SLAC-PUB-5930
September 1992
(E/I)

# The Cheetah Data Management System

**Paul F. Kunz**

**Stanford Linear Accelerator Center**
**Stanford University, Stanford, CA 94309, USA**

**Gary B. Word**

**Department of Physics and Astronomy**
**Rutgers University, Piscataway, NJ 08855-0849, USA**

Cheetah is a data management system based on the C programming language, with support for other languages. Its main goal is to transfer data between memory and I/O streams in a general way. The streams are either associated with disk files or are network data streams. Cheetah provides optional convenience functions to assist in the management of C structures. Cheetah streams are self-describing so that general purpose applications can fully understand an incoming stream. This information can be used to display the data in an incoming stream to the user of an interactive general application, complete with variable names and optional comments.

## 1. The Goal of the Cheetah System

The basic entities managed by Cheetah are structures as defined in the C programming language. To quote the introductory paragraph in the chapter on structures in *The C Programming Language* by Kernighan and Ritchie[1].

*A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called "records" in some languages, notably Pascal.) [...] Structures help to organize complicated data, particularly in large programs, because in many situations they permit a group of related variables to be treated as a unit instead of as separate entities.*

The design of Cheetah is based on the premise that the structures of the C language are ideally suited to organize data within high energy physics (HEP) programs, such as Monte Carlo simulation programs and data acquisition systems. C structures are clearly the equivalent of the "banks" referred to by many HEP developed data management systems based on FORTRAN. These FORTRAN-based systems all suffer, to a greater or lesser degree, in poor integration with the compiler, the operating system, and/or the symbolic debugger. The goal of the Cheetah system is to exploit the features of the C language while adding data management tools such as input and output streams to storage or to a client-server network connection. Cheetah is designed to allow the code writer to use native C language constructs, with only a few required Cheetah function calls and a number of optional Cheetah *convenience* functions. Cheetah maintains symbol table information about the user's data, which is used to provide machine-independent and self-describing data streams. Cheetah also provides support for data structures of other languages such as C++ and FORTRAN.

## 2. Introduction to C Structures

In this section, a short and simplified introduction to C structures is presented. Consider the following C structure

* Work supported by Department of Energy, contract DE-AC03-76SF00515.
*Contributed paper to the Conference on Computing in High Energy Physics, Annecy, France, September 21-25, 1992.*

definition which might be used to describe a track in a Monte Carlo program:

```
struct track {
    float p[3];
    float ptot;
    int charge;
};
```

The keyword *struct* introduces a structure declaration which is a list of declarations enclosed in braces. The symbol *track*, called the structure tag, becomes a new data type in the C language. The standard C declarations within the braces name the variables that are members of a structure of type *track*. Members of a C structure can be of any C data type including pointers to other structures.

The structure declaration does not allocate any memory storage. Rather storage is allocated with a statement like

```
    struct track mytrack;
```

More frequently, one declares a variable which is a pointer to storage which is allocated at execution time by calling the C library function *malloc*. A code fragment that does this looks like:

```
struct track *mytrack;
mytrack = malloc( sizeof(struct track ) );
```

where *sizeof* is a compiler operator which in this case returns the number of bytes needed for the track structure.

In many cases, one handles a collection of structures of the same type. There are many ways to handle this situation in C, such as a linked list or an array of structures. One convenient way is to allocate each structure individually as needed and to allocate an array of pointers to these structures. In the following code fragment:

```
struct track **tracks;
tracks = malloc(numtrks*(sizeof(struct track *)))
tracks[i] = malloc( sizeof(struct track) );
```

*numtrks* is a variable containing the number of pointers desired and *tracks[i]* is a pointer to the *i*-th structure. These statements may look complex to the beginning C programmer, but they are commonly used and with more experience they become quite familiar. One should also note that compared to FORTRAN-based data management systems, these few lines replace a very large amount of complex code that is not any easier, and frequently much harder, to understand. This method has the advantage of making more efficient use of virtual memory space by *not* requiring that the structures be contiguous in memory.

Having allocated storage for structures in this way, one can access members of the structure as illustrated in the following code fragment:

```
ptot = tracks[i]->ptot;
```

The string *tracks[i]–>ptot* can be used in an expression anywhere a variable can be used. That is, it can appear either on the left or right side of the equal sign, used as an argument of a function call, *etc.* Thus, one has clean and concise mnemonic access to data in the structures. An important advantage, compared to FORTRAN-based systems, is that the symbolic debugger can be used very effectively since the syntax is a part of the C language.

## 3. The Cheetah List

It is usually the case that a program will contain many different structure types and for each type many structures of that type. For example, a program may want to manipulate a list of tracks, a list of vertices and a list of calorimeter clusters. For ease of handling, the collection of lists may themselves be collected into a list. Lists are therefore important entities. Cheetah defines a list structure called a *chList* which one can use for data management and that Cheetah uses to manage input and output streams.

A *chList* is created by calling *chNew* with a list name, a comment string, and information on the type of data being maintained in the list. For example, in the following code fragment:

```
chList *trackList;
trackList = chNew("Tracks","MCtracks",track_T());
```

*trackList* is declared to be a pointer to a Cheetah list which itself is created by the *chNew* function call. The last argument, *track_T()*, is a function that returns the type information. It and a utility to create it automatically will be described in the next section.

A structure is added to a *chList* by using the function *chAdd*. Thus, if *mytrack* is a pointer to a structure of type *track*, then it is added to the *trackList* by:

```
chAdd( trackList, mytrack );
```

The variable *trackList* becomes a convenient handle when manipulating collections of structures. It collects the type information and the pointers to the structures together which is needed for Cheetah input and output.

