n3051 Operator overloading in C

jacob navia

Contents

Contents 3

1 Motivation 5
1.1 The proliferation of numeric types 5
1.2 The lack of counted strings and arrays 5
1.3 The lack of support for fat pointers 6
1.4 Lack of support for new operations 7
1.5 Computer science context 7

2 Syntax 9
2.1 Rules 10
2.1.1 Operator Arguments and result types 10
2.1.2 Rules 12
2.1.3 Choosing the right function for a given set of arguments 12
2.2 Other features 13
2.2.1 Using external libraries 13
2.2.2 Avoiding operator overloading 13
2.3 Open questions 13
2.3.1 'operator' is not a new keyword 13

3 Applications 15
3.1 New number types 15
3.2 Bounds checked tables 16
3.2.1 Read only tables, copy on write tables, sparse tables 16
3.3 Sample implementation 16

1 Motivation

This proposal tries to address several problems in the C language with a single modification to its syntax: operator overloading.

The problems I try to address here are:

- The proliferation of numeric types.
- The lack of counted strings and arrays in C.
- The lack of array properties like read-only, copy on write, and many others.
- The lack of support for fat pointers, i.e. pointers that carry information about the array they are pointing to.
- The lack of support for new operations already present in most modern CPUs like clamped addition, and others.

I will describe below what I mean with each of those items and how my proposal addresses them.

1.1 The proliferation of numeric types

C has the following numeric types:

- 10 integer types (5 basic types: char, short, int, long and long long, and their 5 unsigned counterparts.)
- 6 floating point types (float, double, long double, and another three decimal ones)
- 3 complex types based on the three real types (float complex, double complex, etc)
- There are proposals for variable precision integers (bignums), 128 bit integers, etc.

This makes for 19 different types of numbers already defined, and as time passes and CPUs grow ever more powerful, many others will arrive. This is confusing.

If this proposal is realized the standard would shrink from 19 to 13: the traditional integer types (10) the traditional floating point types (3) and that's all. Decimal and complex numbers would leave the language specification and would be part of recommended standard operator overloading features. That would be part of the standard library.

At the same time, since any user can define a new type of number, the language would be much more powerful than today. The standard committee would *not* be forced to decide in matters that are better decided by specialist in their domain: mathematicians, software developers with specific needs, etc. The C language standard would shrink considerably.

1.2 The lack of counted strings and arrays

This is a sore point of the language. We all know the problem of buffer overflows, exploits, etc. This problem has been with us too long and it is time to think about a general solution to the problem, a solution that takes care of all usage needs and all combinations of:

- Performance.

- Memory footprint.
- Memory allocation/deallocation.
- Support for UTF8 and different types of display for strings.

Obviously this is an impossible goal. No possible implementation can satisfy *all* those requirements. That is why the standard committee will *not* decide how strings should be done but will propose a compatible interface using similar names as the known ones (Strcmp, Strchr, etc) to make all those string libraries compatible and able to interchange data between them. This means that a common *cast operation* will be provided to convert any string type to a base string type, and all libraries will be required to convert from the basic string type to their type.

As everyone agrees since ages, counted strings are faster in use than zero terminated ones. The ubiquitous strlen is called far too many times today.

The huge difficulty of the task at hand has provoked the still stand in the development of this part of the language. It is time to acknowledge that a single universal string type is an impossible goal and to design a common interface for *many* string types that are suited each to its own domain.

We do not have a single type of number, we have integers of different sizes, we have floating point, etc. Each one of them is adapted to a certain size of their contents and to the machines that can handle them. Strings are no different. For numbers the language provides the basic conversions from one into another. It will be the same for strings.

1.3 The lack of support for fat pointers

One of the most important characteristics of a string is its size. In the Annex K, there is an implementation of more secure string libraries where the size of the string is passed along together with the pointer to the string contents.

It is better to pass that information as a *single entity*, i.e. a "fat" pointer. This pointers consist of two parts: a metadata part, that describes certain characteristics of the object, and the data itself, following the data header. In the case of a string we have:

```
1 typedef struct tagString {
2    size_t size; char chars[size];
3 } String;
```

This bare bones string can be improved with other annotations, like the encoding of the characters, for instance. Many other applications are possible, like implementing read-only strings, copy-on-write strings, etc.

[1] What operator overloading contributes to this schema is the ability to index strings just as before:

```
1    String str1,str2;
2
3    // Strchr returns a copy on write string
4    str2 = Strchr(str1,'\n'); // str2 is a copy on write of str1
5    // subtraction verifies that str2 points somewhere WITHIN str1
6    str1[str2-str1]=0;
7    // The copy on write is triggered now
8    str2[12]='m';
```

[1] Operator overloading complements the _Generic feature, allowing to specify different forms of the Strcmp function by dispatching according to the types of the arguments.

