C++ Design

Tim Bailey

Version 2.0
March 24, 2006

1 Introduction

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.* C. A. R. Hoare, The 1980 ACM Turing Award Lecture.

C++ is much more than an object-oriented programming language [Str95]. It supports procedural programming, definition of modular *abstract data types*, object-oriented programming, and generic programming. Each of these concepts has its relative advantages, and C++ does not constrain the programmer to any one style but permits their use in combination—a concept known as "multi-paradigm design".

Ideally a high-level programming language facilitates writing code that is readable, portable and, above all, scales well with the size of the program. For large programs, it is important that complexity does not increase more than linearly with size. Modularity and encapsulation are the key to managing complexity through grouping related operations and data, and decoupling modules from each other. A large program composed of encapsulated code-modules acts not as an elaborate monolith, but as a collection of small programs interacting through well-defined interfaces.

Mention the word "encapsulation" and many programmers think of "data hiding". However, this has misleading connotations, and fosters convoluted designs that serve only to baffle the programmer who later tries to understand the low-level implementation. Encapsulation is not about hiding the details of an algorithm or obscuring program flow. Rather it is about making code self-contained, and easy to use and maintain through clean *separation* of the public interface from implementation details. "Data isolation" might be a better term; an encapsulated module is accessible only through a well-defined public interface, which isolates users from incidental complexity and prevents needless dependencies with other modules.

C++ is a powerful and expressive language that provides considerable design freedom. This freedom is double-edged and certainly it is very easy to write dangerous and contorted code. However, with a little discipline and an understanding of basic C++ design idioms, it is possible to write software that is correct, elegant, flexible and remarkably efficient. This paper describes methods to promote encapsulation in C++; to design software that is intuitive and suited to reuse.

2 Writing Intuitive Code

A well written code-module is easy to use correctly and difficult to use wrong. The C++ language provides various ways to prevent incorrect access to classes and functions, and this section presents a number of design techniques and idioms for implementing intuitive interfaces.

2.1 Make Class Interfaces Minimal and Complete

> *Avoiding the gargantuanism that plagues large projects is essential. Far too often people add features "just in case" and end up doubling and quadrupling the size and runtime of systems to support frills.* Bjarne Stroustrup [Str00a, Section 23.4.7]

C++ design practices have progressed since the 1980s. C++ is no longer viewed as simply an object-oriented language and not everything is best represented as an class hierarchy [Str00a, Section 24.2.5]. Inheritance is seen to be widely overused and hierarchies too deep [Sut98, Sut99]. Often a mix of procedural functions, templates and concrete classes can provide the best encapsulation.

A very common flaw in class design is "fat" interfaces. Classes with a lot of member functions supporting too-many non-essential features tend to be confusing to use and difficult to maintain. A better approach is to design a class interface so that it is minimal and complete [Mey98, Item 18], [Str00a, Sections 10.3.2 and 23.4.3.2]. Auxiliary functionality can then be written as non-member functions that access the class via its public interface. The following algorithm, presented in [Mey00], is a useful rule-of-thumb for class interface design.

```
if (f needs to be virtual)
    make f a member function of C;
else if (f is operator>> or operator<<)
{
    make f a non-member function;
    if (f needs access to non-public members of C)
        make f a friend of C;
}
else if (f needs type conversions on its left-most argument)
{
    make f a non-member function;
    if (f needs access to non-public members of C)
        make f a friend of C;
}
else if (f can be implemented via C's public interface)
    make f a non-member function;
else
    make f a member function of C;
```

For example, arithmetic operators, such as `operator+()`, are typically best implemented as non-member functions.[1]

```
Complex operator+(const Complex &c1, const Complex &c2)
{
    Complex result(c1);
    result.real += c2.real;
    result.imag += c2.imag;
    return result;
}
```

As a non-member function, the addition operator provides a symmetric interface and permits implicit type conversion of its arguments.

[1] The data members of class `Complex` are most likely private, in which case the addition operator would need to be a `friend` of `Complex`.

2.2 Expose the Public Interface Only

Software design is primarily the art of *organisation*. Grouping related data and operations, exposing those interfaces required for external access, and making all other parts inaccessible is the essence of encapsulation.

