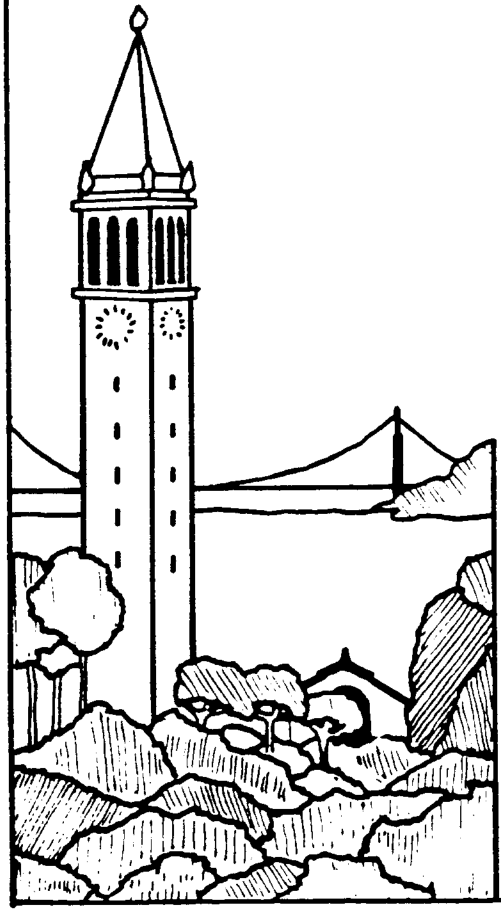
The C Information Abstractor

Yih-Farn Chen and C. V. Ramamoorthy



Report No. UCB/CSD 86/300

June 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The C Information Abstractor

Yih-Farn Chen
C. V. Ramamoorthy

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

ABSTRACT

Program understanding is one of the most time-consuming processes in software maintenance. This is partially due to the human inability to memorize complex interrelations among the software entities of a large software system. The situation worsens when the programs are not written by the software maintainers and little documentation is available. The basic idea in *information abstraction* is to extract relational information among the software entities of programs, store the information in a database, and make it available to users in a form that can be easily understood. We have implemented an Information Abstractor to extract relational information from C programs and store the information into a program database. High level access utilities are provided so that program maintainers or developers can easily retrieve the information they need for understanding the software. Besides program understanding, we found that the availability of the program database can also promote the research in four software engineering areas: multiple software views, software reusability, software metrics, and software restructuring.

1.0 Introduction

The cost of software maintenance typically represents more than sixty percent of the total cost of a software life cycle Software Engineering Perspectives 1984 The primary sources of maintenance problems include

(1) lack of good documentation
(2) inconsistency between the documents and the code
(3) ill-structured code caused by deadline pressures or dirty fixes
(4) the fast turnover of software personnel

Therefore, in many cases, software maintainers are faced with a *blackbox* system, one with internal structures unknown, which they have to understand, modify, and test.

The only reliable document that software maintainers can depend on is usually the program source itself. However, a typical software system consists of tens to hundreds of software modules and complex interrelations build up in the software if software objects reference one another across the boundaries of functions (procedures) or software modules (see Figure 1).

This work was supported in part by Ballistic Missile Defense under contract No. BMDSC DAS660-85-C-0029 (1-482427-25545).

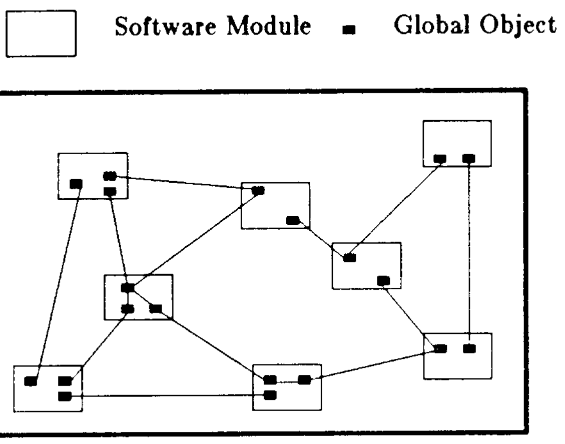


Figure 1. Complex Interrelations among Software Modules

Take a typical function in C programs [2] for example†:

```
DrawCurve()
{
    POINT *p1;
    ELT *e1;
    POINT *plist;
    char *text;

    if (SEQ < 2) return;
    plist = PTInit();
    p1=POINTLIST;
    do {
     PTMakePoint(p1, &plist);
     p1=PTNextPoint(p1);
    } while (!Nullpoint(p1);
    if (GRSetCurve(plist) !=0 ) return;
    UNForget();
    e1 = DBCreateElt(CURVE, plist, CBRUSH, 0, text, &PICTURE);
    DISCLearSetDisplay():
    ClearPoints();
    CHANGED=TRUE;
}
```

This function consists of the following references to objects defined outside the function boundary:

(1) Data types: **POINT, ELT**. (count = 2)

(2) Global Variables: **SEQ, POINTLIST, PICTURE, CHANGED**. (count = 4)

(3) Constants (macros): **CURVE, BRUSH, TRUE**. (count = 3)

(4) functions: **PTInit, PTMake, PTNextPoint, GRSetCurve, UNForget, DBCreatElt, DISClearSetDisplay, ClearPoints, malloc**. (count = 9)

Although this function is only twenty-one lines long, it consists of 23 external references. To fully understand this function and be able to modify it, a software maintainer has to resolve all these external references by locating the definitions of the objects referenced‡ and locating all the

† This example is actually a simplified version of a function from a graphics editor called Gremlin. The original function has more function calls than what we show here and no comments are associated with any statements.
‡ The Gremlin program has 21 source files and near 12,000 lines of C code.

objects that may be affected by the modifications. Moreover, each referenced object may have its own references to other objects as shown in Figure 1. Therefore, understanding and modifying software is difficult without easy access to the following information:

(1) The location and contents of a software object.
(2) The relations between a software object and other objects.

Our goal was to develop an information abstraction system that can provide the above information to software maintainers. This information has to be stored in such a form that it can be easily retrieved and processed for obtaining various views of the program. Many ideas we propose in this paper can be applied to most programming languages; however, we decided to concentrate on the implementation of an Information Abstractor for C programs. C has become very popular recently due to the popularity of Unix† programming environments. Most system programs and application programs on Unix system are written in C, and it is a common practice for programmers to modify these programs to adapt to their particular needs. Therefore, tools that help programmers understand C programs are in great demand.

Following our goal, we have implemented an information abstraction system for C programs. This system accepts C programs, extracts information about object locations and object relations, and stores the information in a program database. High level commands are provided to users for easy access to the program information. We discovered that the program database not only aids in program understanding; it also promotes the following research tasks:

(1) *Multiple Software Views:* The locational and relational information stored in the database simplifies the tasks of graphics packages that display multiple views of the programs using multiple-window systems.

(2) *Software Reusability:* The database helps a programmer retrieve a function and all its dependent objects and to reuse that function in other programs.

(3) *Software Metrics:* The database can be used to calculate several software metrics for analyzing the program structure.

(4) *Software Restructuring:* The software metrics provide some guidelines on software restructuring, and the retrievability of software objects make the restructuring tasks (like moving and clustering of functions) easier.

