Concept and Implementation of C++, an Extension of C++ to Support User-Defined Operator Symbols and Control Structures

Christian Heinlein

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

Abstract. The first part of this report presents the 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 *fixary operator combinations* consisting of a sequence of associated operator symbols to connect a fixed number of operands as well as *flexary operators* connecting any number of operands are supported. Finally, operators with *lazily evaluated operands* are supported which are particularly useful to implement new kinds of *control structures*, especially as they accept whole blocks of statements as operands, too. In the second part of the report, the implementation of C++ by means of a "lazy" precompiler for C++ is described in detail.

1 Introduction

Programming languages such as Ada [12] and C++ [10, 4] 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 [3], Prolog [1], and modern functional languages such as ML [7] and Haskell [9], 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 0 and 9). 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 report (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, *fixary operator combinations* consisting of a sequence of associated operator symbols to connect a fixed number of operands as well as *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. These operators are particularly useful to implement new kinds of *control structures*, especially as they accept whole blocks of statements as operands, too.

Sec. 2 describes the basic features of C+++, an extension of C++ supporting the introduction of new operator symbols. Secs. 3, 4, 5, and 6 illustrate these with numerous examples, demonstrating in particular the advances mentioned before. Sec. 7 describes the implementation of C+++ by means of a "lazy" precompiler for C++, including some minor limitations of the approach. Finally, Sec. 8 concludes the report with a discussion of related work.

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. 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 and appropriate universal character names are treated as letters) or sequences of one or more *operator 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 analysis 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.)

Just like other identifiers, new operator symbols which are identifiers are recognized as such only if they are not part of a larger identifier (or other token). For example, an operator `abc` is not recognized as such in the input `abcd` (part of a larger identifier) nor in the input `0x123abc` (part of a hexadecimal integer literal).

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 is 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 pushdown 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. (Sec. 7.4.3 describes this in more detail.)

The initial precedence order, which contains operator classes for all built-in C++ operators and therefore is actually a total order, can be extended freely as long as new declarations do not introduce any conflicts. (For example, declaring a new operator to bind stronger than `*`, but weaker than `+`, would be illegal, since `*` already binds stronger than `+`.) In particular, it is possible to insert new operators between adjacent classes of built-in operators (e. g., between `*` and `+`) and at the “ends” of the spectrum of built-in operators, i. e., to introduce operators binding stronger than `->*` (the strongest regular C++ infix operator) or weaker than `,` (comma, the weakest C++ infix operator). In analogy to standard C++, however, expressions used as function arguments or variable initializers (using the `=` notation for initialization) must not contain operators weaker than comma or assignment operators, respectively, except when nested in parentheses.

3 Unary and Binary Operators

3.1 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 of the usual arithmetic conversions, the new operator cannot only be applied to `double`, but also to `int` values, e. g., `2 ^^ 10`. To make sure, however, that the result of such an application is also of type `int`, an overloaded variant of the operator function can be supplied:

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

3.2 Container Operators

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

```
new operator # unary;

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

By defining the operator function `operator#` as a function template, the operator is basically applicable to objects `c` of any type `C`.[2] If `C` does not declare a member function `size`, however, the corresponding template instantiation will be rejected by the compiler.

[1] Because the operator declaration introduces # as an operator symbol, it will be treated as such in the remaining input. Therefore, it can no longer be used as a C++ preprocessor symbol afterwards. Since `#include` directives are typically placed at the very beginning of a translation unit and the use of other preprocessing directives (`#define` in particular) is highly discouraged in general, this appears to be an acceptable restriction.

[2] Defining the type of `c` as `const C&` instead of just `C` is a common C++ idiom expressing that `c` 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`).

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 `c` when applied prefix (`@c`) or postfix (`c@`), respectively:[3]

```
new operator @ unary;

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

template <typename C>
typename C::value_type operator@ (const C& c, int postfix) {
    return c.back();
}
```

4 Fixary Operator Combinations

4.1 Principle

In addition to ordinary unary and binary operators, C++ provides a special ternary operator combination `?:` to express conditional execution inside an expression. When viewing `?` and `:` as separate binary operators, their binding properties are equal to those of assignment operators, except that their middle operand (between `?` and `:`) might be *any* expression containing in particular assignment and comma operators, even though the latter bind weaker than `?` and `:`. Conceptually, this exceptional rule can be eliminated and reduced to the normal precedence rules by defining that the subexpression between the two operators is always implicitly *grouped*, as if it would be surrounded by parentheses. Furthermore, a `:` operator must not occur without a preceding `?` operator (and vice versa).

This principle is generalized in C+++ to so-called *fixary operator combinations* (sometimes also called distfix or mixfix operators): By declaring that one operator must only appear `after` another operator in an expression, any subexpression between these operators is implicitly grouped and it is checked that the former operator is not used without the latter. Declaring that an operator `y` must only appear *after* an operator `x`, turns out to be more flexible than declaring the opposite, i. e., that `x` must *only* appear before `y`, because in the first case `x` might well appear without `y`, but not vice versa. To make the concept even more flexible, it is possible to specify multiple operators `x` which might precede `y`; in such a case, `y` must only appear after one of these operators, and the subexpression between `y` and the nearest such operator is implicitly grouped.

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

4.2 Inserting Into a Container

The following declarations:

```
new operator INSERT unary;
new operator INTO after INSERT right equal =;

template <typename T>
T operator INSERT (T x) { return x; }

template <typename C>
void operator INTO (typename C::value_type x, C& c) { c.push_back(x); }
```

define an operator combination `INSERT − INTO` which can be used as follows to insert the result of assigning `1` to `x` into the vector `c`:

```
int x; vector<int> c;
INSERT x = 1 INTO c;
```

If `INTO` would have been declared as a normal binary operator without an `after` clause, the last line of the example would get parsed as `(INSERT x) = (1 INTO c)` according to the normal rules. By declaring `INTO after INSERT`, however, the subexpression between these operators is implicitly grouped and afterwards the complete expression gets parsed as `(INSERT (x = 1)) INTO c`.[4]

By introducing another operator `BEFORE`:

```
new operator BEFORE after INTO right equal =;

template <typename C>
pair<C*, typename C::iterator>
operator BEFORE (C& c, typename C::iterator i) {
    return pair<C*, typename C::iterator>(&c, i);
}

template <typename C>
void operator INTO
(typename C::value_type x, pair<C*, typename C::iterator> ci) {
    C& c = *ci.first;
    typename C::iterator i = ci.second;
    c.insert(i, x);
}
```

it is possible to generalize the previous example as follows to insert the result of assigning `1` to `x` into the vector `c` before the position determined by the iterator `i` (whose type is `typename C::iterator`):

