.

- - - - -

Major Gilb didn't have much to cover, at least not immediately. Soon he wound down and said: "Yes. Yes, you'll be fine. Welcome to the team."

We shook hands, and I grinned. "Thanks. When will I be reassigned?"

"What? Oh, you already are. Don't worry about packing. Your things are already in the dome. They were moved while you were on watch; it's all right."

Before I could quite recover from that, I was taken along and presently found myself in a car that had been waiting in the transit station. While Gilb was still making cheerful small talk, its door closed, and we accelerated into the tunnel in the direction of the dome.

10 Roots

Why can't a Derived** be assigned to a Base**? A peek beneath the covers of the C++ object model.

The obelisk was larger than it had seemed over video.

I stood at the lip of a man-made escarpment, at the edge of the excavation into a surface of slightly brownish-tinted ice. The side of the obelisk facing me appeared to be pure crystal, with a kind of sheen on the surface; it rose upward to its tapered peak about a dozen meters above my head, as though a pointer to where, still further above, the light gray inner surface of the dome covered us and enclosed the entire area.

There was the distant hum of heating units, but despite my clothes I felt cold. Our breath lingered quietly in the air before us. The temperature was bearable, but still kept well below freezing; it would not be good for the ice to melt.

Below me, widening slightly as it went, the obelisk plunged another two dozen meters to where it connected to the top, only partly excavated and visible, of a wider structure which I knew to be the roof of a building still nearly completely embedded in the ice. With only the roof visible below me, the building's size was impossible to judge visually, but I knew from soundings that it went deep indeed, probably to the surface under the ice.

"The entrance is down on the far side," my escort said. "Buster?" My thoughts returned to the present, and I nodded, giving her a brief smile. We made our way down to the floor of the excavated area and walked around parked machinery and people at work in twos and threes.

The entrance at the base of the obelisk was the only thing that was man-made. Its rough edges where until recently a wall had been were clearly not part of the ancient building's original design.

The entrance way was somewhat dark, but temporary lighting had been installed not far beyond. I must have hesitated, because my escort's hand found and squeezed my arm reassuringly. "Well," I smiled back to Jeannine, "I did always want to get to the base of things."

"Yes. Ah, here's our controller."

- - - - -

"I'll see you at lunch, then," I said to Anna, my new girlfriend, and hung up the phone. We had been going out long enough that I was comfortable with the fact that she was the Guru's daughter – something I hadn't known when we'd started dating.

"Hey, Junior!" Bob's voice interrupted my pleasant musings. "I thought I told you not to break the build."

Turning, I said in my best patient voice: "Bob, as you'll recall, the last time I 'broke the build' it was because of some. . . shall we say, less than ideal code in your module." Yep, sipping away at his latte as usual.

"Yeah, well it's your problem this time, Junior." Slurp. "My code keeps crashing because of the changes you made."

I sighed, not wanting to argue with him. "OK, Bob, just tell me where the problem is, and I'll look into it."

"That's more like it," Bob smiled unpleasantly. "Just remember who's got more experience around here, kid. Don't think that sucking up to Her Weirdness is gonna help, either – her authority around me is quite limited." He wrote down the name of the file that was causing him problems and left. I resigned myself to the inevitable and checked out the module.

An hour later, I was still struggling with it. "Oh, man, this is too weird for me," I complained to no one in particular. It did appear to be a problem with my code Bob's code worked fine for the original class, but not with the revisions I had made.

The class hierarchy was fairly simple at this point:

```
class parent
{
public:
  virtual void f();
// etc...
};

class child : public virtual parent
{
public:
  void f();
};
```

One of my modifications was to make **child** inherit virtually from **parent**, for use elsewhere in the hierarchy.

I had tried my best, but it was looking like I had no choice I was going to have to dig into Bob's code. "Once more unto the breach, dear friends," I muttered as I set a breakpoint and prepared to step into the function. After half an hour in the crucible of debugging, I had burned away the excess and had found the root of the problem:

```
void parentPtrPtr(parent **b)
{
  (*b)->f();
}

void childPtrPtr(child **d)
```

```
{
  parentPtrPtr(reinterpret_cast<parent **>(d));
}
```

I had my suspicions about the last function. I had learned to be wary of casting, especially reinterpret_cast. I removed the reinterpret_cast, and sure enough the compiler balked at that line, complaining that parent ** and child ** were unrelated types.

"Your suspicions are correct, my apprentice." For once, the Guru's soft voice behind me did not startle me. Well, not much. "A pointer-to-pointer-to-child," she continued, "cannot be implicitly converted to a pointer-to-pointer-to-base."

I kicked back and stretched. "I thought so. Looks like Bob was doing whatever it took to shut the compiler up. I don't understand why it worked for the original class, though, and crashes now. Heck, I don't even understand why the compiler can't do the implicit conversion."

"The crash you are experiencing is the very reason for the prohibition," the Guru brushed a lock of silvering hair behind her ear. "Your derived class inherits virtually from the base class." My deer-in-the-headlights stare quickly let her know that I didn't understand. "Consider a case of simple inheritance," she said, picked up a marker, and began writing on my whiteboard:

```
class parent { /* whatever, including at least one virtual
   function */ };
class child : public parent { /* whatever */ };
```

"In a typical implementation of virtual functions," she continued, "the compiler will lay out the class in memory with the pointer to the virtual function table (the vptr) first, followed by the parent class data, followed by the child class data:"

Table 3:

parent::vptr
parent data
child data

"Wait a second," I interrupted. "Aren't you always telling me that the virtual function table is not required by the Standard, and therefore it's an implementation detail?"

"That is true, my child. The Holy Standard does not require virtual function tables indeed, it does not even mention them. However, one of the goals of the blessed members of the standards committee was to codify practices of existing compilers. Allowing an implicit conversion from **child ** to **parent ** would

cause problems on many existing compilers in particular, the ones that behave as I am about to describe. If I might be allowed to continue, that is." She fixed me with a stern glare. I subsided.

"This layout means that the address pointed to by **this** can be the same for both the base class and the derived class. With our hypothetical compiler, **this** always points to the **vptr**. Member functions of the derived class can determine the address of member data by simply taking **this** and adding an offset that consists of the size of the base class data and the size of the **vptr**.

"The situation becomes more complicated, however, when you add virtual inheritance into the mix. Consider:"

```
class parent { /* whatever */ };
class child1 : public virtual parent { /* whatever */ };
class child2 : public virtual parent { /* whatever */ };
class multi : public child1, public child2 { /* whatever */
   };
```

"Clearly, the compiler cannot lay out the intermediate classes **child1** and **child2** in the same manner as it did class **child**, above, with the **parent** class data immediately followed by the **child** class data, because the virtual inheritance specifies that only one copy of the base class data should appear. The solution or rather, a solution is to put a **vptr** at the beginning of each subobject, like this:"

Table 4:

parent::vptr
parent data
child1::vptr
child1 data
child2::vptr
child2 data
multi::vptr
multi data

"Then, as the compiler invokes various member functions, it can dynamically adjust the implicit this parameter so that member functions can access the member data properly."

"Oh, I get you," I said. "When you have a multi object, a pointer to it will have slightly different values depending on whether it's pointing to a parent subobject or a child1 subobject. Say, that also explains why the intermediate class has to use the virtual keyword I always thought it was a mistake, and that multi should be the class to use virtual. So, anyway, back to the problem at

hand. This means that... um... No, I don't get you. How does this apply to pointers-to-pointers?"

