

# Extensible Literals

Ian McIntosh ([ianm@ca.ibm.com](mailto:ianm@ca.ibm.com))  
Michael Wong ([michaelw@ca.ibm.com](mailto:michaelw@ca.ibm.com))  
Raymond Mak ([rmak@ca.ibm.com](mailto:rmak@ca.ibm.com))

Document number: N1892=05-0152

Date: 2005-09-30

Project: Programming Language C++, Evolution Working Group

Reply-to: Michael Wong ([michaelw@ca.ibm.com](mailto:michaelw@ca.ibm.com))

Revision: 2

## Abstract

This paper proposes additional forms of literals using modified constructor syntax and semantics to provide extensible user-defined literals. Extensible literals allow user-defined classes to provide new literal syntaxes and / or data representations, capabilities previously available only for basic types. It increases compatibility with C99 and with future C enhancements.

An extensible literal constructor defines the literal syntax it accepts, and translates that to the data representation.

The proposal requires a fairly simple change to the core language.

## The Problem

C++ recognizes literals for its basic data types. For those, the syntax of the literal (e.g., the presence of a decimal point, exponent or alphabetic suffix) identifies the type. The magnitude further specializes the exact type. The type and implementation determine the data representation.

To add a new data type to a non-extensible language such as C [C99], the language syntax and semantics must be modified by adding the type's name, the type's operations, and where appropriate the type's literal syntax.

To add a new data type to an extensible language such as C++ [C++03], the preferred approach is to leave the language unchanged and define a new class implementing the type's operations. Defining new literal syntax can be a problem, and providing compatibility with C is a problem.

This paper draws on two basic principles of C++ design:

- User-defined types should have all the same support facilities as built-in types, and currently that facility is not extended to user-defined literals [12]

- C compatibility should be maintained as far as possible and currently, we cannot accept certain C literals [9,10,11,16]

The existing mechanisms work well when existing literals (integers, floating-point and string literals) are suitable. Other proposals [1,2] would extend that to classes which are aggregates of existing types by adding user-defined literals formed by grouping basic data type literals; e.g., **complex(1,2)**. This proposal allows additional forms of non-standard literals and uses modified constructor syntax and semantics to provide extensible user-defined literals.

C++ already has extensible data types using classes and templates and overloaded operations on them. What it also needs is extensible literals to match the robustness that extensible types deliver.

## The Solution: The Basic Idea

The basic idea is that when the compiler recognizes that a token is a literal but not a valid literal in phase 7 [**lex.phases**] of translation for a basic data type, it checks whether it matches the syntax accepted by any declared *extensible literal constructor*. If it does, the literal's characters in the source program are used to construct a string literal parameter and passed to the matching extensible literal constructor; e.g., **12.34d** would become parameter "**12.34d**" and passed to the constructor for literals ending with "**d**".

An extensible literal constructor contains one or more statements to be executed to parse the literal string parameter and construct the desired object. In general the constructor body would be executed just before **main ( )** is, so extensible literals would not be compile time constant expressions (although a compiler might be able to execute some at compile time).

## Goals

The main goal for literal suffixes is to handle every suffix currently in or proposed for C. A second goal is to handle every suffix in common extensions to C. A third goal is to handle every string literal and character literal prefix.

The goal for data representation is to be able to produce data for every existing, proposed or future numeric **or string** data format, including integer, binary floating-point and decimal floating-point, in any reasonable size or precision and representation.

## Extensible Literal Syntax

An extensible literal is either an *extensible numeric literal* or an *extensible string literal*.

An extensible numeric literal must start with a digit and may contain any characters that would be allowed in an integer or floating-point literal, followed by an alphanumeric *extensible literal suffix* ("**d**" in the example above) accepted by some extensible literal constructor; for example:

**1234d**

**12.34df**

**12.34e5dd**

**12.34e-10dq**

**1\_I**

**123456789012345678901234567890123456789012345678901234567890verylong**

**12.34.56p**

The length may exceed the maximum for basic types, and the character sequence need not match their syntax.

An extensible string literal consists of a prefix (the *extensible literal prefix*) followed by a quoted character string; e.g.:

**utf16"abc"**

## Extensible Literal Pattern Syntax

An extensible literal constructor needs to specify the *extensible literal pattern* for the extensible literals it accepts so that matching of the literal can occur. Several options in the constructor syntax could be considered. (Note: In the following, the character sequence in bold is to be added as additional syntax to the constructor. The exact syntax of these character sequences within a constructor declaration will be discussed later.)

