Parameterized Types for C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Type parameterization is the ability to define a type in terms of another, unspecified, type. Versions of the parameterized type may then be created for several particular parameter types. A language supporting type parameterization allows specification of general container types such as *list*, *vector*, and *associative array* where the specific type of the elements is left as a parameter. Thus, a parameterized class specifies an unbounded set of related types; for example: list of int, list of name, list of shape, etc. Type parameterization is one way of making a language more extensible.

In the context of C++, the problem are

[1] Can type parameterization be easy to use?
[2] Can objects of a parameterized type be used as efficiently as objects of a ''hand-coded'' type?
[3] Can a general form of parameterized types be integrated into C++?
[4] Can parameterized types be implemented so that the compilation and linking speed is similar to that achieved by a compilation system that does not support type parameterization?
[5] Can such a compilation system be simple and portable?

A design is presented for which the answer to all of these questions is *yes*. The implementation of this scheme is a fairly simple extension of current C++ implementations.

WARNING: The scheme for providing parameterized types described here is not implemented. It is not part of the C++ language, nor is there any guarantee that it ever will be. This paper was written to stimulate and focus discussion about the usefulness of a parameterized type facility for C++ and about the possible forms such a facility might take.

1 Introduction

For many people, the largest single problem using C++ is the lack of an extensive standard library. A major problem in producing such a library is that C++ does not provide a sufficiently general facility for defining ''container classes'' such as lists, vectors, and associative arrays. There are two approaches for providing such classes/types:

[1] The Smalltalk approach: rely on dynamic typing and inheritance.

[2] The Clu approach: rely on static typing and a facility for arguments of type *type*.

The former is very flexible, but carries a high run-time cost, and more importantly defies attempts to use static type checking to catch interface errors. The latter approach has traditionally given rise to fairly complicated language facilities and also to slow and elaborate compile/link time environments. This approach also suffered from inflexibility because languages where it was used, notably Ada, had no inheritance mechanism.

Ideally we would like a mechanism for C++ that is as structured as the Clu approach with ideal run-time and space requirements, and with low compile-time overheads. It also ought to be as flexible as Smalltalk's mechanisms. The former is possible; the latter can be approximated for many important cases.

Note that C++ appears to have sufficient expressive power to cope with the demands of library writing

provided there is a single basic kind of object, such as a character (for string manipulation, pattern matching, character I/O, etc.), a double precision floating point number (for engineering libraries), or a bitmap (for graphics libraries). The ‘‘container class problem’’ is particularly serious, though, since container classes are needed to specify all but the simplest interfaces; they are the ‘‘glue’’ for larger systems.

2 Class Templates

A C++ parameterized type will be referred to as a class template. A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed. A vector class template might be declared like this:

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

The `template <class T>` prefix specifies that a template is being declared and that an argument `T` of type *type* will be used in the declaration. After its introduction, `T` is used exactly like other type names within the scope of the template declaration. Vectors can then be used like this:

```
vector<int> v1(20);
vector<complex> v2(30);

typedef vector<complex> cvec;   // make cvec a synonym for vector<complex>
cvec v3(40);            // v2 and v3 are of the same type

v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

Clearly class templates are no harder to use than classes. The complete names of instances of a class template, such as `vector<int>` and `vector<complex>`, are quite readable. They might even be considered more readable than the notation for the built-in array type: `int[]` and `complex[]`. When the full name is considered too long, abbreviations can be introduced using `typedef`.

It is only trivially more complicated to declare a class template than it is to declare a class. The keyword `class` is used to indicate arguments of type *type* partly because it appears to be an appropriate word, partly because it saves introducing a new keyword. In this context, `class` means ‘‘any type’’ and not just ‘‘some user-defined type.’’

The `<...>` brackets are used in preference to the parentheses `(...)` partly to emphasize the different nature of template arguments (they will be evaluated at compile time) and partly because parentheses are already hopelessly overused in C++.

The keyword `template` is introduced to make template declarations easy to find, for humans and for tools, and to provide a common syntax for class templates and function templates.

3 Member Function Templates

The operations on a class template must also be defined. This implies that in addition to class templates, we need function templates. For example:

```
template<class T> T& vector<T>::operator[](int i)
{
    if (i<0 || sz<=i) error("vector: range error");
    return v[i];
}
```

A function template is a specification of a family of functions; `template<class T>` specifies that `T` is a template argument (of type *type*) that must somehow be supplied to specify a particular function.

Note that you don't usually have to specify the template arguments to use a function template. For example, the template argument for `vector<T>::operator[]` will be determined by the vector to which the subscripting operation is applied:

```
vector<int> v1(20);
vector<complex> v2(30);

v1[3] = 7;       // vector<int>::operator[]()
v2[3] = complex(7,8);   // vector<complex>::operator[]()
```

Member functions of a class template are themselves function templates with the template arguments specified in the class templates. Function templates and member function templates will be discussed in greater detail in §9 and §12.

4 Outline of an Implementation

The basic idea for an implementation that incurs no additional costs in run-time or space compared with ''hand coding'' is to ''macro-expand'' a template for each different set of template arguments with which it is used. Naturally, template expansion is not really/just macro expansion; it obeys proper scope and syntax rules. Names such as `vector<int>` can be encoded into composite class names such as `__PTvector_int`.

The example above expands into:

```
class __PTvector_int {
    int* v;
    int sz;
public:
    __PTvector_int(int);
    int& operator[](int);
    int& elem(int i) { return v[i]; }
    // ...
};

class __PTvector_complex {
    complex* v;
    int sz;
public:
    __PTvector_complex(int);
    complex& operator[](int);
    complex& elem(int i) { return v[i]; }
    // ...
};

__PTvector_int v1(20);
__PTvector_complex v2(30);
__PTvector_complex v3(40);

v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

A compiler need not have a separate template expansion pass. Since the information to do such an expansion exists in the compiler's tables, the appropriate actions can simply be taken at the proper places in the analysis and code generation process.

In addition to this expansion mechanism, a facility is needed for detecting which member functions have been used for which instances of a parameterized type. The example above used:

```
__PTvector_int::__PTvector_int();       // constructor
__PTvector_complex::__PTvector_complex();   // constructor
__PTvector_int::operator[]();           // subscripting
__PTvector_complex::operator[]();       // subscripting
__PTvector_complex::elem();
```

Note that the full list of such functions for a program can be known only after examining every source file. The linker provides a form of this list as part of its list of undefined objects and functions.

The definition of an operation on a class template might look like this:

```
template<class T> T& vector<T>::operator[](int i)
{
    if (i<0 || sz<=i) error("vector: range error");
    return v[i];
}
```

From this, the following two function definitions will have to be generated:

```
int& __PTvector_int::operator[](int i)
{
    if (i<0 || sz<=i) error("vector: range error");
    return v[i];
}

complex& __PTvector_complex::operator[](int i)
{
    if (i<0 || sz<=i) error("vector: range error");
    return v[i];
}
```

This approach ensures that no run-time efficiency is lost compared to ‘‘hand-coding’’. Code space might wasted by creating separate copies of functions that could have shared implementation. For example, `vector<int>` and `vector<unsigned>` need not have separate subscripting operations. Such waste can, if necessary, be reduced through suitable coding practices (see § 11) and/or through a clever compile time environment.

A programmer can provide a definition for a particular version of an operation on a class by specifying the template argument(s) in a function definition:

```
int& vector<int>::operator[](int i) { return v[i]; }
```

The general version of such a function as defined by its template will be used to create a function for a particular argument type only when no user-provided version is specified for that type.

Replacing the default implementation of a function as defined by a template is useful where implementations with greater precision, higher efficiency, etc. can be provided given some understanding of a particular type. It may also be useful for debugging and for supplying different versions of a function to different parts of a program (using `static` functions).

5 Some Design Considerations

Let us consider a few choices that were made to write the example above:

[1] Should all template arguments be of type *type*?

[2] Should a user be required to specify the set of operations that may be used for a template argument of type *type*?

[3] Should a user be required to explicitly declare what versions of a template can be used in a program?

[4] Should it be possible for a user to declare variables of type *type*?

The answer to all (in the context of C++) is *no*. Let us examine them in turn.