"Let us," she intoned, "assign arbitrary memory addresses to these values and assume that each class contains exactly one integer, that the size of an integer and the size of a pointer are each four bytes, and that there are no padding or alignment issues. Let us also add some code to instantiate a multi class:"

```
void f()
{
  multi m;
  multi *pm = &m;
  multi **ppm = &pm;
//...
```

Table 5:

Address	Description	Value
0x1000	parent::vptr	?
0x1004	parent data	?
0x1008	child1::vptr	?
0x100c	child1 data	?
0x1010	child2::vptr	?
0x1014	child2 data	?
0x1018	multi vptr	?
0x101c	multi data	?
0x1020	pm	0x1000
0x1024	ppm	0x1020

"The compiler uses the address of the first byte of the **multi** object as the **multi** pointer, that is, **0x1000**. Now, suppose you add another couple of variables to this function:"

```
//...
  child1 * pc1 = &m;
  child1 ** ppc1 = &pc1;
//...
```

"The compiler simply takes the address of **m**, adjusts it to the correct subobject in this case, the start of the **child1** subobject – and assigns it to **pc1**. **ppc1** simply contains the address of **pc1**:"

"But suppose, now, you want to assign **ppm** to **ppc1**:"

```
//...
  ppc1=ppm; // How to handle this?
//...
```

Table 6:

Address	Description	Value
0x1028	pc1	0x1008
0x102c	ppc1	0x1028

"Suppose the compiler allowed you to assign the value contained in **ppm** directly to **ppc1**. **ppc1** would then be set to the value **0x1020** – which is the address of pm, which in turn points to the base address of multi, **0x1000**. When you dereference **ppc1**, you would expect to get a **pointer-to-child1**. However, you would actually get a **pointer-to-parent** (or **multi**, as they share the same base address in our hypothetical compiler)."

"I get it," I exclaimed. "But wait – how come it worked before?"

"With non-virtual inheritance, this particular compiler assigned the same base address to the **child** and **parent** subobjects. In essence, the bit-patterns of the two addresses happened to be identical, so the **reinterpret_cast** happened to do the right thing."

"Great," I replied. "I understand the theory. Now how do I fix the problem?"

"Remember these words: 'Every problem can be solved with an extra layer of indirection.' In this case, my child, we have too many layers of indirection."

"Too many layers," I pondered. Then, the penny dropped. "Remove a layer of indirection!" I scribbled some code on the whiteboard:

```
void
childPtrPtr(child **d)
{
  parent *pp = *d;
  parentPtrPtr(&pp);
}
```

"Very good, my child. The adjustment to the this parameter occurs when you assign from the child* to the parent*. The parentPtrPtr function now receives a true pointer-to-pointer-to-parent."

"And just in time, too," I said, as I spied Anna approaching. "Here comes my lunch date." The Guru made a graceful exit. Just then, Bob wandered up, on his way to get a fresh latte. He greeted Anna.

"Hey, sugar, what brings you in? Are we having lunch today?"

"Maybe next time, Dad. I'm going to lunch with my boyfriend," Anna pointed at me.

I stared at Bob, aghast. He stared back. Both of us turned to Anna, blurting "He's your what?!"

- - - - -

"He's our what?" I blinked.

"Our controller," Jeannine waved at a sergeant who was coming out to join us. "We'll have to report through him."

"Hello, Doctor Carruthers," the sergeant said to Jeannine. "This is –?" Jeannine nodded and introduced me. "Your suits are here inside,"the sergeant said, leading us in. "The protocol is that you'll always have to wear them down there with helmets closed, even in pressurized rooms."

Inside the obelisk, the lighted area turned out to be a medium-sized room with suits on the walls, and in the center a large hole ringed with a mesh fence. "Put your hand there to sign in," the sergeant instructed; "then let's get those suits on and checked." As we did so, a platform ascended from inside the hole and stopped level with the floor; we entered through a gate in the fence, paused, and then the elevator started to descend.

11 Abstract Factory, Template Style

"Do we know what it is?" I gestured at the device lying on the cold metal table before us. It had been one of the first to have been retrieved by the recon parties exploring the immense building buried below us in the ice.

Jeannine shook her head. "Nyet, tovarisch. It could be a toaster, a hyperdrive motor, a child's toy, or just junk. Last week we thought we had the power system worked out, for this and the other devices, but when we ran the juice through them nothing happened. The techs are back to the drawing board as far as the power goes, but they're hopeful that they'll get it right soon."

"And what makes them think they will?"

"Hope."

I glanced at the clock on the wall of the chamber. "Well, it's nearly time. I'm beat, and I'm freezing. Every time we ask the recon teams to give us something new to work on, they produce something new and different. I just wish we could understand even one of these."

"I wish we knew more about what was coming out of our little historical 'artifact factory' here," Jeannine agreed.

"That reminds me of something that happened to me back when–"

Jeannine rolled her eyes. "Tell me when we get someplace warm."

- - - - -

"Bad news," I called glumly over the cubicle wall to Wendy.

"Wazzup?" she gophered up.

"You know that class I worked on last week?"

"Not intimately, but do go on."

I ignored that. "Well," I continued, "I have to create a class factory for it. The Guru suggested I look at the class factory the client group wrote."

"Yeah, so?" Then a sudden look of dismay crossed her face: "Oh. Bob wrote that, didn't he?"

I nodded glumly. "Well, one thing I'll give him credit for – every time I crack open his code, I learn how not to do something." Wendy giggled and sank back into her chair.

I sighed and checked out the source code. It wasn't quite as bad as I had anticipated. There was only one humongous series of cascading if statements – I had expected a lot worse. Still, it was pretty hairy code, so with the Guru's blessing I set about implementing an Abstract Class Factory [1]. Absent any

[1] Gamma, Helm, Johnson, Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software (Addison Wesley, 1995)

requirements for multithreading or other concurrency, I decided to implement the factory as a singleton:

```
class BaseFactory
{
    typedef std::auto_ptr<Base> (*BaseCreateFn)();
    typedef std::map<std::string, BaseCreateFn> FnRegistry;
    FnRegistry registry;

    BaseFactory() {}
    BaseFactory(const BaseFactory &); // Not implemented
    BaseFactory & operator=(const BaseFactory &); // Not
       implemented

public:
    static BaseFactory & instance() { static BaseFactory bf;
       return bf; }

    bool RegCreateFn(const std::string &, BaseCreateFn);

    std::auto_ptr<Base> Create(const std::string &) const;
};

bool BaseFactory::RegCreateFn(const std::string & className,
   BaseCreateFn fn)
{
    registry[className] = fn;
    return true;
}

std::auto_ptr<Base> BaseFactory::Create(const std::string &
   className) const
{
    std::auto_ptr<Base> theObject(0);
    FnRegistry::const_iterator regEntry = registry.find(
       className);
    if (regEntry != registry.end())
    {
        theObject = regEntry->second();
    }
    return theObject;
}
```

In the Base implementation file, I added:

```
namespace
{
    std::auto_ptr<Base> CreateBase()
    {
        return std::auto_ptr<Base>(new Base);
```

```
    }

    bool dummy = BaseFactory::instance().RegCreateFn("Base",
        CreateBase);
}
```

"OK, that's cool," I thought. "You register a function with the factory, and creating an instance is as simple as one, two, three:"

```
int main()
{
    std::auto_ptr<Base> anObject = BaseFactory::instance().
        Create("Base");
}
```

I then proceeded to create a derived class, to test creating it via the factory. In the derived class's implementation file, I added:

```

namespace
{
    std::auto_ptr<Derived> CreateDerived()
    {
        return std::auto_ptr<Derived>(new Derived);
    }

    bool dummy = BaseFactory::instance().RegCreateFn("Derived
        ", CreateDerived);
}
```

But the compiler stopped me cold – it complained that it couldn't convert CreateDerived to the correct type. After puzzling over it for a moment, I remembered the problem I had experienced previously with implicit conversions to pointers-to-Base [1]. I realized I had just run into another situation where the compiler couldn't implicitly convert the pointers, so I had to rewrite the create function slightly:

```
namespace
{
    std::auto_ptr<Base> CreateDerived()
    {
        return std::auto_ptr<Base>(new Derived);
    }

    bool dummy = BaseFactory::instance().RegCreateFn("Derived
        ", CreateDerived);
```

[1] Jim Hyslop and Herb Sutter. "Conversations: Roots," C/C++ Users Journal C++ Experts Forum, May 2001, http://www.cuj.com/experts/1905/hyslop.htm

```
}
```

Looking back at the base class, I realized that the registration code was almost identical, so I created a macro to wrap it up:

```
#define REGISTER_CLASS(BASE_CLASS, DERIVED_CLASS) \
namespace \
{ \
    std::auto_ptr<BASE_CLASS> Create##DERIVED_CLASS() \
    { \
        return std::auto_ptr<BASE_CLASS>(new DERIVED_CLASS); \
    } \
    bool dummy=BaseFactory::instance().RegCreateFn( \
        #DERIVED_CLASS, Create##DERIVED_CLASS); \
}
```

Using the macro was quite simple:

```
REGISTER_CLASS(Base, Base)
```

"Not bad for a few hours' work," I muttered when I was done.

