Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++

Scott Meyers and Moises Lejter

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-51
August 1991

Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++

Scott Meyers and Moises Lejter

sdm@cs.brown.edu and mlm@cs.brown.edu
Department of Computer Science
Brown University, Box 1910
Providence, RI 02912

Abstract

In this paper we argue that there is sufficient experience with C++ to justify the development of a tool that examines C++ programs for the presence of likely programmer errors, and we describe a number of common mistakes that could be detected by such a tool. We show that such a tool would be both straightforward to develop and efficient to apply. We also discuss how such a tool could be extended to detect violations of design constraints expressed in some as-yet-to-be-developed C++ metalanguage.

Introduction

It is a fact of life that programmers make mistakes when writing programs. The kinds of mistakes that interest us are not those that are errors (i.e., violations of the rules of the language), but are instead legal constructs that mean something other than what the programmer likely intends them to mean. Some of the mistakes of this kind are straightforward enough that it is possible to write a program to detect them, and some are common enough that it is worthwhile to do so.

C programmers often rely on lint to ensure that their software is free of common errors of this kind. In some sense, lint is an distillation of the collective historical experience of the C programming community, a terse summary of the lessons learned by many people from many programs over a long period of time. In this paper, we propose that there is now sufficient experience with C++ to initiate the development of a C++ counterpart to lint, a lint++.

Since C++ is based on C, a straightforward approach would be to base a lint++ on lint. Our experience in working with C++ and in teaching the language to scores of professional C programmers, however, indicates that the kinds of errors made in C++ are fundamentally different from those made in C, and are a direct outgrowth of the new features offered by C++. In this paper, we identify a number of common C++ programming errors, and we describe how a lint++ could detect them.

In compiling the list that follows, our goal has been to identify constructs that are "almost always wrong." The "almost always" part of this philosophy makes explicit our recognition that there are some legitimate uses for these constructs, but such uses are comparatively rare. The "wrong" part means that the construct fails to do what the programmer intends it to do, the construct violates accepted language convention, or the construct violates accepted software engineering practice. By "wrong" we do *not* mean "bad style."

Stylistic conventions vary radically from place to place, and we have no illusions that we might be able to foist our stylistic preferences on the C++ community by codifying them in the rules that follow.

Anything that is almost always wrong is occasionally right. As a practical matter, this means that any lint++ must provide programmers with some mechanism for suppressing warnings where the programmers find them to be inappropriate. The details of such a mechanism do not concern us here, but the scope of such suppression will probably need to include at least statements, files, and runs.

Candidate Errors

We propose that a lint++ should detect at least the following conditions:

1. **Failing to match constructor calls to new with destructor calls to delete.**

   A common source of memory leaks is to allocate memory in a constructor without deallocating it in the destructor. The discrepancy often arises when a new pointer member is added to a class. The pointer must typically be initialized or assigned in each class constructor as well as in the class assignment operator, and after making the appropriate changes to these member functions, it is easy (and common) to forget to update the class destructor to delete the memory referenced by the pointer.

   A more ambitious lint++ might differentiate between simple pointers and pointers to arrays, issuing a warning if a pointer to an array were not deleted using the array deletion syntax.

2. **Declaring a non-virtual destructor.**

   The following code skeleton is very common:

   ```
   Base *bp = new Derived;
   ...
   delete bp;
   ```

   Unfortunately, the destructor for class Derived is never called unless the destructor for class Base is declared virtual, a fact overlooked by almost all new C++ programmers. Fortunately, detection of this error is simple. A more sophisticated lint++ might want to issue a warning only if the class defines at least one virtual function. In fact, the justification for destructors not being virtual by default is that it is sometimes essential to create classes requiring no virtual table[ES90, p. 278].

3. **Failing to define a copy constructor or operator= for a class with dynamically allocated memory.**

   If a C(const& C) constructor (a *copy constructor*) or operator= member function is not declared, C++ provides a default implementation for them based on memberwise initialization/assignment. For pointer members, this is bitwise copy. When objects contain pointers to dynamically allocated memory, the result of a call to a default copy constructor or operator= is multiple objects pointing to the same data. Usually this is not what is intended. (When such data sharing *is* intended, it is usually accompanied by reference counting, which itself requires a user-defined copy constructor and operator= for proper behavior.)

