The C++ Information Abstractor

Article · November 1997

Source: CiteSeer

1 author:

Yih-Farn Robin Chen
AT&T
104 PUBLICATIONS **3,950** CITATIONS
SEE PROFILE

The C++ Information Abstractor

Judith E. Grass
Yih-Farn Chen

AT&T Bell Laboratories
Murray Hill, NJ 07974

The *C++ Information Abstractor*, *cia++*, builds a database of information extracted from C++ programs. The database can serve as a foundation for the rapid development of C++ programming tools. Such tools include tools that graphically display various views of the program structure, tools that answer queries about program symbols and relationships, and tools that extract self-contained components from a large system. *Cia++* is a new abstractor implementation based on the design of the *C Information Abstractor*. This paper describes the conceptual model of *cia++* and examples of relational views and applications developed on the C++ program database. It also presents some aspects of the implementation.

1 Motivation

C++ is rapidly becoming a language of choice for developing large application programs due to its object-oriented paradigm. Many new C++ programmers come from the C programming community. The transition from C to C++ is made easier by the fact that these languages share a common core. It is made more difficult by the lack of tools to support C++ program development. C programmers enjoy the support of many program development and analysis tools. Unfortunately, few of these tools transfer gracefully to C++. The lack of tools designed for C++ discourages developers from adopting it. The critical need for C++ programming tools is only beginning to be seriously addressed.

This paper describes the *C++ Information Abstractor* (*cia++*) and the system of tools built around the database that it generates[1]. *Cia++* extracts information about the various entities[2] declared in a C++ program and the relationships between those entities. The abstracted information is stored in a database that can be used by any number of tools. Each tool provides a different view of the way complex C++ programs are structured.

The *C Information Abstractor* [1, 2] , usually referred to as *cia*, builds a program database for C. *Cia++* provides the functionality of *cia* for C++ programs. Because C++ is a semantically richer language than C, *cia++* must reflect new entity attributes and relationships between entities that do not exist in C programs. One example of such a relationship is inheritance, a relationship between two classes. This paper presents a specification of the conceptual model and database schema used to store the information abstracted from a C++ program. Some aspects of a *cia++* implementation are also explored here.

2 The Design Rationale

Cia++ is designed to be an extensible platform for rapidly developing program analysis tools. Several features of *cia++* facilitate this:

[1]By convention, *cia++* written in lower case letters refers just to the program that abstracts data from a C++ program. *CIA++* written in capital letters refers to the system made of that abstractor and all of the tools that use the database it generates.

[2]Entities include files, macros, types, functions and variables.

- *Separation of Information Extraction and Presentation:* The process of extracting information from a program and the process of presenting that information should be separate. This eliminates the need for each C++ analysis tool to duplicate the parsing process and allows C++ tools to share the information and present it in different ways. A similar doctrine was developed in the *Interlisp* [3] project. Unfortunately, many C and C++ tools today still violate this basic principle.

- *A Well-Conceived Conceptual Model:* A conceptual model based on the entity-relationship model [4] was designed to describe the entities, attributes, and relationships in C++ programs. The model serves as an accurate high level specification for the contents of the program database.

- *Relational Database:* To support reuse, *cia++* stores abstracted information in the form of a relational database. This makes it possible for that information to be accessed by a wide variety of existing database query systems.

- *Incremental Database Construction: Cia++* builds and maintains a program database for large systems efficiently because it allows incremental updates of a database. When a set of C++ source files is updated, only that portion has to be reabstracted to update the database.

These basic design principles give tools based on *cia++* an advantage over many traditional development tools, which usually merge the information extraction and presentation processes in a single tool and do not share the information with other tools.

The C program database generated by *cia* has had many new C tools developed on top of it. These tools were developed in a matter of days by using the common database, but different approaches. Some tools were written as *awk* or *ksh* scripts that call database query commands. Others are C programs that call a query library. Other applications have been written that use INGRES [5] or DataShare[3] queries to extract information from the C program database.

