Issues in the Design of C++ Browsers

Darrell R. Raymond

Centre for Advanced Studies
IBM Canada Laboratory
895 Don Mills Road
North York, Ontario

ABSTRACT

This document describes problems and opportunities that arose during the design and implementation of a C++ browser constructed during the fall of 1992. The browser provides integrated access to source code and documentation. It relies on a compiler-generated database for information about the source code. Effective browsing of C++ is hampered by our lack of theory about what information should be displayed. The NIH class library is used as an example throughout the paper.

1. Introduction.

Modern software engineering extols code reuse, and hence requires an environment in which programmers can study existing code. Such environments are generally known as *browsers*. As with many other object-oriented concepts, Smalltalk provided the archetype; in its canonical form, a browser is a set of windows that shows a path in an inheritance hierarchy, revealing some detail about the classes at each step in the path. Browsers today are commonly integrated with debuggers, editors, interpreters, and other tools of a full-blown development environment.

With the burgeoning popularity of C++, many developers have rushed to construct C++ browsers, both commercially and for the public domain. Virtually every C++ development environment has at least a text-based class browser, and some have graphical browsers as well.† While many of these systems are quite powerful, they are also often very slow, complicated, monolithic, expensive, difficult to learn, idiosyncratic, tightly coupled to specific theories of program development, or too resource-intensive[17]. In a recent test, one well-known system required more than 83 Mbytes of memory to handle some 7,500 lines of code[1]. As a consequence, there is still a recurrent, plaintive appeal on electronic newsgroups, ‘‘anyone have a decent C++ browser?’’

† For a list of such environments, see the C++ Product List posted in comp.lang.c++ or available by anonymous ftp from ftp.th-darmstadt.de, in /pub/programming/languages/C++/c++-products.

Behind the appeal, of course, lies the question ‘‘what would a decent C++ browser look like?’’ Research on browsers appears to be at early stages, and most design recommendations are tentative or weakly supported. Research by some psychologists tends to suggest that browsing the code alone is insufficient, and that ancilliary documentation is equally important. Green *et al.*, for example, suggest that a ‘‘description level’’ be added to permit programmers to add facts about the code[5], while Carey and Spall suggest using inheritance to support the reuse of design rationales[2]. It is not clear if these are practical suggestions, given the tendency for documentation and software to be out-of-sync, if the documentation exists at all. However, such suggestions indicate that simply displaying the class hierarchy alone is insufficient to educate programmers about class libraries.

The prototype browser described in this paper does not solve the problem of browsing C++ code. Indeed, one of the main themes of the paper is that have yet to get a firm grasp on the problems. On the other hand, the prototype described herein does exhibit some features that (insofar as I am aware) are not present in existing browsers. Part of the reason for this novelty is that this browser is not an adaptation of the Smalltalk browser model to C++, and hence does not contain the implicit assumption that displaying the class hierarchy is tantamount to having explained the code. Instead, the fundamental assumption is that code is a kind of text, and that text-based display features ought to be of some benefit to a browser. Considering code in this way allows for the possibility of new interpretations of code, just as literary texts are

themselves subject to new interpretation.

The prototype browser was constructed as part of the 4Thought software development project at the IBM Centre for Advanced Studies. 4Thought's goal is the creation of a flexible software development environment that permits programmers to state theories about software, and treat their code as models of those theories. [16] Such theories are, in effect, database schemas, and their display and management will necessarily involve some kind of browser.

The browser's design was driven by two ideas. First, we wanted to explore the use of better text display tools than is commonly found in browsers; in particular, we wanted to encourage the use of multiple, proportional fonts, selective suppression of text, and multiple styles for viewing. These features are supported by the text display engine *lector*[14]. Second, we thought it important that the browser avoid, as far as possible, embedding specific knowledge about C++ in its design. This was important partly because 4Thought should be a language-independent environment, and partly because knowledge about C++ is already possessed by compilers and interpreters, and hence should not be duplicated in the browser itself. In particular, the IBM xlC C++ compiler can generate data about the program that is usable by other tools. A C++ file compiled with the `-qbrowse` option generates a `.brs` file that can be turned into set of Prolog facts about the source code. This database was the main source of information for our browser.

The browser is only a proof-of-concept of the new features we intended to explore. It does not attempt to support other equally interesting and useful features, such as displaying call graphs, doing selective expansion of preprocessor information, or the many other essential tasks necessary for effective code manipulation. These capabilities should be considered if the prototype is further developed.

The source code we used to test the browser is the NIH class library, a freely available library of basic container classes such as sets, bags, lists, and hash tables[4]. NIH has several virtues: it is of substantial size, it is well known, it is fairly advanced in its use of C++ features, and it is accompanied by extensive documentation.

2. Overview of the browser.

The user's view of the browser is shown in Figure 1. The browser consists of four windows. The leftmost window is an index to the class library; in this case, the NIH class library. The top right window displays source code. The middle right window displays documentation. The bottom right window displays the result of queries.

The basic function of the browser is to support access to individual classes. When the user points at a class in the index window, the documentation and source code for that class are displayed in their respective windows. The user can page through the source code or its documentation independently, and can use a variety of styles for viewing them. The styles employ multiple fonts and text suppression to provide views of the code that aid in learning about its structure.

