Tutorial on the C Information Abstraction System

Michael Nishimoto

Yih-Farn Chen

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

ABSTRACT

This tutorial describes the C Information Abstraction System, a tool which helps fill the needs of C programmers working with large software programs. The system consists of two subsystems: the C Information Abstractor(CIA) and the Information Viewer(INFOVIEW). CIA is a program which abstracts compilable C source code into a set of relations and summaries of software objects. This set is stored in a program database where INFOVIEW can use it to produce vital information necessary for program understanding. The main role of INFOVIEW is to provide a simple user interface to the CIA database; however, it also functions as a programming interface for individuals who wish to use the available information to create their own systems. Written originally on a VAX 11/750 operating 4.3BSD, CIA and INFOVIEW have been ported to UNIX System V† and SUN 3.0 systems.

Table of Contents

	Page
1. INTRODUCTION	2
2. THE C INFORMATION ABSTRACTOR	2
3. THE INFORMATION VIEWER	6
4. PROGRAM DEVELOPMENT	20
5. STATUS AND FUTURE WORK	24

Appendices

A. Summary of CIA Options	26
B. Detailed Description of Information in the Database	27
C. Summary of INFOVIEW Commands	31
D. Summary of INFOVIEW Library Commands	32

This work is supported in part by MICRO under contract No. 532434-19900.
†UNIX is a trademark of Bell Laboratories.

1. INTRODUCTION

Maintenance responsibility for programs frequently shifts from person to person throughout a software life cycle. Each time this occurs, the ease of transition is determined by the complexity of the software system and the amount of documentation available. A programmer is often confronted with either the situation of modifying inadequately documented code or the situation of trying to remember specific details about his own code. It would be helpful to have a software tool that stores and retrieves program structure information, so programmers can reduce the time involved with program understanding.

The C Information Abstractor System(CIAS) is such a software tool. Two subsystems make up CIAS: the C Information Abstractor(CIA) and the Information Viewer(INFOVIEW). The C Information Abstractor is a program which abstracts compilable C source code into a set of relations and summaries of software objects. This set is stored in a database where INFOVIEW can use it to produce vital data necessary for program understanding. INFOVIEW's main role is to provide a simple user interface to CIA's database; however, it also functions as a programming interface for individuals who wish to use the available information to create their own systems. Figure 1 illustrates how CIA and INFOVIEW interact. A decision to build CIA separately from the C compiler was made because the C compiler contains many complexities unnecessary for our purpose. Moreover, information abstraction is usually performed after all syntax errors have been removed from programs.

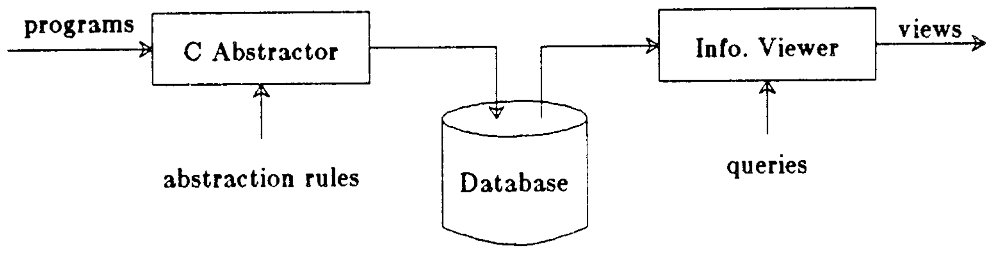

Figure 1. Outline of the C Information Abstraction System

This tutorial corresponds to version 1.0 of the CIA and INFOVIEW system. Written originally on a VAX 11/750 operating 4.3BSD, CIA and INFOVIEW have been ported to UNIX System V and SUN 3.0 systems with little effort due to their portable C implementation. The only differences in code resulted from limitations of the various environments such as the length of a file name or the number of open files.

2. THE C INFORMATION ABSTRACTOR

The C Information Abstractor views programs as software objects and relationships between these objects. Each object has a set of attributes which describe the location, data type, and other information of the object. Figure 2 shows the conceptual model of the C program database we adopted. When C programs are abstracted by CIA, information about the five object types —files, functions, global variables, types, and macros— symbolized by boxes, and the eleven relationships, symbolized by lines, are abstracted and placed into a program database. Solid lines represent currently implemented relationships while dotted ones represent future extensions. Table 1 summarizes the nine present relationships and two future extensions of CIA. For a discussion on the design of the conceptual model, see[1]. Appendix B gives a more detailed information of the database files.

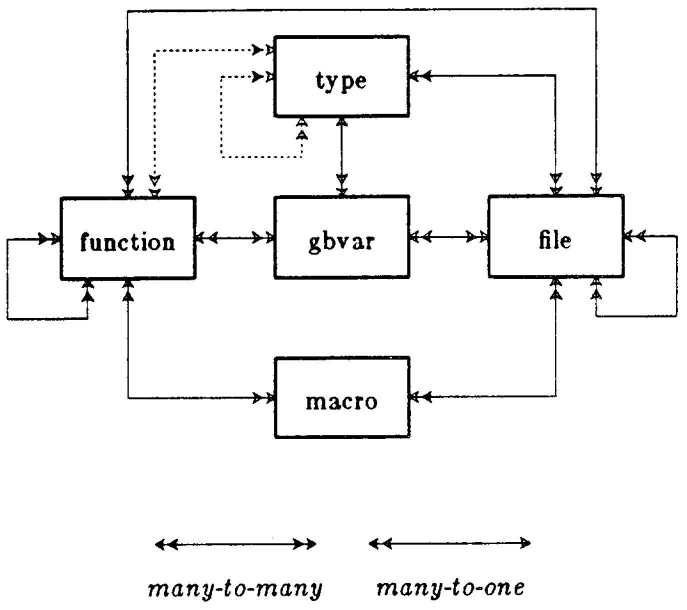

Figure 2. The Conceptual View of the C Program Database

Table 1. Definitions of the Eleven 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	type	file	m-to-m	type1 referenced or defined in file2
9	gbvar	type	m-to-1	gbvar1 defined as type2
10*	type	function	m-to-m	type1 referenced or defined in function2
11*	type	type	m-to-m	type1 referenced in type2

* Future Extensions

2.1. Starting-up

As CIA and INFOVIEW are not a standard part of BSD UNIX at this time, they may not be on your current machine. The executable code should be in /usr/local/cias. If it is not, please send mail to cia@ucbarpa.Berkeley.EDU to receive the latest information on the location of the CIA/INFOVIEW object code. Once the location is known, place it into your path. See the path mechanism in *csh(1).*

2.2. An Example

Throughout Section 2 and Section 3, we will use the following C program source as an example to show the basic usage of the CIA system. The example program contains software objects representative of what a C programmer might encounter, and it should be available in /usr/local/lib/cias/example. Copy files in that directory to a directory of your own, so you can create/modify your own program database as you proceed.

