Generalizing Overloading for C++2000

Bjarne Stroustrup

AT&T Labs, Florham Park, NJ, USA

Abstract

This paper outlines the proposal for generalizing the overloading rules for Standard C++ that is expected to become part of the next revision of the standard. The focus is on general ideas rather than technical details (which can be found in AT&T Labs Technical Report no. 42, April 1,1998).

Introduction

With the acceptance of the ISO C++ standard, the time has come to consider new directions for the C++ language and to revise the facilities already provided to make them more complete and consistent.
A good example of a current facility that can be generalized into something much more powerful and useful is overloading. The aim of overloading is to accurately reflect the notations used in application areas. For example, overloading of + and * allows us to use the conventional notation for arithmetic operations for a variety of data types such as integers, floating point numbers (for built-in types), complex numbers, and infinite precision numbers (user-defined types). This existing C++ facility can be generalized to handle user-defined operators and overloaded whitespace.

The facilities for defining new operators, such as :::, <>, pow , and abs are described in a companion paper [B. Stroustrup: "User-defined operators for fun and profit," Overload April, 1998].
Basically, this mechanism builds on experience from Algol68 and ML to allow the programmer to assign useful - and often conventional - meaning to expressions such as

```
double d = z pow 2 + abs y;
```

and

```
if (z <> ns:::2) // …
```

This facility is conceptually simple, type safe, conventional, and very simple to implement.

Basic Whitespace Overloading

Here, I describe the more innovative and powerful mechanism for overloading whitespace. Consider x*y. In programming languages (e.g. Fortran, Pascal, and C++), this is the conventional notation for multiplying two values. However, mathematicians and physicists traditionally do not use the operator *. Instead they use simple juxtaposition to indicate multiplication. That is, for variables x and y of suitable types,

```
x y
```

means multiply x by y.

This is simply achieved by overloading the space operator for double-precision floating-point values:

```
double operator (double d1, double d2) { return d1*d2; }
```

Or - more explicitly - equivalently

```
double operator ' '(double d1, double d2) { return d1*d2; }
```

Given one of these definitions, a physicist can use his (or her) conventional notation rather than the notation that has become conventional among computer scientists:

```
double f(double x, double y, double z)
{
        return a + x y pow z;       // using also a user-defined operator pow
}
```

Clearly, the space operator has (by default) a precedence lower than pow and higher than +. The mechanism for assigning precedence to user-defined operators is described in detail in the companion article.

Naturally, this requires that overloading is allowed for built-in types. However, to avoid absurdities, it is (still) not allowed to provide new meanings for the built-in operators for built-in types. Thus, the language remains extensible but not mutable. In fact, generalizing the overloading rules allows us to provide a unified clean framework for built-in and user-defined types as well as for built-in and user-defined operators. This improvement furthermore opens the opportunity to eliminate many of the anarchic and error-prone traditional implicit conversions inherited from C in the next revision of the C++ standard.

Previous Work

The overloading mechanisms described here are partly inspired by the pioneering work of Bjørn Stavtrup [B. Stavtrup: "Overloading of C++ Whitespace." JOOP April, 1992]. However, Dr. Stavtrup failed to take object types into account so that his system was far less flexible than the mechanisms described here. He also made the - not uncommon - mistake of tying his innovative linguistic mechanism up with a peculiar design methodology and a proprietary toolset.

FFPL [Francois French and Paul Lawson : "A language for Free Form Programming." POPL, 1992] and White [G. LeBlanc: "Whitespace overloading as a fundamental language design principle." JIR, Vol. 24, no. 3, May 1994] were academic projects that never had any users - except possibly their designers. It is not clear that White was ever implemented and Dr. Wimmelskaft of the university of Horsens , Denmark, have conjectured that it was, in fact, unimplementable [Wimmelskaft: "A refutation of White." JIR, Vol. 26, no. 4, March 1996].