4. **Failing to have operator= return a const reference to the class.**

   Assignments to primitive types such as int in both C and C++ return references to their left hand arguments, as do default versions of operator= generated by C++ . This convention allows for statements of the form `w = x = y = z` to work as expected for class objects. Unfortunately, many programmers are unaware of this convention; many define operator= to return void. While some may argue that the return type of operator= is a matter of style, we believe that its return type should be based on consistency with the remainder of the language. In our view, defining operator= to return other than a reference to its class is no more defendable than is defining operator+ to mean subtraction for a numerical class.

   Unfortunately, simply returning a reference to a member of the class allows one to assign a value to the *result* of an assignment:

   ```
   (a = b) = c;   // assign b to a, then assign c to a
   ```

   As Robert Murray has noted [Mur90], "using assignment statements as lvalues is extremely unobvious and error prone. To block people from doing this, you can have operator= return a const [reference] instead of a normal [reference]."

5. **Failing to check for assignment to self in operator=.**

   Given an object x, the need to cope with assignments of the form `x = x` has long been recognized[Str86, p. 179]. Yet the possibility of this type of assignment is frequently overlooked – even experts forget it, e.g., [Han90, p. 234] and [Lip89, p. 267]. Its omission can be disastrous. For classes containing dynamically allocated data, the common idiom is to delete the old data (for the object on the left hand side of the assignment), dynamically allocate new space to hold the new data, then copy the data (from the object on the right hand side of the assignment) over to the new space. When the object on the left hand side *is* the object on the right hand side, however, the result of this sequence of operations is undefined.

   Regrettably, the problem of detecting that a check for `x = x` is missing is, in the general case, undecidable. However, functions that include this check almost always start out as follows:

   ```
   const C& operator=(const C& rhs) {
     if (this == &rhs) return *this;
     ...
   ```

   As a first approximation, it would suffice for a lint++ to look for a test of this form (or the equivalent check with the arguments to == reversed).

   A complicating factor is that testing for equality between `this` and `rhs` is the *wrong way* to check for `x = x` if x contains a duplicated base class, i.e., inherits from the same class through more than one nonvirtual path. Such situations are rare, but a programmer familiar only with single inheritance or with multiple inheritance using only virtual bases might well be unaware that what suffices in those cases may fail in the case of nonvirtual multiple inheritance. A particularly sophisticated lint++, then, might issue a warning if the address-based check for `x = x` *was* present in an assignment operator for a class that was duplicated in any other object class in the

system. Readers interested in a more complete discussion of the problems involved in detecting object identity in C++ may wish to consult Adcock's article [Adc91].

6. **Problems related to the return value of a member function.**

   (a) **Returning a non-const reference to or pointer to data within an object from a const function.** As a language, C++'s meaning of "const" is "bitwise const." We firmly believe that const member functions should strengthen this contract to be "conceptually const," as well.[1] A conceptually const member function, in addition to leaving the object on which it operates unchanged (as far as the outside world can tell), must also refrain from returning "handles" through which the caller could modify the state of the object. Failure to enforce this convention renders the notion of a const object utterly useless.

   For example, consider the following skeletal class for strings:

```
class String {
  private:
    char *data;  // the characters in the string
  public:
    operator char*() const { return data; }
};
```

   Here, operator char* is bitwise const, hence the compiler will allow it to be declared as a const member function and invoked on const String objects. Yet it is clearly inappropriate to return the const object's internal pointer as a non-const, since that pointer can be used to modify the conceptual object – in this case the characters contained in the string.

   (b) **Returning a non-const reference to a data member less accessible than the function.**

   Returning such a reference essentially grants the caller direct read/write access to the underlying member. This precludes a later decision to compute instead of store a value. See the discussion under item 9 below.

7. **Overriding an inherited non-virtual function.**

   When a function f is defined in a base class B as non-virtual, it is an assertion that f's behavior is invariant over specialization of the class; B::f will *always* provide correct behavior for an object, even if that object is of a derived class D. If f is redefined in a derived class, the assertion implicit in B::f's declaration is contradicted. When this happens, something is almost always wrong. Usually the problem is either that B::f should have been declared virtual, or the writer of D::f was unaware of the fact that f was being inherited.

