Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

DISC++: A C++ BASED LIBRARY FOR OBJECT ORIENTED SIMULATION

Eric L. Blair
Sathyakumar Selvaraj

Department of Industrial Engineering
Texas Tech University
Lubbock, Texas 79409

0. ABSTRACT

The Object Oriented Programming (**OOP**) paradigm is generating considerable interest and excitement among systems analysts and programmers concerned with a wide range of applications. This paper presents **DISC++** (**DI**screte event **S**imulation in **C++**), a library of routines written in C and C++ which supports the design and programming of simulation models under both the event scheduling and process interaction world-views. **DISC++** allows the simulator to construct simpler models from "standard" library objects or design more complex models by deriving specialized and sophisticated objects from the library objects. The **OOP** philosophy encourages an evolutionary model building process in which code (object classes) is reused and continuously upgraded. A simple example is given to illustrate an application of the process interaction modeling capabilities.

1. INTRODUCTION

The driving force behind the development of **DISC++** is the belief that to be truly successful, a simulation language should support more than one world-view and accommodate a full range of user sophistication and needs. A testament to this logic is provided by the success of such simulation languages as **SIMSCRIPT II.5** (Russell 1983) **SLAM** (Pritsker 1986), and **SIMAN** (Pegden 1982). Each of these languages supports both the event scheduling and process interaction world-views. **SLAM** and **SIMAN** are **FORTRAN** based and permit the user to code event routines, special functions and subprograms in **FORTRAN**. **SIMSCRIPT II.5** is itself a powerful language which can be used for developing programs other than simulations.

DISC++ supports both the event scheduling and process interaction world-views for simulation modeling. **DISC++** is a library of C and C++ functions which provide a framework for constructing and running simulation models using the C++ language. Since C++ is a superset of C, it supports procedural programming. This is fully sufficient to implement an "event scheduling" world-view for model development as illustrated by its predecessor **DISC** (Selvaraj et al. 1988) which consists entirely of C functions. The object oriented programming (**OOP**) features of C++ and **DISC++** make it a natural environment in which to build process interaction based simulation models.

This paper presents a brief discussion of **OOP** and its potential use in simulation. It also describes the implementation of **OOP** concepts in **DISC++** and presents arguments for the serious consideration of **DISC++** as the language of choice for the development of simulation models.

2. OBJECT ORIENTED PROGRAMMING:

OOP is a paradigm (conceptual framework) for program development which supports abstract data types, encapsulation, operator and function overloading, and inheritance. It is not our intention here to provide a definitive statement of the more subtle aspects of **OOP**, but rather to provide insight as to why **OOP** and C++ are powerful developments in the art and science of computer programming with major significance for the future of simulation modeling. A more thorough discussion of **OOP** and C++ can be found in Cox 1986 and Stroustrup 1986, 1988. Its use in simulation is discussed in Roberts and Heim 1988.

Abstract Data Types

Data abstraction allows the programmer to create data structures which can then be manipulated in the same way and with the same level of efficiency as language-defined types (such as int and float). The key construct to implementing abstract data types in C++ is the **class**. An **object** is a particular realization or instance of a class. A class is defined to have both a private and a public part. The private part is essentially a data structure in the tradition of **Pascal** and C. The elements of this data structure are referred to as "member variables." Class member variables are hidden from the outside world except for functions which are explicitly given permission by the programmer of the class with the designation (in the class declaration) of "friend." The public part of a class is a declaration of friend and "member" functions and operators. A member function (or operator) is defined with the class while a friend is defined external to the class (possibly with another class). The important thing to note is that only those functions which are explicitly declared as members or friends may access class member variables. Functions and operators may be "overloaded", that is, having multiple definitions each distinguished by the set of operands or arguments types; the compiler selects the correct interpretation at run time based on these types.

