C+++: User-Defined Operator Symbols in C++

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract: The paper presents the basic concepts of C+++, an extension of C++ allowing the programmer to define new operator symbols with user-defined priorities by specifying a partial precedence relationship. Furthermore, so-called flexary operators accepting any number of operands and operators with lazily evaluated operands are supported. The latter are particularly useful to implement new kinds of control structures.

1 Introduction

Programming languages such as Ada [Tu01] and C++ [St00] support the concept of *operator overloading*, i. e., the possibility to redefine the meaning of *built-in* operators for *user-defined* types. Since the built-in operators of many languages are already overloaded to a certain degree in the language itself (e. g., arithmetic operators which can be applied to integer and floating point numbers, or the plus operator which is often used for string concatenation as well), it appears rather natural and straightforward to extend this possibility to user-defined types (so that, e. g., plus can be defined to add complex numbers, vectors, matrices, etc., too).

Other languages, e. g., Smalltalk [GR89], Prolog [CM94], and modern functional languages such as ML [MTH90] and Haskell [Pe03], also allow the programmer to introduce *new operator symbols* in order to express application-specific operations (such as determining the number of elements contained in a collection `c`) more directly and naturally (e. g., as `#c`) than with overloaded built-in operators (e. g., `*c` in C++) or with methods or functions (e. g., `c.size()` or `size(c)`).

The introduction of new operator symbols (especially if they denote *infix* operators) immediately raises the question about their *binding properties*, i. e., their *precedence* with respect to built-in and other user-defined operators, and their *associativity*. In the above languages, the programmer introducing a new operator symbol is forced to assign it a *fixed* precedence level on a predefined *absolute* scale (e. g., an integral number between 1 and 10). This approach is both inflexible (for example, it is impossible to define a new operator that binds stronger than plus and minus but weaker than mult and div, if there is no gap between these operator classes in the predefined precedence scale) and overly prescriptive (because the programmer is always forced to establish precedence relationships between *all* operators, even though some of them might be completely unrelated and never appear

together in a single expression).

The approach described in this paper (which is not restricted to C++ conceptually) advances existing approaches in the following ways:

- The *precedence* of new operators need not be fixed on an absolute scale, but only *relative* to other operators, i. e., the precedence relationship is not a complete, but only a *partial order* on the set of operator symbols, which can be incrementally extended on demand.
- In addition to well-known unary and binary operators, *flexary* operators connecting any number of operands are supported.
- Finally, operators whose operands are only evaluated *on demand* (roughly comparable to *lazy evaluation* in functional languages) are supported in a language such as C++ whose basic execution model is imperative.

Sec. 2 describes the basic features of C+++, an extension of C++ supporting the introduction of new operator symbols. Secs. 3, 4, and 5 illustrate these with a number of examples, demonstrating in particular the advances mentioned before. Finally, Sec. 6 concludes the paper. An accompanying Technical Report [He04] describes the implementation of C+++ by means of a precompiler for C++.

2 New Operators in C+++

New operator symbols in C+++ are introduced by *operator declarations* at global or namespace scope (i. e., outside any function or class definition) starting with the keyword sequence `new operator`. These are both existing C++ keywords which cannot occur in juxtaposition, however, in the original language [St00]. Therefore, the already large set of C++ keywords need not be extended to support this language extension. Inside an operator declaration, however, numerous "local" or "context-dependent" keywords which will not be treated as such elsewhere (e. g., `unary`, `left`, `right`, etc.) can be used to describe properties of the new operator.
New operators are either *identifiers* as defined in the base language C++ (i. e., sequences of letters and digits starting with a letter, where the underscore character is treated as a letter) or sequences of one or more *special characters* (all characters except white space, letters, digits, and quotation marks). A new operator symbol of the latter kind becomes a *token* of the lexical analyses as soon as it has been declared, i. e., it might influence the parsing process of the remaining input. To give an artificial example, a sequence of five plus signs (without intervening white space or comments) is parsed as three tokens `++`, `++`, and `+` in original C++ (i. e., the lexical analyzer is "greedy"). If a new operator `+++` is introduced, the same sequence gets parsed as two tokens `+++` and `++` afterwards. (Of course, such "operator puzzles" can be avoided by always separating tokens by white space.)