The C++ language provides several constructs for controlling access and visibility to facilitate hierarchical organisation. These include file-scope visibility (inherited from C), class access specifiers, and namespaces. Thus, code organisation may be addressed at three levels: classes, modules and libraries.

- Class level. C++ provides three class access specifiers. A class interface should differentiate between those member-functions required for client use (`public`), those used solely for class implementation details (`private`), and those that must be accessible only to derived classes (`protected`). Data members should almost always be `private`.

- Module level. A module is usually defined by a pair of files: a source file (.cpp, .cxx, etc) and a header file (.h, .hpp. etc). Source files should contain all implementation details and private classes, functions, constants, etc.[2] These parts are termed the *private interface* and are not accessible to users. Header files should contain the *public interface* only.[3] The public interface is the set of declarations that must be available to users. When designing a header-and-source pair, it is important to put only the public interface in the header and relegate all internal-use classes etc to the source file. Private declarations in a source file should be wrapped in an unnamed namespace[4] [Str00a, Section 8.2.5.1], which restricts their visibility to that file.

  ```
  namespace {
      // Private function/class declarations, constants, etc.
  }
  ```

  The exception to this rule is *template* classes (and template functions) composing implementation details of components in the public interface. These must appear in the header file. To prevent name clashes and accidental use, a common convention is to place these template definitions in a namespace called `detail`.

  ```
  namespace detail {
      // Private template classes and functions.
  }
  ```

- Library level. A library should be enclosed in its own namespace. This prevents cluttering of the global namespace and greatly reduces the risk of name clashes. Separate namespaces at the module level is too fine grained; all modules in a library should share a common namespace.

2.3 Encapsulate Order of Operations

An intuitive class public interface should present a high-level abstraction that is difficult to misuse. One aspect of simple usage is to wrap-up a sequence of operations in a single function to preserve correct order. Consider the following member function from a `Mutex` class, which wraps a series of *pthread* operations for locking a thread mutex.

[2] We assume functions and top-level data declared within source files are given file-scope visibility (via anonymous namespace) and so are inaccessible from other modules. It is possible to declare them with *external* scope, but global declarations are bad practice in general as they thwart modularity.

[3] The terms *private interface* and *public interface* should not be confused with the C++ `private` and `public` access specifiers. Here we are referring to a convention for module-level visibility, not language keywords for class-level accessibility.

[4] Unnamed namespaces replace the C-style `static` declaration for file-scope visibility, which is deprecated.

```
void Mutex::lock()
{
    // Check for recursive locking (ie, deadlock)
    pthread_t self = pthread_self();
    if(pthread_equal(owner, self) != 0)
        throw LockError("Mutex lock deadlocked");

    // Acquire lock
    int status = pthread_mutex_lock(&mtx);
    if (status == EDEADLK)
        throw LockError("Mutex lock deadlocked");
    assert(status == 0);
    assert(islocked == false);
    islocked = true;
    owner = self;
}
```

This function encapsulates error checking (for deadlocks) and sanity-check assertions for implementation bugs, resulting in a robustness not present in a simple call to `pthread_mutex_lock()`. The member-function ensures all operations and checks are performed in the right order and that the class instance remains in a valid state.

The need to enforce correct order-of-operations can also arise in a class hierarchy. Suppose you have a base-class `Widget` that has an member-function `process()`.[5]

```
class Widget
{
public:
    virtual ~Widget();
    virtual void process(Gadget&);
    // ...
};
```

The `process()` function executes a series of operations, some of which should be customised by derived classes.

```
void Widget::process(Gadget&)
{
    // 1. Perform an operation customisable by a derived class.
    // 2. Perform a series of non-customisable operations.
    // 3. Perform an operation customisable by a derived class.
}
```

This gives rise to a problem as to how to permit customisation while still enforcing correct operation sequence. The derived class could override the entire function, but this requires verbatim cut-and-paste implementation of the non-customisable code for each derived class. Alternatively `process()` could be split into three separate functions, two of which are customised, but this then places the onus on users to call them in correct order.