```
int x; vector<int> c; vector<int>::iterator i;
INSERT x = 1 INTO c BEFORE i;
```

Because `INTO` and `BEFORE` have been declared right-associative, the last expression gets parsed as `(INSERT (x = 1)) INTO (c BEFORE i)`. Therefore, the result of evaluating `c BEFORE i` (which is just a pair consisting of a pointer to `c` and `i`) becomes the second operand of `INTO` which performs the actual `insert` operation.

It should be noted that the previous example (without using `BEFORE`) is still valid because `INTO` can still be used without `BEFORE`, but not vice versa. This is exact the increased flexibility mentioned

[4] Note that it is possible in principle to use `INSERT` without `INTO` (e. g., `INSERT 1`), which is semantically useless, however. On the other hand, using `INTO` without `INSERT` would be erroneous.

above that is gained by declaring `BEFORE after INTO` instead of the opposite `INTO before BEFORE` (which is actually not supported) that would forbid the use of `INTO` without a succeeding `BEFORE`.

Finally, by extending the declarations of `INTO` and `BEFORE` as follows:

```
new operator INTO after BEFORE;
new operator BEFORE after INSERT;
```

it is allowed to use `INTO` either after `INSERT` (original declaration) or after `BEFORE` (extended declaration) and to use `BEFORE` either after `INTO` (original declaration) or after `INSERT` (extended declaration). Together with appropriate overloadings of the involved operator functions, insertions can now be done in any of the following ways (where the latter two are expected to be equivalent):

```
INSERT x INTO c;
INSERT x INTO c BEFORE i;
INSERT x BEFORE i INTO c;
```

4.3 User-Defined Control Structures

User-defined control structures, such as `FOREACH x IN c DO ...`, are another typical use of fixary operator combinations. However, since their implementation requires the additional concept of lazily evaluated operands, their discussion is postponed until Sec. 6.

5 Flexary Operators

5.1 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. 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. By declaring the predefined function `inline`, the compiler is instructed to expand its calls in place, which in this case actually means to eliminate them to avoid unnecessary run time penalties.

5.2 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 $a < b \text{ and } b < c$. 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 $0 \le i < n$ are occasionally useful and more convenient than their logical expansions (such as $0 \le i \text{ and } i < n$), 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 .<. 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`, while `0 .<=. i .<. n` is equivalent to `0 <= i && i < n`, i. e., it is possible to mix flexary operators of equal precedence.

6 Operators with Lazily Evaluated Operands

6.1 Logical Implication

The built-in operators `&&` and `||` expressing logical conjunction and disjunction, respectively, are special and different from all other built-in operators (except the already mentioned ternary `?:` operator combination) 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`. Of course, this behaviour could be made more explicit by rephrasing the body of the operator function with an explicit `if` statement:

```
bool operator=> (lazy<bool> x, lazy<bool> y) {
    if (x()) return y();
    else return true;
}
```

6.2 User-Defined Control Structures

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 combination 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, an operator `UNLESS` might be defined that executes its first operand unless the evaluation of its second operand yields true (i. e., the latter is evaluated before the former):

```
new operator UNLESS left equal ?* lazy;

template <typename T>
T operator UNLESS (lazy<T> body, lazy<bool> cond) {
    T res = T();
    if (!cond()) res = body();
    return res;
}
```

Using the additional concept of fixary operator combinations (cf. Sec. 4), it is also possible to define a `REPEAT` − `UNTIL` operator combination:

```
new operator REPEAT unary;
new operator UNTIL after REPEAT left equal ?* lazy;

template <typename T>
T operator REPEAT (T body) { return body; }

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

6.3 Database Queries

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 such as `FIRST`/`ALL`/`COUNT` − `FROM` − `WHERE` which can be used as follows to express "database queries" resembling SQL [6]:

```
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. [10]) are not sufficient to construct them.

6.4 Blocks as Operands

In contrast to normal operators with eagerly evaluated operands where operands of type `void` do not make sense, it is possible and quite useful to use expressions of type `void` as lazily evaluated operands. If, for example, `f` happens to be a function with result type `void`, an expression such as `i > 0 ?* f(i--)` obviously makes sense.

It is even possible (with some restrictions, however) to use *blocks*, i. e., sequences of statements and declarations enclosed in curly brackets, as lazily evaluated operands of type `void`. This is again useful in combination with operators such as `?*` or `REPEAT` − `UNTIL` expressing control structures:

```
REPEAT {
    // Do something complex
    // that is hard to express as a single expression.
} UNTIL (/* some condition */);
```

The precise rules regarding blocks as operands are as follows:

- A block is treated as an operand if and only if it appears between a new operator and the nearest subsequent semicolon (on the same nesting level of curly brackets), and neither the operator nor the block are enclosed in round or square brackets. Furthermore, the enclosing expression must be an expression *statement*.
  This rule is necessary to unambiguously distinguish blocks used as operands from normal blocks used as statements.
- If such a block is not immediately followed by a user-defined operator having an `after` clause (cf. Sec. 4.1), a semicolon is implicitly inserted after the block which terminates the enclosing expression.
- Before such a block, a binary lazy pseudo-operator whose operator function name is `operator{}` is inserted implicitly.
  The prefix application of this operator is predefined as the identical function (note that a block has type `void`):

  ```
  lazy<void> operator{} (lazy<void> x) { return x; }
  ```

  This is equivalent in effect to saying that the operator is inserted only if the block is not immediately preceded by another operator.
  Infix applications of this operator can be defined as needed (cf. the examples below).
  There are no predefined precedence relationships for this operator, because they are not needed for its typical applications, but it is possible in principle to declare such relationships.

Using blocks as operands, it is possible to write expressions which look exactly like C++ control structures. For example, given the following definitions:

```
// Operators foreach and in.
new operator foreach unary;
new operator in weaker = stronger ,;

// operator in combines a variable v of type V
// and a container c of type C into a pair.
template <typename V, typename C>
pair<V*, const C*> operator in (V& v, const C& c) {
    return pair<V*, const C*>(&v, &c);
}

// operator foreach returns such a pair vc unchanged.
template <typename V, typename C>
pair<V*, const C*> operator foreach (pair<V*, const C*> vc) {
    return vc;
}

