Operator Overloading in C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the mechanism for operator overloading provided in C++[1]. Except for minor details C++ is a superset of the C programming language[2]. A programmer can define a meaning for the standard C++ operators when applied to objects of a specific class[3]. Most operators can be defined for objects of a class; in addition to arithmetic, logical, and relational operators, call ``( )'' and element selection ``[ ]'' can be defined, and both assignment and initialization can be redefined. It is not possible to change the meaning of operators applied to non-class objects, nor to change the syntax or precedence rules for operators. Constructors can be used both to provide constants for a user-defined type and to specify conversions between types.

Examples of how to define data types like `complex`, `matrix`, and `string` are presented. In these examples it is shown how to handle expressions involving objects of mixed type, large objects, and objects of varying size.

The overloading facilities are discussed in some detail to demonstrate the run-time efficiency that can be achieved when using them and the simplicity of their implementation in the compiler. C++ has been in use for non-trivial applications since October 1983; there are now more than a hundred installations.

[1] Bjarne Stroustrup: "The C++ Programming Language - Reference Manual" In this volume.

[2] B.Kernighan and D.Ritchie: "The C Programming Language" Prentice-Hall 1978.

[3] Bjarne Stroustrup: "Data Abstraction in C++" In this volume.

1 Introduction

Programs often manipulate objects that are concrete representations of abstract concepts. For example, the C++ data type `int` together with the operators + - * / etc., provides a (restricted) implementation of the mathematical concept of integers. Such concepts typically include a set of operators representing basic operations on objects in a terse, convenient, and conventional way. Unfortunately, only very few such concepts are or can be directly supported by a programming language. For example, ideas like complex arithmetic, matrix algebra, logic signals, and strings receive no direct support in C++. Classes provide a facility for specifying a representation of non-primitive objects in C++ together with a set of operations that can be performed on such objects. Defining operators to operate on class objects sometimes allows a programmer to provide a more conventional and convenient notation for manipulating class objects than could be achieved using only the basic functional notation. For example:

```
class complex {
        double  re, im;
public:
        complex(double r, double i) { re=r; im=i; }
        friend complex operator+(complex, complex);
        friend complex operator*(complex, complex);
}
```

defines a simple implementation of the concept of complex numbers, where a number is represented by a pair of double precision floating point numbers manipulated (exclusively) by the operators + and *. The programmer provides a meaning for + and * by defining functions named `operator+` and `operator*`. For example, giver `b` and `c` of type `complex`, `b+c` means (by definition) `operator+(b,c)`. It is now possible to approximate the conventional interpretation of complex expressions. For example:

```
complex a = complex(1, 3.1);
complex b = complex(1.2, 2);
complex c = b;

a = b+c;
b = b+c*a;
c = a*b+complex(1,2);
```

The usual precedence rules hold so the second statement assigns `b+(c*a)` to `b`.

2 Overview

First the aims of the design of the overloading mechanism are briefly stated. Then the facilities for defining class `complex` are described. A complex number has a trivial representation, the set of basic operations is well known, and there exists a conventional notation for these operations. This allows the discussion of complex numbers to focus on the problem of providing that conventional notation. This will involve discussion of several language facilities that do not directly concern operator overloading. These facilities ("friend functions", "constructors", and "overloaded function names") can be used to make a user-defined data type convenient to use. It will be shown how to specify rules for initializing variables, how to provide constants for a type, how to specify type conversion rules, and how to accommodate the need to use conventional names like `abs` and + for operations on different types.

Once these techniques have been presented, it is possible to consider data types where the representation is non-trivial. Class `matrix` is an example of a data type where it is prohibitively expensive to copy an object each time an operation is performed on it. A new data type "reference" is introduced to cope with this. Class `string` shows how a data type whose implementation involves free store management and sharing of objects can be implemented. Finally, the techniques used to implement the overloading facilities are discussed. The relation of the C++ operator overloading facilities to those provided in other languages is not discussed.

3 Aims

The main aims for the design of the overloading mechanism described here are:

[1] It must be possible to define data types like `complex`, `matrix`, and `string` with a user interface as elegant as one would expect from a built in data type.

[2] It must be possible to provide efficient implementations of the basic operations on such user defined data types.

[3] The base language must remain immutable, that is, it should not be possible to redefine the meaning of operators applied to objects of a non-class type.

[4] Programs using the overloading facilities must be relatively easy to compile.