With minor changes to adjust to the different format of the C++ program database, these tools can be applied to C++ programs. Many new tools can be developed in a short period of time by sharing the well structured information in the database. This makes it possible to offer C++ programmers a higher level of tools support than has been possible up to now.

Our conceptual model for a C++ program database contains the information to answer a wide variety of queries, including

- what is the inheritance hierarchy of this class?
- what are all the public, inherited members of this class?
- where are all the references to this class member?
- where are all the declarations of this overloaded function?

It can be time consuming and difficult to discover the answers to these questions if no more support is available than huge collections of source files, *grep* and a text editor.

A program database can be used as a foundation to build several types of browsers as well. This has been done with the C program database. The C++ program database should be a good foundation for building browsers as well; however, we would like to stress that browsers are only one of many applications of the database.

[3]DataShare is an AT&T proprietary database manager written by Rick Greer.

3 The Conceptual Model

A well defined conceptual model is a key ingredient in the success of an information abstraction system. The conceptual model specifies which software entities, relationships and attributes should be stored in the program database. It also serves as a requirements specification for the C++ information abstractor. The current version of *cia++* tracks only *global* entities, members of *global* types and interactions between them. This succinct model limits the size of the resulting database, yet provides sufficient information for performing a large number of browsing and analysis tasks.

3.1 Entities

The *CIA++* conceptual model recognizes five kinds of software entities in a C++ program. These are *files, macros, types, variables* and *functions*. All of the entities present in C++ programs can be fit into this framework. For example, definitions of overloaded operators are considered to be function definitions and data members of a class are treated as variables.

Appendix A contains an abridged C++ program with examples of the following entities [6] :

file:	stream.h, tree.h, tree.c, main.c
type:	Tree, Node, IntNode, UnaryNode, BinaryNode, ...
variable:	Tree::p, UnaryNode::op, UnaryNode::opnd, BinaryNode::left, ...
function:	operator<<(ostream&, const Tree&), Node::print(ostream&), ...

Each one of the five kinds of *cia++* entities has a unique list of attributes, which specify various information about an entity declaration. *Cia++* records all of the declarations (including definitions) that are explicit in the source code. In addition, it records definitions of functions and variables generated by *cfront*. These implicit definitions usually carry unusual names and are easy to identify.

3.2 Relationships

Cia++ records reference relationships between global entities and members of classes, structures and unions. It also tracks references to static variables and functions declared global to a file and extracts information about the relationships between the recorded entities. Table 1 lists all of the reference relationships recorded by *cia++*. Given two global entities or members A and B, there is a relationship between them if:

- A is a file and B is another file included by A. This is an *include* relationship from A to B.
- Both B and A are types and B inherits A. This is an *inheritance* relationship from B to A.
- B is a type and A is a function or type, and B nominates A as a friend. This is a *friendship* relationship from B to A.
- Neither A nor B is a file, B is neither a friend nor a parent class of A, and some declaration of A refers to B. This is a *reference* relationship from A to B.

Each relationship has a *usage* attribute that stores information about where relationships were discovered in the source files. The information is mainly to be used by C++ browsers.

The *include* and *reference* relationships also exist in the C program database; however the inheritance and friendship relationships are unique to C++ programs and they have additional attributes.

The small sample program contains many reference relationships. Here are a few examples:

Entity Reference Relationships		
entity A	*entity B*	*definition*
function	function	function A refers to function B
function	variable	function refers to variable
function	type	function refers to type
function	macro	function refers to macro
variable	function	variable refers to function
variable	variable	variable A refers to variable B
variable	type	variable refers to type
variable	macro	variable refers to macro
type	variable	type refers to variable
type	type	type A refers to type B
type	macro	type refers to macro

Table 1: Table of Reference Relationships

```
entity 1          relationship     entity 2          comment
------------------------------------------------------------------------------
tree.h            includes         <stream.h>        file to file
Node              refers to        Tree              type to type
Tree              is a friend of   Node              type to type
IntNode           inherits         Node              type to type
BinaryNode::left  refers to        Tree              mem_variable to type
Tree::~Tree       refers to        Node::use         mem_function to mem_variable
Tree::Tree        refers to        IntNode::IntNode  mem_function to mem_function
main              refers to        Tree              function to type
IntNode::print    refers to        IntNode::n        mem_function to mem_variable
```

4 Relational Views

The program database generated by *cia++* contains a lot of information packed into a relatively small space. It uses an ASCII representation that is human readable, but the amount of information and the terseness of its representation make direct use untenable without relational database tools.

