Contents

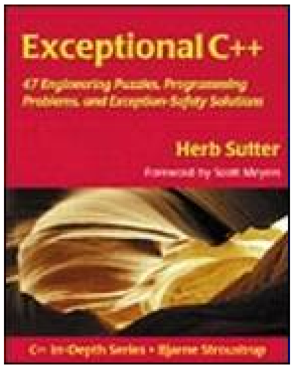
Exceptional C++
47 Engineering Puzzles, Programming Problems, and Exception-Safety Solutions
Herb Sutter
Foreword by Scott Meyers
C++ In-Depth Series • Bjarne Stroustrup

Table of Contents

Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions
By Herb Sutter

Publisher : Addison Wesley
Pub Date : November 18, 1999
ISBN : 0-201-61562-2
Pages : 240

Exceptional C++ shows by example how to go about sound software engineering in standard C++. Do you enjoy solving thorny C++ problems and puzzles? Do you relish writing robust and extensible code? Then take a few minutes and challenge yourself with some tough C++ design and programming problems.

The puzzles and problems in Exceptional C++ not only entertain, they will help you hone your skills to become the sharpest C++ programmer you can be. Many of these problems are culled from the famous Guru of the Week feature of the Internet newsgroup comp.lang.c++.moderated, expanded and updated to conform to the official ISO/ANSI C++ Standard.

Each problem is rated according to difficulty and is designed to illustrate subtle programming mistakes or design considerations. After you've had a chance to attempt a solution yourself, the book then dissects the code, illustrates what went wrong, and shows how the problem can be fixed. Covering a broad range of C++ topics, the problems and solutions address critical issues such as:

- Generic programming and how to write reusable templates
- Exception safety issues and techniques
- Robust class design and inheritance
- Compiler firewalls and the Pimpl Idiom
- Name lookup, namespaces, and the Interface Principle
- Memory management issues and techniques
- Traps, pitfalls, and anti-idioms
- Optimization

Try your skills against the C++ masters and come away with the insight and experience to create more efficient, effective, robust, and portable C++ code.

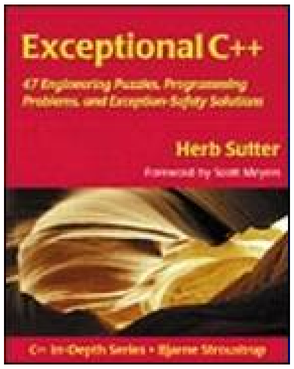

Table of Contents

Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions

By Herb Sutter

Publisher : Addison Wesley
Pub Date : November 18, 1999
ISBN : 0-201-61562-2
Pages : 240

I l@ve RuBoard PREVIOUS NEXT

I l@ve RuBoard PREVIOUS NEXT

Copyright

Library of Congress Cataloging-in-Publication Data

Sutter, Herb.

Exceptional C++ : 47 engineering puzzles, programming problems, and solutions / Herb Sutter.

p. cm.

Includes bibliographical references and index.

ISBN 0-201-61562-2 (alk. paper)

Foreword

This is a remarkable book, but it wasn't until I had nearly finished reading it that I realized just how remarkable it is. This could well be the first book ever written for people who are already familiar with C++—all of C++. From language features to components of the standard library to programming techniques, this book skips from topic to topic, always keeping you slightly off balance, always making sure you're paying attention. Just like real C++ programs. Class design bumps into the behavior of virtual functions, iterator conventions run up against name lookup rules, assignment operators sideswipe exception safety, compilation dependencies cross paths with exported templates. Just like they do in real programs. The result is a dizzying maelstrom of language features, library components, and programming techniques at once both chaotic and magnificent. Just like real programs.

I pronounce GotW such that it rhymes with "Gotcha," and perhaps that's fitting. As I compared my solutions to the book's quizzes against Sutter's answers, I fell into the traps he (and C++) laid before me more often than I'd like to admit. I could almost see Herb smiling and softly saying "Gotcha!" for each error I made. Some may argue that this proves I don't know much about C++. Others may claim it demonstrates that C++ is too complex for anyone to master. I believe it shows that when you're working in C++, you have to think carefully about what you're doing. C++ is a powerful language designed to help solve demanding problems, and it's important that you hone your knowledge of the language, its library, and its programming idioms as finely as you can. The breadth of topics in this book will help you do that. So will its unique quiz-based format.

Veteran readers of the C++ newsgroups know how difficult it is to be proclaimed a Guru of the Week. Veteran participants know it even better. On the Internet, of course, there can be only one guru each week, but, backed by the information in this book, you can reasonably hope to produce guru-quality code every time you program.

Scott Meyers
June 1999

Preface

Exceptional C++ shows by example how to go about solid software engineering. Along with a lot of other material, this book includes expanded versions of the first 30 issues of the popular Internet C++ feature Guru of the Week (or, in its short form, GotW), a series of self-contained C++ engineering problems and solutions that illustrate specific design and coding techniques.

This book isn't a random grab-bag of code puzzles; it's primarily a guide to sound real-world enterprise software design in C++. It uses a problem/solution format because that's the most effective way I know to involve you, gentle reader, in the ideas behind the problems and the reasons behind the guidelines. Although the Items cover a variety of topics, you'll notice recurring themes that focus on enterprise development issues, particularly exception safety, sound class and module design, appropriate optimization, and writing portable standards-conforming code.

I hope you find this material useful in your daily work. But I also hope you find at least a few nifty thoughts and elegant techniques, and that from time to time, as you're reading through these pages, you'll suddenly have an "Aha! Gnarly!" moment. After all, who says software engineering has to be dull?

How to Read This Book

I expect that you already know the basics of C++. If you don't, start with a good C++ introduction and overview (good choices are a classic tome like Bjarne Stroustrup's The C++ Programming Language, Third Edition[1] or Stan Lippman and Josée Lajoie's C++ Primer, Third Edition[2]), and then be sure to pick up a style guide like Scott Meyers' classic Effective C++ books (I find the browser-based CD version convenient and useful).[3]

[1] Stroustrup B. The C++ Programming Language, Third Edition (Addison Wesley Longman, 1997).

[2] Lippman S. and Lajoie J. C++ Primer, Third Edition (Addison Wesley Longman, 1998).

[3] Meyers S. Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs (Addison Wesley Longman, 1999). An online demo is available at http://www.meyerscd.awl.com.

Each item in this book is presented as a puzzle or problem, with an introductory header that looks like this:

Item ##. The Topic of This Puzzle Difficulty

Difficulty: X

The topic tag and difficulty rating (typically anything from 3 to 91/2 , based on a scale of 10) gives you a hint of what you're in for. Note that the difficulty rating is my own subjective guess at how difficult I expect most people will find each problem, so you may well find that a given 7 problem is easier for you than another 5 problem. Still, it's better to be prepared for the worst when you see a 9 monster coming down the pike.

You don't have to read the sections and problems in order, but in several places there are "miniseries" of related problems that you'll see designated as "Part 1," "Part 2," and so on—some all the way up to "Part 10." Those miniseries are best read as a group.

This book includes many guidelines, in which the following words usually carry a specific meaning:

-
- always = This is absolutely necessary. Never fail to do this.
-
-
- prefer = This is usually the right way. Do it another way only when a situation specifically warrants it.
-
-

How We Got Here: GotW and PeerDirect

The C++ Guru of the Week series has come a long way. GotW was originally created late in 1996 to provide interesting challenges and ongoing education for our own development team here at PeerDirect. I wrote it to provide an entertaining learning tool, including rants on things like the proper use of inheritance and exception safety. As time went on, I also used it as a means to provide our team with visibility to the changes being made at the C++ standards meetings. Since then, GotW has been made available to the general C++ public as a regular feature of the Internet newsgroup comp.lang.c++.moderated, where you can find each new issue's questions and answers (and a lot of interesting discussion).

Using C++ well is important at PeerDirect for many of the same reasons it's important in your company, if perhaps to achieve different goals. We happen to build systems software—for distributed databases and database replication—in which enterprise issues such as reliability, safety, portability, efficiency, and many others are make-or-break concerns. The software we write needs to be able to be ported across various compilers and operating systems; it needs to be safe and robust in the presence of database transaction deadlocks and communications interruptions and programming exceptions; and it's used by customers to manage tiny databases sitting inside smart cards and pop machines or on PalmOS and WinCE devices, through to departmental Windows NT and Linux and Solaris servers, through to massively parallel Oracle back-ends for Web servers and data warehouses—with the same software, the same reliability, the same code. Now that's a portability and reliability challenge, as we creep up on half a million tight, noncomment lines of code.

To those of you who have been reading Guru of the Week on the Internet for the past few years, I have a couple of things to say:

- Thank you for your interest, support, e-mails, kudos, corrections, comments, criticisms, questions—and especially for your requests for the GotW series to be assembled in book form. Here it is; I hope you enjoy it.
- This book contains a lot more than you ever saw on the Internet.

Exceptional C++ is not just a cut-and-paste of stale GotW issues that are already floating out there somewhere in cyberspace. All the problems and solutions have been considerably revised and reworked—for example, Items 8 through 17 on exception safety originally appeared as a single GotW puzzle and have now become an in-depth, 10-part miniseries. Each problem and solution has been examined to bring it up to date with the then-changing, and now official, C++ standard.

So, if you've been a regular reader of GotW before, there's a lot that's new here for you. To all faithful readers, thanks again, and I hope this material will help you continue to hone and expand your software engineering and C++ programming skills.

Acknowledgments

First, of course, thanks to all the GotW readers and enthusiasts on comp.lang.c++.moderated, especially the scores of people who participated in the contest to select a name for this book. Two in particular were instrumental in leading us to the final title, and I want to thank them specifically: Marco Dalla Gasperina for suggesting the name Enlightened C++, and Rob Stewart for suggesting the name Practical C++ Problems and Solutions. It was only natural to take these a step further and insert the pun exceptional, given the repeated emphasis herein on exception safety.

Many thanks also to series editor Bjarne Stroustrup and to Marina Lang, Debbie Lafferty, and the rest of the Addison Wesley Longman editorial staff for their continued interest and enthusiasm in this project, and for hosting a really nice reception at the Santa Cruz C++ standards meeting in 1998.

I also want to thank the many people who acted as reviewers—many of them fellow standards-committee members—who provided thoughtful and incisive comments that have helped to improve the text you are about to read. Special thanks to Bjarne Stroustrup and Scott Meyers, and to Andrei Alexandrescu, Steve Clamage, Steve Dewhurst, Cay Horstmann, Jim Hyslop, Brendan Kehoe, and Dennis Mancl, for their invaluable insights and reviews.

Finally, thanks most of all to my family and friends for always being there, in so many different ways.

Herb Sutter
June 1999

Generic Programming and the C++ Standard Library

To begin, let's consider a few selected topics in the area of generic programming. These puzzles focus on the effective use of templates, iterators, and algorithms, and how to use and extend standard library facilities. These ideas then lead nicely into the following section, which analyzes exception safety in the context of writing exception-safe templates.

I l@ve RuBoard PREVIOUS NEXT

I l@ve RuBoard PREVIOUS NEXT

Item 1. Iterators

Difficulty: 7

Every programmer who uses the standard library has to be aware of these common and not-so-common iterator mistakes. How many of them can you find?

The following program has at least four iterator-related problems. How many can you find?

```
int main()
{
  vector<Date> e;
  copy( istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ) );
  vector<Date>::iterator first =
        find( e.begin(), e.end(), "01/01/95" );
  vector<Date>::iterator last =
        find( e.begin(), e.end(), "12/31/95" );
  *last = "12/30/95";
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
  e.insert( --e.end(), TodaysDate() );
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
}
```

Solution

```
int main()
{
  vector<Date> e;
  copy( istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ) );
```

This is fine so far. The Date class writer provided an extractor function with the signature operator>>(istream&, Date&), which is what istream_iterator<Date> uses to read the Dates from the cin stream. The copy() algorithm just stuffs the Dates into the vector.

```
vector<Date>::iterator first =
      find( e.begin(), e.end(), "01/01/95" );
vector<Date>::iterator last =
      find( e.begin(), e.end(), "12/31/95" );
*last = "12/30/95";
```

Error: This may be illegal, because last may be e.end() and therefore not a dereferenceable iterator.

The find() algorithm returns its second argument (the end iterator of the range) if the value is not found. In this case, if "12/31/95" is not in e, then last is equal to e.end(), which points to one-past-the-end of the container and is not a valid iterator.

```
copy( first,
      last,
      ostream_iterator<Date>( cout, "\n" ) );
```

Error: This may be illegal because [first,last) may not be a valid range; indeed, first may actually be after last.

For example, if "01/01/95" is not found in e but "12/31/95" is, then the iterator last will point to something earlier in the collection (the Date object equal to "12/31/95") than does the iterator first (one past the end). However, copy() requires that first must point to an earlier place in the same collection as last—that is, [first,last) must be a valid range.

Unless you're using a checked version of the standard library that can detect some of these problems for you, the likely symptom if this happens will be a difficult-to-diagnose core dump during or sometime after the copy().

```
e.insert(--e.end(), TodaysDate() );
```

First error: The expression "--e.end()" is likely to be illegal.

Item 2. Case-Insensitive Strings—Part 1

Difficulty: 7

So you want a case-insensitive string class? Your mission, should you choose to accept it, is to write one.

This Item is composed of three related points.

1.

1. What does "case-insensitive" mean?
1.
2.

2. Write a ci_string class that is identical to the standard std::string class but that is case-insensitive in the same way as the commonly provided extension stricmp().[1] A ci_string should be usable as follows:

2. [1] The stricmp() case-insensitive string comparison function is not part of the C or C++ standards, but it is a common extension on many C and C++ compilers.
2.

```
ci_string s( "AbCdE" );
// case insensitive
//
assert( s == "abcde" );
assert( s == "ABCDE" );
// still case-preserving, of course
//
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

3.

3. Is making case sensitivity a property of the object a good idea?
3.

Solution

The answers to the three questions are as follows.

1.

1. What does "case-insensitive" mean?

1. What "case-insensitive" actually means depends entirely on your application and language. For example, many languages do not have cases at all. For those that do, you still have to decide whether you want accented characters to compare equal to unaccented characters, and so on. This Item provides guidance on how to implement case-insensitivity for standard strings in whatever sense applies to your situation.
1.
2.

2. Write a ci_string class that is identical to the standard std::string class but that is case-insensitive in the same way as the commonly provided extension stricmp().

2. The "how can I make a case-insensitive string?" question is so common that it probably deserves its own FAQ—hence this Item.
2.

2. Here's what we want to achieve:
2.

```
ci_string s( "AbCdE" );
// case insensitive
//
assert( s == "abcde" );
assert( s == "ABCDE" );
// still case-preserving, of course
//
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

2. The key here is to understand what a string actually is in Standard C++. If you look in your trusty string header, you'll see something like this:
2.

```
typedef basic_string<char> string;
```

2. So string isn't really a class; it's a typedef of a template. In turn, the basic_string<> template is declared as follows, possibly with additional implementation-specific template parameters:
2.

```
template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_string;
```

Item 3. Case-Insensitive Strings—Part 2

Difficulty: 5

How usable is the ci_string we created in Item 2? Now we'll focus on usability issues, see what design problems or tradeoffs we encounter, and fill in some of the remaining gaps.