In general, built-in operators in C++ can be applied *prefix*, *infix*, or *postfix*, and there are several operators which can be applied both prefix and infix (`+`, `-`, `*`, `&`, and `::`) or both

prefix and postfix (`++` and `--`). In analogy, new operators are categorized as either *unary* (meaning prefix and postfix applicable) or *binary* (meaning prefix and infix applicable).

As in standard C++, the semantics of operators are defined by *operator functions*, i. e., functions whose name consists of the keyword `operator` followed by an operator symbol. Functions corresponding to prefix and infix applications of an operator take one resp. two arguments representing the operator's operand(s). To distinguish postfix from prefix applications, operator functions corresponding to the former receive a dummy argument of type `int` in addition to the argument representing the operator's operand. (Since the same operator cannot be applied both infix and postfix, it is always well-defined whether a two argument operator function corresponds to an infix or postfix application.)
To define generic operators, it is possible to define operator functions as function templates. Unlike built-in operators, new operators cannot be implemented by member functions of a class, but only by ordinary (i. e., global or namespace-scope) functions.
To retain the original C++ rule that the meaning of built-in operators applied to built-in types must not be changed, it is forbidden to define an operator function whose operator symbol and parameter types are all built-in. In other words, only definitions where either the operator symbol or one of the parameter types (or both) is user-defined, are allowed.

As in standard C++, postfix operators are applied left to right and bind more tightly than prefix operators which are applied right to left and bind more tightly than infix operators. The latter are organized in an (irreflexive) *partial precedence order* (i. e., an irreflexive, transitive, and asymmetric relationship `stronger` with an inverse relationship `weaker`) containing *operator classes* (i. e., sets of operators with `equal` precedence). Furthermore, infix operators may be declared left- or right-associative to express that an operator appearing earlier in an expression binds stronger resp. weaker than one of the same operator class appearing later in the expression.
After the application of postfix and prefix operators (which can be identified simply by their syntactic position) and, if appropriate, the recursive evaluation of parenthesized subexpressions, the remaining expression consists of an alternating sequence of operands and infix operators. In order to get successfully parsed, such an expression must contain either no operator at all or a *unique weakest operator*, i. e., exactly one operator binding weaker than all other operators of the expression. Furthermore, the two subexpressions resulting from splitting the expression at this operator must fulfill the same rule recursively. Otherwise, the expression is rejected as being ambiguous. In such a case, the programmer might either use parentheses for explicit grouping or declare additional precedence relationships to resolve the conflict.
Parsing such an expression and testing it for ambiguity can be done efficiently using a simple push-down automaton: Operands and infix operators are processed from left to right and pushed onto a stack. Before an operator is pushed, it is checked whether the previous operator on the stack (if any) binds stronger than the current operator; if so, it is replaced, together with its operands, by a new compound operand, and the check is repeated. If, after these reductions, the previous operator on the stack (if any) and the current operator are incomparable, the expression is ambiguous.

3 Unary and Binary Operators

Exponentiation. The following operator declaration introduces a new binary, right-associative operator `^^` that binds stronger than the built-in multiplicative operators:

```
new operator ^^ right stronger *;
```

Since the multiplicative operators bind in turn stronger than the built-in additive operators, and because the precedence relationship is transitive, the new operator binds stronger than, e. g., `+`, too. Therefore, an expression such as `a + b ^^ c ^^ d * e` will be interpreted as `a + ((b ^^ (c ^^ d)) * e)`, while `x ^^ y ->* z` (where `->*` is a built-in operator binding stronger than `*`, too) is rejected as ambiguous since `^^` and `->*` are incomparable. On the other hand, `p ^^ q * r ->* s` is successfully parsed as `(p ^^ q) * (r ->* s)` since `^^` and `->*` both bind stronger than `*`.
To define the meaning of `x ^^ y`, a corresponding operator function `operator^^` taking two arguments is defined which computes, e. g., the value of `x` raised to the power of `y` (using the predefined library function `pow`):

```
double operator^^ (double x, double y)
{ return pow(x, y); }
```

Because a binary operator cannot only be applied infix, but also prefix, it is possible to define a separate meaning for that case by defining an additional operator function taking only one argument. For example, the following function defines the meaning of `^^x` as the value of e (the base of the natural logarithm) raised to the power of `x`:

```
double operator^^ (double x) { return exp(x); }
```