When *cia* was written, a specially tailored database retrieval system called *InfoView* was also developed to extract information from the C program database. Most InfoView commands can also be used with the database generated by *cia++*, but because InfoView was specifically designed to work with *cia*, it cannot present information about the additional fields and relationships tracked by *cia++*. As a result, it cannot easily answer the queries presented in Section 2.

There are great advantages to adopting a more general database manager for CIA++ that could be used with a broader family of information abstractors. DataShare appears to be well suited for this purpose[4]. DataShare provides a powerful query language for advanced data processing and a translation system that converts queries into efficient C programs. Several InfoView-like commands have been developed in a short period of time using DataShare.

The following sections show two major types of queries that can be run on the CIA++ database: entity queries and relationship queries. In each of these examples, the first line shows a computer prompt (**$**) and a database query command. The lines that follow are the results of that query.

[4] Several other database management systems have been used with *CIA* and should be useful with *CIA++* as well.

4.1 Entity Queries

Entity queries deal mainly with attributes of all entities. They take two arguments, entity kind and entity name, and return the attribute information. The first sample query displays all of the declarations of a function called `print`. This finds all functions with that name regardless of overloading or class membership.

```
$ Def function print
file            dtype                  ms spec name                   bline df
=============== ====================== == ==== ====================== ===== ==
tree.h          void ( )(ostream&)     pv vi   BinaryNode::print      53    df
tree.h          void ( )(ostream&)     pv vi   IntNode::print         32    df
tree.h          void ( )(ostream&)     pt vi   Node::print            10    df
tree.h          void ( )(ostream&)     pv vi   UnaryNode::print       40    df
```

The output shows that there are three private (`pv`) and one protected (`pt`) member function definitions (`df`) of `print`. These functions are all declared virtual (`v`) and inline (`i`) in `tree.h` and they all have the same signatures.

An option `-u` is provided to present the output information in a raw form. For example, the following query shows all the attributes of the member function `UnaryNode::print` in its unformatted form, which makes pipeline processing convenient.

```
$ Def -u function UnaryNode::print
578:print:p:tree.h:void ( )(ostream&)::40::44:df:pv:UnaryNode:vi
```

This example also shows that an entity name can be qualified with its parent class name [5]. Such a qualified name takes this form: *class_name*::*name*, where *name* is a member of *class_name*.

It was easy to implement a command, *Viewdef*, that shows the definitions and declarations of selected entities as they appear in the source code. *Viewdef* simply runs an *Awk* script on the output of *Def -u* commands. For example, here it retrieves the definition of UnaryNode::print:

```
$ Viewdef -f -n fu UnaryNode::print
tree.h:40     void print (ostream& o) {
tree.h:41               o << "(" ;
tree.h:42               o << op  ;
tree.h:43               o << opnd ;
tree.h:44               o << ")"; }
```

The option `-f` turns on the printing of filenames and `-n` turns on the printing of line numbers.

Selection clauses in the form of *attribute=value* can be attached at the end of the *Def* or *Viewdef* command line to restrict the query output further. *Egrep* style regular expressions also can be used to specify entity or attribute names. For example, this query displays all private member functions of the class `BinaryNode`:

```
$ Def function BinaryNode:: mscope=pv
file            dtype                  ms spec name                   bline df
=============== ====================== == ==== ====================== ===== ==
tree.h          void ( )(char *, Tree, pv i    BinaryNode::__ct       52    df
tree.h          void ( )(ostream&)     pv vi   BinaryNode::print      53    df
```

[5] It is also possible to further restrict the query to functions that take a specific argument list. This requires a bit more command syntax than it is appropriate to discuss here.

