Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

SIMULATION WITH C

Floyd H. Grant, III, Ph.D.
Douglas G. MacFarland
Pritsker & Associates, Inc.
P.O. Box 2413
West Lafayette, IN 47906

INTRODUCTION

The C Programming Language was developed at Bell Laboratories in 1972 by Dennis Ritchie. Since that time, C has had major acceptance as a modern programming language suitable for a large variety of applications. Those applications include the operating system UNIX, which was written in C for portability. It requires only a C compiler and the rewriting of some low level routines in machine language for implementation on virtually any computer. The best documentation of C is provided in a book entitled The C Programming Language by Kerninghan and Ritchie.[1]

This paper explores the use of C as the host language for discrete event simulation. One of the primary motivators is the portability of C code. C compilers are now available for a variety of computers from microcomputers to mainframes. Hence, a simulation model developed in C could execute on a microcomputer or a mainframe, given a standard C compiler. The standard for C is well defined via compilers which have a full implementation of C. Kerningham and Ritchie's text provides the documentation for that standard.

As simulation modeling typically requires extensive programming, the features available in C are of interest. C permits the user to program on a variety of levels. This may be as detailed as being close to machine code, or it may be on the same level as FORTRAN. This flexibility provides many alternatives for model development not available in earlier host languages.

Additionally, C contains many features that are ideal for simulation. These include capabilities such as named attributes, pointer datatypes for entities in the event calendar and in files, dynamic core allocation, and other organization features that make simulation models easier to develop. Details on these various features will be discussed in the subsequent sections of this paper. C is also actively being extended and refined. A new version of C called C++ [2] has been developed at Bell Laboratories. C++ provides additional capabilities in the form of a preprocessor which allows the user to extend the language by creating new data types and operators. For simulation, this means that we can create operators that will perform the functions most often required in a simulation model.

The objective of this paper is to illustrate how C capabilities can be used in simulation. We will discuss the components typically found in a simulation modeling language and describe how C can be used to address these capabilities in ways not previously available. Examples will be provided to illustrate the various concepts.

ENTITIES AND ATTRIBUTES

In discrete event simulation the programmer is typically concerned with the flow of "entities" through the simulation model. Depending upon the particular model, an entity could be a customer, a part being manufactured, a telephone call, or a cart in an AGV system. In many models there will be several types of entities in the system at the same time.

Each entity in the system will have attributes associated with it. The attributes of an entity might be its weight, the amount of a particular resource it will require, and the time at which it entered the system. Since the code written by a programmer to describe his system will often refer to these attributes, and because these attributes may differ from entity to entity, it is very desirable to be able to refer to an entity, and its attributes, by names which are meaningful to the programmer. The C programming language provides such a capability through the use of structures, unions and pointers. The following sections provide details on the use of C constructs to support the use of entities and their attributes in simulation models.

Structures and Named Attributes

The C programming language, through the use of unions and structures of data elements, provides an excellent environment for referencing and manipulating entities and attributes by name. An entity is defined as a structure composed of the data elements which represent its attributes. The C programmer can create a data representation in which each entity, and each attribute of each entity, may be referred to by a meaningful name such as "entity.car" or "entity.car.color".

By entering the entity and attribute declarations into a separate "header" file, the C preprocessor "#include" command may be used to reference these declarations from many different source files. An example of what this entity and attribute definition header file might look like is shown in Figure 1.

```
struct Car_typ {
    float ar_time;
    int color;
    int body;
    int engine;
    int options;
} ;

struct Truck_typ {
    float ar_time;
    int color;
    int bed;
    int engine;
    int trans;
} ;
```

```
union Entity {
    struct Car_typ car;
    struct Truck_typ truck;
} ;
```

Figure 1. Entity Attribute Definition.

In this example, the model has two types of entities, cars and trucks. The programmer has defined a structure consisting of the attributes of the car, and named it Car_typ. He has also defined the structure Truck_typ, which contains the attributes for a truck. Note that, although these attributes are similar, by defining the entities as separate structures, the programmer is able to give them names which are specific to each entity. The final statement in the header file defines the type Entity to be a union which may contain either of the two entity structures. The reader should note that these declarations serve only to define the entity types and do not create any occurrences of these types. This is done elsewhere in the model.