"Yes, indeed, my child. But you can do better."

I jumped yet again at the Guru's soft voice behind me. "Wha–?"

"Macro expansions like that," she explained, "are difficult to read and understand, and macros are not type safe. Also, what is the purpose of the bool dummy variable?"

"Well," I said, slightly defensively, "it was the only way I could think of to ensure that the registration function got called automatically. I wanted to get away from the typical factory code that requires a modification each time you add a new class."

She inclined her head. "Wise thinking, my child. However, your factory is not very generic. We are dealing with more and more classes that require abstract factories. I want you to create a factory that will handle, not require, modification for each class hierarchy it will be used on."

"A generic abstract factory? That's a pretty tall order, isn't it?"

"My young apprentice, you are not afraid of a challenge, are you? Experience is by industry achieved... and perfected by the swift course of time," I finished [1]. "No, I'm not afraid of the challenge, but I don't want the swift course of time to turn the schedule aside."

"You are most of the way there already. All you need to do is think generically," she observed.

"Think generically... as in, templates!" I exclaimed and turned back to my monitor. A few minutes later, I glanced behind me and saw that the Guru had

[1] William Shakespeare. Two Gentlemen of Verona, I iii 22

disappeared again as silently as she had appeared. I shivered, but only slightly, and went back to work.

After a short while, I came up with a factory template:

```
template <class ManufacturedType, typename ClassIDKey=std::
    string>
class GenericFactory
{
    typedef std::auto_ptr<ManufacturedType> (*BaseCreateFn)()
        ;
    typedef std::map<ClassIDKey, BaseCreateFn> FnRegistry;
    FnRegistry registry;

    GenericFactory();
    GenericFactory(const GenericFactory&); // Not implemented
    GenericFactory &operator=(const GenericFactory&); // Not
        implemented

public:
    static GenericFactory &instance();

    void RegCreateFn(const ClassIDKey &, BaseCreateFn);

    std::auto_ptr<ManufacturedType> Create(const ClassIDKey &
        className) const;
};
```

I realized that not all classes would necessarily need a std::string as the key, so I made the key type a parameter to the template. The implementation of each function was the same as I had used for BaseFactory, except for the registration function. I came up with a more elegant template for that:

```
template <class AncestorType, class ManufacturedType,
    typename ClassIDKey=std::string>
class RegisterInFactory
{
public:
    static std::auto_ptr<AncestorType> CreateInstance()
    {
        return std::auto_ptr<AncestorType>(new
            ManufacturedType);
    }

    RegisterInFactory(const ClassIDKey &id)

    {
        GenericFactory<AncestorType>::instance().RegCreateFn(
            id, CreateInstance);
    }
```

```
};
```

Now, each class derived from the base class simply had to add one line to get a type-safe registration of the creation function:

```
RegisterInFactory<Base, Base> registerMe("Base");
```

The RegisterInFactory template's constructor registers the name of the class in the creation registry [1].

I glanced behind me, just in time to see the Guru approaching. I smiled to myself; for once I was slightly ahead of her game. "Very good, my apprentice," she said, coming up behind me. "Your factory is generic, portable, and does not rely on registration tables, DLL sentries, or other cumbersome techniques."

"There is one drawback, though," I interrupted. "Because the registration relies on a static object being initialized, there is no guarantee that all the creation functions will be registered if you use the factory before main begins execution."

"Exactly, my apprentice, your factory is subject to the Static Initialization Order Fiasco, as explained by the prophet Cline [2]." She turned to leave.

"Wait a second," I called. The Guru turned back. I was trying to figure out a graceful way of asking the question that was on my mind. The Guru patiently tucked a lock of her hair behind her ear while she waited. "About Bob. If he's such a bad programmer," I managed to say, "how come. . . "

"Why is he still here?" the Guru supplied. I nodded. The Guru looked down in thought. "Have you noticed that Bob hangs out with the execs? Senior management thinks he's some sort of golden boy and can do no wrong. Because of our. . . um. . . past, Bob has them convinced that any of my complaints about his job performance are just the petty grievances of a bitter ex-wife. And, according to him, anyone who agrees with me has been drawn into my little plots against him – his words, not mine."

"Man, that must be tough. Why do you put up with it? You could find a new job like that." I snapped my fingers.

She shrugged. "Other than Bob, I like it here. Besides, if I left, I wouldn't get the satisfaction of bugging the heck out of him with this act." We both laughed. "Seriously, though," she continued, "this company has a lot of opportunities for career growth. For example, our new biomedical device division is going to need a head of software development – I've applied for the position."

"Cool! Good luck," I said. The Guru nodded her thanks. I turned back to my workstation, to put the finishing touches on the abstract factory template.

[1] The complete factory file, along with a small driver program, is available on the CUJ website at hyslop.zip

[2] Marshall Cline. C++ FAQ-Lite, http://www.parashift.com/c++-faq-lite/

It was late at night a few days later when the news came, but I didn't find out about it until I got up the next morning. Still bleary-eyed, I was walking into the mess when the unusual level of noise made me put my hands over my ears and wince. "What's with all the chatter?" I muttered grumpily at the dozen people who were already there, talking excitedly. "It's too early for you guys to be this perky."

It was Major Gilb who answered, smiling. "We think we've got it, my boy, we think we do."

"What?" I grumped through mental cobwebs.

"Looks like the power's on," Jeannine said simply.

12 How to Persist an Object

It was 00:10 local Europa Base time when I slumped down the corridor. I was exhausted; three days of afternoon watch were getting to me.

A small beam of soft light spilled out across the cool metal corridor. It was coincidence, I think, that Major Gilb's door was slightly ajar, although the fact that the ventilation system had been suffering lately probably also had something to do with that. It was more coincidental that the Colonel was visiting the Major in the latter's office, instead of the reverse.

It was, on the other hand, certainly no coincidence that, when I heard low voices, I slowed down and listened. The sigh and hum of the ventilation made the listening more difficult, but it also concealed my presence.

"–me but that they're persistent mother's sons," the other voice, which I later found out to be the Colonel's, was saying.

There was a silence, and then Gilb's voice: "They're en route, then? How many? What's the ETA?"

"At least six ships." Pause. "Give 'em a month. It's imperative that the UN retain control of this installation until we can determine what technologies we're sitting on here. However long it takes."

There was what had to be Gilb's laughter. "The UN? You mean the Union."

"It's all the same. The Asians are persistent. We have to discourage them."

- - - - -

"Y'know, pardner, you've seemed a little distracted lately," Wendy said as she mixed her wasabi into the soy sauce, getting ready to dig into her sushi. "Everything all right?"

I concentrated on my California rolls for a minute. "It's Anna," I confessed. "Ever since I found out that she's Bob's daughter, I've been avoiding her. She

calls me every once in a while, but I haven't been returning her calls. I deleted her phone number from my PDA. I like her, but I don't know if I can handle. . . "

"Anna was at my baby shower last weekend," Wendy nodded understandingly. "She really likes you." When I didn't answer, Wendy continued: "I'm going to tell you what I told her. When I first met my husband, I was working a lot of overtime. It was one of those companies that always seems to be in deadline mode, you know? Anyway, Tom kept asking me out, and I wanted to, but I always had to work. Finally, I decided to make time. And the rest, as they say, is history. Tom kept trying, and so I told Anna to keep trying. Persistence pays off."

"Agh! Don't talk to me about persistence!" I grumbled.

Wendy blinked. "Huh?"

"Oh," I shook my head, "I'm trying to implement some simple object persistence in my class hierarchy. I haven't been able to figure out how to do it. I asked the Guru for some help, but she said that she had taught me everything I need to know. I just can't figure it out."

Wendy pursed her lips in thought. "This one time I'll waive our agreement about no shop talk over lunch. Fill me in a little more."

"I know it's not that difficult," I said around a mouthful of rice. "Once I get going, I know I'll be fine. The main application can instantiate a derived class. I need to be able to write that class to an **ostream**. No problem. But I'm stuck on how to reconstitute the correct class when I read it back in."

"The Guru's right – you do know everything you need to know. How do you create an object, when you don't know the exact type of the object until run time?"

"Use a class factory," I said.

"Right, just like the one you implemented last month [1]. Now, what do you pass to the factory to get the appropriate type?"

"Some sort of token. I used a **std::string**."

"OK, now where do you get that token?"

