Document Number: WG14 N480/X3J11 95-081

C9X Revision Proposal

Title: Function Literals
Author: Paul Long
Author Affiliation: Performance Computing Incorporated
Postal Address: 15050 SW Koll Parkway, Suite 2B
Beaverton, Oregon 97006 USA
E-mail Address: plong@perf.com or paul_long@ortel.org
Telephone Number: +1 503 6411221
Fax Number: +1 503 6413344
Sponsor:
Date: 1995-07-27

Proposal Category:

- __ Editorial change/non-normative contribution
- __ Correction
- X_ New feature
- __ Addition to obsolescent feature list
- __ Addition to Future Directions
- __ Other (please specify) ____________________

Area of Standard Affected:

- __ Environment
- X_ Language
- __ Preprocessor
- __ Library
  - __ Macro/typedef/tag name
  - __ Function
  - __ Header
- __ Other (please specify) ____________________

Prior Art:

Lisp lambda expressions [1] and CA-Clipper code blocks [5]. These are important features in their respective languages, and their counterpart would be an important addition to C. Here are roughly equivalent examples coded in Lisp, CA-Clipper, and C.

```
(lambda (x) (print x))
{|x| qout(x)}
(void ()(char *)){|x| puts(x); }
```

Lisp has been around for about thirty-five years. There are, conservatively, 5000 Lisp programmers. CA-Clipper is a nine-year-old superset of the dBase language. Code blocks were added five years ago. There are between 3000 and 4000 CA-Clipper programmers.

Target Audience:

All C programmers. The typical use of function literals, which would effect all C programmers, is as call-back functions, such as the comparison function called through a function pointer by the standard-library "bsearch" function. The ability to succinctly define jump tables using function literals could particularly benefit those programmers who use event-driven architectures, such as GUI programmers and embedded-systems programmers.

This language feature would probably increase the use of function pointers throughout the C programming community because function pointers would be easier to form--a function could be defined where its designator is used. (By ''function pointer,'' I specifically mean an expression having type pointer to function and *not* consisting solely of an identifier declared as having function type.)

Related Documents (if any):

1. ANSI X3.226:1994, ''American National Standard for Programming Language--Common LISP'', X3J13.
2. Baker, Henry G. ''Iterators: Signs of Weakness in Object-Oriented Languages''. *ACM OOPS Messenger 4*,3 (July 1993), 18-25.
3. ISO/IEC 9899:1990, ''Programming Languages--C''.
4. Stepanov, A. and Lee, M. ''The Standard Tmeplate Library''. Hewlett-Packard Laboratories, 1994.
5. Tilney, William E. *Using Clipper, Special Edition*. Que Books, 1993.
6. Prosser and Keaton. C9X Proposal, ''Compound Literals''. WG14 N403/X3J11 93-004, 1995.

Proposal Attached: __ Yes X_ No, but what's your interest?
Abstract:

A *function literal* is an unnamed function. The syntax is similar to that of compound literals [6]. A function literal is distinguished from a compound literal by a type name that is syntactically a function declaration and by a bar-enclosed optional identifier list immediately following the left curly brace.

```
function-literal:
   ( type-name ) { | identifier-list opt |
   declaration-list opt statement-list opt }
```

The parameter types in the parameter-type list of the type name are the respective types of the parameters whose identifiers are in the identifier list. The number of parameter types shall be equal to the number of identifiers in the identifier list, with the exception that there is no identifier for void type.

The type of a function literal is the named type. A function literal undergoes the same conversion-to-pointer scheme as any other function designator.

Declarations that are made within the same scope and at outer scopes to the scope within which a function literal occurs are visible to the function literal. When a reference to an object with automatic storage duration that is no longer guaranteed to be reserved is evaluated, the behavior is undefined.

72

Identical function literals containing references to the same objects need not be distinct.

Examples

Here is a function literal used as a call-back function to the "bsearch" function.

```
   p = bsearch(&key, a, sizeof a / sizeof *a, sizeof *a,
         (int ()(const void *, const void *)){|x, y|
         return *(int *)x - *(int *)y;});
```

Here is an example that illustrates how scope and storage duration relate to a function literal.

```
   foo() {
      int a;
      int (*bar)();
      {
         int i = 1;
         {
            int b;
            bar = (int ()()){|| return i; };
            b = bar();
            {
               int i = 2;
               int c = bar();
            }
         }
      }
      a = bar();
   }
```

As the function, "foo", is executed,
- "b" is set to 1,
- "c" is set to 1 and not 2 because the function literal contains a reference to i that is bound to the object at the scope within which the function literal appears, not when it is called, and
- the behavior is undefined when the function literal is called at the outer-most scope because the function literal refers to an object with automatic storage duration that is no longer guaranteed to be reserved.