The C Preprocessor and Symbolic Constants

One of the most powerful features of the C programming language in enhancing the readability of its source code is the preprocessor of the C compiler. One of the features of the preprocessor is to allow the programmer to represent any text string with another text string through the "#define" command. By using the #define command, the user may replace constants, parts of expressions, or even entire statements, with names which are more meaningful to the application.

For example, consider the attribute entity.car.color from the previous section. For efficiency of storage and manipulation, this attribute is declared to be an integer. We could agree that the number 1 would represent the color red, and then the statement "if(entity.car.color == 1)..." would test the color attribute to see if it is red. This statement would invariably cause the reader to refer to his copy of the color definitions that had been agreed upon. If, however, we use the C preprocessor command "#define RED 1" this statement becomes:

"if(entity.car.color == RED)...".

Readability is even further enhanced in switch constructs such as the one in Figure 2. In this case, we want to process the "car" entity in different ways based on its color. (The use of the "switch" construct is also a major reason that the color attribute was declared as an integer in the first place, since the object of a case label must be an integer constant.)

```
switch(entity.car.color) {
  case RED:
      .
      .
      .
  case BLUE:
      .
      .
      .
```

Figure 2. String Replacement with #define.

Pointers for Entity Manipulation

One of the distinguishing features of the C programming language is its ability to deal with pointers, which are similar to machine addresses. This provides a considerable advantage in efficiency when passing, or otherwise manipulating, complex data structures. For example, by using a pointer to the entity union defined above, the C programmer can reference any element and field in that structure by name through the use of the -> operator. As an example, if "eptr" has been declared to be a pointer to a union of type Entity, then the programmer may refer to the car color field of the structure pointed to by eptr as "eptr->car.color".

In programming a descrete event simulation, the programmer will often make use of functions which perform tasks involving entities (filing, scheduling, etc.). By using a pointer to an entity structure, access to all of the attributes of an entity may be passed to these functions with a single argument. Possibly even more significant is the fact that, while a function may return only a single value, this value may be a pointer. By taking advantage of this, we can write functions returning entity pointers which lead to concise and readable notation such as "entity = get (1,QUEUE1);".

Dynamic Storage Allocation

Most modern programming languages provide facilities for the dynamic allocation of storage. C is no exception. Through the use of the functions sbrk, alloc, and free, the programmer may allocate and free blocks of storage at program run time. This has obvious advantages in a simulation environment.

When constructing a simulation model, the programmer often has little or no prior knowledge as to the amount of congestion in the system (i.e., the number of entities in the system concurrently). In languages which provide only static storage allocation, notably FORTRAN, this situation causes the programmer to grossly over-allocate the storage for entity lists or to risk termination of his simulation run due to lack of core storage. The problem is compounded by the fact that in order to alter the amount of storage allocated it is necessary to recompile one or more modules, and relink the entire system.

By allowing the entity storage requirements to be allocated at run time, C relieves the programmer of the task of estimating the requirements of a model run and may also help reduce the load on the computing system by reducing the load on the system memory resources.

Note that while there is some overhead involved with dynamically allocating and releasing storage, this can be minimized by properly designing the functions which handle entity creation and destruction. Rather than allocating and freeing storage as each entity is created an destroyed, this storage could be allocated in groups of some fixed number of entities. The size of the group of entities for which storage is allocated is a parameter which could be tuned to the type of model which is run most often at a particular installation.

LIST MANIPULATION

Since the manipulation of lists is so prevalent in simulation, one of the services provided by a discrete event simulation package should be a number of functions for dealing with lists. Functions should be available for inserting an entity in a list, removing an entity from a list and finding an entity which meets a specific description from within a list. In addition there should be a function to schedule an event.

The C programming language has some capabilities which make it very well suited to writing functions which perform list manipulation. The following sections discuss these capabilities and the specific areas of discrete event simulation to which they apply.

Pointers for List Manipulation

As mentioned in previous sections, C provides a powerful facility for the manipulation of entities through the pointer data type. This capability is even more advantageous in the manipulation of lists of entities.