<FILE: chain.h>

```c
#define LEN 10
#define NIL ((LINK *) 0)

typedef char NAME;

struct link {
    NAME   name[LEN];
    struct link *next;
};
typedef struct link LINK;

static LINK *tempLink;
```

<FILE: chain.c>

```c
#include <stdio.h>
#include "chain.h"

/*::******************************
*  file: chain.c
*  contents: High level routines
******************************::*/
/*:****************************:*/
/*:* function: FindName      *:*/
/*:* purpose : Return one if *:*/
/*:*    name found in list,   *:*/
/*:*   otherwise return zero *:*/
/*:****************************:*/
int
FindName(head, name)
LINK *head;
NAME *name;
{
    while (IsNotNIL(head)) {
        if (!strcmp(head->name,name))
            return(1);
    }
    return(0);
}/* end of FindName */

InsertName(head,name)
LINK *head;
NAME *name;
{
    LINK *temp;

    temp=(LINK *)malloc(sizeof(LINK));
    strcpy(temp->name,name);
    temp->next = head;
    head = temp;
} /* end of InsertName */
```

<FILE: util.c>

```c
/*::*****************************
*  file: util.c
*  contents: Very low level
*      primitives.
*****************************::*/
#include "chain.h"

/*:******************************
*  function: IsNotNIL
******************************:*/
IsNotNIL(head)
LINK *head;
{
    return(head->next != NIL);
}/* end of IsNotNIL */

GetName(name)
NAME *name;
{
    scanf("%s",name);
}/* end of GetName */

PrintResults(found)
int found;
{
    if (found == 1)
        printf("Found\n");
    else
        printf("Not Found\n");
}  /* end of PrintResults */
```

<FILE: main.c>

```c
#include <stdio.h>
#include "chain.h"
LINK *chain=NIL;
NAME *name;

main(argc, argv)
int argc;
char *argv[];
{
    int found;
    name=(NAME *)malloc(LEN);
    GetName(name);
    while (strcmp(name,"quit")) {
        InsertName(chain, name);
        GetName(name);
    }
    GetName(name);
    found=FindName(chain, name);
    PrintResults(found);
}  /* end of main */
```

Example 1. A Representative, Sample Program

2.3. Building a CIA Database

The abstractor's syntax and options, where appropriate, were patterned after UNIX's C compiler *cc(1)*. To build a program database for the above program segment, a user would execute

```
% cia chain.h chain.c util.c main.c
```

% is the prompt character. Your prompt may be different from what is shown here. If CIA executes successfully, the following would appear on your screen:

```
reading from file chain.c        Exit value=0 for chain.h
reading from file chain.c        Exit value=0 for chain.c
reading from file util.c         Exit value=0 for util.c
reading from file main.c         Exit value=0 for main.c
```

In the current directory, the following files should exist: *comment.data, file.data, function.data, gbvar.data, macro.data, type.data, filefile.data, funcfunc.data, gbvrfunc.data, macrfunc.data.* They form what we term the *minimum database.* This set holds the information most commonly needed by programmers. Although they can be examined with the standard *cat(1)* or *more(1)* commands because of their ASCII storage format, use of the Information Viewer (described in Section 3) is recommended. The names and formats of the database files are subject to change.

A few options which expand CIA's database by producing additional files are available. More descriptions of the these options and the contents of each data file can be found in Sections 3-5.

3. THE INFORMATION VIEWER

The Information Viewer (INFOVIEW) provides a set of commands for interactively accessing the program database created by the C Information Abstractor. To use most of the INFOVIEW commands, a user does not have to memorize the names of object attributes. Only the objects and relations shown in Figure 2 need to be remembered.

3.1. Info — Getting Attribute Information

The *info* command displays attribute values of a program object, e.g., where it is located and what its type is. The syntax of the command is

```
info [-u] object_type object_name
```

For example,

```
% info function FindName

in_file         func_type       func_name       static bline hline eline
=============== =============== =============== ====== ===== ===== =====
chain.c         int             FindName        n      8     17    24
```

The output informs us that the integer function FindName is located in the file chain.c and is not a static function. FindName starts at line 8; its header ends at line 17; and its body ends at line 24. Definitions of a function header and function body are given in Section 3.6.

Similarly, try the following examples to see the attribute values of several other object types.

```
% info gbvar tempLink
% info macro LEN
% info type NAME
% info file chain.c
```

In order to simplify the processing of the attribute information by other application programs, an option flag **-u** is provided for producing unformatted output. Fields in this output are separated by the character **:**. For example,

```
% info -u function FindName
chain.c:int:FindName:n:8:17:24
```

If you are interested in building your own programs using the attribute information of program objects, the following shell script[2] may help you understand how the unformatted output of the *info* command can be used to construct an *editfunc* command. Readers not familiar with shell programming or the *awk(1)* program[3] should skip the rest of this sub-section.

```
#! /bin/sh
# editfunc: invoke the vi editor to edit a function
# Usage: editfunc func_name
LineAndFile='info -u function $1 |
        awk '
        BEGIN    {FS=":"}
        {printf "%s %s", $5, $1}' '
vi +$LineAndFile
```

The variable `LineAndFile` in this shell script will be evaluated at the 5th and 1st fields of the unformatted output line, i.e., the `bline` and the `in_file` fields, respectively. These two fields become arguments to the *vi(1)* editor. For instance, to edit the function `FindName`, a user would type

```
% editfunc FindName
```

This command would translate to "`vi +8 chain.c`", which automatically invokes *vi(1)* on the file `chain.c` and places the cursor at line 8. *editfunc* is similar to the combination of the UNIX's *ctags(1)* and the *tags* command of *vi(1)*. However, because of the program database, we can easily build a set of integrated software tools without rederiving information already collected.

3.2. View — Viewing Program Objects

The *view* command† is used to print out the contents of a programming object. The syntax is

†*View* should not be confused with the UNIX *view* command (usually under the directory **/usr/ucb**) which is a read-only version of *vi(1)*. To avoid invoking the UNIX *view* command, place **/usr/local/cias** before **/usr/ucb** in your path variable; see *csh(1)*.

```
view [-n] object_type object_name [file_name]
```

For example, to see the definition of the data type "struct link", simply enter

```
% view type "struct link"
struct link {
    NAME   name[LEN];
    struct link *next;
};
```

After seeing the above definition, a programmer would want to know how the type NAME is defined:

```
% view type NAME
typedef char NAME;
```

For the purpose of name matching, "struct link" is considered a single name. Line numbers can be associated with the output if the -n flag is given. The optional file_name argument of the *view* command should be provided if the same object name is shared by two objects in two different source files, e.g., when two static functions using the same name are declared in two different files.

