How to: Overload Operators

E

Just as most people want government benefits without having to pay for them, most C++ programmers want implicit type conversions without incurring any cost for temporaries.
Scott Meyers
More Effective C++, p. 105

E.1 Overloading Overview

C++ allows you to overload operators. This means, for example, that you can write expressions that are natural, such as

```
BigInteger a,b;
cout << "enter two integer values ";
cin >> a >> b;
cout << "a + b = " << (a+b) << endl;
```

Here operators `<<`, `>>` and `+` are overloaded for `BigInteger` values. Of course it's possible to run amok with operator overloading and use `+` to mean multiply just because you can. Rather than dwell on when to overload operators, this How to will explain how to overload operators. Many books show the syntax for declaring overloaded operators, but few offer guidelines for keeping the amount of code you write to a minimum and for avoiding code duplication. The guidelines in this How to do not necessarily result in the most efficient code from an execution standpoint, but development efforts are minimized while efficiency and maintainability from a coding standpoint are emphasized. Of course once you succeed in implementing overloaded operators you can then concentrate on making things efficient. To quote Donald Knuth (as cited in [McC93]):

> *Premature optimization is the root of all evil.*

E.2 Arithmetic Operators

Arithmetic operators include `+=`, `-=`, `*=`, `/=`, `%=` and their binary cousins `+`, `-`, `*`, `/`, `%`. The easiest way to implement these operators is to implement the arithmetic assignment operators as member functions, and then to implement the binary operators using the arithmetic assignment functions. The binary operators are implemented as free, non-member functions.

E.2.1 Binary Operators

Here we assume that all arithmetic assignment operators have been implemented, and discuss how to implement the binary arithmetic operators. We'll use + as an example, assuming we're implementing addition for a class `BigInt`, but the example applies to all the binary arithmetic operators for any class.

```
BigInt operator + (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns lhs + rhs
{
    BigInt copy(lhs);
    copy += rhs;
    return copy;
}
```

The code here is straightforward. A copy of the parameter `lhs` (left-hand-side) is made and the sum accumulated in this copy which is then returned. Assuming that `+=` is implemented properly it's possible to shorten the body of the function:

```
BigInt operator + (const BigInt & lhs, const BigInt & rhs)
// postcondition: returns lhs + rhs
{
    return BigInt(lhs) += rhs;
}
```

This implementation actually uses the return value of `operator +=` (see Section E.2.2) and is potentially more efficient though less clear to read at first. The efficiency gains are spelled out in some detail in [Mey96]; we'll mention them briefly later in this section.

Symmetry Is Good: Why `operator +` Is Not a Member Function. In some textbooks, `operator +` is implemented as a member function. In the example above `operator +` is a *free function*, not a member of any class. The problem with making it a member function is that it must have an object that it can be applied to. For example, consider `operator +` as a member function:

```
BigInt BigInt::operator +(const BigInt & rhs)
 // postcondition: returns (*this) + rhs
 {
     BigInt copy(*this);
     // code here to add rhs to copy, and return result
 }
```

The copy of `*this` is required since evaluating `a + b` should not result in changing the value of `a`. Note that `a + b` is the same as `a.operator +(b)` when `operator +` is a member function. The real drawback here is that the following statements are legal when `operator +` is a member function:

```
BigInt a = Factorial(50); // a large number
BigInt b = a + 1;         // one more than a large number
```

However, the following statements are **not** legal:

```
BigInt a = Factorial(50);   // a large number
BigInt b = 1 + a;           // one more than a large number
```

The expression `a + 1` compiles and executes because (we're assuming) that there is a `BigInt` constructor that will create a `BigInt` from an int, (i.e., the constructor below is implemented). We'll have more to say on constructors that act as implicit converters later.

```
BigInt::BigInt(int num);
// postcondition: *this has the value num
```

This constructor creates an *anonymous* BigInt variable for the int value 1. This anonymous variable is passed to the function `operator +`. However, the symmetric expression `1 + a` cannot be evaluated if `operator +` is a member function because the translation to `1.operator +(a)` is syntactic nonsense — 1 is a literal, it cannot have a member function applied to it nor will C++ create an anonymous variable so that a member function can be applied.