"Ummm. . . well, it isn't going to appear out of thin air, so I guess I read it from the file." At that moment, the penny dropped. "Oh, of course. When I write out the object, first I write out the token indicating what type of object it is, and then I write the object's information. Reconstituting it is a simple matter of reading the token, passing it to the factory, and then streaming it in from the file."

"Right," Wendy replied. "And if you want to get really fancy, you can. . . ."

"Hold it!" I signaled a time-out. "Thanks for helping me clear my mental

[1] Jim Hyslop and Herb Sutter. "Conversations: A Factory, Template Style," C/C++ Users Journal C++ Experts Forum, June 2001

log jam. Now that it's clear, I'm re-invoking the 'no-shop-talk' rule." I dug into the main course.

Later, when Wendy and I got back to the office, I sat at my computer and hammered out the base class fairly quickly:

```
class Base
{

private:
    // ... whatever data ...

    virtual std::string classID() const { return "Base"; }

protected:
    // Derived classes should call their parent's
    // implementation of this function after they
    // load themselves.
    virtual void do_read( std::istream& );

    // Derived classes should call their parent's
    // implementation of this function after they
    // save themselves.
    virtual void do_write( std::ostream& ) const;

public:
    // ... whatever virtual functions are required ...

    // Streaming functions.
    void read( std::istream& );
    void write( std::ostream& ) const;

    virtual ~Base();
};

// Streaming helpers. Note that they are not friends!
std::ostream& operator <<
    ( std::ostream& o, const Base& b) { b.write(o); }
std::istream& operator >>
    ( std::istream& o, Base& b) { b.read(o); }
// \footnote{A full, working example program is available to
   download from hyslop.zip.}
```

"Very good, my child," I heard the Guru's voice behind me. As I jumped, I cursed at myself for not predicting her arrival. She continued: "Give me a quick walk-through of your writings."

"Simple enough," I said, as the Guru sat in the guest chair. "Base::write is called via the overloaded insertion operator. Here's its implementation:"

```
void Base::write( std::ostream& o ) const
```

```
{
    o << classID() << std::endl;
    do_write(o);
}
```

"Ah, the Template Method pattern," the Guru nodded. "Very wise, my child[1]."

"The implementation is fairly simple," I continued. "It simply writes out the class identifier and then invokes the **do_write** virtual function. **do_write** handles the gory details of writing the object's information to the stream. Each derived class must call its parent's **do_write** function. Reading is slightly more complicated, though." I called up the function I had written to encapsulate the reading:

```
std::auto_ptr<Base> loadBase( std::istream& inFile )
{
    std::string className;
    std::getline( inFile, className );

    std::auto_ptr<Base> newBase =
        genericFactory<Base>::instance().create(className);

    if( newBase.get() != 0 ) {
        inFile >> *newBase;
    }

    return newBase;
}
```

"It's a three-step process. First, I retrieve the class ID from the file. Then I use that identifier to get a new object from the class factory, and finally I reconstitute the class's data, via the **read** function:"

```
void Base::read( std::istream& i )
{
    do_read(i);
}
```

"I'm using the Template Method pattern here, too, but that's probably overkill for a simple function like this. **do_read**, of course, handles the gory details of reading in the information, validating it, and so on. Like the **do_write** function, a derived class's **do_read** should call its parent's **do_read** function."

"Well done, my child," the Guru said as she stood up.

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995).

"Before you go," I blurted out. The Guru paused. "Tell Anna I've lost her phone number, but I'll be home tonight if she wants to call me." The Guru nodded and glided away.

- - - - -

"'Discourage.' Ah." It was Gilb's voice again. I blinked and shook my head to clear away the tired haze. "What do they know?"

"Not much, we think. The Asians suspect something."

"They don't believe our cover story of communications problems that happen to cut off all of their nationals from talking with them?" Gilb's voice was openly sarcastic.

"They suspect," the other voice repeated. "And they're coming. We need to deal with it. What have your tech teams discovered so far?"

I was feeling increasingly uncomfortable standing in the corridor and didn't wait to hear more. There would be time enough after a good night's sleep.

13 Hungarian wartHogs

"Breakthrough!" one of the officers called, poking her head into the mess hall.

Several people, Jeannine and I among them, looked up. "What?" "Whose breakthrough?" "What happened?"

The sudden excitement was palpable, and no wonder – things were worse than ever since the senior officers had made known we'd lost contact with the orbiting station and the main surface base. Likely, they were overrun by the incoming forces, although our officers had not yet officially admitted that we were in a hostile situation. When the news came, we had been sealed into the local base near the excavation site for two days already. And progress on the alien artifacts had continued to be maddeningly slow, knowing how near we must be to a breakthrough. Jeannine seemed closer than ever to figuring out the power requirements.

"Jöger says he's figured out part of the language, that's what," she informed us. The obscure writing on the ancient, and mostly defunct, equipment had been a major obstacle to progress. Until now, attempts to decipher it had been largely unsuccessful. "Something about a built-in redundancy – he's doping it out. Don't all rush in to bug him; he needs to concentrate. Thought you'd like to know." And then the officer's head disappeared as she continued elsewhere to spread the news.

"Redundant information," I sat back and mused aloud. "Does that ever sound familiar. Why, back on my first job. . . "

- - - - -

"Warts!"

"Beg pardon?" I turned at the sound of the Guru's voice.

"You have warts, my child."

Suddenly feeling rather self-conscious, I asked: "Uh, where? My medicated creams usually contr–"

"Your latest code is covered in warts!" she interrupted. "Otherwise known as Hungarian notation. Your variables are beginning to look like this." She picked up the whiteboard marker and wrote: wartHog. "In this case, the variable hog's wart has nothing to do with the pottery school [1]."

"Oh, that," I breathed a sigh of relief. "Hungarian notation? Is that all? Sure, yeah, I read a cool article about Hungarian notation, and it sounded like a good idea. Apparently it was quite the trendy thing for a while. Almost kind of a type calculus-ish direction, sort of in a way, you know. So I–"

"Gibberish!" the Guru exclaimed.

"Ah, well," I faltered. "I thought I said that rather clearly."

"Not you, my child," the Guru corrected. "The naming convention you chose to experiment with results in code that looks like gibberish. Even the prophet Petzold had his off days."

"Hmmmph," I hmmmphed, only slightly mollified. "Well, it's supposed to make code easier to write and read. For example, I can catch some type errors just by reading the code. If I write something like `strcpy( szDestination, pachSource);` I can see a problem immediately – the second parameter is a pointer to an array of characters, but not necessarily null-terminated as strcpy requires. Or if I write `printf("%s", ulValue );` , I can see that I'm passing an unsigned long where a null-terminated string ought to go. The code tells me that I'm doing thewrong thing."

"Of dubious practical advantage even for type-unsafe calls in C, or in environments when nearly everything is a type-challenged int or void* handle," she shook her head sadly, "and of no relevance at all in a type-safe object-oriented language like C++. This, you must unlearn. Warts are not information; they are disinformation.

"Consider your own parables. In your first example, in C++ you would use strings, which are objects, and problems like the first cannot arise because a

[1]The first reader to correctly identify the source of this oblique reference, as well as why it's appropriate for this November 2001 column, will receive an autographed copy of Sutter's More Exceptional C++ when it is available in November 2001. All submissions must be sent by email to `hsutter@acm.org` with the subject line "I know! I know!" and must include a valid (non-munged) return email address and snail-mail postal address. Contest closes at midnight on October 15, 2001.

string's semantics are well defined and encapsulated. One who writes string destination = source; just cannot get the buffer-copying semantics wrong, for they are never exposed and the calling programmer never required to assist them; the implementation details are well and truly encapsulated and always managed for you. As for your second example, in C++ you would normally use streams or other type-safe methods to write output, and type checking along with overload resolution guarantee a type-correct result even if a conversion is needed. One who writes `cout << value;`

just cannot get the type wrong: if value is an unsigned long, the `operator`<< for unsigned longs will be invoked; or, if value has a type for which no `operator` << is defined but value's type can be converted unambiguously to a type for which `operator`<< is defined, that `operator`<< will be invoked; otherwise, a compile-time error occurs. Not only does the C++ language and standard library detect what would be run-time type errors with other type-unsafe calls, it turns them into compile-time errors.