In either the source or documentation window, the user can select a function call by highlighting the string that is its name. This results in a query to the Prolog database that returns the location of the function definition. Often there is more than one function definition with the same name; C++ permits overloading of names so long as the functions can be distinguished by the types of their arguments. In this case, a list of function definitions is shown in the query results window.

The windows are functionally coupled, but their layout is not fixed. Users can rearrange the layout of the windows, change their sizes, or change the number of columns of text displayed at any time during the browsing session. Layout preferences can be saved on an individual basis.

3. Browser structure and components.

The browser is a multiple process structure. The individual modules are described in Table 1; the data flow diagram is shown in Figure 2. The four largest boxes correspond to the (visible) invocations of *lector*. The dotted boxes correspond to the (invisible) support programs.

The underlying support programs perform two basic functions. First, they massage data streams so that they are comprehensible by succeeding programs; this typically involves adding markup or removing unnecessary diagnostics. Second, they coordinate the activities of the various processes, ensuring that data flows to the desired targets and that a minimal amount of state information is retained.

Figure 1. Screen view of the browser

Module name	Type of module	Description
Control	C program	Reads class names and passes them on to Tagger. Control also reads strings from Info and Source and sends them to Query. Control maintains a list of query results and sends them to Results for display.
C++grind	*lex* program	Generates a file of tags to be inserted in the C++ source code. The tags demarcate syntactic features of the source that are not represented in the Prolog database (e.g., braces, line breaks, indenting, comments).
Fn	*sh* script	Executes a query (*pq*) on the Prolog database, and then massages the output of the query (*awk*, *grep*, *sed*) to produce a file of tags to be inserted in the C++ source code. The tags indicate the start of function definitions in the source code.
FnCall	*sh* script	Similar to Fn, but generates tags that correspond to function invocations. The tags include the name of the function being invoked.
Grind	*lex* program	Reads a TEXinfo stream from stdin and writes a tagged version to stdout.
Index	*lector* invocation	Displays the class hierarchy, and permits the selection of a specific class for browsing.
Info	*lector* invocation	Displays the marked-up documentation files. The user may select a string from the text to be submitted to Control.
Insert	C program	Takes a file of tags to be inserted into a source file, and produces a target file with the tags inserted in the appropriate positions. The file of tags is specified as a list of 3-tuples of the form *(column, row, tag)*. Insert also generates a map of offsets that can be used to map original file positions into modified positions.
Macro	*sh* script	Similar to Fn, but marks up the macros in the source code.
Pqfilter	*sh* script	Removes Prolog diagnostics and irrelevant output.
Query	*sh* script	Reads strings from stdin and treats them as queries on the Prolog database (for function definitions). A query is submitted to the Prolog database (*pq*); its results are filtered to remove Prolog status information (*pq filter*) and then passed back to Control. Query is also responsible for starting up the name server (*pdnsd*) and the Prolog database (*pd*), and for shutting them down (*pdkill*).
Results	*lector* invocation	Displays the query results.
Source	*lector* invocation	Displays the marked-up source code. The user may select a string from the text to be submitted to Control.
Stderr	C program	Copies its stdin to its stderr. This process is necessary because a shell script cannot write output to an arbitrary descriptor (in this case, descriptor 2).
Tagger	*sh* script	Invokes several other programs to mark up the C++ source code so it can be displayed by Source.
Trans	*sh* script	Converts offsets received from Index to a string corresponding to the class name. Sends the documentation for the class to Grind for tagging, and echos the string to Stderr.

Table 1. Process descriptions.