Program Tip E.1: Overloaded operators for classes should behave like operators for built-in types The binary arithmetic operators are commutative. When they're overloaded they should behave as users expect them to. So for symmetry and commutativity, binary arithmetic operators should not be member functions.

The alternative is to make `operator +` a friend function, then it has access to the private instance variables of the class for which it is overloaded. However, the approach outlined above where `operator +` is implemented in terms of `operator +=` avoids declaring friend functions. Since friend status should be granted sparingly, and since clients of a class cannot grant friendship after the class declaration is fixed, the approach outlined here should be used.

Consequences. The approach here uses a local variable that is a copy of one of the parameters. A copy is also made when the value is returned from the function. Since the function must return by-value, the copy on return cannot be avoided. Since we don't want `a + b` to have the side effect of altering the value of `a`, a copy of `a` cannot be avoided. Furthermore, compiler optimization should be able to avoid the copy in many situations, particularly if the one-line implementation of the operator shown above is used. This implementation, reproduced here

```
return BigInt(lhs) += rhs;
```

facilitates what's called the *return value optimization* [Mey96]. Smart compilers can generate efficient code so that the cost of temporaries is negligible or nothing in evaluating statements like the following:

```
x = a + b;
```

If you've benchmarked a program, determined that the line below is executed millions of times and is using temporaries and time:

```
x = a + b + c + d + e;
```

then you can recode the line using the corresponding arithmetic assignment operator:

```
x += a; x += b; x += c; x += d; x += e;
```

This code won't create any temporaries. This code should be as efficient as you can get it to be, and you have two benefits: ease of developing overloaded operators and efficiency when you need it.

E.2.2 Arithmetic Assignment Operators

Again we'll use `operator +=` for a class `BigInt` as an exemplar of the syntax and semantics for overloading arithmetic assignment operators.

```
const BigInt& BigInt::operator += (const BigInt & rhs)
// postcondition: rhs has been added to *this,
//                *this returned
```

Using this prototype the code below compiles:

```
BigInt a = Factorial(25);
BigInt b = Factorial(30);

a += b;
BigInt c = (b += b);
```

Note that `operator +=` returns a value (a constant reference) that is assigned to `c`. This isn't typical, but it's legal C++ for the built-in arithmetic operators, so it should be legal for overloaded arithmetic operators. As we saw in the implementation of `operator +`, it's possible to make good use of the return value of `operator +=`.

Program Tip E.2: Overloaded operators should have the same semantics as their built-in counterparts. This means that arithmetic assignment operators should return values. The return type must be a reference to avoid a copy, and it should be const.

Return Values from Overloaded Operators. A reference is returned since there is no reason to make a copy. A const reference is returned so that the returned value is not an lvalue, that is, so that it cannot be assigned to:

```
BigInt a = Factorial(25);
BigInt b = Factorial(30);

(a += b) = b;              // this is NOT legal C++ !!!
```

The expression `(a += b)` is not an lvalue since the value returned is const reference. The const modifier is the essential piece of preventing the return value from being an lvalue.

The expression returned from an overloaded arithmetic operator should be `*this`, the value of the object being operated on:

```
const BigInt& BigInt::operator += (const BigInt & rhs)
// postcondition: rhs has been added to *this,
//                *this returned
{
     // code here
     return *this;
}
```

Aliasing. In one of the examples above the expression `b += b` is used. In this case the parameter `rhs` will be an alias for the object on which `operator +=` is invoked. This can cause problems in some situations since the value of `rhs` may change during the computation of intermediate results (well `rhs` doesn't change, it's const, but it's an alias for *this whose instance variables may be changing as the function `operator +=` executes).

When aliasing could cause a problem this needs to be checked as a special case just as it is for overloaded assignment operators (of which the arithmetic assignment operators are a special case).

```
if (this == &rhs)  // special case
```

In some situations it may be possible to use other overloaded operators to handle the special cases. For example, the code below is from the implementation of the `BigInt` class `operator +=`.

```
if (this == &rhs)     // to add self, multiply by 2
{   *this *= 2;
    return *this;
}
```

This will not always be possible because `operator *=` will not always be overloaded for int values.