The problem is resolved using the "template method" design pattern [GHJV94, HS00, Sut01].[6] The `process()` function is declared public-non-virtual, and `Widget` declares two additional pure-virtual functions, one for each customisable operation.

[5] This example is adapted from [Sut01].

[6] The template method pattern should not be confused with the C++ `template<>` feature. They are unrelated.

```
class Widget
{
public:
    virtual ~Widget();
    void process(Gadget&);
    // ...
private:
    virtual void doProcessPhase1(Gadget&) = 0;
    virtual void doProcessPhase2(Gadget&) = 0;
    // ...
};
```

These functions are declared `private`, which permits a derived class to override them but prevents their direct invocation. The virtual functions can only be called via the base-class.

```
void Widget::process(Gadget &g)
{
    doProcessPhase1(g); // 1. Customised.
    // 2. Perform a series of non-customisable operations.
    doProcessPhase2(g); // 3. Customised.
}
```

Thus, correct order-of-operations can be enforced by `process()` with no requirements on the user or the implementor of the derived class. The function may be invoked directly

```
DerivedWidget dw;
dw.process();
```

or by pointer or reference to the base class.

```
Widget *w = new DerivedWidget;
w->process();
```

2.4 Constrain Class Usage

> *If there's more than one way to do a job, and one of those ways will result in disaster, then somebody will do it that way.* Major Edward A. Murphy, Jr., 1949. [Murphy's Law]

A class is more intuitive if users are prevented from inappropriate access. The following are some useful access restriction idioms.[7]

- Use `const` for pointer or reference arguments that are meant to be non-modifiable. e.g.,

  ```
  void my_func(const MyClass &m); // my_func cannot modify m
  ```

- Use `const` to denote member functions that do not alter the class object itself. e.g.,

  ```
  bool MyClass::isempty() const; // isempty cannot modify *this
  ```

- If a class is not supposed to be copied or assigned, enforce this by declaring the copy constructor or copy-assignment operator `private`, and not defining an implementation [Mey98, Item 27], [Sut, Item 69].[8] e.g.,

[7] These techniques are important for public interfaces, but less critical for private interfaces, where use is restricted to the library implementor not the user. However, they can still be useful in the private interface for catching design bugs.

[8] An alternative idiom is to privately inherit from a "non-copyable" class (e.g., see Boost `noncopyable` [BCL]).

```
private:
    MyClass(const MyClass& other);            // prevent copying
    MyClass& operator=(const MyClass& other); // prevent assignment
```

- If a class must be used only as a base-class, enforce this by declaring its constructor `protected`. e.g.,

```
protected:
    MyClass(); // enforce derivation
```

Various other access constraints are possible. For example, restricting class access to a single other class only is possible by making its interface `private` and declaring the other class a `friend`. An object can be forced to be heap allocated by declaring its destructor `private`. Alternatively, heap allocation can be forbidden by privately declaring a class-specific `new`, and not defining an implementation. It is also possible to define constraints on template operations [Sut02, Str].

The following example of access restriction is a generic container for holding object pointers, adapted from [Mey98, Item 42]. The motivation of this example is to create a template class `Stack<T>` for pointers without inducing code bloat. Bloat is common in programs that make heavy use of templates as each instantiated type generates its own version of the code. The solution given here is to implement the bulk of the class operations using `void*`, which is generic but not type-safe. Type-safety is then added by defining a lightweight templatised *interface* class. Since the unsafe low-level implementation class `GenericStack` should not be accessible to users, access restriction techniques are applied.

```
class GenericStack {
protected:
    GenericStack() {}   // cannot create GenericStack directly, must derive
    ~GenericStack() {}  // cannot destroy derived class via pointer-to-base
    void push(void *ptr)
    {
        data.push_back(ptr);
    }
    void *pop()
    {
        void *p = data.back();
        data.pop_back();
        return p;
    }
    std::size_t size() const
    {
        return data.size();
    }
private:
    GenericStack(const GenericStack& other);            // prevent copying
    GenericStack& operator=(const GenericStack& other); // prevent assignment

    std::vector<void*> data;
};
```