Encapsulation

Encapsulation and overloading are the key elements of **OOP** which distinguish it from other programming paradigms. The encapsulation principle refers to the design of classes as self-contained program units. With conventional programming methods, the data structures and the functions which use or change the data are developed with knowledge of each other but separately. Under this paradigm, a programmer is free to use the data structures developed elsewhere, but the responsibility of assuring that the access is correctly performed is left to the user, not the original developer of the structure. This, of course, facilitates the creation of new uses for old data structures. It also requires that the user have a detailed knowledge of the data structure and other accessing functions so as to effectively implement the new functions. Clearly, there is a significant risk of data corruption as new functions are added to the access base.

Encapsulation requires that present and future uses for a data structure be explicitly recognized by the developer of the data (a class) and that the interface between the data (member variables) and the outside world be declared (as member or friend functions). As such, the class is a black box which offers a selection of services but hides the details of how these services are actually performed within the member functions. The responsibility for designing the interface between the class and the outside world is transferred to the developer of the class. This leads to the design of a class as a self-contained encapsulated programming entity, which in turn leads to reusable code. Reusable code is <u>the</u> motivation for **OOP**!

Overloading

The overloading construct of **C++** is a very cleaver approach to the generalization of functions and operators. The key concept here is that the same operator symbol (e.g., +, -, *, etc.) or function name can refer to different procedures; the specific procedure chosen will depend upon the object type(s) supplied as operands or arguments. In conventional programming languages, this property exists for operators and language-defined types; e.g., in **C**, the "+" operator is defined for both int and float types. Overloading allows for the extension of this property to operators and functions concerned with the manipulation of user-defined types (i.e., objects).

Inheritance

Inheritance allows for the construction of a class to include all the members (data and functions) of another class. The class whose members are included is called the **base** class; the class being constructed from the base class is called the **derived** class. Inheritance encourages the design of classes with the object of creating "building block" and a hierarchical set of classes. This also promotes reusable code since a new class is more easily constructed from an old class which already has some (but not all) of the desired members.

3. C++ AND SIMULATION

C++ presents an excellent language base from which to build simulation models. This is true regardless of the world-view being employed. There is a natural analogy which may be made between the physical objects of the system to be simulated and C++ classes. This is clearly true for entities such as "customer", "machines", "bank clerks" and is also appropriate for more abstract system elements such as "files", "sets", "statistics", etc.

4. DISC++

DISC++ consists of two libraries of C functions and C++ classes which support the development of simulation models using the **ZORTECH C++** compiler. The first library consists of basic tools which are used to perform the "mechanics" of discrete event simulation. These are objects which enable the creation and manipulation of records and files, the passage of simulated time, the generation of random variables and the collection and reporting of statistics. There are also objects which are used to create a user environment such as windows, plots and interrupts. This library contains all the elements necessary to support the use of the event scheduling world-view for simulation. The user writes the event logic as C routines or C++ objects and defines access to library objects through function calls which have been declared as "virtual friend" in the class declaration (a virtual function is one which has been declared but not defined in the class declaration).

The second library consists of a set of C++ classes which serve as base classes for the development of object classes to support a process interaction world-view of simulation modeling. This functionality includes both the processes/resources implementation (as exemplified by **SIMSCRIPT II.5**) and the network implementation (prominent in **SLAM** and **SIMAN**). The following discussion will illustrate the use of network constructs.

Consider a simple network model of a single server queueing system as shown in figure 1. The network consists of five elements (or objects). There are three types of nodes: a **g_node** (generator), a **q_node** (queue) and a **t_node** (terminator). The first **arc** (ARC_1) serves to direct the path of flow. The second **arc** (ARC_2) directs flow and represents the service activity. The customers are represented by an object class called "record."

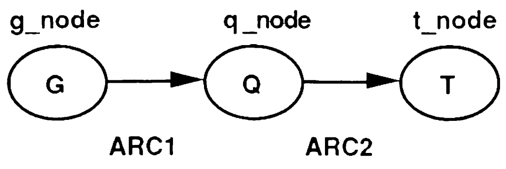

Figure 1. Network diagram for a single server queue.