Special Cases. Sometimes, often for efficiency (but make it right before making it fast), arithmetic operators are overloaded more than once for a given class. For example, the class `BigInt` has the following overloaded member functions and free functions.

```
// member functions

const BigInt & operator *= (const BigInt &);
const BigInt & operator *= (int num);

// free functions
```

```
BigInt operator *(const BigInt & lhs, const BigInt & rhs);
BigInt operator *(const BigInt & lhs, int num);
BigInt operator *(int num, const BigInt & rhs);
```

Here it's possible to evaluate `b * 5` for a `BigInt b` variable, without converting the 5 to an anonymous variable. This may be done for efficiency or because the specialized versions of `operator +=` and `operator +` are used in implementing the nonspecialized versions. Note that for symmetry `operator +` is overloaded twice for adding `BigInt` and `int` values.

E.3 Relational Operators

Implementing the boolean relational operators <, >, <=, >=, ==, and != requires a technique similar to the method discussed in Section E.2.1 for binary arithmetic operators. This is because we want to be able to write the code below (all three comparison expressions involving <):

```
BigInt x;
// code giving x a value

if (x < y)   // do something

if (x < 128) // do something

if (1024 < x) // do something
```

For reasons similar to those outlined in Section E.2.1, the creation of anonymous variables for either left- or right-hand sides of a relational expression (e.g., involving < or ==) requires that these operators not be member functions. If they're implemented as free functions, then they'll need to be friend functions unless the approach outlined here is used.

Although relational operators can be implemented as friend functions, there is an easy method for implementing them that is similar to the method using arithmetic assignment operators such as += to implement the corresponding relational operator, in this case +, that avoids declaring any friend functions.

For example, consider a class `Date` for representing calendar dates, such as January 23, 1999. Determining if two dates are equal, or if one comes before another, can be done simply if == and < (and the other relational operators) are overloaded for `Date` objects. The approach I use is illustrated by the partial declaration of the `Date` class that follows:

```
class Date
{
  public:
    // constructors and other member functions elided
    // functions for implementing relational operators

    bool equal(const Date & rhs) const;
    bool less(const Date & rhs)  const;

  private:
    // stuff here
};
```

Here the functions `equal` and `less` determine if one Date is equal to or less than another, respectively. These functions are implemented to facilitate overloading the relational operators although these functions can be useful in debuggers. The code below shows `equal` in use.

```
Date a(1,1,1998), b(12,31, 1997);

if (a.equal(b+1)) // just checking
```

Using functions `equal` and `less` is the method in Java for comparisons, so using this approach in C++ has the added benefit of easing a transition to Java. But this method is useful on its own, especially with inheritance as we'll see later. Once the functions are implemented, implementing the overloaded relational operators is straightforward. Again, for the class `Date` we have:

Program E.1 datecomps.cpp

```
// relational operators for Date class

bool operator == (const Date & lhs, const Date & rhs)
// post: return true iff lhs == rhs
{
    return lhs.equal(rhs);
}

bool operator != (const Date & lhs, const Date & rhs)
// post: return true iff lhs != rhs
{
    return ! (lhs == rhs);
}

bool operator <  (const Date & lhs, const Date & rhs)
// post: return true iff lhs < rhs
{
    return lhs.less(rhs);
}
```

```
bool operator >  (const Date & lhs, const Date & rhs)
// post: return true iff lhs > rhs
{
    return rhs < lhs;
}

bool operator <= (const Date & lhs, const Date & rhs)
// post: return true iff lhs <= rhs
{
    return ! (lhs > rhs);
}

bool operator >= (const Date & lhs, const Date & rhs)
// post: return true iff lhs >= rhs
{
    return rhs <= lhs;
}
```

datecomps.cpp

In these examples only == and < use the member functions `equal` and `less` directly, the other overloaded operators are implemented in terms of == and <. However, it's clearly possible to use `equal` and `less` only for implementing all the overloaded operators.

