Wrapping C++ Member Function Calls

Bjarne Stroustrup

AT&T Labs - Research
Florham Park, New Jersey
USA

ABSTRACT

This paper presents a simple, general, and efficient solution to the old problem of ‘‘wrapping’’ calls to an object in pairs of prefix and suffix code. The solution is also non-intrusive, applies to existing classes, allows the use of several prefix/suffix pairs, and can be implemented in 15 simple lines of Standard C++. A robust version of the wrapper is also presented. The claim of efficiency is backed by measurement.

The paper is organized around a series of examples evolving the basic idea into a final robust template class.

1 Introduction

Often, we want to ‘‘wrap’’ a piece of code with some prefix code and some suffix code. For example:

```
grab-lock do-something release-lock
begin-transaction do-something end-transaction
trace-entry do-something trace-exit
```

Typically, there is a prefix/suffix pair that should be applied to many different sections of code. In particular, in a language with classes – such as C++ – the problem often becomes one of ensuring that a *prefix()* is called before a call of a member function and *suffix()* after the call. This is easily achieved explicitly for a few calls. For example:

```
void fct(X* p)
{
    prefix();
    p->f();
    suffix();

    prefix()
    p->g();
    suffix();
}
```

However, application programmers must remember to bracket each call in its proper *prefix()/suffix()* calls. This is tedious and error prone.

The obvious alternative is to add the *prefix()/suffix()* calls to the definitions of the member functions that need them. For example:

```
class X {
public:
    void f() { prefix(); /* f´s own stuff */ suffix(); }
    void g() { prefix(); /* g´s own stuff */ suffix(); }
    // ...
};
```

```
void fct(X* p)
{
    p->f();
    p->g();
}
```

This solves the problem for the class user, but is still tedious for the class implementer. Worst of all, this solution requires foresight by the class implementer. Because the solution is intrusive – the class member function code must be modified to add, remove, or change a suffix – wrapping can only be done by someone able and willing to modify the source code.

One advantage of this approach is that it allows some, but not all functions to be wrapped. This can sometimes be a significant advantage. Consider the case where the prefix/suffix provides locking. That is, the class is a form of monitor [Hoare,1974]. In that case, it is not uncommon that a few functions can be performed without locking because they don't modify shared data or access only data that is accessed and modified atomically [Mitchell,1979].

This paper will concentrate on the case where every operation on a class needs to be wrapped.

2 History

''Wrapping'' is an old problem that has been solved many times in various languages and contexts. Monitors provide a solution to the problem of controlling access to a resource in a concurrent system by wrapping calls in a acquire-lock-or-wait and release-lock pair. Many languages provide primitives for wrapping code in acquire/release lock pairs (for example, Mesa's *MONITOR* [Mitchell,1979], Java's *synchronized* [Lea,1997], and Modula-3's *LOCK* [Nelson,1991]).

More general solutions are provided by CLOS where a pair of *:before* and *:after* methods can wrap calls to objects of classes derived from the one providing the *:before* and *:after* methods [Keene,1989]. I briefly adopted a variant of that idea for C++'s direct ancestor C with Classes [Stroustrup,1994]. There, one could define a function that would implicitly be called before every call of every member function (except the constructor) and another that would be implicitly called before every return from every member function (except the destructor). The functions providing this prefix/suffix semantics were called *call()* and *return()*. They were used to provide synchronization for the monitor class in the original task library [Stroustrup,1980]:

```
class monitor {
    // ...
    call()  { /* grab lock */ }
    return() { /* release lock */ }
    // ...
};

class X : public monitor {
public:
    void f();
    void g();
    // ...
};

void fct(X* p)
{
    p->f();    // monitor::call(); f()'s own stuff; monitor::return()
    p->g();    // monitor::call(); g()'s own stuff; monitor::return()
}
```

Call and return functions were removed from the language because nobody (but me) used them and because I completely failed to convince people that *call()* and *return()* had important uses.