This paper examines the design considerations and various components of our information abstraction system and discusses it's future extensions and applications. The rest of this paper is organized as follows: Section 2 discusses some related work on program understanding tools. Section 3 discusses the important design decisions made during the implementation of the C Information Abstractor (CIA). Section 4 describes the performance and experience of CIA in using it as a program understanding tool. Section 5 outlines our plan for future extensions to CIA. Section 6 describes the potential applications of CIA. Section 7 is the conclusion.

2.0 Related Work on Program Abstraction

Before we proceed to describe our abstraction system, we first examine several systems and tools that have been built for abstracting information from programs. The major ideas and problems of these systems provide a guideline for the implementation of our system.

2.1 OMEGA System

The idea of storing program information into a database was proposed by Mark Linton and implemented in his experimental system OMEGA[3]. The OMEGA implementation includes (1) the design of a relational schema for a Pascal-like language called Model, (2) a program that takes software text and translates it into the database representation, and (3) a simple interface for viewing program information. One of the major goals of the OMEGA system is the ability to

† Unix is a trademark of Bell Laboratories

reconstruct the program from the program database. Therefore, detailed information about the variables, expressions, statements, and relations among them have to be stored in the database. This information takes significant time to process and requires a large database to be manipulated. According to [3], the prototype implementation of OMEGA system has poor response time in retrieving the body of a procedure. This is because the different objects within the procedures have to be retrieved, and each retrieval requires a separate database query. Another limitation of the OMEGA system is that it totally ignores comments, which usually contain information critical to the understanding of the software. The reason for ignoring comments is obvious; most programming languages (Smalltalk-80[4] is one exception) do not provide any means for a programmer to specify the linking between a comment and its associated object.

2.2 Program Slicing

Program Slicing[5] is another proposed method for abstracting from programs. Given a subset of program's behavior, slicing reduces that program to a minimal form, which still produces that behavior. For example, given the following Pascal-like program[6]:

```
1    begin
2    read(X,Y)
3    diff=X-Y
4    sum=X+Y
5    if (Y=1)
6    then Z=sum
7    else Z=diff
8    write(X,Z)
9    end
```

and the slicing criteria "value of **sum** at statement 8," the slice obtained is:

```
read(X,Y)
sum=X+Y
```

This is because only these two statements can affect the value of **sum** before statement 8. By selecting different slicing criteria, a program can be represented in many forms.

Program slicing is good for program debugging; however, it is not effective in obtaining information regarding the interrelations among software objects in different modules. Each computation of program slicing only provides partial program information. When one tries to understand a program in the beginning, one will have many questions regarding the structure and the interconnection of components in the program. It takes many slicing computations to get the information one needs. On the contrary, using the database approach as in OMEGA or our C Information Abstractor, all information can be retrieved in a single scan of the program and stored in the database. All later retrievals of information can go to the database directly without redundant processing on the program.

2.3 Cflow

The *Cflow(1)* command available on Unix System V [7] analyzes a collection of C, YACC, LEX, assembler, and object files and outputs an indented text showing the external references used in the program. Following is a typical output generated by *Cflow*:

```
1    main: int(), <f1.c 4>
2         f: int(), <f1.c 11>
3              h: char *(), <f2.c 12>
4              i: int, <f1.c 1>
5         g: int(), <f3.c 8>
```

The output shows the structure of the calling tree starting from the function **main**. For each external reference, it shows its data type and the location (filename and line number) where the referenced object is defined.

The major problem of the Cflow command is that the output cannot be easily processed or retrieved by database systems or similar utilities. Moreover, only the beginning line of each object is specified; this makes it difficult to retrieve an object since the ending line number is not provided. Finally, information about macro definitions, data types, and the files included are not extracted; this information is also critical for understanding the program structure.

3.0 Our Approach

From the survey of related work, we feel that several features are desirable for a program abstraction system:

(1) *Global Information Abstraction*: Emphasis should be placed on global references rather than local references to objects inside a function or procedure†. This helps reduce the size of the program database and increase the abstraction speed. A programmer can usually locate the local information he needs easily, but might have difficulty in obtaining information about global interrelations or the location of an externally referenced object. This information can only be obtained by scanning through several program files or software modules, which is a time-consuming process without adequate tools.

(2) *Database Support*: The information abstracted from programs should be stored in a form that can be processed by database utilities. Different views of the programs or program metrics can then be obtained by processing information in the database.

(3) *Simple Database Queries*: A simple query language using a simple syntax should be provided. Normal users should not be forced to memorize the details of the database schema of the program database.

(4) *Efficiency*: Fast retrieval of software objects is critical for the program understanding process. This can be achieved by storing only the pointers (line numbers in the source program files) in the database, then access objects in the source files using the pointers.

None of the existing systems we know provide all these features; therefore, we decided to build our own information abstraction system.

Ideally, the program information can be abstracted from two sources: the design documents and the source program. However, as mentioned previously, in many cases, the source program is the only reliable document a software maintainer can get, since many software documents are incomplete or inconsistent with the code. Therefore, our abstraction system only deals with programs. In the following subsections, we describe the structure of the C Information Abstractor (CIA), our design philosophy, and various components of the CIA.

3.1 Overview of the C Information Abstractor

The structure of the C Information Abstractor is shown in Figure 2. The basic approach is the following. We first define an object-attribute-relationship model (also known as the conceptual model) for the C program database. The degree of elaboration of the conceptual model is determined by the amount of information to be extracted from the C programs. A set of abstraction rules is then derived from this model. The C parser then extracts the program information according to the abstraction rules and stores the information in a set of data files, which can then be accessed using the *Information Viewer* described in section 3.3, or can be loaded and accessed using INGRES database utilities[8].

† However, abstracting local information may be necessary for calculating certain software metrics; e.g., complexity of the internal control structure of a function.

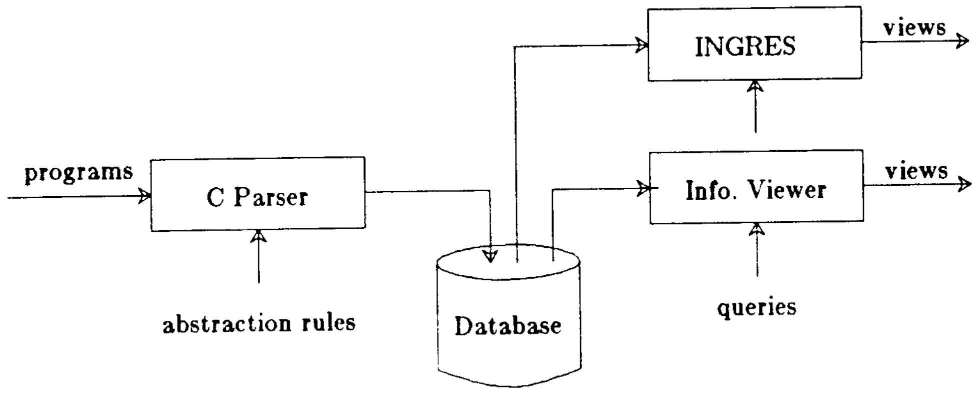


Figure 2. Outline of the C Information Abstractor

3.2 C Parser

The main design philosophy of CIA is to recognize the importance of global software objects, which can be referenced across file boundaries or function boundaries. Information about local details are basically ignored. Following this philosophy, we designed a conceptual view (Figure 3) for the C program database. This conceptual view is obtained by integrating several useful global views of the C programs.