8. **Declaring a function that could lead to ambiguous calls.**

   One of the great pitfalls of C++ is that it is legal to overload a function name such that each function definition is valid, but calls to the function name may be ambiguous. This ambiguity can remain undiscovered until long after the functions are written, since some calls may be unambiguous. For example:

[1] Actually, we sometimes wish to deliberately violate bitwise constness (e.g., to allow caching) even as we maintain conceptual constness.

```
void f(int x, int y = 0);
void f(int x);

f(1, 3);   // fine, calls first f
f(4);      // ambiguous
```

If only calls of the first form are used, there will be no problems, but when the first call of the second form is used, the program will fail to compile. In this example, the ambiguity is brought about by the use of default parameter values, but ambiguity can also arise if a class inherits more than one function f via multiple inheritance.

9. **Declaring a data member in the public interface.**

   Declaring data members in the public interface is poor software engineering, plain and simple. It complicates the interface – users must keep track of when to use functional notation and when not to, it precludes a later decision to compute instead of store a value, and it increases coupling between classes. The use of inline member functions to get and set non-public values costs almost nothing, simplifies the public interface, reaps the benefits of functional abstraction, and decreases coupling between classes. It is difficult to envision any situation in which public data members are appropriate.

10. **Casting object pointers down the inheritance graph.**

    Casting a pointer down the inheritance graph, i.e., casting a pointer-to-Base to a pointer-to-Derived because you "know" that the pointer "really" points to a Derived object, is not an uncommon practice in C++ programs. However, it circumvents C++'s strong typing, it's brittle in the face of code changes (what you used to "know" isn't true anymore!), and it is illegal for pointers to virtual base classes. In every case such capricious casting can be replaced with safe casts through virtual functions[D'S90], and only rarely is the performance penalty serious enough to warrant concern.

11. **Passing an object by value.**

    Passing an object by value can result in a flurry of calls to constructors: for the class of the object, for its base class(es), for its members, for the members of the base classes, etc. In addition, every constructor call upon entry to the function may be matched by a destructor call upon exit. For large objects and/or objects with many base classes or members, the performance cost of these constructor/destructor calls can be staggering. They are also frequently unanticipated, especially by inexperienced programmers who are unaware of precisely what is going on when an object is passed by value, or in cases where the programmer is unaware of the true size of the object (e.g. because of typedefs). Finally, they are rarely necessary; passing a constant reference to the same object almost always suffices, with significant performance gains.

    An additional problem with passing by value is that it results in what is sometimes called "the slicing problem." Many inexperienced programmers are startled to discover that if they call a function taking a passed-by-value base class parameter p, and they pass in a derived class object, the specialized behavior normally exhibited by the derived class object is "sliced off," and p will always behave like a base class object, even when virtual functions are called on it.

12. **Having a function return a reference to an object on the stack or a dereferenced object pointer initialized by `new` within the function.**

In our experience, references are the most difficult concept for new C++ programmers to become comfortable with. Not quite objects and not quite pointers, they are frequently misused. Within a function, returning a reference to a local object results in the caller receiving a reference to an object which was destroyed when the function exited. Returning a dereferenced pointer initialized by new within the function results in the disappearance of the pointer, which often leads to a memory leak.

13. **Listing elements in a constructor's member initialization list in an order other than that in which they will actually be initialized.**

    The initialization order for subparts of an object, i.e., its base class data members and its own data members, is well defined, although it can be fairly complicated in the presence of virtual base classes. (Somewhat less well defined is the initialization order of static subparts, but that is immaterial to this discussion.) Class constructors can provide expressions to be used during the initialization of an object via a member initialization list that is part of the constructors' definitions. Frequently, programmers fail to understand that the order of initialization expressions in a constructor's member initialization list is ignored when initializing the object's subparts, and this can lead to bugs that are extremely difficult to diagnose.