In particular, the first aim is considered to imply that

[a] all storage management issues in the implementation of a new type can be hidden from a user.

[b] a binary operator can be defined to accept one operand of a user-defined type and the other operand of some other user-defined or basic type.

[c] constants can be defined for a user defined type.

[d] variables of a user defined type can be initialized like variables of basic data types.

It is not the intention to provide facilities for the user to change the syntax of the language in any way. Furthermore, except for minor details C++ is a superset of C; see reference 1†.

It is not the intention to provide any "default semantics" for operators, nor is it the intention to restrict definitions of operators to some preconceived idea of what is reasonable. For example, it is quite possible to define = to mean plus and + to mean assignment. The only protection provided against idiotic use is the guarantee that the base language is immutable.

4 Friends and Members

Two kinds of functions can be defined to manipulate the representation of a user-defined data type: member functions and friend functions. They differ in both scope and calling syntax. Consider a simple class `complex` with a conjugation function of each kind:

```
class complex {
        float re, im;
public:
        complex member_conj();
        friend complex friend_conj(complex arg);
};
```

The member function has the more elegant definition since it can refer to the representation of the object for which it was invoked directly; the friend function must refer to it through an argument:

```
complex complex.member_conj()
{
        return complex(re,-im);
}

complex friend_conj(complex arg)
{
        return complex(arg.re,-arg.im);
}
```

However, only the friend can be called using conventional mathematical notation; the member must use *object.member* notation:

```
complex a;
a.member_conj();
friend_conj(a);
```

† The major incompatibility is that in C++ `int f()` declares a function taking no arguments, whereas in C it declares a function accepting any number of arguments of any type.

For a conjugation operation the calling syntax for the friend function is obviously preferable. The calls of a member function `conj()` are simply too ugly to live with. Consider, for example, `a+b.conj()` and `(a+b).conj()`.

Friends are ideal for implementing traditional arithmetic operations. However, if it is necessary to modify the contents of one operand, as in a function defining an assignment operator, a member function must be used. Furthermore, where there is no other reason to prefer the one implementation over the other an operation should be a member. A definition of a member is typically shorter than the definition of the equivalent friend, the implicit passing of the pointer identifying the object for which a member is invoked is potentially more efficient than argument passing, and the name of a member is restricted to its proper scope rather than global. Finally, constructors (§5) and destructors (§15) MUST be members.

5 Constructors

A class may have a function member with the name of the class itself, like `complex()` in class `complex` above. Such a function is called a constructor. A constructor is a prescription for initializing a class object; in other words, it constructs a value of the class type. If a class has a constructor then the constructor will be called to initialize objects of the class before their use. If the constructor requires arguments they must be provided. The following examples are equivalent, that is, the three variables will be initialized to the same value:

```
complex a = complex(1,2);
complex b(1,2);
complex c = b;
```

Initialization with an object of the same type is an alternative to calling the constructor.

A constructor can also be used in expressions, often replacing explicit use of a temporary variable. For example, `operator+()` can be defined without use of an explicit temporary variable:

```
complex operator+(complex a1, complex a2)
{
        return complex(a1.re+a2.re, a1.im+a2.im);
}
```

6 Operator Functions

Functions defining meanings for the following operators can be declared for a class:

```
+     -     *     /     %     ^     &     |     ~     !     =     <     >
+=    -=    *=    /=    %=    ^=    &=    |=    <<    >>    >>=   <<=   ==
!=    <=    >=    &&    ||    ++    --    []    ()    new   delete
```

The last four are subscript (§14), function call (§13), free-store allocation, and free-store deallocation. The following operators can not be defined or re-defined:

```
->    .    sizeof    ?:    ,
```

The name of an operator function is the keyword `operator` followed by the operator itself, for example `operator<<`. An operator function is declared and can be called like any other function; a use of the operator is only a shorthand for an explicit call of the operator function.

A binary operator can be defined either by a member function taking one argument or by a friend function taking two arguments. Thus, for any binary operator `@`, `aa@bb` can be interpreted as either `aa.operator@(bb)` or `operator@(aa,bb)`. If both `operator@` functions are defined the former interpretation is used. A unary operator, whether prefix or postfix, can be defined by either by a member function taking no arguments or a friend function taking one argument. Thus, for any unary operator `@`, `both aa@` and `@aa` can be interpreted as either `aa.operator@()` or `operator@(aa)`. If both `operator@` functions are defined the former interpretation is used. For example, `X.operator&()` defines the operator usually called "address of" for a class `X`, and not the binary operator "and". When the operators `++` and `--` are overloaded, it is not possible to distinguish prefix application from postfix application.