1. Specify just the single suffix or prefix string; e.g.:

**"df"**

**"DQ"**

**"utf32"**

Often that would require writing two otherwise identical constructors.

2. Specify a list of synonymous strings; e.g.:

**"df", "DF"**

That allows one constructor to handle multiple suffixes or prefixes, but complicates the syntax.

Neither of these lets the compiler do any syntax checking for the constructor.

3. Specify a basic typename and the suffix or prefix string(s); e.g.:

**double "dd", "DD"**

**int "long128"**

For extensible numeric literals, this allows the compiler to check that the syntax matches the specified type except for the suffix, number of digits and exponent range.

For numeric literals the typename describes the syntax not the size.

Typenames **int** and **double** accept any literal in integer or floating-point syntax with only the specified suffix(es). Typename **unsigned long** accepts any integer literal with a **u** or **U** suffix followed by the specified suffix(es), and **float** accepts a floating-point literal with an **f** or **F** suffix then the specified one(s).

For extensible string and character literals it allows the base character type to be specified; e.g.:

**w\_char "utf\_16"**

4. Instead of a type followed by a quoted string, specify a *literal keyword*. This literal keyword would identify the literals in the C Standard that are known to

be missing from the C++ standard. This is somewhat less robust but is easier to describe. For example we can use the keyword `FLOATING_LITERAL` to signify the character sequence before the suffix to be a floating literal, and then write the suffix character sequence after it. (Also, we can omit the quotes in these syntaxes.) e.g.:

**FLOATING\_LITERAL\_i**

5. Specify a *regular expression* describing the type; e.g.:

**"[0-9]{1-28}long128"**

That allows much better checking. The extra programming effort is small for the benefit, especially if a sample floating-point regular expression is available.

This would also allow patterns to match literals like

**1234d5**

by accepting a numeric string, **"d"**, and another numeric string.

6. Some combination of those. There are good reasons to allow both regular expressions and suffix / prefix strings with optional type names.

The exact syntax chosen is open for suggestion and is dependent on how much compile-time error checking is desired.

In many cases the constructor must be able to report other errors such as a value that is too large or an exponent that is out of range. The Extensible Literal Errors section below deals with that.

## Extensible Literal Constructor Syntax

An extensible literal constructor constructs an object of its class, so its syntax should resemble other constructors. It must differ, though, because in addition to the string parameter by which the compiler provides the specific literal, it needs to specify the *extensible literal pattern* it matches.

An extensible literal constructor would be declared similarly to a standard constructor with a **const char\*** or **const w\_char\*** parameter which contains the literal string, and the extensible literal pattern parameter(s). Unless the pattern is specified as a regular expression, the parameter order indicates whether a prefix or suffix is involved; e.g.:

```
class DecimalFloat {
public:
    DecimalFloat (const char* literalString, double "df", "DF") { . . . }
}
// The above specifies an extensible numeric literal.
// literalString passes the character sequence of the literal in the source code.
// double specifies the type of the literal formed by the character sequence if
// the suffix were ignored. This information permits the compiler to
// do some verifications.
// "df", "DF" is the list of recognized suffixes.
```

and e.g.:

```

class Unicode {
public:
    Unicode (w_char "unicode", const char* literalString) { . . . }
}
// The above specifies an extensible string literal.

```

Writing extensible literal constructors is simplified by having the literal pattern prefix or suffix characters removed before the literal string is passed to the constructor; e.g.:

```

class Imaginary {
    double imaginary_value;
public:
    Imaginary (const char* literalString, double "j"):
        imaginary (atod (literalString)) { }
    . . . }

```

```

// The literal 12.34j would invoke the constructor with the parameter literalString
// passing the string "12.34", without the j suffix.

```

## Extensible Literal Errors

The compiler reports an error if a standard literal is invalid. For extensible literals that is not always possible. If the literal pattern includes a type name as in Option 1, 2 and 3 above on the section *Extensible Literal Pattern Syntax*, then the compiler can detect deviations from that type's literal syntax. These three options form the most visually understandable syntax. On the other hand, a type name may create other issues. For example, if in a constructor declaration we have:

```
float "j"
```

the constructor would recognize the literal **12.34fj**. **12.34f** is the sequence of characters forming a **float** type floating-point literal. "**j**" is the suffix, and the string passed to the constructor is "12.34f". An alternative would be to treat **float** as if it were **double**, and accept **12.34fj** only if the literal pattern suffix was "**fj**".