If the name component of a qualified name is missing, then it's equivalent to a ".*", so it matches any member in the specified class.

4.2 Relationship Queries

Relationship queries deal mainly with the reference database. Each reference query takes four arguments that specify the parent entity kind, parent entity name, child entity kind, and child entity name.

If a query is intended to find all entities that refer to a known entity, then the child entity name is usually specified as a wildcard, "-". For example, this query shows all of the types that refer to the class `Node`. The results of the query are displayed in the formatted style.

```
$ Ref type - type Node
k1 file1        name1          k2 file2        name2          rk
== ============ ============== == ============ ============== ==
t  tree.h       Tree           t  tree.h       Node           r
t  tree.h       Tree           t  tree.h       Node           f
t  tree.h       IntNode        t  tree.h       Node           i
t  tree.h       UnaryNode      t  tree.h       Node           i
t  tree.h       BinaryNode     t  tree.h       Node           i
```

This shows that the classes `IntNode, UnaryNode` and `BinaryNode` are all derived from `Node`. The class `Tree` is a friend and has another simple reference relationship to `Node`. The *Ref* command also takes selection clauses. The command "`Ref type - type Node rkind=i`" would return only the last three rows of information.

On the other hand, if a query is intended for finding all entities that are referred to by a known entity, then the parent entity name is usually specified as a wildcard. For example, this query returns all the variables referred to by `main`. There happens to be only one.

```
$ Ref function main variable -
k1 file1        name1             k2 file2        name2             rk
== ============ ================= == ============ ================= ==
p  main.c       main              v  CC/iostream.h cout             r
```

The output of *Ref* can also be displayed in an unformatted style.

5 Applications

The strength of the *CIA++* system lies in the tools that can be built on top of the database. This section describes some of the experimental tools we are currently using with the *CIA++* database.

Queries presented in section 4 are most often used to answer specific questions about a small number of program components. The database can also be used to investigate relationships between large numbers of components. Frequently C++ programmers want to have a global picture about how the pieces of the program are interconnected. This kind of information can be presented in a number of ways, but often the most easily understood representation is graphical.

Several tools have been developed to extract structural information from the C++ program database and display the structure as a graph. The following sample graphs were generated from the database for the program in Appendix A.

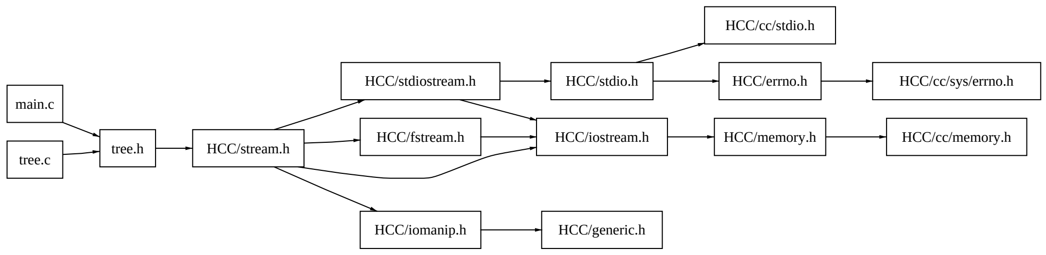

Figure 1: A File Include Graph

5.1 A File Include Graph

Often it can be difficult to know exactly what files get pulled into a C++ module when it is compiled. Part of that is because files that are explicitly included in the source file may include other files, which in turn may bring in more files, and so on. This is also affected by the environment. File names that are not completely specified may be resolved in several ways. A graph that shows the inclusion relationship can answer many such questions. Figure 1 shows the file include graph for `tree.c` and `main.c`. It was generated by the following command:

```
$ Dagen file file | sed -e 's,/usr/local/include/CC,HCC,' | dag -Tps | lpr
```

Dagen takes two arguments that specify the relationships to be drawn and outputs a graph description that can be processed by *dag* [7] . The *sed* command is used to abbreviate long pathnames. This example also shows that users can easily insert their own filters to alter the output.

Dagen accepts all kinds of relationships in the C++ program datbase. For example, "`Dagen function function`" gives a function call graph. The following three sections show additional examples related to the type to type relationships.