`GenericStack` cannot be created or destroyed directly, as its constructor and destructor are `protected`, so it can only be used as a base-class. The copy-constructor and copy-assignment operator are declared private and left undefined, so this class, and any class derived from it, cannot be copied. The `Stack<T>` class inherits privately from `GenericStack`, and simply forwards its operations on to the generic implementation. Its templatised interface enforces compile-time type-safety.

```
template<class T>
class Stack : private GenericStack {
public:
    void push(T *ptr) { GenericStack::push(ptr); }
    T* pop() { return static_cast<T*>(GenericStack::pop()); }
    std::size_t empty() const { return GenericStack::size(); }
};
```

Aside. *Idioms are the vernacular of a language; the way the language is used and understood by a native speaker. Experienced C++ programmers convey design intentions clearly and concisely by following certain idiomatic conventions. Ignorance of common C++ idioms may elicit confusion when reading another person's code. Idiom fluency, on the other hand, aids communication. The declaration of a* `private` *copy-constructor to indicate a non-copyable class is one example. Another is the convention of declaring a* `virtual` *destructor if a class is designed to be used as a polymorphic base-class, but declaring a non-virtual destructor if it should not be used as a polymorphic base-class. These idioms can be confusing to the novice, but bring clarity to the initiated.*

2.5 Resource Acquisition is Initialisation

The "resource acquisition is initialisation" (RAII) idiom [Str00a, Section 14.4] is an important technique for managing resources. The basic idea is that resources are acquired in constructors and are released in destructors. This automates the release of resources, since stack-allocated objects are destroyed implicitly when they go out-of-scope. RAII improves code safety in two ways. It removes the need for programmers to remember to release resources, and it ensures release in the event of an exception being thrown.

Common resource operations are obtaining-releasing dynamic memory, locking and unlocking mutexes, opening-closing files, etc. A simple example of RAII, from [Str00a, Section 14.4].

```
class File_ptr {
    FILE *p;
public:
    File_ptr(const char *n, const char *a) { p = fopen(n, a); }
    ~File_ptr() { fclose(p); }
    operator FILE*() { return p; }
};
```

The file pointer handle class behaves entirely like a normal file pointer, but automatically closes the file when it goes out-of-scope.

```
{
    File_ptr fp("hello.txt", "w");
    fprintf(fp, "Hello world\n");
} // File automatically closed
```

3 Reusable Design Strategies

Programs designed top-down for a specific application rarely lead to reusable code development. Reusable modules are usually constructed as a product of bottom-up design and by factoring out commonly used elements of a program structure. Two useful strategies for reusable design are component-oriented design and layered design.

3.1 Component Design

Component-oriented design is the process of building general-purpose classes and modules that perform certain tasks, but are not tailored to any specific program. The C++ standard library is a good example, notably the STL container classes. The Boost C++ library [BCL] is another example of a collection of reusable components. An advantage of designing and accumulating reusable components is that they form an ongoing toolbox of high-level functionality that can be carried forward from project to project. With time and use, they tend to become increasingly reliable and bug-free, so that debugging can be focused on other parts of a program.

When encountering a new problem that is not addressed by components already available in your toolbox, prefer to design a new general-purpose component rather than writing code to solve the specific problem at hand. Component-based designs tend to be more elegant and easier to adapt as a program evolves. Oftentimes designing for the general-case also results in a solution that is *simpler* than one designed for a specific-case.

3.2 Layered Design

Layered design is the concept of building higher-level systems out of lower-level subsystems. Each subsystem is a separate library that performs a particular set of tasks and presents its own public interface. Bottom-level subsystems perform primitive operations, such as file input-output or thread management. Higher-level subsystems combine facilities from the layers below and build more powerful abstractions from them. The result is a multi-layered collection of libraries that provides a flexible combination of primitive operations and high-level abstractions. The C++ Adaptive Communication Environment (ACE) [Sch] is a good example.[9]

A potential problem with layered design is that dependencies can occur between subsystems. Ideally, each subsystem is decoupled from the others, but this is difficult to realise in practice. In particular, high-level layers tend to be dependent on multiple subsystems from lower layers. This means a program using a particular library has to also import files (i.e., headers, etc) of various dependent libraries. Dependence on auxiliary libraries is an oft overlooked complexity in managing larger projects, and a significant deterrent to reuse, especially if the number of auxiliary libraries are many, or if they are large and only small portions are actually needed. An ideal layered design will partition disparate components of a library into separate sub-libraries that can be imported separately as required.