5.1 Template Arguments

‘‘Should all template arguments be of type *type*?’’ No, there appear to be useful examples of type parameters of ‘‘normal’’ types. For example, a vector template might be parameterized with an error handling function:

```
typedef void (*PF)(char*);

template<class T, PF error> class vector {
    T* V;
    int sz;
public:
    T& operator[](int i) {
        if (i<= || sz<=i) error("vector: range error");
        return v[i];
    }
};

void my_error_fct() { ... }
vector<complex,&my_error_fct> v(10);
```

This example implies that default arguments might be useful:

```
template <class T, PF error=&standard_error_fct> class vector { ... }
```

Another example is a buffer type with a size argument:

```
template<class T, int sz=128> class buffer {
    T v[sz];
    // ...
};

void f()
{
    buffer<char> buf1;
    buffer<complex,20> buf2;
    // ...
}

buffer<char*,1000> glob;
```

Making `sz` an argument of the template `buffer` itself rather than of its objects implies that the size of a `buffer` is known at compile time so that a `buffer` can be allocated without use of free store. It appears that default arguments will be at least as useful for template arguments as they are for ordinary arguments. Default arguments of type *type* might even be useful:

```
template<class T, class TEMP = double> class store {
    // ...
    T sum() { TEMP sum = 0; ... return sum; }
};

store<int,long> istore;
store<float> fstore;
```

These examples demonstrate that the range of templates with which a type can be parameterized should be restricted only if there are compelling arguments that the restriction will significantly ease the implementation of templates. I see no such argument.

5.2 Type Argument Attributes

‘‘Should a user be required to specify the set of operations that may be used for a template argument of type *type*?’’ For example:

```
    // The operations =, ==, <, and <=
    // must be defined for an argument type T

template <
        class T {
            T& operator=(const T&);
            int operator==(const T&, const T&);
            int operator<=(const T&, const T&);
            int operator<(const T&, const T&);
        };
    >
    class vector {
    // ...
    };
```

No. Requiring the user to provide such information decreases the flexibility of the parameterization facility without easing the implementation or increasing the safety of the facility.

Consider `vector<T>`. To provide a sort operation one must require that type `T` has some order relation. This is not the case for all types. If the set of operations on `T` must be specified in the declaration of `vector` one would have to have two vector types: one for objects of types with an ordering relation and another for types without one. If the set of operations on `T` need not be specified in the declaration of `vector` one can have a single vector type. Naturally, one still cannot sort a vector of objects of a type `glob` that does not have an order relation. If that is tried, the generated sort function `vector<glob>::sort()` would be rejected by the compiler.

It has been argued that it is easier to read and understand parameterized types when the full set of operations on a type parameter is specified. I see two problems with this: such lists list would often be long enough to be de facto unreadable and a higher number of templates would needed for many applications.

Should experience show a need for specifying the operations on a parameterized type then such a facility can be easily and compatibly added later.

5.3 Source Code

There might be a more fundamental reason for requiring that the operations performed on a template argument of type *type* should be listed in the template declaration. The implementation technique outlined here achieves near optimal run-time characteristics by requiring the complete source code of a template to be available to the compiler when processing a use of the template. In some contexts, this is considered a deficiency and an implementation of templates that requires only the object code for functions implementing the function templates would be preferable.

At first glance it would appear that requiring the full set of operations on a template argument to be specified would make it significantly easier to produce such an implementation. In this case, a function template would be implemented by code using calls through vectors of function pointers to perform operations on template arguments of type *type*. The specification of the set of operations for a *type* argument would provide the definition for such vectors. Such an implementation would trade run-time for compile and link time, but would be semantically equivalent to the implementation scheme presented here.

Could an implementation along these lines be provided without requiring the user to list the set of operations needed for each function template argument of type *type*? I think so. Given a function template, the compiler can create a vector layout for the required set of operations without the help of a user. Given the full set of function definitions for the members of a class, the compiler can again create a vector layout for the required set of operations without the help of a user. If the compile and link environment cannot provide such a list a less optimized scheme where each member function has its own vector of operations can be used.

It thus appears that both implementation styles can be used even in the absence of template argument attribute lists so that we need not require them to preserve the implementers' freedom of action. It might be noticed that a virtual function table is in many ways similar to a vector of operations for a template so that the benefits of the vector of operations approach can often be achieved by a coding style relying on virtual functions rather than the expansion of function templates. Class `pvector` presented in §11 is an example of this.

5.4 Type Instantiation

''Should a user be required to explicitly declare what versions of a template can be used in a program?'' For example, should one require the use of an operation like Ada's `new`? No. Such a requirement would increase the size of the program text and decrease the flexibility of the template facility without yielding any benefits to the programmer or the implementer.

