Title: Operator Overloading Without Name Mangling
Author: Marcus Johnson
Company: Self
Date: November 11th 2023
Document: n3182
Proposal Category: New Feature
References: C2Y Feature
Acknowledgements: Jens Gustedt

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

Abstract:
Allow initialization, assignment, comparison, 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 keyword that requires two trailing tokens, the first is the operator to overload, as either it's literal symbol or the name of the operator from iso646.h; the second token is the name of a previously declared function that implements this operation.

There are some rules for how the named functions must behave, the return value of the function must match it's logical counterpart, for example comparison operators must return bool.

The number of parameters of the named functions must match the operator expected, e.g. == and != must take two arguments and return a bool, 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 comparison operator, if one is defined, if a comparison operator is not defined, the != operator is also undefined.

Syntax:
_Operator <Operator> <PreviouslyDeclaredFunctionName> ; is a keyword that takes two tokens, the first tolen is the operator to overload, overloadable operators are one of the following:

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

Mathematical Operators: + (unary addition), - (subtraction), * (multiplication), / (division), % (modulus), & (binary AND), | (binary OR), ^(binary XOR).

Comparison Operators: == (compares-equal), < (less-than), <= (less-than-or-equal), > (greater-than), >= (greater-than-or-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 token in an _Operator declaration may be either the literal operator token e.g. "|=", or the ISO646.h name of the operator i.e. "or_eq".

The second token in an _Operator declaration is the name of a previously declared function 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 previously declared.

Example code:

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

#pragma once

#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:
```
#include "UTF8String.h"

typedef struct UTF8String {
```

```
    size_t SizeInCodeUnits;
    char8_t *Data;
} UTF8String;

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);
    char *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];
        }
    }
    return true;
}

int main(int argc, const char *argv[]) {
    UTF8String String = u8"Hello, ";
    String += "World!\n";
    printf("%s", String);
    return 0;
}
```