From: Frank Farance, William Rugolsky Jr.
Organization: Farance Inc.
Telephone: +1 212 486 4700
Fax: +1 212 759 1605
E-mail: frank@farance.com, rugolsky@farance.com
Date: 1995-12-22
Document Number: WG14/N525 X3J11/95-126
Subject: Issues on overloading

The purpose of overloading is: (1) to create alternate meanings for a symbol, based upon the context (2) to allow the translator to determine the context.

We let the translator determine context because: it may be error prone for the programmer to determine the context, or it may be difficult for the programmer to determine. For example, when writing an overloaded square root function, "square_root(X)", if we had to change the function name every time the type of X changed, the result would be application programmers would make mistakes matching the type of X to the function type.

Another solution might be to always promote (or demote) to a known type (e.g., the C "sqrt()" function promotes its argument to "double"). The problems here are loss of information (by demotion to narrower types), unexpected results (by promotion to wider types), or inefficiency (wider type takes more space and/or time).

The following are the goals of implementing an overloading mechanism:

- Determine what symbols can be overloaded (operator, function).
- Determine what meanings are associated with the symbol.
- Determine how the context points to a particular meaning.
- Determine how the compiler generates internal (linker) names for the different meanings.

## OPERATORS VS. FUNCTIONS

There isn't much difference between overloading operators and functions. The primary difference is syntax.

## MEANINGS AND CONTEXT

There are many ways to describe the alternative meanings. In C++, the meanings are itemized individually. There is a one-to-one correspondence between the context (the prototype) and the meaning (its body). This simple technique, however, is flawed when we have hundreds of meanings for a symbol, e.g., arithmetic operations typically have more than a hundred meanings (12 * 12 possibilities: char, signed/unsigned char, signed/unsigned short,

signed/unsigned int, signed/unsigned long, float, double, long double). In fact, the translator varies on how this is implemented for arithmetic operators for C types: some implement all cases (itemize all meanings), some implement only a subset (promotion tiers), and some implement a subset plus certain special optimizations.

Given that translators do this differently, we should expect that programmers would want to do this differently, too.

- Itemize all cases. This is how it is done in C++.

- Implement some cases. This could be done in C++, but there are some cases recognizing a ''pattern'' (e.g., the type is "unsigned" something).

- Implement some cases and some special cases. This may require identifying certain patterns and a list of special cases -- all pointing to the same meaning.

When there is a one-to-one correspondence between contexts and meanings, then the solution is straightforward: connect the context (prototype) to the meaning (body). This (the C++ overloading technique) looks like a typical C function, except that there are multiple definitions for the function, one for each different context (prototype).

But what about multiple contexts that map into the same meaning, multiple contexts that map into similar meanings, or multiple contexts that are too numerous to itemize (e.g., arithmetic operations)? To address these problems, we need to add two new features: (1) multiple contexts that map into a single meaning, (2) inspecting the context to determine the meaning in an ad hod way. This technology is fairly old and implemented in most commands on most operating systems: the command determines the context (based on the number of parameters, the type of paramters, and options associated with the parameters) and then determines the appropriate meaning. Not only is the technology old, but these programming techniques are known to almost all application programmers: they know how to process "argc" and "argv" in the "main" function. For example, a function for exponent might handle the following patterns:

(1) Both the base and the exponent are integral types. Use the normal multiplication and looping (negative exponents just return zero) to produce the result.

(2) The exponent is an integral type and the base is a floating type. Use the normal multiplication, division, and looping to produce the result.

(3) The exponent is a floating type. Use the "pow", "powf", or "powl" functions to produce the result.

In some cases the may be specialized versions of the "pow" functions available to handle mixed arguments (e.g., a "float" and a "long double").

Unfortunately, if we don't give the programmers to expressive notation that matches their needs (and, presumably, it addresses their performance concerns) then they will either produce automatically-generated volumes of code (large, hard to maintain, error-prone to program) or avoid overloading and require the application programmer to explicitly identify the meaning (error-prone to program, hard to maintain, current practice in C).

INTERNAL NAMES

For each of the function bodies (possibly associated with more than one context or meaning), the function must be given a unique name so the linker can distinguish among the function bodies. This wouldn't be a problem for a single source module because the compiler could generate the correct hidden name and map the context to the meaning. However, these hidden names must be agreed upon across source modules, especially if they are compiled at different times (e.g., libraries). There are several techniques avilable to handle this problem:

(1) Mangle the names using a well known algorithm so that the context (i.e., prototype) is mapped (hashed) into the same abbreviation each time, regardless of which source module is compiled or when it is compiled.

(2) Allow overloading only in inline functions. Since inline functions have no external linkage, there isn't any need for an developing an agreed upon algorithm.

(3) Allow the programmer to specify the name of the context. For example:

```
int exponent "int" 1 (int,int);
int exponent "int" 2 (short,int);
int exponent "int" 3 (int,short);
int exponent "int" 4 (long,int);
int exponent "int" 5 (int,long);
float exponent "float" 6 (float,float);
double exponent "double" 7 (float,double);
double exponent "double" 8 (double,float);
double exponent "double" 9 (double,double);
long double exponent "long double" 10 (float,long double);
long double exponent "long double" 11 (double,long double);
long double exponent "long double" 12 (long double,float);
long double exponent "long double" 13 (long double,double);
long double exponent "long double" 14 (long double,long double);
```

The "exponent" function has four bodies associated

with is: "int", "float", "double", and "long double". There are 14 meanings associated with "exponent".

These functions would actually be implemented with a "va_list":

```
int exponent "int" (int context, ...)
{
        switch ( context )
        {
        case 1: /* get int and int */ break;
        case 2: /* get short and int */ break;
        case 3: /* get int and short */ break;
        case 4: /* get long and int */ break;
        case 5: /* get int and long */ break;
        }
        /* ... */
}
```

Here, this function's internal name is, say, "exponent__int". The "context" parameter is passed, determined by which context was matched. From an implementation perspective, the context doesn't have to be passed as a parameter: there could be multiple entry points to the function or a jump table within the function. Regardless, it is possible to implement this without any run-time overhead (including avoiding the switch statement), yet with no changes to the linker.

In summary, the C++ technique is understood, but requires all meanings to be itemized, 2) requires name mangling, may exceed the capability of linkers. The inline technique eliminates linker problems, but still requires itemizing all the combinations of meanings. The explicit prefix method allows the user to specify the prefix (user-supplied mangling) and allows for the collapse of multiple cases into a single case.