N1511=03-0094
September 18, 2003

Literals for user-defined types

Bjarne Stroustrup

(bs@research.att.com)

Abstract

This note proposes a notion of user-defined literals based on literal constructors without requiring new syntax. If combined with the separate proposal for generalized initializer lists, it becomes a generalization of the C99 notion of compound literals.

Basically, a constructor defines a user-defined literal if it is inline and specifies a simple mapping of its arguments to object representation values and is invoked with constant expressions or objects that can be trivially copied (such as pointers).

The Problem

C++ does not provide a way of defining literals for user-defined types. Instead, constructors are used. For example:

```
15              // int literal
"15"            // string literal (zero terminated array of characters)
complex(15)     // "sort of complex literal"
```

When a constructor is simple and inlining is done well, such constructor calls provide a reasonable substitute for literals. However, a constructor is a very general construct and there have been many requests for a way to express literals for user-defined types in such a way that a programmer can be confident that a value will be constructed at compile time and potentially stored in ROM. For example:

```
complex z(1,2);          // the variable z can be constructed at compile time
const complex cz(1,2);   // the const cz can potentially be put in ROM
```

A Solution

The most direct and obvious solution would be to introduce syntax to distinguish a literal constructor and to distinguish literals of user-defined types. For example:

```
class X {
        int x,y,z;
public:
        literal X(int a, int b) :x(a+1),y(0),z(b) {}     // literal constructor
        // ...
```

```
};

X"1,2"          // a literal of type X
```

This syntax is just for the illustration of the idea; a better syntax is suggested below.

This "literal constructor" illustrates the requirements for any specification of a literal for a user-defined type. It specifies a (simple) mapping from a set of arguments to the representation of the type. Often, that will simply specify a value for each member of the type's representation, but slight generalizations are possible and sometimes useful. Here, I have indicated that the member **y**'s value need not be specified by the user and that a slight transformation takes place on the argument used to specify a (**x** becomes **a+1**).

The body is empty. Since the construction of the representation of a value takes place at compile time, very few constructs could reasonably be allowed in a literal constructor body. The simplest rule would be to require that body to be empty. That is, the mapping of arguments to representation (member values) must be specified as member initializers. In addition, a literal constructor must be inline.

What can be accepted as an argument type? An argument must of a type that can be copied without the use of a nontrivial copy constructor (e.g. **int**s, pointers, and references). What can be accepted as an initializer? An initializer can be another argument, a value of a type that can be copied without the use of a non-trivial copy constructor, or a constant expression.

Note that this definition is recursive in that it allows the use of literals of user defined types as arguments to be used. For example:

```
class Y {
        complex x, y;
        literal X(complex a, int b) : x(a), y(complex"a,0") {}
        // ...
};

const int c = 3;
Y"complex"1,2",c";
```

This simple definition could be elaborated. For example, should we accept floating point expressions, such as **d+1.7** where **d** is an argument of floating point type? I think not. Even if **d** is a literal so that the expression to be evaluated is something like **2.3+1.7**, I suspect that the complication of requiring floating point arithmetic at compile time is not worth the bother – especially for cross compilers.

Syntax for user-defined literals

The syntax used to illustrate the idea of a "literal constructor" above has some obvious problems. Consider that last use:

```
const int c = 3;
Y"complex"1,2",c";
```

That use of quotes (chosen to emphasize the literal nature of the construct) would clearly confuse any traditional lexer (and many human readers). Also, it doesn't exactly extend to string literal arguments. Furthermore, it would not be easy to get used to the idea that elements of the string-like part are separate values and that variable can occur there. I think that a much more natural (i.e. familiar) and readable notation would use parentheses:

```
const int c = 3;
Y(complex(1,2),c);
```

After all, parentheses are the way we usually express arguments. However, by doing so, we lost the syntactic distinction of the literal. That is,

```
Y(complex(1,2),c)
```

is a literal because it is expressed in terms of the literal constructors of Y and complex. Does this lack of distinction matter? Consider:

```
int i;
const int c = 3;
Y(complex(1,2),i);     // can't be a literal
Y(complex(1,2),c);     // is a literal
```

We have three alternatives:

(1) The result of the constructor call is a literal if its arguments are acceptable to a literal constructor (and then evaluate the constructor at compile time)
(2) Make it an error to call a literal constructor with unacceptable arguments
(3) Syntactically distinguish a user-defined literal (as before)

I think that (2) is unacceptable because the kind of arguments from which we'd want to construct literals will always be very similar to the kind of arguments we'd want to construct non-literal values from. Often, the distinction would be only **const** vs. non-**const** arguments.