"In sum, the compiler already knows much more than you do about an object's type. Changing the variable's name to embed type information adds little value and is in fact brittle. And if there ever was reason to use some Hungarian notation in C-style languages, which is debatable, there certainly remains no value when using type-safe languages."

"Maybe so," I acquiesced. If nothing else, she had out-soliloquized me. "But you have to admit that, once you know the rules, using Hungarian makes it easier to create variable names."

"So certain are you?" the Guru arched her eyebrows. She placed herself in the guest chair. I knew what that meant: she was warming up for a debate. "I had that in mind when I said 'brittle.' How is it easier, say you?"

"Because the types tell you what to call the variables. It's almost mechanical," I said. "The variable name is pretty much generated from its type. Once you know the type, you can generate the name," I rambled, then realized I was rambling, and stopped. When I had begun, I had somehow had the sense there would be something much more profound to say; now that I heard the words coming out of my mouth, it all sounded a bit superficial. I remember wondering why, when it had seemed so much deeper the first time.

"And if the type changes. . . ?" she prompted me just then.

"Well, you'd have to change the variable name, I guess. But!" I exclaimed, "That then forces you to examine each usage of the variable in your program, to ensure that it is still being used properly."

"Not a bit of it," she riposted. "It 'forces' you to do no such thing. In many cases, the programmers will simply forget – or worse, not bother – to consistently and globally change the variable name, never mind check the usage. And once they do not change the name, the code is lying to you, thus violating

the commandment that says: 'Speak truth each one of you with his cubicle neighbor.' This," she shook her head quietly, "is reprehensible. Such is the disinformational evil that must of necessity follow, sooner or later but probably sooner, from the deceptive wartHog style."

"Ah," I smiled, "that's what global search-and-replace is for!"

"Most assuredly not," she shook her head again. "Never mind that you could inadvertently change other similar names of objects whose types have not changed! But even if the global replacement were done correctly, what value has it added? For it leaves the programmer no better off for his troubles than he would have been otherwise. If an error has been introduced by the change in type, such as because of implicit conversions, the error remains the same regardless of the object's superficial name, and you have merely added the menial and meaningless documentation work of changing the name. This too is vanity and a striving after wind."

It was time for me to fall back and regroup. "But you'd have the error no matter what naming system you used. If I have int count; and I change it to short count;, then many programmers might not bother to check the usage at all and just hope the compiler catches any range problems."

"That," the Guru acknowledged, "is what I just said. The problem is the same whether you uglify the variable's name or not, and by uglifying it you have merely added useless maintenance work because then you must additionally maintain the warty name. If you are not yet convinced, my child, I have one small question for you now: How would you apply Hungarian notation to templates?"

That stumped me. "Touch, I guess," I acknowledged. "A template doesn't really have a type of its own, because the template generates an unknown number of types, one for each set of parameters it's instantiated with. There's no type until it's instantiated, so you can't really create a variable name that encodes the type of the template itself."

"Well spoken, well said. Even more," the Guru added, "inside the template definition itself, how would you wartify the names of objects of a template parameter type? You do not know what they are."

"Oh, I see," I said. "You mean like this." On the whiteboard, I scribbled an offhand example:

```
template<typename T>
T AddOne( T wartT ) // what wart should wart be?
                    // papuch? lpsz? huh?
                    // (handle to unbounded harm)
{
 return wartT + T(1);
}
```

"I believe source code is a form of communication," the Guru pressed on. "The question is, who is that communication aimed at? The compiler? No. Source code is a medium of communication from one programmer to another. It is an expression of intent, of what is desired to happen. We must strive to keep that communication as simple and clear as possible. In order to do that, variable names should reflect the roles that those variables play. The exact type is secondary to the role. A variable name such as sz tells you only that you are looking at a C-style string. It conveys no information as to the role that string is to play."

"Well," I put in, "I don't think anyone would use just sz for a variable name."

"No? If we are to discuss Hungarian notation, we should discuss the canonical version presented by Dr. Simonyi [1]. His examples use variables named sz, pch, and so on. Such names, alas, present no useful information. If the variables were instead called xyzzy and yeti, respectively, or even merely x and y, I would still know simply by looking at their declarations that the first indicates an array of characters, and the second is a pointer to a character. Calling them sz and pch adds no useful information not already present in the code, and in particular, it adds no information not already well known to the compiler. Worse, it could be a lie if the type has changed since the wart was chosen. In any event, even if the wart lies not, the questions are still: What are the variables? What are they for? What do they do? How are they used? The wart helps not at all. I have to study the code to understand the roles they play. The code fails to communicate the intent of the programmer; therefore the names sz and pch are poor choices."

"In your opinion," I ventured.

"In my opinion," the Guru nodded. "Yes, this is still somewhat an area of opinion, rather than hard fact alone, although fact it is that names such as sz are next to useless and that Hungarian warts are brittle in the face of change. When we drafted the coding style guidelines several years ago, Hungarian notation was but one of many areas of lively debate. Unlike some discussions, this one was actually reasonably civilized. We examined the naming convention, listened to the experiences of those who had used it, and eventually came to the consensus that we did not like the convention. Although we did not dislike it enough to prohibit its use outright, we did consider it brittle enough to actively discourage it. Hence, to quote this team's standards:

"Avoid Hungarian notation. It will make a liar out of you.

Warts are not information, but disinformation."

"Hungarian is not only mendacious, but it is high-cholesterol; it is suspected

[1] harles Simonyi. "Hungarian notation." Reprinted at http://msdn.microsoft.com/library/en-us/dnvsgen/html/hunganotat.asp. Readers are encouraged to study the naming convention and decide for themselves, rather than rely on the opinions of the authors.

of being fattening, and it is in all probability a flagstone on the road leading to a wasted and dissolute life. Indeed, I recall only one time when Hungarian notation was useful on a project."

This hook intrigued me. "What was that?"

"One of the programmers on the project was named Paul," the Guru explained. "Several months into the project, while still struggling to grow a pony-tail and build his report-writing module, he pointed out that Hungarian notation had helped him find a sense of identity, for he now knew what he was..." She paused.

I blinked. It took me about ten seconds, and then I shut my eyes and grimaced painfully. "Pointer to array of unsigned long," I groaned.

She smiled, enjoying my pain. "True story," she said [1].

It was then that I thought I had found a way to corner her in an inconsistency. "Well," I asked innocently, having quickly recovered from the awful pun, "what about our convention of using a trailing underscore suffix for member variables? Isn't that kind of a watered-down version of Hungarian?"

The Guru smiled pleasantly. "So it may appear to the uninitiated, but appearances are in this case deceiving. The underscore has nothing to do with type – it has to do with flagging scope and privacy. It also has a small practical benefit: we realized that passing a parameter to a member function – particularly an initialization function or a constructor – we would often want to use the same name for both the member variable, and for the passed parameter. For example:

```
class T
{
 int count_;
public:
 T() : count_(0) {}
 void init(int count) { count_=count; }
};
```

"We are giving the init function a count. Why come up with a different name for the parameter, when both the parameter and the member play the same role? We actually discussed several options for such cases: prefixing member variable names with my (or with our for static data), prefixing the parameter name with given, and so on. In the end, we decided to adopt a trailing underscore for the member, and no underscore for the provided parameter."

I frowned. "Isn't this convention a little debatable?"

The Guru grinned wryly. "Wendy's personal preference is the other way around. But then no one is perfect, and she did just produce a most exceptionally beautiful child in Jeannine [2], so one must make allowances."

[1] Indeed a true story that happened to one of the authors.

[2] Not the same Jeannine as in the framing story. See *Conversations: Back To Base-ics,*

14 New Bases, Part 1

Things were very quiet in the Ballroom.

That wasn't entirely unexpected; after all, we had no idea how we would ever leave it. It was the second day after the attacking forces had invaded our position inside the alien city beneath the Europan ice. With no help within reach from our own political faction on Earth, Gilb ordered retreat into the Ballroom and we barricaded the doors. The room was immense, but fortunately had few entrances. A few of our people had left us and surrendered in the disorder of the retreat; Jeannine and I didn't get the chance, for we were watched too closely by Gilb and his officers.

It was cold in the Ballroom, and getting colder. Gilb authorized turning on the two portable heaters, since we had more energy cells with us than our group was likely to use in a year. It helped.