5.2 A Type Inheritance Graph

The semantics of C++ introduces many concepts related to classes. The inheritance relationship is central to object-oriented programming. Often it is useful to represent class hierarchies defined by inheritance as graphs. Figure 2 shows the type inheritance hierarchies for all the classes used in tree.c. It was generated by the following command:

```
$ Dagen type type rkind=i | dag -Tps | lpr
```

Similar to *Def* and *Ref*, optional selection clauses can be used to limit the output of Dagen to a subset.

`Tree.c` defines only one class hierarchy directly. That is the very simple class hierarchy rooted at the `Node` class. The other two hierarchies shown here are from the *iostream* library [8] pulled in by including *stream.h*. The hierarchy rooted at the class `ios` is interesting. Single inheritance class hierarchies always should appear as trees. *Ios* is the root of a hierarchy that uses multiple inheritance. This cannot be represented with a simple tree.

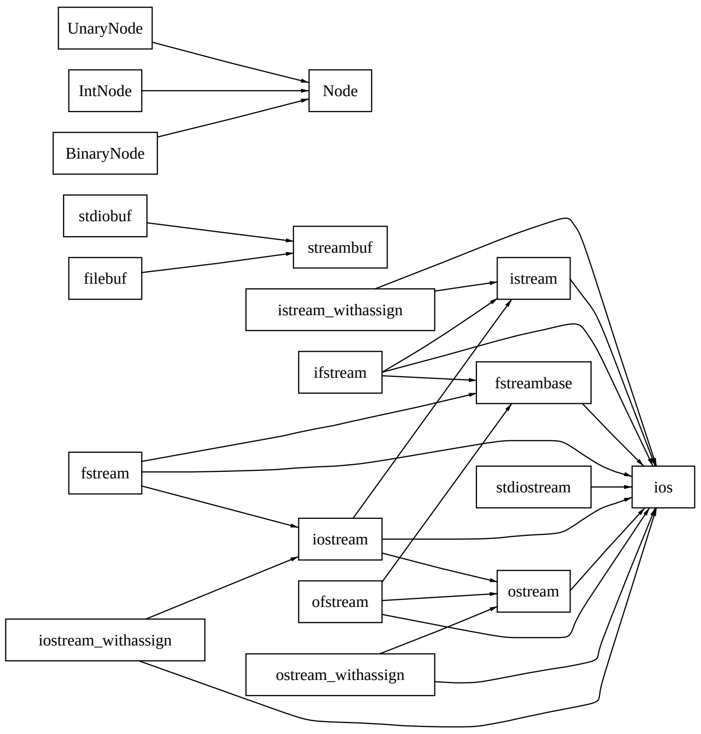

Figure 2: A Type Inheritance Graph

5.3 A Friendship Graph

Friendship is another relationship between classes and other entities that may be graphed. Figure 3 shows the friend relationships found in *tree.c*. There is a cycle in the graph because `Node` is declared as a friend of `Tree` and `Tree` is declared as a friend of `Node`. The figure was generated by the following command:

```
$ Dagen type type rkind=f | dag -Tps | lpr
```