The meanings of some operators are defined to be equivalent to some combination of other operators on the same arguments. For example: `++a` means `a+=1` which in turn means `a=a+1`. Such relations do not hold for overloaded operators unless the user defines them that way. For example, the definition of `operator+=` cannot be deduced from the definitions of `operator+` and `operator=`. No assumptions are made about the meaning of overloaded operators. In particular, since an overloaded "assignment operator" is not assumed to implement assignment to its first argument no test is made to ensure that that argument is an lvalue.

Because of historical accident the operators `=` and `&` have pre-defined meanings when applied to class objects. There is no elegant way of "undefining" these two operators. They can, however, be disabled for a class `X`. For example, by declaring `X.operator&()` but not providing a definition for that function one can ensure that no program taking the address of an `X` will run.

An operator function must either be a member or take at least one class object argument (functions re-defining the `new` and `delete` operators need not). This rule ensures that a user cannot change the meaning of any expression not involving a user-defined data type. In particular, it is not possible to define an operator function which operates exclusively on pointers. It is not possible to define a new operator or to change the precedence of an existing operator.

Note that an operator function intended to accept a basic type as its first operand cannot be a member function. For example, consider addition of a complex variable `aa` and and integer `2`: `aa+2` can with a suitable declared member function be interpreted `aa.operator+(2)`, but `2+aa` cannot since there is no class `int` for which to overload `+` to mean `2.operator+(aa)` (and even if there was, two different functions would be needed to cope with `2+aa` and `aa+2`).

7 Overloaded Function Names

The implementation of complex numbers presented in the introduction is too restrictive to please anyone, so it must be extended. This is mostly a trivial repetition of the techniques presented above. However, two desirable features cannot be handled that easily: unary minus and mixed operations on complex and real numbers. Both require two different interpretations for a single operator symbol. For example:

```
class complex {
        double  re, im;
public:
        complex(double r, double i) { re=r; im=i; }

        complex operator-();                          /* unary minus */
        friend complex operator-(complex, complex);   /* binary minus*/
        friend complex operator+(complex, complex);
        friend complex operator*(complex, complex);
        friend complex operator*(complex, double);
        friend complex operator*(double, complex);
};
```

When several (different) function declarations are specified for a single name, that name is said to be overloaded. When that name is called, the correct function to execute is selected by comparing the types of the actual arguments with the argument types in the function declarations. Of the "usual arithmetic conversions" defined in §6.6 of the C++ reference manual[1] only the conversions *char−>short−>int, int−>double, int−>long,* and *float−>double* are performed.

With this declaration of `complex` we can now write:

```
complex a(1,1), b(2,2), c(3,3), d(4,4), e(5,5);
a = -b-c;
b = c*2.0*c;
c = (d+e)*a;
```

Any function name can be overloaded, not just `operators`. The names of non-member functions must explicitly be declared overloaded. For example:

```
overload abs;
int      abs(int);
double   abs(double);
complex  abs(complex);
```

When an overloaded name is called, the list of functions is scanned in order to find one which can be invoked. For example `abs(12)` will invoke `abs(int)` and `abs(12.0)` will invoke `abs(double)`. Had the order of declarations been reversed, both calls would have invoked `abs(double)`.

All operators are by definition overloaded.

8 Conversion

Writing a function for each combination of `complex` and `double`, as for `operator*()` above, is unbearably tedious. For example, a realistic facility for complex arithmetic provides more than 10 functions taking two complex or real arguments[4]. An alternative is to declare a constructor that creates a `complex` given a `double`. For example

```
class complex {
        ...
        complex(double r) { re=r; im=0; }
};
```

A constructor requiring a single argument need not be called explicitly. For example

```
complex z1 = complex(23);
complex z2 = 23;
```

are both legal, and `z1` and `z2` will both be initialized calling by `complex(23)`.