We can avoid using the type name by using a literal keyword, as described in Option 4 above. We can also use a regular expression as in Option 5. A literal deviating from the specification would mean the match fails for that particular constructor; and if the literal did not match any extensible literal pattern, the program is not well-formed.

In any case the constructor may find other errors. A way to report errors (e.g., an exponent out of range) is needed. Since literal constructors execute before **main** (), they are always executed so the detection of an error would require a single execution test and would not depend in any way on input data.

The proposed solution is for the constructor to call a function which would write the message (including the literal string, source filename and line number) to standard error. That implies that iostreams or some other output mechanism must be initialized before extensible literals.

## Performance

Like static initialization, some extensible literals can be handled at compile time but some would be processed just before **main** ( ).

The number of extensible literals will be small, the performance difference should not be significant, and existing alternatives would also require execution time conversions, but it would be preferable to completely handle all literals at compile time instead of executing extensible literal constructors at run time.

In fact some extensible literal constructors can be handled at compile time, using the proposed Generalized Constant Expressions. The `_Imaginary` example above requires only that the compiler consider **atof** ( ) with a constant parameter as a constant expression and that it interprets it to do a conversion it already knows how to do.

## Restrictions

Because literal constructors will not necessarily be executed at compile time, the literals they construct are not constant expressions and cannot be put into ROMs to protect them from modification or for use in embedded systems.

The author of a class can and should write appropriate `<<` and `>>` iostream operators for it, but like any other new class there are restrictions on using **printf** ( ) and **scanf** ( ). A type like **Imaginary** can be cast to floating-point and **printfed** with `"%fj"`, but types like **\_Decimal32** have new internal representations so could only be **printfed** by first converting to a string.

## Usage Examples

We will use the syntax in Option 3 above to demonstrate some use cases.

### *Imaginary and Complex*

C99 [C99] added the **\_Imaginary** and **\_Complex** groups of types. There are no actual **\_Imaginary** literals – they are formed by multiplying **I** by a float, double or long double literal. The complex value with real and imaginary parts 2 and 3 respectively is written as **2. + I\*3.** (**I** is the value of the imaginary unit). But Imaginary literals could be supported. With suffixes **fj**, **j** and **lj** (or their upper case variants), the constructors

```
Float_Imaginary (const char* literalString, float "j", "J")
    : imaginary_value (atof (literalString)) { }
Double_Imaginary (const char* literalString, double "j", "J")
    : imaginary_value (atod (literalString)) { }
LongDouble_Imaginary (const char* literalString, long double "j", "J")
    : imaginary_value (atold (literalString)) { }
```

would recognize literals like

```
1.j
12.34fj
3e-9LJ
```

With the compiler removing the suffix, it just needs to call **atof** / **atod** / **atold** ( ) and initialize the data member.

Once **\_Imaginary** types are available, the **\_Complex** types can be built on them.

### *Decimal Floating-Point*

A C proposal [7, 8] would add the `_Decimal32`, `_Decimal64` and `_Decimal128` decimal floating-point types, and a C++ proposal [5] would add corresponding classes. The C literals would resemble float, double and long double ones, but with different suffixes, somewhat different operations, different functions, and significantly different representations (although the representation is not finalized yet). The constructors

```
_Decimal32 (const char* literalString, double "df", "DF") { . . . }  
_Decimal64 (const char* literalString, double "dd", "DD") { . . . }  
_Decimal128 (const char* literalString, double "dq", "DQ") { . . . }
```

would recognize these.

Note: The literal patterns all specify double instead of **float**, **double** and **long double** to avoid recognizing the standard **float** "f" and **long double** "ld" suffixes prior to the decimal floating-point ones.