Container Operators. To introduce a new unary operator `#`[1] which conveniently returns the size (i. e., number of elements) of an arbitrary container object `x` of the C++ standard library (or in fact any object that possesses a parameterless `size` member function), the following declarations will suffice:

```
new operator # unary;

template <typename T>
int operator# (const T& x, int postfix = 0)
{ return x.size(); }
```

By defining the operator function `operator#` as a function template, the operator is basically applicable to objects `x` of any type `T`.[2] If `T` does not declare a member function

[1] Since # denotes a special symbol for the C++ preprocessor when used at the beginning of a line or in the replacement text of a macro definition, this operator must not be used at these places. However, the "number sign" # is such a frequently used symbol to denote cardinalities that this restriction appears acceptable, especially since the usage of macros is strongly discouraged in general.

`size`, however, the corresponding template instantiation will be rejected by the compiler. By giving the function an optional second parameter of type `int`, it can be called with either one or two arguments, i. e., it simultaneously defines the meaning of # for prefix (one argument) and postfix applications (additional dummy argument of type `int`).

Even though it is possible in principle to define completely different meanings for prefix and postfix applications of the same unary operator, care should be exercised in practice to avoid confusion. To give an example, where different, but related meanings make sense, consider the following operator `@` which returns the first or last element of a container `x` when applied prefix (`@x`) or postfix (`x@`), respectively:[3]

```
new operator @ unary;

template <typename T>
typename T::value_type operator@ (const T& x)
{ return x.front(); }

template <typename T>
typename T::value_type operator@ (const T& x, int post)
{ return x.back(); }
```

4 Flexary Operators

Average Values. The following operator `avg` computes the average of two `double` values `x` and `y`:

```
new operator avg left stronger + weaker *;

double operator avg (double x, double y)
{ return (x + y)/2; }
```

When applied to three values `x avg y avg z`, however, the result is equivalent to `(x avg y) avg z` (because the operator is declared left-associative) which is usually different from the overall average value of `x`, `y`, and `z`. To avoid such accidental misinterpretations, it would be more reasonable to define the operator non-associative causing the expression `x avg y avg z` to be rejected due to ambiguity.
Alternatively, `avg` could be interpreted as a *flexary operator*, i. e., an operator accepting conceptually any number of operands concatenated by infix applications of the operator.

[2] Defining the type of `x` as `const T&` instead of just `T` is a common C++ idiom expressing that `x` is passed by reference (symbol `&`) to avoid expensive copying of the whole container while at the same time not allowing the function to change it (keyword `const`).

[3] `typename T::value_type` denotes the type `value_type` declared inside the container type `T`, i. e., the container's element type.

For that purpose, the above operator function `avg` is replaced by the following definitions which do not directly compute the average value of their arguments, but rather collect the necessary information (number of values and sum of all values processed so far) in an auxiliary structure of type `Avg`:

```
struct Avg {
    int num; double sum;
    Avg (int n, double s) : num(n), sum(s) {}
};
Avg operator avg (double x, double y) {
    return Avg(2, x + y);
}
Avg operator avg (Avg a, double z) {
    return Avg(a.num + 1, a.sum + z);
}
```

Additionally, a pseudo operator function `operator...` (where `...` is not a meta-symbol in the text denoting an omission, but rather a real C++ token) is defined which converts this intermediate information to the actual average value:

```
double operator... (Avg a) { return a.sum / a.num; }
```

This pseudo operator function is called automatically for every expression or subexpression containing user-defined operators, whenever all operators of a particular precedence level have been applied, before operators of the next lower precedence level will be applied. For example, if the operator `avg` is defined as above (i. e., left-associative with precedence between `+` and `*`), the expression `a*b avg c/d avg e%f + g avg h` (with double values `a` to `h`) is equivalent to

```
operator...(operator avg(operator avg(a*b, c/d), e%f))
+ operator...(operator avg(g, h))
```

i. e., it computes the sum of the average value of `a*b`, `c/d`, and `e%f` (`e` modulo `f`) and the average value of `g` and `h`.
Because the compiler actually does not know whether an infix operator shall be interpreted as a normal binary operator (which does neither need nor want the call to `operator...`) or as a flexary operator (which needs it), the calls are actually always inserted as described above. Furthermore, the function is predefined as the identical function

```
template <typename T>
inline T operator... (T x) { return x; }
```

for any argument type `T` to make sure that it has actually no effect on the evaluation of the expression, unless it has been specialized for a particular type `T` such as `Avg` above.