Now try a few more examples:

```
% view function InsertName
% view gbvar chain
% view -n macro LEN
% view file chain.h
```

The *view* command provides two important functions needed during software development and maintenance:

(1) *Tracing*: By invoking a set of *view* commands, a user can easily trace through a program that references many data structures, global variables, and functions. Tedious searching of all source files is no longer required.

(2) *Cutting and Pasting*: Using *view* to retrieve program objects and redirection to concatenate them, this cutting and pasting of programs becomes easier than using traditional editors. For example, the following simple shell script file cluster can be used to concatenate several functions in a single file.

```
#! /bin/sh
# cluster: create a cluster of functions
# Usage: cluster f1 f2 ... fn
for i in $*
do
    view function $i
done
```

Suppose we want to create a file name.c that consists of the following functions: FindName, InsertName, and GetName. Enter the following:

```
% cluster GetName FindName InsertName > name.c
```

One drawback with the `cluster` command is that the newly created file `name.c` does not compile because of missing references. In the next section, we shall see how the retrieval power of *view* can be enhanced when used with the *rel* command to retrieve all related objects of a particular object.

3.3. Rel — Accessing Relations

The *rel* command shows the relations between an object and other objects. The syntax is

```
rel [-u] object_type1 object_type2 object_name1 object_name2 [r|d]
```

The relations understood by the Information Viewer are defined in Figure 2 and Table 1. There are basically two types of relations: ***definition*** relations and ***reference*** relations. If object A is defined inside object B, then there is a ***definition*** relation between object A and object B. If object A is referenced inside object B, then there is a ***reference*** relation between object A and object B.

In the case of function-function relationships, only reference relations exist. For example, enter:

```
% rel function function main -

*******************************************
reference relations: from funcfunc.data
*******************************************

caller_file         caller_func        callee_file         callee_func
=================== ================== =================== ===================
main.c              main               chain.c             FindName
main.c              main               util.c              GetName
main.c              main               chain.c             InsertName
main.c              main               util.c              PrintResults
```

The dash sign - in the command line means "don't care", i.e., any functions that are called by the function `main` should be listed. Similar to the *info* command, output can be unformatted if the -u flag is given.

Another example here shows the existence of both types of relations between the two object types *gbvar* and *file*:

```
% rel file gbvar main.c -

*******************************************
reference relations: from gbvrfunc.data
*******************************************

var_name             used_in_file         used_in_func
==================== ==================== ====================
name                 main.c               main
chain                main.c               main

*******************************************
definition relations: from gbvar.data
*******************************************

in_file         data_type            var_name        static bline eline
=============== ==================== =============== ====== ===== =====
main.c          LINK *               chain           n      4     4
main.c          NAME *               name            n      5     5
```

The reference relations show all the global variables referenced in the file main.c. The definition relations show all the global variables defined in the file main.c. Relations listed can be limited to only one type by an optional fifth argument: If r is specified, then only *reference* relations will be listed. If d is specified, then only *definition* relations will be listed.

Now try some more examples:

```
% rel file file chain.c -
% rel function function - IsNotNIL
% rel -u gbvar function - main
% rel function macro - LEN
% rel function file - chain.c
% rel file macro chain.h - d
% rel type file - chain.h
% rel gbvar type - "LINK *"
```

Recall the problem of the *cluster* command we mentioned in the previous section. When a program needs to reuse a certain function in other programs, it is necessary to retrieve not only the function itself, but also macros, types, global variables, and functions referenced by that function. Moreover, these objects may have additional references. It is like solving a *ripple effect* problem. With the *rel* and *view* commands, the task of retrieving all related objects can be greatly simplified.

3.4. List — List Contents of a Data File

Note that although all commands in previous categories require only the knowledge of the names of *object types*, the *list* command will require knowledge of the actual names of the *database files*.

The syntax of the *list* command is

```
list [-u] datafile_name
```

For instance,

```
% list type

in_file             data_type           bline eline
=================== =================== ===== =====
chain.h             NAME                4     4
chain.h             struct link         6     9
chain.h             LINK                10    10
```

Names of all database files will be found in Section 2. The output can be unformatted if the **-u** flag is given. For example,

```
% list -u macrfunc
LEN:main:main.c
NIL:IsNotNIL:util.c
```

Try these:

```
% list gbvrfunc
% list -u macrfunc
% list funcfunc
% list filefile
% list -u function
```

3.5. Structured Comments and the Comment Command

Since comments in C programs can be placed anywhere, parsers cannot be easily built to associate comments with objects. To solve this problem, we introduce the concept of *structured comments*. Three types of structured comments are recognized by CIA: file comment, pre-comment, and post-comment.

(1) **File Comment:** A file comment can be used once anywhere within each file. Normally, it is placed at the beginning of a file and follows the following syntax:

```
/*::      comments      ::*/
```

As long as the two colons appear immediately after the asterisk, CIA will consider the comment as a file comment. This comment should not be used to give in-depth explanations of each function and variable defined, rather it should explain what group of functions and variables are defined within the file. The file comment was intended to be a place where programmers could explain characteristics particular to entire files.

(2) **Pre-comment:** Pre-comments are placed before a software object and they follow the following syntax:

```
/*:*      comments      *:*/
```

Again, only the symbols **:*** and ***:** distinguish the pre-comment.

(3) **Post-comment:** Post-comments are normally placed after a software object and they follow the following syntax:

```
/**:      comments      :**/
```

Table 2 lists the comment types and with what each can associate. While global variables and types can have pre- and post-comments attached to themselves, functions can only be associated with pre-comments. Furthermore, pre-comments must appear completely before the object to which they are attached. In the case of functions and global variables, this means that it must be placed before the type. Analogously, post-comments must appear after the software objects, i.e., semi-colons for global variables and type declarations and closing curly braces for functions.

Table 2. Comment Associations with Software Objects

num	*object*	*file comment*	*pre-comment*	*post-comment*
1	file	X		
2	function		X	
3	global variable		X	X
4	macro			
5	type		X	X

The special syntax of the three comment types must be conformed to. Although CIA only needs to recognize the first four characters to determine which type it is looking at, a warning is issued if the proper ending is not used. In any case, the first occurrence of a */ ends the comment.