Class declarations of these elements are given in listings 1 and 2. The record class consists of four member variables: last_rec, id_no, i_array and f_array. The last_rec member is declared as "static," indicating that a single location in memory is shared jointly by all existing instances of the class record. The last_rec member is initialized to zero and incremented each time a new record is created. The new record's id_no member is set equal to last_rec before it is incremented, thus providing a unique and sequential serial number for each instance of the class record. The i_array and f_array members point to arrays of integer and float attributes respectively. Customer related attributes such as "time of entry to the system", etc. can be stored in the integer and float attributes. For this simulation, there is no need for entity attributes and these pointers could be set to NULL with the constructor function call (the first member function). Other member functions are included to set and return the current value of the attributes.

The link class is provided to place record objects into files without adding members to the record for storage of predecessor/successor information. Although this scheme takes more memory, it allows the same record object to be in multiple files at the same time. The constructor for the link class has two arguments which pass pointers to predecessor and successor links when the link is created. The two classes file and file_iterator are declared as "friend" permitting objects of these classes access to link private members. The code for member functions and the class declaration for file_iterator are omitted in the interests of brevity.

The file class has five member variables: a unique I.D. number, pointers to the first and last links in the file, the file capacity and the current number of links in the file. The constructor has two arguments which determine the id_no and capacity values when the file is created. Two member functions are declared to insert links into a file. The first, with void as the argument, will place the link at the end of the list (as in a FIFO queue). The second insert function has an integer argument k; the link is placed in the kth position from the top of the file. This is a good example of function overloading in C++. The task of removing a record from the file can be performed in two flavors also, analogous to the insert functions.

Listing 2 displays the class declarations of the network components. The g_node class is created by calling its constructor, specifying the I.D. number, the total number of record objects to be generated and a vector of indices for arcs which may be selected to receive the record. The function to determine the (random) time between record creations, schedule_next(), is declared to be "virtual" and must be defined by the programmer. The control class is declared as a friend. It contains the functions to manage the simulation calendar and execute events. As each record is created, its path is selected by the virtual function select() which is also supplied by the programmer. The variable member count is incremented and compared to the member total_count. If the count is equal to the total_count, the process is terminated.

The q_node class is derived from the file class. It inherits all the member variables and functions of the file class. In addition to the members provided by file, it has members of its own to identify the possible sources and dispositions of records (input_arcs and output_arcs, respectively). It also has member variables which are used as accumulators to collect simulation data necessary to compute the mean and standard deviation of the the number of records in the queue. The destination of records leaving the q_node must be selected with a user provided function select().

The t_node class is also derived from the file class. It collects records in its file and destroys them. Each record terminated causes the count member variable to be incremented by one. When the count is equal to (or greater than) the total_count member variable, the simulation is halted and run statistics are generated and reported.

The arc class is derived from the file class to provide the capability to simulate multi-channel service systems. The (random) service time is determined by the virtual function schedule_next() which must be supplied by the programmer.

Note that the example code is incomplete in that only the class declaration is provided; the function definitions are excluded to keep the presentation of reasonable length. It should also be noted that the class structures are "stripped down" versions of the node structures which would be required for more complex networks (included in the **DISC++** library). The objective of this paper is to illustrate the ease with which process interaction models may be constructed using **DISC++** library classes. This simplicity is illustrated in listing 3 which details the main() program implementing the simple network model of figure 1. The first two declarations of main() declare the two vectors as1 and as2. The first vector as1 is used to prescribe the set of arcs which emanate from node 1 (the g_node). In this example, it is also the list of arcs terminating in node 2 (the q_node). Vector as2 is the list of arcs emanating from (terminating in) the q_node (the t_node). In each vector, the first element is used to store the number of arc indicies in the vector; i.e., the list defined by as1 has one arc, indexed as number 1 (i.e., as1[1] = 1).

The next five lines of main() invoke the constructors of the network elements. The arguments for the g_node indicate the node I.D., the generate number and the set of emanating vectors respectively. The "-1" given as the argument for the generation count is interpreted by the constructor function to indicate that the number of created records is not to be limited.