### *Alternate Floating-Point Formats*

Today most computers implement IEEE floating-point, but some [15, 17] have alternate formats or both IEEE and others. IEEE allows either single precision or double precision to be extended. C99 "Future language directions" suggests longer floating-point types. The constructors

```
FloatExtended (const char* literalString, double "fe") { . . . }  
CrayDouble (const char* literalString, double "cray") { . . . }  
IBMHFloat (const char* literalString, float "hex") { . . . }  
LongLongDouble (const char* literalString, double "lld") { . . . }
```

would recognize literals like

```
12345.67fe  
1234567890.1234567890e6cray  
12345.6e20hex (Note the "f" because the pattern type is float.)  
12345678901234567890.12345678901234567890e-3lld
```

### *Longer Integers*

Some application programs need larger integer types. The IEEE 754R [13] draft standard requires that every implementation provide an integer type (not necessarily built in) the same size as the largest floating-point type supported, so supporting quad precision floating-point requires supporting 128-bit integers. The constructors

```
int128 (const char* literalString, int "lll", "LLL") { . . . }  
uint128 (const char* literalString, unsigned int "ulll", "ULLL") { . . . }
```

would recognize literals like

```
0lll  
123LLL  
100ulll
```

C++ does not yet include C99's **long long int** and **unsigned long long int** until the C++0X proposal on Long Long int [6] is accepted; however, even that proposal does not allow scalability with larger integer literals which may come in the future. Numeric literal constructors can allow arbitrarily larger literal types with a growth path for the future.

### *Scaled Fixed-Point*

Some languages include scaled fixed-point types. One example is IBM's C/370 [14] decimal types. These are from 1 to 31 digits, with an optional decimal point at either end or any position between digits, and the suffix **d**; e.g., **1234d**, **12.3456d**, or **.07d**. These are stored as scaled integers, with the internal representation being packed decimal [17]. Other languages have similar types but with other representations including various forms of zoned decimal or binary integer.

If this were to be handled, distinguishing the many (about a thousand) different types would need regular expression literal patterns, with for example

```
"[0-9]{5}.[0-9]{2}d"
```

matching 7 digit numbers with 5 before the decimal point and 2 after, using the constructor

```
Decimal 7 2 (const char* literalString, "[0-9]{5}.[0-9]{2}d") { ... }
```

### *Alternate String and Character Literals*

Alternate character sets or representations can use extensible string literal constructors. The parameter order is reversed, to indicate the use of representation name prefixes instead of suffixes (that would not be necessary if patterns were always described using regular expressions). Prefixes imply that the literal is a quoted string.

The compiler needs to know how big the resulting literal string will be, or more precisely the maximum size it could be, so that space can be reserved for it. By default that would be the same size as the input literal string (including the terminator). If the literal pattern includes a type, the maximum size would be multiplied by the size of the base type of the type specified. Using a base type `wchar_t` would not imply that the prefix would automatically include "L". The constructors

```
Italian ("Italian", "italian", const char* literalString) { ... }
```

```
EBCDIC ("EBCDIC", "Ebcddic", "ebcddic", const char* literalString) { ... }
```

```
Unicode (wchar_t "Unicode", "unicode", "U", "u",  
const wchar_t* literalString) { ... }
```

would recognize literals like

```
Italian"caio"
```

```
ebcddic"abc"
```

```
u"universal"
```

The constructor could of course define its own \ or other escape sequences; e.g., an ASCII class could recognize escape sequences for the ASCII control characters with literals containing things like **{HT}**, **{FF}**, or **{ESC}**.

The same approach should apply to character literals, using ' instead of ". The type specified as part of the literal pattern is used to distinguish literal constructors accepting a **single** character or a string. Types like **char** or **wchar\_t** or **int** would mean a character literal, while **char\*** or **wchar\_t\*** or **int\*** would mean a string literal. Another possibility is that a regular expression pattern should make it obvious, based on it containing two single quotes or two double quotes

## **Demonstration of a Simple Implementation**



We now examine a possible way to implement this. For our purpose here we avoid adding new syntax in the constructor as proposed above. New syntax needs more careful study. We are interested in finding out the basic steps required to translate an extensible literal, and the part of the standard that is affected. A more elaborate syntax can be added later and bridge on essentially the same mechanism, but with more comprehensive handling of searches for the correct literal constructor.

Preprocessing tokens are formed in translation phase 3. This includes *pp-number* and *string-literal* (2.4). After macro expansions, character set mapping, and string concatenation, we reach phase 7 where preprocessing tokens are converted into tokens and meanings attached. The basic idea is to insert a mechanism at the end of phase 7 of translation, where preprocessing tokens are converted to literals.

White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token (2.6). The resulting tokens are syntactically and semantically analyzed and translated.

[*Note:* Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation.]