Several structured comments can be grouped together and associated with a single software object. Blank lines or normal C comments can appear in between the comments, but no code can be interspersed within the structured comments. For example,

```
/*:******************************:*/
/*     function: GetName           */
/*:******************************:*/

/*:    more comments here        :*/
GetName()
{
   ....
}
```

INFOVIEW provides a command *comment* for viewing comments. The syntax is similar to that of the *view* command:

```
comment [-n] object_type object_name [file_name]
```

For instance,

```
% comment function FindName
/*:***************************:*/
/*:* function: FindName      *:*/
/*:* purpose : Return one if *:*/
/*:*   name found in list,   *:*/
/*:*   otherwise return zero *:*/
/*:***************************:*/
```

Here is another example:

```
% comment file util.c
/*::***************************
*  file: util.c
*  contents: Very low level
*      primitives.
***************************::*/
```

3.6. Function-related Commands

Functions are probably the most important objects in the program database. Several commands are solely created for dealing with function objects:

(1) *header*: print out the header of a function. A header is defined as the comments (if any) and the formal parameter definitions. The syntax of *header* Is

```
header [-n] function_name [file_name]
```

For example,

```
% header FindName
/*:***************************:*/
/*:* function: FindName      *:*/
/*:* purpose : Return one if *:*/
/*:*   name found in list,   *:*/
/*:*   otherwise return zero *:*/
/*:***************************:*/
int
FindName(head, name)
LINK *head;
NAME *name;
```

(2) *body*: print out the body of a function. The syntax is similar to that of *header*:

```
body [-n] function_name [file_name]
```

For example,

```
% body -n InsertName
29 : {
30 :       LINK *temp;
31 :
32 :       temp=(LINK *)malloc(sizeof(LINK));
33 :       strcpy(temp->name,name);
34 :       temp->next = head;
35 :       head = temp;
36 : }  /* end of InsertName */
```

The next three commands: *formals, listcall,* and *viewcall* should only be executed when the database is constructed with the **-f** option of CIA. The **-f** option prompts our system to collect two more pieces of information:

a) Formal parameters of each function definition — stored in *formals.data.*

b) Real parameters given for each function call — stored in *funcccall.data.*

This option can increase the disk block usage by a significant amount†. Exercise caution when issuing it.

To test the next three commands, first create a new database with the formal parameter and function call information:

```
% cia -f chain.h chain.c util.c main.c
```

In addition to the *minimum database*, you should also find *formals.data* and *funccall.data* in your current directory.

(3) *formals*: print out the formal parameters of a function in a nice form. The syntax is

```
formals [-u] function_name [file_name]
```

For example,

```
% formals InsertName

file_name     :  chain.c
function_name :  InsertName

arg_type                        arg_name
=============================   =======================================
LINK *                          head
NAME *                          name
```

The output of the *formals* command will be unformatted if the option flag **-u** is given.

(4) *listcall*: list all the instances of function calls made to a particular function. The syntax is

† This option was implemented to help support a structure chart generator.

```
listcall [-n] callee [file] [caller]
```

This command helps locate and count function calls. For example, to print out all the instances of the function call GetName, enter

```
% listcall GetName

file                    caller              callee              bline eline
======================= =================== =================== ===== =====
main.c                  main                GetName             12    12
main.c                  main                GetName             15    15
main.c                  main                GetName             17    17
```

The listing can be limited to function calls in a particular file or even in a particular function. For example,

```
% listcall IsNotNIL chain.c FindName

file                    caller              callee              bline eline
======================= =================== =================== ===== =====
chain.c                 FindName            IsNotNIL            19    19
```

(5) *viewcall*: print out the actual statements that invoke a particular function call. The syntax is similar to that of listcall:

```
viewcall [-n] callee [file] [caller]
```

For example,

```
% viewcall -n InsertName
14 :            InsertName(chain,name);
```

(6) *summary*: print out the summary of a function. The summary includes

(1) function attributes

(2) function header

(3) global variables and macros referenced in the function

(4) functions that call this function

(5) functions that are called by this function

This is our first attempt at automatic generation of design documents from the program database. The document can be used to check against the original paper design (if there was any).

The syntax of this command is

```
summary function_name [file_name]
```

The output generated by *summary* is usually very long; we have to leave out an example here because of the space limitation.

3.7. Miscellaneous Commands

In this section, we list a few other INFOVIEW commands that do not fall into the previous categories:

(1) *see*: print out a section of a source file specified by the beginning and ending line numbers. The syntax is

```
see [-n] filename num1 num2
```

For example,

```
% see -n chain.h 6 10
6 : struct link {
7 :     NAME   name[LEN];
8 :     struct link *next;
9 : };
10 : typedef struct link LINK;
```

(4) *schema*: print out the database schema. The syntax is

```
schema [datafile_name]
```

For example,

```
% schema gbvar

in_file         data_type            var_name        static bline eline
=============== ==================== =============== ====== ===== =====

% schema funcfunc

caller_file        caller_func        callee_file        callee_func
================== ================== ================== ==================
```

If the datafile name is not given, then the whole database schema will be printed.

(5) *retrieve*: retrieve database entries that satisfy a set of field specifications. The syntax is

```
retrieve rel_name field1=value1 ... fieldn=valuen
```

For example,

```
% retrieve function bline=8

in_file         func_type       func_name       static bline hline eline
=============== =============== =============== ====== ===== ===== =====
chain.c         int             FindName        n      8     17    24
util.c          int             IsNotNIL        n      8     12    15
% retrieve gbvar static=y

in_file         data_type            var_name        static bline eline
=============== ==================== =============== ====== ===== =====
chain.h         LINK *               tempLink        y      12    12
```

Knowledge of field names, which can be obtained through the *schema* command, is required in order to use the *retrieve* command. Note that all commands in other categories do not require any knowledge of the field names, and recall that the field names are subject to change in the future.

(6) *infoview*: print out the command syntax of all INFOVIEW commands. A summary of all INFOVIEW commands can be found in Appendix C.

3.8. Prefix Matching

All object types can be specified by a proper prefix when invoking INFOVIEW commands. The minimum prefixs necessary for each object type follow: `fi` for file, `fu` for function, `gbv` for gbvar, `t` for type, and `ma` for macro. For example,

```
% info gbvar see                              can be replaced by
% info gbv see

% retrieve function static=y                  can be replaced by
% retrieve fu static=y
```

For specifying database files that store relations, the following abbreviations can be used: `fifi` for filefile, `fufu` for funcfunc, `gf` for gbvrfunc, and `mf` for macrfunc. For example,

```
% list filefile                               can be replaced by
% list fifi

% retrieve gbvrfunc used_in_file=do_info.c    can be replaced by
% retrieve gf used_in_file=do_info.c
```

3.9. Environment Variables

A programmer may want to switch back and forth between several program databases; therefore, a set of environment variables is provided for locating program files or database files. If none of the environment variables are set, the default is to search files in the current directory.