Consider again the solution for Item 2 (ignoring the function bodies):

```
struct ci_char_traits : public char_traits<char>
{
  static bool eq( char c1, char c2 )   { /*...*/ }
  static bool lt( char c1, char c2 )   { /*...*/ }
  static int compare( const char* s1,
                      const char* s2,
                      size_t n )       { /*...*/ }
  static const char*
  find( const char* s, int n, char a ) { /*...*/ }
};
```

For this Item, answer the following related questions as completely as possible:

1.

1. Is it safe to inherit ci_char_traits from char_traits<char> this way?
1.
2.

2. Why does the following code fail to compile?

2.
```
ci_string s = "abc";
cout << s << endl;
```

3.

3. What about using other operators (for example, +, +=, =) and mixing strings and ci_strings as arguments? For example:

3.
```
string    a = "aaa";
ci_string b = "bbb";
string    c = a + b;
```

Solution

The answers to the three questions are as follows.

1.

1. Is it safe to inherit ci_char_traits from char_traits<char> this way?

1. Public inheritance should normally model IS-A / WORKS-LIKE-A as per the Liskov Substitution Principle (LSP). (See Items 22 and 28.) This, however, is one of the rare exceptions to the LSP, because ci_char_traits is not intended to be used polymorphically through a pointer or reference to the base class char_traits<char>. The standard library does not use traits objects polymorphically. In this case, inheritance is used merely for convenience (well, some would say laziness); inheritance is not being used in an object-oriented way.
1.

1. On the other hand, the LSP does still apply in another sense: It applies at compile-time, when the derived object must WORK-LIKE-A base object in the ways required by the basic_string template's requirements. In a newsgroup posting, Nathan Myers[2] puts it this way:
1.

1. [2] Nathan is a longtime member of the C++ standards committee and the primary author of the standard's locale facility.
1.

1. In other words, LSP applies, but only at compile-time, and by the convention we call "requirements lists." I would like to distinguish this case; call it Generic Liskov Substitution Principle (GLSP): Any type (or template) passed as a template argument should conform to the requirements listed for that argument.

1. Classes derived from iterator tags and traits classes, then, are subject to GLSP, but classical LSP considerations (for example, virtual destructor, and so forth) may or may not apply, depending on whether run-time polymorphic behaviors are in the signature specified in the requirements list.

1. So, in short, this inheritance is safe, because it conforms to GLSP (if not LSP). However, the main reason I used it here was not for convenience (to avoid writing all the other char_traits<char> baggage), but to demonstrate what's different—that we had to change only four operations to get the effect we want.
1.

1. The reason behind this first question was to get you to think about several things: (1) the proper uses (and improper abuses) of inheritance; (2) the implications of the fact that there are only static members; (3) the fact that char_traits objects are never used polymorphically.
1.
2.

2. Why does the following code fail to compile?

Item 4. Maximally Reusable Generic Containers—Part 1

Difficulty: 8

How flexible can you make this simple container class? Hint: You'll learn more than a little about member templates along the way.

How can you best implement copy construction and copy assignment for the following fixed-length vector class? How can you provide maximum usability for construction and assignment? Hint: Think about the kinds of things that client code might want to do.

```
template<typename T, size_t size>
class fixed_vector
{
public:
  typedef T*       iterator;
  typedef const T* const_iterator;
  iterator       begin()       { return v_; }
  iterator       end()         { return v_+size; }
  const_iterator begin() const { return v_; }
  const_iterator end()   const { return v_+size; }

private:
  T v_[size];
};
```

Note: Don't fix other things. This container is not intended to be fully STL-compliant, and it has at least one subtle problem. It's meant only to illustrate some important issues in a simplified setting.

Solution

For this Item's solution, we'll do something a little different. I'll present a proposed solution, and your mission is to supply the explanation and critique the solution. Consider now Item 5.

Item 5. Maximally Reusable Generic Containers—Part 2

Difficulty: 6

Historical note: The example used in this Item is adapted from one presented by Kevlin Henney and later analyzed by Jon Jagger in issues 12 and 20 of the British C++ magazine Overload. (British readers beware: The answer to this Item goes well beyond that presented in Overload #20. In fact, the efficiency optimization presented there won't work in the solution to this problem.)

What is the following solution doing, and why? Explain each constructor and operator. Does this design or code have any flaws?

```
template<typename T, size_t size>
class fixed_vector
{
public:
  typedef T*       iterator;
  typedef const T* const_iterator;
  fixed_vector() { }

  template<typename O, size_t osize>
  fixed_vector( const fixed_vector<O,osize>& other )
  {
    copy( other.begin(),
          other.begin()+min(size,osize),
          begin() );
  }

  template<typename O, size_t osize>
  fixed_vector<T,size>&
  operator=( const fixed_vector<O,osize>& other )
  {
    copy( other.begin(),
          other.begin()+min(size,osize),
          begin() );
    return *this;
  }

  iterator       begin()       { return v_; }
  iterator       end()         { return v_+size; }
  const_iterator begin() const { return v_; }
  const_iterator end()   const { return v_+size; }

private:
  T v_[size];
};
```

Solution

Let's analyze the solution above and see how well it measures up to what the question asked. Recall that the original question was: How can you best implement copy construction and copy assignment for the following fixed-length vector class? How can you provide maximum usability for construction and assignment? Hint: Think about the kinds of things that client code might want to do.

Copy Construction and Copy Assignment

First, note that the first question is a red herring. Did you spot it? The original code already had a copy constructor and a copy assignment operator that worked just fine, thank you very much. Our solution proposes to address the second question by adding a templated constructor and a templated assignment operator to make construction and assignment more flexible.

```
template<typename O, size_t osize>
fixed_vector( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
}

template<typename O, size_t osize>
fixed_vector<T,size>&
operator=( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
  return *this;
}
```

Note that the above functions are not a copy constructor and a copy assignment operator. Here's why: A copy constructor or copy assignment operator specifically constructs/assigns from another object of exactly the same type—including the same template arguments, if the class is templated. For example:

```
struct X
{
  template<typename T>
  X( const T& );    // NOT copy constructor, T can't be X

  template<typename T>
  operator=( const T& );
                    // NOT copy assignment, T can't be X
};
```

"But," you say, "those two templated member functions could exactly match the signatures of copy construction and copy assignment!" Well, actually, no—they couldn't, because in both cases, T may not be X. To quote from the standard (12.8/2, note 4):

Alternative: The Standard Library Approach

I happen to like the syntax and usability of the preceding functions, but there are still some nifty things they won't let you do. Consider another approach that follows the style of the standard library.

1.

1. Copying

1. template<class RAIter>

```
fixed_vector( RAIter first, RAIter last )
{
  copy( first,
        first+min(size,(size_t)last-first),
        begin() );
}
```

1. Now when copying, instead of writing:
1.

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v);  // initialize using 4 values
```

we need to write:

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v.begin(), v.end());
                            // initialize using 4 values
```

Stop and think about this for a moment. What do you think of it? For construction, which style is better—the style of our proposed solution or this standard library-like style?

In this case, our initial proposed solution is somewhat easier to use, whereas the standard library-like style is much more flexible (for example, it allows users to choose subranges and copy from other kinds of containers). You can take your pick or simply supply both flavors.

2.

2. Assignment

2. Note that we can't templatize assignment to take an iterator range, because operator=() may take only one parameter. Instead, we can provide a named function:
2.

```
template<class Iter>
fixed_vector<T,size>&
assign( Iter first, Iter last )
{
  copy( first,
        first+min(size,(size_t)last-first),
```

Item 6. Temporary Objects

Difficulty: 5

Unnecessary and/or temporary objects are frequent culprits that can throw all your hard work—and your program's performance—right out the window. How can you spot them and avoid them?

("Temporary objects?" you might be wondering. "That's more of an optimization thing. What's that got to do with generic programming and the standard library?" Bear with me; the reason will become clear in the following Item.)

You are doing a code review. A programmer has written the following function, which uses unnecessary temporary objects in at least three places. How many can you identify, and how should the programmer fix them?

```
string FindAddr( list<Employee> emps, string name )
{
  for( list<Employee>::iterator i = emps.begin();
       i != emps.end();
       i++ )
  {
    if( *i == name )
    {
      return i->addr;
    }
  }
  return "";
}
```

Do not change the operational semantics of this function, even though they could be improved.

Solution

Believe it or not, this short function harbors three obvious cases of unnecessary temporaries, two subtler ones, and two red herrings.

The two more-obvious temporaries are buried in the function declaration itself:

```
string FindAddr( list<Employee> emps, string name )
```

The parameters should be passed by const&—that is, const list<Employee>& and const string&, respectively—instead of by value. Pass-by-value forces the compiler to make complete copies of both objects, which can be expensive and, here, is completely unnecessary.

Guideline

Prefer passing objects by const& instead of passing by value.

The third more-obvious avoidable temporary occurs in the for loop's termination condition:

```
for( /*...*/ ; i != emps.end(); /*...*/ )
```

For most containers (including list), calling end() returns a temporary object that must be constructed and destroyed. Because the value will not change, recomputing (and reconstructing and redestroying) it on every loop iteration is both needlessly inefficient and unaesthetic. The value should be computed only once, stored in a local object, and reused.

Guideline

Prefer precomputing values that won't change, instead of recreating objects unnecessarily.

Next, consider the way we increment i in the for loop:

```
for( /*...*/ ; i++
```

Item 7. Using the Standard Library (or, Temporaries Revisited)

Difficulty: 5

Effective reuse is an important part of good software engineering. To demonstrate how much better off you can be by using standard library algorithms instead of handcrafting your own, let's reconsider the previous Item to demonstrate how many of the problems could have been avoided by simply reusing what's already available in the standard library.

How many pitfalls in Item 6 could have been avoided if only the programmer had used a standard library algorithm instead of handcrafting the loop? Demonstrate. (As with Item 6, don't change the semantics of the function, even though they could be improved.)

To recap, here is the mostly-fixed function:

```
string FindAddr( const list<Employee>& emps,
                 const string&         name )
{
  list<Employee>::const_iterator end( emps.end() );
  for( list<Employee>::const_iterator i = emps.begin();
       i != end;
       ++i )
  {
    if( i->name == name )
    {
      return i->addr;
    }
  }
  return "";
}
```

Solution

With no other changes, simply using the standard find() algorithm could have avoided two temporaries, as well as the emps.end() recomputation inefficiency from the original code. For the best effect to reduce temporaries, provide an operator==() taking an Employee& and a name string&.

```
string FindAddr( list<Employee> emps, string name )
{
  list<Employee>::iterator i(
    find( emps.begin(), emps.end(), name )
    );
  if( i != emps.end() )
  {
    return i->addr;
  }
  return "";
}
```

Yes, you can get even fancier with functors and find_if, but see how much this simple reuse of find saves in programming effort and run-time efficiency.

Guideline

Reuse code—especially standard library code—instead of handcrafting your own. It's faster, easier, and safer.

Reusing existing code is usually preferable to handcrafting your own. The standard library is full of code that's intended to be used and reused, and to that end a lot of thought and care has gone into the design of the library, including its standard algorithms, such as find() and sort(). Implementers have spent hours sweating over efficiency details, usability details, all sorts of other considerations so that you don't have to. So reuse code, especially code in the standard library, and escape the trap of "I'll-write-my-own."

Combining this with the other fixes, we get a much improved function.

```
string FindAddr( const list<Employee>& emps,
                 const string&         name )
{
  list<Employee>::const_iterator i(
    find( emps.begin(), emps.end(), name )
    );
  if( i != emps.end() )
  {
    return i->addr;
  }
```

Exception-Safety Issues and Techniques

First, a little of the history of this topic is in order. In 1994, Tom Cargill published the seminal article "Exception Handling: A False Sense of Security" (Cargill94).[1] It demonstrated conclusively that, at that time, the C++ community did not yet fully understand how to write exception-safe code. In fact, we didn't even know what all the important exception-safety issues were, or how to correctly reason about exception safety. Cargill challenged anyone to demonstrate a conclusive solution to this problem. Three years passed. A few people wrote partial responses to aspects of Cargill's example, but no one showed a comprehensive solution.

[1] Available online at http://www.gotw.ca/publications/xc++/sm_effective.htm.

Then, in 1997, Guru of the Week #8 appeared on the Internet newsgroup comp.lang.c++.moderated. Number 8 generated weeks of discussion and presented the first complete solution to Cargill's challenge. Later that year, a greatly expanded version that was updated to match the latest changes to draft standard C++ and demonstrating no fewer than three complete solutions, was published in the September and November/December issues of C++ Report under the title "Exception-Safe Generic Containers." (Copies of those original articles will also appear in the forthcoming book C++ Gems II [Martin00].)

In early 1999, on the Effective C++ CD (Meyers99), Scott Meyers included a recombined version of the articles, together with Cargill's original challenge, with the updated text of his classic books Effective C++ and More Effective C++.

This miniseries has come a long way since its original publication as Guru of the Week #8. I hope you enjoy it and find it useful. Particular thanks go to fellow committee members Dave Abrahams and Greg Colvin for their insights into how to reason about exception-safety and their thoughtful critiques of several drafts of this material. Dave and Greg are, with Matt Austern, the authors of the two complete committee proposals for adding the current exception-safety guarantees into the standard library.

This miniseries tackles both major features, exception handling and templates, at once, by examining how to write exception-safe (works properly in the presence of exceptions) and exception-neutral (propagates all exceptions to the caller) generic containers. That's easy enough to say, but it's no mean feat.

So come on in, join the fun, and try your hand at implementing a simple container (a Stack that users can push and pop) and see the issues involved with making it exception-safe and exception-neutral.

Item 8. Writing Exception-Safe Code—Part 1

Difficulty: 7

Exception handling and templates are two of C++'s most powerful features. Writing exception-safe code, however, can be difficult—especially in a template, when you may have no idea when (or what) a certain function might throw your way.

We'll begin where Cargill left off—namely, by progressively creating a safe version of the Stack template he critiqued. Later on, we'll significantly improve the Stack container by reducing the requirements on T, the contained type, and show advanced techniques for managing resources exception-safely. Along the way, we'll find the answers to such questions as:

- What are the different "levels" of exception safety?
- Can or should generic containers be fully exception-neutral?
- Are the standard library containers exception-safe or exception-neutral?
- Does exception safety affect the design of your container's public interface?
- Should generic containers use exception specifications?

Here is the declaration of the Stack template, substantially the same as in Cargill's article. Your mission: Make Stack exception-safe and exception-neutral. That is, Stack objects should always be in a correct and consistent state, regardless of any exceptions that might be thrown in the course of executing Stack's member functions. If any exceptions are thrown, they should be propagated seamlessly through to the caller, who can deal with them as he pleases, because he knows the context of T and we don't.

```
template <class T> class Stack
{
public:
 Stack();
 ~Stack();

   /*...*/
```

Solution

Right away, we can see that Stack is going to have to manage dynamic memory resources. Clearly, one key is going to be avoiding leaks, even in the presence of exceptions thrown by T operations and standard memory allocations. For now, we'll manage these memory resources within each Stack member function. Later on in this miniseries, we'll improve on this by using a private base class to encapsulate resource ownership.

Default Construction

First, consider one possible default constructor:

```
// Is this safe?

template<class T>
Stack<T>::Stack()
: v_(0),
  vsize_(10),
  vused_(0)         // nothing used yet
{
  v_ = new T[vsize_]; // initial allocation
}
```

Is this constructor exception-safe and exception-neutral? To find out, consider what might throw. In short, the answer is: any function. So the first step is to analyze this code and determine which functions will actually be called, including both free functions and constructors, destructors, operators, and other member functions.