3.3 Portable Design

Designing for portability is related to layered design. Most real world applications are not written entirely in standard C++, but utilise additional libraries, including non-portable interfaces for specific platforms. It is important to prefer standard facilities where possible, and to avoid littering a program with non-portable operations. When compelled to use a platform-specific interface, it should be isolated in a separate source-file and wrapped in portable interface. This is the lowest-level of layered design—a thin adapter layer that encapsulates a non-portable interface. The advantage of a portable-interface layer is that porting code to a different platform requires reimplementation of only the module beneath the interface, all code built on top of the interface layer works unchanged.

3.4 Designing for Others

Typically, the person who most benefits from a reusable design is the original implementor. Being able to carry over code from one project to the next is a significant boon. However, to convince other people to use a library requires far greater effort.

[9] Although, in its current form, ACE is an enormous monolithic library with many interdependencies. Hopefully a future version will be factored into several smaller decoupled libraries.

Most importantly, software reuse is based on trust. Another person is unlikely to use a library unless they are confident it has been written by someone with reasonable programming skill. Also, the library must meet a need that cannot be easily implemented from scratch (e.g., packages for linear algebra, threads, sockets, etc). For high-performance or safety-critical systems, it might have to meet prescribed performance specifications. Further, the library should come with a certain level of support in terms of documentation and demonstrative examples. And clients developing long-term projects are likely to want continued library support in the form of maintenance, advice and bug-fixes. They may also want assurance of interface stability and backwards compatibility as the library evolves. Finally, keep it simple; it is worth remembering that the most attractive library for any project is usually the simplest one that does the job.

Designing software for reuse by others is not merely about embracing modular design, but also the ongoing development of a reputable, well-supported product.

4 Design Examples

The C++ language incorporates a large number of distinct features—classes, function overloading, virtual functions, templates, etc. In isolation, they support a variety of different programming styles. In combination, they are complementary, and the synergy of their attributes permits powerful and flexible class library designs. This section examines two examples to demonstrate the possibilities of multi-paradigm C++ design.

4.1 Wrapper Class

This example is a cut-down adaptation of the generic wrapper class presented in [Str00b]. The basic idea is to enclose accesses to an object's members within a pair of prefix and suffix functions. A simple scenario is a thread-safe shared object. Before accessing the object, a mutex must be locked and after each access it must be unlocked. This is to be transparent; the user should not be required to perform explicit lock-unlock operations. Also, it is assumed that the class definition cannot be modified, so an intrusive strategy is not possible.

For this demonstration, a dummy mutex class is defined, which simply prints "lock" and "unlock" as required.

```
class Mutex {
public:
    void lock() { cout << "lock" << endl; }
    void unlock() { cout << "unlock" << endl; }
};
```

The wrapper module is composed of two classes. The first class is `CallProxy`, which performs mutex unlocking in its destructor and provides a `->` operator to allow pointer-like access to a `T` object.

```
template<class T>
class CallProxy {
public:
    CallProxy(T &tt, Mutex &m) : t(tt), mtx(m) { }
    ~CallProxy() { mtx.unlock(); }
    T* operator->() { return &t; }

private:
    T &t;
    Mutex &mtx;
};
```

The second class is `Wrap`, which provides a `->` operator that first locks the mutex and then returns a `CallProxy` object.

```
template<class T>
class Wrap {
public:
    Wrap(T &tt, Mutex &m) : t(tt), mtx(m) { }
    CallProxy<T> operator->() { mtx.lock(); return CallProxy<T>(t, mtx); }

private:
    T &t;
    Mutex &mtx;
};
```