When a *pp-number* cannot be converted, and before it is flagged as invalid, we give it a second chance to see if it is an extensible literal. If a *pp-number suffix* sequence cannot be converted into a token, we replace it with the macro invocation `__EXTENSIBLE_NUMERIC_LITERAL_suffix ("pp-number")` and repeat phase 4.

The macro is an easy way to handle the mapping of suffixes to constructors. In the full solution, this would be replaced by a searching algorithm which determines the constructor to invoke. The full solution can provide better checking and diagnostic capability.

The program then defines the corresponding macro as follows:

```
#define __EXTENSIBLE_NUMERIC_LITERAL_DF(t) DecimalFloat (#t)
```

which maps the suffix to the class.

We can treat suffixes for base types as if the compiler has predefined macros for them:

```
#define __EXTENSIBLE_NUMERIC_LITERAL_L(t) ...
```

For classes which are known by the implementation (as the `dfp` classes could be), the compiler can either intercept the macro and do its own magic, or intercept the constructor and do special optimization. The user can override the macro, expanding to something other than a constructor. We can also use an `__extensible_literal` class as described below to provide a way for the compiler to recognize the constructor. We may be able to further restrict the body of such constructors to be empty.

Extensible string and character literals can be handled similarly.

*Changes required for this simple implementation*

As an illustration, we provide the suggested changes below to the C++ Standard.

Add after 2.13.3:

### 2.13.3a Extensible literals

***extensible-literal*** : *extensible-numeric-literal*  
*extensible-string-literal*  
*extensible-character-literal*

***extensible-numeric-literal*** : *pp-number-first-part* *suffix*

***extensible-string-literal*** : *prefix* *string-literal*

***extensible-character-literal*** : *prefix* *character-literal*

An *extensible numeric literal* is formed from a preprocessing number. It is neither an integer literal nor a floating literal. It is a preprocessing number consisting of two parts: the first part shall be the longest sequence of characters which have the lexical form of an integer literal or a floating literal; the second part shall be a non-empty sequence of characters called the *suffix*.

If the first part begins with the sequence **0x** or **0X**, the suffix shall be the sequence of characters satisfying the following:

***suffix*** : *nonhexdigit*  
*suffix nondigit*  
*suffix digit*

***nonhexdigit*** : { *nondigit* excluding *a-f, A-F* }

Note: *nondigit* is defined in 2.10.

If the first part does not begin with the sequence **0x** or **0X**, the suffix shall be the sequence of characters satisfying the following:

***suffix*** : *nondigit*  
*suffix nondigit*  
*suffix digit*

During translation phase 7 when preprocessing numbers are converted to tokens, if an extensible numeric literal is encountered, it is replaced with the following macro invocation:

`__EXTENSIBLE_NUMERIC_LITERAL_suffix ( pp-number-first-part )`

and then translation phase 4 is repeated. The program is not well-formed if the function like macro `__EXTENSIBLE_NUMERIC_LITERAL_suffix` is not defined at this point of the translation.

An *extensible-string-literal* consists of two parts: the first part is a prefix and the second a string-literal. The prefix shall be the longest non-empty sequence of characters so that the second part is still a valid string-literal.

An *extensible-character-literal* consists of two parts: the first part is the prefix and the second a character-literal. The prefix shall be the longest non-empty sequence of characters so that the second part is still a valid character-literal.

During translation phase 7 when an extensible string literal or an extensible character literal is encountered, it is replaced with the following macro invocations respectively:

```
__EXTENSIBLE_STRING_LITERAL_prefix ( string-literal )
```

```
__EXTENSIBLE_CHARACTER_LITERAL_prefix ( character-literal )
```

and then translation phase 4 is repeated. The program is not well-formed if the function-like macro

```
__EXTENSIBLE_STRING_LITERAL_prefix
```

or

```
__EXTENSIBLE_CHARACTER_LITERAL_prefix,
```

respectively, is not defined at this point of the translation.

### ***Recommended practice***

If the extensible numeric literal has the type of class T, it is recommended that T provides a constructor with the following prototype:

```
class __extensible_literal;
```

```
T ( const char *, __extensible_literal );
```

where `__extensible_literal` is a class defined by the implementation. It is used in a constructor prototype to signify an extensible literal constructor. It need not be used in the constructor. The macro is defined by:

```
#define __EXTENSIBLE_NUMERIC_LITERAL_suffix( t ) T( #t, __ext_lit )
```

where `__ext_lit` is a global object of type `__extensible_literal` defined by the implementation. The class `__extensible_literal` and the object `__ext_lit` can be defined in an implementation-provided header. The class `__extensible_literal` can also encapsulate machinery to output diagnostics. It can be done through the standard streams or some other means in the case of embedded systems.