This Stack constructor first sets vsize_ to 10, then attempts to allocate some initial memory using new T[vsize_]. The latter first tries to call operator new[]() (either the default operator new[]() or one provided by T) to allocate the memory, then tries to call T::T a total of vsize_ times. There are two operations that might fail. First, the memory allocation itself, in which case operator new[]() will throw a bad_alloc exception. Second, T's default constructor, which might throw anything at all, in which case any objects that were constructed are destroyed and the allocated memory is automatically guaranteed to be deallocated via operator delete[]().

Hence the above function is fully exception-safe and exception-neutral, and we can move on to the next what? Why is the function fully robust, you ask? All right, let's examine it in a little more detail.

1.

1. We're exception-neutral. We don't catch anything, so if the new throws, then the exception is correctly propagated up to our caller as required.

1. Guideline

Item 9. Writing Exception-Safe Code—Part 2

Difficulty: 8

Now that we have the default constructor and the destructor under our belts, we might be tempted to think that all the other functions will be about the same. Well, writing exception-safe and exception-neutral copy and assignment code presents its own challenges, as we shall now see.

Consider again Cargill's Stack template:

```
template <class T> class Stack
{
public:
  Stack();
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  /*...*/
private:
  T*     v_;      // ptr to a memory area big
  size_t vsize_;  //  enough for 'vsize_' T's
  size_t vused_;  // # of T's actually in use
};
```

Now write the Stack copy constructor and copy assignment operator so that both are demonstrably exception-safe (work properly in the presence of exceptions) and exception-neutral (propagate all exceptions to the caller, without causing integrity problems in a Stack object).

Solution

To implement the copy constructor and the copy assignment operator, let's use a common helper function, NewCopy, to manage allocating and growing memory. NewCopy takes a pointer to (src) and size of (srcsize) an existing T buffer, and returns a pointer to a new and possibly larger copy of the buffer, passing ownership of the new buffer to the caller. If exceptions are encountered, NewCopy correctly releases all temporary resources and propagates the exception in such a way that nothing is leaked.

```
template<class T>
T* NewCopy( const T* src,
            size_t   srcsize,
            size_t   destsize )
{
  assert( destsize >= srcsize );
  T* dest = new T[destsize];
  try
  {
    copy( src, src+srcsize, dest );
  }
  catch(...)
  {
    delete[] dest; // this can't throw
    throw;         // rethrow original exception
  }
  return dest;
}
```

Let's analyze this one step at a time.

1.

1. In the new statement, the allocation might throw bad_alloc or the T::T's may throw anything. In either case, nothing is allocated and we simply allow the exception to propagate. This is both leak-free and exception-neutral.
1.
2.

2. Next, we assign all the existing values using T::operator=(). If any of the assignments fail, we catch the exception, free the allocated memory, and rethrow the original exception. This is again both leak-free and exception-neutral. However, there's an important subtlety here: T::operator=() must guarantee that if it does throw, then the assigned-to T object must be destructible.[4]

2. [4] As we progress, we'll arrive at an improved version of Stack that does not rely on T::operator=.
2.
3.

3. If the allocation and copy both succeed, then we return the pointer to the new buffer and relinquish ownership (that is, the caller is responsible for the buffer from here on out). The return simply copies the pointer value, which cannot throw.
3.

Item 10. Writing Exception-Safe Code—Part 3

Difficulty: 9

Are you getting the hang of exception safety? Well, then, it must be time to throw you a curve ball. So get ready, and don't swing too soon.

Now for the final piece of Cargill's original Stack template.

```
template <class T> class Stack
{
public:
  Stack();
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
size_t Count() const;
void  Push(const T&);
T    Pop(); // if empty, throws exception
private:
  T*     v_;      // ptr to a memory area big
  size_t vsize_;  //  enough for 'vsize_' T's
  size_t vused_;  // # of T's actually in use
};
```

Write the final three Stack functions: Count(), Push(), and Pop(). Remember, be exception-safe and exception-neutral!

Solution

Count()

The easiest of all Stack's members to implement safely is Count, because all it does is copy a builtin that can never throw.

```
template<class T>
size_t Stack<T>::Count() const
{
  return vused_;  // safe, builtins don't throw
}
```

No problem.

Push()

On the other hand, with Push, we need to apply our now-usual duty of care.

```
template<class T>
void Stack<T>::Push( const T& t )
{
  if( vused_ == vsize_ )  // grow if necessary
  {                       // by some grow factor
    size_t vsize_new = vsize_*2+1;
    T* v_new = NewCopy( v_, vsize_, vsize_new );
    delete[] v_;  // this can't throw
    v_ = v_new;   // take ownership
    vsize_ = vsize_new;
  }
  v_[vused_] = t;
  ++vused_;
}
```

If we have no more space, we first pick a new size for the buffer and make a larger copy using NewCopy. Again, if NewCopy throws, then our own Stack's state is unchanged and the exception propagates through cleanly. Deleting the original buffer and taking ownership of the new one involves only operations that are known not to throw, so the entire if block is exception-safe.

After any required grow operation, we attempt to copy the new value before incrementing our vused_ count. This way, if the assignment throws, the increment is not performed and our Stack's state is unchanged. If the assignment succeeds, the Stack's state is changed to recognize the presence of the new value, and all is well.

Guideline

Item 11. Writing Exception-Safe Code—Part 4

Difficulty: 8

Mid-series interlude: What have we accomplished so far?

Now that we have implemented an exception-safe and exception-neutral Stack<T>, answer these questions as precisely as possible:

1.

1. What are the important exception-safety guarantees?
1.
2.

2. For the Stack<T> that was just implemented, what are the requirements on T, the contained type?
2.

Solution

Just as there's more than one way to skin a cat (I have a feeling I'm going to get enraged e-mail from animal lovers), there's more than one way to write exception-safe code. In fact, there are two main alternatives we can choose from when it comes to guaranteeing exception safety. These guarantees were first set out in this form by Dave Abrahams.

1.

1. Basic guarantee: Even in the presence of exceptions thrown by T or other exceptions, Stack objects don't leak resources. Note that this also implies that the container will be destructible and usable even if an exception is thrown while performing some container operation. However, if an exception is thrown, the container will be in a consistent, but not necessarily predictable, state. Containers that support the basic guarantee can work safely in some settings.
1.
2.

2. Strong guarantee: If an operation terminates because of an exception, program state will remain unchanged. This always implies commit-or-rollback semantics, including that no references or iterators into the container be invalidated if an operation fails. For example, if a Stack client calls Top and then attempts a Push that fails because of an exception, then the state of the Stack object must be unchanged and the reference returned from the prior call to Top must still be valid. For more information on these guarantees, see Dave Abrahams's documentation of the SGI exception-safe standard library adaptation at: http://www.gotw.ca/publications/xc++/da_stlsafety.htm .
2.

2. Probably the most interesting point here is that when you implement the basic guarantee, the strong guarantee often comes along for free.[7] For example, in our Stack implementation, almost everything we did was needed to satisfy just the basic guarantee—and what's presented above very nearly satisfies the strong guarantee, with little or no extra work.[8] Not half bad, considering all the trouble we went to.

2. [7] Note that I said "often," not "always." In the standard library, for example, vector is a well-known counter-example in which satisfying the basic guarantee does not cause the strong guarantee to come along for free.
2.

2. [8] There is one subtle way in which this version of Stack still falls short of the strong guarantee. If Push() is called and has to grow its internal buffer, but then its final v_[vused_] = t; assignment throws, the Stack is still in a consistent state, but its internal memory buffer has moved—which invalidates any previously valid references returned from Top(). This last flaw in Stack::Push() can be fixed fairly easily by moving some code and adding a try block. For a better solution, however, see the Stack presented in the second half of this miniseries. That Stack does not have the problem, and it does satisfy the strong commit-or-rollback guarantee.
2.

2. In addition to these two guarantees, there is one more guarantee that certain functions must provide

Item 12. Writing Exception-Safe Code—Part 5

Difficulty: 7

All right, you've had enough rest—roll up your sleeves, and get ready for a wild ride.

Now we're ready to delve a little deeper into the same example, and write not just one but two new-and-improved versions of Stack. Not only is it, indeed, possible to write exception-safe generic containers, but by the time this miniseries is over, we'll have created no fewer than three complete solutions to the exception-safe Stack problem.

Along the way, we'll also discover the answers to several more interesting questions:

- How can we use more-advanced techniques to simplify the way we manage resources and get rid of the last try/catch into the bargain?
- How can we improve Stack by reducing the requirements on T, the contained type?
- Should generic containers use exception specifications?
- What do new[] and delete[] really do?

The answer to the last question may be quite different from what one might expect. Writing exception-safe containers in C++ isn't rocket science; it just requires significant care and a good understanding of how the language works. In particular, it helps to develop a habit of eyeing with mild suspicion anything that might turn out to be a function call—including user-defined operators, user-defined conversions, and silent temporary objects among the more subtle culprits—because any function call might throw.[9]

[9] Except for functions declared with an exception specification of throw() or certain functions in the standard library that are documented to never throw.

One way to greatly simplify an exception-safe container like Stack is to use better encapsulation. Specifically, we'd like to encapsulate the basic memory management work. Most of the care we had to take while writing our original exception-safe Stack was needed just to get the basic memory allocation right, so

Solution

We won't spend much time analyzing why the following functions are fully exception-safe (work properly in the presence of exceptions) and exception-neutral (propagate all exceptions to the caller), because the reasons are pretty much the same as those we discussed in detail in the first half of this miniseries. But do take a few minutes now to analyze these solutions, and note the commentary.

Constructor

The constructor is fairly straightforward. We'll use operator new() to allocate the buffer as raw memory. (Note that if we used a new-expression like new T[size], then the buffer would be initialized to default-constructed T objects, which was explicitly disallowed in the problem statement.)

```
template <class T>
StackImpl<T>::StackImpl( size_t size )
  : v_( static_cast<T*>
          ( size == 0
            ? 0
            : operator new(sizeof(T)*size) ) ),
    vsize_(size),
    vused_(0)
{
}
```

Destructor

The destructor is the easiest of the three functions to implement. Again, remember what we learned about operator delete() earlier in this miniseries. (See "Some Standard Helper Functions" for full details about functions such as destroy() and swap() that appear in the next few pieces of code.)

```
template <class T>
StackImpl<T>::~StackImpl()
{
    destroy( v_, v_+vused_ ); // this can't throw
    operator delete( v_ );
}
```

We'll see what destroy() is in a moment.

Some Standard Helper Functions

The Stack and StackImpl presented in this solution use three helper functions, one of which

Item 13. Writing Exception-Safe Code—Part 6

Difficulty: 9

And now for an even better Stack, with fewer requirements on T—not to mention a very elegant operator=().

Imagine that the /*????*/ comment in StackImpl stood for protected. Implement all the member functions of the following version of Stack, which is to be implemented in terms of StackImpl by using StackImpl as a private base class.

```
template <class T>
class Stack : private StackImpl<T>
{
public:
  Stack(size_t size=0);
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Count() const;
  void   Push(const T&);
  T&     Top();    // if empty, throws exception
  void   Pop();    // if empty, throws exception
};
```

As always, remember to make all the functions exception-safe and exception-neutral.

(Hint: There's a very elegant way to implement a fully safe operator=(). Can you spot it?)

Solution

The Default Constructor

Using the private base class method, our Stack class will look something like this (the code is shown inlined for brevity):

```
template <class T>
class Stack : private StackImpl<T>
{
public:
  Stack(size_t size=0)
    : StackImpl<T>(size)
  {
  }
```

Stack's default constructor simply calls the default constructor of StackImpl, that just sets the stack's state to empty and optionally performs an initial allocation. The only operation here that might throw is the new done in StackImpl's constructor, and that's unimportant when considering Stack's own exception safety. If it does happen, we won't enter the Stack constructor body and there will never have been a Stack object at all, so any initial allocation failures in the base class don't affect Stack. (See Item 8 and More Exceptional C++ Items 17 and 18, for additional comments about exiting constructors via an exception.)

Note that we slightly changed Stack's original constructor interface to allow a starting "hint" at the amount of memory to allocate. We'll make use of this in a minute when we write the Push function.

Guideline

Observe the canonical exception-safety rules: Always use the "resource acquisition is initialization" idiom to isolate resource ownership and management.

The Destructor

Here's the first elegance: We don't need to provide a Stack destructor. The default compiler-generated Stack destructor is fine, because it just calls the StackImpl destructor to destroy any objects that were constructed and actually free the memory. Elegant.

The Copy Constructor

Item 14. Writing Exception-Safe Code—Part 7

Difficulty: 5

Only a slight variant—of course, operator=() is still very nifty.

Imagine that the /*????*/ comment in StackImpl stood for public. Implement all the member functions of the following version of Stack, which is to be implemented in terms of StackImpl by using a StackImpl member object.

```
template <class T>
class Stack
{
public:
  Stack(size_t size=0);
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Count() const;
  void   Push(const T&);
  T&     Top();   // if empty, throws exception
  void   Pop();   // if empty, throws exception
private:
  StackImpl<T> impl_;  // private implementation
};
```

Don't forget exception safety.

Solution

This implementation of Stack is only slightly different from the last. For example, Count() returns impl_.vused_ instead of just an inherited vused_.

Here's the complete code:

```
template <class T>
class Stack
{
public:
  Stack(size_t size=0)
    : impl_(size)
  {
  }

  Stack(const Stack& other)
    : impl_(other.impl_.vused_)
  {
    while( impl_.vused_ < other.impl_.vused_ )
    {
      construct( impl_.v_+impl_.vused_,
                 other.impl_.v_[impl_.vused_] );
      ++impl_.vused_;
    }
  }

  Stack& operator=(const Stack& other)
  {
    Stack temp(other);
    impl_.Swap(temp.impl_); // this can't throw
    return *this;
  }

  size_t Count() const
  {
    return impl_.vused_;
  }

  void Push( const T& t )
  {
    if( impl_.vused_ == impl_.vsize_ )
    {
      Stack temp( impl_.vsize_*2+1 );
      while( temp.Count() < impl_.vused_ )
      {
        temp.Push( impl_.v_[temp.Count()] );
      }
      temp.Push( t );
      impl_.Swap( temp.impl_ );
    }
    else
    {
      construct( impl_.v_+impl_.vused_, t );
      ++impl_.vused_;
```

Item 15. Writing Exception-Safe Code—Part 8

Difficulty: 9

That's it—this is the final leg of the miniseries. The end of the line is a good place to stop and reflect, and that's just what we'll do for these last three problems.

1.

1. Which technique is better—using StackImpl as a private base class, or as a member object?
1.
2.

2. How reusable are the last two versions of Stack? What requirements do they put on T, the contained type? (In other words, what kinds of T can our latest Stack accept? The fewer the requirements are, the more reusable Stack will be.)
2.
3.

3. Should Stack provide exception specifications on its functions?
3.

Solution

Let's answer the questions one at a time.

1.

1. Which technique is better—using StackImpl as a private base class, or as a member object?

1. Both methods give essentially the same effect and nicely separate the two concerns of memory management and object construction/destruction.
1.

1. When deciding between private inheritance and containment, my rule of thumb is always to prefer the latter and use inheritance only when absolutely necessary. Both techniques mean "is implemented in terms of," and containment forces a better separation of concerns because the using class is a normal client with access to only the used class's public interface. Use private inheritance instead of containment only when absolutely necessary, which means when:
1.
   -

   - You need access to the class's protected members, or
   -
   -

   - You need to override a virtual function, or
   -
   -

   - The object needs to be constructed before other base subobjects[10]

   - [10] Admittedly, in this case it's tempting to use private inheritance anyway for syntactic convenience so that we don't have to write "impl_." in so many places.
   -
2.