Suppose your program and database files are located in the directory *project*, and the current directory is not *project*. Before you invoke any INFOVIEW commands, set the environment variable *ciadir* to *project*. If you are using the C shell, type

```
% setenv ciadir project
```

If you are using the Bourne shell[2], type

```
% ciadir=project
```

In all later examples, we will assume that you are using the C shell.

Two additional environment variables, *datadir* and *pgmdir*, are provided for the separation of the program database and source files. *Datadir* and *pgmdir*, when either or both are set, overwrite the environment variable *ciadir*. Table 3 shows how the settings of environment variables affect the searching of database or program files.

Table 3. Locating program database and source files

environmental variables			**files located in**	
ciadir	*pgmdir*	*datadir*	*program*	*database*
-	-	-	current	current
-	-	B	current	B
-	A	-	A	current
-	A	B	A	B
C	-	-	C	C
C	-	B	C	B
C	A	-	A	C
C	A	B	A	B

For example, suppose the following commands are executed:

```
% pwd
/b/os/user1/project
% mkdir ciabase
% mv *.data ciabase
% setenv datadir ciabase
```

then all database accesses will go to */b/os/user1/project/ciabase* until the environment variable *datadir* is changed or unset. Source files are still assumed to be located in the current directory.

Users may prefer to separate the program database and source files for several reasons:

(1) Simply to avoid a crowd of files in the source directory.

(2) To maintain different subsets of the program database by running the C abstractor on different subsets of the source files. This is particularly useful when some related programs that share source files are located in the same directory. For example, suppose program p1 is constructed from lib1.c, lib2.c, and p1.c, and program p2 is constructed from lib1.c, lib2.c, and p2.c. Two databases can be created in the following way (-c option is explained in Section 4.1):

```
% cia -c p1.c p2.c lib1.c lib2.c
% cia p1.cst lib1.cst lib2.cst
% mkdir p1.base
% mv *.data p1.base
% cia p2.cst lib1.cst lib2.cst
% mkdir p2.base
% mv *.data p2.base
```

(3) To maintain multiple versions of the program database so as to keep a history of the structural changes in a program. Only attribute information and relational information can be accessed in old versions of databases. Source access to old versions using the *view* command would be difficult until some facilities that integrate the SCCS files[4] and the program database become available.

To find out the current settings of the environment variables used by the INFOVIEW System, use the command *pgmenv*. For example,

```
% pgmenv
ciadir= /b/os/user1/project
pgmdir=
datadir= /b/os/user1/project/ciabase
```

3.10. Initialization File

Before CIA starts processing C programs, it reads in an initialization file **.ciarc** from the current directory. This file can contain parameters to control the abstraction of C programs, which use special libraries or require different databases. As a result, directories can be personalized to reflect the special needs of the programs within.

One of CIA's options **-I** allows users to specify additional paths when searching for include files. See Section 4.3 for a full explanation. If source files within one directory depend on include files in another directory such that the **-I...** option is required with each CIA execution, **.ciarc** has a mechanism to prevent the repetitive typing. For example, say we would normally execute:

```
% cia -I/b/os/miken/cia/lib -I/b/os/miken/lib chain.h chain.c util.c main.c
```

We could, instead, place **.ciarc** in the current directory with the following contents:

```
incdir=/cia/lib
incdir=/b/os/miken/lib
```

Now, to build a database, we would simply type

```
% cia chain.h chain.c util.c main.c
```

The general include file search path becomes

```
(1) Current directory
(2) First added directory via -I... option
(3) Second added directory via -I... option
        .
        .
        .
(I) First added directory via .ciarc file
(J) Second added directory via .ciarc file
        .
        .
        .
(N) /usr/include directory
```

Search start positions are identical to those explained in Section 4.3.

The three environment variables used by the INFOVIEW system can also be placed in the **.ciarc** file. Individuals will then be able to view different programs in different directories without resetting the variables. The syntax is identical to that of **incdir**, e.g.,

```
ciadir=/b/os/miken/ciasource
datadir=/b/os/miken/ciasource/data
pgmdir=/b/os/miken/ciasource
```

Pointers in the initialization file take precedence by replacing their respective environment variable.

3.11. Redirection of the Database

Instead of explicitly moving database files into a directory, you can issue a CIA option **-d***Directory*. Using the second major example of Section 3.9, it becomes

```
% cia -c p1.c p2.c lib1.c lib2.c
% mkdir p1.base
% cia -dp1.base p1.cst lib1.cst lib2.cst
% mkdir p2.base
% cia -dp2.base p2.cst lib1.cst lib2.cst
```

3.12. INFOVIEW Library Routines

The C program database provides an excellent opportunity for developing software tools that utilize the valuable attribute and relational information. In order to facilitate the construction of these commands, a set of library functions is provided as an interface to the C program database. For details of these functions, see *INFOVIEW(3)*. A short summary is given in Appendix D.

4. PROGRAM DEVELOPMENT

Developed originally as a maintenance tool, CIA became a more complete programming tool when modifications were made enabling it to be used in program development. This section describes incremental abstraction and gives an example of a makefile[5] that uses incremental abstraction for more efficient maintenance of large program databases.

4.1. Incremental Abstraction

Analogous to a C compiler's incremental compilation, the C Information Abstractor has a method of incremental abstraction used to lower overall abstraction time when frequent changes are made to source. Unlike C compilers, however, CIA can abstract include files, creating a few dissimilarities between the two.

Intermediate abstraction files in CIA lingo are called symbol table files and take the form `*.hst` and `*.cst`. To create them using the example of Section 2.2,

```
% cia -c chain.h chain.c util.c main.c
```

Upon successful completion, `chain.hst`, `chain.cst`, `util.cst`, and `main.cst` should exist in the current directory. To link the *minimum database* from these files, a user would type

```
% cia chain.hst chain.cst util.cst main.cst
```

As a result,

```
% cia chain.h chain.c util.c main.c                <method 1>
```

produces the same database as

```
% cia -c chain.h chain.c util.c main.c             <method 2>
% cia chain.hst chain.cst util.cst main.cst
```

except the last method will reabstract faster than the first. For example, say we made a change to the file chain.c. To recreate the database using method one, we would execute

```
% cia chain.h chain.c util.c main.c
```

exactly what we did earlier; however, by using method two

```
% cia -c chain.c
% cia chain.hst chain.cst util.cst main.cst
```

The time to completion is considerably less than method one because chain.h, util.c and main.c are not reabstracted in the later example. Typically, abstraction time for a symbol table is equal to 50% of the compilation time for that same source file.