A typical list is doubly linked, requiring a forward and backward pointer. In languages such as FORTRAN, "pointers" usually consist of integer subscripts into a large array which comprises the list space. This results in some problems in writing procedures which return an entity from a list. Either they must copy the attributes of the entity into another array, a time consuming task if there are many attributes, or they must return a subscript into the list space resulting in very awkward references to the attributes of that entity.

In C this problem can be totally avoided by returning a pointer to the attributes of the entity. This allows the code of the simulation model to refer to these attributes by name without any need to physically move them to a new location. In fact, through the use of pointers, it should never be necessary to physically move the attributes of an entity in memory, regardless of how many lists it appears on.

Another advantage of using pointers in the manipulation of lists is that the simulation code need not have any knowledge as to the structure of the pointer system. Although a header containing a forward and backward pointer must be bound onto an entity in order to link it into the event list and other lists, it is possible, through pointer arithmetic and casting, to completely hide the existence of this header from the model code which refers to the attributes of the entity. Again, this is done without the need to copy attributes into a temporary variable.

Figure 3 is an example of how the C programmer might write a segment of code to remove the third entity from a list called QUEUE1, and place it in the list called QUEUE3 using functions designed for the manipulation of pointers. The function Q_size returns the number of entities currently occupying a particular list, and is used to determine if there are at least 3 entities in QUEUE1. Note that although this code segment is concise and compact,it is still very readable.

```
struct Entity *get();

    if(Q_size(QUEUE1) >= 3)
      put(QUEUE3, get(3,QUEUE1));
```

Figure 3. List Transfer in C.

List Ordering

In simulation, as in any application which deals with lists, a matter of great concern is the ordering of lists. To avoid unnecessary searching, the event list should be maintained in increasing order of event time. Queues are often FIFO or LIFO, but more complex orderings are not uncommon. Means for efficiently ordering and searching lists are of paramount importance in the performance of the simulation model. C provides some unique capabilities for the manipulation of entity lists.

In C it is possible not only to pass the address of a procedure to another procedure, but to store this address in a pointer variable. Taking advantage of this capability, the address of a user-written comparison function could be passed to, and saved by, the simulation executive at the time when the list was initialized. This comparison function would be passed pointers to the attributes of two entities, and would return their relationship in the desired ordering. (Greater than, less than or equal.) This function would then be used by the simulation executive in determining the ordering for that list. Through the use of a comparison function, it is possible to maintain very complex orderings, without the need for the simulation executive to know about the structure of the entities being manipulated.

When a list is to be maintained in a LIFO or FIFO order, the simple doubly linked list is quite sufficient. For more complex orderings, however, the doubly linked list results in a linear search, either on insertion, or removal from the list. This may be avoided by structuring the list as a binary tree. The same forward and backward pointers that serve to chain a doubly linked list together may be used a left and right pointers in a tree structured list. The time required to search such a binary tree is a log factor less than would be required in a linear search of a list of the same size. Combined with the user-written comparison function discussed above, this is a very powerful capability for efficiently maintaining or searching lists with other than LIFO or FIFO orderings. Both linked lists and binary trees could coexist in the C based simulation model.

The Event List

The event list is a very special list in the simulation environment. It is used to store entities while they are awaiting processing by the simulation executive. This list requires two additional attributes: the event type and the event time. With many descrete event simulation packages, these additional attributes must be carried at all times by every entity in the system, regardless of which list they are currently occupying. Through the use of pointers and dynamic storage allocation, the C programming language provides the capability to bind these additional attributes onto an entity only while the entity is actually on the event list, and to completely hide the existence of these attributes from the application code.

In addition to the efficiency of storage afforded by eliminating the need to carry the additional attributes throughout the system, considerable speed would be gained by maintaining the event list as a binary tree in increasing order of event time. This would ensure a that minimal time would be spent in finding and removing the next event.

MULTIPLE CLOCK TYPES FOR THE SIMULATION EXECUTIVE

One topic which is seldom addressed in the creation of a descrete event simulation package is the issue of multiple clock types. The simulation executives of most packages use a single precision, floating point clock. While this is quite adequate for most simulation needs, there are surprising number of situations which require the accuracy of a double precision clock, or the speed of an integer clock.