These two classes provide transparent lock-unlock wrapping for any given object. To demonstrate, we define a class `MyClass`, with two trivial member-functions, to represent the shared object.

```
class MyClass {
public:
    void f() { cout << "f()" << endl; }
    void g() { cout << "g()" << endl; }
};

int main()
{
    MyClass x;
    Mutex m;
    Wrap<MyClass> wx(x, m); // wrapped handle for x

    wx->f(); // simple pointer-like access
    wx->g();
}
```

The result of this code shows the implicit operations.

```
lock
f()
unlock
lock
g()
unlock
```

The pair of wrapper classes implement a clever variant of RAII—using destructors and object lifetimes to automatically invoke operations. The `Wrap ->` operator first calls the `Mutex::lock()` operation, and then returns a temporary `CallProxy` object, which acts as a handle to the shared `MyClass` object. The `CallProxy ->` operator returns a pointer to the shared object and this invokes its member function. Finally, at the end of the statement, the temporary `CallProxy` object goes out-of-scope and is destructed, calling `Mutex::unlock()`.

4.2 Heterogeneous Container Class

The following example is a heterogeneous queue class, which is able to contain objects of any type. Since the variety of types is not known at compile-time, static type-checking is not possible. Nevertheless, the container provides complete runtime type-safety, and throws an exception if invalid access is made. Type-checking is invisible to the user.

The key component for holding heterogeneous data while still being type-safe is the `ObjectHandle` interface class. This base-class simply provides a pure-virtual function for querying type.

```
class ObjectHandle {
public:
    virtual ~ObjectHandle() {}
    virtual const std::type_info& type() const = 0;
};
```

The actual generic handle for each type is a template class `ObjectHandleT<T>`. This class stores a copy of a `T` object and provides access to it and its type.

```
template<typename T>
class ObjectHandleT : public ObjectHandle {
public:
    ObjectHandleT(const T &t) : obj(t) {}
    const std::type_info& type() const { return typeid(T); }
    const T& get() { return obj; }

private:
    T obj;
};
```

The above classes are implementation details, and should be hidden in the module's private interface (i.e., within a "detail" namespace). These classes are used exclusively to implement the heterogeneous queue operations. (They may also be reused to implement other heterogeneous container types.)

The queue class `HQueue` exports the module's public interface, and provides push, pop, queue-size, and type-querying operations. The public member-functions are templates, and are generic and type-safe. The `push()` function creates a type-specific `ObjectHandleT<T>` which carries type information into the non-template internal queue representation via its base-class. The `pop()` function checks for a match between the requested type and the type of the next item in the queue, and throws an exception otherwise. This operation provides type-safety. The `is_type()` functions permit non-throwing type queries.

```
class HQueue {
public:
    HQueue() {}
    ~HQueue()
    {
        ObjQueue::iterator i;
        for (i = data.begin(); i!= data.end(); ++i)
            delete *i;
    }

    template<typename T>
    void push(const T &t)
    {
        data.push_back(new ObjectHandleT<T>(t));
    }

    template<typename T>
    void pop(T &t)
    {
```

```
        if (!is_type<T>())
            throw HContainerError("HQueue Type Mismatch");
        t = static_cast<ObjectHandleT<T>*>(data.front())->get();
        delete data.front();
        data.pop_front();
    }

    template<typename T>
    bool is_type() const { return typeid(T) == data.front()->type(); }

    template<typename T>
    bool is_type(const T &t) const { return is_type<T>(); }

    std::size_t size() const { return data.size(); }

private:
    HQueue(const HQueue& rhs);              // prevent copying
    HQueue& operator=(const HQueue& rhs);   // prevent assignment

    typedef std::list<ObjectHandle*> ObjQueue;
    ObjQueue data;
};
```

An example of queue usage is shown below.

```
int main()
{
    HQueue hq;
    int i = 5, j;
    double x = 9.8, y;

    hq.push(i);
    hq.push(x);

    cout << hq.is_type(i) << ' '     // implicit type queries
         << hq.is_type(x) << ' '
         << hq.is_type<int>() << ' ' // explicit type queries
         << hq.is_type<double>() << endl;

    //hq.pop(y); // Error: will throw an exception
    hq.pop(j);
    hq.pop(y);

    cout << j << ' ' << y << endl;
}
```

This code produces the following output.

```
1 0 1 0
5 9.8
```

This example is a simple demonstration of how combining inheritance and templates can increase code flexibility. A final note: this implementation is not exception-safe. The `push()` member-function will leak resources if `ObjQueue::push_back()` throws. Making this operation exception-safe is left to the reader as an exercise.