// operator {} combines such a pair vc and a block b.
// It iterates through the container c,
// binds the variable v to each element in turn,
// and executes block b for it.
template <typename V, typename C>
void operator {} (lazy< pair<V*, const C*> > vc_, lazy<void> b) {
    pair<V*, const C*> vc = vc_();
    V& v = *vc.first;
    const C& c = *vc.second;
```

```
    for (typename C::const_iterator i = c.begin(); i != c.end(); i++) {
        v = *i;
        b();
    }
}
```

it is possible to write code like this:

```
vector<int> c; int i;
foreach (i in c) { cout << i << endl; }
```

If a block used as an operand executes a jump statement (`break`, `continue`, `goto`, or `return`) that transfers control out of the block, the operand is terminated as if it had thrown an exception (of an unknown type) that also terminates the operator function and all possibly active intermediate functions called directly or indirectly from it. This includes the process of "stack unwinding" [4], i. e., calling destructors for all "automatic objects" (objects with automatic storage duration) constructed during their execution. Then, instead of continuing execution after the point where the operator function has been called, control is transferred to the destination of the jump statement.

For example, `return` can be used in the usual way to terminate a `foreach` loop prematurely:

```
// Does container c contain value x?
template <typename C>
bool contains (const C& c, typename C::value_type x) {
    typename C::value_type i;
    foreach (i in c) {
        if (i == x) return true;
    }
    return false;
}
```

However, `break` and `continue` statements cannot be used inside a `foreach` loop to just break out of *this* loop or to continue with its next iteration, respectively, because they will terminate or continue an enclosing normal loop (or `switch` statement). It is possible, however, to use exceptions for such purposes. If, for example, the implementation of `operator{}` is changed as follows:

```
template <typename V, typename C>
void operator {} (lazy< pair<V*, const C*> > vc_, lazy<void> b) {
    pair<V*, const C*> vc = vc_();
    V& v = *vc.first;
    const C& c = *vc.second;

    for (typename C::const_iterator i = c.begin(); i != c.end(); i++) {
        v = *i;
        try {
            b();
        }
        catch (bool cont) {
            if (cont) continue;
            else break;
        }
    }
}
```

it is possible to terminate the current iteration of the `foreach` loop by throwing an exception of type `bool` whose actual value indicates whether the whole loop shall continue or not.

Other types of exceptions might be used to implement more sophisticated iteration control. For example, by throwing an `int` value `i` the iteration body `b` could indicate that the iteration shall continue with the `i`-th next element of the container.

7 Implementation of C+++

7.1 Basic Approach

The language extensions to C++ described in this report have been implemented by a “lazy” precompiler. Here, the term “lazy” has nothing to do with lazy or eager evaluation of expressions, but shall describe the fact that the precompiler does not “eagerly” do a complete parse of its input (which is impossible for C++ without doing a complete semantic analysis), but rather copies most of it through unchanged without actually “understanding” it. (Therefore, it might also be called a “stupid” precompiler.) Only when encountering particular keywords, special tokens, or combinations thereof, it performs a “local” parse of their context and possibly a corresponding code transformation. These “sensitive” token sequences and their processing are described in the sequel.

7.2 Operator Declarations

If the keywords `new` and `operator` appear in juxtaposition, the subsequent input up to the next semicolon is parsed as an operator declaration according to the following EBNF grammar (where quoted strings represent tokens of the lexical analyzer):

```
opdecl:
    "new" "operator" newopsym [ "unary" ] { "after" opsym }
    { "left" | "right" | "stronger" opsym | "weaker" opsym }
    [ "lazy" ] ";"
```

Here, `opsym` represents either built-in operators or new operator symbols introduced earlier. On the other hand, `newopsym` might represent a new operator symbol (such as `+++`) which consists of multiple tokens (`++` and `+` in this case) since it is not yet known to be a single token. The precise rule is that it might be either an identifier as defined in the base language C++ (i. e., a sequence of letters and digits starting with a letter, where the underscore character and appropriate universal character names are treated as letters) or a sequence of one or more *operator character tokens*, i. e., tokens consisting solely of so-called operator characters. These are all characters of the basic character set except white space (including comments), letters, digits, and quotation marks, i. e., actually the following:

```
{ } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \
```

Immediately after the `newopsym` has been read, its complete character sequence is declared as a new single token to the lexical analyzer causing it to be recognized as such in the remaining input.

The lexical analyzer is basically identical to a standard C++ scanner which recognizes so-called preprocessing tokens [4] (and, of course, white space and comments). The only difference is the fact that the set of operators which are recognized as single tokens is not hard-coded into the scanner, but maintained in an extensible table. Whenever a sequence of operator characters is found in the input, it is broken into one ore more tokens according to this table.

7.3 Operator Function Names

If a new operator symbol (i. e., one that has been previously declared by an operator declaration) appears after the keyword `operator` (possibly with intervening white space or comments), e. g., `operator+++`, this is replaced by a unique function name derived from the operator symbol, e. g.,

`__plus_plus_plus` (where the leading underscores indicate that this is an "implementation-internal" name).

This obviously transforms operator function declarations and definitions to normal function declarations and definitions which can be processed by a C++ compiler, but it also transforms operator function names appearing elsewhere (i. e., as function pointer or reference values) into normal function names.

7.4 Expressions Containing New Operator Symbols

7.4.1 Basic Principle

In C++, it is very difficult to recognize expressions in general, because they might appear in various places, e. g., as expression statements, initializers of variables, member initializers of constructors, etc. Furthermore, without additional knowledge it is impossible to distinguish some kinds of expressions from declarations. For instance, `a * b` looks like a multiplication of `a` and `b` at first glance, but it might actually be a declaration of `b` as a pointer to `a` if `a` is known to be a type name in the current scope.

Therefore, the C+++ precompiler does not even attempt to recognize *all* expressions in its input, but only those containing new operator symbols. The latter will be analyzed as described below, and the new operators will be replaced by corresponding function calls (cf. Sec. 7.4.3). All other expressions, i. e., those containing only built-in operators, simply get passed through unchanged and will be correctly interpreted by the C++ compiler, even if some of their operators have been overloaded.

As the example `a * b` demonstrates, it would even be dangerous for the precompiler to transform *all* operators to corresponding function calls uniformly, because the transformed code `__asterisk(a, b)` is of course no longer a declaration. Furthermore, if `a` and `b` were compile-time constants, `a * b` would also be such a constant, which could be used, e. g., as an array bound or a case expression. Again, the transformed expression `__asterisk(a, b)` would not be an equivalent substitute in these (and some other) contexts, because a function call expression is never regarded as a constant expression, even if the function `__asterisk` would be predefined for all types `T` as follows:

```
template <typename T>
T __asterisk (T x, T y) { return x * y; }
```

Expressions containing new operators are simply recognized by these new operator symbols, i. e., whenever a token is encountered that has been declared as a new operator symbol (and this token is not preceded by the keyword `operator`, cf. Sec. 7.3), the surrounding context must be an expression. In order to correctly analyze and transform this expression, its begin and end is determined by searching backward resp. forward until a *left* resp. *right expression delimiter* is found, i. e., a token that definitely and unambiguously delimits an expression. Fortunately, the sets of these tokens are rather small and well-defined.

7.4.2 Expression Delimiters

The set of left expression delimiters contains:

`;` end of preceding statement

`:` end of preceding (case) label
(Sec. 7.5 explains how this is distinguished from a `:` operator.)

`(` begin of subexpression or (member) initializer
(Subexpressions surrounded by parentheses are transformed separately from their surrounding expression where the former are treated as simple operands.)

`[` begin of array bound or subscript
(Normally, array bounds must be constant expressions which cannot contain user-defined opera-

tors; in a *new-expression*, however, the first array bound might be an arbitrary expression which might contain user-defined operators.)

`{` begin of block

`}` end of preceding block

`return`
`else`
`do`
Keywords which might be followed by an expression or expression statement.

The set of right expression delimiters contains:

`;` end of expression statement

`)` end of subexpression

`]` end of array bound or subscript

While searching for these expression delimiters, complete bracket expressions `(...)` and `[...]`, i. e., token sequences containing balanced opening and closing brackets, are treated as single units. In particular, their brackets do not constitute expression delimiters.

To make sure that an appropriate left expression delimiter is always found, substatements of conditional and iteration statements (`if`, `switch`, `while`, `for`, `do`) which are not surrounded by curly brackets are automatically augmented with these (which does not introduce any semantic difference).

If a block `{...}` is encountered while searching for a right expression delimiter, it is treated as a block operand, and the pseudo-operator `{}` is inserted before. If the block is not immediately followed by a user-defined operator having an `after` clause, a semicolon is inserted after the block which terminates the enclosing expression (and will cause a syntax error in the generated C++ code if the expression is enclosed in round or square brackets).

7.4.3 Transformation Algorithm

Having identified begin and end of an expression containing user-defined operators, the expression is transformed as follows:

1. If it begins with one or more operators, these must be prefix operators (otherwise the expression is erroneous) which are saved on a stack.
2. The subsequent tokens up to the next operator (or the end of the expression) make up the first operand which is saved on the stack, too.
   (Operands might consist of several tokens, e. g., function calls such as `f(x, y)`, subscript expressions such as `x[i][j]`, and the like.)
3. As long as the following tokens are postfix operators (and consequently are not infix applicable, cf. Sec. 2), they are applied to the previous operand on the stack.
   Here, applying an operator to an operand either means to simply combine them into a compound operand (if the operator is built-in, e. g., combine `x` followed by `++` into the compound operand `x++`) or to transform them into an operator function call (if the operator is user-defined, e. g., transform `x` followed by `+++` into the operator function call `__plus_plus_plus(x,0)`) that is also treated as a compound operand.
   In any case, the result of this application replaces the previous operand on the stack.
4. If any postfix operators have been applied in step 3, go back to step 2, since the operand constructed in step 3 (e. g., `__plus_plus_plus(x,0)`) might be followed by other operand tokens (such as `[i]`), which might themselves be followed by other postfix operators, etc.

5. Afterwards, all prefix operators which have been saved on the stack in step 1 will be applied to the operand on top of the stack in reverse order, finally yielding a single compound operand on top of the stack.

6. The next token, if any, must be an infix operator which is pushed onto the stack. (It might also be prefix applicable, but according to its position in the expression, it must be applied infix.)
   Before it is actually pushed, it is checked whether the previous (infix) operator on the stack (if any) binds stronger than the current operator; if so, this previous operator (which is the 2nd to top element on the stack) is applied to its operands (which are the 3rd to top and the topmost element on the stack, respectively) and the resulting compound operand replaces them on the stack.
   This check is performed repeatedly until the previous operator on the stack (if any) binds weaker than the current operator (in which case the current operator is actually pushed onto the stack) or the two operators are incomparable (in which case the expression is rejected as being ambiguous).

7. Afterwards, the whole procedure from step 1 to 6 is repeated until the expression is exhausted, and finally all infix operators remaining on the stack are successively applied to their operands.

7.5 Special C++ Operators

The operators `::`, `.` (dot), and `->` are not treated as operators by the precompiler, but rather as parts of operands, because they bind more tightly than prefix and postfix operators and therefore do not fit into the general scheme described in Sec. 2, where infix operators bind less tightly than unary operators.

The ternary operator combination `?:` is treated like a pair of operators `?` and `:` where the latter must only appear `after` the former, causing any subexpression between these operators to become automatically grouped (cf. Sec. 4.1). Furthermore, to distinguish a colon constituting an operator from colons having other meanings, in particular those following (case) labels and therefore constituting left expression delimiters, only colons for which a matching question mark is found are treated as operators.

The C++ keywords `sizeof`, `typeid`, and `delete` as well as the token sequence `delete[]` are treated just like normal prefix operators.

So-called *new-expressions* are treated as single operands in order to avoid `*` and `&` tokens appearing in their *new-type-id* [4] to be treated as operators.

In contrast to `return` *statements*, `throw` instructions are actually *expressions* with the same precedence as assignment expressions. Therefore, the keyword `throw` is actually a prefix operator with a very low precedence. In order to make it fit into the general operator scheme mentioned before, however, it is treated as an infix operator (having the same precedence as assignment operators) whose left operand is empty resp. missing.

Furthermore, since it is possible to use `throw` without an associated expression (in which case the currently handled exception is rethrown), its right operand might be empty resp. missing, too. This is the case if and only if `throw` is either the last token of an expression or it is followed by an infix operator with lower precedence. (In C++, the only such operator is the comma operator[5], but in C+++ other such operators might be defined, of course.)

[5] The case that `throw` is immediately followed by `:` is covered by the rule that the middle operand of `?:` might be any expression. In this case, `throw` is the last token of this subexpression.

7.6 Lazily Evaluated Operands

7.6.1 Non-local Gotos

The implementation of lazily evaluated operands is based on a non-local goto facility similar to the C library functions `setjmp` and `longjmp`.

There is a data type `Context` whose instances are able to store the current execution context of a process or thread, including its program counter, the base pointer of the current stack frame, the stack pointer referring to the top of the stack, and possibly some other register values. The function