In 1988, Mike Tiemann suggested an alternative solution called ''wrappers'' [Tiemann,1988]:

```
struct foo {
    foo();
    ~foo();
```

```
        virtual int ()foo(int(foo::* pmf)(int), int i)    // virtual wrapper
        {
                prefix();
                int r = (this->*pmf)(i);            // invoke a function
                suffix();
                return r;
        }

        int g(int);
        virtual int h(int);
};
```

Such a wrapper function was syntactically identified by a pair of parentheses in front of the class name and would wrap calls to functions of its class. The first argument of a wrapper was the member function to invoke, the second and subsequent arguments were the arguments to that function. The basic idea was that a member function called by a user was transformed a call of the wrapper with the arguments needed for the wrapper to do the actual call. For example:

```
void fct(foo* p)
{
        int i = p->g(1);        // p->()foo(&foo::f,1)
        int j = p->h(2);        // p->()foo(&foo::g,2)
        // ...
}
```

This proposal died – after some experimental use – because of complexities of handling argument and return types, and because it was intrusive. That is, you wrapped a class by deriving it from a wrapper base class and that base class had to be written to deal with all combinations of argument and return types needed by the derived classes. This required too much foresight.

This proposal had the interesting property that the wrapper had access to the arguments and the return value of a call. Also, different wrapper functions could be provided for functions with different types.

3 Prefix

In the following sections, a solution to the wrapping problem is introduced in stages. The solution does not require language changes; it consists of two simple template classes.

Overloading the `->` has long been a popular way of specifying a prefix. For example:

```
template<class T>
class Prefix {
        T* p;
public:
        Prefix(T* pp) :p(pp) { }
        T* operator->() { /* prefix code */ return p; }
};

X my_object;
Prefix<X> pref_obj(&my_object);        // available for prefixed use

void fct(Prefix<X> p)
{
        p->f();    // prefix code; f()
        p->g();    // prefix code; g()
}
```

Note how the prefix is attached to the object (and its functions) non-intrusively. I can use ***Prefix*** to control access to classes that I cannot change. I can also use ***Prefix*** to control access to classes that have not been specifically been designed for controlled access. In this, the templated ***Prefix*** approach is more general and more flexible than the C with Classes approach and the Tiemann wrapper approach. These approaches required that controlled classes be derived from a specific base class. The templated ***Prefix*** approach requires less foresight on the part of the programmer.

4 Suffix

Unfortunately, it was not obvious how to extend the templated *Prefix* approach to deal with suffixes. However, it can be done. One non-intrusive way of getting something done ‘‘later’’ is to create an object with a class with a destructor that does that something. The destructor will be executed at the end of the object’s lifetime. For example:

```
class Suffix {
public:
        ~Suffix() { /* suffix code */ }
};

void fct(X* p)
{
        Suffix suf;
        p->f();
        // ...
        // the suffix code is implicitly executed here
}
```

This is a simplified variant of the technique commonly known as ‘‘resource acquisition is initialization’’ [Stroustrup,2000]. This technique has the nice property that the suffix code is executed even if *f()* throws an exception.

The problems with the ‘‘resource acquisition is initialization’’ technique in this context are:
[1] the user must explicitly declare an object
[2] the user must name the *Suffix* object
[3] the suffix isn’t executed until the end of the block

5 Prefix and Suffix

These problems can be solved by combining the use of a destructor to invoke a suffix with the use of an *operator->* to invoke a prefix. As before, *operator->* executes the prefix and returns something that identifies the object for which the function is to be called:

```
template<class T>
class Wrap {
        T* p;
public:
        Wrap(T* pp) :p(pp) { }
        Call_proxy<T> operator->() { prefix(); return Call_proxy<T>(p); }
};
```

What is different here is that the value returned by *operator->()* is an object holding the pointer to the object to be called rather that the pointer itself. We can define *Call_proxy*, with a destructor that calls the suffix:

```
template<class T>
class Call_proxy {
        T* p;
public:
        Call_proxy(T* pp) :p(pp){ }
        ~Call_proxy() { suffix(); }
        T* operator->() { return p; }
};
```