A similar recommendation applies to extensible string and character literals. The same `__extensible_literal` class can be used in all three cases.

**End 2.13.3a**

Instead of using macros, we can use class definitions directly. The specification of suffixes can be done through special class names defined under a standard namespace. For example, the literal suffix DF for the DecimalFloat class can be specified as follows:

```
namespace std_extensible_literal {
    class DF_t;
}

class DecimalFloat {
public:
    // The DF_t* parameter allows the presence of another constructor
    // which takes a const char* parameter for other purpose.
    // DF_t* is not actually used by the constructor.
    DecimalFloat( const char*, std_extensible_literal::DF_t*) { /* ... */ };

    // ...
};

namespace std_extensible_literal {
    class DF_t { // maps suffix DF to class DecimalFloat
    public:
        DecimalFloat make_literal(const char* s)
            { return DecimalFloat(s, (std_extensible_literal::DF_t*)0); }
        } DF;
    }
}
```

The above definitions are to be provided by the implementer of the DecimalFloat class. The compiler follows the similar processing steps as described in 2.13.3a, but instead of replacing an extensible literal with a macro, the compiler replaces it with the following sequence of tokens (suppose the literal is 12.34DF):

```
std_extensible_literal::DF.make_literal("12.34")
```

The implementation for the complete solution would require additional lookup searches relating to selecting the right constructor, and there would be new syntax in the constructor. In this section of the paper we have demonstrated in principle how a simple implementation could be done with changes only in translation phase 7 when tokens are formed. The changes do not affect existing valid programs.

## Related papers

This is compatible with and orthogonal to other proposals including literals for user-defined types [1], generalized initializer lists [2], and braces initialization overloading [4], and would be improved by the generalized constant expressions proposal [3].

These other papers primarily propose grouping literals using existing known literals. [1] in particular identifies the possibility of a unique syntax using the **literal** keyword as a constructor, and limits what can be placed inside the constructor so that it can achieve ROMability.

## References

### C++

- [1] Bjarne Stroustrup. Literals for user-defined types. N1511
- [2] Bjarne Stroustrup and Gabriel Dos Reis. Generalized Initializer Lists. N1509
- [3] Gabriel Dos Reis. Generalized Constant Expressions. N1521
- [4] Daniel Gutson. Braces Initialization Overloading. N1493
- [5] Robert Klarer. Decimal Types for C++. N1839
- [6] J. Stephen Adamczyk Adding the long long type to C++ (Revision 3). N1811
- [C++03] ISO/IEC 14882:2003(E), *Programming Language C++*.

### C

- [7] Extension for the programming language C to support decimal floating-point arithmetic. N1137
- [8] The type and representation of unsuffixed floating constant. N1108
- [C99] ISO/IEC 9899:1999(E), *Programming Language C*.

### Other

- [9] Herb Sutter. The New C++: C and C++: Wedding Bells? Oct 2002, C++ User's Journal.
- [10] Bjarne Stroustrup. C and C++: Siblings. July 2002, C++ User's journal.
- [11] Bjarne Stroustrup. C and C++: A Case for Compatibility. Aug 2002, C++ User's Journal.
- [12] Bjarne Stroustrup. The Design and Evolution of C++. Addison-Wesley. 1994.
- [13] IEEE 754R - Draft Standard for Floating Point Arithmetic P754/D0.14.2.
- [14] IBM C/370 Language Reference Manual.
- [15] Cray architecture Manual.
- [16] Bjarne Stroustrup. *Sibling Rivalry: C and C++*. (AT&T Labs — Research Technical Report TD-54MQZY, January 2002), <[www.research.att.com/~bs/sibling\\_rivalry.pdf](http://www.research.att.com/~bs/sibling_rivalry.pdf)>.
- [17] IBM z/Architecture Principles of Operation, <[publibz.boulder.ibm.com/cgi-bin/bookmgr\\_OS390/BOOKS/DZ9ZR003/CCONTENTS?SHELF=DZ9ZBK03&DN=S A22-7832-03&DT=20040504121320](http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DZ9ZR003/CCONTENTS?SHELF=DZ9ZBK03&DN=S A22-7832-03&DT=20040504121320)>