The net_sim function constructs a control object (not included in the listings) which has the functions which manipulate the calendar and control the simulation. The single argument to this function call is a float number indicating a termination time value. If the simulation is not terminated by the expiration of a t_node, the simulation will stop when the simulated time reaches or exceeds 1000.0 units.

5. CONCLUSIONS

First and foremost, the conclusion reached is that C++ and **DISC++** offer a simulation language/methodology worthy of serious consideration for the future of computer simulation. The main advantage of **DISC++** is that the user can choose the level of programming sophistication and world-view which best fit his/her own abilities and the particular system to be modeled. **DISC++** supports both the event scheduling and process interaction world-views or a combination of the two. **DISC++** is totally compatible with one of the most popular microcomputer-implemented languages available today, **C**, and the language with the fastest popularity growth rate, **C++**.

The simplicity with which process interaction models can be developed is illustrated by the example presented in section 4. Although the total number of lines of code in listings 1-3 seem excessive for such a simple simulation (and of course they are!!), it should be noted that listings 1 and 2 contain class definitions which could become "standard" modules self contained (encapsulated) and used again and again to construct simulation models. For such "standard" network classes, a considerable amount of additional design work would be required (and warranted). More complex and specialized node classes could easily be derived by sophisticated users from simpler node classes exploiting the fundamental concept of OOP. Once designed, these classes could be incorporated into new and more powerful libraries extending the modeling capabilities for all levels of users.

Although we have chosen to focus here on the development of network classes, it should be clear that a wide range of classes could be designed to facilitate the development of simulation models for specific system types. For example, it would be relatively easy to design classes of objects to represent elements of a material handling system (e.g., classes of conveyers, AGVs, carousels, trucks, etc.). Similar applications can be made to communications, health care, and flexible manufacturing systems. Again, **DISC++**, with the help of **C++**, allows the user to choose model components from an expanding library of already defined classes or to construct his/her own model components, possibly derived from the library classes.

REFERENCES:

Cox, Brad J., *Object Oriented Programming*, Addison-Wesley, 1989

Pegden, C. Dennis, *Introduction to SIMAN*, Systems Modeling Corp., 1982.

Pritsker, A. Allan B., *Introduction to Simulation and SLAM II*, 3rd edition, John Wiley & Sons, 1986.

Roberts, Stephen D. and Joe Heim, "A Perspective on Object-oriented Simulation," *Proceedings of the 1988 Winter Simulation Conference*, Dec. 1988.

Russell, Edward C., *Building Simulation Models with SIMSCRIPT II.5*, CACI, 1983.

Selvaraj, S. et al. "C Based Discrete Event Simulation Support System," *Proceedings of the 1988 Winter Simulation Conference*, Dec. 1988.

Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, 1986.

Stroustrup, Bjarne, "What is Object-Oriented Programming?", *IEEE Software Magazine*, May 1988.

```
//----- CLASS RECORD -----
class record
{
  static int last_rec; // A global counter
  int id_no;          // A unique I.D. number
  int* i_array;       // Array of int attributes
  float* f_array;     // Array of float attributes
public:
  record(int* i, float* f); // Constructor
  get_id();                // Returns the I.D.
  get_ia(int k);  // Returns the kth int attrib.
  get_if(int k);  // Returns the kth float attrib.
  set_ia(int k, int j);  // Set int attrib. k to j
  set_if(int k, float f); // Set float attrib. k to f
};

//----- CLASS LINK -----
class link
{
  link* predecessor; // Pointer to predecessor link
  link* successor;   // Pointer to successor link
  record* rec;       // Pointer to a record object
public:
  link(link* p, link* s); // Constructor, place at
                 // end of file
  friend class file;      // File objects need
                 // access
  friend class file_iterator; // Member functions of
                  // this class facilitate file
                 // search
};

//----- CLASS FILE -----
class file
{
  int id_no;     // A unique I.D. number
  link* head;    // Pointer to the first link
  link* tail;    // Pointer to the last link
  int capacity;  // The maximum allowable no. of
            // links
  int count;     // The number of links in the file
public:
  file(int k, int c);         // Constructor
  friend class file_iterator; // File search methods
  int insert();               // Insert record at end
  int insert(int k);          // Insert at position k
  record* remove();           // Remove first record
  record* remove(int k);      // Remove kth record
};
```