5.5 Type Variables

''Should it be possible for a user to declare variables of type *type*?'' For example:

```
type t = int;

void f(type t)
{
    switch (t) {
    case int:
        ...
    case char*:
        ...
    case complex:
        ...
    default:
        ...
    }
}
```

Such a facility would be useful in many contexts, but does not appear suitable for C++. In particular, it is not possible to assign integer values to represent constants of type *type* such as `int`, `line_module*`, `double(*)(complex*,int)`, and `vector<complex>` while maintaining the current style of separate compilation. Since the C++ type system is open such assignment of values in general requires an unbounded number of bits to represent a type. In practice, even simple cases require lots of bits (the current cfront scheme for encoding function types in character strings regularly uses dozens of characters) or some system of hashing involving a database of types. Furthermore, the introduction of such variables would require an order of magnitude greater changes to the C++ language and its implementations than the scheme (without type variables) described here.

6 Type Inquiries

It would be possible to enable a programmer to inquire about properties of a template argument of type *type*. This would allow the programmer to write code that depends on specific properties of the actual types used.

6.1 An Inquiry Operator

Consider providing a print function for a vector type that sorts the elements before printing if and only if sorting is possible. A facility for inquiring if a certain operation, such as `<`, can be performed on objects of a given type can be provided. For example:

```
template<class T> void vector<T>::print()
{
    if (?T::operator<) sort();  // if (T has a <) sort_this_vector
    for (int i=0; i<sz; i++) { ... }
}
```

Because the `<` operation is defined for *int*s, printing of a `vector<int>` gives rise to an expansion:

```
void __PTvector_int::print()
{
    sort();     // that is, this->sort()
    for (int i=0; i<sz; i++) { ... }
}
```

On the other hand, printing a `vector<glob>` where the `<` operation is not defined for `glob`s gives rise

to an expansion:

```
void __PTvector_glob::print()
{
    for (int i=0; i<sz; i++) { ... }
}
```

Tests on expressions of the form *?typ::oper* (''does type *typ* have an operation *oper*?'') must be evaluated at compile time and can be part of constant expressions.

It would probably be wise *not* to include such a type inquiry feature in the initial experimental implementation but to wait and see what properties (if any) programmers would find useful. Potentially every aspect of a type known to the compiler can be made available to the programmer; `sizeof` is a most rudimentary version of this kind of facility.

The absence of a type inquiry facility will be compensated for by the ability to define a function for a particular set of template arguments, thus overriding the generation of the ''standard'' version from the template. Furthermore, it can sometimes be preferable to define separate templates to represent the different concepts. For example, one might have both a `vector<T>` class and a `sorted_vector<T>` class derived from it.

6.2 The `typeof` Operator

Writing code where the control flow depends of the properties of a type parameter doesn't appear to be necessary, but defining variables of types dependent on type parameters does. Given a template argument of type *type*, `T`, one can express a variety of derived types using the declarator syntax; for example, `T*`, `T&`, `T[10]`, `T(*)(T,T)`. One can also express types obtained by template expansion such as `vector<T>`. However, this does not conveniently express all types one might like. In particular, the ways of expressing types that depends on two or more template arguments are weak. To compensate, one might introduce a `typeof` operator that yields the type of its argument. For example:

```
template<class X, class Y> void f(X x, Y y)
{
    typeof(x*y) temp = x*y;
    // ...
}
```

It would probably be wise *not* to introduce a `typeof` operator before gaining more experience. The uses of `typeof` appears to be quite limited and the scope for misuses large. In particular, `typeof` appears more suited for the writing of macros (which templates are designed to replace in many contexts) than for templates and heavy use of typeof will reduce the compilers ability to pinpoint type errors.

7 More about Implementation

So how can we generate the proper code for definitions of operations on a template for a given set of arguments? Assume that we know that versions of `vector`'s subscripting operation

```
template<class T> vector<T>::operator[](int) { ... }
```

are needed for `T==int` and `T==complex`. How can we create the proper expansions (as presented above)?

We might have a compiler option, `-X`, for creating such expansions. Assuming that the definitions for `vector`'s member functions resides in a file called `vector.c`, one might call the compiler like this:

```
CC -X "vector<int>" vector.c
CC -X "vector<complex>" vector.c
```

and have the appropriate `.o` files created. This would create not only the required subscript operator functions but also functions for any other vector operation that has its definition stored in `vector.h`. The strategy for splitting a program into separately compiled parts is in the hands of the programmer. Where a finer granularity is required of `.o` files for a library, the programmer can handle it using standard C library techniques.