A constructor is a prescription for making a value of a given type. Where a value of a type is expected, and where such a value can be created by a constructor, given the value to be assigned, the constructor will be used. For example, class `complex` could be declared like this:

```
class complex {
        double re, im;
public:
        complex(double r, double i) { re=r; im=i; }
        complex(double r)           { re=r; im=0; }

        friend complex operator+(complex, complex);
        friend complex operator*(complex, complex);
};
```

and operations involving `complex` variables and integer constants would be legal. An integer constant will be interpreted as `complex` with the imaginary part zero. For example: `a=b*2` generates code like `a=operator*(b,complex(double(2)))`.

An assignment to an object of class `X` is therefore legal if either the assigned value is an `X`, or if `X` has a constructor accepting a single argument of the type of the assigned value.

In some cases a value of the desired type can be constructed by repeated use of constructors. This must be handled by explicit use of constructors; only one level of implicit construction is legal. Note that the standard conversions described above are always performed and do not count as "implicit construction". In some cases a value of the desired type can be constructed in more than one way. Such cases are illegal. For example:

[4] Leonie V. Rose and Bjarne Stroustrup: "Complex Arithmetic in C" In this volume.

```
class x { ... x(int); x(char*); };
class y { ... y(int); };
class z { ... z(x); };

overload f;
x f(x);
y f(y);

z g(z);

f(1);          /* illegal: ambiguous f(x(1)) or f(y(1))  */
f(x(1));
f(y(1));
g("asdf");     /* illegal: g(z(x("asdf"))) not tried */
g(z("asdf"));
```

Where an overloaded function takes an argument of a type for which a constructor exists more than one interpretation may appear legal. For example:

```
class x { ... x(int); };
overload h(double), h(x);
h(1);
```

The call could be interpreted either as `h(double(1))` or as `h(x(1))` and would appear to be illegal according to the rule above. However, the first interpretation is an "exact match" and will be chosen; the rule against ambiguities applies only to user-defined conversions.

These rules for conversion are neither the simplest to implement, the simplest to document, nor the most general which could be devised. Consider the requirement that a conversion must be unique to be legal. A simpler approach would allow the compiler to use any conversion it could find; thus it would not be necessary to consider all possible conversions before declaring an expression legal. Unfortunately, this would mean that the meaning of a program depended on which conversion was found. In effect, the meaning of a program would in some way depend on the order of the declaration of the conversions. Since these will often reside in different source files (written by different programmers), the meaning of a program would depend on the order in which its parts were merged together. Alternatively, implicit conversions could be disallowed. Nothing could be simpler, but this rule leads to either inelegant user interfaces or an explosion of overloaded functions as seen in the class `complex` in the previous section.

The most general approach would take all available information into account. For example, using the declarations above, it would handle `aa=f(1)` because the type of `aa` will determine a unique interpretation. If `aa` is an `x`, `f(x(1))` is the only one yielding the `x` needed in the assignment. The most general approach would also cope with `g("asdf")` because `g(z(x("asdf")))` is a unique interpretation. The problem with this approach is that it requires extensive analysis of a complete expression to determine the interpretation of each operator and function call. This leads to slow compilation and also to surprising interpretations and error messages, as the compiler considers conversions defined in libraries etc. It simply takes more information into account than the programmer writing the code can be expected to know!

9 More about Conversion

A type conversion from type `T1` to type `T2` is defined as a constructor for `class T2` accepting a single argument of type `T1`. For example

```
class complex {
        ...
        complex(double d) { re=d; im=0; }
};
```

This has implications that can be undesirable:

[1] There can be no implicit conversion from a user-defined type to a basic type (since the basic types are not classes).

[2] It is not possible to specify a conversion from a new type to an old one without modifying the declaration for the old one.

[3] It is not possible to have a constructor with a single argument without also having a conversion.

To cope with these problems a facility for defining a type conversion from one type into another as part of the declaration of the "source type" was added. A member function `X::operator T()` where `T` is a type name defines a conversion from `X` to `T`. For example:

```
class boolean {
        int b;
public:
        friend boolean operator+(boolean, boolean);
        boolean(int i) { b = i!=0; }
        operator int() { return b; }
};

boolean b1 = 1, b2 = 0;
int i1 = 10, i2 = 20;

i1 = b1;            /* i1 = 1 */
b2 = i2;            /* b2 = 1 */
i = b1+b2;

extern f(int);
f(b1);              /* f(1) */
```

The rules for handling of ambiguities are unchanged: A user defined conversion must be unique to be applied implicitly, so given the declarations above the expression

```
b1+1
```

is an error because it could be interpreted as either

```
b1.operator int() + 1
```

or

```
b1.operator+(1)
```

Declaring both

```
class X { ... X(Y); };
```

and

```
class Y { ... operator X(); };
```

is an error since anycd use of `operator X()` would be ambiguous.