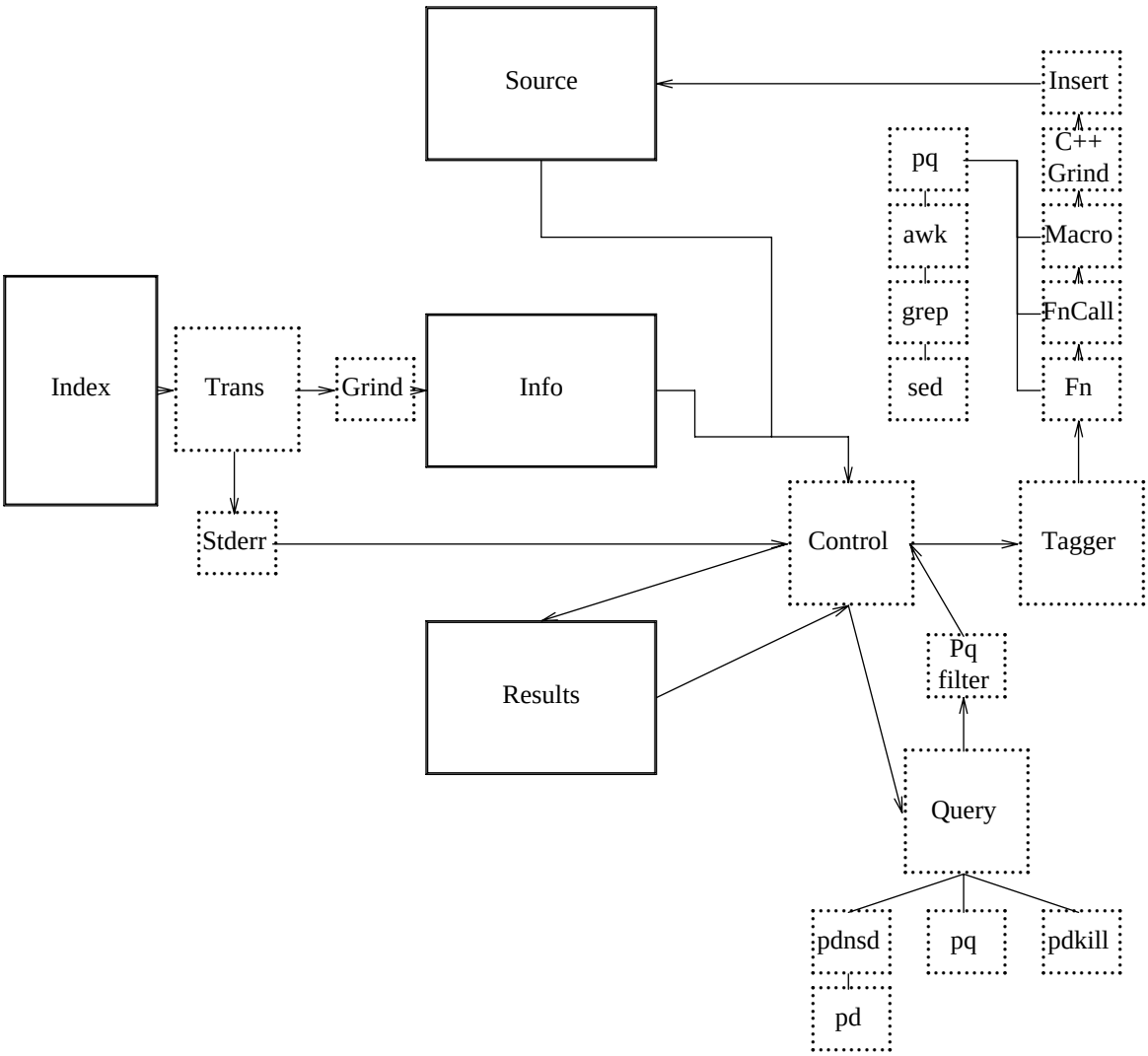

Figure 2. Process structure.

Activity in the process structures flows from left to right across the diagram. The user selects a class in Index; this results in the generation of tagged documentation (via Trans and Grind), and tagged source code (via Tagger, Function, Macro, C++ Grind, and Insert). The tagged files are sent to the Info and Source windows, respectively, where the user can page through them and use different presentation styles to view the tagged information. In either Info or Source, the user can select a function name and thereby submit a query to Control. This is passed on to Query, which consults the Prolog database for the location of the definition of all functions with that name. The results from the search are returned to Control, which displays them in Results.

The process structure is not a simple pipeline, and so it cannot be configured with existing shells such as *sh* or *csh*. I make use of *ash*, a simple custom shell that can set up an arbitrary network of connections. *ash* does not appear on the diagram, as it terminates once it has forked the process structure and established communications.

Several of the modules in the browser are shell scripts. The use of shell scripts may seem inefficient, especially since some of them are run each time a new class is processed (e.g., Function, Macro, Query). In practice, shell script interpretation is much less of a bottleneck than consulting the Prolog database to tag the source code file. Indeed, the use of shell scripts turned out to have some unanticipated benefits. One is that parts of the design can actually be upgraded dynamically; for example, tagging

filters can be changed at run time. The second unexpected benefit is that the system gains a measure of robustness; if a shell script cannot complete its task, it merely exits, without the rest of the application falling over. The next (correct) invocation will complete normally.

The three main activities in the browser are tagging the documentation, tagging the source code, and handling queries.

3.1. Tagging the documentation.

The NIH documentation is in TEXinfo format, a variant of TEX used in the GNU project to make online documentation accessible through the EMACS editor. TEXinfo format contains embedded menus and control information, as well as standard TEX macros.

TEXinfo format is not suitable for use with *lector* for two reasons. First, *lector* cannot parse TEXinfo markup codes because of their syntactic complexity. *lector* handles a very limited subset of regular expressions, while TEXinfo uses a context-free grammar. Second, TEXinfo menus and control information require semantics that are not supported by *lector*. Thus, it was necessary to write a small transduction program that removed the menu and control information, and converted the markup to tags more suitable for use with *lector*.

The transduction program is written in *lex*. It demarcates documentation elements, such as the title, the synopsis, the source code, class structure (base, derived, related), and some typographical elements, such as emphasis, points, and line breaks. Transduction is sufficiently fast that it can be run on demand; good performance does not require storing converted versions of the documentation.

The transduction program is presented in Appendix 3.

3.2. Tagging the source code.

Tagging the source code is much more difficult than tagging the documentation. The documentation markup is simply an adaptation of the existing tags to enable presentation by a different display tool. The code, however, contained no structural markup beyond that implicit in its grammar. In order to effectively display the code, markup must be generated to make this structure explicit. Also, while we can embed large quantities of structural information in documentation without seriously disrupting its use, embedding anything in code risks affecting its interpretation by other support software, such as compilers, source code control systems, and makefiles.