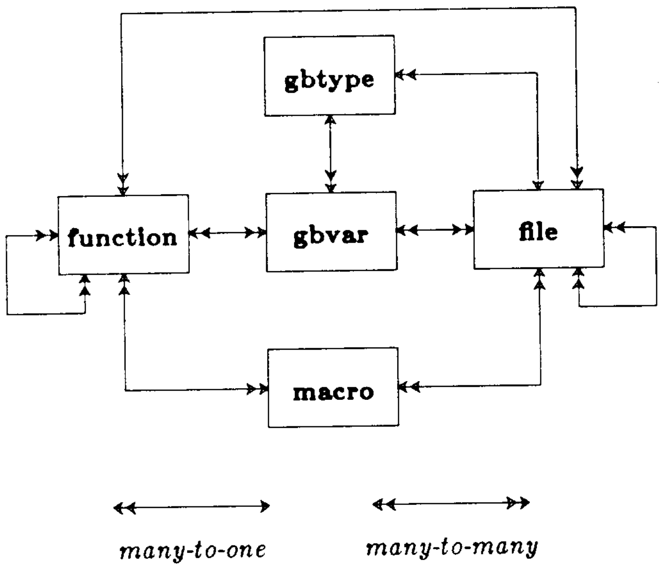


Figure 3. The Conceptual View of the C Program Database

Each box in Figure 3 represents an object. Five object types are defined: *file, function, gbvar, gbtype,* and *macro*†. Objects of these five types can be accessed across file or function boundaries in C programs. We define objects of these types as *global objects*. A special object type *comment*, which is not shown in Figure 3, is also available for processing *structured comments*, which will be described in Section 5.1. The comment object type has a one-to-one relationship associated with all the other object types.

Each object has its own attributes. Typical attributes are

† Macros are those identifiers defined in the *#define* statements in C programs.

(1) the file where the global object is defined

(2) the beginning line number and ending line number of the object in that file

(3) the data type of that object

Attribute information is helpful, e.g., with several tens or hundreds of files, it is difficult for a programmer to find out where a mysterious global variable is defined and what it is about. Using the attribute information described above, one can locate or retrieve the global variable in a much easier manner.

Each arc between two boxes in Figure 3 represents a relationship. Definitions of the nine relationships are shown in Table 1. The relationships listed are certainly not all the relationships one can find among all the object types, but they provide most of the relational information a programmer might need from a program database.

Table 1. Definitions of the Nine Relationships

num	*obj_type1*	*obj_type2*	*rel_type*	*definition*
1	file	file	m-to-m	file1 includes file2
2	function	function	m-to-m	function1 calls function2
3	gbvar	function	m-to-m	gbvar1 referenced in function2
4	macro	function	m-to-m	macro1 referenced in function2
5	function	file	m-to-m	function1 referenced or defined in file2
6	macro	file	m-to-m	macro1 referenced or defined in file2
7	gbvar	file	m-to-m	gbvar1 referenced or defined in file2
8	gbtype	file	m-to-1	gbtype1 defined in file2
9	gbvar	gbtype	m-to-1	gbvar1 defined as a gbtype2 variable

To see how the conceptual model can be used, we provide here an example C program that manages a queue:

```
file: queue.c

#include <stdio.h>
#include <queue.h>

QUEUE *QueueInit(name, length)
char *name;                     /* queue name */
int length;             /* queue length */
{
    short i;                            /* counts buffers */
    QUEUE *q;                           /* buffer to be added */

    q = (QUEUE *) allocate_memory (1, sizeof(QUEUE));
    q->mutex = seminit(1, bq->mname);  /* set mutex sem to unlocked */
    q->head = bq->tail = NIL;          /* set pointers to NIL */
    q->name = string;                  /* copy name pointer */
    QueueCount ++;
    return (q);                        /* return pointer */
}
```

To understand the above program, a user need to obtain the following views:

(1) Which functions call the function `QueueInit`?

(2) Where is the structure `QUEUE` defined?

(3) Which functions use the global variable `QueueCount`?

(4) Which files include the file `queue.h`?

(5) What is the global constant NIL?

The answers to all these questions can be found through the program database because the conceptual model incorporates all these views.

One approach to building the C parser is to modify the C compiler so that it generates additional information besides the original compiling tasks. We abandoned this approach for two reasons:

(1) The C compiler is too complex to make modifications for our purpose, and it is very difficult to maintain.

(2) In most cases, we are trying to understand a program which has been compiled correctly, so it is not necessary to process many program details that the compiler must go through.

So we decided to write our own parser. Our parser scans the C programs according to the conceptual model as shown in Figure 3, and is simpler than the one used in the C compiler. The set of rules were implemented using the standard tools Lex[9] and Yacc[10], which are available on most Unix systems. The parser generates ten data files: *file.data, function.data, gbvar.data, gbtype.data, macro.data, comment.data, filefile.data, funcfunc.data, macrfunc.data, and gbvrfunc.data*. All the attribute information in the five object files and the information of the ten relationships are encapsulated in these ten data files†. These data files can be accessed by the Information Viewer, which is described in the next section.

3.3 Information Viewer

Information Viewer (InfoView) is a set of programs that provides high level access to the data files created by the C Parser (see Figure 2). Since InfoView is designed for accessing program databases, the design considerations are different from normal relational database systems (e.g INGRES) in several ways:

(1) *Concurrency Control:* The InfoView commands only perform read operations on the program data files. Updates to the database are only performed by the C parser when a new version of the source program is created. At that time, the whole database files will be rewritten. Therefore, the Information Viewer does not provide any concurrency control. However, in the future, our Information Abstractor may be used in a multi-user programming environment where updates and read accesses to the database can perform concurrently. At that time, concurrency control may be necessary.

(2) *Object Retrieval:* One of the goals of our system is the easy retrieval of software objects, whose length can range from one to several hundred lines. Unfortunately, in traditional relational database systems, it is difficult to store or retrieve objects of arbitrary length; e.g., portions of documents or programs. POSTGRES — the successor of INGRES — is an attempt to provide better support for complex objects[11]; however, the system is still not available. To solve the problem, we store the beginning line number and ending line number of each software object in the database and use these line numbers to access the source file in which it is defined‡.

(3) *Open Design:* Most database systems provide a query language to users, but a user must first enter the database environment in order to use the query language; thus, the advantages of many Unix system features can not be used, e.g., pipelining and input/output redirection. The Information Viewer is designed is such a way that every command can be executed at the Unix command level; therefore, no startup time is involved and the retrieved data can be easily processed by other Unix commands. Moreover, InfoView commands access the data files directly, it is not necessary to load the data into the database first. The same approach is used in the FFG database system[12].

† It is not necessary to create a datafile for each relationship. For example, the relationship between gbtype and file can be found in *gbtype.data*, the relationship between gbtype and gbvar can be found in *gbvar.data*.
‡ For each function, we also store the ending line number of its header.

The C parser stores data files in the same directory in which the source files are kept. The name of this directory is stored as an environment variable to InfoView. A summary of the major commands implemented in InfoView follows:

(1) **swsrc** *source_dir:* switch the context of InfoView to a new source directory.

(2) **view** *obj_type obj_name:* print out the whole object.

(3) **info** *obj_type obj_name:* print out an object's attributes.

(4) **rel** *obj_type1 obj_type2 obj_name1 obj_name2:* print out the relations between obj_type1 and obj_type2 (see Table 1); if obj_name2 (obj_name1) is specified as '-', print out all the objects of obj_type2 that are related to obj_name1 (obj_name2).