A complete example of the use of Cheetah lists and other Cheetah functions is shown in Fig. 1 with the Cheetah functions highlighted in bold case. The example function

```
chList *findVees( chList *trackList )
{
  chList        *vertexList;
  struct track **tracks;
  struct vertex *avertex;
  int    i, j, count;

  if ( !chIsKindOf( trackList, track_T() ) ) {
    fprintf( stderr, "findVees: bad input\n");
    return NULL;
  }
  vertexList = chNew( "Vertices", "2 prongs",
                     vertex_T() );
  tracks = chDataPtr( trackList );
  count = chCount( trackList );
  for ( i = 0; i < count-1; i++ ) {
    for ( j = i+1; j < count; j++ ) {
      avertex = goodVee( tracks[i],tracks[j] );
      if ( avertex != NULL ) {
        chAdd( vertexList, avertex );
      }
    }
  }
  return vertexList;
}
```

*Fig. 1 Complete example of code with Cheetah.*

takes as an argument a Cheetah list containing track structures and returns a Cheetah list containing all good two prong vertices. The *chIsKindOf()* function is one of Cheetah's convenience functions. It checks that the contents of the input Cheetah list is of the type expected. Other Cheetah convenience functions are *chDataPtr()* and *chCount()* which return the contents of the list as an array of pointers and the size of the array respectively. It is not difficult for an experienced C programmer to imagine how these functions are implemented or the fact that they may be implemented as macros instead of true functions.

## 4. Creating Data Typing Information

In order for Cheetah to perform input and output operations on data, it needs type information of the data for each type in the stream. As stated in the previous section, this information is returned by a Cheetah type function, *i.e.* the functions with the trailing *_T* in the examples above. These type functions can be generated by the *chgen* utility which is modeled after Sun's *rpcgen* utility[3]. It parses files containing the declaration of the structures. The syntax of the data definition language used in these files is close to standard C structure declaration syntax. Two files are generated by the *chgen* utility. They are a *.h* header file and a *.c* source file. The *.h* file contains declarations of the structure and the corresponding type function. The *.c* file contains the source code of the type function.

The Cheetah typing mechanism is not limited to C structures. Most commonly used data structures, even those in other programming languages, can be described to Cheetah. For example, data contained within a FORTRAN common block, or even the whole common block, can be handled by Cheetah as if it were a C structure. It is by this means that Cheetah can provide support for FORTRAN data structures and the Cheetah *chgen* utility can even be used to produce files that are to be included in FORTRAN source code.

The Cheetah data description language allows for a rich assortment of data types including structures, vectors, variable-length arrays, enumerations, discriminated unions, C-style pointers, and FORTRAN 77 style "pointers" (relative word offset pointers). Multiply referenced data, such as occur in doubly-linked lists, are also handled correctly.

C++ object persistence is in development, as well as support for Fortran 90 derived data types. A subset of the data description language can be used to achieve common, named access to data in FORTRAN common blocks from both FORTRAN and C.

## 5. Cheetah Input and Output

Cheetah input and output is modeled after the C I/O system. There are functions to open a stream, which return a pointer to the opened stream. There are several kinds of Cheetah streams. They can be binary or plain text. The binary streams can either use the industry-standard XDR format[3], the architecture's native format, or a close-packed format. The text streams can either be in a format similar to C-style initialization, or in a binary-faithful format, which is a direct translation of the format used in the binary file. A stream can be opened either to a file system, a TCP/IP network connection or to a memory buffer.

The Cheetah read and write functions take two arguments as shown in Fig. 2. The first is the pointer to the stream and the second is a Cheetah list. No distinction is needed on what kind of stream is being used. Thus, Cheetah I/O is network transparent. In the example, the main program writes the contents of a Cheetah list called *event* to a file. The contents of *event* is a Cheetah list containing tracks and if any vertices are found a Cheetah list containing vertices.

```
main() {
  chList *eventList, *trackList, *vertexList;
  chFile *chout;

  chout = chOpenFile( "output.chdata", "wb" );
  eventList = chNew( "MCEvent", "MC Event",
                     chList_T() );
  trackList = MCEvent();
  vertexList = findVertices( trackList );
  chAdd( eventList, trackList );
  if ( vertexList && chCount(vertexList) > 0) {
    chAdd( eventList, vertexList );
  }
  chWrite( chout, eventList );
}
```

*Fig. 2 Example using Cheetah output function.*

The Cheetah read function returns a Cheetah list which is a replica of the one written. Thus not only is the data re-established in memory, but also the type information. The type information can be used to check that the data is of the expected type (*e.g.* with the *chIsKindOf()* function). It can also be used by general purpose interactive programs to present to the user detailed information on what was read. This information includes the type information for members of C structures, their names, and even comments.

## 6. Summary

Cheetah is a useful and flexible data management system based on the C programming language with support for other languages. It facilitates the transfer of data between memory and I/O streams while minimizing interference with the native methods of data access. The streams are associated with disk files, memory buffers, or are network data streams. The Cheetah list construct is used to direct the I/O and can optionally be used by the programmer to help manage the data.

The Cheetah kernel is written in ANSI standard C, but can be compiled using C++ compilers. Cheetah is highly portable with no machine-dependent switches. The source code and complete documentation for Cheetah will be made available via anonymous ftp from the file server `heplib.slac.stanford.edu`.

## References

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (Prentice Hall, Englewood Cliffs, 1978).

[2] Sun Microsystems, Inc., *rpcgen Programming Guide*, (see also man pages on UNIX systems)

[3] Sun Microsystems, Inc., *XDR External Data Representation Standard*, RFC1014, (see also man pages on UNIX systems)..