The overload mechanism described here generalizes the built-in use of concatenation for string literals in C and C++. In particular, space is predefined to mean C-style string concatenation. For example,

```
"this is" "a single " "C-style string"
```

is by the lexical analyzer turned into

```
"this is a single C-style string"
```

Thus whitespace between two C-style string literals is interpreted as concatenation. The facility was missing in K&R C, introduced into ANSI C, and adopted by C++ in the ARM (Ellis and Stroustrup: "The Annotated C++ Reference Manual, " Addison-Wesley 1989).

Overloading Separate Forms of Whitespace

There are of course several forms of whitespace, such as space, tab, // comments, and /* */ comments. A comment is considered a single whitespace character. For example,

```
/* this comment is considered a single character
   for overloading purposes
*/
```

It was soon discovered that it was essential to be able to overload the different forms of whitespace differently. For example, several heavy users of whitespace overloading found overloading of newline ('\n'), tab ('\t'), and comments as the same arithmetic operator is counterintuitive and error prone. Consider:

```
double z1 = x y;   // obvious
double z2 = x
        y;         // obscure
double z3 = x /* asking for trouble */ y;
```

In addition, different overloading of different whitespace characters can be used to mirror conventional two-dimensional layout of computations (see below).

Stavtrup claimed that it was important to distinguish between different number of adjacent whitespace characters, but we did not find that mechanism useful. In fact, we determined it to be error-prone and omitted for Standard C++.

Overloading Missing Whitespace

After some experimentation, it was discovered that the overloading mechanism described so far did not go far enough. When using the mechanism, the physicists tended to omit the space character and write

```
xy
```

rather than

```
x y
```

This problem persisted even after the overloading rules had been clearly and repeatedly explained. What was needed wasn't just the ability to overload explicit use of whitespace, but also implicit application. This is easily achieved by modifying the lexical analyzer to recognize

```
xy
```

as the two tokens

```
x y
```

when x and y are declared. The "missing whitespace" between two identifiers are assumed to be a space.

Deciding how to resolve the ambiguity that arise for xy when x, y, and xy are all declared was one of the hardest issues to resolve for the whitespace overloading design.

One obvious alternative is to apply the "Max Munch" rule (also know as the greedy parsing rule) to this so that xy means the single identifier xy rather than x y. However, this has the unfortunate effect that the declaration of xy can completely change the meaning of a conforming program. That is, adding "int xy;" to

```
int x, y;
// …
int z = xy;          // means x y
```

yields

```
int x,y,xy;
// …
int z = xy;          // means xy
```

when space is overloaded to mean multiplication. It was therefore decided that the "Max Munch" resolution was unsuitable for large-scale programming.

Instead, it was decided to by default limit identifiers to a single character:

```
int xy;    // error: two-character identifier
```

This may seems Draconian at first . However, since we now have the full Unicode character set available, we don't actually need hard-to-read long names. Such long names only make code obscure by causing unpleasantly long lines and unnatural line breaks. Multi-character names are a relic of languages that relied heavily on global name and encouraged overly-large scopes.

Mathematicians and physicists in particularly appreciate the ability to use Greek letters:

```
double β = φλ;
```

This facility was also an instant success in China and Japan where the Chinese character set provides a much richer set of single characters than does the various Latin alphabets.

Less traditional symbols are also useful. For example:

```
☎->✆(); // take my phone (☎) off hook (✆)
```

This example become even more natural when - as is common - the whitespace operator is overloaded to mean -> for the telephone class:

```
class Phone {
        // …
        Phone* operator ' ' () { return this->operator->(); }
        void ✆();            // off-hook
        // …
};
```

```
Phone ☎;

☎✆();    // take phone (☎) off hook (✆)
```

It is also common to overload newline to mean application without arguments, that is (), so that what used to be the long-winded and ugly

```
my_phone->off_hook();
```

becomes plain and simple

```
☎✆;
```

Finally, semicolon is most often redundant as a statement terminator so the grammar has been improved to make it optional in most context. Thus, we get:

```
☎✆
```

Extensive use of such special characters together with imaginative and thoughtful use of whitespace overloading has had an immense impact on maintenance cost.