1.4 Lack of support for new operations

Operator overloading makes it trivial to add new operations like clamped addition, and makes obsolete `ckd_*` functions: checking for overflow can be done easily. What is more important, all existing code can be checked for overflow just with a change of the types without any other modification. A simple `#define` allows for that.
[2]

1.5 Computer science context

Operator overloading is a common feature in modern programming languages. It is supported in:
`Ada, Algol66 (yes, since 1966!)  C#, C++, D, Eiffel, Fortran, F#, Haskell, Java, Lua, MATLAB, Object Pascal, Free Pascal, Delphi (since 2005), PHP`[3]`, Perl, Python, R, Ruby, Visual Basic.NET` and that is not an exhaustive list.

As far as this proposal is concerned, it tries to follow the C++syntax to avoid surprises to many users that use that programming language. Obviously C is not C++ and differences are explained in the text.

[2] The problem of the existing standard ckd_* functions is that you have to rewrite all code that uses those functions! No longer c=(a+b)/(a-b) but ckd_add(a,b)/ckd_sub(a,b))
Operator overloading is useful for defining new number types AND new operators that do something completely different.

[3] (using magic methods, ArrayAccess interface, or Operator extension)

2 Syntax

Operator overloading will be considered by the compiler as an option only for user defined types, i.e., structures. Please note that this makes it impossible to redefine the basic types, such as int or double. The following typedef is assumed in the examples below:

`typedef struct tagComplex { double x; double y; } COMPLEX;`

To define a new operator, proceed as follows:

```
1    COMPLEX operator +(const COMPLEX left, const COMPLEX right)
2    {
3        COMPLEX result;
4        result.x = left.x + right.x;
5        result.y = left.y + right.y;
6        return result;
7    }
```

Note several things here:

- This declaration can happen only at the global level, like all function declarations.
- This function returns a *new* object.
- The input arguments to this function are pointers to objects, or as in the example, the objects themselves passed by value.
- This declaration is equivalent to a function you would have written with a long name (see how this name is derived below). Each time the compiler finds a match for this function call within an addition, it will call this function.

Formally then, we have:

`result-type operator symbol '(' argument-list ')'`

Assume then that after this definition is seen by the compiler, you write the following code:

```
1    void example(void)
2    {
3        COMPLEX a = {2.0,0.0},b = {3.0,0.0},c;
4        c = a+b;
5    }
```

This instruction will be interpreted as:

`c = _op_plus_COMPLEX_COMPLEX(a,b);`

The name of the function is derived as follows:

1. It starts with the prefix `_op_` .
2. Followed by the name of the operator. The operator names are documented below.
3. Followed by an underscore to separate types, except for the last type.

4. If any spaces or parentheses appear in the type name, they will be substituted by an underscore. Thus `unsigned int` will become `unsigned_int` the type `BitInt(27)` becomes `BitInt_27_` . [1]

2.1 Rules

1. All operators should have at least one argument that is a user-defined structure.
2. Type conversions can be realized by defining different operators for different input arguments. You can write several operators +, each with different types of arguments.

Here is an example. We implement addition of c = b + double precision value in the above context.

```
1  COMPLEX operator +(const COMPLEX left, const double right)
2  {
3      COMPLEX result;
4      result.x = left.x + right;
5      result.y = left.y;
6      return result;
7  }
```

Then you can write in your code:
`COMPLEX c = a + 5.9;`
and the compiler will call the right operator for you. Note that the compiler-generated name for this operator will be `_op_plus_COMPLEX_double`, which differentiates it from the preceding example.

2.1.1 Operator Arguments and result types

In the table below, `T`, `T1`, and `T2` can be all different types, or be all the same or any combination thereof. At least one of `T1` or `T2` must be a user defined type or a pointer to a user defined type. The "Name" column defines the name that will be used when generating the underlying function name.

	Op. Name	Syntax	Description and remarks
The four operations			
+	plus	`T operator+ (const T1 arg1,const T2 arg2)`	If one of the arguments is a pointer, the other argument can't be an integer expression. [2]
-	minus	`T operator- (const T1 arg1,const T2 arg2)`	If one of the arguments is a pointer, the other argument can't be an integer expression.
*	multiply	`T operator* (const T1 arg1,const T2 arg2)`	No pointer restrictions
/	divide	`T operator/ (const T1 arg1,const T2 arg2)`	No pointer restrictions
Comparisons			

[1] Rationale: This is a very simple schema that works well. It is readable by people and conveys all the needed information. More important, it allows users to call directly those functions.

[2] Note that the addition operator is NOT commutative, i.e. a+b != b+a. This is against my personal viewpoint but allows compatibility with C++ that made the mistake of allowing that in the first place. This proposal is bug compatible with C++ and you can do horrible things like "abc"+"def". Obviously "abcdef" is different from "defabc".