(5) **header** *function_name:* print out the header of the function.

(6) **body** *function_name:* print out the body of the function.

(7) **summary** *function_name:* print out the summary of the relations between a function and other software objets.

These commands are useful for finding out where an object is defined, what it is about, and the relationship it holds to other software objects. In the following, we use an example to show how the Information Viewer can be used step by step to get the information a user might need for understanding a function. In each step, the output follows the command line, which is shown in bold face. Explanations of the steps follow the example.

[1] info function inouter

```
   in_file         data_type        func_name      static bline hline eline
--------------- --------------- --------------- ------ ----- ----- -----
inouter.c       int             inouter         n      14    18    31
```

[2] header inouter

```
/*:*  purpose: handles input and output queues;;  *:*/
inouter(inoutq, endq, runq)
JQUEUE *inoutq;             /* INOUTER's input queue of JCB's */
JQUEUE *endq;               /* ENDER's input queue of JCB's */
JQUEUE *runq;               /* RUNNER's input queue of JCB's */
```

[3] view gbtype JQUEUE

```
typedef struct jqueue JQUEUE;           /* job queue */
```

[4] view gbtype struct jqueue

```
struct jqueue {                         /* job queue */
    SEM *count;                             /* number of jobs on queue */
    SEM *mutex;                             /* for exclusive access to queue */
    JCB *head;                              /* first job on queue */
    JCB *tail;                              /* last job on queue */
    char *name;                             /* pointer to queue name */
    char cname[20];                         /* name for count semaphore */
    char mname[20];                         /* name for mutex semaphore */
};
```

[5] rel function function inouter -

```
    caller_file          caller_func          callee_file          callee_func
-------------------- -------------------- -------------------- --------------------
inouter.c            inouter              iosim.c              iosim
inouter.c            inouter              jqueue.c             jget
inouter.c            inouter              jqueue.c             jput
```

[6] rel function function - inouter

```
    caller_file         caller_func         callee_file         callee_func
------------------- ------------------- ------------------- -------------------
main.c              main                inouter.c           inouter
```

[7] rel gbvar function - inouter

```
      var_name            used_in_file          used_in_func
-------------------- -------------------- --------------------
queue_length         inouter              inouter.c
```

Step 1:	Find out the attributes of the function `inouter`.
Step 2:	Take a look at the header of `inouter`.
Step 3:	Find out what the data type `JQUEUE` is.
Step 4:	Find out the structure of `jqueue`.
Step 5:	Find out which functions are called by `inouter`.
Step 6:	Find out which functions call `inouter`.
Step 7:	Find out which global variables are used inside `inouter`.

Without the Information Viewer, a user will have to search through many files to get the above information.

The query syntax used in InfoView is less complex than INGRES. For most database accesses, users are not required to memorize the field names of each relation; they only have to use the names of the five object types and names of the referenced objects to construct a query. For example, compare the following two queries; both are used to print out the functions called by the function **main**:

(a) Information Viewer:
```
rel function function main -
```

(b) INGRES:
```
range of ff is funcfunc
retrieve (ff.all) where ff.caller_func="main"
```

However, INGRES provides many powerful functions that are not available in the Information Viewer; e.g. join operations, aggregate functions, etc. Therefore, we provide an utility command that creates an INGRES database schema (using the conceptual model shown in Figure 3), and loads the data files automatically. A user can then invoke INGRES commands on the database using sophisticated relational queries.

4.0 Performance

Our C Information Abstractor (CIA) has been implemented and successfully tested on several programs. To illustrate the performance of CIA, we collected some data by running CIA on the following source programs:

(1) *Socket*: a remote job execution program.

(2) *Air*: an air line reservation program — a typical student project.

(3) *Grn*: a typesetting preprocessor for picture files.

(4) *Toy.os*: an experimental operating system used by Berkeley students†.

† The version we used has several modifications.

(5) *Gremlin* a graphics editor for SUN workstations.

The data we collected are listed in Table 2 and Table 3. Table 2 lists the following items: (1) the number of source files of the program, (2) the total number of functions in that program, (3) the total number of lines in all the source files of that program, (4) the average function length, (5) the number of relations extracted from the source files, (6) the number of cpu seconds spent by our C parser on abstracting the program, and (7) the number of cpu seconds spent by the C compiler on the same program. Both (6) and (7) are measured on a VAX 780. Table 3 shows the size of the output data files (excluding the *comment.data* file) created by CIA on the five programs.

Table 2. The Performance of CIA on Five Programs

program name	*num. of files*	*num. of functions*	*num. of lines*	*avg. func. length*	*num. of relations*	*abstraction time*	*compilation time*
Socket	6	7	414	34	84	17.7	25.8
Air	9	22	1083	34	194	19.4	41.0
Grn	6	26	1519	33	398	18.7	37.6
Toy.os	43	75	2378	20	806	66.1	119.0
Gremlin†	21	388	11436	17	3900	427.2	904.8

Table 3. The Size of the Data files for Five Programs

datafile	*Socket*	*Air*	*Grn*	*Toy.os*	*Gremlin*
file.data	*6*	*9*	*6*	*43*	*21*
gbvar.data	*8*	*18*	*47*	*44*	*239*
gbtype.data	*3*	*2*	*9*	*19*	*10*
macro.data	*7*	*10*	*62*	*59*	*203*
function.data	*7*	*22*	*26*	*75*	*388*
filefile.data	*25*	*23*	*9*	*78*	*147*
funcfunc.data	*6*	*26*	*41*	*217*	*874*
gbvrfunc.data	*8*	*19*	*117*	*120*	*871*
macrfunc.data	*14*	*65*	*81*	*151*	*1147*
total	*84*	*194*	*398*	*806*	*3900*

As shown in Table 2, the time the C parser spent on the abstraction process strongly depends on the number of functions. Because CIA ignores most details within functions except for references to global objects, the number of functions has a larger impact on the abstraction time than the number of lines does. We can also see from Table 2 that the abstraction time CIA spent on a program is approximately half of the compilation time for the same program.

Table 3 reveals some structure information of the five programs:

Socket has a relatively large number of file-file bindings (25) because it uses a considerable amount of header files for networking. This implies that it is relatively system-dependent, because some Unix systems may not have all the header files it needs.

Air has only two data types, which implies that the data structure complexity of this program is relatively low.

Grn needs to set up a global typesetting environment, thus it has a large number of global variables (47), macros‡ (62), and strong bindings between functions and global variables (117). This implies that it is difficult to reuse a function of *Grn* in other application programs, since a large

† Gremlin has 21 source files and 85 icon data files. The icon data files are ignored by the information abstractor. Because Gremlin can only be compiled on SUN workstations, the compilation time is measured on a SUN-2 workstation instead of a VAX780.

‡ These macros are mainly constant definitions.

global environment needs to be built before using that function.

Toy.os employs a large number of functions (75) and function-function bindings (217). To reuse a function in *Toy.os,* a large number of functions have to be retrieved at the same time.

Gremlin has problems similar to what *Toy.os* and *Grn* have.

The above discussion gives some hints as to how one uses the program database to calculate some metrics, and how one interprets those metrics. In Section 6.3, we will further explore this topic.