When using the STL (Standard Template Library) the header file `<function>` is typically included. Templated function declarations in this file implement all relational operators in terms of < and == so typically only these operators are overloaded for classes that are used in environments in which STL is available. For example, part of the SGI implementation of the header file `function.h` is shown below:

```
template <class T>
inline bool operator!=(const T& x, const T& y)
{
   return !(x == y);
}

template <class T>
inline bool operator>(const T& x, const T& y)
{
    return y < x;
}
```

If you use STL, you typically will overload only `operator <` and `operator ==`; by including the header file `<function>`, you'll include templated functions that will implement the other relational operators in terms of < and ==. Note that these templated functions are defined as **inline** functions for efficiency. Functions defined as inline *may* be implemented without calling the function by literally substituting the code in the body of the function where the call is made, with parameters instantiated appropriately. The *inline* declaration is a request to the compiler, not a requirement.

E.4 I/O Operators

We'll look first at overloading the insertion operator, `operator <<`, for stream output. Here it's absolutely not possible to make the operator a member function of the class for which output is being defined. The statement

```
cout << x;
```

could be interpreted by the compiler as `cout.operator <<(x)` , where the insertion operator is a member function of the ostream class of which `cout` is an instance. The insertion operator could also be a free function with two parameters, much like `operator +` is as discussed above. Since programmers don't typically have access to redefining the standard I/O classes, the I/O operators are typically implemented as free functions. The header for the insertion operator for BigInt is shown below.

```
ostream& operator <<(ostream & out, const BigInt & big)
// postcondition: big inserted onto stream out
```

The return type must be a reference type because the stream on which the object is inserted is returned for subsequent insertion operations. This is what allows insertions to be chained together:

```
BigInt b = factorial(val);
string s = " factorial = ";

cout << s << b;
```

The last statement could be written more cumbersomely as follows since `operator <<` is overloaded as a free function for both string and BigInt objects.

```
operator << (operator << (cout,s), b);
```

However, it's essential that `operator <<` be an operator and not a function since the order in which arguments are evaluated in C++ is not defined. In the statement `x = min(sqrt(x),sqrt(y))`, compilers are not required to evaluate `sqrt(x)` before evaluating `sqrt(y)` (this is a C legacy, it's too bad that the order in which arguments are evaluated isn't prescribed). However, the associativity of `operator <<` is defined, it's left associative, which means that

```
cout << x << y << z;
```

requires that x be inserted before y, and that y be inserted before z.

Now that we know the prototype for the overloaded `operator <<`, how do we implement the operator? As with overloaded arithmetic operators there are two choices:

- Make `operator <<` a friend of the class whose output is being overloaded, for example, of BigInt in the examples above.
- Create a member function that can be used in implementing `operator <<` as a free, nonfriend function.

We'll adopt the second approach, since it avoids the coupling entailed by creating friend classes and the solution we'll use is easily extensible to other, nonstream output, such as on a graphics display.

E.4.1 The Function `tostring()`

One very simple way to provide output is to create a member function `tostring()` that converts an object to a string-ized form. Assuming that `tostring()` returns a string, and that `operator <<` is overloaded for strings, we can write:

```
 ostream& operator <<(ostream & out, const BigInt & big)
// postcondition: big inserted onto stream out
{
    out << big.tostring();
    return out;
}
```

Note that in the code above you cannot determine just by reading if `tostring()` returns a standard string, an apstring, a tstring, or some other type—it must return a type for which stream insertion is overloaded.

The implementation above works for any class for which a member function `tostring()` exists (this is how Java overloads + to work as a string catenator with any object, which is then used for output in Java).

Inheritance and `tostring()`. In an inheritance hierarchy, requiring all subclasses to implement `tostring` by making it (pure) virtual in superclasses makes it possible to write one overloaded `operator <<` that works with every class in the hierarchy. In Chapter 13 we explored a hierarchy of classes for implementing digital logic. Part of the abstract super class `Gate` and an appropriately overloaded operator are shown below.

```
// from gates.h
class Gate
{
  public:
    virtual ~Gate() {}
    virtual string tostring() const = 0;
    ..
};
// from gates.cpp
ostream& operator << (ostream& out, const Gate& g)
{
    out << g.tostring();
    return out;
}
```

All `Gate` subclasses: `AndGate`, `Inverter`, `OrGate`, and `CompositeGate` are "printable" since these concrete classes must supply an implementation of `tostring`

and the overloaded `operator <<` uses the polymorphic `tostring`. Clients designing new `Gate` subclasses get output for free as well.