We can now write:

```
#include<iostream>
using namespace std;

void prefix() { cout << "prefix "; }
void suffix() { cout << " suffix\n"; }
```

```
template<class T> class Call_proxy { /* ... */ };

template<class T> class Wrap { /* ... */ };

class X {   // one user class
public:
      X() { cout << "make an X\n"; }
      int f() const { cout << "f()"; return 1; }
      void g() const { cout << "g()"; }
};

class Y {   // another user class
public:
      Y() { cout << "make a Y\n"; }
      void h() const { cout << "h()"; }
};

int main()        // simple test code
{
      Wrap<X> xx(new X);
      Wrap<Y> yy(new Y);

      if (xx->f()) cout << "done\n";
      xx->g();
      yy->h();
      return 0;
}
```

Each call of *xx* and *yy* is bracketed by a pair of prefix()/suffix() calls, so the program produced:

```
make an X
make a Y
prefix f() suffix
done
prefix g() suffix
prefix h() suffix
```

Note that wrapping of all calls of all functions is not a simple transformation. For example, you cannot write a function or a macro to do so. Even writing a macro to wrap a single call is nontrivial because of the need to perform the suffix before returning the result.

It is important that *Wrap* is implemented completely in Standard C++ [Stroustrup,2000]. No language extensions or preprocessor magic are required. This implies that variations of the idea can be used to suit needs and tastes. Also, the idea is immediately applicable using current C++ implementations.

6 Ownership

The simple *Wrap* and *Call_proxy* classes do not implement a general and safe model of ownership. In particular, they do not define copy constructors and assignment operators that ensure that objects pointed to are deleted exactly once.

Also, *Call_proxy* isn't intended as a general class, so we can ensure that *Call_proxy*s are created by *Wrap* only and their lifetimes are limited to a single wrapped call:

```
template<class T> class Wrap;

template<class T>
class Call_proxy {
      T* p;
      mutable bool own;

      Call_proxy(T* pp) :p(pp), own(true) { }        // restrict creation
      Call_proxy(const Call_proxy& a) : p(a.p), own(true) { a.own=false; }
      Call_proxy& operator=(const Call_proxy&);       // prevent assignment
public:
      template<class U> friend class Wrap;
```

```
        ~Call_proxy() { if (own) suffix(); }

        T* operator->() const { return p; }
};
```

The *own* variable is needed to ensure that only the last object in a chain of copies executes *suffix()*. The way *Wrap* uses *Call_proxy*, no copying is needed, and most (all?) compilers are clever enough to avoid spurious copies. However, to be cautious, I introduced *own* for the benefit of compilers with poor optimizing skills.

The wrapper itself is intended for general use. Objects with a lifetime controlled by the user are passed to *Wrap* as object (by reference). Objects that *Wrap* is supposed to delete are passed as pointers. In the latter case, the wrapper maintains a use count:

```
template<class T>
class Wrap {
        T* p;
        int* owned;
        void incr_owned() const { if (owned) ++*owned; }
        void decr_owned() const { if (owned && --*owned == 0) { delete p; delete owned; } }
public:
        Wrap(T& x) :p(&x), owned(0) { }
        Wrap(T* pp) :p(pp), owned(new int(1)) { }

        Wrap(const Wrap& a) :p(a.p), owned(a.owned) { incr_owned(); }
        Wrap& operator=(const Wrap& a)
        {
                a.incr_owned();
                decr_owned();
                p = a.p;
                owned = a.owned;
                return *this;
        }

        ~Wrap() { decr_owned(); }

        Call_proxy<T> operator->() const { prefix(); return Call_proxy<T>(p); }

        T* direct() const { return p; }  // extract pointer to wrapped object
};
```

See [Stroustrup,2000] for a discussion of the non-intrusive use-count scheme used.