Chainable Comparison Operators. Comparison operators are another source of potential misinterpretations, at least for novice programmers. While the C++ expression `a < b`

corresponds exactly to the mathematical term $a < b$, the meaning of the expression `a < b < c` is quite different from its mathematical counterpart $a < b < c$, the latter meaning $a < b$ *and* $b < c$. The former is actually interpreted as `(a < b) < c`, which compares the Boolean-valued result of comparing `a` and `b` with `c`. In many programming languages, this will lead to a compile time error since Boolean values and numbers cannot be compared to each other. In C++, however, the Boolean values `true` and `false` are implicitly converted to the integer values `1` and `0`, respectively, when necessary causing the expression `a < b < c` to be actually well-defined, but probably not producing the desired result.
Because "chained" comparisons such as $a < b < c$ or $0 \leq i < n$ are occasionally useful and more convenient than their logical expansions (such as $0 \leq i$ *and* $i < n$), one might want to define corresponding operators in a programming language. Similar to the `avg` operator above, such operators must not only return a Boolean value representing the result of the current comparison, but also the value of their right operand which might be needed as the left operand of the following operator, too. This can again be achieved by introducing an appropriate auxiliary structure:

```
template <typename T>
struct Cmp {
    bool res; T val;
    Cmp (bool r, T v) : res(r), val(v) {}
};
template <typename T>
bool operator... (Cmp<T> c) { return c.res; }

new operator .<. left stronger = weaker ||;

template <typename T>
Cmp<T> operator.<. (T x, T y) {
    return Cmp<T>(x < y, y);
}
template <typename T>
Cmp<T> operator.<. (Cmp<T> c, T z) {
    return Cmp<T>(c.res && c.val < z, z);
}
// Likewise for operators  .<=.  .>.  .>=.  .==.  .!=.
```

Now, an expression such as `a .<. b .<. c` is indeed equivalent to `a < b && b < c`.

5 Operators with Lazily Evaluated Operands

The built-in operators `&&` and `||` expressing logical conjunction and disjunction, respectively, are special and different from all other built-in operators (except the even more

special ternary `?:` operator) in that their second operand is evaluated *conditionally* only when this is necessary to determine the value of the result. If these (or any other) operators are overloaded, this special and sometimes extremely useful property is lost, because an application of an overloaded operator is equivalent to the call of an operator function whose arguments (i. e., operands) are unconditionally evaluated before the function gets called.
Therefore, it is currently impossible to define, e. g., a new operator `=>` denoting logical implication which evaluates its second operand only when necessary, i. e., `x => y` should be *exactly* equivalent to `!x || y`. To support such operator definitions, the concept of *lazy evaluation* well-known from functional languages is introduced in a restricted manner: If an operator is declared `lazy`, its applications are equivalent to function calls whose arguments do not represent the *evaluated* operands, but rather their *unevaluated* code wrapped in *function objects* (closures) which must be explicitly invoked inside the operator function to cause their evaluation on demand. The type of such a function object is `lazy<T>` if `T` is the type of the evaluated operand.
Using this feature, the operator `=>` can indeed be defined and implemented as follows:

```
new operator => left equal || lazy;

bool operator=> (lazy<bool> x, lazy<bool> y) {
    return !x() || y();
}
```

Because the second operand of the built-in operator `||` is evaluated conditionally, the invocation `y()` of the second operand `y` of `=>` is executed only if the invocation `x()` of the first operand `x` returns `true`.