The Function `print()`. A member function `print()` is like using `tostring()`. Typically, `print` takes a stream parameter.

```
ostream& operator <<(ostream & out, const BigInt & big)
// postcondition: big inserted onto stream out
{
    big.print(out);
    return out;
}
```

This works without using a string class, but is restricted to stream output. To write an object on a graphics screen, conversion to string is usually simpler since most graphics screens have functions to facilitate text display.

Overloading for Input. You can overload `operator >>` for input as `operator <<` is overloaded for output. It's also possible to implement an overloaded `getline` function that reads a line at-a-time rather than using white-space delimited input which is expected with `operator >>`. By far the easiest way to do input is to convert from a string. This is easy, but not always completely general since string input is required to be white space delimited. For example, if you're implementing an overloaded input operator for the `BigInt` class, what value is read by the line of text that follows?

```
1234567891234567890is a large number
```

Ideally the characters `is a number` will remain on the stream and input of the `BigInt` will stop with the zero. However, this requires reading one character at a time rather than a string at a time. You'll need to decide what method is best: converting from a string or parsing input one character at a time, based on the constraints of the problem you're solving.

E.5 Constructors and Conversions

The techniques we describe for overloading binary arithmetic and relational operators were motivated in part by concerns for symmetry. For example, we wanted to write both `a + 2` and `2 + a` when using `BigInt` variables. As we noted, symmetry in this case is made possible by a constructor that creates a `BigInt` from an `int`. Because constructors permit this kind of implicit conversion, unexpected behavior can occur when conversions happen that the programmer doesn't expect. The second statement below may be a typo, or the programmer may mean to assign to `u` a vector of one element.

```
tvector<int>  u(10); // vector of 10 elements
u = 1;               // we meant u[0] = 1
```

We know there's a vector constructor that takes an `int` argument since it's used in the first statement. This means it's possible that the second statement does two things:

- Creates an anonymous/temporary `int` vector with one element.
- Assigns/copies this temporary to `u`.

However, in the `tvector` class the second statement doesn't compile. To limit implicit conversions with the vector class, the keyword **explicit** is used with the constructor:

```
explicit tvector(int size); // size and capacity = size
```

A constructor modified by `explicit` cannot act as an implicit converter; an explicit use of the class name is required. The two statements that follow make an explicit use of the constructor:

```
tvector<int>  u(10); // vector of 10 elements
u = tvector<int>(1); // copy one-element vector to u
```

By using `explicit`, it's harder for unanticipated conversions to take place in client code—it's unlikely a programmer would type the second line above by mistake.

Bibliography

[AA85] Donald J. Albers and G.L. Alexanderson. *Mathematical People*. Birkhäuser, 1985.

[ACM87] ACM. *Turing Award Lectures: The First Twenty Years 1966–1985*. ACM Press, 1987.

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press and McGraw-Hill, 1996.

[Asp90] William Aspray. *Computing Before Computers*. Iowa State University Press, 1990.

[Aus98] Matthew H. Austern *Generic Programming and the STL*. Addison-Wesley, 1998.

[Ben86] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[Ben88] Jon Bentley. *More Programming Pearls*. Addison-Wesley, 1988.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991.

[Boo94] Grady Booch. *Object Oriented Design and Analysis with Applications*. 2nd ed. Benjamin Cummings, 1994.

[BRE71] I. Barrodale, F.D. Roberts, and B.L. Ehle. *Elementary Computer Applications in Science Engineering and Business*. John Wiley & Sons Inc., 1971.

[Coo87] Doug Cooper. *Condensed Pascal*. W.W. Norton, 1987.

[Dij82] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[DR90] Nachum Dershowitz and Edward M. Reingold. Calendrical calculations. *Software-Practice and Experience*, 20(9):899–928, September 1990.