We also had with us several of the alien artifacts, now powered but still mysterious, for we had only just managed to activate them and had had no time to experiment. Jäger was with us, which was something; he knew the mysterious city's language better than anyone.

The three 10-foot-diameter balls still hung suspended above the floor. Each still showed a different scene: the field of yellowish grasses with something like trees nearby; the dusk beach scene; and the metallic room much like our own, with the cabinets, where we had seen the gray-clad human who had started at seeing us and then had run away. He had not returned, nor had anyone else. We knew that throwing small objects into the balls put them into the scenes; a spent energy cell thrown through the meadow gate – we were beginning to think of the balls as "gates" now – pushed through the growths and bounced on the grassy ground before coming to rest. The energy cell was still visible, and nothing had jumped on it or otherwise reacted to the experiment.

I turned to Jeannine, and we spoke of other cold and quiet times. It helped to distract us, while Jäger tried to make himself more useful at the other end of the room by continuing to experiment carefully with one of the artifacts.

- - - - -

Things were very quiet around the office.

That wasn't entirely unexpected; after all, it was January 2, and many people had taken the day off. I was puttering around with some small bits of programming on my to-do list, because my mind wasn't ready for anything big.

C/C++ Users Journal C++ Experts Forum, September 2001, www.cuj.com/experts/1909/hyslop.htm.

I had recently started getting "internal library" development requests, which was a real thrill for me even though the tasks were small. You see, our company didn't have an officially separate team whose job was to build software library infrastructure for the whole company, which I thought was a good thing. That organization style never made sense to me, nor I think to the Guru. But we did have a set of internal libraries for common things that were shared by several projects. The internal libraries had no single "owner," but items to be added or changed were reviewed by one of the more experienced developers. If we wrote something that was a likely candidate for reuse, we'd submit it to the shared libraries. Less often, someone would request a feature that wasn't in the shared area yet but probably should be, and someone capable would pick up the request and write it.

So I was thrilled because being asked to write an addition to the internal library branded me as "possibly capable." I was also thrilled because Bob wasn't ever asked, and this knowledge gave me a little ego boost. Ah, the vanity of youth.

The problem seemed easy: "Write a **ConvertBase** function that takes a **string** representing a number in base N and converts it to a **string** representing the same number in base M."

My first reaction was to write a hard-coded solution, something like this:

```
string ConvertBase( size_t base1, size_t base2,
                    const string& src )
{
  long value = 0;

  // Read a base base1 number from src
  for( int i = 0; i < src.size(); ++i )
  {
    // if src[i] is a valid base1 character,
    //  multiply value by base1 and add the next digit
    // else if src[i] is whitespace
    //  break
    // else
    //  throw logic_error
  }

  // Write a base base2 number to dest
  string dest;
  while( value > 0 )
  {
    // find highest base2 digit
    // append to dest
    // reduce value appropriately
  }
```

```
}
```

I had roughed in about that much and was still typing the last comment when a gentle throat-clearing behind me alerted me to the Guru's presence.

I froze. There was silence.

Then I heard a brief rustle of pages that I knew without looking was the Guru putting a marker into her book and closing it, and a calm low voice: "Goodeve, my child."

"Ah, goodeve, master," I replied. If she was in her Guru shtick, then Bob had to be nearby, and appearances were important to successfully bugging Bob.

"Your code," she gestured, walking around me into view and tapping on my display with her pen, "bears a striking resemblance to library request 247."

"Ah, yes, I hope it does," I acknowledged. "That's what I was working on. It looks like it's going to get kind of longish."

She nodded thoughtfully. There was another pause. After a moment she added: "Did you happen to notice library request 314?"

"Ah, er. . . no," I admitted.

"No harm done. Request 314 asks for a facility to read and write a number of arbitrary base from a text stream. I presume that whoever codes 314 will end up writing much the same code as you were about to write there. May I make a suggestion, my apprentice?"

"Sure. You want me to do 314 too?" I asked, hope rising. If it was an honor to be asked to write one shared library facility, imagine the honor of being asked to write two after such a short time on the team!

"Yes, but not immediately," said the Guru, arranging a wisp of graying hair behind her ear. "A principle of programming is that we often gain useful insights into our code by writing the test cases first. Beck, Fowler, Martin, and others preach this gospel to anyone who will listen and to many who will not. In this case, why not write the code for request 247 assuming that you already had the code for 314? Then write a simple test case for 247 and hard code just enough of a stub for 314 to make that test case compile. Finally, when you see in your 247 code the form that the solution to 314 should take, confirm that that form is acceptable to the people requesting 314, and do 314 too."

I blinked. "I think I got that. So I'll take a first cut at 247 that uses 314's not-yet-written facility, which will let me see what 314's facility ought to look like. Right?"

"Indeed."

"Okay." I thought for a moment and then started typing. I could assume that I had "a facility to read and write a number of arbitrary base from a text stream." All right. If I had that, I could use streams to translate between bases. After further thought, I eventually came up with this code:

```
string ConvertBase( size_t base1, size_t base2,
                    const string& src )
{
  stringstream s1( src );
  long value;

  if( !( s1 >> Num( base1, value ) ) ||
      !( s1 >> std::ws ).eof() )
    throw logic_error
      ( "src is not a valid number" );

  stringstream s2;
  if( !( s2 << Num( base2, value ) ) )
    throw logic_error
      ( "unexpected error emitting converted number" );

  return s2.str();
}
```

Then I wrote a simple test case, with just enough of a **Num** stub hardwired to make the case compile:

```
class Num
{
public:
  Num( size_t base, long& value ) :
    base_(base), value_(value) { }

  size_t Base() const { return base_;  }
  long& Value()       { return value_; }

private:
  size_t base_;
  long& value_;
};

istream& operator>>( istream& i, Num& n )
{
  string s;
  i >> s;
  // todo: really convert input to n's base_
  n.Value() = 255;
  return i;
}

ostream& operator<<( ostream& o, const Num& /* n */ )
{
  // todo: really output n.value_ in n.base_
  return o << "FF";
```

```
}

int main()
{
  string result = ConvertBase( 10, 16, "255" );
  cout << result << endl;
  return result == "FF" ? 0 : 1;
}
```

After the compiler caught a few typos, I was satisfied that this skeletal attempt worked. Next, I knew I would have to more fully address request 314 by actually implementing << and >> for **Nums**.

I looked at my watch; it was well past quitting time, but I could still make my train and therefore my date with Anna. Request 314 could wait till the morning.

- - - - -

I looked at my watch; it was well past time for our turn to sleep, so we took the opportunity. There was periodic banging on the other side of one of the doors, presumably from the invading force, but that was all – so far. Unless something more happened, the strange balls, or gates, could wait till the morning. Although Gilb's people were still trying to use our portable comms gear to get a message to Earth, or to ships in transit I didn't know about, I had a feeling that the gates would be our only way out.

15 New Bases, Part2

The change in control came suddenly.

Gilb had been watching the main Ballroom door, where we had heard occasional pounding. The attack, when it came, was through the two side doors and caught us while most of us were asleep.

I woke to the sound of loud banging and the glare from the intense points of light moving across the two opposing doors. It was immediately obvious that the enemy were cutting their way in, and quickly. Gilb's orders had been explicit – in case of attack, grab our packs and alien artifacts and jump through the forest gate, and that's what we tried to do.

Gilb and several of his officers made it through and quickly disappeared out of sight in the other world. Most of us were still racing for the ball-gate as the doors crumpled and armor-clad soldiers started pouring in toward us from two sides, shouting orders to stop. I was near the gate, ahead of Jeannine, and hesitated . . . then shots were fired, and I jumped forward.

I felt nothing unusual as I passed through the gate, except that my suit's external temperature reading jumped to 28C. I took a quick glance backward – still in the Ballroom, Jeannine was dropping her pack and holding out her arms. I didn't wait to watch her get captured; I hurried after Gilb's party into the new world's underbrush.

- - - - -

The change in management came suddenly.

Because neither our company nor our acquirer were publicly traded, the boards were able to negotiate the deal quietly and keep it under wraps, even from the employees, until just a couple of weeks before the actual closing date. So it was that shortly after New Year's Day we first heard the announcement of the merger (such a polite term for "acquisition" or "buyout" given that we were much smaller and ailing than our buyer) and within two weeks found ourselves already working for the new boss.