Personally, I prefer (1): basically, a value is a literal if it is composed out of literals and implemented by a literal constructor. The problem with that is that some people will not trust compilers to do proper resolution, placement in ROM, placement in text segment, etc. Choosing that solution would require text in the standard to constrain and/or guide implementations.

So consider (3): What syntactic alternatives do we have? I conjecture that the notation must be somewhat similar to constructor calls, be terse to emphasize the simple semantics of the construct, and not offend simple tools. For example:

```
Y"complex"1,2",c"          // offends lexers, doesn't indicate argument passing
Y"(complex(1,2),c)         // offends lexers
Y'(complex(1,2),c)         // offends lexers
Y'(complex(1,2),c)'        // odd, offends lexers
Y"(complex(1,2),c)"        // odd, offends lexers
Y@(complex(1,2),c)
@Y(complex(1,2),c)
Y<complex<1,2>,c>          // too(?) template like
literal Y(complex(1,2),c)  // verbose
Y{complex{1,2},c}          // nice generalization of the initializer-list syntax
(Y) { (complex) { 1,2 }, c }  // odd, verbose, and almost C99 compatible
```

I consider "offends lexers" a serious problem because tools as well as compilers have the current lexical rules built in.

Requiring extra syntax, such as **@** or **literal**, to distinguish simple semantics from a more complicated semantics seems backwards. For example:

```
complex z = a + literal complex(1,2);        // elaborated for restricted semantics
complex z = a + complex(1,2);        // simple notation for general semantics
```

The simplest and most obvious syntax should express the simples and most efficient implementation (assuming simplicity and efficiency isn't at odds, and they don't seem to be in this case).

That leaves just two alternatives:

```
Y{complex{1,2},c}        // generalized initializer-list syntax
Y(complex(1,2),c)        // usual constructor syntax
```

These two notations are roughly equivalent. In fact, the Stroustrup and Dos Reis proposal for generalizing initializers makes them both valid and semantically equivalent whenever a user hasn't defined a sequence constructor. Therefore, taking the generalized initializer syntax to indicate user-defined literals seem unattractive. That proposal also addresses C99 compatibility as any proposal touching upon a **{}**-based notation in expressions must.

Syntax for literal constructors

Do we need syntax to distinguish literal constructors? Not really, we could define a literal constructor simply as a constructor with the constraints that allows it to define literals.

Why would we want syntax to define literal constructors? I see three reasons:

- to allow the compiler to check that a constructor we think meets the criteria for a literal constructor really does meet them
- to discourage compiler writers from implementing a literal constructor as a run-time construct. This could also be expressed, possibly more politely, as "reassuring the programmer that literal constructors really are implemented at compile time", though it is hard to guarantee that
- to ease the teaching of constructors and the discussion of literal constructors.

The main reason for not distinguishing a literal constructor is that we often (typically) will want an identical non-literal constructor. For example:

```
class complex {
        double re, im;
public:
        literal complex(double r, double i) : re(r), im(i) { }
        complex(double r, double i) : re(r), im(i) { }        // not literal
        // ...
};

complex z1(1,2);        // literal
complex z2(1,sqrt(2));        // not literal
```

That would appear to be a pointless redundancy. Programmers do not like to type the same thing twice, and here there are no significant complementary benefits.

By having "literal" mean "potentially literal", we might abbreviate this to

```
class complex {
        double re, im;
public:
```

```
        literal complex(double r, double i) : re(r), im(i) { } // literal and not literal
        // ...
};

complex z1(1,2);            // literal
complex z2(1,sqrt(2));      // not literal
```

However, that's rather a small benefit from introducing a new keyword. Note that the current semantics already implies this without the use of a keyword:

```
class complex {
        double re, im;
public:
        complex(double r, double i) : re(r), im(i) { } // literal and not literal
        // ...
};

complex z1(1,2);            // might be literal
complex z2(1,sqrt(2));      // not literal
```

A quandary and a suggested resolution

So, many users want the facility – user-defined literals – but the obvious and ideal syntax is already provided by the language and the obvious semantics is already allowed by the standard. The problem is that we traditionally don't "legislate" performance or specific implementation approaches. What would be the best way – if any – to ensure efficient implementation of literal constructors and to make users confident with the concept?