The undefined behavior resulting from evaluating an object with automatic storage duration that is no longer guaranteed to be reserved is no more onerous than the behavior that results from the following passage in section 6.1.2.4 [3]--\`\`The value of a pointer that referred to an object with automatic storage duration that is no longer guaranteed to be reserved is indeterminate.'' Here are analogous examples of referring to objects with automatic storage duration when they are no longer guaranteed to be reserved. First, via a pointer to such an object:

```
  int *bar() {
      int i = 3;
      return &i;
   }

   foo() {
      int j = *bar();
   }
```

Next, via a function literal that refers to such an object:

```
   int (*bar())() {
      int i = 3;
      return (int ()()){|| return i; };
   }

   foo() {
      int j = bar()();
   }
```

Rationale

If compound literals are accepted into the revised C Standard, all complete types will have literal and non-literal forms except void type and function types. It would make no sense to have a void literal (probably), but function literals would be quite useful. Of function literals and the compound literals, I--and I believe that most programmers--would find function literals the most useful, followed by structure compound literals and then array compound literals. Function literals would complete the set of literals in C.

I came to consider their addition to C by way of disliking iterators (see [2]), such as those in the Standard Template Library (STL) [4]. Instead, wherever possible, I favor macros or functions for traversing and searching. Functions of this type use a call-back function for their action. Examples are the STL "for_each" template function and the C standard-library "bsearch" function. However, a problem with call-back functions is that their definition is lexically distant from their reference. This often makes code ponderous and therefore difficult to understand. A solution is to use a function literal in place of the reference to a function that is defined elsewhere.

Implementation Issues

Function literals present opportunities for code-space optimization, e.g., just as these two string literals may occupy the same memory location, so may the first two function literals. Binding complicates things, though--syntactically identical function literals, such as the first, second, and third ones, may bind to a different set of objects.

```
   foo() {
        char *s1, *s2;
        void (*f1)(void), (*f2)(void), (*f3)(void);
        unsigned lineno;

        s1 = "hello";
        s2 = "hello";
        f1 = (void ()(void)){|| ++lineno; };
        f2 = (void ()(void)){|| ++lineno; };

        {
          unsigned lineno;

          f3 = (void ()(void)){|| ++lineno; };
        }
    }
```

Language Compatibility Issues

Differences between function literals and the cited prior art include:

- syntax
- function literals employ lexical scoping, and do not form closures, as does the prior art

Subsetting

Because function literals are such a fundamental feature, I do not recommend subsetting.

Alternatives

Although a programmer would probably expect to be able to refer to identifiers declared in all outer scopes, function literals could act like regular function definitions and disallow references to identifiers declared within other functions.

Here are the four, often conflicting, goals for function-literal syntax, ordered in decreasing importance, by which this proposal was guided.

1. similarity to compound-literal syntax
2. what a programmer would expect
3. similarity to function-declaration syntax
4. brevity

This is an example using the proposed syntax.

```
        (int ()(const void *, const void *)){ |x, y|
        int diff = *(int *)x - *(int *)y;
        return diff; }
```

Alternative syntaxes could emphasize different orders. Here are some progressive alternatives, expressing the same example. They de-emphasize goals 1 and 2 then emphasize goals 3 and 4. These are offered only for thought--some may actually represent ambiguous grammars.

1. Remove the declaration list from the function body.

```
    (int ()(const void *, const void *)){ |x, y|
    return *(int *)x - *(int *)y; }
```

2. Instead of an optional statement list, the function body is an optional expression. Sequence and alternation are accomplished with the comma operator and the conditional operator, respectively. Iteration and the direct transfer of control (i.e., via the goto statement) are not possible. This leaves plenty of room for expressive logic but discourages lengthy function literals. At this point, "expression literal" might be a better term than "function literal."

```
    (int ()(const void *, const void *)){ |x, y|
    *(int *)x - *(int *)y; }
```

3. Abandon the parenthesized type name in favor of integrating the function declaration with the identifier list. This avoids the K&R-like function definition where identifier list and parameter type declarations are separate, but it no longer uses syntax that is similar to compound literals.

```
    { int |const void *x, const void *y|
    *(int *)x - *(int *)y; }
```

4. The use of parentheses instead of bars to enclose the parameter type list.

```
    { int (const void *x, const void *y)
    *(int *)x - *(int *)y; }
```