7 Parameterization

So far, *Wrap* has called the global functions *suffix()* and *prefix()*. That is not flexible enough or general enough. We'd like to wrap different objects with different prefix/suffix pairs and to wrap a single object in more than one pair. This is simply handled by making *prefix* and *suffix* parameters.

When parameterizing *Wrap*, the major question is whether to provide per-class parameterization or per-object parameterization. I chose the latter because it is the most flexible solution.

Call_proxy must have a suffix parameter:

```
template<class T, class Pref, class Suf> class Wrap;

template<class T, class Suf>
class Call_proxy {
        T* p;
        mutable bool own;
        Suf suffix;

        Call_proxy(T* pp, Suf su) :p(pp), own(true), suffix(su) { }          // restrict creation

        Call_proxy& operator=(const Call_proxy&);       // prevent assignment
public:
        template<class U, class P, class S> friend class Wrap;
        Call_proxy(const Call_proxy& a) : p(a.p), own(true), suffix(a.suffix) { a.own=false; }
        ˜Call_proxy() { if (own) suffix(); }

        T* operator->() const { return p; }
};
```

Wrap needs a prefix and a suffix parameter:

```
template<class T, class Pref, class Suf>
class Wrap {
        T* p;
        int* owned;
        void incr_owned() const { if (owned) ++*owned; }
        void decr_owned() const { if (owned && --*owned == 0) { delete p; delete owned; } }
        Pref prefix;
        Suf suffix;
public:
        Wrap(T& x, Pref pr, Suf su) :p(&x), owned(0), prefix(pr), suffix(su) { }
        Wrap(T* pp, Pref pr, Suf su) :p(pp), owned(new int(1)), prefix(pr), suffix(su) { }

        Wrap(const Wrap& a)
                :p(a.p), owned(a.owned), prefix(a.prefix), suffix(a.suffix) { incr_owned(); }
        Wrap& operator=(const Wrap& a)
        {
                a.incr_owned();
                decr_owned();
                p = a.p;
                owned = a.owned;
                prefix = a.prefix;;
                suffix = a.suffix;
                return *this;
        }

        ˜Wrap() { decr_owned(); }

        Call_proxy<T,Suf> operator->() const { prefix(); return Call_proxy<T,Suf>(p,suffix); }

        T* direct() const { return p; } // extract pointer to wrapped object
};
```

Given that, here is a simple test program:

```
#include<iostream>
using namespace std;

void prefix() { cout << "prefix "; }
void suffix() { cout << " suffix\n"; }

class X {  // one user class
public:
        X() { cout << "make an X\n"; }
        ˜X() { cout << "destroy an X\n"; }
        int f() const { cout << "f()"; return 1; }
        void g() const { cout << "g()"; }
};

class Y {  // another user class
public:
        Y() { cout << "make a Y\n"; }
        ˜Y() { cout << "destroy a Y\n"; }
        void h() const { cout << "h()"; }
};

struct Pref { void operator()() const { cout << "Pref "; } };
struct Suf { void operator()() const { cout << " Suf\n"; } };
```

```
int main()      // test program
{
      Wrap<X,void(*)(),void(*)()> xx(new X,prefix,suffix);
      Wrap<Y,void(*)(),void(*)()> yy(new Y,prefix,suffix);
      Wrap<X,void(*)(),void(*)()> x2 = xx;
      X x;
      Wrap<X,Pref,Suf> x3(x,Pref(),Suf());

      if (xx->f()) cout << "done\n";
      xx->g();
      x2->g();
      xx = x2;
      x2->g();
      x3->g();
      yy->h();
      return 0;
}
```