The utility commands of the Information Viewer were originally written in Unix shell scripts[13]; we then transformed most of the commands to C programs for speed. Table 4 shows the performance of four typical Information Viewer commands on the five program databases. The running time is measured in the number of cpu seconds (on a VAX780). The four InfoView commands are:

(1) **list function** : print out the attributes of all functions
(2) **info function main** : print out the attributes of the function main
(3) **rel function function main -** : print out functions called by the function main
(4) **view function main** : print out the function main

Table 4. The Performance of Information Viewer on Five Programs

program	list	info	rel	view
Socket	0.1	0.1	0.1	0.3
Air	0.2	0.1	0.2	0.6
Grn	0.2	0.1	0.2	0.3
Toy.os	0.6	0.2	0.5	0.4
Gremlin	3.7	0.7	1.6	0.8

Each of the above InfoView commands accesses one datafile; the *view* command additionally accesses one source file. In general, the speed of InfoView commands depends on the length of the datafile accessed. As shown in Table 4, these commands are fast and there is no startup overhead associated with these commands. This fact makes the InfoView system highly interactive.

5.0 Extensions

Through the experience in implementing and using the C Information Abstractor (CIA), we found that several extensions are necessary. We have several ongoing plans for these extensions, which are described in the following subsections.

5.1 Structured Comments

Some information cannot be automatically derived from the code, e.g., the assumptions made by a programmer, the specific algorithm used by a particular function, and the computational complexity of an algorithm, etc. One possibility is to store the information in comments; however, the C language does not provide a way to specify the association between the comments and software objects like modules, functions, and global variables. Take an example from C programs:

```
int person;
/* number of persons; */
count() { ... };
```

How can the C Parser tell whether the comment is for the global variable **person** or for the function **count**? This makes it impossible for the InfoView to retrieve an object with its comments. Because of the ineffectiveness of putting information into comments, many programmers do not pay attention to it. The result is that software maintainers may spend several hours trying to figure out a minor detail in a program written by other people.

Structured comments is a way to make associations between objects and comments by using some simple comment placement rules. A structured comment has two possible forms: *before comment* and *after comment*. A *before comment* should be placed before a software object; an *after comment* should be placed after a software object. Each structured comment begins and ends with a special marker. The general form of a *before comment* is shown here:

```
/*:*
     attribute1: value1;;
     attribute2: value2;;
     ....
     attributen: valuen;;
*:*/
```

An *after comment* is basically the same, with the exception that it starts with `/**:` and ends with `:**/`. With the help of an advanced editor, some normal comments in existing programs can be transformed to structured comments very easily. We force these comment placement rules in order to make the association between comments and software objects easier. For example, given a piece of C program like the following:

```
int queue_length;
/*:* purpose: initializes a queue ;; *:*/
InitQueue() { ... }
```

Since the comment used is a *before comment,* the parser can easily identify that the comment is intended for **InitQueue** rather than for **queue_length**. Therefore, the comment and the function can be retrieved together by the *view* command in InfoView.

Besides making the association of comments and objects, we also suggest the idea of *comment attributes* to organize the contents of comments in a structured way. Useful comment attributes for function objects include: (1) purpose, (2) assumption, (3) condition, (4) assertion, (5) algorithm, and (6) complexity. Here is an example:

```
/*:*
    purpose: to sort an array of integers ;;
    assumption: only positive integers are specified in the array ;;
    condition: this function can only be called after arrayinit ;;
    assertion: after execution, global variable SORTED is set to 1 ;;
    algorithm: Quick sort, reference [Ullman 1984] ;;
    complexity:  O(N*logN), N is the length of the array ;;
*:*/
Qsort (input_array)
int input_array[];
{ .... }
```

These attributes are optional, the programmer may select to use only a few of them that are relevant to a particular object. The set of attributes to be recognized by the parser can then be specified in the abstraction rules.

A set of new InfoView commands can be developed to take advantage of the structured comments. For example,

(1) **comment** *object_type object_name:* retrieve the comment of an object.

(2) **showattr** *object_type object_name attribute_name:* retrieve a specific attribute of an object

5.2 Incremental Updates of the Database

Current implementation of the CIA requires that the whole database be regenerated to get the most up-to-date information if any of the source file is updated. This is acceptable if a user just wants to understand a relatively steady program. However, if the software is still under development, it is desirable to have the C Parser update only the portion of the database that is relevant to the changes in the software, in a spirit similar to incremental compilation.

We plan to solve this problem by employing a strategy similar to the one used by the C compiler. Each file can be processed independently by the C Parser, and a symbol table file that records the object attributes and references is created. A final linking process combines symbol tables for all the source files and creates the final data files. Whenever any change is made in a file, it is only necessary to regenerate the symbol table for that file, and then performs the linking process again to get a new set of data files.

5.3 Abstraction on Other Textual Forms

The Information Abstractor can handle all C source programs; however, Lex and Yacc files are usually a part of many software on Unix systems. For example, the C Parser itself consists of one Lex file, two Yacc files, and several C files. Therefore, we would like to extend the information abstraction system to understand the structure of Lex and Yacc files and store information like the relation between rules and tokens, and where rules and tokens are defined.

A software system usually consists of requirements, design documents, user manuals, test cases, and source programs. The C Information Abstractor only handles the source programs; however, an information abstractor can be developed to create a database for each type of structured text used in software systems. The relationship among these these software elements can then be traced using these databases. The importance of this type of traceability is discussed in[14] and[15].

6.0 Applications

We can explore many applications to take advantage of the program database. We will discuss four of them in this section: (1) *Multiple Software Views*, (2) *Software Metrics*, (3) *Software Reusability*, and (4) *Software Restructuring*. The ideas we discussed in these subsections can be applied to programs written in most programming languages — if an Information Abstractor is available for each of the languages.

6.1 Multiple Software Views

A graphics program with the ability to provide multiple views of programs greatly simplifies the task of program understanding. Using the program database, such a graphics interface can be implemented easily. For example, in Figure 4, when a user is trying to understand the function **bput** in **bqueue.c**, he needs to view the functions **bqinit** and **bget**; when he switches to **bqinit**, then he may need to know what the global variable **state** is. Views on these objects can be retrieved from the source files very easily using InfoView commands and displayed on workstations that support window management. If a user wants to get a high level view of the object **bput**, then another window could be created to display the the relations between **bput** and other global objects. We are planning to build such a system on SUN workstations.

6.2 Software Reusability

Reusing a piece of software that has been tested will greatly reduce the development time for a new software product; however, two necessary conditions for reusing a software object are: (1) the limitation and functionality of that object must be well understood, (2) all software objects that are referenced in that object must also be identified and retrieved at the same time.

Information described in (1) can best be provided by the programmer who wrote that piece of program; and this information can be described using the structured comments described in the Section 5. Information described in (2) can be easily derived from the program database.