Source code tagging is the most complex subsystem in the browser. The tagger applies a series of programs to the source code, each of which determines some aspect of the program's structure, and then outputs a list of 3-tuples of the form *(column, row, tag)*. Each 3-tuple defines a specific tag and its desired location in the original source code file. These lists are collected in a markup file, which is sorted and inserted in the code in a single pass.

Two types of structure are extracted from the source code. Syntactic information (keywords, white space, comments, and braces) is extracted by a simple *lex*-based program. Semantic information is extracted by means of queries to the Prolog database derived from the `.brs` files produced by the xlC compiler. These programs are relatively simple shell scripts that submit a query to the Prolog database and massage the output in order to produce tag lists. Presently there are separate scripts for extracting macro definitions, function definitions, and function invocations. Additional structure can be added by writing similar shell scripts to extract the additional structure, and then ensuring that this output is appended to the markup file.

The markup file is sorted before being inserted into the source code, so that the insertion program can make a single pass over the text. If sorting is not done intelligently, the insertion program will place some end tags after start tags of the next element leading to errors in display. For example, if a `</while>` tag and a `<if>` tag have the same position, the insert program will incorrectly place the `<if>` tag before the `</while>` tag, following the lexicographical order of the tags (rather than their logical order, which says that end-of-while occurs before a start-of-if tag). In general this problem should be addressed by adding tag knowledge to Insert; currently, however, it is ''solved'' by simply choosing tag names whose lexicographical order is the same as their desired order.

Transduction of the source code is sufficiently slow (on the order of five to eight seconds) that it would benefit from storing converted versions of the source code.

The source code tags are shown in Appendix 2; the transduction program is shown in Appendix 4.

3.3. Handling queries.

The browser supports a very limited interactive querying facility. If the programmer selects a function name from either the documentation or the source code windows (by using *lector*'s *Search String* function), a query is submitted to the Prolog database that searches for the location of the function's definition. Since functions in C++ may have the same name if they have different arguments, the query may return multiple hits. These hits are displayed in the Results window, along with the position in the file that contains their definition.

Figure 3. Source code styles.

At one stage, the browser actually displayed the source code of the target in the Source window (this was before the Source window could itself be the source of a query). Currently, nothing is done with selections of the hits in the Results window, although they are returned to Control.

It would be desirable to support a more general kind of search with the browser.

4. The styles.

The documentation, code, and class index are all displayed by invocations of *lector*, which supports multiple viewing styles. Style sheets were developed for each of these texts.

4.1. Source code styles.

The original white space and indenting of the source code is not maintained by the source code styles. Instead, a uniform indenting policy is followed, based on the nesting level of the code. This may be seen as an advantage or a disadvantage. If the programmer took special care in indentation, some useful layout information may be lost; on the other hand, if the programmer was not particularly careful in laying out the code, then spurious and misleading layout information may be removed. In either case, maintaining the original layout information is of problematical value. First, the text has been heavily modified by the addition of markup, which disturbs the original layout. Second, the programmer's layout is based on the use of constant-width fonts, whereas the code styles employ proportional-width fonts. Third, the programmer's layout is based on a fixed screen width, whereas the code styles permit the user to adjust the display's size to taste.

C++ Grind—This is the main style for viewing C++ source code. It uses oblique Helvetica for comments, bold Helvetica for keywords, and roman Helvetica otherwise.

Standard—This style shows the code as it is most commonly seen by programmers; that is, in a constant-width font (in this case Courier), and with no special typesetting other than indenting and white space. This style is useful mainly in contrast with the other styles.

Function Definitions—This style shows a list of the functions defined in the code. It is useful as an index to the code, for those searching for a particular function.

Function Calls—This style shows function invocations. The source code is shown in Courier; after each line of code, a list of function calls is shown in oblique Helvetica. This includes all hidden function calls such as constructors and destructors, and so is a useful indication of the implicit complexity of individual lines of code.

Reduced—This style attempts to give an idea of the complexity of individual functions. Function headers are shown in bold Helvetica; control flow statements (e.g., `if`, `for`, `while`, `return`) are shown in roman Helvetica, and all other statements are shown in the tiny *Nil* font.

No Comments—This style shows the code as in *C Grind*, but removes both C++ and C-style comments. This is useful for code that was developed on large projects, which typically has large amounts of relatively unimportant header commentary.

Comments Only—This style shows only the comments. It can be used to illustrate the lack of internal documentation of a module.

Uninterpreted—This style shows the tagged source code file without interpreting the tags. Its main use is for debugging.

Views of the source code styles are shown in Figure 3. The specification file for the styles is in Appendix 7.

4.2. Documentation styles.

The documentation describes the purpose of each class and the operations of its functions. The documentation styles provide a number of ways of viewing this information.

Full Text—This style is the standard for reading the documentation. It presents the text and code in readable format, using bold Helvetica for titles, roman Helvetica for running text, roman Courier for source code, and oblique Helvetica for emphasis.

Titles—This style presents a table-of-contents view by showing only the titles and subtitles. It can serve as an index to a long documentation file.

Functions—This style shows only the parts of the documentation marked as function descriptions. All text is thus presented in Courier. This style is useful for querying the location of the function definition (in the source code).

Functions and Titles—This style combines the previous two styles, presenting an index to the documentation file and a view of the functions described in each section of the documentation.

Associated Classes—This style shows only the related classes, along with their descriptive headings.