Here, I use function objects primarily to get better inlining and therefore better run-time performance on an average C++ implementation (see §9). However, function objects have an important advantage over functions in that they can be used to hold data. For example, if we want to wrap calls in a pair of lock/unlock operations, we often need to say what lock is to be used. This can be done either by having one function for each lock, or by having function objects that can be initialized with the lock to be used. For example:

```
struct Lock {
      sys_lock lck;
      Lock(sys_lock& x) : lck(x) { }
      void operator()() const { grab(lck); }
};

struct Unlock {
      sys_lock lck;
      Unlock(sys_lock& x) : lck(x) { }
      void operator()() const { release(lck); }
};

Wrap<X,Lock,Unlock> x3(x,Lock(screen_lock),Unlock(screen_lock));
Wrap<X,Lock,Unlock> x4(y,Lock(lock3a),Unlock(lock3a));
```

8 Notational Convenience

The flexibility was bought at the expense of notational convenience. I don't think anyone likes to read or write declarations like:

```
      Wrap<X,void(*)(),void(*)()> xx(new X,prefix,suffix);
```

Typically, only a few forms of wrapping are used in a program and people want to refer to those with the greatest degree of simplicity. The most general notation is rarely necessary.

A derived class can be used to provide more concise notation by specialization. Deriving a class also opens the possibility of defining the most appropriate parameters to constructors:

```
template<class T>
class Shared : public Wrap<T,Lock,Unlock>
{
public:
      Shared(T& obj,sys_lock& lck) : Wrap<T,Lock,Unlock>(obj,Lock(lck),Unlock(lck)) { }
      Shared(T* ptr,sys_lock& lck) : Wrap<T,Lock,Unlock>(ptr,Lock(lck),Unlock(lck)) { }
};

Shared<X> x3(x,screen_lock);
Shared<X> x4(y,lock3);
```

Unfortunately, we have to specify both the wrapped type and the wrapped object. Because of the generality of the constructor mechanism, it is not possible to deduce the one from the other. However, we could define a function to take advantage of type deduction:

```
template<class T, class Pref, class Suf>
Wrap<T,Pref,Suf> make_wrap(T& x, Pref pr = Pref(), Suf su = Suf())
{
    return Wrap<T,Pref,Suf>(x,pr,su);
}

void f(X& x)
{
    g(make_wrap(x));
    // ...
}
```

Unfortunately, we must specify the type of the object to which we assign the result of *make_wrap()*. In this example, we must define the argument type for *g()*. This defeats this technique for simplifying the *Wrap* notation except where the target is itself a template.

Naturally, if we wanted to wrap only a single type, or if we wanted to make wrapping a particular type notationally simple, we could introduce a name for that particular wrapper type. For example:

```
template<class T>
struct Tracer : public Wrap<T,void(*)(),void(*)()> {
    Tracer(T& x) : Wrap<T,void(*)(),void(*)()>(x,trace_on,trace_off) { }
};

X x;
Tracer<X> xx(x);

typedef Tracer<X> Xtracer;
Xtracer xxx(x);
```

9 Efficiency

The *Wrap* class and its auxiliary *Call_proxy* class have doubled in source code size from the initial version to the version that is more flexible, general, and safer for a programmer to use. How does this evolution affect efficiency? To find out I did a simple timing test wrapping calls of an empty, noninlined member function.

```
class X { // one user class
public:
    void g() const; // note: not even virtual
};

void X::g() const {/* cout << "g()"; */}
```

I measured direct calls of the function

```
X* p = new X;
// ...
p->f();
```

and calls using the simple *Wrap* from §5

```
X x;
Wrap<X> xx(&x);    // simple Wrap from §5
// ...
xx->f();
```

and calls using the robust and parameterized *Wrap* from §7 with pointers to inline functions as prefix and suffix

```
inline void pref() { }
inline void suf() { }
```

```
Wrap<X, void(*)(), void(*)()> xx(&x, pref, suf);    // robust and parameterized Wrap from §7
// ...
xx->f();
```

and calls using the robust *Wrap* from §7 using function objects as prefix and suffix to simplify inlining

```
struct Pref { void operator()() const { } };     // use to function objects to simplify inlining
struct Suf { void operator()() const { } };

Wrap<X, Pref, Suf> xx(&x, Pref(), Suf()); // robust and parameterized Wrap from §7
// ...
xx->f();
```

Thus, the measurements show the overhead of wrapping compared to the overhead of a call of an empty function.