In the simulation of communications systems, events often occur in very short time intervals, thus requiring a double precision clock if the model is to be run for any significant period of time. Computer systems are another application where event intervals may be measured in nanoseconds. However, in some computer simulation models events occur in integer multiples of the machine clock cycle, suggesting that an integer clock be used in the executive for these models.

Many micro-computers lack any hardware instructions for floating point mathematics. On these machines, floating point math must be done in software, and is therefore often quite slow. In porting a simulation package to such a micro-computer it may be desirable to use an integer clock in the simulation executive in order to greatly increase the execution speed of the model. This will, of course, depend upon whether or not the model lends itself to this type of clock.

With some existing packages, changing the executive from one clock type to another may be accomplished by editing the source code and changing the declaration of any variables used to hold a clock time. With other packages, the assumption of a single precision floating point clock is much more deeply rooted and would require a nearly complete re-write in order to change to another type. In either of these two cases the result would be multiple copies of the source code for the simulation package, all of which would have to be maintained separately. In addition, the source code for a model would most likely not be compatible with versions of the package other than the one for which it was written.

The ability of the C programming language to define new data types in terms of existing types provides a simple solution to this problem. By using the typedef statement to define a new data type with the symbolic name CLOCK, and which represents the actual data type of the clock, we can hide the actual data type from the code of the simulation executive and from the programmers model code. If all variables and functions which hold or return this data type are declared to be of type CLOCK, changing the type of the clock will be as easy as changing a single typedef statement in a header file. (See Figure 4)

```
typedef double CLOCK;
CLOCK  tbeg, tfin, tnow, tlast;
```

Figure 4. Alternate Clock Types.

While the use of the typedef statement would still result in multiple executables, there would only be a single source file, greatly reducing maintenance requirements.

In addition to the data type of the clock, the type used for many of the other executive functions could be parameterized in this fashion also. Some likely candidates would be the statistical accumulators, the random number seed and some of the other variables used in random number generation. By making good use of the typedef statement it would be possible to generate versions of the simulation package for a wide variety of machines from a single source file. This extreme portability, and the ease of porting the C compiler itself, is one of the major factors in the increasing popularity of C.

STATISTICS COLLECTION

The purpose for constructing and executing a computerized simulation model is obviously to gain some insight into the behavior of the system it represents. The normal means for determining the performance of a simulation model is through the collection of statistics based upon parameters of the model which are of interest to the modeler. Simulation statistics may be grouped into two categories: time weighted statistics, such as the mean utilization of a resource over the simulated period, and observation based statistics, such as the mean waiting time in the queue for a machine. The following discusses issues in which C can provide much support for the collection of statistics.

Assignment Operators And Register Variables

The major advantage of C in the collection of observation based statistics is that of speed. The C programming language provides a wealth of operators; among these are the assignment operators. Assignment operators are unique to C and are very close to the instruction set of the machine itself. They result in compiled code which is very compact and efficient.

Another feature of C which helps generate more efficient code is the register variable type. By declaring a variable to have the register storage class, a programmer may inform the compiler to make an extra effort to treat that variable in an efficient manner. This varies from machine to machine, but usually results in the variable being maintained in a machine register, rather than in memory. This eliminates the memory fetch step, and the time required to execute it.

The typical way in which a simulation package provides observation based statistics to the programmer is through a call to a statistical collection function. This function is passed an observation and the index of the stream in which the observation is to be collected. The way this function usually maintains the statistics is through updating a counter, normalized sum of observations and normalized sum of squared observations. In performing this task, the collection function must perform multiple memory fetches of both the observation and stream index. By declaring both of these parameters to be register variables and by using assignment operators to update the statistical accumulators, it is possible to write very efficient routines for the collection of observation based statistics.

Pointers For Time Weighted Statistics

Time weighted statistics, because of the nature of their calculation, must be collected at the occurrence of every event. To relieve the programmer from having to do this himself, this task is usually performed by the simulation executive. Before the simulation executive can collect time weighted statistics, it is necessary for it to know which variables are to be collected. In most programming languages this is done by placing a set of variables in a predetermined location (common block or external array) and requiring the programmer to use these variables to contain any values within his model about which he desires time weighted statistics.