10 Constants

It is not possible to define constants of a class type in the sense that `1.2` and `12e3` are constants of type `double`. However, constants of the basic types can often be used instead if class member functions are used to provide an interpretation for them. Constructors taking a single argument provides a general mechanism for this. Where constructors are "simple" and inline substituted, as in class `complex` above, it is quite reasonable to think of constructor invocations as constants. For example `zz1*3+zz2*complex(1,2)` will cause three function calls and not five.

11 References

For each use of a `complex` binary operator a copy of the second operand is passed as an argument to the function implementing the operator. The overhead of copying two `doubles` is noticeable but probably quite acceptable. Unfortunately, not all classes have a conveniently small representation. To cope with this, one could try to declare operator functions to take pointer arguments. For example:

```
class matrix {
        double m[4][4];      /* 128 bytes on a VAX */
public:
        matrix();
        friend matrix operator+(matrix*, matrix*);
        friend matrix operator*(matrix*, matrix*);
};
```

Unfortunately, this leads to rather strange looking expressions.

```
matrix a, b, c;
a = &b + &c;
```

This is clearly not acceptable. Furthermore, the declarations of the `matrix` operator functions above are illegal, since they do not take any class object arguments (pointers to class objects do not count). The alternative is to define matrix operations to take "references" as arguments, rather than pointers.

A reference, like a pointer, embodies the idea of an address of an object. The notation `X&` means reference to `X`. A reference differs from a pointer in that

[1] When a reference is used the dereference operator `*` will be implicitly applied.

[2] When a variable is declared to be a `X&`, it can be initialized by an `X`. The address operator `&` will be implicitly applied. It is also legal to initialize a `X&` by a `X*`.

For example:

```
matrix m;
matrix& r = m;   /* means   r = &m                                 */
matrix m2 = r;   /* means m2 = *r                                  */
r = m;           /* means *r =  m                                  */
m = r;           /* means  m = *r                                  */
```

Note that initialization of a reference is treated very differently from assignment to it. This is reasonable, since dereferencing of an uninitialized variable is known to be meaningless. For example:

```
int a;
int& r1 = a;      /* means  r1 = &a                                */
int& r2; r2 = a;  /* error: uninitialized reference                */
```

Argument passing and function value return are considered to be initializations, so:

```
matrix& f(int);
matrix operator+(matrix&, matrix&);
m = f(1);        /* means m = *f(1);                                */
f(2) = m2;       /* means *f(2) = m2;                               */
m+m2;            /* means +(&m,&m2)                                 */
r+m;             /* means *r+m that is +(&*r,&m) that is +(r,&m)    */
```

Despite appearances, no operator operates on a reference. For example,

```
int ii = 0;
int& rr = ii;
rr++;
```

is legal, but `rr++` does not increment the reference `rr`; rather, the interpretation is `(*rr)++`. That is, ++ is applied to an `int` which happens to be `ii`.

The value of a reference cannot be changed after initialization. To get a pointer `pp` to denote the same object as a reference `rr` one can write `pp=&rr`. This will be interpreted as `pp=&*rr`.

Consider the declaration

```
double& dr = 1;
```

Here, the initializer is not an lvalue. In fact, it is not even of the right type. In such cases
[1] first the conversion rules are applied,
[2] then the resulting value is placed in a temporary variable,
[3] finally the address of this is used as the value of the initializer.
Thus, the interpretation of the example above is:

```
double* dr;
double temp;
temp = double(1);
dr = &temp;
```

References allow the use of conventional expressions involving the usual arithmetic operators for "large objects" without requiring an implementation that causes excessive copying. The plus operator could be defined like this:

```
matrix operator+(matrix& arg1, matrix& arg2)
{
        matrix sum;
        int i, j;

        for (i=0; i<4; i++)
                for (j=0; j<4; j++)
                        sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
        return sum;
}
```

The dot is used for member selection, rather than `->`, because `arg1` and `arg2` are implicitly dereferenced; `arg1.mem[i][j]` means `(*arg1).mem[i][j]`.