Note that an expansion using the template expansion option, `-X`, may give rise to a program that uses an

instance of a template that has not already been used in the program. This implies that another stage of ‘‘missing template implementation detection’’ is required after each expansion. Expansion is really a recursive activity. The depth of this recursion will typically be 1, though. It will be necessary to have a mechanism protecting against recursive expansion. For example:

```
template<class T> void f(T a) { T* p; ... f(p); }
```

Naturally, one would try to ensure that `CC -X` is used to generate `.o` files only for definitions of templates when

[1] a new template was used, or

[2] a new set of template arguments was used, or

[3] the declaration of a template was changed.

I imagine that after a short startup period all the necessary `.o` files for templates for a program/project will reside in a library and not interfere with the compilation process. When a program/project reaches this state the compilation overhead incurred by using templates becomes negligible.

7.1 Tools for Ensuring Consistent Linking

Consider having the tools described above:

[1] a C++ compiler handling the expansion of class templates into class declarations, and

[2] a `-X` option on this compiler to handle the expansion of function templates into function definitions.

One could then compile a C++ program using templates. A little manual intervention would be needed to get a complete program to link and load.

What additional tools would be needed to

[1] guarantee consistent and complete expansion and linking?

[2] make programming reasonably convenient?

I conjecture that [1] is perfectly feasible, but non-trivial, where portability across operating systems, compile and link time efficiency, and flexibility are all required. I also conjecture that very little is needed to achieve [2]. Experience using such a system is clearly needed, but it might well be sufficient to modify a tool with access to the complete compiled program, such as `munch` or the linker itself, to produce

[1] a list of function definitions required, or

[2] a list of files for which `CC -X` needs to be run (assuming some correspondence between type names and file names), or

[3] a make script for running `CC -X` for an appropriate set of files.

It would also be important to ensure that `CC` produces readable error messages when an operation is applied to a particular template argument of type *type* for which it is not defined. For example:

```
"foo.c", line 144: error: operator<= applied to glob in vector<glob>::sort()
```

This discussion of how one might provide a minimal and portable mechanism supporting templates in C++ should not be taken as an indication that such a mechanism provides the ideal programming environment. On the contrary, it is exactly a *minimal* facility. Much better facilities can be built (think of a smart `make`, an incremental compiler, a Smalltalk-like browser, etc.), However, a minimal facility *must* exist to ensure portability of C++ programs between all implementations since there is no hope that a single maximal programming environment will ever be agreed on and implemented on every system supporting C++.

8 Function Templates

In addition to providing class templates, it is necessary to provide function templates. Consider providing a general `sort` function:

```
template<class T> void sort(vector<T>);
```

Given a vector `v`, one might call such a function like this:

```
sort(v);
```

The compiler can deduce the type of the sort function from the type of the vector. For example, had `v` been declared

```
vector<int> v(10);
```

the sort function `sort<int>` would have been required. On the other hand had the declaration of `v` been

```
vector<double> v(2000);
```

the sort function `sort<double>` would have been used.

8.1 Overloading

Declaring a function template is simply a way of declaring a whole bundle of overloaded functions at one time. This implies that we can use functions with arguments that can be distinguished by the overloaded function resolution mechanism only. The following function cannot be used because it takes no argument:

```
template<class T> T* create() { return (T*) malloc(sizeof(T)); }
```

The C++ syntax could be extended to cope with this by allowing the full generality of the *name*<*type*> notation so that template arguments could be supplied explicitly in a call:

```
int* pi = create<int>();    // create_int()
char* pc = create<char>();  // create_char()
```

Unless programmers define templates sensibly this form of resolution can become quite cryptic:

```
template<class X, class Y> f(Y,X);  // template argument order differs
                     // from function argument order
 ...
f<char*,int>(1,"asdf");
```

I think it would be wise not to include any explicit resolution method in an initial implementation. I suspect that the implicit resolution provided by the overloaded function resolution rules are sufficient – and more elegant – in almost all cases and it is not obvious that a mechanism for explicit overloading is worth the added complexity.

Allowing explicit resolution would imply that a C++ compiler should treat function template names differently from other names and similarly to the way keywords and class names are treated. For example, without special rules for template names the last expression above would be parsed as two comparisons and a parenthesized comma expression:

```
(g<123)>(vv,10);
```

8.2 A Problem

Consider writing a function `apply()` that applies another function to all the elements of a vector. A traditional first cut would look something like this:

```
template<class T> void apply(vector<T>& v, T& (*g)(T&))
{
    for (int i = 0; i<v.size(); i++) v[i] = (*g)(v[i]);
}
```

This follows the C style of using a pointer to function. Potential problems with this are

[1] efficiency, because there can be no inline expansion of the applied function, and

[2] generality, because standard operations of built-in types such as `-` and `~` for `int`s cannot be applied.