2. How reusable are the last two versions of Stack? What requirements do they put on T, the contained type?

2. When writing a templated class, particularly something as potentially widely useful as a generic container, always ask yourself one crucial question: How reusable is my class? Or, to put it a different way: What constraints have I put upon users of the class, and do those constraints unduly limit what those users might want to reasonably do with my class?
2.

2. These Stack templates have two major differences from the first one we built. We've discussed one of the differences already. These latest Stacks decouple memory management from contained object construction and destruction, which is nice, but doesn't really affect users. However, there is another important difference. The new Stacks construct and destroy individual objects in place as needed,

Item 16. Writing Exception-Safe Code—Part 9

Difficulty: 8

How well do you understand the innocuous expression delete[] p? What are its implications when it comes to exception safety?

And now, for the topic you've been waiting for: "Destructors That Throw and Why They're Evil."

Consider the expression delete[] p;, where p points to a valid array on the free store, which was properly allocated and initialized using new[].

1.

1. What does delete[] p; really do?
1.
2.

2. How safe is it? Be as specific as possible.
2.

Solution

This brings us to a key topic, namely the innocent looking delete[] p;. What does it really do? And how safe is it?

Destructors That Throw and Why They're Evil

First, recall our standard destroy helper function (see the accompanying box):

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
  while( first != last )
  {
    destroy( &*first ); // calls "*first"'s destructor
    ++first;
  }
}
```

This was safe in our example above, because we required that T destructors never throw. But what if a contained object's destructor were allowed to throw? Consider what happens if destroy is passed a range of five objects. If the first destructor throws, then as it is written now, destroy will exit and the other four objects will never be destroyed. This is obviously not a good thing.

"Ah," you might interrupt, "but can't we clearly get around that by writing destroy to work properly in the face of T's whose destructors are allowed to throw?" Well, that's not as clear as one might think. For example, you'd probably start writing something like this:

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
  while( first != last )
  {
    try
    {
      destroy( &*first );
    }
    catch(...)
    {
      /* what goes here? */
    }
    ++first;
  }
}
```

The tricky part is the "what goes here?" There are really only three choices: the catch body rethrows the exception, it converts the exception by throwing something else, or it throws nothing and continues the loop.

Item 17. Writing Exception-Safe Code—Part 10

Difficulty: 9

The end—at last. Thank you for considering this miniseries. I hope you've enjoyed it.

At this point, you're probably feeling a little drained and more than a little tired. That's understandable. So here's a final question as a parting gift—it's designed to make everyone remember the equally (if not more) tired people who had to figure this stuff out on their own from first principles and then scrambled hard to get reasonable exception-safety guarantees put into the standard library at the last minute. It's appropriate at this time to repeat public thanks to Dave Abrahams, Greg Colvin, Matt Austern, and all the other "exceptional" people who helped get the current safety guarantees into the standard library—and who managed to complete the job literally days before the standard was frozen in November 1997, at the ISO WG21 / ANSI J16 meeting at Morristown, N.J., USA.

Is the C++ standard library exception-safe?

Explain.

Solution

Exception Safety and the Standard Library

Are the standard library containers exception-safe and exception-neutral? The short answer is: Yes.[14]

[14] Here, I'm focusing my attention on the containers and iterators portion of the standard library. Other parts of the library, such as iostreams and facets, are specified to provide at least the basic exception-safety guarantee.

- All iterators returned from standard containers are exception-safe and can be copied without throwing an exception.
- All standard containers must implement the basic guarantee for all operations: They are always destructible, and they are always in a consistent (if not predictable) state even in the presence of exceptions.
- To make this possible, certain important functions are required to implement the nothrow guarantee (are required not to throw)—including swap (the importance of which was illustrated by the example in the previous Item), allocator<T>::deallocate (the importance of which was illustrated by the discussion of operator delete() at the beginning of this miniseries) and certain operations of the template parameter types themselves (especially, the destructor, the importance of which was illustrated in Item 16 by the discussion headed "Destructors That Throw and Why They're Evil").
- All standard containers must also implement the strong guarantee for all operations (with two exceptions). They always have commit-or-rollback semantics so that an operation such as an insert either succeeds completely or else does not change the program state at all. "No change" also means that failed operations do not affect the validity of any iterators that happened to be already pointing into the container.
- There are only two exceptions to this point. First, for all containers, multi-element inserts ("iterator range" inserts) are never strongly exception-safe. Second, for vector<T> and deque<T> only, inserts and erases (whether single- or multi-element) are strongly exception-safe as long as T's copy constructor and assignment operator do not throw. Note the consequences of these particular limitations. Unfortunately, among other things, this means that inserting into and erasing from a vector<string> or a vector<vector<int> >, for example, are not strongly exception-safe.

Item 18. Code Complexity—Part 1

Difficulty: 9

This problem presents an interesting challenge: How many execution paths can there be in a simple three-line function? The answer will almost certainly surprise you.

How many execution paths could there be in the following code?

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    cout << e.First() << " " << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

To provide a little structure here, you should start by relying on the following three assumptions, and then try to expand on them.

1.

1. Different orders of evaluating function parameters are ignored, and failed destructors are ignored.
1.
2.

2. Called functions are considered atomic.
2.
3.

3. To count as a different execution path, an execution path must be made up of a unique sequence of function calls performed and exited in the same way.
3.

Solution

First, let's think about the implications of the given assumptions:

1.

1. Different orders of evaluating function parameters are ignored, and exceptions thrown by destructors are ignored.[15] Follow-up question for the intrepid: How many more execution paths are there if destructors are allowed to throw?

1. [15] Never allow an exception to propagate from a destructor. Code that does this can't be made to work well. See Item 16 for more about "destructors that throw and why they're evil."
1.
2.

2. Called functions are considered atomic. For example, the call "e.Title()" could throw for several reasons (for example, it could throw an exception itself, it could fail to catch an exception thrown by another function it has called, or it could return by value and the temporary object's constructor could throw). All that matters to the function is whether performing the operation e.Title() results in an exception being thrown or not.
2.
3.

3. To count as a different execution path, an execution path must be made up of a unique sequence of function calls performed and exited in the same way.
3.

So, how many possible execution paths are there? Answer: 23 (in just three lines of code!).

If you found:	**Rate yourself:**
3	Average
4–14	Exception-aware
15–23	Guru material

The 23 are made up of:

•

• 3 nonexceptional code paths

•

•

• 20 invisible exceptional code paths

•

By nonexceptional code paths I mean paths of execution that happen even if there are no exceptions thrown.

Item 19. Code Complexity—Part 2

Difficulty: 7

The challenge: Take the three-line function from Item 18 and make it exception-safe. This exercise illustrates some important lessons about exception safety.

Is the function from Item 18 exception-safe (works properly in the presence of exceptions) and exception-neutral (propagates all exceptions to the caller)?

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    cout << e.First() << " " << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

Explain your answer. If it is exception-safe, does it support the basic guarantee, the strong guarantee, or the nothrow guarantee? If not, how must it be changed to support one of these guarantees?

Assume that all called functions are strongly exception-safe (might throw but do not have side effects if they do throw), and that any objects being used, including temporaries, are exception-safe (clean up their resources when destroyed).

To recap the basic, strong, and nothrow guarantees, see Item 11. In brief, the basic guarantee ensures destructibility and no leaks; the strong guarantee, in addition, ensures full commit-or-rollback semantics; and the nothrow guarantee ensures that a function will not emit an exception.

Solution

First, before we get into the solution proper, a word about assumptions.

As the problem stated, we will assume that all called functions—including the stream functions—are strongly exception-safe (might throw but do not have side effects), and that any objects being used, including temporaries, are exception-safe (clean up their resources when destroyed).

Streams happen to throw a monkey wrench into this because of their possible "un-rollbackable" side effects. For example, operator<< might throw after emitting part of a string that can't be "un-emitted"; also, the stream's error state may be set. We will ignore those issues for the most part; the point of this discussion is to examine how to make a function exception-safe when the function is specified to have two distinct side effects.

So here's the question again: Is the function from Item 18 exception-safe (works properly in the presence of exceptions) and exception-neutral (propagates all exceptions to the caller)?

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    cout << e.First() << " " << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

As written, this function satisfies the basic guarantee: In the presence of exceptions, the function does not leak resources.

This function does not satisfy the strong guarantee. The strong guarantee says that if the function fails due to an exception, program state must not be changed. EvaluateSalaryAndReturnName(), however, has two distinct side effects (as hinted at in the function's name).

-
- An "...overpaid..." message is emitted to cout.
-
-
- A name string is returned.
-

As far as the second side effect is concerned, the function already meets the strong guarantee, because if an exception occurs the value will never be returned. As far as the first side effect is concerned, though, the

Class Design and Inheritance

Item 20. Class Mechanics

Difficulty: 7

How good are you at the details of writing classes? This item focuses not only on blatant errors, but even more so on professional style. Understanding these principles will help you to design classes that are more robust and easier to maintain.

You are doing a code review. A programmer has written the following class, which shows some poor style and has some real errors. How many can you find, and how would you fix them?

```
class Complex
{
public:
  Complex( double real, double imaginary = 0 )
    : _real(real), _imaginary(imaginary)
  {
  }
  void operator+ ( Complex other )
  {
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
  }
  void operator<<( ostream os )
  {
    os << "(" << _real << "," << _imaginary << ")";
  }
  Complex operator++()
  {
    ++_real;
    return *this;
  }
  Complex operator++( int )
  {
    Complex temp = *this;
    ++_real;
    return temp;
  }
private:
  double _real, _imaginary;
};
```

Solution

This class has a lot of problems—even more than I will explicitly show here. The point of this puzzle is primarily to highlight class mechanics (issues such as "what is the canonical form of operator<<?" and "should operator+ be a member?") rather than point out where the interface is just plain poorly designed. However, I will start off with perhaps the most useful observation: Why write a Complex class when one already exists in the standard library? And, what's more, the standard one isn't plagued with any of the following problems and has been crafted based on years of practice by the best people in our industry. Humble thyself and reuse.

Guideline

Reuse code—especially standard library code—instead of handcrafting your own. It's faster, easier, and safer.

Perhaps the best way to fix the problems in the Complex code is to avoid using the class at all and use the std::complex template instead.

Having said that, let's go through the class as written and fix the problems as we go. First, the constructor:

1.

1. The constructor allows an implicit conversion.

1.
```
Complex( double real, double imaginary = 0 )
  : _real(real), _imaginary(imaginary)
{
}
```

1. Because the second parameter has a default value, this function can be used as a single-parameter constructor and, hence, as an implicit conversion from double to Complex. In this case, it's probably fine, but as we saw in Item 6, the conversion may not always be intended. In general it's a good idea to make your constructors explicit by default unless you deliberately decide to allow the implicit conversion. (See Item 39 for more about implicit conversions.)

1. Guideline
1.

Watch out for hidden temporaries created by implicit conversions. One good way to avoid this is to make constructors explicit when possible, and avoid writing conversion operators.

Item 21. Overriding Virtual Functions

Difficulty: 6

Virtual functions are a pretty basic feature, but they occasionally harbor subtleties that trap the unwary. If you can answer questions like this one, then you know virtual functions cold, and you're less likely to waste a lot of time debugging problems like the ones illustrated below.

In your travels through the dusty corners of your company's code archives, you come across the following program fragment written by an unknown programmer. The programmer seems to have been experimenting to see how some C++ features work. What did the programmer probably expect the program to print, and what is the actual result?

```
#include <iostream>
#include <complex>
using namespace std;
class Base
{
public:
  virtual void f( int );
  virtual void f( double );
  virtual void g( int i = 10 );
};
void Base::f( int )
{
  cout << "Base::f(int)" << endl;
}
void Base::f( double )
{
  cout << "Base::f(double)" << endl;
}
void Base::g( int i )
{
  cout << i << endl;
}
class Derived: public Base
{
public:
  void f( complex<double> );
  void g( int i = 20 );
};
void Derived::f( complex<double> )
{
  cout << "Derived::f(complex)" << endl;
}
void Derived::g( int i )
{
  cout << "Derived::g() " << i << endl;
}
void main()
{
  Base    b;
```

Solution

First, let's consider some style issues, and one real error.

1.

1. "void main()" is nonstandard and therefore nonportable.

1.
```
void main()
```

1. Yes, alas, I know that this appears in many books. Some authors even argue that "void main()" might be standard-conforming. It isn't. It never was, not even in the 1970s, in the original pre-standard C.
1.

1. Even though "void main()" is not one of the legal declarations of main, many compilers allow it. What this means, however, is that even if you are able to write "void main()" today on your current compiler, you may not be able to write it when you port to a new compiler. It's best to get into the habit of using either of the two standard and portable declarations of main:
1.

```
int main()
int main( int argc, char* argv[] )
```

1. You might also have noticed that the problem program does not have a return statement in main. Even with a standard main() that returns an int, that's not a problem (though it's certainly good style to report errors to outside callers), because if main() has no return statement, the effect is that of executing "return 0;". Caveat: As of this writing, many compilers still do not implement this rule and will emit warnings if you don't put an explicit return statement in main().
1.
2.

2. "delete pb;" is unsafe.

2.
```
delete pb;
```

2. This looks innocuous. It would be innocuous, if only the writer of Base had supplied a virtual destructor. As it is, deleting via a pointer-to-base without a virtual destructor is evil, pure and simple, and corruption is the best thing you can hope for, because the wrong destructor will get called and operator delete() will be invoked with the wrong object size.

2. Guideline
2.

Make base class destructors virtual (unless you are certain that no one will ever attempt to delete a derived object through a pointer to base).

Item 22. Class Relationships—Part 1

Difficulty: 5

How are your OO design skills? This Item illustrates a common class design mistake that still catches many programmers.

A networking application has two kinds of communications sessions, each with its own message protocol. The two protocols have similarities (some computations and even some messages are the same), so the programmer has come up with the following design to encapsulate the common computations and messages in a BasicProtocol class.

```
class BasicProtocol /* : possible base classes */
{
public:
  BasicProtocol();
  virtual ~BasicProtocol();
  bool BasicMsgA( /*...*/ );
  bool BasicMsgB( /*...*/ );
  bool BasicMsgC( /*...*/ );
};

class Protocol1 : public BasicProtocol
{
public:
  Protocol1();
  ~Protocol1();
  bool DoMsg1( /*...*/ );
  bool DoMsg2( /*...*/ );
  bool DoMsg3( /*...*/ );
  bool DoMsg4( /*...*/ );
};

class Protocol2 : public BasicProtocol
{
public:
  Protocol2();
  ~Protocol2();
  bool DoMsg1( /*...*/ );
  bool DoMsg2( /*...*/ );
  bool DoMsg3( /*...*/ );
  bool DoMsg4( /*...*/ );
  bool DoMsg5( /*...*/ );
};
```

Each DoMsg () member function calls the BasicProtocol::Basic () functions as needed to perform the common work, but the DoMsg () function performs the actual transmissions itself. Each class may have additional members, but you can assume that all significant members are shown.

Solution

This Item illustrates a too-common pitfall in OO class relationship design. To recap, classes Protocol1 and Protocol2 are publicly derived from a common base class BasicProtocol, which performs some common work.