Uninterpreted—This style displays all text and tags in Courier font. It is mostly used for debugging the output of the tagging program.

Views of the documentation styles are shown in Figure 4, and the specification file is shown in Appendix 5.

Figure 4. Documentation styles.

4.3. Index styles.

The class index is a simple text file created specifically for the browser. It contains the names of the classes, marked up to indicate their inheritance relationship. The index styles present several views of this file.

Full Hierarchy—This style shows the present NIH class hierarchy, indented to indicate the level of inheritance. The main classes are shown in bold Helvetica, others in roman face. This is the main style used when browsing.

Old Classes—This style shows NIH classes from previous versions of the library, merged with the present hierarchy. Oblique Helvetica is used to display old classes. This style gives some indication of how the NIH class structure evolved.

Top Level—This style shows only the main NIH classes. It is useful if the programmer only wishes to consult these classes and prefers to use screen real estate to display other information.

Templates—This style shows the template files for NIH. These template files are used by programmers who want to add new classes to the library, following the NIH style.

Views of the index styles are shown in Figure 5. The specification file for the styles is shown in Appendix 6.

5. Problems in browsing C++ information.

Three classes of problems were encountered in developing the browser. First, adding markup to source code is inherently the wrong way to describe its structure. Second, determining what structure should be presented is extremely difficult, due to C++ itself. Finally, there were some practical problems in extracting the desired information from the Prolog database. We now consider each of these problems in some detail.

5.1. Limitations of markup.

One of the biggest problems with our current approach to displaying text is that it relies on markup to indicate structure. Markup is an inadequate representation for dynamic, non-linear structure, as I have discussed elsewhere[15]. The browser design suffered from two problems due to embedded structure:

- it is static
- it creates phase error

The static nature of markup conflicts with the dynamic nature of source code browsing. While some aspects of source code are static (e.g., a *while* statement is always a control flow statement), others are not (e.g., the execution path of the system varies depending on the input). Many dynamic structures arise in browsing source code, because the queries can themselves be considered ‘‘inputs’’ that guide the execution of the system. For example, a programmer may want to display a skeleton of a class definition file (e.g., only the function headers), along with the full content of the constructor functions, that (for good measure) should be displayed in red. Clearly, the number of possible objects that could be displayed is very large, as is the number of different styles that could be defined. We cannot afford to tag all of these elements statically, nor do we wish to make arbitrary computations at run-time, since this involves an unwanted coupling between display and application logic.

Phase error is the problem that insertion of markup inevitably changes the source code, and thus creates problems for other software that accesses the code. Markup can get in the way of tools like compilers and preprocessors, since it is usually not syntactically neutral C++. The standard solution to this problem is to embed structural information in comments, since these are ignored by tools that parse the code. Unfortunately, this is not a general solution, since not all references to the code are relative to the parse tree. For example, the output from the Prolog database includes line number and column offsets to indicate the positions of various structures; adding markup-as-comments to the source code would require reparsing to keep this positional information in phase.

The current browser suffers from markup-induced problems because it is based on a text display tool that requires markup for structure specification. Care was taken, however, to separate the structure-determining phases of the browser from the markup-embedding phase. Thus, when a display tool is available that does not require markup, it will be relatively simple to convert the system to the use of external structures.

5.2. The complexity of C++.

The reliance on markup is an irritant, but not the fundamental problem. The fundamental problem is that we simply have no good theory about what should be displayed in order to facilitate a programmer’s understanding of C++ code. Some writers claim that developing automatic tools for reliably detecting suspicious C++ constructs is essentially impossible[9]. As a result, C++ programmers have come to rely on a variety of recommendations, rules, and habits that have developed out of hard experience[6]. What follows is list of C++’s peculiarities I have encountered while attempting to develop an understanding of C++ programs.

Figure 4. Index styles.

One important problem is the range of programming styles: C++ code can span the range between styles as diverse as C and Smalltalk. Programmers can write software in which the whole activity is captured by program logic and base data types, or they can develop a system in which virtually the entire structure is stored in class libraries, with only very simple function calls at the top of the code chain. Most programs involve a mixture of these two styles. This range of styles invalidates many theories of browsing; for example, one might make a class browser the foundation of the environment, but in C++ the programmer can choose to use no classes at all, rendering the browser useless.† Conversely, one might think that proper display of nested conditionals is useful; programmers who capture most conditional information in a refinement hierarchy will bypass this facility. A useful code browser must deal with the range of programming styles possible in C++.

A second problem is the variety and extent of transitive dependencies in C++. C++'s combination of strict type checking and its encouragement of (both the use and expression of) dependencies means that programmers often make apparently simple changes that have far-reaching effects. Consider for example the following two function declarations:

```
blah& blah::foo(blah &);
blah& blah::foo(const blah &);
```

The `foo` function operates on instances of the class `blah`, taking one argument of type `blah`, passed as a reference pointer and returning a reference pointer to an instance of that type. The class originally used the first declaration of `foo`, but the programmer has decided to change it to the second. By making the argument a `const`, the programmer wants to state that `foo` is not permitted to modify the content of the argument. The compiler takes this constraint to heart and proceeds to check not only that `foo` does not modify the argument, but that any and all functions that might be invoked by `foo` do not change the argument either. Moreover, the compiler insists that the programmer make `const` any arguments and return values of invoked functions where necessary to ensure that the argument is not altered. These changes may themselves affect more functions. In the worst possible case, the programmer may find that at the bottom of one of these transitive chains, a function is called that sometimes *does* change the value of the initial argument, even though it might not do so in the instances in which it is called from `foo`. Then it becomes necessary to separate the modifying behaviour into a new function.