```
int mark (Context& c);
```

stores the current execution context in its parameter `c` and returns zero, while the function

```
void jump (const Context& c, int v = 1);
```

restores the context saved in `c` and causes execution to continue as if the corresponding call to `mark` had just returned the value `v` (which defaults to one and must be different from zero). For that purpose, the function that has executed `mark` must still be active, i. e., the function that executes `jump` must either be the same function or one that has been called directly or indirectly from it. (In other words, `jump` can only jump "downwards" on the stack.)

Apart from different names, this functionality is identical to `setjmp` and `longjmp`. However, there is a third function

```
int markjump (Context& c, const Context& d, int v = 1);
```

that combines the functionality of `mark` and `jump` by first saving the current execution context in `c` before restoring the destination context `d`. Furthermore, in contrast to a normal `jump`, the current stack pointer is left unchanged (i. e., the one saved in `d` is ignored) causing the stack frames of subsequent function calls to be placed above the frame of the function executing `markjump` and therefore protecting the frames below from becoming overwritten. Thus, in contrast to a normal `jump`, it is possible to return to the context saved in `c` by another call to `jump` (or `markjump`) later, i. e., one can also jump "upwards" on the stack (if the terms "upwards" and "downwards" refer to the logical motion of the base pointer, no matter whether the stack actually grows "forward" or "backward" in terms of absolute addresses).

These functions can be implemented either directly in (machine-dependent) assembly code or (more portably!) by using `setjmp` and `longjmp` to implement `mark` and `jump` (which is straightforward) as well as `markjump` by using these functions in a way that is actually undefined: By executing `longjmp(e, v)` where the context `e` is a copy of `d` except for its stack pointer value which is taken from `c`, it is possible to implement the behaviour of `markjump(c, d, v)`.[6]

7.6.2 Auxiliary Types and Functions

For every lazily evaluated operand, an instance of the following template type `Operand<n>` will be allocated that contains:

- a context object `opnd` to store the execution context of the operand;
- another context object `oper` to store the execution context of the operator function (or a function called directly or indirectly from it) where the operand's evaluation is requested;

[6] Because many compilers allocate function parameters in the *current* stack frame, i. e., beneath the current stack pointer, before allocating the frame of the called function, it is actually necessary to (logically) increase the stack pointer found in `c` by the size of the frame found in `d`, i. e., the difference between the stack and base pointer values found there.
Furthermore, since `setjmp` must be executed *directly* in the function whose context shall be saved, `mark` and `markjump` must not be functions, but rather macros (which should have less common names such as `__MARK` and `__MARKJUMP` then, and which cannot have optional parameters).

- a (maximally aligned) array `val` of `n` bytes, where `n` is the size of the operand's type, to store the result of evaluating the operand.[7]

```
template <int n>
struct Operand {
    Context opnd;
    Context oper;
    union {
        double dummy;   // To force maximum alignment of val.
        char val [n];
    };
};
```

To evaluate an operand `x` and store its result in an appropriate `Operand` object `op`, the following function `eval` will be used:

```
template <typename T>
lazy<T> eval (T x, Operand<sizeof(T)>& op) {
    new(op.val) T(x);
    jump(op.oper);
}
```

It uses the so-called "placement operator new" to copy `x` (which is evaluated automatically when `eval` gets called) into the byte array `op.val` using `T`'s copy constructor. Afterwards, `jump` is used to return to the place `op.oper` where the operand's evaluation has been requested (cf. below).

When calling an operator function with lazily evaluated operands, the latter are passed as objects of the following type `lazy<T>` (which, for reasons explained in Sec. 7.6.3 below, is also used as the formal result type of `eval`, even though `eval` does not actually return a value) where `T` represents the type of the evaluated operand. Such an object is initialized with a reference `op` to an appropriate `Operand` object which is stored inside the `lazy<T>` object. It provides a definition of a parameterless function call operator that causes the operand to be evaluated and returns its value.

For that purpose, `markjump` is used to save the current execution context in `op.oper` and jump to the context saved earlier in `op.opnd` (cf. below, Sec. 7.6.3), where `eval(..., op)` will be executed to actually evaluate the operand, store its value in `op.val`, and return to the current position by executing `jump(op.oper)` (cf. above). Afterwards, the operand's value is copied into a temporary variable `x` before `T`'s destructor is called for `op.val` to complement the constructor call performed by `eval`,[8] and finally, this value `x` is returned.

```
template <typename T>
struct lazy {
    Operand<sizeof(T)>& op;
    lazy (Operand<sizeof(T)>& op) : op(op) {}

    T operator() () {
        markjump(op.oper, op.opnd);
        T* p = (T*)(op.val);
        T x = *p;
        p->~T();
```

[7] Of course, it would be more natural to directly use the operand's type `T` instead of its size `n` as a template parameter for `Operand`. However, this type is not known to the precompiler, and in standard C++ there is no way to abstractly refer to it, such as `typeof(x)` in GNU C++. On the other hand, `sizeof(x)` is available as a portable means to refer to the size of `x`'s type.

[8] Formally, these constructor and destructor calls are well-defined even for basic types such as `int`, even though they will not perform particular actions.

```
        return x;
    }
};
```

7.6.3 Basic Approach

The application of a normal user-defined operator with eagerly evaluated operands, such as `x ^^ y` (where `x` and `y` might be arbitrary subexpressions), is transformed to a call of the corresponding operator function whose arguments are the operands, i. e., `__hat_hat(x, y)`.

If the operator is declared `lazy`, as in the expression `x => y`, its operands are wrapped as follows:

```
Operand<sizeof(x)> __x;
Operand<sizeof(y)> __y;