Because the simple *Wrap* doesn't actually perform any additional computation, we can expect a good optimizer to eliminate all overhead. However, the robust *Wrap* does some computation related to ownership, so we should expect to pay a little – but only a little – for using it.

The code is available from my home pages: http://www.research.att.com/~bs/papers.html.
To get an idea of the impact of compiler technology and machine architecture, I ran the tests on three different machine architectures using a total of seven different C++ implementations.

As is typical for non-trivial C++ code, the most obvious difference was between runs using different level of optimization.

Run times (debug and optimized)						
	impl.1d	impl.1	impl.3d	impl.3	impl.4d	impl.4
no Wrap	.14	.06	.21	.09	.42	.03
simple Wrap, inline function	.47	.07	.45	.10	1.82	.06
robust Wrap, inline function	.74	.25	.68	.23	2.40	.14
robust Wrap, function object	.86	.11	.59	.15	2.52	.10

These numbers are the average of many runs and the results between runs do not show significant variation. The unit is microseconds per wrapped or unwrapped call.

I did not try to tune the code by selecting compiler and optimizer options. In each case, a default ''debug mode'' and a default ''release mode'' was used. All runs were done with all Standard C++ facilities enabled.

It may surprise some that the run-time cost of using debug mode can be a factor of 25. Disabling inlining and all non-trivial operations leave templated code very inefficient. Templates and inlining were designed with optimization in mind.

Note that for implementations 1 and 3, there is no statistical difference between the cost of a direct call and the cost of a call through the simple wrapper.

Please not compare the different implementations – they run of completely different hardware and these simple measurements are not suitable benchmarks.

Run times (optimized)							
	impl.1	impl.2	impl.3	impl.4	impl.5	impl.6	impl.7
no Wrap	.06	.07	.09	.03	.03	.04	.04
simple Wrap, inline function	.07	.07	.10	.06	.15	.14	.06
robust Wrap, inline function	.25	.24	.23	.14	.22	.28	.12
robust Wrap, function object	.11	.17	.15	.10	.15	.22	.09

Implementations 2 and 3 ran on the same machine. Similarly, implementations 4, 5, 6, and seven ran on the same machine. I conclude that the overhead depends more on compiler technology than on machine architecture.

Note that implementations 1, 2, and 3 provide proof by example that simple wrapping need not incur overhead. Implementations 1 and 3 show that the cost of robust wrapping can be less than a function call, though it typically is equivalent to almost two function calls.

Clearly, implementations are still better at inlining function objects than at optimizing calls through a pointer to an inline function.

Two implementations didn't support friend templates, so I had to modify *Call_proxy* to make more functions public. However, this does not affect the generated code – beyond allowing code to be generated from compilers that isn't yet quite up to the standard in this area.

10 Limitations

A fundamental limitation of this prefix/suffix approach is that neither prefix nor suffix has access to the arguments to the called function or the result produced. For example, it is not possible to provide a prefix/suffix to record the results of calls or to map calls into message send operations. In this, the prefix/suffix approach differs from the Tiemann wrappers [Tiemann,1988].

Another limitation of this approach is that the prefix and suffix are independent. For example, it is not possible to write a prefix/suffix pair that catch exceptions thrown by the wrapped functions. It is not possible to open a try-block in the prefix and close it in the suffix.

However, it is possible for a prefix and a suffix to communicate. A simple use of that is for a prefix and a suffix to access the same variable as is typically done for locks and trace state. To do the call, *Wrap* and *Call_proxy* get a pointer to the object called. Thus, they might keep a record of objects accessed, but they do not know which function is called for those objects.