A key to identifying the problem is the following:

Each DoMsg () member function calls the BasicProtocol::Basic () functions as needed to perform the common work, but the DoMsg () function performs the actual transmissions itself.

Here we have it: This is clearly describing an "is implemented in terms of " relationship, which, in C++, is spelled either "private inheritance" or "membership." Unfortunately, many people still frequently misspell it as "public inheritance," thereby confusing implementation inheritance with interface inheritance. The two are not the same thing, and that confusion is at the root of the problem here.

Common Mistake

Never use public inheritance except to model true Liskov IS-A and WORKS-LIKE-A. All overridden member functions must require no more and promise no less.

Incidentally, programmers in the habit of making this mistake (using public inheritance for implementation) usually end up creating deep inheritance hierarchies. This greatly increases the maintenance burden by adding unnecessary complexity, forcing users to learn the interfaces of many classes even when all they want to do is use a specific derived class. It can also have an impact on memory use and program performance by adding unnecessary vtables and indirections to classes that do not really need them. If you find yourself frequently creating deep inheritance hierarchies, you should review your design style to see if you've picked up this bad habit. Deep hierarchies are rarely needed and almost never good. And if you don't believe that but think that "OO just isn'tOO without lots of inheritance," then a good counter-example to consider is the standard library itself.

Guideline

Never inherit publicly to reuse code (in the base class); inherit publicly in order to be reused (by code that uses base objects polymorphically).[4]

Item 23. Class Relationships—Part 2

Difficulty: 6

Design patterns are an important tool in writing reusable code. Do you recognize the patterns used in this Item? If so, can you improve them?

A database manipulation program often needs to do some work on every record (or selected records) in a given table, by first performing a read-only pass through the table to cache information about which records need to be processed, and then performing a second pass to actually make the changes.

Instead of rewriting much of the common logic each time, a programmer has tried to provide a generic reusable framework in the following abstract class. The intent is that the abstract class should encapsulate the repetitive work by first, compiling a list of table rows on which work needs to be done, and second, by performing the work on each affected row. Derived classes are responsible for providing the details of their specific operations.

```
//--------------------------------------------------
// File gta.h
//--------------------------------------------------
class GenericTableAlgorithm
{
public:
  GenericTableAlgorithm( const string& table );
  virtual ~GenericTableAlgorithm();

  // Process() returns true if and only if successful.
  // It does all the work: a) physically reads
  // the table's records, calling Filter() on each
  // to determine whether it should be included
  // in the rows to be processed; and b) when the
  // list of rows to operate upon is complete, calls
  // ProcessRow() for each such row.
  //
  bool Process();

private:
  // Filter() returns true if and only if the row should be
  // included in the ones to be processed. The
  // default action is to return true (to include
  // every row).
  //
  virtual bool Filter( const Record& );

  // ProcessRow() is called once per record that
  // was included for processing. This is where
  // the concrete class does its specialized work.
  // (Note: This means every row to be processed
  // will be read twice, but assume that that is
  // necessary and not an efficiency consideration.)
```

Solution

Let's consider the questions one by one.

1.

1. This is a good design and implements a well-known design pattern. Which pattern is this? Why is it useful here?

1. This is the Template Method (Gamma95) pattern (not to be confused with C++ templates). It's useful because we can generalize a common way of doing something that always follows the same steps. Only the details may differ and can be supplied by a derived class. Further, a derived class may choose to reapply the same Template Method approach—that is, it may override the virtual function as a wrapper around a new virtual function—so different steps can be filled in at different levels in the class hierarchy.
1.

1. (Note: The Pimpl Idiom is superficially similar to Bridge, but here it's only intended to hide this particular class's own implementation as a compilation dependency firewall, not to act as a true extensible bridge. We'll analyze the Pimpl Idiom in depth in Items 26 through 30.)

1. Guideline
1.

Avoid public virtual functions; prefer using the Template Method pattern instead.

1. Guideline
1.

Know about and use design patterns.

2.

2. Without changing the fundamental design, critique the way this design was executed. What might you have done differently? What is the purpose of the pimpl_ member?

2. This design uses bools as return codes, with apparently no other way (status codes or exceptions) of reporting failures. Depending on the requirements, this may be fine, but it's something to note.
2.

2. The (intentionally pronounceable) pimpl_ member nicely hides the implementation behind an opaque pointer. The struct that pimpl_ points to will contain the private member functions and member variables so that any change to them will not require client code to recompile. This is an important

Item 24. Uses and Abuses of Inheritance

Difficulty: 6

"Why inherit?"

Inheritance is often overused, even by experienced developers. Always minimize coupling. If a class relationship can be expressed in more than one way, use the weakest relationship that's practical. Given that inheritance is nearly the strongest relationship you can express in C++ (second only to friendship), it's only really appropriate when there is no equivalent weaker alternative.

In this Item, the spotlight is on private inheritance, one real (if obscure) use for protected inheritance, and a recap of the proper use of public inheritance. Along the way, we will create a fairly complete list of all the usual and unusual reasons to use inheritance.

The following template provides list-management functions, including the ability to manipulate list elements at specific list locations.

```
// Example 1
//
template <class T>
class MyList
{
public:
  bool   Insert( const T&, size_t index );
  T      Access( size_t index ) const;
  size_t Size() const;
private:
  T*     buf_;
  size_t bufsize_;
};
```

Consider the following code, which shows two ways to write a MySet class in terms of MyList. Assume that all important elements are shown.

```
// Example 1(a)
//
template <class T>
class MySet1 : private MyList<T>
{
public:
  bool   Add( const T& ); // calls Insert()
  T      Get( size_t index ) const;
                         // calls Access()
  using MyList<T>::Size;
  //...
};
```

Solution

This motivating example helps to illustrate some of the issues surrounding the use of inheritance, especially how to choose between nonpublic inheritance and containment.

The answer to Question 1—Is there any difference between MySet1 and MySet2—is straightforward: There is no substantial difference between MySet1 and MySet2. They are functionally identical.

Question 2 gets us right down to business: More generally, what is the difference between nonpublic inheritance and containment? Make as comprehensive a list as you can of reasons why you would use inheritance instead of containment.

- Nonpublic inheritance should always express IS-IMPLEMENTED-IN-TERMS-OF (with only one rare exception, which I'll cover shortly). It makes the using class depend upon the public and protected parts of the used class.
- Containment always expresses HAS-A and, therefore, IS-IMPLEMENTED-IN-TERMS-OF. It makes the using class depend upon only the public parts of the used class.

It's easy to show that inheritance is a superset of single containment—that is, there's nothing we can do with a single MyList<T> member that we couldn't do if we inherited from MyList<T>. Of course, using inheritance does limit us to having just one MyList<T> (as a base subobject); if we needed to have multiple instances of MyList<T>, we would have to use containment instead.

Guideline

Prefer containment (a.k.a. "composition", "layering", "HAS-A", "delegation") to inheritance. When modeling IS-IMPLEMENTED-IN-TERMS-OF, always prefer expressing it using containment, not inheritance.

That being the case, what are the extra things we can do if we use inheritance that we can't do if we use containment? In other words, why use nonpublic inheritance? Here are several reasons, in rough order from most to least common. Interestingly, the final item points out a useful(?) application of protected inheritance.

I l@ve RuBoard PREVIOUS NEXT

I l@ve RuBoard PREVIOUS NEXT

Item 25. Object-Oriented Programming

Difficulty: 4

Is C++ an object-oriented language? It both is and is not, contrary to popular opinion. For a change of pace (and to get a brief break from staring at code), here is an essay question.

Don't just turn quickly to the answer—give it some thought.

"C++ is a powerful language that provides many advanced object-oriented constructs, including encapsulation, exception handling, inheritance, templates, polymorphism, strong typing, and a complete module system."

Discuss.

I l@ve RuBoard PREVIOUS NEXT

I l@ve RuBoard PREVIOUS NEXT

Solution

The purpose of this Item was to provoke thought about major (and even missing) features in C++, and to provide a healthy dose of reality. Specifically, I hope that your thinking about this Item has generated ideas and examples that illustrate three main things.

1.

1. Not everyone agrees about what OO means. "Well, what is object orientation, anyway?" Even at this late date, if you ask 10 people, you're likely to get 15 different answers. (Not much better than asking 10 lawyers for legal opinions.)
1.

1. Just about everyone would agree that inheritance and polymorphism are OO concepts. Most people would include encapsulation. A few might include exception handling, and perhaps no one would include templates. The point is that there are differing opinions on whether any given feature is OO, and each viewpoint has its passionate defenders.
1.
2.

2. C++ is a multiparadigm language. C++ is not just an OO language. It supports many OO features, but it doesn't force programmers to use them. You can write completely non-OO programs in C++, and many people do.
2.

2. The C++ standardization effort's most important contribution to C++ is stronger support for powerful abstraction to reduce software complexity (Martin95).[12] C++ is not solely an object-oriented language. It supports several programming styles, including both object-oriented programming and generic programming. These styles are fundamentally important because each provides flexible ways to organize code through abstraction. Object-oriented programming lets us bundle an object's state together with the functions that manipulate it, and encapsulation and inheritance let us manage interdependencies and make reuse cleaner and easier. Generic programming is a more recent style that lets us write functions and classes that operate on other functions and objects of unspecified, unrelated, and unknown types, providing a unique way to reduce coupling and interdependencies within a program. A few other languages currently provide support for genericity, but none yet support it as strongly as C++. Indeed, modern generic programming was made possible by the unique C++ formulation of templates.

2. [12] The book contains an excellent discussion of why one of object-oriented programming's most important benefits is that it lets us reduce software complexity by managing code interdependencies.
2.

2. Today, C++ provides many powerful ways to express abstraction, and the resulting flexibility is the most important result of C++ standardization.
2.
3.

3. No language is the be-all and end-all. Today, I'm using C++ as my primary programming language;

Compiler Firewalls and the Pimpl Idiom

Managing dependencies well is an essential part of writing solid code. I've argued (Sutter98) that C++'s greatest strength is that it supports two powerful methods of abstraction: object-oriented programming and generic programming. Both are fundamentally tools to help manage dependencies and, therefore, manage complexity. It's telling that all of the common OO/generic buzzwords—including encapsulation, polymorphism, and type independence—and all the design patterns I know of describe ways to manage complexity within a software system by managing the code's interdependencies.

Item 26. Minimizing Compile-time Dependencies—Part 1

Difficulty: 4

When we talk about dependencies, we usually think of run-time dependencies like class interactions. In this Item, we will focus instead on how to analyze and manage compile-time dependencies. As a first step, try to identify (and root out) unnecessary headers.

Many programmers habitually #include many more headers than necessary. Unfortunately, doing so can seriously degrade build times, especially when a popular header file includes too many other headers.

In the following header file, which #include directives could be immediately removed without ill effect? You may not make any changes other than removing or rewriting #include directives. Note that the comments are important.

```
//  x.h: original header
//
#include <iostream>
#include <ostream>
#include <list>
// None of A, B, C, D or E are templates.
// Only A and C have virtual functions.
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D
#include "e.h"  // class E
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  std::list<C> clist_;
  D            d_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Solution

Of the first two standard headers mentioned in x.h, one can be immediately removed because it's not needed at all, and the second can be replaced with a smaller header.

1.

1. Remove iostream.

1. `#include <iostream>`

1. Many programmers #include <iostream> purely out of habit as soon as they see anything resembling a stream nearby. X does make use of streams, that's true; but it doesn't mention anything specifically from iostream. At the most, X needs ostream alone, and even that can be whittled down.

1. Guideline
1.

Never #include unnecessary header files.

2.

2. Replace ostream with iosfwd.

2. `#include <ostream>`

2. Parameter and return types need only to be forward-declared, so instead of the full definition of ostream we really only need its forward declaration.
2.

In the old days, you could just replace "#include <ostream>" with "class ostream;" in this situation, because ostream used to be a class and it wasn't in namespace std. Alas, no more. Writing "class ostream;" is illegal for two reasons.

1.

1. ostream is now in namespace std, and programmers aren't allowed to declare anything that lives in namespace std.
1.
2.

2. ostream is now a typedef of a template; specifically, it's typedef'd as basic_ostream<char>. Not only would the basic_ostream template be messy to forward-declare in any case, but you couldn't reliably forward-declare it at all, because library implementations are allowed to do things like add

Item 27. Minimizing Compile-time Dependencies—Part 2

Difficulty: 6

Now that the unnecessary headers have been removed, it's time for Phase 2: How can you limit dependencies on the internals of a class?

Below is how the header from Item 26 looks after the initial clean-up pass. What further #includes could be removed if we made some suitable changes, and how?

This time, you may make changes to X as long as X's base classes and its public interface remain unchanged; any current code that already uses X should not be affected beyond requiring a simple recompilation.

```
//  x.h: sans gratuitous headers
//
#include <iosfwd>
#include <list>
// None of A, B, C or D are templates.
// Only A and C have virtual functions.
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D
class E;
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  std::list<C> clist_;
  D            d_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Solution

There are a couple of things we weren't able to do in the previous problem.

- We had to leave a.h and b.h. We couldn't get rid of these because X inherits from both A and B, and you always have to have full definitions for base classes so that the compiler can determine X's object size, virtual functions, and other fundamentals. (Can you anticipate how to remove one of these? Think about it: Which one can you remove, and why/how? The answer will come shortly.)
- We had to leave list, c.h, and d.h. We couldn't get rid of these right away because a list<C> and a D appear as private data members of X. Although C appears as neither a base class nor a member, it is being used to instantiate the list member, and most current compilers require that when you instantiate list<C>, you are able to see the definition of C. (The standard doesn't require a definition here, though, so even if the compiler you are currently using has this restriction, you can expect the restriction to go away over time.)

Now let's talk about the beauty of Pimpls. (Yes, really.)

C++ lets us easily encapsulate the private parts of a class from unauthorized access. Unfortunately, because of the header file approach inherited from C, it can take a little more work to encapsulate dependencies on a class's privates. "But," you say, "the whole point of encapsulation is that the client code shouldn't have to know or care about a class's private implementation details, right?" Right, and in C++, the client code doesn't need to know or care about access to a class's privates (because unless it's a friend, it isn't allowed any), but because the privates are visible in the header, the client code does have to depend upon any types they mention.

How can we better insulate clients from a class's private implementation details? One good way is to use a special form of the handle/body idiom (Coplien92) (what I call the Pimpl Idiom because of the intentionally pronounceable pimpl_ pointer[1]) as a compilation firewall (Lakos96, Meyers98, Meyers99, Murray93).

[1] I always used to write impl_. The eponymous pimpl_ was actually coined several years ago by Jeff Sumner (chief programmer at PeerDirect), due in equal parts to a penchant for Hungarian-style "p" prefixes for pointer variables and an occasional taste for horrid puns.

A Pimpl is just an opaque pointer (a pointer to a forward-declared, but undefined, helper class) used to hide the private members of a class. That is, instead of writing:

```
// file x.h
class X
```

Item 28. Minimizing Compile-time Dependencies—Part 3

Difficulty: 7

Now the unnecessary headers have been removed, and needless dependencies on the internals of the class have been eliminated. Is there any further decoupling that can be done? The answer takes us back to basic principles of solid class design.

The Incredible Shrinking Header has now been greatly trimmed, but there may still be ways to reduce the dependencies further. What further #includes could be removed if we made further changes to X, and how?