__equal_greater(
    mark(__x.opnd) ? eval(x, __x) : __x,
    mark(__y.opnd) ? eval(y, __y) : __y
)
```

This means, that before the operator function `__equal_greater` is called, `mark` is called for every operand to save its execution context in `__x.opnd` and `__y.opnd`, respectively. Since `mark` returns zero (which is equivalent to false), `eval` will *not* be called now, but rather `__x` resp. `__y` will be passed as arguments to `__equal_greater`. To make this type-correct, `eval` has been declared above (Sec. 7.6.2) with result type `lazy<T>` if it is called with an expression of type `T`, and because this type provides a constructor accepting an object of type `Operand<sizeof(T)>`, `__x` resp. `__y` are implicitly converted (according to the rules of determining the type of an `?:` expression) to objects of type `lazy<X>` resp. `lazy<Y>` (if `X` resp. `Y` is the type of `x` resp. `y`) using this constructor.

In summary, this means that `__equal_greater` is called with objects of type `lazy<X>` resp. `lazy<Y>` containing references to the `Operand` objects `__x` resp. `__y`. If `__equal_greater` requests the evaluation of, e. g., `x` by executing `x()`, the call to `markjump` performed in `lazy<X>::operator()` (cf. above, Sec. 7.6.2) causes the call of `mark(__x.opnd)` to return a non-zero value causing `eval(x, __x)` to be called now. As described above (Sec. 7.6.2), this evaluates the operand `x` and transfers control back to its function call operator `x()` which will return the operand's value.

7.6.4 Problem and Modified Approach

The approach described so far works fine except for a subtle detail: If temporary objects are created during the evaluation of a lazily evaluated operand, their constructors will be executed at the point where the objects are created, but their destructors will be executed, according to the C++ Standard [4], at the end of the enclosing *full expression*. This point, however, is not reached immediately, because `eval` does not return regularly, but rather transforms control back to the operator function. Therefore, the destructors will be executed only after the operator function returns, which would not be a problem if the operands were evaluated exactly once. If they are evaluated multiple times, however (as is typical for operators implementing iterations), the temporary objects' constructors will be executed multiple times, too (which is correct), but their destructors will only be executed *once* at the end of the enclosing full expression (because the C++ compiler, of course, does not expect the operands to get evaluated multiple times).

Given this observation, the normal C++ rule regarding the execution of destructors must be modified to say that temporary objects constructed during the evaluation of a lazy operand are destructed at the end of this evaluation instead of at the end of the enclosing full expression. To implement this behaviour with a precompiler, i. e., without modifying the underlying C++ compiler, it is necessary to transform every lazy operand to an independent full expression. This is achieved by modifying the transformation scheme shown above (Sec. 7.6.3) as follows:

```
Operand<sizeof(x)> __x;
if (mark(__x.opnd)) { eval(x, __x); jump(__x.oper); }

Operand<sizeof(y)> __y;
if (mark(__y.opnd)) { eval(y, __y); jump(__y.oper); }

__equal_greater(
    false ? eval(x, __x) : __x,
    false ? eval(y, __y) : __y
)
```

The calls to `mark` returning zero have been replaced by the Boolean value `false` which has the same effect on the evaluation of the `?:` expressions, and even though `eval` will never get executed here, its result type is still needed by the C++ compiler to correctly determine the type of these expressions. The original calls to `mark` with their subsequent calls to `eval` when the former returns non-zero have been moved before the enclosing full expression (which might be larger than the call to `__equal_greater` shown above) to make them full expressions of their own. Furthermore, the calls to `jump` which have been part of `eval` above (Sec. 7.6.2) are performed as separate statements after the execution of `eval` here to make sure that the end of these full expressions (i. e., the "semicolon" after the call to `eval`) is actually reached. Therefore, `eval` is now simply defined as follows:

```
template <typename T>
lazy<T> eval (T x, Operand<sizeof(T)>& op) {
    new(op.val) T(x);
}
```

7.6.5 Operands of Type `void`

As has been mentioned in Sec. 6.4, a lazily evaluated operand might have type `void`, which would lead to syntax errors in the above code since an expression of type `void` can neither be used as an operand of the `sizeof` operator nor as a function argument of `eval`. To remedy this problem, an auxiliary type `Void` (with an upper-case `V`) is introduced, which acts as a "right neutral element" of the comma operator:

```
struct Void {};

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

Furthermore, the expressions `x` and `y` in the above code are replaced by `(x, Void())` and `(y, Void())`, respectively, which, due to this definition, does not have any effect on their meaning, except if their type is `void`: In that case, the overloaded comma operator is not applicable (because it cannot have a parameter of type `void`), and therefore, the built-in definition of the comma operator is used, yielding an expression of type `Void`. To make sure, however, that an operator function accepting lazy operands of type `void` actually receives these as arguments of type `lazy<void>`, an overloaded definition of `eval` for the type `Void` is provided whose formal result type is `lazy<void>`:

```
lazy<void> eval (Void, Operand<sizeof(Void)>&) {}
```

Because instantiating this type from the general template `lazy<T>` shown earlier (Sec. 7.6.2) would be erroneous (because `sizeof(void)` is undefined and the function call operator must not return anything), it is defined as an explicit template specialization as follows:

```
template <>
struct lazy<void> {
    Operand<sizeof(Void)>& op;
    lazy (Operand<sizeof(Void)>& op) : op(op) {}
```

```
    void operator() () {
        markjump(op.oper, op.opnd);
    }
};
```

7.6.6 Exceptions Thrown by Lazy Operands

Since lazy operands are executed in the stack frame of their enclosing function, and the C++ run time system is not aware of the still active operator function whose stack frame is above this, an exception thrown during the evaluation of an operand would not pass through the operator function, but rather terminate it abruptly, which would have two undesired consequences: First, destructors for local objects of this function would not be executed correctly, and second, the operator function would not have the chance to catch the exception as described at the end of Sec. 6.4.

To remedy these problems, the transformation scheme described above (Secs. 7.6.4 and 7.6.5) is modified once more:

```
Operand<sizeof(x, Void())> __x;
if (mark(__x.opnd)) {
    try { eval((x, Void()), __x); jump(__x.oper, 1); }
    catch (...) { jump(__x.oper, 2); }
}

Operand<sizeof(y, Void())> __y;
if (mark(__y.opnd)) {
    try { eval((y, Void()), __y); jump(__y.oper, 1); }
    catch (...) { jump(__y.oper, 2); }
}

__equal_greater(
    false ? eval((x, Void()), __x) : __x,
    false ? eval((y, Void()), __y) : __y
)
```