12 Returning a Class Object

In the example above `operator+(matrix&,matrix&)` operates on the operands to `+` through references, but returns an object value rather than a reference. Returning a reference would appear to be more efficient. For example:

```
class matrix {
        ...
        friend matrix& operator+(matrix&, matrix&);
        friend matrix& operator*(matrix&, matrix&);
};
```

This would be legal, and the matrix expression `a*b+c` would be interpreted as `operator+(operator*(&a,&b),&c)`. However, the user would have to implement some kind of temporary variable to hold the result of `operator*()`. Since a reference to it will be passed out of the function as the return value, it cannot be a local variable. It would typically be allocated on the free store, and later freed for re-use. Copying the return value will often be cheaper (in execution time, code space, and data space) and simpler to program.

For medium sized objects the style of declaring an `operator` function to take reference arguments and return an object value is probably the best approach. The implementation of functions returning medium and large objects relies on passing a pointer to an object where the result is deposited. If a function, like the original `operator+(matrix&,matrix&)` above, composes its return value in a local variable and then returns it, the compiler will ensure that the result value is composed right in the object where it would eventually have been copied to (this will be explained in greater detail below). This means that the code generated for an operator taking reference arguments and returning an object value operates on three objects through pointers: the two reference arguments and the result through a compiler provided target pointer.

13 Overloading Function Call: operator()

Function call, that is, the notation *expression(expression-list)*, can be interpreted as a binary operation and the call operator () can be overloaded in the same way as other operators. For example:

```
class string {
        char*    s;          /* representation */
        int      length;
        string*  sub_of;     /* substring of   */
public:
        string(char*);
        string();
        string  operator=(string&);             /* assignment          */
        string  operator()(int pos, int len);   /* substring selector  */
};
```

With suitable definitions of operator=() and operator()() one can define assignment and subscripting to be used like this:

```
string a="foo", b="bar", c;
c = b;
c = b(1,2);
a(0,1) = b;
```

The basic idea is to let the substring function return an object containing a pointer to the original string which can be used to manipulate that string. For example:

```
string string.operator()(int pos, int lgt) {
{
        string sub;
        sub.sub_of = this;
        sub.length = lgt;
        sub.s = s+pos;
        return sub;
}

void string.operator=(string& from)
{
        if (sub_of) {   /* assign to sub_of->s, etc. */
        }
        else {          /* assign to s, etc. */
        }
}
```

This technique for assigning to a part of an object through an object returned by a selector function is useful in many contexts.

An argument list for an operator() function is evaluated and checked according to the usual argument passing rules.

14 Overloading Subscripts: operator[]

An operator[] function can be used to give subscripts a meaning for class objects. For example:

```
class string {
        char* s;
        ...
        char& operator[](int i) { return (0<=i && i<length) ? s[i] : 0; }
};
```

Note that because the value returned by string.operator[]() is a char& that function can be used on either side of an assignment. For example:

```
string ss = "asdf";
ss[1] = ss[3];
```

The assignment is interpreted as `*(ss.operator[](1))=*(ss.operator[](3))` that is, the new value of `ss` is `"afdf"`.

The second argument (the subscript) of an `operator[]` function may be of any type. This makes it possible to define associative arrays, etc.

15 Destructors

A destructor is a function that will be called (implicitly) when a class object is destroyed. A class object is destroyed if it goes out of scope or if it is explicitly deleted using the `delete` operator. The name of the destructor for class `X` is `~X`, and it takes no argument. Destructors are often useful to clean up secondary data structures. For example:

```
class vector {
        int* v;
        int size;
        ...
        vector(int sz) { v = new int[size=sz]; }
        ~vector()      { delete v; }
};
```

Note that a destructor is not invoked when a value is destroyed by assignment. For example,

```
vector v1(10), v2(20);
v1 = v2;
```

will not cause the destructor `~vector()` to be invoked for `v1`. Otherwise, assignment of an object to itself, for example `v1=v1`, would lead to chaos. To ensure that `v1`'s vector of integers is not lost `vector.operator=()` can be defined.

16 Constructors Revisited

When `operator=()` is declared it will be called for all uses of the assignment operator `=`. However, an object can also be copied as an argument to a function, as a return value, or as an initializer for a new class object. Each of these three cases involves the construction of a value in an uninitialized object. They are treated identically and are collectively referred to as initialization.