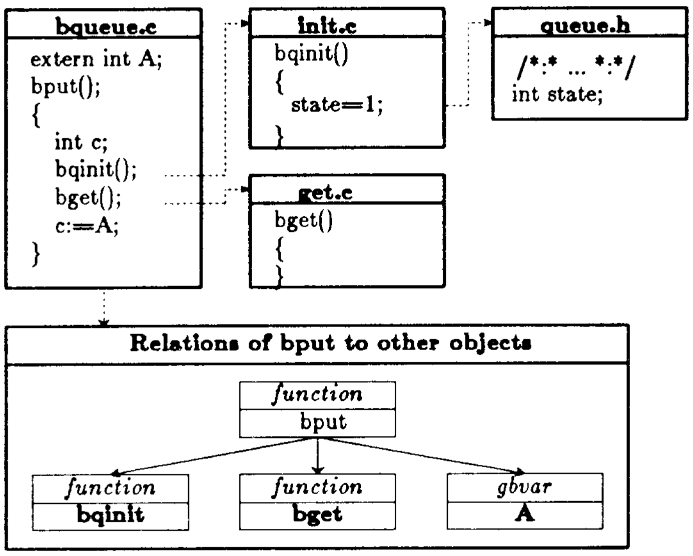


Figure 4. Multiple Program Views

6.3 Software Metrics

In large software projects, metrics play an important role in estimating software quality, amount of testing required, maintenance requirements, etc. Examples of software metrics are:

(M1) function-file binding: the number of relations between a function and global objects in another file

(M2) file-file binding: the number of relations between global objects in two files

(M3) retrievability: the number of objects associated with a particular function

(M4) data structure complexity: the number of data types used in the program

(M5) the average length of functions

(M6) the average number of functions used in a software module

(M7) the number of calling paths starting from a function

(M8) the depth of a calling path

(M9) McCabe's cyclomatic number[16]: the maximum number of independent paths in a program

(M10)Halstead's metrics[17]: the vocabulary, length (both in terms of unique operands and operators) and the volume of the program.

These metrics are helpful in finding poorly structured software modules. For example, a large value of (M1) indicates that a function should be moved to the file that has strong bindings with it. A large value of (M2) indicates that two files should be merged. A large value of (M3) indicates a function can not be reused without bringing together a large context. Similarly, (M4) to (M8) reflects the program structure in various ways. Metrics (M1) through (M8) can be obtained from the program database created by the C Information Abstractor (CIA). (M9) and (M10) can be obtained by modifying the conceptual model used by CIA.

6.4 Software Restructuring

The metrics information usually leads to the conclusion that a program must be restructured. Since all global interrelations of the programs are stored in the database, the effort of restructuring programs is greatly reduced. The C Information Abstractor can help trace the ripple effects caused by a restructuring operation by examining the relations stored in the database.

A software restructuring process consists of several basic operations: *moving, renaming, deleting,* and *inserting* a software object. Each operation requires careful examination of the affected objects. For example, Figure 5 shows that a function **p1** is to be moved from t1.c to t2.c:

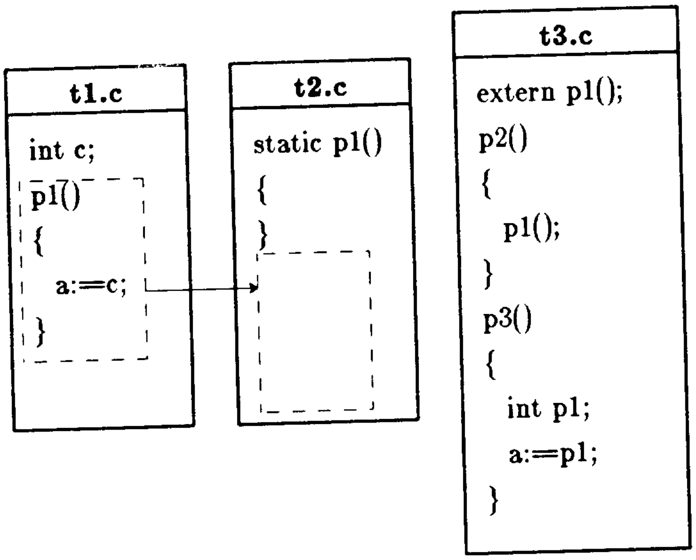


Figure 5. Move the function **p1** from t1.c to t2.c

Although the task looks simple, it is difficult for a programmer to do it right. One way a programmer might do is the following:

(1) Find out all global objects that are defined in t1.c and used in function **p1** — in this case, the global variable **c**;

(2) If the global variable **c** is used in other places (other than the function **p1**) in t1.c, then declare

```
extern int c;
```

in t2.c; otherwise, move declaration "**int c;**" from t1.c to t2.c;

(3) Check if there is another procedure in t2.c which uses the same name **p1** — in this case, yes;

(4) Rename the function **p1**, which just got moved from t1.c, to some new name **px**;

(5) Check if there are functions in other files that reference the original **p1** in t1.c — in this case, yes, in t3.c;

(6) Replace the reference (to **p1**) in t3.c by a new reference to **px** — but be careful, don't change the local variable **p1** in function **p3**.

As we can see, even a simple movement of function may end up with detailed checkings over many functions and files. Similar checkings must be done for deletion and insertion of objects. We would like to be able to handle these details automatically using the relational information available in the program database.

7.0 Conclusion

We have implemented an information abstraction system for C programs. Through our experience, we found that the idea of abstracting program information into a database provides effective support for software understanding. However, a simple conceptual model for the program database and efficient data retrieval commands were critical to the success of this abstraction system. The successful implementation and the availability of the program database promotes more research effort on several software engineering areas: Multiple Software Views, Software Reusability, Software Metrics, and Software Restructuring.

Acknowledgement

Many people contributed to the implementation of CIA. Mike Nishimoto and Lenora Eng implemented the C Parser — the kernel part of CIA; Joo-Seok Song wrote some of the shell scripts for the Information Viewer; Wen-Ling Chen converted some shell scripts to C programs and maintained the INGRES database for Information Abstractor. We would also like to thank Atul Prakash, Wei-Tek Tsai, Yutaka Usuda, Vijay Garg, Tsuneo Yamaura, Anupam Bhide, Rajeev Aggarwal, Bruce Wachlin, Doyt Perry, Charles Fritz, Michael Buckley, and Robert Yacobelis for valuable comments and discussion throughout the development of this project. Special thanks are due to Ms. Bibbs for editorial advice.

References

1. C.V. Ramamoorthy, Atul Prakash, Wei-Tek Tsai, and Yutaka Usuda, "Software Engineering: Problems and Perspectives," *IEEE Computer*, vol. 17, no. 10, pp. 191-209, October 1984.
2. Dennis M. Ritchie, "The C Programming Language - Reference Manual," *UNIX Programmer's Manual*, vol. 2, 1984.
3. M. A. Linton, "Implementing Relational Views of Programs," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984.
4. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley Publishing Company, 1984.
5. M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352-357, July 1984.
6. Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report,* Springer-Verlag, 1975.
7. AT&T Bell Laboratories, *Unix System V Programmer's Manual*, 1985.
8. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems*, vol. 1, no. 3, pp. 189-222, September 1976.
9. M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator," *Unix Programmer's Manual*, vol. 2, 1984.
10. Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler," *Unix Programmer's Manual*, vol. 2, 1984.
11. Michael Stonebraker and Lawrence A. Rowe, "The Design of Postgres," *UCB/ERL 85/95*, University of California, Berkeley, November 1985.
12. Doug Comer, "The Flat File System FFG: a database system consisting of primitives," *Software - Practice and Experience*, November 1982.
13. S. R. Bourne, "An Introduction to the UNIX Shell," *UNIX Programmer's Manual*, vol. 2, 1978.
14. Y. Usuda, "The Design and Implementation of Evolution Support Environment (ESE)," Master Report, Computer Science Division (EECS), University of California, Berkeley, May 1985.
15. C. V. Ramamoorthy, Y. Usuda, W. -T. Tsai, and A. Prakash, "Genesis: An Integrated Environment for Development and Evolution of Software," *COMPSAC*, 1985.
16. Thomas J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, December 1976.