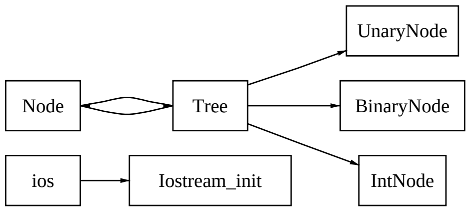

Figure 3: Friend Relationships

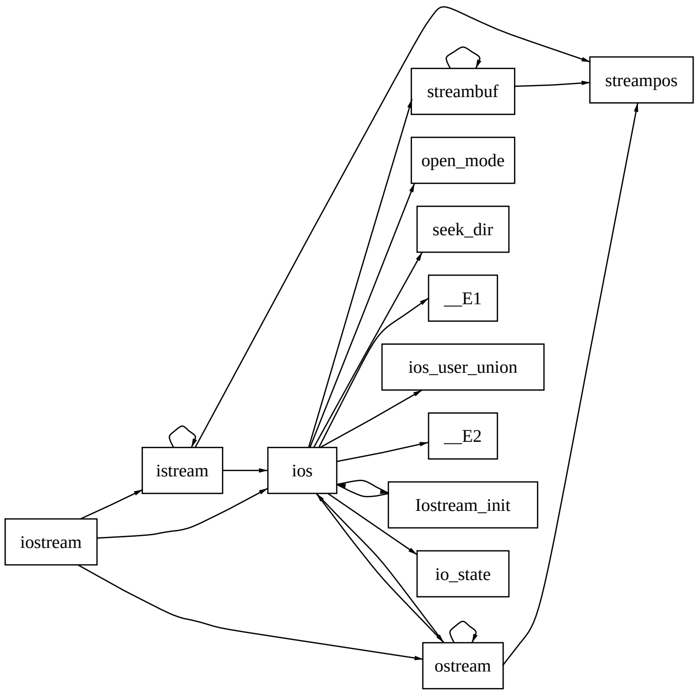

Figure 4: The Subsystem of iostream

5.4 Subsystem Extraction

Graphical tools can also be used to isolate self-contained components of a program. Type to type relationships define one kind of interesting subsystem. Figure 4 shows the subsystem of `iostream`; i.e., all the data types referred to directly or indirectly by `iostream`. This includes types that `iostream` uses for its members, the types that are friends and the types it inherits. The types `__E1` and `__E2` are names that *cfront* generated for the two anonymous enumerator types defined inside the class `ios`. The graph does not include relations to such basic types as `int, char` or `long`. The figure was generated by the following command:

```
$ Subsys -e type iostream type | Dagen -i type type | dag -Tps | lpr
```

6 Implementation Notes

The design and implementation of *cia++* involve developing a database schema and writing code to populate a conforming database with information extracted from a C++ program. In designing a database schema and format, we tried to deviate as little as possible from the schema developed earlier for C. The commonality of structure makes it possible to directly use some *cia* based tools and to rapidly adapt others. It also makes it possible for C programmers who have used *cia* to more easily adapt to C++ and *cia++*. More importanly, it allows us to merge C and C++ program databases with little effort, which is critical for projects that involve both C and C++ programs.

Cia++ was implemented using *cfront* version 2.0 as a starting point. This approach has many advantages and avoids many potential pitfalls of writing an abstractor from scratch. Some of the advantages derive from the fact that C++ is a young language and undergoing change. This was especially true while the abstractor was being built. At the time, C++ version 2.0 was not officially released. Developing *cia++* by embedding its code in *cfront* allowed it to keep in synchrony with the changes being made. *Cia++* requires detailed semantic information to extract relevant relations and entity attributes. Using *cfront* as a base ensures that *cia++* has an accurate understanding of the semantics that agrees exactly with that of *cfront*.

Using *cfront* as a platform has many implementation advantages as well. No parsing of any kind need be written, as *cia++* uses *cfront*'s code for this. Nor does *cia++* have to deal with any of the intricacies of name resolution or scoping. All of the code for this already exists in *cfront*. The way the abstractor works is conceptually simple. *Cfront* processes a program one external definition at a time. The definition is parsed to build a syntax tree. The syntax tree then goes through a declaration process. The declaration process resolves names to a symbol table reference that contains complete information about that program entity's attributes. After the declaration process, some program transformations may occur before the tree is walked to emit C code. *Cia++* allows *cfront* to do its declaration work unimpeded. This gives a tree full of interesting information. *Cia++* walks this tree looking for new entity declarations and for interesting relationships. Anything it finds, it adds to its database. Because *cia++* has a parasitical relationship to *cfront*, its view of a program's semantics is *cfront*'s view of a program's semantics. Because it sees what *cfront* sees, it can report on the code that the user wrote and the transformations that *cfront* made and know the difference between them. As an interesting side effect, *cia++* can produce program databases for programs that are incomplete or that contain syntax errors, and emit error messages as well.