The obvious approach of introducing new syntax is too heavy handed and would introduce more problems than it is would solve. Instead, we should introduce the term "literal constructor" into the standard with wording encouraging efficient implementation of literal constructors. I fear that such wording would have to be non-normative, but it would express the intent of the committee and introduce the concepts "literal constructor" and "literal of user-defined type" (and/or "user-defined literal") into common use:

A constructor is a literal constructor – that is, a constructor used to define literals of a user defined type ("a user-defined literal") – if

- it is inline (either explicitly or because it is defined in-class)
- it's body is empty
- each initializer expression (if any) is either a constant expression or a copy implemented by a built-in operation or a literal constructor

For example:

```
class X {
        int a;
        X* b;
        Y& c;
        complex d;
public:
        X(int n, X* p, Y&x, complex<float> z)
                : a(n+2),b(p),c(x),d(z) {}          // a literal constructor
```

```
        X(int n, X* p, Y&x) : a(n+2),b(p+1),c(x) {}     // not a literal constructor
        X(int n) { a=n; }                               // not a literal constructor
        // …
};
```

Unless specifically defined to be a variable (e.g. by being declared as a non-**const** local variable or created using **new**), an object created by a literal constructor can be placed if ROM.

Relationship to other proposals

This proposal should not be considered in isolation from other proposals dealing with initialization.

C99 compound literals

In C99, a initializer list of expressions prefixed by a (C-style) cast is called a compound literal and serves a similar role to the user-defined literals proposed here. In C99, the expressions in a compound literal must be constant expressions if the compound literal is in file scope, but do not need to be constant if the compound literal is in function scope (so a function scope "compound literal" is not a literal as I used the term above).

In the absence of a joint policy of C/C++ compatibility, this is not by itself a compelling argument for accommodating the C99 construct, but it would certainly reduce confusion and match some users' expectations/wishes if the C99 construct was accepted as a special case with identical semantics in C++. Accepting something with identical syntax and slightly differing semantics would be unfortunate.

I suggest that we define the C-style cast syntax as equivalent to the constructor syntax. For example:

```
(struct Y) { (struct complex) { 1,2 }, c }      // C99 (and proposed C++)
```

is equivalent to

```
Y(complex(1,2),c)     // C++
```

This would fit both with the lack of syntactic distinction of the **const** and non-**const** cases and with the companion proposal for generalized constructors (see below).

Should we allow non-const compound literals (using the C-style cast syntax) and non-const user-defined literals (using the constructor syntax) in namespace scope? I suggest that both should be allowed, following the usual rules for initializers in namespace scope. For example:

```
extern int x;
Y a(complex(1,2),x);                // ok: run-time evaluation
Y b = Y(complex(1,2),x);            // ok: run-time evaluation
Y c = (Y) { (complex) { 1,2 }, x }  // ok: run-time evaluation
```

In other words, the C99 syntax should be allowed with extensions to match the more general C++ rules.

In C99, it is explicitly allowed to take the address of a compound literal. For example:

```
f(&(struct foo) { 1,2 });
```

This makes sense only if we assume that the **{1,2}** is stored in a data segment (like a string literal, but different from a int literal). I see no problem allowing that, as long as it is understood that unless **&** is

explicitly used or the literal, a user-defined literal is an rvalue with which the optimizer has a free hand. For example:

```
void f(complex*);
// …
f(&complex(1,2));     // ok
```

whereas

```
complex z1 = complex(1,2);
complex z2 = (complex) {1,2};
```

do not require **{1,2}** to be stored as an object somewhere.

It would be tempting to expand this rule to user-defined literals bound to references. For example:

```
void f(complex&);
// …
f(complex(1,2));       // ok?
```

However, this would touch upon some rather brittle parts of the overload resolution rules to do with rvalue vs. lvalue. For example:

```
vector<int> v;
// ...
vector<int>().swap(v);        // ok
swap(vector<int>(), v);       // would become ok
```

I suggest we don't touch this unless we are looking at the rvalue/lvalue rules for other reasons.

Generalized initializer lists

This proposal fits together with the proposal for generalized initializer lists (Stroustrup and Dos Reis).

Generalized constant expressions

The notion of constant expression is severely limited and easily extended. For example, we could accept an inline function defined as an expression of constant expressions as a constant expression, etc.:

```
inline int next(int a) { return a+1; }
int a1[next(2)];                // ok: next(2) cold be a constant expression

int s[] = { 1,2,3,4 };
int a2[s[2]];                   // could be evaluated at compile time

int a2[complex(1,2).real()];  // access to member of literal constructed object

inline length(A) { return sizeof(A) / sizeof(*A); }          // constant expression
```

template<int N> int size(const T(&)[N]) { return N; } // constant expression

Gabriel Dos Reis will present a proposal to allow a limited form of inline functions in constant expressions. His proposal fits with this proposal for literal constructors and enhances its usefulness.

Acknowledgements

Gabriel Dos Reis made constructive comments on several early version of this note.