To keep the declaration of lazy operators simple and general, it is not possible to mix eagerly and lazily evaluated operands, i. e., *all* operands are either evaluated eagerly (before the operator function is called) or lazily (if the operator is declared `lazy`). However, by invoking a function object representing an operand immediately at the beginning of the operator function, the behaviour of an eagerly evaluated operand can be easily achieved.
Because an operand function object can be invoked multiple times, operators resembling iteration statements can be implemented, too, e. g.:

```
new operator ?* left weaker = stronger , lazy;

template <typename T>
T operator?* (lazy<bool> cond, lazy<T> body) {
    T res = T();
    while (cond()) res = body();
    return res;
}
```

Using operators to express control structures might appear somewhat strange in a basically imperative language such as C++. However, C++ already provides built-in operators

corresponding to control structures, namely the binary comma operator expressing sequential execution of subexpressions similar to a statement sequence and the ternary `?:` operator expressing conditional execution similar to an if-then-else statement. Therefore, introducing operators similar to iteration statements is just a straightforward and logical consequence. To give a simple example of their usage, the greatest common divisor of two numbers `x` and `y` can be computed in a single expression using the well-known Euclidian algorithm:

```
int gcd (int x, int y) {
    return (x != y) ?* (x > y ? x -= y : y -= x), x;
}
```

The possibility to express control structures with user-defined operators might appear even more useful when control flows are needed which cannot be directly expressed with the built-in operators or statements of the language. For example, operators `unless` and `until` might be defined to express conditional and iterative executions, respectively, where the condition is specified as the second operand. Using some C++ "acrobatics" (i. e., defining one operator to return an auxiliary structure that is used as an operand of another operator), it is even possible to define operator combinations (sometimes called distfix or mixfix operators) such as `first`/`all`/`count` – `from` – `where` which can be used as follows to express "database queries" resembling SQL [MS02]:

```
struct Person {
    string name;
    bool male;
    ......
};

set<Person> db; // Or some other standard container.
Person p;

Person ch = first p from db where p.name == "Heinlein";
set<Person> men = all p from db where p.male;
int abcd = count p from db where "A" .<=. p.name .<. "E";
```

Writing equivalent expressions with C++ standard library algorithms such as `find_if` or `count_if` would require to write an auxiliary function for every search predicate because the standard building blocks for constructing function objects (such as predicates, binders, and adapters, cf. [St00]) are not sufficient to construct them.

6 Conclusion

The paper has presented the basic concepts of C+++, an extension of C++ allowing the programmer to define new operator symbols with user-defined priorities. Even though the

basic idea of this approach dates back to at least ALGOL 68 [Wi75], it has not found widespread dissemination in mainstream imperative programming languages. Compared with Prolog and modern functional languages, which support the concept in principle, the approach presented here offers a more flexible way to specify precedence relationships, the additional concept of flexary operators (which is rather dispensable in these languages as their effect can be achieved in a similarly convenient manner with unary operators applied to list literals), and the concept of lazily evaluated operands in an imperative language (which is of course nothing special in functional languages). It might be interesting to note that this latter concept has already been present in ALGOL 60 [Na63], known as the (in)famous "call by name." While this is indeed not well-suited as a general parameter passing mechanism, the examples of Sec. 5 should have demonstrated that the basic principle is useful when applied with care because it opens the door to implement user-defined control structures and thus might be considered a step towards generally extensible programming languages.

References

[CM94] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[GR89] A. Goldberg, D. Robson: *Smalltalk-80. The Language*. Addison-Wesley, Reading, MA, 1989.

[He04] C. Heinlein: *Concept and Implementation of C+++, an Extension of C++ to Support User-Defined Operator Symbols*. Nr. 2004-02, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, July 2004. http://www.informatik.uni-ulm.de/pw/berichte/

[MS02] J. Melton, A. R. Simon: *SQL:1999. Understanding Relational Language Components*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[MTH90] R. Milner, M. Tofte, R. Harper: *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[Na63] P. Naur (Ed.): "Revised Report on the Algorithmic Language ALGOL 60." *Numerische Mathematik* 4, 1963, 420–453.

[Pe03] S. Peyton Jones (ed.): *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press, Cambridge, 2003.

[St00] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[Tu01] S. Tucker Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder (eds.): *Consolidated Ada 95 Reference Manual with Technical Corrigendum 1* (ANSI/ISO/IEC-8652:1995 (E) with COR. 1:2000). Lecture Notes in Computer Science 2219, Springer-Verlag, Berlin, 2001.

[Wi75] A. van Wijngaarden et al. (Eds.): "Revised Report on the Algorithmic Language ALGOL 68." *Acta Informatica* 5, 1975, 1–236.