17. M.H. Halstead, *Elements of Software Sciences,* Elsevier, New York, 1977.

NAME

cia – information abstractor; a program to extract relations from C programs and place them into a database.

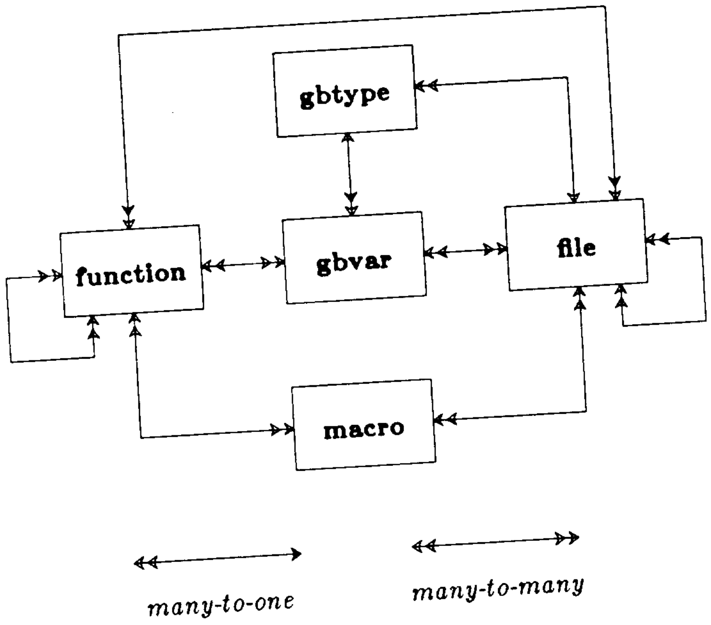
SYNOPSIS

cia [-c][-d][-h][-i][-l][-m][-o][-p][-s][-u][-w][-D][-I][-U] filename1 filename2 ...

DESCRIPTION

The C Information Abstractor is a program which extracts relations from C source files and place them into a C program database where they can be accessed with an information viewer; see **infoview(1).**

A conceptual model of the C program database used by *cia(1).* is shown below in Figure 1. This model defines the objects and relations recognized by the information abstractor. For more details on the specific relations, see **infoview(1).**



Many of *cia's* options will rarely be used by the average programmer, while others were added only for the *cia* administrator. They are as follows:

-m	Print menu of commands for *cia.*
-c	Incremental compilation of files given to *cia*. Output files are of form file1.st, file2.st... To combine all information, pass *.st files to *cia*. [Not implemented yet]
-u	Incremental update of database. Once a file is modified but has already been run through *cia,* use this option to rebuild the database by passing files modified or added to program being parsed. [Not implemented yet]
-i	Output information about include files. See limitations below.
-s	Create another relation which will enable a user to see which system calls were made in which functions.
-w	Suppress output diagnostic messages of *cia*. These are messages telling users the status of each file's parse. If syntax errors occur within *cia,* error messages will be printed regardless of this flag.

Initialize define value of macro. Works exactly like option in *cc(1)*, where flag can be of the form **-Dname=expansion** or **-Dname.** In the first case, **name** gets initialized to **expansion,** and in the second, **name** gets initialized to **one.**

Add include file directory to current directory and /usr/include directory. The order of search is:

1) Current directory
2) -I directories
3) .incdir directories, see **STARTING UP.**
4) /usr/include directories.

The order is identical to that of *cc(1)* except that .incdir directories are inserted after the -I directories but before /usr/include directories.

-U Undefine the first occurrence of the given macro.

CIA maintenance options

-d Debugging flag—Output simple debugging information about tokens, strings, and end-of-files.

-h Debugging flag—Output hash values of strings.

-l Debugging flag—Output low level debugging information about state changes within the parser.

-o Leave output of pre-processor files in current directory. Files will end with ".co"; this suffix can not be changed.

-p Input files were processed by the pre-processor of *cia* and do not need to be processed again before parsing can begin.

STARTING UP

Cia was written on a VAX-11/750. If the instruction set of the machine is different, *cia* will have to be recompiled. Otherwise, *cia* can be placed in any directory, and can be run by putting this directory in the user's path, see *csh(1)*. *Cia* must be executed from the directory, where source files exist.

An option exists for adding include file directories in the search path of the *cia* without having to use the -I option. Create a file in the directory containing the source files[where *cia* will be run] which has the full path names of these directories. Only one directory name can be place on a line; and a maximum of about 8 is enforce by the pre-processor of *cia*.

OUTPUT FILES

The output files of the cia are the database files used by *infoview(1)* to display information about source files to users. Intermediate compilation files and pre-processor files are also output and have the form *.st and *.co, respectively. The database files output are: comment.data, expand.data, file.data, filefile.data, funcfunc.data, function.data, gbvar.data,gbvrfunc.data, gbtype.data, macro.data, macrfunc.data, system.data, and unknown.data. Entries of each database file are:

comment.data
1) File name where comment found
2) Object comment belonged to
3) Name of object
4) Begin and end lines of comment

expand.data
1) File name of where expansion occurs
2) Name of macro expansion
3) Line number of expansion

file.data
1) File name passed *cia*
2) Number of lines in file

filefile.data
1) File name passed *cia*
2) Included files of that file

funcfunc.data
1) Function name
2) Name of file containing the function
3) Other functions called within the given function
4) File names of where these other functions can be found.

function.data
1) Function name
2) Name of file containing the function
3) Type of the function
4) Staticness of function
5) Begin line of [pre-comment for] function
6) Line where body of function starts
7) End line of [post-comment for] function

gbvar.data
1) Global variable name
2) Name of file containing the global variable
3) Type of global variable
4) Staticness of variable
5) Begin line of [pre-comment for] variable
6) End line of [post-comment for] global variable

gbvrfunc.data
1) Function name
2) Name of file containing function
3) Names of global variables found in function

gbtype.data
1) Name for a global type
2) Name of the file containing the declaration of that type
3) Begin line of [pre-comment for] type
4) End line of [post-comment for] type

macro.data
1) Name of macro
2) Name of the file containing the definition of macro
3) Begin and end line of macro

macrfunc.data
1) Name of function

2) Name of file containing function
3) Macro name used within function

system.data

1) Name of function
2) Name of file containing function
3) System call within function

unknown.data

1) Name of function
2) Name of file containing function
3) Unknown word within function

ERROR MESSAGES

WARNING: Include file stack underflow. Usually occurs on parse of yacc output.

ERROR: Include file stack underflow. Should never happen. Notify *cia* administrator.

Others exist, but should not be seen by legal C programs. [legal=compilable]

LIMITATIONS

1) When processing an include file, information about its contents are placed in the database once. For example, if **bar.c** includes **foo.h** and **sna.c** includes **foo.h** and both **bar.c** and **sna.c** are parsed by *cia* with the "-i" flag, information about foo.h will be output only once as should be. However, if **foo.h** contains different code due to conditional statements like "#ifdef", "#ifndef", or "#if xxx", *cia* will only output information from **foo.h** on the initial parse of **foo.h.**

2) When viewing relations of macros/global variables/functions used in functions, *cia* might say, for example, that a global variable appears in a function while subsequent viewing of the function shows otherwise. This occurs when the object is placed inside of the function as a result of a macro expansion.

