Title: Operator Overloading Without Name Mangling v2
Author: Marcus Johnson
Email: MarcusLJohnson1991@gmail
Company: Self
Date: December 11th 2023
Document: n3201
Proposal Category: New Feature
References: C2Y Feature
Acknowledgements: Jens Gustedt, Keith S. Thompson

Revision History:

n3201: Operator function bodies MUST included the overloaded operator in their function body; Editorial fixes from Keith.

n3182: Initial WG14 proposal, Revised and simplified syntax based on reddit feedback, renamed the keyword from _Overload to _Operator

Abstract:

Allow initialization, assignment, relational, equality, and mathematical operators to be overloaded for User Defined Types

Motivation:

Buffer overflows are a consistent source of bugs, and the main cause of this class of bugs being so prevalent in C is due to the fact that objects and the sizes of those objects are kept separate in C, the most obvious example is C-strings i.e. null terminated strings.

Name Mangling: My proposal does NOT require name-mangling, the function that implements the operation is directly named by the programmer, and that name is directly referenced in the _Operator declaration. Name mangling is not at all required for this proposal.

Semantics:
_Operator is a declaration that requires two arguments, the first is the operator to overload, as either its literal symbol or the name of the operator from iso646.h; the second argument is the name of a previously declared function that implements this operation for the types in question.

There are some rules for how the named functions must behave, the return value of the _Operator function must match its logical counterpart, for example relational operators must yield a result of type int.

The number of parameters of the named functions must match the operator expected, e.g. == and != must take two arguments, the two arguments must either be the same User Defined Type, or a User Defined Type and a fundamental type.

With the way the NOT operator is defined, != operations are synthesized by the compiler as being the inverse of a defined equality operator, if one is defined, if a comparison operator is not defined, the != operator is also undefined.

Functions must contain the matching operator in their function bodies. i.e. _Operator declarations that associate the compares-equal operator with a function, must contain the compares-equal operator in the body of the function

named in the _Operator declaration. (iostream-esque shenanigans with overloading the bitwise shift operators to read/write characters and strings isn't allowed).

Syntax:
_Operator <Operator> <VisibleFunctionName> ; is a declaration that takes two arguments, the first argument is the operator to overload, overloadable operators are one of the following:

Initialization Operators: = (Initialization) (only literal initialization is allowed; the empty initailizer is not allowed to be overloaded.

Assignment Operators: = (assignment), += (add-assign), -= (subtract-assign), *= (multiply-assign), /= (divide-assign), %= (modulo-assign), ^= (binary XOR-assign), |= (binary OR-assign), &= (binary AND-assign), <<= (left-shift assign), >>= (right-shift assign)

Mathematical Operators: + (unary addition), - (subtraction), * (multiplication), / (division), % (modulus), & (binary AND), | (binary OR), ^(binary XOR), << (left bitwise shift), >> (right bitwise shift)

Relational Operators: < (less-than), <= (less-than-or-equal), > (greater-than), >= (greater-than-or-equal)

Equality Operators: == (compares-equal), != (not-equal)

Forbidden Operators: [](Array index), *(Pointer dereference), ->(Member dereference) .(Member access), ,(Comma operator), ()(Function call operator), ()(Type conversion), ++(Increment), --(Decrement), &(Address-of), &&(Logical AND), ||(Logical OR), sizeof, typeof.

The first argument in an _Operator declaration MUST be present within the body of the named function so that overloaded operators have no cognitive overhead. this rule is intended to disallow iostream-esque redefining operators to mean something completely different like << to write characters/strings in iostream; Overloaded operators in C have the same semantics as builtin operators. They just allow programmers to define how the operators work with user defined types.

The second argument in an _Operator declaration is the name of a visible function name that implements this operation for the relevant user defined data types, and there are a few restrictions on the prototypes available for these functions, namely.

Functions that implement binary operators must take two parameters, the LHS of the operator expression is to be the first parameter of the called function, and the RHS of the operator expression is to be the second parameter of the called function.

_Operator declarations are allowed and encouraged to be declared in headers, the only requirement for doing so is that the named function referenced in the _Operator declaration is itself previously declared.

Example code:

UTF8String.h:
```
#include <assert.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>

#ifndef UTF8String_h
#define UTF8String_h

typedef struct UTF8String UTF8String;

UTF8String UTF8String_Init(const char8_t *CodeUnits);
_Operator = UTF8String_Init;

bool UTF8String_Append(UTF8String String, const char8_t *String2Append);
_Operator += UTF8String_Append;

#endif /* UTF8String_h */
```

UTF8String.c:

```c
#include "UTF8String.h"

typedef struct UTF8String {
    size_t SizeInCodeUnits;
    char8_t *Data;
} UTF8String;

char8_t *UTF8String_GetCString(UTF8String String) {
    return String.Data;
}

UTF8String UTF8String_Init(const char8_t *CodeUnits) {
    UTF8String String = {};
    String.SizeInCodeUnits = strlen(CodeUnits);
    memcpy(String.Data, CodeUnits, String.SizeInCodeUnits);
    return String;
}

bool UTF8String_Append(UTF8String String, const char8_t *String2Append) {
    size_t AppendSize    = strlen(String2Append);
    char8_t *Reallocated = realloc(String.Data, String.SizeInCodeUnits +
AppendSize);
    assert(Reallocated != NULL);
    for (size_t ReallocCodeUnit = 0; ReallocCodeUnit < String.SizeInCodeUnits
+ AppendSize; ReallocCodeUnit++) {
        if (ReallocCodeUnit < String.SizeInCodeUnits) {
            Reallocated[ReallocCodeUnit] = String.Data[ReallocCodeUnit];
        } else if (ReallocCodeUnit < String.SizeInCodeUnits + AppendSize) {
            Reallocated[ReallocCodeUnit] = String2Append[ReallocCodeUnit +
AppendSize];
        }
    }
    String.SizeInCodeUnits += AppendSize;
    return true;
}

int main(int argc, const char *argv[]) {
    UTF8String String = u8"Hello, ";
    String += "World!";
    printf("%s\n", UTF8String_GetCString(String)); // Output's "Hello, World!
\n" (Without quotes)
    return 0;
```

```
}
```