11 Techniques

A wrapper can wrap every class. Consequently, it can wrap a wrapper. This can be useful. For example:

```
Tracer< Shared<X> > xx;  // trace accesses to shared data
```

Alternatively, several operations can be combined into a single prefix or suffix. For example:

```
struct TSpref {
    sys_lock lck;
    TSpref(sys_lock& x) : lck(x) { }

    void operator()() const { trace_on(); grab(lck); }
};

struct TSsuf {
    sys_lock lck;
    TSsuf(sys_lock& x) : lck(x) { }

    void operator()() const { release(lck); trace_off(); }
};

template<class T>
class Trace_lock : Wrap<T,TSpref,TSsuf> {
    Trace_lock(T& obj, sys_lock& x) : Wrap<T,TSpref,TSsuf>(obj,TSpref(x),TSsuf(x)) { }
    Trace_lock(T* ptr, sys_lock& x) : Wrap<T,TSpref,TSsuf>(ptr,TSpref(x),TSsuf(x)) { }
    // ...

};

Trace_lock xx(x,screen_lock);
```

This is more work to define, but in general more flexible. To a user, the details of how a wrapper is implemented is of little practical interest.

As mentioned in §1, it is not always a good idea to apply the prefix and suffix to every access to an object. When we need to access an object directly, we can use the original object. For example:

```
void code(X* p)
{
    Shared<X> xx(*p);

    int i = xx->f(2);          // controlled access
    int j = p->f(3);           // direct access
    // ...
}
```

However, this approach has the problem that the programmer has to remember what the original object was. Typically it is simpler and less error-prone to extract a pointer to the wrapped object from the wrapper using the *direct*( ) function supplied for that purpose. For example:

```
void user(Shared<X>& xx)
{
        int i = xx->f(2);               // controlled access
        int j = xx.direct()->f(3);      // direct access
        // ...
}
```

12 Conclusions

Wrapping calls to member functions of a class by a prefix/suffix pair is simple and efficient and requires no extensions to Standard C++. Wrapping is non-intrusive and does not place requirements on the wrapped class. In particular, a wrapped class need not be derived from a particular base class. Current C++ implementations are capable of coping efficiently with the technique. The primary limitation of the prefix/suffix approach is that the arguments to a call and its return value is not accessible to the prefix/suffix.

13 Acknowledgements

My measurements were simplified by code borrowed from Andrew Koenig [Koenig,2000].

14 References

[Hoare,1974]	C. A. R. Hoare: *Monitors: An Operating System Structuring Concept*. Communications of the ACM 17(10), October 1974.
[Keene,1989]	Sonya A. Keene: *Object-Oriented programming in Common Lisp*. Addison-Wesley. 1989. ISBN 0-201-17589-4.
[Lea,1997]	Doug Lea: *Concurrent Programming in Java*™. Addison-Wesley. 1997. ISBN 0-201-69581-2.
[Koenig,2000]	Andrew Koenig and Barbara Moo: *Performance: Myths, Measurements, and Morals*. JOOP 13(1,2). January and March 2000.
[Mitchell,1979]	James G. Mitchell, et.al.: *Mesa Language Manual*. XEROX PARC, Palo Alto, CA. CSL-79-3. April 1979.
[Nelson,1991]	Greg Nelson (Editor): *Systems Programming with Modula-3*. Prentice Hall. 1991. ISBN 0-13-590464-1.
[Stroustrup,1980]	Bjarne Stroustrup: *A Set of C Classes for Co-routine Style Programming*. Bell Laboratories Computer Science Technical Report CSTR-90. November 1980.
[Stroustrup,1995]	Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1995. ISBN 0-201-54330-3.
[Stroustrup,2000]	Bjarne Stroustrup: *The C++ Programming language (Special Edition)*. (2nd edition, 1991; 3rd edition, 1997). Addison-Wesley. ISBN 0-201-88954-4 and 0-201-70073-5.
[Tiemann,1988]	Michael Tiemann: *‘‘Wrappers:’’ Solving the RPC problem in GNU C++*. Proc. USENIX C++ Conference. Denver, CO. October 1988.