This time, you may make any changes to X as long as they don't change its public interface so that existing code that uses X is unaffected beyond requiring a simple recompilation. Again, note that the comments are important.

```
//  x.h: after converting to use a Pimpl
//       to hide implementation details
//

#include <iosfwd>
#include "a.h"  // class A (has virtual functions)
#include "b.h"  // class B (has no virtual functions)
class C;
class E;

class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  struct XImpl;
  XImpl* pimpl_;
    // opaque pointer to forward-declared class
};

inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Solution

In my experience, many C++ programmers still seem to march to the "it isn't OO unless you inherit" battle hymn, by which I mean that they use inheritance more than necessary. See Item 24 for the whole exhausting lecture; the bottom line is simply that inheritance (including, but not limited to, IS-A) is a much stronger relationship than HAS-A or USES-A. When it comes to managing dependencies, therefore, you should always prefer composition/membership over inheritance. To paraphrase Albert Einstein: "Use as strong a relationship as necessary, but no stronger."

In this code, X is derived publicly from A and privately from B. Recall that public inheritance should always model IS-A and satisfy the Liskov Substitution Principle.[2] In this case, X IS-A A and there's naught wrong with it, so we'll leave that as it is.

[2] For lots of good discussion about applying the LSP, see the papers available online at www.gotw.ca/publications/xc++/om.htm, as well as Martin95.

But did you notice the interesting thing about B?

The interesting thing about B is this: B is a private base class of X, but B has no virtual functions. Now, usually, the only reason you would choose private inheritance over composition/membership is to gain access to protected members—which most times means "to override a virtual function."[3] As we see, B has no virtual functions, so there's probably no reason to prefer the stronger relationship of inheritance (unless X needs access to some protected function or data in B, of course, but for now I'll assume this is not the case). Assuming that is, indeed, the case, however, instead of having a base subobject of type B, X probably ought to have simply a member object of type B. Therefore, the way to further simplify the header is remove unnecessary inheritance from class B.

[3] Yes, there are other possible reasons to inherit, but the situations where those arise are rare and/or obscure. See Sutter98(a) and Sutter99 for an extensive discussion of the (few) reasons to use inheritance of any kind. Those articles point out in detail why containment/membership should often be used instead of inheritance.

```
#include "b.h"  // class B (has no virtual functions)
```

Because the B member object should be private (it is, after all, an implementation detail), this member should live in X's hidden pimpl_ portion.

Guideline

Item 29. Compilation Firewalls

Difficulty: 6

Using the Pimpl Idiom can dramatically reduce code interdependencies and build times. But what should go into a pimpl_ object, and what is the safest way to use it?

In C++, when anything in a class definition changes (even private members), all users of that class must be recompiled. To reduce these dependencies, a common technique is to use an opaque pointer to hide some of the implementation details.

```
class X
{
public:
  /* ... public members ... */
protected:
  /* ... protected members? ... */
private:
  /* ... private members? ... */
  struct XImpl;
  XImpl* pimpl_;         // opaque pointer to
                         // forward-declared class
};
```

The questions for you to answer are:

1.

1. What should go into XImpl? There are four common disciplines.
   -
   -
   - Put all private data (but not functions) into XImpl.
   -
   -
   - Put all private members into XImpl.
   -
   -
   - Put all private and protected members into XImpl.
   -
   -
   - Make XImpl entirely the class that X would have been, and write X as only the public interface made up entirely of simple forwarding functions (a handle/body variant).

Solution

Class X uses a variant of the handle/body idiom. As documented by Coplien (Coplien92), handle/body was described as primarily useful for reference counting of a shared implementation, but it also has more-general implementation-hiding uses. For convenience, from now on I'll call X the "visible class" and XImpl the "Pimpl class."

One big advantage of this idiom is that it breaks compile-time dependencies. First, system builds run faster, because using a Pimpl can eliminate extra #includes as demonstrated in Items 27 and 28. I have worked on projects in which converting just a few widely-visible classes to use Pimpls halved the system's build time. Second, it localizes the build impact of code changes because the parts of a class that reside in the Pimpl can be freely changed—that is, members can be freely added or removed—without recompiling client code.

Let's answer the Item questions one at a time.

1.

1. What should go into XImpl?

1. Option 1 (Score: 6 / 10): Put all private data (but not functions) into XImpl. This is a good start, because it hides any class that appeared only as a data member. Still, we can usually do better.
1.

1. Option 2 (Score: 10 / 10): Put all nonvirtual private members into XImpl. This is (almost) my usual practice these days. After all, in C++, the phrase "client code shouldn't and doesn't care about these parts" is spelled "private"—and privates are always hidden.[4]
1.

1. [4] Except in some liberal European countries.
1.

1. There are some caveats.
1.
   -

   - You can't hide virtual member functions in the Pimpl, even if the virtual functions are private. If the virtual function overrides one inherited from a base class, then it must appear in the actual derived class. If the virtual function is not inherited, then it must still appear in the visible class in order to be available for overriding by further derived classes.

   - Virtual functions should normally be private, except that they have to be protected if a derived class's version needs to call the base class's version (for example, for a virtual DoWrite() persistence function).
   -
   -

Item 30. The "Fast Pimpl" Idiom

Difficulty: 6

It's sometimes tempting to cut corners in the name of "reducing dependencies" or in the name of "efficiency," but it may not always be a good idea. Here's an excellent idiom to accomplish both objectives simultaneously and safely.

Standard malloc and new calls are relatively expensive.[5] In the code below, the programmer originally has a data member of type X in class Y.

[5] Compared with other typical operations, such as function calls.

```
// Attempt #1
//
// file y.h
#include "x.h"
class Y
{
  /*...*/
  X x_;
};

// file y.cpp
Y::Y() {}
```

This declaration of class Y requires the declaration of class X to be visible (from x.h). To avoid this, the programmer first tries to write:

```
// Attempt #2
//
// file y.h
class X;
class Y
{
  /*...*/
  X* px_;
};

// file y.cpp
#include "x.h"
Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

This nicely hides X, but it turns out that Y is used very widely and the dynamic allocation overhead is degrading performance.

Solution

Let's answer the Item questions one at a time.

1.

1. What is the Pimpl Idiom's space overhead?

1. "What space overhead?" you ask? Well, we now need space for at least one extra pointer (and possibly two, if there's a back pointer in XImpl) for every X object. This typically adds at least 4 (or 8) bytes on many popular systems, and possibly as many as 14 bytes or more, depending on alignment requirements. For example, try the following program on your favorite compiler.
1.

```
struct X { char c; struct XImpl; XImpl* pimpl_; };
struct X::XImpl { char c; };
int main()
{
  cout << sizeof(X::XImpl) << endl
       << sizeof(X) << endl;
}
```

1. On many popular compilers that use 32-bit pointers, this prints:
1.

```
1
8
```

1. On these compilers, the overhead of storing one extra pointer was actually 7 bytes, not 4. Why? Because the platform on which the compiler is running requires a pointer to be stored on a 4-byte boundary, or else it performs much more poorly if the pointer isn't stored on such a boundary. Knowing this, the compiler allocates 3 bytes of unused/empty space inside each X object, which means the cost of adding a pointer member was actually 7 bytes, not 4. If a back pointer is also needed, then the total storage overhead can be as high as 14 bytes on a 32-bit machine, as high as 30 bytes on a 64-bit machine, and so on.
1.

1. How do we get around this space overhead? The short answer is: We can't eliminate it, but sometimes we can minimize it.
1.

1. The longer answer is: There's a downright reckless way to eliminate it that you should never, ever use (and don't tell anyone that you heard it from me), and there's usually a nonportable, but correct, way to minimize it. The utterly reckless "space optimization" happens to be the same as the utterly reckless "performance optimization," so I've moved that discussion off to the side; see the upcoming sidebox "Reckless Fixes and Optimizations, and Why They're Evil."
1.

1. If (and only if) the space difference is actually important in your program, then the nonportable, but correct, way to minimize the pointer overhead is to use compiler-specific #pragmas. Many

Name Lookup, Namespaces, and the Interface Principle

Item 31. Name Lookup and the Interface Principle—Part 1

Difficulty: 9

When you call a function, which function do you call? The answer is determined by name lookup, but you're almost certain to find some of the details surprising.

In the following code, which functions are called? Why? Analyze the implications.

```
namespace A
{
  struct X;
  struct Y;
  void f( int );
  void g( X );
}

namespace B
{
  void f( int i )
  {
    f( i );   // which f()?
  }
  void g( A::X x )
  {
    g( x );   // which g()?
  }
  void h( A::Y y )
  {
    h( y );   // which h()?
  }
}
```

Solution

Two of the three cases are (fairly) obvious, but the third requires a good knowledge of C++'s name lookup rules—in particular, Koenig lookup.

Let's start simple.

```
namespace A
{
  struct X;
  struct Y;
  void f( int );
  void g( X );
}
namespace B
{
  void f( int i )
  {
    f( i );   // which f()?
  }
```

This f() calls itself, with infinite recursion. The reason is that the only visible f() is B::f() itself.

There is another function with signature f(int), namely the one in namespace A. If B had written "using namespace A;" or "using A::f;", then A::f(int) would have been visible as a candidate when looking up f(int), and the f(i) call would have been ambiguous between A::f(int) and B::f(int). Since B did not bring A::f(int) into scope, however, only B::f(int) can be considered, so the call unambiguously resolves to B::f(int).

And now for a twist:

```
void g( A::X x )
{
  g( x );   // which g()?
}
```

This call is ambiguous between A::g(X) and B::g(X). The programmer must explicitly qualify the call with the appropriate namespace name to get the g() he wants.

You may at first wonder why this should be so. After all, as with f(), B hasn't written "using" anywhere to bring A::g(X) into its scope, so you'd think that only B::g(X) would be visible, right? Well, this would be true except for an extra rule that C++ uses when looking up names:

Koenig Lookup[1] (simplified): If you supply a function argument of class type (here x, of type A::X), then

Item 32. Name Lookup and the Interface Principle—Part 2

Difficulty: 9

What's in a class? That is, what is "part of" a class and its interface?

Hint #1: Clearly nonstatic member functions are tightly coupled to the class, and public nonstatic member functions make up part of the class's interface. What about static member functions? What about free functions?

Hint #2: Take some time to consider the implications of Item 31.

Solution

We'll start off with the deceptively simple question: What functions are "part of" a class, or make up the interface of a class?

The deeper questions are:

-
- How does this answer fit with C-style object-oriented programming?
-
-
- How does it fit with C++'s Koenig lookup? With the Myers Example? (I'll describe both.)
-
-
- How does it affect the way we analyze class dependencies and design object models?
-

So what's in a class? Here's the definition of "class" again:

A class describes a set of data, along with the functions that operate on that data.

Programmers often unconsciously misinterpret this definition, saying instead: "Oh yeah, a class, that's what appears in the class definition—the member data and the member functions." But that's not the same thing, because it limits the word functions to mean just member functions. Consider:

```
//*** Example 1 (a)
class X { /*...*/ };
/*...*/
void f( const X& );
```

The question is: Is f part of X? Some people will automatically say "No" because f is a nonmember function (or free function). Others might realize something fundamentally important: If the Example 1 (a) code appears together in one header file, it is not significantly different from:

```
//*** Example 1 (b)
class X
{
  /*...*/
public:
  void f() const;
};
```

Item 33. Name Lookup and the Interface Principle—Part 3

Difficulty: 5

Take a few minutes to consider some implications of the Interface Principle on the way we reason about program design. We'll revisit a classic design problem: "What's the best way to write operator<<()?"

There are two main ways to write operator<<() for a class: as a free function that uses only the usual interface of the class (and possibly accessing nonpublic parts directly as a friend), or as a free function that calls a virtual Print() helper function in the class.

Which method is better? What are the tradeoffs?

Solution

Are you wondering why a question like this gets a title like "Name Lookup–Part 3"? If so, you'll soon see why, as we consider an application of the Interface Principle discussed in the previous Item.

What Does a Class Depend On?

"What's in a class?" isn't just a philosophical question. It's a fundamentally practical question, because without the correct answer, we can't properly analyze class dependencies.

To demonstrate this, consider a seemingly unrelated problem: What's the best way to write operator<< for a class? There are two main ways, both of which involve tradeoffs. I'll analyze both. In the end we'll find that we're back to the Interface Principle and that it has given us important guidance to analyze the tradeoffs correctly.

Here's the first way:

```
//*** Example 5 (a) -- nonvirtual streaming
class X
{
  /*...ostream is never mentioned here...*/
};
ostream& operator<<( ostream& o, const X& x )
{
  /* code to output an X to a stream */
  return o;
}
```

Here's the second:

```
//*** Example 5 (b) -- virtual streaming
class X
{
  /*...*/
public:
  virtual ostream& print( ostream& ) const;
};
ostream& X::print( ostream& o ) const
{
  /* code to output an X to a stream */
  return o;
}
ostream& operator<<( ostream& o, const X& x )
{
  return x.print( o );
}
```

Item 34. Name Lookup and the Interface Principle—Part 4

Difficulty: 9

We conclude this miniseries by considering some implications of the Interface Principle on name lookup. Can you spot the (quite subtle) problem lurking in the following code?

1.

1. What is name hiding? Show how it can affect the visibility of base class names in derived classes.
1.
2.

2. Will the following example compile correctly? Make your answer as complete as you can. Try to isolate and explain any areas of doubt.
2.

```
// Example 2: Will this compile?
//
// In some library header:
namespace N { class C {}; }
int operator+(int i, N::C) { return i+1; }
// A mainline to exercise it:
#include <numeric>
int main()
{
  N::C a[10];
  std::accumulate(a, a+10, 0);
}
```

Solution

Let's recap a familiar inheritance issue: name hiding, by answering question 1 in the Item:

1.

1. What is name hiding? Show how it can affect the visibility of base class names in derived classes.
1.

Name Hiding

Consider the following example:

```
// Example 1a: Hiding a name
//             from a base class
//
struct B
{
  int f( int );
  int f( double );
  int g( int );
};
struct D : public B
{
private:
  int g( std::string, bool );
};
D   d;
int i;
d.f(i);  // ok, means B::f(int)
d.g(i);  // error: g takes 2 args
```

Most of us should be used to seeing this kind of name hiding, although the fact that the last line won't compile surprises most new C++ programmers. In short, when we declare a function named g in the derived class D, it automatically hides all functions with the same name in all direct and indirect base classes. It doesn't matter a whit that D::g "obviously" can't be the function that the programmer meant to call (not only does D::g have the wrong signature, but it's private and, therefore, inaccessible to boot), because B::g is hidden and can't be considered by name lookup.

To see what's really going on, let's look in a little more detail at what the compiler does when it encounters the function call d.g(i). First, it looks in the immediate scope, in this case the scope of class D, and makes a list of all functions it can find that are named g (regardless of whether they're accessible or even take the right number of parameters). Only if it doesn't find any at all does it then continue "outward" into the next enclosing scope and repeat—in this case, the scope of the base class B—until it eventually either runs out of scopes without having found a function with the right name or else finds a scope that contains at least one candidate function. If a scope is found that has one or more candidate functions, the compiler then stops searching and works with the candidates that it's found, performing overload resolution and then applying access rules.

Memory Management

Item 35. Memory Management—Part 1

Difficulty: 3

How well do you know memory? What different memory areas are there?

This problem covers basics about C++'s main distinct memory stores. The following problem goes on to attack some deeper memory management questions in more detail.

C++ has several distinct memory areas in which objects and nonobject values may be stored, and each area has different characteristics.