Any exception thrown during the evaluation of an operand is caught by a `catch (...)` clause whose associated statement block calls `jump` with a second argument of 2 instead of the default value 1. If this value appears as the result of the matching `markjump` call, the exception is simply rethrown there (which is possible without actually knowing its type) causing it to pass through any active `try` statements as desired:

```
template <typename T>
struct lazy {
    Operand<sizeof(T)>& op;
    lazy (Operand<sizeof(T)>& op) : op(op) {}

    T operator() () {
        switch (markjump(op.oper, op.opnd)) {
        case 1: {
            T* p = (T*)(op.val);
            T x = *p;
            p->~T();
            return x;
            }
        case 2:
            throw;
```

```
        }
    }
};

template <>
struct lazy<void> {
    Operand<sizeof(Void)>& op;
    lazy (Operand<sizeof(Void)>& op) : op(op) {}

    void operator() () {
        switch (markjump(op.oper, op.opnd)) {
        case 1:
            return;
        case 2:
            throw;
        }
    }
};
```

7.6.7 Blocks as Operands

If a block is used as an operand of a lazy operator, e. g., in `x ?* { y }`, this is transformed as follows:

```
Operand<sizeof(x, Void())> __x;
if (mark(__x.opnd)) {
    try { eval((x, Void()), __x); jump(__x.oper, 1); }
    catch (...) { jump(__x.oper, 2); }
}

Operand<sizeof(Void)> __y;
if (mark(__y.opnd)) {
    try {
        BlockOpnd __dummy(__y);
        try { y; __dummy(); }
        catch (...) { __dummy(); throw; }
    }
    catch (...) { jump(__y.oper, 2); }
    jump(__y.oper, 1);
}

try {
    ......
    __question_asterisk(
        false ? eval((x, Void()), __x) : __x,
        lazy<void>(__y)
    )
    ......
}
catch (Operand<sizeof(Void)>* op) { jump(op->opnd); }
```

Because a block cannot be used within the arguments of `sizeof` and `eval`, `sizeof(Void)` is used directly to declare the corresponding `Operand` object, and the block's evaluation is not surrounded by a call to `eval`. Furthermore, the `lazy<void>` object passed to the operator function can be constructed directly in that case, without employing the `?:` trick needed for other operands.

If a block executes a jump statement (`break`, `continue`, `goto`, or `return`) that transfers control out of the block, its effect would be similar to an exception that is not caught immediately: it would terminate the operator function abruptly, resulting in possibly lost destructor calls. However, it is a bit more difficult to "catch" these kinds of "exceptions" and to correctly deal with them. Of course, it would be possible for the precompiler in principle to analyze the statements of the block in order to detect these critical ones. It is easier, however, to declare a dummy object of the following type `BlockOpnd` for that purpose, whose destructor will be called automatically whenever the block in which it is declared is left:

```
struct BlockOpnd {
    Operand<sizeof(Void)>* op;
    BlockOpnd (Operand<sizeof(Void)>& op) : op(&op) {}

    void operator() () { op = 0; }

    ~BlockOpnd () { if (op) modjump(op->opnd, op->oper, 3); }
};
```

The object `__dummy` is initialized with a reference/pointer `op` to the `Operand` object `__y` representing the block operand. If this block terminates normally or throws an exception, the function call operator `__dummy()` is executed which sets `op` to null, in which case the destructor `~BlockOpnd` does nothing and `jump` will be executed with a value of 1 or 2, respectively.[9] Otherwise, i. e., if the block is terminated by a "critical" jump statement, `op` still refers to `__y` when `__dummy`'s destructor gets called, and in that case, another auxiliary function `modjump` similar to `markjump` is used to modify the execution context saved in `op->opnd` as described below and to jump back to the context saved in `op->oper`, i. e., to the execution of `lazy<void>::operator()`, this time passing the value 3. If this value is received there, the address of `op` is thrown as an exception in order to correctly terminate the execution of the operator function as well as the enclosing full expression:

```
template <>
struct lazy<void> {
    Operand<sizeof(Void)>& op;
    lazy (Operand<sizeof(Void)>& op) : op(op) {}

    void operator() () {
        switch (markjump(op.oper, op.opnd)) {
        case 1:
            return;
        case 2:
            throw;
        case 3:
            throw &op;
        }
    }
};
```

This exception is caught by the `catch` clause associated with the `try` statement that surrounds this full expression (which must actually be an expression *statement* because it contains blocks as operands, cf. Sec. 6.4) where another `jump` is executed that transfers control to the modified context `op->opnd`.

[9] To make sure that this destructor is always executed (and thus, constructor and destructor calls are correctly balanced), `jump(..., 2)` is not directly called in the inner `catch` clause, but rather postponed to the outer one by rethrowing the exception just caught (which might of course be eliminated by an optimizing compiler).

The function

```
void modjump (Context& c, const Context& d, int v = 1);
```

uses the base and stack pointer information found in `c` and `d` to determine the position on the stack where the return address of the currently executing destructor `~BlockOpnd` has been stored by the run time system and changes the program counter stored in `c` to this address. Therefore, the final call to `jump(op->opnd)` is actually nothing else but a goto statement to this address, because its execution context is otherwise identical to the one that has initially been stored in `op->opnd`. After this roundtrip from the destructor "up" to the operand evaluation function, then "down" to the expression containing the operator call, and finally back to the destructor's return address, the jump statement whose execution triggered the call to the destructor runs to completion transferring control to its original destination.

If `markjump` would be used instead of `modjump` here, the context stored in `op->opnd` would be completely overwritten by the execution context of the destructor, causing `jump(op->opnd)` to jump back *into* the destructor. Because the latter terminates immediately afterwards, i. e., transfers control to its return address, this seems to achieve the same effect as described above. The problem with this approach, however, is the fact that the destructor's stack frame (in particular, the position on the stack where its return address is stored) might be overwritten by other destructor calls which are performed during the stack unwinding caused by throwing the exception `&op`.

7.7 Implementation Limitations

The fact that C+++ is implemented by a "stupid" precompiler implies a few limitations which are described in the sequel.

7.7.1 Template IDs

If a template ID such as `vector<T>` is used in a constructor or function call, e. g., `vector<T>(100)`, this cannot be distinguished syntactically from a relational expression such as `a < b > (c)`. Therefore, the C+++ precompiler will *always* treat < and > as relational operators in expressions containing user-defined operators.[10]

Given the definition of the cardinality operator # from Sec. 3.2, an expression such as `#vector<T>(100)` will thus be parsed as `(#vector) < T > (100)` by the precompiler causing it to generate erroneous C++ code. Because the precompiler does not transform subexpressions which do not contain user-defined operators, the easiest way to prevent such misinterpretations is to enclose subexpressions containing template IDs in parentheses, e. g., `#(vector<T>(100))`.