From the point of view of program consistency, the compiler is behaving properly; it is merely enforcing a stated constraint in its most general form. From the point of view of the programmer, the compiler is making life difficult; a simple modification has surfaced a raft of sloppy specification, that he must now wearily correct.

Related to the problem of transitive dependencies is the problem of overly-concise syntax that often hides implicit code, operations, or semantics. One example is operator overloading. Consider the programmer browsing a class library for lattices, who comes across the following program fragment:

```
lattice x;
element y;

x += y;
```

The program adds the element y to the lattice x. Or does it? Actually, without seeing the definition of the function `operator+=`, the programmer has no way of knowing what it does—it may remove the element from the lattice, search for the element in the lattice, ignore the element and randomly rearrange the lattice, print the lattice upside down and surround it with a circle of y's—there is absolutely no way to tell. Of course, it is true of any function in any programming language that it may not behave as advertised. Operators are a special case, however, since programmers are conditioned by mathematical training to expect that operators are robust, perform automatic type conversions as necessary, are in many cases commutative, and so on. When operators are overloaded, these intuitions are unreliable.

A different type of syntactic concision is the vast amount of work that is done ''under the covers''. Consider for instance the question of function invocations. In previous work on typesetting C code, I had devised a style that suppressed all content except for procedure headers and invocations[10]. In effect, this was a single-level call graph, ordered by the appearance of the code in the file. This style provided a rough and ready approximation of the complexity of a module, insofar as that was related to the number of function invocations. This style is virtually useless for some C++ code, however, since practically anything in C++ can and does result in a function call:

- assignment statements—call to a copy constructor, and possibly implicit type conversions
- variable declarations—call to a constructor

†Unlike Smalltalk, where most work requires defining new objects.

- operators—call to an operator function
- closing braces—call to destructors for any variables that go out of scope

Moreover, each or any of these calls may result in other calls, particularly if the called function is virtual.

Work done under the covers is often performed by code that the programmer did not write, and so is not present for browsing in the program. Machine-generated code can take several forms. One form is compiler-generated functions. Depending on which functions are not specified in the class definition, the compiler may generate a default constructor, destructor, assignment operator, copy constructor, and address-of operator. Instead of flagging the fact that these operators are missing from the class declaration, the compiler silently generates minimal defaults. These defaults are just enough to allow the program to compile, but in many cases are not enough to ensure robust execution. Since these functions may be missing purely in error (often enough, because the programmer was simply unaware that they should have been defined), it is important to be able to inform the programmer that they have been created.

The Prolog database generated by the xlC compiler contains information on compiler-generated functions. It places them at the end of class declaration file.

A second type of hidden effect is implicit type conversion. In C++, it is possible to define a set of type conversions that are applied automatically by the compiler[8]. C programmers are familiar with the C compiler silently converting base types; for example, a `float` is silently converted to an `int` in programs like the following:

```
int     a;
float   b;

a = b;  /* float converted to int */
```

In C++, user-defined conversions permit the same functionality to be applied to arbitrary types. For example:

```
class prime
{
    int x;

    prime() ;
    ~prime() ;
    operator int() // type conversion
        { return x };
}

int     a;
prime   b;

b = a;  // call to operator int()
```

User-defined type conversions can be used explicitly by the programmer, or implicitly used by the compiler when an assignment requires type conversion. In the latter case, there is no indication in the source code of the conversion. Type conversion is further complicated when there is no direct conversion between two types; if the conversion can be accomplished by means of a combination of standard conversions and one user-defined conversion, then a silent conversion will be performed. If the conversion requires more than one user-defined conversion, then the compiler flags the situation as an error. Silent conversion (and the use of type conversion in general) leads to so many insidious situations that at least one C++ teacher says:

> *There is a biblical analogy I'd like to draw here. Casts are to C++ programming what the apple was to Eve*[9].

Another form of hidden code is templates. Templates are parameterizable skeletons of code. A template for a (minimal) set class, for example, might be defined as follows:

```
template <class Item>
set<Item>
{
    Item    p[];
    int     size;

    Item    pop();
    void    push(Item);
    Item    top();
    void    clear();
}
```

The template `set` can be used for sets of `int`, `float`, or any other type, simply by declaring an object of that kind:

```
set<float>   float_set;
set<int>     int_set;
```

A template looks like C++ source code, but is really a kind of preprocessor statement. The actual source code is produced by the compiler's template instantiator, which generates functions for each instantiation of `set`. For the above examples, the instantiator will automatically generate source code files for the functions:

```
set<float>::pop();
set<float>::push(float);
set<float>::top();
set<float>::clear();
set<int>::pop();
set<int>::push(int);
set<int>::top();
set<int>::clear();
```

If the generated source code is incorrect, the compiler produces error messages that direct the programmer to a machine-generated source file. The problem with this tactic (apart from the fact that the programmer did not write the source file, and so is justifiably mystified about how to correct it), is that incorrect source code can be generated if the programmer's classes do not contain all the functions the template expects. For example, the `set` template probably uses the operator `==` to determine whether two `Item`s are identical. Thus, to create a `set` of `regexp`s (a regular expression class) the programmer must be sure that `==` is defined, or else the compiler will complain.† Another common instance of such a dependency occurs with input/output; the template will probably rely on an output operator being defined for each possible type parameter (see [8] p. 365-366). The result is that libraries containing templates can be particularly troublesome for both library and applications programmers. The library programmer has no way to specify these dependencies in advance, or to recognize them before compile time. The application programmer will be confronted with complex errors during compilation, and will need to study the template code to resolve them. This situation does not promote modularity or reuse.