==	equal	`bool operator==` `(const T1 arg1,const T2 arg2)`	If one argument is a pointer, the other can't be a pointer. The compiler assumes that when one operator is defined, the symmetrical comparison is equivalent. Furthermore, if the operator == is defined, the operator != can be constructed with the negation of equal and vice versa
!=	notEqual	`bool operator!=` `(const T1 arg1,const T2 arg2)`	Same considerations apply as operator ==.
<	less	`bool operator<` `(const T1 arg1,const T2 arg2)`	If one argument is a pointer, the other can't be a pointer. If the operator >= is defined, it is assumed that this operator, if absent, is the negation of the operator >=.
<=	lessEqual	`bool operator<=` `(const T1 arg1,const T2 arg2)`	If one argument is a pointer the other can't be a pointer. If the operator > is defined, it is assumed that this operator is the negation of it.
>=	greater Equal	`bool operator>=` `(const T1 arg1,const T2 arg2)`	If one argument is a pointer the other can't be a pointer. If the operator > is defined, it is assumed that this operator is the negation of it.
Other operators			
!	not	`bool operator!(T arg1)`	
%	mod	`T operator%` `(T1 arg1,T2 arg2)`	No restrictions
<<	leftShift	`T operator<<` `(T1 arg1,const T2 arg2)`	No restrictions
>>	rightShift	`T operator>>` `(T1 arg1,const T2 arg2)`	No restrictions
&	and	`T operator&` `(T1 arg1,const T2 arg2)`	No restrictions
|	or	`T operator|` `(T1 arg1,const T2 arg2)`	No restrictions
Table access			
[]	idx	`T1 operator[]` `(T1 *const arg1,const T2 arg2)`	The first argument is a pointer to a table of type T1, the second must be an integer type. This operator is used in read-only access to a table. The second argument must be within the bounds of the table.
[]=	idxasgn	`T3 operator []=` `(T1 *arg1, T2 idx,T3 newValue)`	This is a table access in write mode. The table must allow write access. Note that the value returned is the value assigned to the table to allow chains of assignments.
=	assign	`T1 *operator=` `(T1 *lvalue,const T2 rvalue)`	`lvalue` points to a location to be modified. [3]

[3] It is assumed that the space required for the assignment is enough. The size should be at least sizeof(T1) but can be bigger. Suppose assigning the integer 1 to a 256 bit integer: the integer size is sizeof(int) but the

( )	cast	`T operator()(T1 arg1)`	This is a generalized conversion operator for converting a `T1` into a `T` using the usual syntax `b = (T) arg1;`

Note that the operators `&&` and `||` can't be overloaded. The explanation is that short circuit evaluation that is implicit in those operators, makes it impossible to substitute their operation with a function call. Since all arguments to a function must be evaluated before the call actually occurs, short circuit evaluation would be impossible.

The composite operators can be synthesized from its components: operator `+=` is an addition followed by an assignment. The same for `++` and `--`.

2.1.2 Rules

- At least one of the parameters for the overloaded operators must be a user defined type. They must be a structure or a union, not just a typedef for a basic type. If you want to overload operations for a basic type, for example the `int` type, you should define a structure like this:
  `typedef struct tagWatchedInt { int data;} WatchedInt;`
- Pointers are accepted as parameters as long as the overloaded operator doesn't interfere with basic pointer operators defined already by the language. For instance the pointer + int operation is defined already and no overloaded addition operator can have pointer and integer arguments. These constraints are specified in the table above.
- The result type of an overloaded operator can be anything except for the comparisons operators and the logical not operation that return a boolean.
  The array indexing operators in read mode (operator `[]`)return a pointer to the element, or, in the case of read only arrays, they should return a copy of the element, not any pointer. The result of the operator for array element assignment (operator `[]=`)is the new value, to allow scattered array assigning:
  `table[23] = table[45] = 67;`
- The number of arguments for each operator is fixed, and can't be replaced with `...` using variable arguments lists.

2.1.3 Choosing the right function for a given set of arguments

There can be only one operator that applies to a given combination of input arguments, i.e. to its signature. If, at the end of the lookup algorithm more than one overloaded operator for the same operator and argument types is found, an error is issued and no object file is generated.

The algorithm is as follows:

1. Compare the input arguments for all overloaded functions for this operator without any promotion at all. If at the end of this operation one and only one match is found return success. If more than one match is found return an error.

2. Compare input arguments ignoring any signed/unsigned differences. If in the implementation sizeof(int) == sizeof(long) consider long and int as equivalent types. The same if in the implementation sizeof(int) == sizeof(short). Consider the enum type as equivalent to the int type. If at the end of this operation one and only one match is found return success. If more than one match is found return an error.

actual size of the memory modified can be much greater

3. Compare input arguments using the usual arithmetic conversions. If only one match is found return success. If more than one match is found return an error.
4. Substitute for comparisons operators the inverted one as specified in the table of operators above. Again, only one match is accepted.
5. If a defined `cast` operator is defined and it can convert a type into one of the accepted types apply those casts and search again.
6. No matching operator has been found. Return failure.