This is quite restrictive in terms of the names and/or types of variables which the programmer may use in constructing his model. These restrictions may be eliminated in C, once again through the use of pointers. A C function may be written which takes a pointer to a variable and the data type of that variable as arguments, and saves them for use by the simulation executive in collecting time based statistics. The programmer may then call this function at the time he initializes the model for each variable he wishes to have collected during the simulation run.

Figure 5 shows a sequence of code in which the programmer has requested that time weighted statistics be collected on a floating point variable res_avail, and on an integer variable serv_stat. The unary operator & is used to take the addresses of the two variables and pass them to the simulation executive for use during the simulation run. The symbolic names FLOAT and INT are defined in the header file simtype.h, and the programmer may use them wherever this file has been included.

```
#include "simtype.h"

     timstat(FLOAT, &res_avail);
     timstat(INT, &servstat);
```

Figure 5. Statistics Collection.

GENERATION OF RANDOM DEVIATES

One of the most frequently executed tasks in a typical simulation model is the generation of random deviates. Because of this, it is very desirable to generate these deviates as efficiently as possible.

The most common random number generators in use today are those classed as linear congruential random number generators. The linear congruential algorithms rely heavily upon modulus division. Because modulus division is one of the slowest operations to perform on a digital computer, much effort is spent trying to avoid it through bit shifting, synthetic division, etc. Although the C programming language provides a rich set of operators for bit shifting and bit-wise logical functions, in most cases the C programmer need not resort to any of these techniques.

If the choices for the coefficients are made carefully, and the seed variable is declared as unsigned, unsigned overflow may be used to perform implicit modulus division. This results in much faster execution than would be possible in a language which can perform only signed math.

Figure 6 shows the source code for a C function to return a uniform (0,1) random deviate of either float or double data type, depending upon the type of the clock. It is written in such a way as to be portable to any machine with a native word size of 16, 32 or 36 bits. The #ifdef statements are compiler directives, and instruct the compiler as to which section of source code is to be compiled. Since they are evaluated at compile time, they have no effect on the execution speed of the function.

The header file simtype.h contains the typedef statements for CLOCK and SEED, as well as a #define statement for ONE of the word size symbols. This is an excellent example of how the typedef statement, #define statement, conditional compilation directives, and inclusion of header files may be combined to make C source code extremely portable and versatile.

```
#include "simtype.h"

  CLOCK random(seed)
/*------------------*/
register SEED *seed;
{
#ifdef 16_bit_seed

        .
     Code for 16 bit seed.
        .

#endif
#ifdef 32_bit_seed

        .
     Code for 32 bit seed.
        .

#endif
#ifdef 36_bit_seed

        .
     Code for 36 bit seed.
        .

#endif
}
```

Figure 6. Random Number Generators.

CONCLUSION

This paper provides a discussion of the use of the C programming language for simulation. It defines important features of C which can support simulation language development and describes how these features can be used. We anticipate that the development of a formal language in C will provide a consistent language across a wide variety of machines and provide capabilities to modelers not currently available.

REFERENCES

[1] Kernighan, B.W., Ritchie, D.M. (1978) The C Programming Language. First Edition, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632 pp.228.

[2] The C++ Programming Language - Reference Manual "Computing Science Technical Report No. 108" Bjarne Stroustrup, AT&T Bell Laboratories, Murray Hill, New Jersey 07974 January 1, 1984.

[3] Law, A.V., Kelton W.D., (1982) Simulation Modeling and Analysis. Consulting Edition, McGraw-Hill Series in Industrial Engineering and Management Science. pp.400

[4] Fishman, G.S. (1973) Concepts and Methods in Discrete Event Digital Simulation. First Edition, John Wiley & Sons, New York. pp. 385

[5] Pritsker, A.B., (1984) Introduction to Simulation and SLAM II. Second Edition, Systems Publishing Corporation, West Lafayette, IN 47906 pp. 612.

[6] Pritsker, A.B., (1974) The GASP IV Simulation Language First Edition, John Wiley & Sons, New York pp.451