Pete Williams seemed like a reasonable fellow at first, even if he was a bit too given to sports metaphors for a manager of a technical team. I would have more of a chance to refine that opinion as time went on, and I gained experience working under him. I did approve of Pete's first action, or perhaps inaction: he kept our team together and didn't monkey much with the way we worked, at first. That was mostly a good thing. Except for Bob, who had somehow thought he would get a promotion out of this, we were all pleased with this turn of events.

A few urgent assignments had taken me away from my lower-priority internal library-building work (not that the internal library wasn't important, but none of the requests assigned to me were, for I was only just starting out). When I next had a breather, I returned to the half-written library code. I had been filling two requests: Request 247 asked for "a ConvertBase function that takes a string representing a number in base N and converts it to a string representing the same number in base M." The Guru had pointed out that Request 314 was for "a facility to read and write a number of arbitrary base from a text stream." Thus guided, I had written the following to try to implement the former in terms of the latter and avoid redundant work:

```
string ConvertBase( size_t base1, size_t base2,
                    const string& src )
{
  stringstream s1( src );
  long value;

  if( !( s1 >> Num( base1, value )).eof() ||
      !( s1 >> std::ws ).eof() )
    throw logic_error( "src is not a valid number" );
```

```
  stringstream s2;
  if( !( s2 << Num( base2, value ) ) )
    throw logic_error( "unexpected error emitting "
                       "converted number" );

  return s2.str();
}
```

That seemed to do it for 247, though as it relied on 314's **Num**, it wouldn't work until I finished that. So far I had just a simple test case, with a **Num** stub hardwired to make the case compile, and while I was looking at it, I added a check to limit the base to a reasonable range:

```
class Num
{
public:
  Num( size_t base, long& value )
  : base_(base), value_(value)
  {
    if( base < 1 || base > 36 )
      throw logic_error( "base must be from "
                         "1 to 36" );
  }

  size_t Base() const { return base_;  }
  long& Value()       { return value_; }
  long  Value() const { return value_; }

private:
  size_t base_;
  long& value_;
};

istream& operator>>( istream& i, Num& n )
{
  string s;
  i >> s;
  n.Value() = 255; // todo: really convert input
                   //       to n's base_
  return i;
}

ostream& operator<<( ostream& o, const Num& /* n */ )
{
  return o << "FF"; // todo: really output
                    //       n.value_ in n.base_
}
```

```
int main()
{
  string result = ConvertBase( 10, 16, "255" );
  cout << result << endl;
  return result == "FF" ? 0 : 1;
}
```

Next, I knew I would have to finish 314 by actually implementing << and >> for Nums. As I was about to start coding, I noticed a serious deficiency in the signatures of my existing operators << and >>. I leaned closer to the screen and was nearly ready to start making changes when the telltale closing of a book behind me alerted me to the Guru's presence.

"Is Bob around?" I whispered.

"No, we can just talk," she said and sat in my visitor's chair. "What on earth possessed you to do that?" she continued, pointing at the screen.

"Yeah, I know, I was just looking at that. I've been away from this code for weeks, and now the obvious thing jumps out at me."

"Ah. Well then, counsel yourself."

I pointed at the offending operator << and >> signatures. "Aw, I accidentally hardwired them for basic char and char_traits<char> streams only."

She smiled and stood. "Very good. Carry on, apprentice," she added, as we both heard the sound of Bob's sloshing latte approaching.

"Uh, that's it?" I asked. "No longer speech, no morality play, no programming lesson?"

She arched an amused eyebrow. "And why would there be? You spotted the wicked flaw in your writings without my help, and now you need time to fix it. What more can I teach until that is done?" And without waiting for a further reply, she smiled quietly, reopened her tome, and glided silently away.

I guessed that I must be improving if she was willing to let me get away without a three-page lesson. So I went back to the code. The first thing was to fix the signatures:

```
template<class CharT, class Traits>
basic_istream<CharT, Traits>& operator>>
  ( basic_istream<CharT, Traits>& i, Num& n );

template<class CharT, class Traits>
basic_ostream<CharT, Traits>& operator<<
  ( basic_ostream<CharT, Traits>& o, const Num& n );
```

That was better. I checked it by eye, and it looked right this time. Next, I decided that input would be easiest. I began adroitly:

```
template<class CharT, class Traits>
basic_istream<CharT, Traits>& operator>>
```

```
  ( basic_istream<CharT, Traits>& input, Num& n )
{
  locale loc = input.getloc();
  CharT c;
```

Here I congratulated myself on my own presence of mind, as I avoided the twin pitfalls of reading simply **chars** and of forgetting about locales. I then remembered that leading whitespace can be optionally skipped, and so I pressed on:

```
  // If we should, then skip leading
  // whitespace, if any.
  if( input.flags() & ios_base::skipws )
  {
    do
    {
      input >> c;
    }
    while( isspace( c, loc ) );
  }

  // Handle sign, if any.
  int sign = ( c == '-' ? 1 : 1 );
  if( c == '-' || c == '+' )
    input >> c;
```

So far, so good. When it came to actually handling the digits, the logic was simple, and it was mostly a matter of deciding on a reasonable way to decipher the digits. After all, green though I was, I was knowledgeable enough to realize that I couldn't just assume that the digit and/or letter characters would be contiguous and in order in the character set (even though each group was contiguous and in order in ASCII). For the time being, I decided on a helper structure that was suitable for searching with `std::find()`:

```
 CharT digits[]
    = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 size_t numDigits
    = sizeof( digits ) / sizeof( digits[0] );

 n.Value() = 0;
 while( input.good() )
 {
    int index = find(digits, digits + numDigits,
                     toupper (c, loc) )
                digits;
    if( index < 0 || index > n.Base() )
    {
      break;
```

```
    }

    n.Value() = n.Value() * n.Base() + index;
    input >> c;
  }

  if ( !input.eof() )
    input.putback( c );

  n.Value() *= sign;
  return input;
}
```

That seemed reasonable enough for input, leaving only output. Output was easier, in part because I made a deliberate decision not to support the output formatting flags like **showpos** and **uppercase**; that could wait until a later request, if it was needed at all–

I glanced furtively over my shoulder, in case the Guru had glided up and would pounce on me, remonstrating about supporting all the standard stream flags. All was dark (it being that time of year) and quiet. There was no Guru there. I waited a few moments, but there was no sound or movement. Satisfied, I turned back to my code.

–so yes, as I was saying, I ignored the issue of formatting flags. If the Guru didn't like it, she could remonstrate after I checked it in, and I'd do a more complete job in version 2.0 of the code. If Wendy didn't like it, I would try to argue my way out of it by saying we didn't yet know if that was a requirement, so why complicate the code unnecessarily up front? If Bob didn't like it, he could go jump in the lake, particularly at this time of year.

So my first cut at a real insertion operator looked like this:

```
template<class CharT, class Traits>
basic_ostream<CharT, Traits>& operator<<
  ( basic_ostream<CharT, Traits>& o, const Num& n )
{
  long value = n.Value();
  basic_string<CharT, Traits> s;

  CharT digits []
    = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

  basic_string<CharT, Traits>::size_type start=0;
  do
  {
    s.insert( start, 1,
              digits[ value % n.Base() ] );
    value /= n.Base();
```

```
  }
  while( value > 0 );

  if( n.Value() < 0 )
    s.insert( start, 1, '-' );

  return o << s;
}
```

- - - - -

I started running the code through a more extensive set of test cases and got ready to tackle the next item on my normal project's workload. Our new manager was big on "time management," which seemed to consist of micromanaging our to-do lists. I wondered how that would work out.

Fortunately, I was ahead of my list. Perhaps today I could leave a little early for a change.

Fortunately, it was a mostly hospitable world we now found ourselves in – for it was obvious, if only from the slightly heavier gravity, that it was a different world from any we had known among Sol's children. Yes, mostly hospitable: as the few rations gave out, we found local flora that was edible, and we did not immediately come across any indigenous humans, or more to the point they did not come across us.

Gilb found out about the local fauna the hard way. We buried him a few days later and started posting sentries against the unfriendly five-legged carnivores we dubbed "Goofies" because of their passing resemblance to the cartoon character. That was when Da Rosa took over command of our small base and instituted a more rigorous testing of some of the artifacts we had determined were, or could be used as, weapons.

I worried about Jeannine.