Another, probably better solution from a conceptual point of view would be to use different tokens as template argument delimiters, e. g., `<|` and `|>`, which will be translated by the precompiler to the original tokens < and > after expressions have been transformed.

7.7.2 Old-Style Casts

So-called old-style casts, e. g., `(T)x` or `(T)(x)`, cannot be recognized as such by a stupid precompiler since the latter expression might also be a function call with redundant parentheses around the function name `T`. Therefore, they are actually treated as parts of an operand which is correct in many cases. However, if the operand of the cast starts with a binary operator as defined in Sec. 2, i. e., an operator that might be applied both infix and prefix (e. g., `(T)-x`), the precompiler will erroneously interpret this as an infix instead of a prefix operator. Again, such misinterpretations can be prevented with additional parentheses, e. g., `(T)(-x)`, or by using a different cast notation, e. g., `T(-x)` (functional cast notation) or `static_cast<T>(-x)` (new-style cast).

[10] It would be possible to treat < and > specially in the context of so-called new style casts (e. g., `static_cast<T>(x)`), because these are introduced by well-defined keywords, but this has not been implemented yet.

7.7.3 Lazy Operands

The final transformation scheme for lazy operands described in Sec. 7.6.6 restricts the use of lazy operands to expressions appearing in function bodies, i. e., it is not allowed to use them in member initializers of constructors, default arguments of functions, initializers of global and namespace variables, and initializers of static data members of classes, because in these contexts there is no place to put the additional `if (mark(...)) ...` code that is needed for the transformation. To circumvent this limitation, auxiliary functions containing the expressions in question can be used.

Furthermore, if an expression containing lazy operands is used to initialize a locally declared variable, this variable is not known in this additional code (because it appears before the declaration). This is unproblematic in almost all cases, since a variable is rarely used within its own initialization (`void* p = &p` is an example of an exception).

7.7.4 Jump Statements Out of Block Operands

The implementation of jump statements transferring control out of a block operand assumes that the exception `&op` thrown by `lazy<void>::operator()` that shall terminate the operator function (and any possible intermediate functions) that called this function operator (cf. Sec. 7.6.7) is not caught by any of these functions (or that it is rethrown afterwards). Otherwise, it will not reach its intended destination, resulting in possibly undefined behaviour. (In particular, the jump statement will not reach its original destination.)

Since the type of this exception is internal to the implementation, it is assumed that a programmer does not actually know it and therefore is simply not able to catch it with an ordinary `catch` clause. Using a `catch` with an ellipsis, however, even exceptions of unknown types can be caught, but in these cases it is typical to rethrow the exception at the end of the `catch` block.

7.8 Status of the Implementation

The main part of C+++'s implementation is straightforward and uses only simple, well-defined C++ concepts, i. e., function calls. The implementation of lazy operands, however, is tricky, not to say bold, in that it uses `setjmp` and `longjmp` in a partially undefined way which might not interact well with certain compilers or optimizers. So far, it has been tested successfully with some versions of GCC, including some levels of optimization.

Therefore, this part of the implementation should be considered a proof of concept rather than a real-world implementation. (In particular, it should not be used for safety-critical applications!) However, by employing these tricks it has been possible to construct a working implementation that is sufficient for exploiting the possibilities of user-defined control structures in a rather short amount of time. As a long-term goal, however, it would be desirable to incorporate the features of C+++ into a real C++ compiler such as GCC.

8 Related Work

This report has presented the 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 [13], it has not found widespread dissemination in mainstream imperative programming languages.

Compared with Prolog [1] and modern functional languages such as ML [7] and Haskell [9], which support the concept in principle, the approach presented here offers a more flexible way to specify operator precedences (because the precedence relationship is not a total, but only a partial order), the additional concepts of of fixary operator combinations and flexary operators (where the latter 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, however, that this concept has already been present in ALGOL 60 [8], 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. 6 have demonstrated that the basic principle is useful when applied with care, because it opens the door to implement user-defined control structures, especially when combined with the concept of fixary operator combinations and if whole blocks of statements are allowed as operands, too. (The latter is again unnecessary in functional languages, because they do not have a notion of statements, but rather denote all computations solely by means of expressions.)

Compared with other languages offering the possibility of user-defined control structures, such as Common Lisp [14], Dylan [2], and (in a limited form) Ruby [11], the approach presented here has the advantage that it does not require separate concepts and language constructs, such as macros with tricky quote and unquote rules in Lisp, macros with rewrite rules, patterns, templates, etc. in Dylan, and special code blocks associated with method calls in Ruby, but simply reuses and generalizes the existing concept of operator overloading to achieve that aim. Compared with Smalltalk [3], which pursues a similar strategy by defining control structures by means of methods and blocks, C+++ also supports lazily evaluated expressions which are not blocks and does not require the programmer to explicitly mark each operand that shall be evaluated lazily by encapsulating it in a block. Instead, the decision whether operands shall be evaluated lazily or not is simply expressed once by declaring an operator `lazy` or not.

Acknowledgement

The basic ideas of C+++ have been implemented in two student projects in 2002 by Michael Altmann and Dietmar Sauer as well as Wolfgang Doll, Heiko Lorenz, and Michael Sonnenfroh. Based on their experiences, the concept has been significantly refined and extended, especially by the concept of lazily evaluated operands. The current precompiler has been implemented with assistance of Wolfgang Doll.

The anonymous referees of [5], a significantly shorter version of this report, missing completely the concepts of fixary operator combinations and blocks as operands as well as the description of the implementation, provided helpful comments to improve both that paper and this report.

References

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

[2] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

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

[4] ISO/IEC: *International Standard: Programming Languages − C++* (Second Edition, ISO/IEC 14882:2003(E)). October 2003.

[5] C. Heinlein: “C+++: User-Defined Operator Symbols in C++.” In: *3. Arbeitstagung Programmiersprachen* (Ulm, Germany, September 2004). Gesellschaft für Informatik e. V., Lecture Notes in Informatics, (in print).

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

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

[8] P. Naur (Ed.): “Revised Report on the Algorithmic Language ALGOL 60.” *Numerische Mathematik* 4, 1963, 420−453.

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

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

[11] D. Thomas, A. Hunt: *Programming Ruby: The Pragmatic Programmer’s Guide* (2nd Edition). Addison-Wesley, 2001.

[12] 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.

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

[14] P. H. Winston, B. K. P. Horn: *LISP* (Third Edition). Addison-Wesley, Reading, MA, 1989.