2.2 Other features

2.2.1 Using external libraries

Very often, the user will be interested in using an existing library for handling a numeric type instead of replicating it. In that case the following syntax is used:

```
1 Tensor operator+ = TensorAdd; // TensorAdd is in some external library
```

It is assumed that `TensorAdd` is in scope and the arguments list matches the requirements of operator `+`.

Formally then, we have

result-type **'operator'** operator-sign **'='** <identifier>

Constraints: Only one such assignment can be written within a compilation unit for a given signature.

2.2.2 Avoiding operator overloading

At any moment the user can call directly the underlying overloaded function. Instead of writing `c=a+b` he/she can write `c = _op_plus_Mytype_Mytype(a,b);`

2.3 Open questions

This document is designed as a strictly "start of the discussion" framework. The ideas here could be off, many things probably are wrong. Nevertheless, it is a start.

A big open question is how to declare and use function pointers in this context. How can you get a function pointer to a multiplication operator function?

One solution could be to use the "mangled" operator name:

Suppose the following code:

```
1 typedef Mytype (*fnPtr)(Mytype,Mytype);
2 Mytype operator*(Mytype,Mytype);
3 fnPtr = _op_mult_Mtype_Mtype; // Use the plain name of the operator
```

2.3.1 'operator' is not a new keyword

In the implementation I have developed, the identifier 'operator' is just a marker that only can be used

- At the global level. No operator declaration is allowed within a function.
- It must be preceded by a type declaration.
- It must be followed by one or more characters from the following set:
  `+ - * / % ( < = > & | !`
- After the operator symbol a left parentheses or an equals sign follows.

3 Applications

3.1 New number types

Here is a very minimal description of a new kind of number.

```
typedef struct tagFloat256 {
   int sign:1;
   int exponent:19;
   _Bitint(236) mantissa;
} Float256; // IEEE octuple precision float

Float256 operator+(const Float256,const Float256);
Float256 operator+(const Float256,const long double);
Float256 operator+(const long double,const Float256);
Float256 operator+(const Float256,const long long);
Float256 operator+(const long long,const Float256);

// The other four operations are very similar.

long double operator()(const Float256); // Conversion of Float256 to long double
Float256 operator()(const long double); // Conversion from long double to Float256
long long operator()(const Float256); // Conversion from Float256 to long long
Float256 operator()(const long long); // Conversion from long long to Float256

// Assignment

Float256 *operator=(Float256 *,const Float256);
Float256 *operator=(Float256 *,const long long);
Float256 *operator=(Float256 *,const long double);

// Comparisons
bool operator<(const Float256,const Float256);
bool operator<(const Float256,const long long);
bool operator≤ (const Float256,const Float256);
bool operator≤ (const Float256,const long double);
bool operator==(const Float256,const Float256);
bool operator==(const Float256,const long long);
bool operator==(const Float256,long double);
```

Surely, passing a 256 bit number (32 bytes) by value could slow down an implementation. To pass everything by address instead of by value, an alternative definition of the above structure could be:

```
   typedef struct tagFloat256 {
      int sign:1; int exponent:19; _Bitint(236) mantissa; } Float256[1];
```

This forces the number to be passed by address.

3.2 Bounds checked tables

```
typedef struct tagTable {
    unsigned size; // up to pow(2,CHAR_BIT * sizeof(unsigned)) elements
    int data[.size];
} intTab;

int operator[](intTab tab,unsigned idx)
{
        if (idx ≥ intTab.size)
            longjmp(outOfBounds,idx);
        return intTab.data[idx];
}
```

Obviously these tables should be passed by reference.

3.2.1 Read only tables, copy on write tables, sparse tables

If we allow for some bits of the 64 bit `size` field to be used for flags, we can implement special kinds of tables like read-only, or sparse ones.

```
typedef struct tagTable {
    unsigned size; // up to pow(2,CHAR_BIT * sizeof(unsigned)) elements
    unsigned flags;
    int data[.size];
} intTab;

// Some possible flags
#define TABLE_READ_ONLY 1
#define TABLE_COPY_ON_WRITE 2
#define TABLE_SPARSE 4
```

Operator overloading allows access in read only mode. The overloaded function checks this flag. Access functions return a COPY of the data, instead of returning a pointer to the data itself.

Copy-on-write tables can be implemented in a similar manner, as well as sparse tables. This simplifies software because all of them are accessed in the natural way: `tab[idx]`.

3.3 Sample implementation

An implementation of the above specifications (with small differences) exists since 2005. It can be downloaded from:

`https://lcc-win32.services.net`