Name as many of the distinct memory areas as you can. For each, summarize its performance characteristics and describe the lifetime of objects stored there.

Example: The stack stores automatic variables, including both builtins and objects of class type.

Solution

The following table summarizes a C++ program's major distinct memory areas. Note that some of the names (for example, "heap") do not appear as such in the standard; in particular, "heap" and "free store" are common and convenient shorthands for distinguishing between two kinds of dynamically allocated memory.

Table 1. C++'s Memory Areas

Memory Area	**Characteristics and Object Lifetimes**
Const Data	The const data area stores string literals and other data whose values are known at compile-time. No objects of class type can exist in this area. All data in this area is available during the entire lifetime of the program. Further, all this data is read-only, and the results of trying to modify it are undefined. This is in part because even the underlying storage format is subject to arbitrary optimization by the implementation. For example, a particular compiler may choose to store string literals in overlapping objects as an optional optimization.
Stack	The stack stores automatic variables. Objects are constructed immediately at the point of definition and destroyed immediately at the end of the same scope, so there is no opportunity for programmers to directly manipulate allocated but uninitialized stack space (barring willful tampering using explicit destructors and placement new). Stack memory allocation is typically much faster than for dynamic storage (heap or free store) because each stack memory allocation involves only a stack pointer increment rather than more-complex management.
Free Store	The free store is one of the two dynamic memory areas allocated/freed by new/delete. Object lifetime can be less than the time the storage is allocated. That is, free store objects can have

Item 36. Memory Management—Part 2

Difficulty: 6

Are you thinking about doing your own class-specific memory management, or even replacing C++'s global new and delete? First, try this problem on for size.

The following code shows classes that perform their own memory management. Point out as many memory-related errors as possible, and answer the additional questions.

1.

1. Consider the following code:

1.
```
class B
{
public:
  virtual ~B();
  void operator delete  ( void*, size_t ) throw();
  void operator delete[]( void*, size_t ) throw();
  void f( void*, size_t ) throw();
};
class D : public B
{
public:
  void operator delete  ( void* ) throw();
  void operator delete[]( void* ) throw();
};
```

1. Why do B's operators delete have a second parameter, whereas D's do not? Do you see any way to improve the function declarations?
1.
2.

2. Continuing with the same piece of code: Which operator delete() is called for each of the following delete expressions? Why, and with what parameters?

2.
```
D* pd1 = new D;
delete pd1;
B* pb1 = new D;
delete pb1;
D* pd2 = new D[10];
delete[] pd2;
B* pb2 = new D[10];
delete[] pb2;
```

3.

3. Are the following two assignments legal?

Solution

Let's take the questions one at a time.

1.

1. Consider the following code:

1.
```
class B
{
public:
  virtual ~B();
  void operator delete  ( void*, size_t ) throw();
  void operator delete[]( void*, size_t ) throw();
  void f( void*, size_t ) throw();
};

class D : public B
{
public:
  void operator delete  ( void* ) throw();
  void operator delete[]( void* ) throw();
};
```

1. Why do B's operators delete have a second parameter, whereas D's do not?
1.

1. The answer is: It's just preference, that's all. Both are usual deallocation functions, not placement deletes. (For those keeping score at home, see section 3.7.3.2/2 in the C++ standard.)
1.

1. However, there's also a memory error lurking in this underbrush. Both classes provide an operator delete() and operator delete[](), without providing the corresponding operator new() and operator new[](). This is extremely dangerous, because the default operator new() and operator new[]() are unlikely to do the right thing. (For example, consider what happens if a further-derived class provides its own operator new() or operator new[]() functions.)

1. Guideline
1.

Always provide both class-specific new (or new[]) and class-specific delete (or delete[]) if you provide either.

1. And the second part of the question: Do you see any way to improve the function declarations?
1.

1. All flavors of operator new() and operator delete() are always static functions, even if they're not

Item 37. AUTO_PTR

Difficulty: 8

This Item covers basics about how you can use the standard auto_ptr safely and effectively.

Historical note: The original simpler form of this Item, appearing as a Special Edition of Guru of the Week, was first published in honor of the voting out of the Final Draft International Standard for Programming Language C++. It was known/suspected that auto_ptr would change one last time at the final meeting, where the standard was to be voted complete (Morristown, New Jersey, November 1997), so this problem was posted the day before the meeting began. The solution, freshly updated to reflect the prior day's changes to the standard, became the first published treatment of the standard auto_ptr.

Many thanks from all of us to Bill Gibbons, Greg Colvin, Steve Rumsby, and others who worked hard on the final refinement of auto_ptr. Greg, in particular, has labored over auto_ptr and related smart pointer classes for many years to satisfy various committee concerns and requirements, and deserves public recognition for that work.

This problem, now with a considerably more comprehensive and refined solution, illustrates the reasons for the eleventh-hour changes that were made, and it shows how you can make the best possible use of auto_ptr.

Comment on the following code: What's good, what's safe, what's legal, and what's not?

```
auto_ptr<T> source()
{
  return auto_ptr<T>( new T(1) );
}
void sink( auto_ptr<T> pt ) { }
void f()
{
  auto_ptr<T> a( source() );
  sink( source() );
  sink( auto_ptr<T>( new T(1) ) );
  vector< auto_ptr<T> > v;
  v.push_back( auto_ptr<T>( new T(3) ) );
  v.push_back( auto_ptr<T>( new T(4) ) );
  v.push_back( auto_ptr<T>( new T(1) ) );
  v.push_back( a );
  v.push_back( auto_ptr<T>( new T(2) ) );
  sort( v.begin(), v.end() );
  cout << a->Value();
}
class C
{
public:    /*...*/
protected: /*...*/
```

Solution

Most people have heard of the standard auto_ptr smart pointer facility, but not everyone uses it daily. That's a shame, because it turns out that auto_ptr neatly solves common C++ coding problems, and using it well can lead to more-robust code. This article shows how to use auto_ptr correctly to make your code safer—and how to avoid the dangerous but common abuses of auto_ptr that create intermittent and hard-to-diagnose bugs.

Why "Auto" Pointer?

auto_ptr is just one of a wide array of possible smart pointers. Many commercial libraries provide more-sophisticated kinds of smart pointers that can do wild and wonderful things, from managing reference counts to providing advanced proxy services. Think of the standard C++ auto_ptr as the Ford Escort of smart pointers—a simple, general-purpose smart pointer that doesn't have all the gizmos and luxuries of special-purpose or high-performance smart pointers, but does many common things well and is perfectly suitable for regular use.

What auto_ptr does is own a dynamically allocated object and perform automatic cleanup when the object is no longer needed. Here's a simple example of code that's unsafe without auto_ptr.

```
// Example 1(a): Original code
//
void f()
{
  T* pt( new T );
  /*...more code...*/
  delete pt;
}
```

Most of us write code like this every day. If f() is a three-line function that doesn't do anything exceptional, this may be fine. But if f() never executes the delete statement, either because of an early return or because of an exception thrown during execution of the function body, then the allocated object is not deleted and we have a classic memory leak.

A simple way to make Example 1(a) safe is to wrap the pointer in a "smarter," pointer-like object that owns the pointer and that, when destroyed, deletes the pointed-at object automatically. Because this smart pointer is simply used as an automatic object (that is, one that's destroyed automatically when it goes out of scope), it's reasonably called an "auto" pointer:

```
// Example 1(b): Safe code, with auto_ptr
//
void f()
{
  auto_ptr<T> pt( new T );
  /*...more code...*/
} // cool: pt's destructor is called as it goes out
```

I l@ve RuBoard PREVIOUS NEXT

I l@ve RuBoard PREVIOUS NEXT

Traps, Pitfalls, and Anti-Idioms

I l@ve RuBoard PREVIOUS NEXT

I l@ve RuBoard PREVIOUS NEXT

Item 38. Object Identity

Difficulty: 5

"Who am I, really?" This problem addresses how to decide whether two pointers really refer to the same object.

The "this != &other" test (illustrated below) is a common coding practice intended to prevent self-assignment. Is the condition necessary and/or sufficient to accomplish this? Why or why not? If not, how would you fix it?

```
T& T::operator=( const T& other )
{
  if( this != &other )    // the test in question
  {
    // ...
  }
  return *this;
}
```

Remember to distinguish between "protecting against Murphy versus protecting against Machiavelli"—that is, protecting against things going wrong on their own versus a malicious programmer going to great lengths trying to break your code.

Solution

Short answer: In most cases, it's bad only if, without the test, self-assignment is not handled correctly. Although this check won't protect against all possible abuses, in practice it's fine as long as it's being done for optimization only. In the past, some have suggested that multiple inheritance affects the correctness of this problem. This is not true, and it's a red herring.

Exception Safety (Murphy)

If T::operator=() is written using the create-a-temporary-and-swap idiom (see page 47), it will be both strongly exception-safe and not have to test for self-assignment. Period. Because we should normally prefer to write copy assignment this way, we shouldn't need to perform the self-assignment test, right?

Guideline

Never write a copy assignment operator that relies on a check for self-assignment in order to work properly; a copy assignment operator that uses the create-a-temporary-and-swap idiom is automaticially both strongly exception-safe and safe for self-assignment,

Yes and no. It turns out that there are two potential efficiency costs when not checking for self-assignment.

-
- If you can test for self-assignment, then you can completely optimize away the assignment
-
-
- Often making code exception-safe also makes it less efficient (a.k.a. the "paranoia has a price principle").
-

In practice, if self-assignment happens often (rare, in most programs) and the speed improvement gained by suppressing unnecessary work during self-assignment is significant to your application (even rarer, in most programs), check for self-assignment.

Guideline

Item 39. Automatic Conversions

Difficulty: 4

Automatic conversions from one type to another can be extremely convenient. This Item covers a typical example to illustrate why they're also extremely dangerous.

The standard C++ string has no implicit conversion to a const char*. Should it?

It is often useful to be able to access a string as a C-style const char*. Indeed, string has a member function c_str() to allow just that, by giving access to a const char*. Here's the difference in client code:

```
string s1("hello"), s2("world");
strcmp( s1, s2 );                // 1 (error)
strcmp( s1.c_str(), s2.c_str() ) // 2 (ok)
```

It would certainly be nice to do #1, but #1 is an error because strcmp requires two pointers and there's no automatic conversion from string to const char*. Number 2 is okay, but it's longer to write because we have to call c_str() explicitly.

So this Item's question really boils down to: Wouldn't it be better if we could just write #1?

Solution

The question was: The standard C++ string has no implicit conversion to a const char*. Should it?

The answer is: No, with good reason.

It's almost always a good idea to avoid writing automatic conversions, either as conversion operators or as single-argument non-explicit constructors.[2] The main reasons that implicit conversions are unsafe in general are:

[2] This solution focuses on the usual problems of implicit conversions, but there are other reasons why a string class should not have a conversion to const char*. Here are a few citations to further discussions: Koenig A. and Moo B. Ruminations on C++ (Addison Wesley Longman, 1997), pages 290–292. Stroustrup, Bjarne. The Design and Evolution of C++ (Addison-Wesley, 1994), page 83.

-
- Implicit conversions can interfere with overload resolution.
-
-
- Implicit conversions can silently let "wrong" code compile cleanly.
-

If a string had an automatic conversion to const char*, that conversion could be implicitly called anywhere the compiler felt it was necessary. What this means is that you would get all sorts of subtle conversion problems—the same ones you get into when you have non-explicit conversion constructors. It becomes far too easy to write code that looks right, is in fact not right and should fail, but by sheer coincidence will compile by doing something completely different from what was intended.

There are many good examples. Here's a simple one:

```
string s1, s2, s3;
s1 = s2 - s3;   // oops, probably meant "+"
```

The subtraction is meaningless and should be wrong. If string had an implicit conversion to const char*, however, this code would compile cleanly because the compiler would silently convert both strings to const char*'s and then subtract those pointers.

Guideline

Item 40. Object Lifetimes—Part 1

Difficulty: 5

"To be, or not to be " When does an object actually exist? This problem considers when an object is safe to use.

Critique the following code fragment.

```
void f()
{
  T t(1);
  T& rt = t;
  //--- #1: do something with t or rt ---
  t.~T();
  new (&t) T(2);
  //--- #2: do something with t or rt ---
}// t is destroyed again
```

Is the code in block #2 safe and/or legal? Explain.

Solution

Yes, #2 is safe and legal (if you get to it), but:

-
- The function as a whole is not safe.
-
-
- It's a bad habit to get into.
-

The C++ standard explicitly allows this code. The reference rt is not invalidated by the in-place destruction and reconstruction. (Of course, you can't use t or rt between the call to t.~T() and the placement new, because during that time no object exists. We're also forced to assume that T::operator&() hasn't been overloaded to do something other than return the object's address.)

The reason we say #2 is safe "if you get to it" is that f() as a whole may not be exception-safe.

The function is not safe. If T's constructor may throw in the new (&t) T(2) call, then f() is not exception-safe. Consider why: If the T(2) call throws, then no new object has been reconstructed in the t memory area, yet at the end of the function, T::~T() is naturally called (since t is an automatic variable) and "t is destroyed again," just as the comment says. That is, t will be constructed once but destroyed twice (oops). This is likely to create unpredictable side effects, such as core dumps.

Guideline

Always endeavor to write exception-safe code. Always structure code so that resources are correctly freed and data is in a consistent state even in the presence of exceptions.

This is a bad habit. Ignoring the exception-safety issues, the code happens to work in this setting because the programmer knows the complete type of the object being constructed and destroyed. That is, the object was a T and is being destroyed and reconstructed as a T.

This technique is rarely, if ever, necessary in real code and is a very bad habit to get into, because it's fraught with (sometimes subtle) dangers if it appears in a member function.

```
// Can you spot the subtle problem?
```

Item 41. Object Lifetimes—Part 2

Difficulty: 6

Following from Item 40, this issue considers a C++ idiom that's frequently recommended—but often dangerously wrong.

Critique the following "anti-idiom" (shown as commonly presented).

```
T& T::operator=( const T& other )
{
  if( this != &other )
  {
    this->~T();
    new (this) T(other);
  }
  return *this;
}
```

1.

1. What legitimate goal does it try to achieve? Correct any coding flaws in this version.
1.
2.

2. Even with any flaws corrected, is this idiom safe? Explain. If not, how else should the programmer achieve the intended results?
2.

(See also Item 40.)

Solution

This idiom[3] is frequently recommended, and it appears as an example in the C++ standard (see the discussion in the accompanying box). It is also exceedingly poor form and causes no end of problems. Don't do it.

[3] I'm ignoring pathological cases (for example, overloading T::operator&() to do something other than return this). See also Item 38.

Fortunately, as we'll see, there is a right way to get the intended effect.

A Warning Example

In the C++ standard, this example was intended to demonstrate the object lifetime rules, not to recommend a good practice (it isn't). For those interested, here it is, slightly edited for space, from clause 3.8, paragraph 7:

```
 [Example:
  struct C {
    int i;
    void f();
    const C& operator=( const C& );
  };
  const C& C::operator=( const C& other)
  {
    if ( this != &other )
    {
      this->~C();     // lifetime of *this ends
      new (this) C(other);
                      // new object of type C created
      f();            // well-defined
    }
    return *this;
  }
  C c1;
  C c2;
  c1 = c2; // well-defined
  c1.f();  // well-defined; c1 refers to a new object of type C
--end example]
```

As further proof that this is not intended to recommend good practice, note that here C::operator=() returns a const C& rather than a plain C&, which needlessly prevents portable use of these objects in standard library containers.