Should you feel the need for longer names - for example, if you don't have a high-resolution screen with a suitable large character set available - you can explicitly specify one using the multi-character identifier operator $:

```
double $xy = 0.0;          // explicitly multi-character name

double x, y;

double Φ = xy x y;                // xy times x times y
```

Naturally this is best avoided. For compatibility, a $ as the first character of a translation unit means that every identifier can be implicitly multi-character. This has proven immensely useful during transition from the old to the new rules. As an alternative to $ as the first character, the header <> can be included:

```
#include<>
```

Overloading \\ (double backslash) to mean "everything before this is a comment" has proven another useful transition tool. It allows old-style and new-style code to coexist:

```
        my_phone->off_hook(); // \\ ☎✆
```

Given a new-style compiler, everything up until the ☎ is ignored whereas an old-style compile ignores everything after the ;

Composite Operators

As described in the companion paper, C++2000 adopts a variant of the overloading of composite operators described in the ARM. This implies that we can define the meaning of

```
x = a*b + c;
```

directly by a single

```
    operator = * + (Vector&, const Matrix&, const Vector&, const Vector&);
```

rather than achieving this indirectly though function objects as described in Stroustrup: The C++ Programming Language (3rd edition). Addison-Wesley 1997.

Naturally, a composite operator can contain whitespace operators. For example,

```
    x = ab + c;
```

can be handled by

```
    operator = ' ' + (Vector&, const Matrix&, const Vector&, const Vector&);
```

where multiplication is as usual represented by concatenation (missing whitespace). Some people go further by representing addition by newline to match the common convention of listing numbers in a column before adding them. Doing that we can define:

```
    operator = ' ' '\n' (Vector&, const Matrix&, const Vector&, const Vector&);
```

to handle

```
    x = ab
        c;   // old-style: x = a*b+c
```

This convention is not universally appreciated and more experience is needed to estimate its impact on maintainability.

Availability

The generalized overloading mechanism described here has been in experimental use for some time and it is expected that most major C++ compiler vendors will ship it as an integral part of new releases in the near future. A preprocessor that implements the facility for any current C++ implementation can be freely downloaded from http://www.research.att.com/~bs/whitespace.html.

In addition to the overloading of missing whitespace, etc., this distributed version includes overloading based on the color of identifiers. Due to the limitations of the printing process used for this article, I cannot give examples, but basically a red x is obviously a different identifier to a green x. This is most useful for making scope differences obvious. For example, I use black for keywords, red for global variables (as a warning), blue for member names, and green for local variables. In all, a given character can be of one of 256 colors. Naturally, this again reduces the need for multiple-character identifiers while increasing readability. The lack of universal availability of color printers and problems of color blind programmers caused me to leave this feature out of the standard.

Current and Future Work

In preparation for standardization, formal specifications of the overloading mechanism in VDF and Z are being constructed. In addition, a simplified teaching environment is being constructed where operators such as *, +, and -> have been eliminated in favor of overloaded whitespace. Initial results indicates that this immensely shortens the time needed to learn C++ and should possibly be compulsory for non-expert programmers. A tool to automatically convert of old-style programs to new-style programs is being constructed; the inverse tool will not be needed.

Naturally, whitespace overloading is essentially language independent. Consequently, we are looking for ways of applying it uniformly across several programming languages to achieve common semantics. In addition, whitespace overloading clearly fit the C9x effort to support traditional numeric programming. Consequently, I confidently predict that the basic whitespace overloading mechanism will be part of the revised C standard.

Finally, work is underway to extend the character set, language syntax, and overloading rules to take advantage of 3D display devices. This will allow us to naturally represent multiplication, addition, and exponentiation as spatial displacements along three different axis. Because this project relies of the ability to fool the brain into accepting a projected image as 3D and because we don't take delivery of the 3D projection device until next spring, this project is usually referred to as "Project April Fool."