Consider the problem of implementing "call by value" for objects of class `string`. A user sees a `string` as a single object, but its implementation consists of two parts: the class object itself and the string representation pointed to by it. When a copy is made of a class `string` object both copies will denote the same representation and further operations on them may corrupt it. To handle explicit assignments `operator=()` can be declared (for simplicity substrings and assignments like `s=s` will be ignored here):

```
void string.operator=(string& from) {
        delete s;
        length = from->length;
        s = new char[length+1];
        strcpy(s,from->s);
}
```

To cope with initialization for a class `X`, a constructor is needed since the default initialization method, bitwise copy, may cause chaos. Intuitively, a constructor `X(X)` would implement construction of one `X` from another. However, to avoid infinite regression the constructor invoked to implement passing of arguments of type `X` cannot itself take an `X` as an argument. Consequently, it is illegal to declare a constructor `X(X)`, and a constructor `X(X&)` is used instead.

The constructor `X(X&)` will typically be simpler than the corresponding `X.operator=()` for the same class. For example:

```
string.string(string& from)
{
        length = from->length;
        s = new char[length+1];
        strcpy(s,from->s);
}
```

One can now use strings without ending up with shared representations.

```
string a = "asdf";
string b = a;      /* a's representation will be copied ( b.string(&a) ) */
f(a);              /* a's representation will be copied (see below)      */
```

17 Function Return Revisited

In the following pseudo-C will be used to describe generated code. Syntactically illegal names will be used freely. Consider the function

```
X f()
{
        X a(1);
        X b(2);
        b->compute();
        ...
        return b;
}
```

where the constructor `X(X&)` has been declared†. The (unoptimized) code generated will be something like this:

```
void f(X* tp) {       /* pass result location */
        X a, b;
        a.X(1);       /* construct value */
        b.X(2);
        b.compute();
        ...
        tp->X(&b);    /* return result */
        a.~X();       /* destroy value */
        b.~X();
}
```

Assuming that the constructor `X()` and the destructor `~X()` have also been declared, they must be called as the scope of the function is entered and left, respectively.

By using the result variable, `*tp`, as the local variable `b` the code can be optimized to something like this:

```
void f(X* tp) {
        X a;
        a.X(1);
        tp->X(2);        /* b.X(2);    */
        tp->compute();   /* b.compute(); */
        ...
        a.~X();
}
```

The object denoted by `tp` now needs to be initialized because it is manipulated under the name `b`. However, `b` is not deleted because it does not belong to the scope of this function.

Now consider a call of `f()`:

† For unintersting historical reasons C++ Release E uses a less elegant method for returning values in this case.

```
        X x(0);
        x = f();
```

Assuming X.operator=() is declared the code generated is

```
        X x;
        X temp;
        x.X(0);
        f(&temp);
        x.operator=(&temp);
        temp.~X()
```

where temp is used nowhere else. Note that this code is independent of whether the return operation is optimized.

An obvious optimization appears to have been missed here. Why wasn't f(&x) generated so that the temporary could be omitted and no copying at all would occur? This further optimization can only be done where it is known that the value of x is not used during the execution of f(). Unfortunately, for most functions this cannot be deduced at compile time. However, where this is known, it is considered legal to optimize away temporary variables of type X even when X(X&), X.operator=(), or ~X() is defined by the user.

18 Argument Passing Revisited

Consider a function f() that takes an argument of class X for which the constructor X(X&) has been declared:

```
        void f(X a) { ... }
```

The code generated will be something like this:

```
        void f(X a) {
                ...
                a.~X();
        }
```

The code generated for calls

```
        f(x);
        f(g());         /* g() returns a class X object */
```

is something like this

```
        X temp;         /* uninitialized */
        temp.X(&x)
        f(temp);
        g(&temp);       /* g() returns its value in temp */
        f(temp);
```

19 Caveat

Like most programming language features, operator overloading can be both used and misused. In particular, the ability to define new meanings for old operators can be used to write programs that are well nigh incomprehensible. Imagine, for example, the problems facing a reader of a program where the operator + has been made to denote subtraction, or a program where all common operations are invoked using the arithmetic operators even though the types used have no conventional association with those operators.

The mechanism presented here should protect the programmer/reader from the worst excesses of overloading by not enabling a programmer to change the meaning of operators for basic data types like int, and by preserving the syntax of expressions and the precedence of operators.

20 Acknowledgements

Stu Feldman, Doug McIlroy, and Jon Shopiro contributed greatly to the design of these mechanisms.