Most of the foregoing has been a discussion of the difficulties of comprehending the logic of C++ programs. Real programming situations also require a thorough understanding of the (in)efficiency of the program or library as well. I have made no effort to study this aspect, but clearly it involves knowledge about the use of pointers, reference operators, memory allocation, and the extent of copying[2]. Any browsing environment that does not provide this information will not address a substantial part of the rationale behind a C++ class structure.

5.3. Problems with the Prolog database.

The third class of difficulties in developing the prototype browser were connected with the xlC-derived Prolog information. Like markup, many of these problems are unique to this particular implementation; however, some (like poor efficiency) are indicative of more general problems with source code databases[3].

Some of the source code structure is derived from the `.brs` files generated by the xlC compiler[7]. This data is indirectly mediated by a Prolog database that stores the information about the source code as facts, permitting arbitrary queries to be posed to the database.

The original plan was to use the Prolog database as the sole source of information about the C++ source code. It was not deemed proper to develop a parser, preprocessor, and so on for C++, since this activity was already being performed (in a robust and thorough manner) by the compiler. If it is possible to consult the compiler about its knowledge of the source code, then little additional information should be needed.

The original plan was clearly a good approach to the problem. In implementation, however, I ran into three basic problems:

1. The Prolog database does not contain all the information that the compiler knows, or contains it in ways so indirect as to be essentially inaccessible.

The key problem here is that the Prolog database stores information about the starting position of various components, but not their ends. This is appropriate for a special purpose browser that can include information about C++ code (as, presumably, the xlC browser does), but is not appropriate in the general case. It might be thought that the end of a component can be determined from the start of the next component; for example, the start of a function definition probably signals the end of the previous definition. This strategy is invalid in the general case since components are mixed heterogeneously, and the start of a component of one type does not necessarily mean the end of the current one. Thus, knowledge of C++ becomes necessary to determine the ends of components (see point 3 below). Secondly, comments and any other components not captured by the Prolog database (see point 2 below) will be erroneously included within the scope of a ‘‘greedy’’ determination of this type. Thirdly,

†In the case of `regexp`, the *programmer* might well complain, since the only reliable procedure for testing equivalence of regular expressions is to convert the expressions to NFAs, convert the NFAs to DFAs, minimize the DFAs, and then check for identity—an exponential task in the worst case.

any such determination of component-ends must be aware that many components contained in the Prolog database are not present in the source file; this includes compiler-generated functions and (judging from the data for the NIH library) constructors for related classes (i.e., base and friend classes). The starting position for such components (given as the last character in the source file) is arbitrary, as would be any end position.

From the code for the `.brs` prettyprinter, one infers that at one time information about block ends was either contemplated or actually present in the `.brs` file, since the prettyprinter contains statements for both BRS_BBEGIN and BRS_BEND information. Prettyprinting NIH `.brs` files, however, reveals no BRS_BEND data, suggesting that xlC does not (currently) store this information in the `.brs` file.

2. The compiler does not know everything about the code.

Comments and white space are important aspects of code to programmers, and hence it is important that they be properly displayed. However, comments and white space are essentially irrelevant to compilers, and so do not appear in the Prolog database. I expect the content of the comments and the white space is thrown away at the scanning stage, and only their offsets are kept (and this only indirectly, as the starting positions of code statements that appear after the comments and white space).

3. C or C++ knowledge is required to interpret some of the information in the Prolog database.

The information in the Prolog database cannot necessarily be used in a ‘‘standalone’’ way. An example is function headers. Consider the function header

```
void Set::reSize(unsigned newSize)
```

The Prolog database stores a fact to represent the function header, giving a line and column number that indicates the start of the header. The column number given is 11; in other words, the database stores a pointer to the member name, without class name and return type. If one wishes to display the full function header, C++-specific knowledge is required to know the start (and end) of the header.

6. Discussion.

The current prototype is an interesting exercise in applying text display tools to code browsing, but is not yet a useful programming aid. It has raised many questions about the display and typography of C++ software, and has pointed to problems with the underlying support databases.

Further work is being undertaken in each of the three problem areas discussed. The limitations of markup are avoided in the next generation of text display software, which can employ either markup-based structure or externally-specified structure[11]. The complexity of C++ code is being investigated as part of work that is applying database dependency theory to the general update problem[12]. This work will investigate how to describe object-oriented class libraries in terms of functional and multivalued dependencies. Finally, the database problems are being explored in the context of the use of partial orders as a fundamental data model[13].