Implementation Considerations

A lint++ for the errors just described would be of little use if it were unreasonably difficult to write or if it were prohibitively expensive to run. What follows are thumbnail sketches of how each of the errors in the previous section might be detected.

1. **Failing to match constructor calls to new with destructor calls to delete.**

   Parse each of the constructors for class C. Collect a list L of all data elements of class C that are initialized with or assigned the result of a call to new. Parse the destructor for class C and verify that all elements of L appear as arguments to a call to delete.

   (We are well aware of the fact that it is easy to come up with scenarios in which this approach would yield "false negatives." For example, a pointer might be initialized with the result of a call to some function f that itself returned the result of calling new. However, the simple approach we describe here will suffice in many cases, and it is easy to implement. We take a similar stance on the other lint++-able errors we describe in this paper.)

2. **Declaring a non-virtual destructor.**

   Parse the class declaration for class C. Determine whether there are any virtual functions in class C. Determine whether the destructor was defined and was declared virtual.

3. **Failing to define a copy constructor or operator= for a class with dynamically allocated memory.**

   Parse the class declaration for class C. Parse the definitions for all members of class C. Determine whether any of the member functions assigns to a data member of C the result of a call to new. Determine whether the copy constructor or assignment operation were declared.

This strategy fails to consider the possibility that friend functions can make assignments to pointer members of the class. Detecting this error in the presence of friend functions requires the parsing and analysis of several source files.

A simpler (and more conservative) algorithm would be to assume that all pointer data members may be assigned the result of a call to new, and their mere declaration would be enough to require copy constructors and assignment operators.

4. **Failing to have operator= return a const reference to the class.**

   Parse the class declaration for class C. Determine whether an assignment operator is declared. If so, determine whether the result is a const reference to C.

5. **Failing to check for assignment to self in operator=.**

   Parse the definition of operator= in class C if one exists. Check for the presence of a match with the regular expression for the code fragment presented earlier in this paper.

6. **Problems related to the return value of a member function.**

   (a) **Returning a non-const reference to or pointer to data within an object from a const function.** Parse the definition of each member function in class C. For each function returning a non-const reference or pointer to non-const object, check to see if the argument to return is a pointer to, a dereferenced pointer to, or a reference to a member of C.

   (b) **Returning a non-const reference to a data member less accessible than the function.**
   Parse the definition of each member function in class C. For each function returning a reference, check to see if the argument to return is a dereferenced pointer to or a reference to a member of C that is less accessible than is the member function.

7. **Overriding an inherited non-virtual function.**

   Parse the declaration of class C and all its base classes. For each member function declared in C, verify that it has a name distinct from all the non-virtual functions that C inherits.

8. **Declaring a function that could lead to ambiguous calls.**

   Detecting this error for member functions only can be implemented by parsing the declaration of class C and all its base classes. For each class, keep a list of all member function names, whether they are virtual, and their signatures. For functions with default parameters, generate multiple entries in the list, one for each different signature they can match. When building the list for C, add entries only for functions that do not redefine inherited virtual functions, and ensure that no entry in its list matches an entry in any of the lists built for its base classes.

   This algorithm can be extended to global functions, but it requires parsing all global functions in the program, which presumably means parsing all source files in the system. The algorithm overlooks the possibility of ambiguous function calls based on alternative sequences of type conversions to make actual parameters match formal parameters. Designing an algorithm to detect ambiguity in such cases would be considerably more complicated than what we have outlined here.

9. **Declaring a data member outside the private interface.**

   Parse the declaration of class *C*. Verify that all data members have access level private.

10. **Casting object pointers down the inheritance graph.**

    Parse the definition of each function and any previously defined classes. For each explicit cast expression, determine whether the static type of the argument to the cast is a base type of the requested type.

11. **Passing an object by value.**

    Parse the declaration of each function and any previously defined classes. Determine whether any of the arguments is an object of class type being passed by value.

12. **Having a function return a reference to an object on the stack or a dereferenced object pointer initialized by new within the function.**

    Parse the definition of each function. Determine whether the function returns a reference to an object. If so, for each return value, determine whether that value was allocated from the heap or stack by the current function.