Naturally, these are not problems to all people. However, an ideal template mechanism will provide solutions.

8.3 A Solution

One might consider the function to be applied by `apply()` a template argument rather than a function argument:

```
template<class T, T& (*g)(T&)> void apply(vector<T>& v)
{
    for (int i = 0; i<v.size(); i++) v[i] = (*g)(v[i]);
}
```

To call `apply()` one must specify the function to be applied. Since this version of `apply()` takes only a single `vector` argument the syntax for disambiguating an overloaded function call using `<...>` must be used:

```
class X { ... };

vector<X> v2(200);

inline void hh(X&) { ... };
void gg(X&);    // not inline

apply<X,hh>(v2);
apply<X,gg>(v2);
```

Clearly, the `X` is redundant and not elegant. Since in principle each such call of `apply()` results in writing a new function `apply()` inlining can be applied where sufficient information is available. Consequently, one would expect a C++ compiler to inline `hh()` in the first call in the example above and generate a standard function call of `gg()`. The fact that function pointers and not functions are passed in C++ is at most a minor annoyance for the compiler writer.

Operators for built-in types can be considered inline functions in this context:

```
vector<int> v(100);
apply< int, &int::operator-- >(v);
```

However, as for the explicit resolution scheme itself, it remains to be seen if this degree of sophistication and complexity is actually needed.

9 Syntax Issues

Consider the declarations:

```
template<class T> class vector { ... };
template<class T> T* index<class T>(vector<T>,int);
```

[1] Why use the `template` keyword?
[2] Why use `<...>` brackets and not parentheses?
[3] Why use the `class` keyword?
[4] What is the scope of a template argument?

9.1 The `template` keyword

If a keyword is to be used `template` seems to be a reasonable choice, but it is actually not necessary to introduce a new keyword at all. For class templates, the alternative syntax seems more elegant at first glance:

```
class vector<class T> {     // possible alternative class syntax
    ...
};
```

Here the template arguments are placed after the template name in exactly the way they are in the use of a class template:

```
vector<int> vi(200);
vector<char*> vpc(400);
```

The function syntax at first glance also looks nicer without the extra keyword:

```
T& index<class T>(vector<T> v, int i) { ... }
```

There is typically no parallel in the usage, though, since function template arguments are not usually

specified explicitly:

```
int i = index(vi,10);
char* p = index(vpc,29);
```

However, there appears to be nagging problems with this ''simpler'' syntax. It is too clever. It is relatively hard to spot a template declaration in a program because the template arguments are deeply embedded in the syntax of functions and classes and the parsing of some function templates is a minor nightmare. It is possible to write a C++ parser that handles function template declarations where a template argument is used before it is defined, as in `index()` above. I know, because I wrote one, but it is not easy nor does the problem appear amenable to traditional parsing techniques. In retrospect, I think that not using a keyword and not requiring a template argument to be declared before it is used would result in a set of problems similar to those arising from the clever and convoluted C and C++ declarator syntax.

9.2 <...> vs (...)

But why use brackets instead of parentheses? As mentioned before, parentheses already have many uses in C++. A syntactic clue (the `<...>` brackets) can be useful for reminding the user about the different nature of the type parameters (they are evaluated at compile time). Furthermore, the use of parentheses could lead to pretty obscure code:

```
template(int sz = 20) class buffer {
    buffer(int i = 10);
    // ...
};

buffer b1(100)(200);
buffer b2(100);     // b2(100)(10) or b2(20)(100)?
buffer b3;      // legal?
```

These problems would become a serious practical concern if the notation for explicit disambiguation of overloaded function calls were adopted. The chosen alternative seems much cleaner:

```
template<int sz = 20> class buffer {
    buffer(sz)(int i = 10);
    // ...
};

buffer b1<100>(200);
buffer b2<100>;     // b2<100>(10)
buffer b3;      // b3<20>(10)
buffer b4(100);     // b4<20>(100)
```

9.3 The `class` keyword

Unfortunately, the ideal word for introducing the name of a parameter of type *type*, that is, `type` cannot be used; `type` appears as an identifier in too many programs. Why use the `class` keyword then? Why not? Classes are already types to the extent that the built-in types can be considered second class citizens in some contexts (you cannot derive a class from a built in type, you cannot take the address of an operation on a built-in type, etc.). What is done here is simply to use `class` in a slightly more general form than is done elsewhere.