Miscellaneous Topics

Item 42. Variable Initialization—Or is it?

Difficulty: 3

This first problem highlights the importance of understanding what you write. Here we have four simple lines of code, no two of which mean the same thing, even though the syntax varies only slightly.

What is the difference, if any, between the following? (T stands for any class type.)

```
T t;
T t();
T t(u);
T t = u;
```

I l@ve RuBoard PREVIOUS NEXT

I l@ve RuBoard PREVIOUS NEXT

Solution

This puzzle demonstrates the difference between three kinds of initialization: default initialization, direct initialization, and copy initialization. It also contains one red herring that isn't initialization at all. Let's consider the cases one by one.

```
T t;
```

This is default initialization. This code declares a variable named t, of type T, which is initialized using the default constructor T::T().

```
T t();
```

A red herring. At first glance, it may look like just another variable declaration. In reality, it's a function declaration for a function named t that takes no parameters and returns a T object by value. (If you can't see this at first, consider that the above code is no different from writing something like int f(); which is clearly a function declaration.)

Some people suggest writing "auto T t();" in an attempt to use the auto storage class to show that, yes, they really want a default-constructed variable named t of type T. Allow me license for a small rant here. For one thing, that won't work on a standard-conforming compiler; the compiler will still parse it as a function declaration, and then reject it because you can't specify an auto storage class for a return value. For another thing, even if it did work, it would be wrong-headed, because there's already a simpler way to do what's wanted. If you want a default-constructed variable t of type T, then just write "T t;" and quit trying to confuse the poor maintenance programmers with unnecessary subtlety. Always prefer simple solutions to cute solutions. Never write code that's any more subtle than necessary.

```
T t(u);
```

Assuming u is not the name of a type, this is direct initialization. The variable t is initialized directly from the value of u by calling T::T(u). (If u is a type name, this is a declaration even if there is also a variable named u in scope; see above.)

```
T t = u;
```

This is copy initialization. The variable t is always initialized using T's copy constructor, possibly after calling another function.

Common Mistake

This is always initialization; it is never assignment, and so it never calls T::operator=().

Item 43. Const-Correctness

Difficulty: 6

const is a powerful tool for writing safer code. Use const as much as possible, but no more. Here are some obvious and not-so-obvious places where const should—or shouldn't—be used.

Don't comment on or change the structure of this program; it's contrived and condensed for illustration only. Just add or remove const (including minor variants and related keywords) wherever appropriate.

Bonus question: In what places are the program's results undefined/uncompilable due to const errors?

```
class Polygon
{
public:
  Polygon() : area_(-1) {}
  void  AddPoint( const Point pt ) { InvalidateArea();
                                     points_.push_back(pt); }
  Point GetPoint( const int i )    { return points_[i]; }
  int   GetNumPoints()             { return points_.size(); }
  double GetArea()
  {
    if( area_ < 0 ) // if not yet calculated and cached
    {
      CalcArea();     // calculate now
    }
    return area_;
  }
private:
  void InvalidateArea() { area_ = -1; }
  void CalcArea()
  {
    area_ = 0;
    vector<Point>::iterator i;
    for( i = points_.begin(); i != points_.end(); ++i )
      area_ += /* some work */;
  }
  vector<Point> points_;
  double        area_;
};
Polygon operator+( Polygon& lhs, Polygon& rhs )
{
  Polygon ret = lhs;
  int last = rhs.GetNumPoints();
  for( int i = 0; i < last; ++i ) // concatenate
  {
    ret.AddPoint( rhs.GetPoint(i) );
  }
  return ret;
}
```

Solution

When I pose this kind of problem, I find that most people think the problem is on the easy side and address only the more-usual const issues. There are, however, subtleties that are worth knowing, hence this Item. See also the box "const and mutable Are Your Friends."

```
class Polygon
{
public:
  Polygon() : area_(-1) {}
  void  AddPoint( const Point pt ) { InvalidateArea();
                                  points_.push_back(pt); }
```

1.

1. The Point object is passed by value, so there is little or no benefit to declaring it const. In fact, to the compiler, the function signature is the same whether you include this const in front of a value parameter or not. For example:

1.
```
   int f( int );
int f( const int );  // redeclares f(int)
                     // no overloading, there's only one function
int g( int& );
int g( const int& ); // not the same as g(int&)
                     // g is overloaded
```

1. Guideline
1.

Avoid const pass-by-value parameters in function declarations. Still make the parameter const in the same function's definition if it won't be modified.

1.
```
Point GetPoint( const int i )   { return points_[i]; }
```

2.

2. Same comment about the parameter type. Normally, const pass-by-value is unuseful and misleading at best.
2.
3.

3. This should be a const member function, since it doesn't change the state of the object.
3.
4.

4. If you're not returning a builtin type, such as int or long, return-by-value should usually return a const

Item 44. Casts

Difficulty: 6

How well do you know C++'s casts? Using them well can greatly improve the reliability of your code.

The new-style casts in standard C++ offer more power and safety than the old-style (C-style) casts. How well do you know them? The rest of this problem uses the following classes and global variables:

```
class  A { public: virtual ~A(); /*...*/ };
A::~A() { }

class B : private virtual A  { /*...*/ };

class C : public  A          { /*...*/ };

class D : public B, public C { /*...*/ };

A a1; B b1; C c1; D d1;
const A  a2;
const A& ra1 = a1;
const A& ra2 = a2;
char c;
```

This Item presents four questions.

1.

1. Which of the following new-style casts are not equivalent to a C-style cast?

1. `const_cast`
```
dynamic_cast
reinterpret_cast
static_cast
```

2.

2. For each of the following C-style casts, write the equivalent new-style cast. Which are incorrect if not written as a new-style cast?

2. `void f()`
```
{
  A* pa; B* pb; C* pc;
  pa = (A*)&ra1;
  pa = (A*)&a2;
  pb = (B*)&c1;
  pc = (C*)&d1;
}
```

Solution

Let's answer the questions one by one.

1.

1. Which of the following new-style casts are not equivalent to a C-style cast?

1. Only dynamic_cast is not equivalent to a C-style cast. All other new-style casts have old-style equivalents.

1. Guideline
1.

Prefer new-style casts.

2.

2. For each of the following C-style casts, write the equivalent new-style cast. Which are incorrect if not written as a new-style cast?

2. `void f()`

```
{
  A* pa; B* pb; C* pc;
  pa = (A*)&ra1;
```

2. Use const_cast instead:
2.

```
pa = const_cast<A*>(&ra1);

pa = (A*)&a2;
```

2. This cannot be expressed as a new-style cast. The closest candidate is const_cast, but because a2 is a const object, the results of using the pointer are undefined.
2.

```
pb = (B*)&c1;
```

2. Use reinterpret_cast instead:
2.

```
  pb = reinterpret_cast<B*>(&c1);

  pc = (C*)&d1;
}
```

Item 45. BOOL

Difficulty: 7

Do we really need a builtin bool type? Why not just emulate it in the existing language? This Item reveals the answer.

Besides wchar_t (which was a typedef in C), bool is the only builtin type to be added to C++ since the ARM (Ellis90).

Could bool's effect have been duplicated without adding a builtin type? If yes, show an equivalent implementation. If no, show why possible implementations do not behave the same as the builtin bool.

Solution

The answer is: No, bool's effect could not have been duplicated without adding a builtin type. The bool builtin type, and the reserved keywords true and false, were added to C++ precisely because they couldn't be duplicated completely using the existing language.

This Item is intended to illustrate the considerations you have to think about when you design your own classes, enums, and other tools.

The second part of the Item question was: If no, show why possible implementations do not behave the same as the builtin bool.

There are four major implementations.

Option 1: typedef (score: 8.5 / 10)

This option means to "typedef <something> bool;", typically:

```
// Option 1: typedef
//
typedef int bool;
const bool true  = 1;
const bool false = 0;
```

This solution isn't bad, but it doesn't allow overloading on bool. For example:

```
// file f.h
void f( int  ); // ok
void f( bool ); // ok, redeclares the same function
// file f.cpp
void f( int  ) { /*...*/ }  // ok
void f( bool ) { /*...*/ }  // error, redefinition
```

Another problem is that Option 1 can allow code like this to behave unexpectedly:

```
void f( bool b )
{
  assert( b != true && b != false );
  }
```

So Option 1 isn't good enough.

Option 2: #define (score: 0 / 10)

Item 46. Forwarding Functions

Difficulty: 3

What's the best way to write a forwarding function? The basic answer is easy, but we'll also learn about a subtle change to the language made shortly before the standard was finalized.

Forwarding functions are useful tools for handing off work to another function or object, especially when the hand-off is done efficiently.

Critique the following forwarding function. Would you change it? If so, how?

```
// file f.cpp
//
#include "f.h"
/*...*/
bool f( X x )
{
  return g( x );

}
```

Solution

Remember the introduction to the question? It was: Forwarding functions are useful tools for handing off work to another function or object, especially when the hand-off is done efficiently.

This introduction gets to the heart of the matter—efficiency.

There are two main enhancements that would make this function more efficient. The first should always be done; the second is a matter of judgment.

1.

1. Pass the parameter by const& instead of by value.

1. "Isn't that blindingly obvious?" you might ask. No, it isn't, not in this particular case. Until as recently as 1997, the draft C++ language standard said that because a compiler can prove that the parameter x will never be used for any other purpose than passing it in turn to g(), the compiler may decide to elide x completely (that is, to eliminate x as unnecessary). For example, if the client code looks something like this:
1.

```
X my_x;
f( my_x );
```

1. then the compiler used to be allowed to either:
1.
   -
   - Create a copy of my_x for f()'s use (this is the parameter named x in f()'s scope) and pass that to g()
   -
   -
   - Pass my_x directly to g() without creating a copy at all, because it notices that the extra copy will never be used except as a parameter to g()
   - The latter is nicely efficient, isn't it? That's what optimizing compilers are for, aren't they?
   -
   - Yes and yes, until the London meeting in July 1997. At that meeting, the draft was amended to place more restrictions on the situations in which the compiler is allowed to elide "extra" copies like this. This change was necessary to avoid problems that can come up when compilers are permitted to wantonly elide copy construction, especially when copy construction has side effects. There are reasonable cases in which reasonable code may legitimately rely on the number of copies actually made of an object.
   -

Item 47. Control Flow

Difficulty: 6

How well do you really know the order in which C++ code is executed? Test your knowledge against this problem.

"The devil is in the details." Point out as many problems as possible in the following (somewhat contrived) code, focusing on those related to control flow.

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;

//  The following lines come from other header files.
//
char* itoa( int value, char* workArea, int radix );
extern int fileIdCounter;

//  Helpers to automate class invariant checking.
//
template<class T>
inline void AAssert( T& p )
{
  static int localFileId = ++fileIdCounter;
  if( !p.Invariant() )
  {
    cerr << "Invariant failed: file " << localFileId
         << ", " << typeid(p).name()
         << " at " << static_cast<void*>(&p) << endl;
    assert( false );
  }
}

template<class T>
class AInvariant
{
public:
  AInvariant( T& p ) : p_(p) { AAssert( p_ ); }
  ~AInvariant()              { AAssert( p_ ); }
private:
  T& p_;
};
#define AINVARIANT_GUARD AInvariant<AIType> invariantChecker( *this )

//-------------------------------------------------------------
template<class T>
class Array : private ArrayBase, public Container
{
  typedef Array AIType;
```

Solution

"Lions and tigers and bears, oh my!"

——Dorothy

Compared with what's wrong with this Item's code, Dorothy had nothing to complain about. Let's consider it line by line.

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;

//  The following lines come from other header files.
//
char* itoa( int value, char* workArea, int radix );
extern int fileIdCounter;
```

The presence of a global variable should already put us on the lookout for clients that might try to use it before it has been initialized. The order of initialization for global variables (including class statics) between translation units is undefined.

Guideline

Avoid using global or static objects. If you must use a global or static object, always be very careful about the order-of-initialization rules.

```
//  Helpers to automate class invariant checking.
//
template<class T>
inline void AAssert( T& p )
{
  static int localFileId = ++fileIdCounter;
```

Aha! And here we have a case in point. Say the definition of fileIdCounter is something like the following:

```
int fileIdCounter = InitFileId();  // starts count at 100
```

If the compiler happens to initialize fileIdCounter before it initializes any AAssert<T>::localFileId, well and

Afterword

If you've enjoyed the puzzles and problems in this book, then I have good news for you. This is not the end, for Guru of the Week #30 was not the last GotW, nor have I stopped writing articles in various programming magazines.

Today, on the Internet, new GotW issues are being published and discussed and debated regularly on the newsgroup comp.lang.c++.moderated, and are archived at the official GotW Website at www.gotw.ca. As I write this, in June 1999, we're already up to #55. To give you a taste for what's coming, a small sampling of the new GotW issues includes fresh material on such topics as the following:

-
- More information on popular themes, including the safe use of auto_ptr, namespaces, and exception-safety issues and techniques, taking the next step beyond Items 8 through 17, 31 through 34, and 37.
-
-
- A three-part series on reference-counting and copy-on-write techniques, including unusual performance implications in multithreaded (or multithread-capable) environments, with extensive test harness code and statistical measurements. There's material here that you usually don't see discussed anywhere else.
-
-
- Many puzzles about the safe and effective use of standard library, especially containers (like vector and map) and the standard streams. This includes more information about how to best extend the standard library, in the spirit of Items 2 and 3.
-
-
- A nifty game: writing a MasterMind-playing program in as few statements as possible.
-

And that's just a small sample. If there is enough interest in the book you're holding in your hands now, my intention is to produce another volume containing expanded and reorganized forms of the next batch of issues, again including the text of the other C++ engineering articles and columns that I'm writing for C/C++ Users Journal, and other magazines.

I hope you've enjoyed the book, and that you'll continue to let me know what interesting topics you'd like to see covered in the future; see the Website mentioned earlier for how to submit requests. Some topics you've read about herein were prompted by e-mails like these.

Thanks again to all who have expressed interest and support for GotW and this book. I hope you've found

Bibliography

Barton94: John Barton and Lee Nackman, Scientific and Engineering C++. Addison-Wesley, 1994.

Cargill92: Tom Cargill, C++ Programming Style. Addison-Wesley, 1992.

Cargill94: Tom Cargill, "*Exception Handling: A False Sense of Security*." C++ Report, 9(6), Nov.–Dec. 1994.

Cline95: Marshall Cline and Greg Lomow, C++ FAQs. Addison-Wesley, 1995.

Cline99: Marshall Cline, Greg Lomow, and Mike Girou, C++ FAQs, Second Edition. Addison Wesley Longman, 1999.

Coplien92: James Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1992.

Ellis90: Margaret Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual. Addison-Wesley, 1990.

Gamma95: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Keffer95: Thomas Keffer, Rogue Wave C++ Design, Implementation, and Style Guide. Rogue Wave Software, 1995.

Koenig97: Andrew Koenig and Barbara Moo, Ruminations on C++. Addison Wesley Longman, 1997.

Lakos96: John Lakos, Large-Scale C++ Software Design. Addison-Wesley, 1996.

Lippman97: Stan Lippman (ed.), C++ Gems. SIGS / Cambridge University Press, 1997.

Lippman98: Stan Lippman and Josée Lajoie, C++ Primer, Third Edition. Addison Wesley Longman, 1998.

Martin95: Robert C. Martin, Designing Object-Oriented Applications Using the Booch Method. Prentice-Hall, 1995.