IMPORTANT: When using incremental abstraction and options other than -c concurrently, careful attention must be paid to the options chosen at symbol table creation time and at database creation time. Abstraction ranges can be summarized as follows:

1) If the symbol tables are created with a specific option and that option is not given in the linking phase, the additional information abstracted at the creation phase is not put into the database.

2) If the symbol tables are created without a specific option, the extra information that would be abstracted by including the option is lost. As a result, if that option is given at link time, the database will not contain any information associated with that option.

Because adding certain options at abstraction time can lead to enormous symbol table files, our approach was to not include all information as the default. Likewise, the use of certain options at the link phase can increase the size of the final database by up to two times. In order to keep storage requirements to a minimum, options must be carried through to all stages of development. For example, to collect function parameter data, use

```
% cia -c -f chain.h util.c chain.c
% cia -f chain.hst util.cst chain.cst
```

If the **-f** option is forgotten on either line, function parameter information is not collected.

4.2. Make and Makefiles

Experienced programmers will notice that the problem of insuring database integrity during frequent source changes can be easily remedied through makefiles. Those programmers unfamiliar with *Make* and makefiles should read the Make manual[5]. Figure 3 shows a subset of the makefile used for the Network Event Manager, a current project at UC Berkeley. It demonstrates the use of incremental abstraction in a makefile to maintain a program database.

```
# Example for maintenance of programs
# Makefile for Network Event Manager

SRC = em.c event.c exec.c status.c socket.c
HDR = em.h
OBJ = em.o event.o exec.o status.o socket.o

CST = em.cst event.cst exec.cst status.cst socket.cst
HST = em.hst

GET = sccs get
CIAFLAGS = -f

.SUFFIXES: .c .o .cst .hst

em: ${OBJ}
     cc ${CFLAGS} -o em ${OBJ}

${OBJ}: em.h

.c.o:
     cc ${CFLAGS} -c $<

# CIA section

.c.cst:
     cia -c ${CIAFLAGS} $<

.h.hst:
     cia -c ${CIAFLAGS} $<

ciabase: ${HST} ${CST}
     cia ${CIAFLAGS} ${HST} ${CST}

# SCCS section

sources: ${SRCS}

${SRCS}:
     ${GET} $@
```

Example 2. A Sample Makefile

The source files are listed after the macro SRC; the header files are listed after HDR; and the corresponding symbol table files are listed after CST and HST. To build or rebuild the Event Manager's database, a user needs just input

```
% make ciabase
```

After HST and CST are expanded, the dependencies for this command become `em.hst`, `em.cst`, `event.cst`, `exec.cst`, `status.cst`, and `socket.cst`. Let's say that none of the symbol table files exist. *Make* would then try and create them by executing:

```
% cia -c -f em.h
% cia -c -f em.c
% cia -c -f event.c
% cia -c -f exec.c
% cia -c -f status.c
% cia -c -f socket.c
```

Make knows to execute these commands because of the dependencies

```
.c.cst:       and       .h.hst:
```

For example, `em.hst` is dependent on `em.h`; to create `em.hst`, it must execute

```
% cia -c -f em.h
```

Now that **ciabase**'s dependencies are met, *Make* automatically issues the command

```
% cia em.hst em.cst event.cst exec.cst status.cst socket.cst
```

This performs what *Make* thinks will create the file **ciabase**; however, ciabase is only conceptually created through the database files. Once the UNIX prompt returns, your database is made; and using INFOVIEW, you can examine your program in a style you have never before experienced. Also, remember that when a change is made to the source, just type

```
% make ciabase
```

4.3. Preprocessor Options

Through options, the C compiler gives a certain amount of control over a compilation to the user invoking the compilation. These options allow users to define and undefine macros as well as give additional paths when searching for include files. CIA, too, has these options so that, according to the command line options given, its database entries will correspond to a correct interpretation of its input. Any macros defined or undefined with the command line, however, will not be entered into CIA's relational database. To define a macro, a user would invoke CIA as follows:

```
% cia -Dmacro=xxx chain.c
```

The effect would be identical to putting the following statement on line 0 of `chain.c`.

```
#define macro xxx
```

-D*macro* is short for **-D***macro=1* and can be used when a macro only needs to be defined as with conditional compilation; undefining macros can be performed through a **-U***macro* option. Deciding to use the **-D**... and **-U**... options should not be difficult. If a program is compiled as follows,

```
% cc -Dunix chain.h util.c chain.c main.c
```

then its database should be built similarly:

```
% cia -Dunix chain.h util.c chain.c main.c
```

An include file search path is used during the CIA's preprocessing stage. Normally, users have no need to learn about the path because they will have only two types of include files: those located relative to the current directory and those located relative to the /usr/include directory. If a programmer wishes to use include files which don't fall into either of the previous two categories, he can add to the search path with the -I*directory* option. The search path becomes:

```
(1) Current directory
(2) First added directory via -I... option
(3) Second added directory via -I... option
     .
     .
     .
(N) /usr/include directory
```

Up to seven or eight directories can be added to the search path in the location shown above. If double quotes surround the include file name in the source, the search begins at position one, the current directory, e.g.,

```
#include "cia.h"
```

However, if angled brackets are used, the search begins at position two. Suppose two directories are added to the search path as follows:

```
% cia -I/b/os/miken/lib -I/b/os/miken/local chain.h util.c chain.c
```

If the following then appears in the source file chain.h,

```
#include <param.h>
```

the *cia* program will search the file param.h starting from the directory /b/os/miken/lib.

5. STATUS AND FUTURE WORK

The C Information Abstraction System has been installed and in production use at UC Berkeley and Columbus Bell Laboratories. The system has proved extremely helpful in supporting the development and maintenance of our own projects. Ivan Brohard and Bruce Wachlin at Columbus Bell Laboratories implemented the GRIST system, which provides a multi-window graphical interface for software development and maintenance based on the C program database created by the C Information Abstractor. Our future work will focus on an integrated set of software tools that exploit the information stored in the program database, e.g., to support very high level operations such as calculation of software metrics, analysis of program structure, automatic program restructuring, analysis of ripple effects, generation of design documents, etc.