5 Conclusion

It is possible to write bad code in any language, and the freedom of the C++ language permits the creation of truly awful code. However, with care, the language features of C++ promote modularity and encapsulation, and one can write expressive, flexible code that scales well with program size. This paper presents several basic idioms for robust and intuitive code design, with a focus on simplifying correct use. It barely scratches the surface of advanced C++ design methods, for which there is an extensive literature.

6 Further Reading

Koenig's text [KM00] is an excellent introduction to the C++ language, and demonstrates idiomatic C++ style from the outset. Scott Meyer's *Effective C++* books [Mey98, Mey96] provide a fast-track to many essential design techniques for beginner to intermediate programmers. The use of classes in modern C++ design is discussed in depth in Stroustrup's text [Str00a, Chapter 25]. The seminal work on object-oriented design patterns is by Gamma *et al.* [GHJV94]. Alexandresu's text presents advanced C++ design techniques [Ale01]. One of the best ways to absorb good design practices is to read good source-code. A great number of quality C++ libraries can be found at Boost [BCL], Sourceforge [SF], and the *Available C++ Libraries FAQ* [Loc]. Prefer libraries written after 1998, when the C++ ISO standard was finalised. Also many current design idioms have only become prominent since about 2000.

References

[Ale01] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[BCL] The Boost C++ Libraries. `http://www.boost.org`.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[HS00] J. Hyslop and H. Sutter. Conversations: Virtually yours. *C/C++ Users Journal*, December 2000. `http://www.cuj.com/experts/`.

[KM00] A. Koenig and B. Moo. *Accelerated C++*. Addison-Wesley, 2000.

[Loc] N. Locke. Available C++ libraries FAQ. `http://www.trumphurst.com/cpplibs1.html`.

[Mey96] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, 1996.

[Mey98] S. Meyers. *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1998.

[Mey00] S. Meyers. How non-member functions improve encapsulation. *C/C++ Users Journal*, February 2000. `http://www.cuj.com/`.

[Sch] D.C. Schmidt. The adaptive communication environment (ACE). `http://www.cs.wustl.edu/~schmidt/ACE.html`.

[SF] Source Forge. `http://sourceforge.net/`.

[Str] B. Stroustrup. C++ style and technique FAQ. http://www.research.att.com/~bs/bs_faq2.html.

[Str95] B. Stroustrup. Why C++ isn't just an object-oriented programming language. In *Addendum to OOPSLA'95 Proceedings*, 1995. http://www.research.att.com/~bs/papers.html.

[Str00a] B. Stroustrup. *The C++ Programming language (Special Edition)*. Addison-Wesley, 2000. Appendices B (Compatibility), D (Locales) and E (Exception Safety) are available online at http://www.research.att.com/~bs/3rd.html.

[Str00b] B. Stroustrup. Wrapping C++ member function calls. *C++ Report*, June 2000. http://www.research.att.com/~bs/papers.html.

[Sut] H. Sutter. Guru of the week. http://www.gotw.ca/gotw/index.htm.

[Sut98] H. Sutter. Uses and abuses of inheritance, part 1. *C++ Report*, October 1998. http://www.gotw.ca/publications/index.htm.

[Sut99] H. Sutter. Uses and abuses of inheritance, part 2. *C++ Report*, January 1999. http://www.gotw.ca/publications/index.htm.

[Sut01] H. Sutter. Sutter's mill: Virtuality. *C/C++ Users Journal*, Sepember 2001. http://www.gotw.ca/publications/index.htm.

[Sut02] H. Sutter. Extensible templates: Via inheritance or traits? *C/C++ Users Journal*, February 2002. http://www.gotw.ca/publications/index.htm.