9.4 Scope of Template Argument Names

The scope of a template argument name is the template declaration and the template name obeys the usual scope rules:

```
const int T;

template<class T>   // hides the const int T
class vector {
    int sz;
    T* v;
public:
    // ...
};

int T2 = T;     // here const int T is visible again
```

Template declarations may not be declaration lists:

```
template<class T> f(T*), g(T);  // error: two declarations
```

This restriction is made to avoid users making unwarranted assumptions about relations between the template arguments in the different templates.

10 Templates and Typedef

The template concept is easily extended to cover all types†. For example:

```
template<class T, int i> typedef T array[i];
 ...
array<int,10> v;    // array of 10 ints
```

This allows great freedom in defining type names. The `typedef` keyword is necessary because

```
template<class T, int i> T array[i];
```

Would define a family of arrays (all called `array`) and not a family of array type.

Consequently, only class, function, and typedef templates will be implemented.

11 Type Equivalence

Consider:

```
template<class T, int i> class X {
    T vec[i];
    // ...
};

array<int,10> x;
array<int,10> y;
array<int,11> z;
```

Here, `x` and `y` is of the same type, but `z` is of the different type. Since the template arguments used in the declarations of `x` and `y` are identical they refer to the same class. Naturally, only a single class declaration is generated by a C generating C++ compiler. On the other hand, the template arguments used in the declaration of `z` differs and gives rise to a different class.

Different template arguments give rise to different classes even if the argument is used in a way that does not affect the type of the generated class:

---

† This section has been changed since the USENIX C++++ conference proceedings version of this paper based on comments by George Gonthier.

```
template<class T, int i> class Y {
public:
    foo() { int buf[i]; ... }
};

Y<int,10> xx;
Y<int,10> yy;
Y<int,11> zz;
```

Template arguments must be types, constants, or integer expression that can be evaluated at compile time.

12 Derivation and Templates

Among other things, derivation (inheritance) ensures code sharing among different types (the code for a non-virtual base class function is shared among its derived classes). Different instances of a template do not share code unless some clever compilation strategy has been employed. I see no hope for having such cleverness available soon. So, can derivation be used to reduce the problem† of code replicated because templates are used? This would involve deriving a template from an ordinary class. For example:

```
template<class T> class vector {        // general vector type
    T* v;
    int sz;
public:
    vector(int);
    T& elem(int i) { return v[i]; }
    T& operator[](int i);
    // ...
};

template<class T>
class pvector : vector<void*> {     // build all vector of pointers
                    // based on vector<void*>
public:
    pvector(int i) : (i) {}
    T*& elem(int i) { return (T*&) vector<void*>::elem(i); }
    T*& operator[](int i) { return (T*&) vector<void*>::operator[](i); }
    // ...
};

pvector<int*> pivec(100);
pvector<complex*> icmpvec(200);
pvector<char*> pcvec(300);
```

The implementations of the three vector of pointer classes will be completely shared. They are all implemented exclusively through derivation and inline expansion relying on the implementation of `vector<void*>`. The `vector<void*>` implementation is a good candidate for a standard library.

I conjecture that many class templates will in fact be derived from another template. For example:

```
template<class T> class D : B<T> {
    ...
};
```

This also ensures a degree of code sharing.

---

† If that really is a problem: memory is cheap, etc. I think it is a problem and will remain so for the foreseeable future. People's expectations of computers have consistently outstripped even the astounding growth in hardware performance.

13 Members and Friends

Here are some more details:

13.1 Member Functions

A member function of a class template is implicitly a template with the template arguments of its class. Consider:

```
template<class T> class C {
    T p;
    T m1() { T a = p; p++; return a; }
};

C<int> c1;
C<char*> c2;

int i = c1.m1();    // int C<int>::m1() { int a = p; p++; return a; }

char* s = c2.m1();  // char* C<char*>::m1() { char* a = p; p++; return a; }
```

These two calls of `m1()` gives rise to two expansions of the definition of `m1()`.