13. **Listing elements in a constructor's member initialization list in an order other than that in which they will actually be initialized.**

    Parse the declaration of each constructor and any previously defined classes. For each member initialization list, check that the order of subparts with initialization expressions in the list is consistent with the actual order of subpart initialization.

It is useful to categorize the difficulty of these implementation strategies. We rank them based on the the minimum amount of work required by the strategy to detect a particular error:

1. Parsing of the declarations present in C++ source code.
2. Semantic analysis of the declarations present in C++ source code.
3. Parsing of the definitions present in C++ source code.
4. Semantic analysis of the definitions present in C++ source code.

Table 1 summarizes our classification of the strategies required to detect the errors described in this paper. The checkmark ($\surd$) indicates the complexity of an algorithm that will detect the corresponding error. In cases where a useful algorithm exists but is not complete (i.e., may fail to detect the error, even though it exists), a question mark (?) is used.

The fact that all of the errors described in this paper can be detected, at least some level of completeness, with no more than standard semantic analysis of source definitions encourages us that development of a lint++ would not be unduly difficult, and our experience working with and teaching the language leaves no doubt in our minds that it would be worthwhile.

The practical implications of the data in Table 1 are that errors in categories 1 and 2 could be detected by running a lint++ on C++ header files (i.e., declarations only), while errors in categories 3 and 4 would require running lint++ on implementation files.

	Complexity				Compilers
	Easier			Harder	Can Easily
Error	1	2	3	4	Detect
1				√	No
2	√				Yes
3	?			√	No
4	√				Yes
5			?		No
6a				√	Yes
6b				√	Yes
7		√			Yes
8		?			No
9		√			Yes
10				√	Yes
11		√			Yes
12				√	No
13				√	No

Table 1: Complexity of detecting different errors.

A common sentiment is that errors of this sort could (and should) be detected by current compilers with only trivial changes; we have indicated the errors that fall into this category in the last column of the table. In fact, many C++ compilers already issue warnings for some of the conditions we describe in this paper. However, a C++ compiler has a fundamentally different purpose than a lint++. A compiler is required only to detect violations of the language – its true purpose is to translate source code into object code. Programmers demand that a compiler be accurate and fast, and that the code it generates be accurate, fast, and small. A lint++, on the other hand, exists to search for constructs that are likely to have unintended consequences, and it might even presuppose that a program is syntactically correct (as verified by a compiler) when it begins its work. Such programs are typically invoked much less frequently than a compiler, and programmers are less stringent in their demands on its performance. In addition, as we discuss below, a tool like lint++ could be extended to consider issues well outside the language proper, such as verifying that design constraints expressed in some language outside C++ are satisfied within the C++ source code. Functionality such as that is well beyond the scope of traditional compilers.

A Possible Extension

An enormous number of design decisions, some of great importance, some of lesser significance, go into the development of a C++ program. Some of these decisions can be codified directly in C++, such as the fact that certain class members are not to be accessed outside the class; this is indicated by the keyword private. Other decisions can only be implied through the language, such as the fact that certain functional behavior should be invariant during class derivation; this is signified by declaring a function non-virtual (see the earlier discussion about error 7). For a great number of design decisions, however, there is no way

to express the desired semantics in C++.

For example, consider a virtual member function className that is designed to return the name of the class corresponding to the current dynamic type of an object. For this function to behave properly, it *must* be redefined whenever a new class is derived. Yet there is no way to express this constraint in C++ . As a result, the constraint may easily be violated; the design is rendered ineffectual.

One way to cope with this kind of problem is to develop a metalanguage for C++ that allows source code to be annotated with explicit design constraints. Given such constraints in the source code, it would be possible to write a tool to read both the annotations and the source code, and to issue warnings if any inconsistencies were found. A simple way to do this would be to extend a lint++ to look for design constraints expressed in the form of specially formatted comments in the C++ source code. This is the approach used by lint, although the metalanguage recognized by lint is much more primitive than what we envision for a lint++. A complementary approach is to customize the behavior of a lint++ through the use of configuration files; this is the tactic being pursued by xlint, described below.