16 Template Specializations

We watched the gate from a short distance, alert that the invaders might follow us through from the Europan base. It appeared, however, that the enemy was quite content to abandon us, or at least wait us out. The gate remained quiet.

Despite constant alertness, Goofies had got a couple of our sentries. We had killed many Goofies – sometimes with our conventional weapons, sometimes as tests of the alien artifacts, at least one of which spewed huge energy bolts that

made it dangerous to aim at anything too close nearby – but the Goofies seemed to be in unlimited supply.

So it was that we were only partly surprised by the mutiny.

- - - - -

"Hey, Junior, how's it going?" I tensed at Bob's voice. But then it struck me – Bob was not his usual condescending self. He was almost... dare I say... civil.

"Not bad, Bob," I replied, turning to face him. We stared at each other for a moment.

"Uh, say, Junior, I was wondering if I could get a little help."

I was stunned. Speechless. Bob, the worst programmer in the department; the man who delighted in writing obscure, difficult to maintain code; the man whose arrogance was so thick you could smell it a block away; the man who stopped by my cubicle only long enough to dump his problems in my lap (and latte on my desk) was standing at my cubicle, asking me for help?

"Aaah, sure, Bob," I finally answered. "What's up?"

Bob sat himself down in the guest chair. "OK, here's the scoop. I've managed to condense it down to this source code." He took over on the keyboard and called up a file on my workstation. I looked closely at it:

```
#include <iostream>

template<typename T, std::size_t size = 10>
class c
{
  T m[size];
public:
  void print_size()
  {
    std::cout << size << std::endl;
  }
};

template<> class c<char>
{
  char m[100];
public:
  void print_size()
  {
    std::cout << 100 << std::endl;
  }
};

int main()
{
```

```
  c<char>().print_size();
  c<char, 10>().print_size();
}
```

"First, I don't see how it can compile. I mean, I thought the template argument has to be a class or type name, not a... well, an ordinary parameter."

"You mean the size_t parameter?" I quizzed. Bob grunted, which I took to mean an affirmative. "Yeah, the first time I saw a non-type parameter, my brain stopped for a moment too. I was used to seeing class or typename declarations, not regular parameters. The Guru assured me that it was allowed, but she made me look it up in the Standard for myself. You can only use integral or enumeration types, pointers or references to objects or functions, or pointers to member [1]. In fact, with integral types, the parameter becomes an integral constant expression...."

"A whozer whatzit?" Bob interrupted.

"A compile-time constant," I translated from Standardese to English. "Anyway, since it's an integr– er, that is, since it's a compile-time constant, that allows you to use size to declare the array of T objects. Pretty nifty.

"Once I learned this, I was toying with the idea of using a template to provide a more flexible switch statement, something like," I opened a new text file and typed in:

```
template<int value1, int value2, int value3>
void f(int currentValue)
{
    switch(currentValue)
    {
    case value1:
        ...
    case value2:
        ...
    case value3:
        ...
    }
}
void g()
{
    f<1, 2, 3>(1); // executes value1 case
    f<2, 3, 1>(1); // executes value3 case
}
```

"I couldn't come up with a practical application for it, so I filed it in the 'nifty but no practical application for it' category..."

"Alongside other useless information that gets trotted out at parties, no doubt," Bob interrupted, glancing at his watch. "That's, uh, real interesting,

Junior, but getting back to my problem," he reached over, closed my text window, and restored his original program. "See, when I compiled and ran this, the first template prints out 100, just like I thought it would, 'cause it um... instantiates the... uh.... specialization c<char>. I expected the second template to print out 10, because I figured it'd match up the 10 with the default parameter of 10, but it prints out 100 as well."

I studied the code. Bob sipped his latte.

"Well," I welled to fill in the silence, "My first reaction would be that I'd expect both templates to print out 10."

"Y'know, that's what I told Her Holine– uh, I mean, that's what I thought at first." Ah, that was a little more like the Bob I knew – he was about to call the Guru a nasty name.

"I try to think of default parameters as 'lazy shortcuts'," I continued. "Whenever I write something that takes advantage of default parameters, I have to remember that the compiler treats it as if I had actually written the parameter."

"I know that," Bob said impatiently. He took over the keyboard and modified the program slightly:

```
int main()
{
  c<char>().print_size();
  // compiler acts as if I'd written
  // c<char, 10>().print_size();

  c<char, 10>().print_size();
}
```

"This is an exact match for the base template. In that case, though, the program would print out 10 for each size, but it prints out 100, so what's going on?" Bob asked impatiently, glancing again at his watch.

I savored this moment. Not only was Bob aware that I knew something he didn't, but he was actually acknowledging it!

"That's not the only place the default parameter is used," I finally said. Bob gave me a quizzical look. "The specialization uses it, too." I took over the keyboard, after carefully moving Bob's latte out of the way. "The specialization that you wrote as..."

```
template<> class c<char>
```

"...acts as if you wrote:"

```
template<> class c<char, 10>
```

"So," I continued, "you have not specialized on c<char>, but on c<char, 10>, where the size template parameter is an integral constant-expression with the value 10. In other words, your program acts as if you had written:"

```
#include <iostream>
template<typename T, std::size_t size = 10>
class c
{
  T m[size];
public:
  void print_size()
  {
    std::cout << size << std::endl;
  }
};

template<> class c<char, 10>
{
  char m[100];
public:
  void print_size()
  {
    std::cout << 100 << std::endl;
  }
};

int main()
{
  c<char, 10>().print_size();
  c<char, 10>().print_size();
}
```

"I get it," Bob smiled as he leaned back in his chair. "In main(), both instantiations of c instantiate the specialization." I didn't like the look of that smile on his face. He stood up, then in a louder tone of voice, he continued, "And remember, Junior, that default parameters apply to template specializations as well." I was puzzled by his sudden change. I spotted the Guru approaching, an open tome in her hand as usual.

She stopped in front of Bob. "Why are you interrupting my apprentice's devotions?" the Guru demanded.

"Hello, doll-face," Bob beamed at the Guru. "I was just explaining to Junior here, why the correct output of that program we were talking about is 100 100." He glanced at his watch. "Yep, coming up on two o'clock – I better get to Pete's office and go over this with him."

The Guru and I watched him disappear in the direction of Pete Williams's office. Pete was the new manager brought in when our company was acquired last month.

"Figured it out, did he?" the Guru murmured as she turned to me. "Now matters are worse." My stomach did a flip-flop.

"Um, what was that all about?" I asked.

The Guru sat down wearily in the guest chair. With Bob gone, she was back in "normal" mode. "You asked me once why Bob was still around, even though he's an. . . " I could see her struggling for words.

"Incompetent, arrogant, misogynistic. . . " I supplied, which the Guru capped with a word I would not care to repeat to my mother, even if she is stone deaf.

"Yes, exactly," the Guru continued, calmer now. "Bob was in quite tight with the previous upper management. He somehow had them hoodwinked into believing he's a. . . well, a golden boy. All my attempts to expose him for what he is were swept aside because he had the upper management convinced that I was still bitter about our divorce. As if!

"Anyway," she continued, "when we had our change in management, I saw a perfect opportunity. I convinced Pete Williams to pose him a challenging question, which I knew Bob would never figure out. Or so I thought. Pete agreed that the question I posed would be simple enough even for you to answer." I bristled at that, but the Guru, absorbed in her explanation, ignored me. "If Bob could not figure out the answer by two o'clock, Pete was going to keep a close eye on him, perhaps even put him on probation. Eventually, I'm sure I could have got Bob what he deserved – a pink slip. Now, though, it's going to be much harder."

She sighed and silently stood and glided away.

My stomach rose in my throat. "What have I done?" I muttered to myself.

- - - - -

It was just before daybreak that the conspirators acted. Several officers went armed into Da Rosa's tent, and we quickly discovered that only the mutineers' sergeants had functioning weapons. Da Rosa put up no real opposition; I think he was tired and already resigned to the fact that holding out further would be futile. He gave up his sidearm, Hinckel took command, and we prepared to surrender.

A man was sent to throw a surrender note through the gate. We could see a brief glimpse of movement through the gate as the note was retrieved and soon came the reply. Following instructions, we put down our weapons and filed through the gate one by one. When my turn came, I stepped through into the dimly lit Ballroom and turned to find myself staring at the business end of weapons carried by grim-faced men. I sighed and was bound like the others.