[(ed91] Allen B. Tucker (ed.). *Computing Curricula 1991 Report of the ACM/IEEE-CS Joint Curriculum Task Force*. ACM Press, 1991.

[EL94] Susan Epstein and Joanne Luciano, editors. *Grace Hopper Celebration of Women in Computing*. Computing Research Association, 1994. HopperBook@cra.org.

[Emm93] Michele Emmer, editor. *The Visual Mind: Art and Mathematics*. MIT Press, 1993.

[GÖ95] Denise W. Gürer. Pioneering women in computer science. *Communications of the ACM*, 38(1):45–54, January 1995.

[Gar95] Simson Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, 1995.

[GHJ95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Programming* Addison-Wesley, 1995

[Gol93] Herman H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, 1993.

[Gri74] David Gries. On structured programming - a reply to smoliar. *Communications of the ACM*, 17(11):655–657, 1974.

[GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1993.

[Har92] David Harel. *Algorithmics, The Spirit of Computing*. Addison-Wesley, second edition, 1992.

[Hoa89] C.A.R. Hoare. *Essays in Computing Science*. Prentice-Hall, 1989. (editor) C.B. Jones.

[Hod83] Andrew Hodges. *Alan Turing: The Enigma*. Simon & Schuster, 1983.

[Hor92] John Horgan. Claude e. shannon. *IEEE Spectrum*, April 1992.

[JW89] William Strunk Jr. and E.B. White. *The Elements of Style*. MacMillan Publishing Co., third edition, 1989.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 1 Fundamental Algorithms. Addison-Wesley, third edition, 1997.

[Knu98a] Donald E. Knuth. *The Art of Computer Programming*, volume 2 Seminumerical Algorithms. Addison-Wesley, third edition, 1998.

[Knu98b] Donald E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, third edition 1998.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[KR96] Samuel N. Kamin and Edward M. Reingold. *Programming with class: A C++ Introduction to Computer Science*. McGraw-Hill, 1996.

[Mac92] Norman Macrae. *John von Neumann*. Pantheon Books, 1992.

[McC79] Pamela McCorduck. *Machines Who Think*. W.H. Freeman and Company, 1979.

[McC93] Steve McConnell. *Code Complete*. Microsoft Press, 1993.

[MGRS91] Albert R. Meyer, John V. Gutag, Ronald L. Rivest, and Peter Szolovits, editors. *Research Directions in Computer Science: An MIT Perspective*. MIT Press, 1991.

[Neu95] Peter G. Neumann. *Computer Related Risks*. Addison Wesley, 1995.

[Pat96] Richard E. Pattis. *Get A-Life: Advice for the Beginning Object-Oriented Programmer*. Turing TarPit Press, 1999.

[Per87] Alan Perlis. The synthesis of algorithmic systems. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[RDC93] Edward M. Reingold, Nachum Dershowitz, and Stewart M. Clamen. Calendrical calculations, ii: Three historical calendars. *Software-Practice and Experience*, 23(4):383–404, April 1993.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[Rob95] Eric S. Roberts. Loop exits and structured programming: Reopening the debate. In *Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, pages 268–272. ACM Press, March 1995. SIGCSE Bulletin V. 27 N 1.

[Rob95] Eric S. Roberts. *The Art and Science of C*. Addison-Wesley, 1995.

[Sla87] Robert Slater. *Portraits in Silicon*. MIT Press, 1987.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1987.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[Mey92] Scott Meyers. *Effective C++*. Addison Wesley, 1992.

[Mey96] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.

[Wei94] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin Cummings, 1994.

[Wil56] M.V. Wilkes. *Automatic Digital Computers*. John Wiley & Sons, Inc., 1956.

[Wil87] Maurice V. Wilkes. Computers then and now. In *ACM Turing Award Lectures: The First Twenty Years*, pages 197–205. ACM Press, 1987.

[Wil95] Maurice V. Wilkes. *Computing Perspectives*. Morgan Kaufmann, 1995.

[Wir87] Niklaus Wirth. From programming language design to compiler construction. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.