Naturally a member template may also be declared:

```
template<class T> class C {
    template<class TT> void m(TT*,T*);
};
```

This case will be discussed below. However, explicit use of class template arguments in member function names is unnecessary and illegal:

```
template<class T> class C {
    T m<T>();               // error
};

template<class T> C<T>::m<T>() { ... }      // error

template<class T> C<T>::m() { ... }         // correct
```

This also applies to constructors:

```
template<class T> class C {
    C();            // correct, a constructor
    C<T>(int);      // error constructor
};

template<class T> C<T>::C() { ... } // correct
```

To avoid confusion it is not legal to define a template as a member with the same template argument name as was used for the class template:

```
template<class T> class C {
    template<class T> T m(T*);               // error
};
```

13.2 Friend Functions

A friend function differs from other functions only in its access to class members. In particular, a friend of a class template is not implicitly a template. Consider:

```
template<class T> class C {
    friend f1(T a);
    template<class TT> friend f2(TT a);
};
```

The definitions of `f1()` and `f2()` are legal, but clearly not equivalent.

The `friend` declaration of `f1(T)` specifies that for all types `T`, `f1<T>` is a friend of `C<T>`. For example, `f1<int>` is a friend of `C<int>`. However, `f1<int>` is not a friend of `C<double>`. The definition of `f1()` would probably look something like this:

```
template<class TT> f1(TT a) { ... };
```

The friend `f1()` need not be a template, but if it isn't the programmer might have a tedious time constructing the necessary set of overloaded functions ''by hand.''

The declaration of `f2()` specifies that for all types `T` and `TT`, `f2<TT>` is a friend of `C<T>`. For example `f2<int>` is a friend of `C<double>`.

Note that a friend function of a parameterized class need not itself be parameterized:

```
template<class T> class C {
    static int i;
    friend f() { i++; }
};
```

13.3 Static Members

Each version of a class template has its own copy of the static members of the class:

```
template<class T> class C { static T a; static int b; ... };

C<int> xx;
C<double> yy;
```

This implies allocation of the static variables:

```
static int C<int>::a;
static int C<int>::b;

static double C<double>::a;
static int C<double>::b;
```

Similarly, each version of a parameterized function has its own copy of static local variables:

```
template<class T> f() { static T a; static int b; ... };
```

13.4 Friend Classes

Friend classes can (as usual) be declared as a shorthand for declaring all functions friends:

```
template<class T> class C {
    friend template<class TT> class X;  // all X<TT>s
    friend class Y<T>;          // only Y<T>
    friend class Z<int>;            // only Z<int>
};
```

14 Examples of Templates

Here are some more examples of potentially useful templates. Versions of many of the templates used as examples in this paper have been created using macros and actually used in real programs. ''Faking'' templates using macros have been a major design technique for the template facilities. In this way the language facilities could be designed in parallel with the key examples and techniques they were to support.

An associative array:

```
template<class E, class I> class Map {
    // arrays of Es indexed by Is
    // ...
    E& operator[](I);
};
```

A ''record'' stream; the usual stream of characters is a special case:

```
template<class R> class ostream {
    // ...
    ostream<R>& operator<<(R&); // output an R
};
```

An array for mapping information from files into primary memory:

```
template<class T, int bsz> class huge {
    T in_core_buf[bsz];
    // ...
    T& operator[](int i);
    seek(long);
    // ...
```

A linked list class:

```
template<class T> class List { ... };
```

A queue tail template for sending messages of various types:

```
template<class T> class mtail : public qtail {
    // ...
    void send(T arg)
    {
        // bundle ``arg'' into a new message buffer
        // and put than on the queue
    }
};
```

A counted pointer template (for user-defined automatic memory management):

```
template<class T> class CP {
    // ...
public:
    CP();
    CP(T);
    CP(CP<T>&);
    // ...
};
```

15 Conclusions

A general form of parameterized types can be cleanly integrated into C++. It will be easy to use and easy to document. The implementation can be efficient in both run-time and space. It can be implemented portably and efficiently (in terms of compiler and link time) provided some responsibility for generating the complete set of definitions of function templates is placed on the programmer. This implementation can be refined, but probably not without loss of either portability or efficiency. The required compiler modifications are manageable. In particular, cfront can be modified to cope with templates. Compatibility with C is maintained.

16 Caveat

The key thing to get right for a C++ template facility is assuring that basic parameterized classes are implemented in an easy to use and efficient way for the relatively simple key examples. The compilation system *must* be efficient and portable at least for these examples. The most reasonable approach to building a template system for C++ would be to achieve this first, make the inevitable changes in concepts based on that experience, and proceed with more advanced features *only* as far as they makes sense *then*.

17 Acknowledgements

Andy Koenig, Jon Shopiro, and Alex Stepanov wrote many template-style macros to help determine what language features was needed to support this style of programming. Jim Coplien, Margaret Ellis, Brian Kernighan, and Doug McIlroy supplied many valuable suggestions and questions.