Acknowledgement

Many people contributed to the implementation of the C Information Abstraction System. Michael Nishimoto implemented the C Abstractor with help from Lenora Eng. Joo-Seok Song wrote some early shell scripts for the INFOVIEW system. Wen-Ling Chen implemented most of the final INFOVIEW system in C programs with help from Yih-Farn Chen. We would also like to thank Professor Ramammoorthy, our colleagues, Atul Prakash, Benjamin Chang, Wei-Tek Tsai, Yutaka Usuda, Vijay Garg, Tsuneo Yamaura, Anupam Bhide, Rajeev Aggarwal, and the technical staff at Columbus Bell Labs, in particular Charles Fritz, Bruce Wachlin, Ivan Brohard, Doyt Perry, and Michael Buckley for valuable comments and discussion throughout the development of this project. Special thanks are due Andy Guest for his help during the preparation of this tutorial.

References

1. Y. F. Chen and C. V. Ramamoorthy, "The C Information Abstractor," *Computer Software and Applications Conference (COMPSAC)*, Chicago, October 1986.
2. S. R. Bourne, "An Introduction to the UNIX Shell," *UNIX Programmer's Manual*, vol. 2, 1978.
3. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, "Awk - A Pattern Scanning and Processing Language," *Unix Programmer's Manual*, vol. 2, 1984.
4. M.J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364-370, December 1975.
5. S. Feldman, "Make - A program for Maintaining Computer Programs," *Unix Programmer's Manual, 4.2 BSD*, 1978.

Appendix A — Summary of CIA Options

USER Options: c, d, e, f, i, m, s, u, w, x, D, I, U

-c	Incremental abstraction. See Section 4.1.
-d	Directory for CIA database. See Section 3.11.
-e	No external function information. Some programs make numerous calls to library/system functions. If the external definitions are not placed within an include file and instead are placed within each module calling that function, *funcfunc.data* may grow large. See **Function to Function Relation** in Appendix B. To curtail the size of *funcfunc.data,* we don't output relations involving functions which are defined externally and which don't appear in any of CIA's input files.
-f	Collect formal parameter & function call information. See Section 3.6.
-i	Collect information in include files. When variables or functions are defined within an include file, non-relational information is not abstracted for them. This option will cause that data to be collected. Be aware that data files can increase tremendously depending on your style of programming.
-m	Print out a menu of CIA options
-s†	Collect information on system function calls. The data file produced is *system.data.*
-u†	Collect unknown identifiers within functions. The data file produced is *unknown.data.*
-w	Suppress normal [stdout] messages
-x†	Collect macro expansion information. The data file produced is *expand.data,* and it will grow very large.
-D	Define a macro — same format as *cc(1)* See Section 4.3
-I	Add directory to include search path — same format as *cc(1)* See Section 4.3
-U	Undefine a macro — same format *cc(1)* See Section 4.3

ADMINISTRATION Options - h, l, t

-h	Print hash table values
-l	Print low level debug information-show state changes
-t	Print tokens, identifiers, EOF, ...

†Currently, there is no sophisticated interface to this file.

Appendix B — Detailed Description of Information in the Database

The various relationships between the five software objects were illustrated in Figure 2. This appendix will explain the objects and their relationships in more depth while listing, after each heading, the data files from which specific information was derived.

Files — file.data

The CIA system collects three pieces of information related specifically to files; they are file comment, file length, and compile-time file length. Information on file comments is placed in *comment.data* and has been previously discussed in Section 3.5. The meaning of file length should be obvious; however, the concept of a compile-time file length may be a new. Compile-time length more accurately reflects than normal file length how much time an abstraction or compilation will take on a given file.

In the process of writing large C programs, source code is broken down into modules and include files. When C compilers and our C abstractor begin parsing module files, they expect a pre-processor, */lib/cpp in BSD UNIX*, to have already passed over the file. The pre-processor will process all the compiler control lines, those beginning with a #. One operation of *cpp* is to actually merge in include files, making the module file parsed by CIA much larger. The resulting file length is the compile-time length of a file.

Functions — function.data, formals.data, and funccall.data

Functions, being the cornerstone of structured programming, are the most informative software objects available. CIA provides their type, location, length, staticness (all in *function.data*), formal parameters (*formals.data*), parameters of each call (*funccall.data*), and comments. Header and body breakdowns are also collected. A function header is defined as the comments (if any) and the formal parameter definitions. Although comment information is stored implicitly through the function header (*function.data*), explicit data is kept in *comment.data*.

Global Variables — gbvar.data

Type, definition location, length, and staticness are the attributes abstracted by CIA for global variables. Comments can be placed both before and after definitions, and INFOVIEW's *view* command will display both if both are used. Unlike function definitions, however, global variable definitions can be ambiguous. 4.3 BSD's C compiler, for instance, allows multiple definitions of a variable within a program although not within a single file.

CIA takes the stance of *define once, declare many.* The difference between definition and declaration is illustrated in the following code segment:

```
#include <stdio.h>
extern CHAR *symbol_table[];
int index;
static long bit_mask;

.
.
```

In the above code segment, the structure `symbol_table` would be considered a declaration while `index` and `bit_mask` would be definitions.

CIA depends on programmers to define variables only once and properly declare them elsewhere. If multiple definitions are used, CIA will abstract them all, but INFOVIEW will only display the first occurrence within the database.

Macros — macro.data

Only two pieces of information are stored for macros: definition location and expansion locations. DO NOT use structured comments with macros; CIA may not understand your program. Macros cannot have structured comments associated with them because of the manner in which macro expansions are resolved by the pre-processor. We decided to not modify the pre-processor due to the slow down it would incur. If structured comments are attached to macros, CIA aborts from the file it is currently abstracting or output false data. Expansion information (stored in *expand.data*) will not be abstracted unless the **-x** option is given. Moreover, the extra data file produced may be enormous. See Appendix A.

Types — type.data

Definition location and attached comments are abstracted from types. Like global variables, types can have both pre and post-comments. Presently, we don't parse the contents of type definitions, so that information is not available. See **Type to Type Relation.**

File to File Relation — filefile.data

Often times an individual executes

```
% grep "#include" source.c
```

to find out which files are included in some source file or include file. He may also type

```
% grep include.h *.c
```

This prints files which include a particular include file. With CIA both these commands become unnecessary. *filefile.data* contains the relations between include files and source files, and *rel* can display the data quickly.

Function to Function Relation — funcfunc.data

In the absence of good comments, this relationship gives more useful information about a program than any other. With it, a program's entire call structure can be recreated. The C Abstractor finds the functions which are called by each function and the functions which call a certain function.