Related Work

There is no certainly no shortage of spirited opinion as to what makes "good" or "bad" C++ code, but to the best of our knowledge, this paper is the first time that a set of specific constructs have been identified that are (1) likely to be errors, and (2) detectable by a program. However, other workers have offered general guidelines as to how good classes should be developed. Riel and Carter have proposed a minimal public interface (a set of public functions) that all classes should offer [RC90], and Packstone has suggested a set of guidelines for properly implementing overloaded operators [Pac90]. Lieberherr and his colleagues have proposed the *Law of Demeter*, a language-independent rule for achieving good style in object-oriented programs [LHR88, LH89]. This law, which in C++ imposes constraints on the behavior of member functions, is formal enough to be checked by a program (as noted by its developers). Our work has some interesting relationships to this law, such as the fact that a violation of our error 6a allows a calling function to violate the Law of Demeter without even knowing it.

xlint is a program that catches some of the errors we list in this paper, but it is really designed to enforce a particular set of corporate programming guidelines, and as such is less general that the kind of tool we propose [Gre91]. However, work is underway to extend xlint so that the kinds of conditions it looks for are specified in an external file. When this work is completed, xlint will be much more configurable.

Support for formal design constraints in the form of assertions or annotations was designed into Eiffel [Mey88], has been grafted onto Ada in the language Anna [LvHKBO87], and has been proposed for C++ in the form of A++ [CL90b, CL90a]. This work, however, has grown out of the theory of abstract data types [LG86], and has tended to limit itself to formally specifying the semantics of individual functions and/or collections of functions (e.g., how the member functions within a class relate to one another). This is important work, but has so far been too limited in scope to address the kinds of design considerations we outlined above.

Summary

In this paper we argue that there is sufficient experience with C++ to justify the development of a tool that examines C++ programs for the presence of likely programmer errors, and we describe a number of common mistakes that could be detected by such a tool. We show that such a tool would be both straightforward to develop and efficient to apply. We also discuss how such a tool could be extended to detect violations of design constraints expressed in some as-yet-to-be-developed C++ metalanguage.

Acknowledgments

John Shewchuk suggested that a lint++ should check for an ordering mismatch between member initialization lists and the actual order of object subpart initialization.

Scott Meyers received support for this research from the NSF under grant DCR 8605567; by DARPA under contract N00014-83-K-0146, ARPA order 6320; and by the Digital Equipment Corporation under agreement 393. Moises Lejter received support from AFOSR contract F49620-88-C-0132, ARPA order 6426.

References

[Adc91] Jim Adcock. Is This Identity? *The C++ Report*, 3(2), February 1991.

[CL90a] Marshall P. Cline and Doug Lea. The Behavior of C++ Classes. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 81–91, September 1990.

[CL90b] Marshall P. Cline and Doug Lea. Using Annotated C++ . In *Proceedings of C++ at Work - '90*, pages 65–71, September 1990.

[D'S90] Desmond D'Souza. Inheritance, Virtual Base Classes, and Casts. *The C++ Report*, 2(7), July/August 1990.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[Gre91] Roger Gregory. Personal Communication. January 1991.

[Han90] Tony L. Hansen. *The C++ Answer Book.* Addison Wesley, 1990.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[LH89] Karl J. Lieberherr and Ian M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, pages 38–48, September 1989.

[LHR88] K. Lieberherr, I Holland, and A. Riel. Object-Oriented Programming: An Objective Sense of Style. In Norman Meyrowitz, editor, *Proceedings of the 1988 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pages 323–334. ACM Press, 1988. Published as *ACM SIGPLAN Notices* 23:1, November 1988.

[Lip89] Stanley B. Lippman. *C++ Primer*. Addison Wesley, 1989.

[LvHKBO87] D. Luckham, F. von Henke, B. Krieg-Bruckner, and O. Owe. *Anna, A Language for Annotating Ada Programs: Reference Manual*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Mur90] Robert Murray. The C++ Puzzle. *The C++ Report*, 2(10), November/December 1990.

[Pac90] Jim Packstone. Heuristics for Error-Free Operator Overloading. *The C++ Insider*, 1(2), October 1990.

[RC90] Arthur J. Riel and John A. Carter. Towards a Minimal Public Interface for C++ Classes. *The C++ Insider*, 1(1), October 1990.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.