This description makes the process of reusing *cfront* code seem much easier than it really is. The source code for *cfront* is very complex and not at all easy to understand or modify. Sometimes finding the exact spot in the code where all the information *cia++* needs is present and where nothing needed has been destroyed is literally impossible. So far, experimentation has usually shown a way to get around such dilemmas. In general, we have avoided making deep changes in the code of *cfront*.

7 Conclusion and Status

The current version of *cia++* has been successfully run on complex C++ programs, including the *iostream* package, *cfront*, and *cia++* itself. Our current vision of *cia++* should fill the need for a wide range of static program analysis tools. Eventually this will develop into a base for browser-like tools. Some components of *cia++* could probably be incorporated into a source language debugger. Our immediate goals are more limited.

We would like to thank Rick Greer for his help in showing us how to write efficient DataShare queries. Peter Selfridge and Peter Kirslis reviewed a draft of this paper and provided valuable comments.

References

[1] Y. F. Chen. The C Program Database and Its Applications. In *Usenix Summer 1989 Conference Proceedings*, Baltimore, MD, June 1989.

[2] Y. F. Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, March 1990.

[3] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. *IEEE Computer*, 14(4):25–34, April 1981.

[4] P. S. Chen. *The Entity-Relationship Approach to Software Engineering*. Elsevier Science, 1983.

[5] M. Stonebraker, E. Wong, R. Kreps, and G. Held. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[6] Andrew Koenig. An Example of Dynamic Binding in C++. *Journal of Object-Oriented Programming*, 1(3), August 1988.

[7] E. R. Gansner, S. C. North, and K. P. Vo. DAG – A Program that Draws Directed Graphs. *Software – Practice and Experience*, 18(11), November 1988.

[8] AT&T. UNIX System V AT&T C++ Language System: Release 2.0, Library Manual, 1989. Select code 307–145.

Appendices

A A Sample C++ Program

Reprinted with permission from the *Journal of Object Oriented Programming.* This appeared in Volume 1, Number 3, August/September 1988.

```
/*<tree.h>*/

#include <stream.h>

class Node {
    friend class Tree;
    friend ostream& operator<< (ostream&, const Tree&);
private:
    int use;
protected:
    Node() { use = 1; }
    virtual void print (ostream&) { }
    virtual ~Node() { }
};

class Tree {
public:
Tree(int);
Tree(char*,Tree);
Tree(char*,Tree,Tree);
Tree(const Tree& t){ p = t.p; ++p->use; }
~Tree() { if (--p->use == 0) delete p; }
void operator=(const Tree& t);
private:
    friend class Node;
    friend ostream& operator<< (ostream&, const Tree&);
    Node* p;
};

class IntNode:  public Node {
    friend class Tree;
    int n;
    IntNode (int k):  n (k) { }
    void print (ostream& o) { o << n; }
};

class UnaryNode:  public Node {
    friend class Tree;
    char *op;
    Tree opnd;
    UnaryNode (char* a, Tree b):  op (a), opnd (b) { }
    void print (ostream& o) {
            o << "(" ;
            o << op ;
            o << opnd;
            o << ")"; }
};

class BinaryNode:  public Node {
    friend class Tree;
    char* op;
    Tree left;
    Tree right;
    BinaryNode (char* a, Tree b, Tree c):  op (a), left(b), right(c) { }
    void print (ostream& o) { o << "(" << left << op << right << ")"; }
};
```

```
/*<tree.c>*/
```

```
#include "tree.h"
Tree::Tree(int n) { p = new IntNode (n); }
Tree::Tree(char* op, Tree t) { p = new UnaryNode (op, t); }
Tree::Tree(char* op, Tree left, Tree right) { ...  }
void Tree::operator=(const Tree& t){ ...  }
ostream& operator<< (ostream& o, const Tree& t) { ...  };
```

```
/*<main.c>*/
```

```
#include "tree.h"
main()
{
    Tree t = Tree ("*", Tree("-", 5), Tree("+", 3, 4));
    cout << t << "\n";
    t = Tree ("*", t, t);
    cout << t << "\n";
}
```

Contents