Three different variations of called functions are abstracted by CIA. They are

1) Functions defined by the user. [e.g., main()]
2) Externally defined functions. [e.g., extern int HashValue();, FILE *fopen();]
3) Library or system functions: functions not defined or declared. [e.g., strcpy()]

CIA needs to find the file name where the called function is defined to complete this relation. In the first case, the file name where the programmer defines the function is obviously used. For the second case, a slightly different algorithm must be utilized. If these functions are not defined anywhere, CIA indicates that the definition file is where it found the initial external definition or declaration. *It would be to your advantage to put external definitions in include files unless they are used only once. Otherwise, your program may appear to have multiple definitions due to the algorithm explained above.* For the last type, we do not abstract a pure function to function relation; however, we can create an extra database file (*system.data*) which holds the library/system function calls. Presently, declared library/system functions are treated as the second type of function. Many standard I/O functions, like fopen(), are declared within *<stdio.h>* and are handled as such. Note that library/system function calls will presently produce entries within *funcfunc.data* which are not consistent with *function.data*. For instance, the database for

```
<FILE: p1.c>
main()
{
    FILE *fp;
    fp=fopen("data","r");
}
```

would contain the following entry in *funcfunc.data*:

```
p1.c:main:/usr/include/stdio.h:fopen
```

However, the function `fopen` is defined nowhere.

Global Variable to Function Relation — gbvrfunc.data

This relation is very similar to the previous one. CIA abstracts a list of all global variables each function uses and a list of functions which use a particular variable.

Two types of global variables are recognized:

1) Those defined by the user.
2) Those variables defined externally.

As explained earlier, this may be a point of confusion since C compilers may handle definitions and declarations differently. We take the stance of *define once, declare many.* See the **Global Variable** object section in this appendix. A variable declaration must have an `extern` keyword in the statement. Like the function to function relation, if CIA can find no definition, it will report the variable as being defined where it was declared. This situation should not occur as frequently, though, because C has few system variables.

Macro to Function Relation — macrfunc.data

We abstract all macros each function uses and all functions which use certain macros. If a macro is substituted by other macro(s), then the original macro and substituting macro(s) will all be abstracted.

Function to File Relation — from function.data and funcfunc.data

The next four relations are built a little differently than those already explained. Up to this point, this appendix has discussed how one software object references another object. In the function to file relation, the C Information Abstraction System provides those functions referenced or defined in each file and those files which reference or define particular functions. With static functions, multiple function definitions can exist. This relation is not abstracted explicitly by the core abstractor CIA; instead, it is provided by INFOVIEW. Using two database files, INFOVIEW can create this relation through exhaustive search.

Macro to File Relation — from macro.data and macrfunc.data

Macros referenced or defined within a file and all files referencing or defining a macro are collected.

Global Variable to File Relation — from gbvar.data and gbvrfunc.data

CIA abstracts global variables referenced or defined within a file and files referencing or defining particular global variables.

Type to File Relation — from type.data

The type to file relation is not completed at this time. We can presently offer data showing which types are defined in which files but not which types are referenced in each file.

Global Variable to Type Relation — from gbvar.data

Global variables cannot reference types, so this relation gives just those variables defined as certain types.

Type to Function Relation [Future Extension] — typefunc.data

Types used within each function and functions which use particular types will be abstracted. This relation is a future extension of CIA we hope to implement. If it works in the version you have, use it; otherwise, send mail to cia@ucbarpa.Berkeley.EDU for the latest details. At this moment, we are uncertain whether this data will be output as a default or another option.

Type to Type Relation [Future Extension] — typetype.data

Finally, we offer you a type to type relation for quick understanding of complex data structures. The syntax for any future extensions should be analogous to existing ones.

Appendix C — Summary of INFOVIEW Commands

info [-u] object_type object_name
rel [-u] object_type1 object_type2 object_name1 object_name2 [r|d]
view [-n] object_type object_name [file_name]
comment object_type object_name [file_name]
list [-u] datafile_name
header function_name [file_name]
body function_name [file_name]
formals [-u] function_name [file_name]
listcall [-u] callee [file] [caller]
viewcall [-n] callee [file] [caller]
retrieve rel_name field1 value1 field2 value2 ...
summary function_name [file_name]
see [-n] file_name begin_line end_line
schema [rel_name]
pgmenv
infoview

Note: The following environment variables should be set properly:

pgmdir: the pathname of the program directory
datadir: the pathname of the data directory
ciadir: the default pathname when **pgmdir** or **datadir** is not set

If none of the above variables is set, then the current directory is searched for data files or program files.

Appendix D — Summary of INFOVIEW Library Routines

(1) *formals*: obtain information about a function's formal parameters.

```
formals(func_name, file_name, buffer)
char *func_name;
char *file_name;
char *buffer;
```

(2) *getfields*: get data fields from a data line.

```
getfields (line, field)
char line[MAXLINE];
char field[NUMFIELDS][MAXLINE];
```

(3) *header*: fill a buffer with the header of a function.

```
header(function_name, file_name, buffer)
char *function_name;
char *file_name;
char *buffer;
```

(4) *info*: obtain information about an object's attributes.

```
info(obj_type, obj_name, buffer)
char *obj_type;
char *obj_name;
char *buffer;
```

(5) *interpret*: interpret an error code.

```
interpret(error_code)
int error_code;
```

(6) *match*: match a line with a table that has pairs of field numbers and field values.

```
match(line, num, rel_table)
char line[MAXLINE];
int num;
struct rel_tab rel_table[MAXFIELDS];
```

(7) *open_source*: open a file of a particular type in a specified directory.

```
FILE *open_source(openfile, filetype)
char *openfile;
int filetype;
```

(8) *prefix*: transform the prefix of a relation name to its full name.

```
prefix(rel_name)
char *rel_name;
```

(9) *rel*: obtain the relational information between two object types.

```
rel(obj_type1, obj_type2, obj_name1, obj_name2, ref_def, buffer)
char *obj_type1;
char *obj_type2;
char *obj_name1;
char *obj_name2;
int ref_def;
char *buffer;
```

(10) *see*: fill a buffer with a part of the specified program file.

```
see(filename, begin_line, end_line, numflag, buffer)
char *filename;
int begin_line;
int end_line;
int numflag;
char *buffer;
```

(11) *view*: print out a specified object.

```
view(obj_type, obj_name, file_name, numflag, buffer)
char obj_type[MAXNAME];
char obj_name[MAXNAME];
char file_name[MAXNAME];
int numflag;
char *buffer;
```