3) Files passed to *cia* must be individually compilable. In other words, if the file can not be compiled with "cc -c filename.c", it should not be passed through *cia*. ".h" files passed to *cia* must be of the same format as ".c" files, that is must be able to be renamed from "xxx.h" to "xxx.c" and compiled with "cc -c xxx.c".

4) Pre-comments and post-comments can not be used for macros, conditional statements, or include files.

BUGS

1) Some versions of *cia* may not be able to handle "#if xxx" statements properly. It might not process information inside of this kind of construct.

2) *Cia* may not operate properly if a global variable is declared using C's option of no type specifier. For example, defining a global variable as follows:

```
variable;          or
variable = 5;
```

may cause problems. This style of defining integer variables is not allowed inside functions and so poses no problems to *cia*.

3) At the end of files, software objects may not be recognized by *cia* or may show up as being recognized in another file. If this happens, please contact the *cia* administrator.

4) Hopefully no others.

FUTURE IMPROVEMENTS

1) Allow user to specify suffixes for pre-processor output file and incremental compilation file.

2) Allow incremental updating of database.

3) Add more relations from data already available to *cia*.

4) Allow incremental compilation of database[if not done yet].

5) Most of the possibilities lie in the area of using the data produced by *cia* to perform metrics and source manipulation within files. Workstation window commands are another possibility.

6) The possibilities are endless.

SEE ALSO

infoview(1), cc(1)

AUTHOR

Cia was implemented by Michael Nishimoto with help from Lenora Eng. Program designers were Michael Nishimoto and Yih-Farn Chen. The relational database schema was mainly designed by Yih-Farn Chen with modifications by Michael Nishimoto. This manual is written by Michael Nishimoto.

NAME

infoview – a set of commands that access the C program database created by *cia(1).*

SYNOPSIS

swsrc source_directory
info [**-u**] object_type object_name
rel [**-u**] object_type1 object_type2 object_name1 object_name2
view [**-n**] object_type object_name [file_name]
list -u datafile_name
header function_name [file_name]
body function_name [file_name]
summary function_name [file_name]
see [**-n**] file_name begin_line end_line
pgm
infoview

DESCRIPTION

Information Viewer is a set of commands that access the C program database created by the C information abstractor; see *cia(1).* The conceptual model of the C program database used by *cia(1)* is shown in Figure 1. This model defines the objects and relations recognized by the information abstractor.

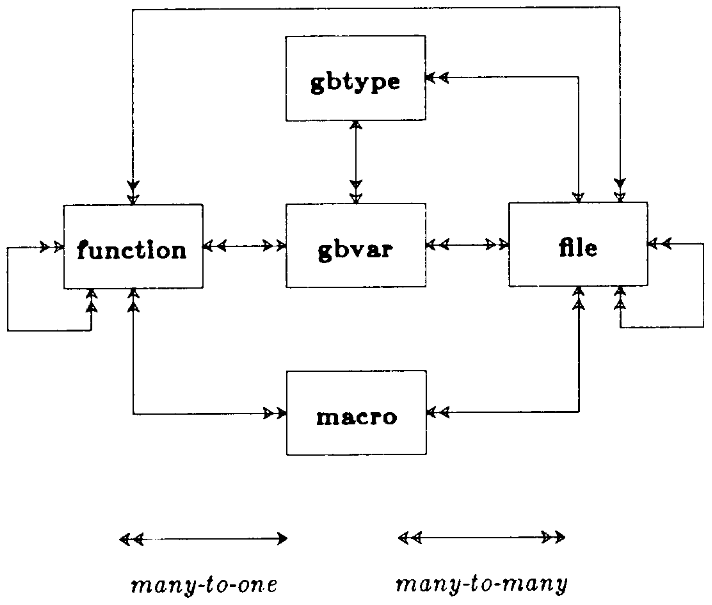


Figure 1. The Conceptual Model of the C Information Abstractor

Each box in Figure 1 represents an object. Five object types are defined: *file, function, gbvar, gbtype,* and *macro.* Note that macros are those identifiers defined in the *#define* statements. Each arc in Figure 1 represents a relationship. Definitions of the nine relationships are shown in Table 1. The relationships listed are certainly not all the relationships one can find among all the object types; but they provide most of the relational information a programmer might need from the program database.

#	*obj_type1*	*obj_type2*	*rel_type*	*definition*
1	file	file	m-to-m	file1 includes file2
2	function	function	m-to-m	function1 calls function2
3	gbvar	function	m-to-m	gbvar1 referenced in function2
4	macro	function	m-to-m	macro1 referenced in function2
5	function	file	m-to-m	function1 referenced or defined in file2
6	macro	file	m-to-m	macro1 referenced or defined in file2
7	gbvar	file	m-to-m	gbvar1 referenced or defined in file2
8	gbtype	file	m-to-1	gbtype1 defined in file2
9	gbvar	gbtype	m-to-1	gbvar1 defined as a gbtype2 variable

Table 1. Definitions of the Nine Relationships

Swsrc switches the context of information viewer to a new source directory; the ten data files created by *cia(1)* and source files in the specified source directory will be used by all the following *infoview* commands.

Example: swsrc ~/ditroff/grn

Info gives the information about an object's attributes. An object is specified by its object_type and object_name. If several objects in different files share the same name, information about all these objects will be listed. If "-u" option is specified, then the output will not be formatted.

Example: info function QueueInit

Rel lists the relational information between two objects. All the nine relations listed in Table 1 are explicitly or implicitly stored in the ten data files created by *cia(1)*. The *rel* command accesses these data files and prints out the relational information requested. If "-u" option is specified, then the output will not be formatted. The two object types can be specified in any order. One of the object names should be specified as "-". This is best explained by the following examples:

Example1: rel function gbvar - queue
{print out all the functions that reference the global variable queue}

Example2: rel file file - stdio.h
{print out all the files that include the file stdio.h}

View prints out the specified object. If "-n" option is given, then a line number will be printed before each line. The file_name argument is necessary if two or more objects in different files share the same name.

Example1: view function InitDB db.c

Example2: view -n gbtype "struct buf"

List prints out the contents of the specified data file in a nice format. Any of the ten data files, namely *file.data, function.data, gbvar.data, macro.data, gbtype.data, comment.data, filefile.data, funcfunc.data, gbvrfunc.data, macrfunc.data,* can be printed. If "-u" option is specified, then the output will not be formatted.

Example: list gbvar

Header prints out the function header, which includes the comments and argument list, of the specified function. If two or more functions in different files share the same name, an optional filename can be specified.

Example: `header PushStack stack.c`

Body prints out the body of the specified function. If two or more functions in different files share the same name, an optional filename can be specified.

Example: `body PopStack`

Summary prints out the summary of the relationships between a function and and other objects, namely relationship 2, 3, and 4 listed in Table 1. If two or more functions in different files share the same name, an optional filename can be specified.

Example: `summary PushStack`

See prints out a part of the specified source file, from begin_line to end_line. If "-n" option is specified, then the line number is shown before each output line.

Example: `see -n main.c 23 45`

Pgm tells you from which source directory the infoview commands are accessing the data files and source files.

Infoview gives a listing of the commands and options of all the infoview commands.

SEE ALSO

cia(1)

AUTHORS

Most of the InfoView commands were originally written in shell scripts by Joo-Seok Song and Yih-Farn Chen, and later translated to C programs by Wen-Ling Chen and Yih-Farn Chen.