One important lesson that resulted from this experience is that the notion of displaying ‘‘the’’ source code for C++ is essentially misleading. The code typed by the programmer is only one version of the source. The code generated by the preprocessor (after including header files and class definitions) is a second version of the source. The code generated by the template instantiator (resulting in the definition of many more source functions) is a third version of the source. Finally, the code generated by the compiler (including compiler-generated constructors and destructors, implicit casts, and function inlining) is a fourth version. There are further ‘‘virtual’’ versions of the source that might be generated, corresponding to an actual execution of the program (i.e., a display of source code showing an actual call graph under resolution of polymorphism). The source for a C++ program is thus a complex, dynamic entity that is constructed on the fly. Any good browsing tool must be able to deal with this dynamic entity.

A second lesson is that code browsers based on simplistic theories are likely to be inadequate for large class libraries. Smalltalk, Lisp, Hermes, and other languages that provide only a few types of constructs appear to be captured by browsers that describe only those constructs, while C++ appears to be a knotty problem for browsing because of its many complicated ways of expressing relationships and dependencies. Fundamentally, however, the browser addresses the same need: to expose the abstract data model that is implicit in the design of a large software artifact, and to enable the programmer to make use of it in the best way possible. Curiously enough, it is the knotty syntax of C++ that makes it impossible for us to fool ourselves (as we can with Smalltalk) into thinking that a simple class browser is adequate. In neither case is

it sufficient, for programs of moderate to large size, and Smalltalk systems will benefit from better access to abstract representations of their implicit models, just as will C++ programs.

7. Acknowledgements.

Arthur Ryman commissioned the development of this browser, insisted on the separation of C++ knowledge from the remainder of the code, and motivated the use of the xlC-derived information. Financial support was provided by an IBM Canada Fellowship, by the Information Technology Research Centre, and by the Natural Sciences and Engineering Research Council of Canada.

8. References

1. James C. Armstrong, Jr., ‘‘Review—SMARTsystem CASE Tool,’’ *SunWorld*, pp. 75-78 (October 1992).
2. T.T. Carey and R. Spall, Douglas E. Harms, and Bruce W. Weide, ‘‘Copying and Swapping: Influences on the Design of Reusable Software Components,’’ *IEEE Transactions on Software Engineering*, **17**(5) pp. 424-435 (May 1991).
3. Stefano Ceri, Stefano Crespi-Reghizzi, Andrea Di Maio, and Luigi A. Lavazza, ‘‘Software Prototyping by Relational Techniques: Experiences with Program Construction Systems,’’ *IEEE Transactions on Software Engineering*, **14**(11) pp. 1597-1609 (1988).
4. Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico, *Data Abstraction and Object-Oriented Programming in C++*, John Wiley and Sons, West Sussex, England (March 1991).
5. T.R.G. Green, D.J. Gilmore, B.B. Blumenthal, S. Davies, and R. Winder, ‘‘Towards a Cognitive Browser for OOPS,’’ *International Journal of Human-Computer Interaction*, **4**(1) pp. 1-34 (1992).
6. Mats Henricson and Erik Nyquist, ‘‘Programming in C++: Rules and Recommendations,’’ M 90 0118 Uen, Ellemtel Telecommunication Systems Laboratories, Älvsjö, Sweden (1992).
7. Sharam Javey, Kin’ichi Mitsui, Hiroaki Nakamura, Tsyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, and Richard Helm, ‘‘Architecture of the XL C++ Browser,’’ *Ideas in Action: CASCON ’92*, pp. 369-380 IBM Centre for Advanced Studies, IBM Canada Laboratory, (November 9-12, 1992).
8. Stanley B. Lippman, *A C++ Primer, 2nd Edition*, Addison-Wesley, Reading, Massachusetts (1991).
9. Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, Reading, Massachusetts (1992).
10. Darrell R. Raymond, ‘‘Reading Source Code,’’ IBM Canada Laboratory Technical Report TR 74.070, IBM Canada, Toronto, Ontario (October 1991).
11. Darrell R. Raymond, ‘‘Evolutions in Typesetting Systems,’’ *Ideas in Action: CASCON ’92*, pp. 19-28 IBM Centre for Advanced Studies, IBM Canada Laboratory, (November 9-12, 1992).
12. Darrell R. Raymond and Frank Wm. Tompa, ‘‘Applying Dependency Theory to Software Engineering,’’ submitted to IEEE Transactions on Software Engineering (December 1992).
13. Darrell R. Raymond, ‘‘Partial Order Databases,’’ *Ph.D. thesis proposal*, Department of Computer Science, University of Waterloo, (April 1992).
14. Darrell R. Raymond, ‘‘Flexible Text Display with *Lector*,’’ *IEEE Computer*, **25**(8) pp. 49-60 (August 1992).
15. Darrell R. Raymond, Frank Wm. Tompa, and Derick Wood, ‘‘Markup Reconsidered,’’ *First International Workshop on Principles of Document Processing*, (October 21-23, 1992).
16. Arthur Ryman, ‘‘The Theory-Model Paradigm in Software Design,’’ TR 74.048, IBM Canada Laboratory, Toronto, Ontario (October 11, 1989).
17. Veli-Pekka Tahvanainen and Kari Smolander, ‘‘An Annotated CASE Bibliography,’’ *ACM SIGSOFT*, **15**(1) pp. 79-92 (January 1990).