Listing 1. record, link and file class declarations.

```
//----- CLASS G_NODE -----
class g_node
{
  int id_no;          // Unique I.D. number
  int* output_arcs;  // Vector of output arc no.s
  int total_count;   // Maximum no. of records to
                 // create
  int count;          // Count of records generated
public:
  friend class control;   // Simulation control
                  // pgm.
  g_node(int k, int t, int* as); // Constructor
  virtual schedule_next(); // Schedule the next
                   // generation
  virtual int select();     // Select output arc
};

//----- CLASS Q_NODE -----
class q_node: public file
{  // q_node is a class derived from the class file
  int* input_arcs;  // Vector of input arc no.s
  int* output_arcs;  // Vector of output arc no.s
  float t_last;      // Time of last change in que,
               // delta_t = t_now - t_last
// The following are used to calculate mean and
// variance for line length (file count)
  float sum_count;      // Sum of count*delta_t
  float sum_count_sqrd; // Sum of
                  // (count*delta_t)^2
public:
  friend class control;  // Simulation control pgm.
  friend class arc;      // Arcs need access
  q_node(int k, int* p, int* q); // Constructor
  virtual int select();    // Select output arc
};

//----- CLASS T_NODE -----
class t_node: public file
{   // t_node is a class derived from the class file
  int id_no;        // Unique I.D. number
  int* input_arcs;  // Vector of input arc no.s
  int count;        // Cumulative terminations
  int total_count;  // Maximum terminations
public:
  friend class control; // Simulation control pgm.
  t_node(int k, int t); // Constructor
};
```

Listing 2. Network component classes (continued on next page).

```
//----- CLASS ARC -----
class arc: public file
{
  int arc_id;        // Unique I.D. number
  int input_node;  // Input node
  int output_node; // Output node
public:
  friend class control; // Simulation control Pgm.
  friend class q_node; // nodes need access
  friend class g_node;
  arc(int i, int j, int k);   // Constructor
  virtual schedule_next(); // Schedule the next
                    // release
  virtual scan_node();     // Scan next node for
                    // permission to enter
  };
```

Listing 2. Network component classes (continued from previous page).

```
//----- NETWORK SIMULATION PROGRAM -----
#include "network.hpp"
main()
{
  static int as1[2] = {1, 1}; // Define an arc set
                    // with 1 element equal
                    // to 1.
  static int as2[2] = {1, 2}; // Define another arc
                    // set with 1 element
                    // equal to 2.
  g_node(1, -1, as1);       // Invoke the g_node
                    // (node no. 1)
  arc(1, 1, 2);             // Invoke arc no. 1
                    // (path only)
  q_node(2, as1, as2);      // Invoke the q_node
                    // (node no. 2)
  arc(2, 2, 3);             // Invoke arc no. 2
                    // (activity)
  t_node(3, 1000);          // Invoke the t_node
                    // (node no. 3)
  net_sim(1000.0);          // Invoke the simulation
                    //control program.
}

//--- SELECTION AND TASK TIME ROUTINES ---
//
// The following virtual functions must be
// provided by the user or chosen from those
// available in the DISC++ library.
//
g_node::schedule_next()
{ // code to determine random time to next record
  // creation
}
g_node::select_node() { return output_arcs[1];}
q_node::select()      { return output_arcs[1];}
arc::schedule_next()
{ // code to determine random time to release of
  // record
}
arc::scan_node()
{ // code to determine if output_node can accept
  // record
}
```

Listing 3. Network simulation program.

AUTHORS' BIOGRAPHIES

Eric L. Blair is an associate professor in the Department of Industrial Engineering at Texas Tech University. His current research interests range from combinatorial programming to the development of simulation modeling languages. He received a B.S. in management engineering and a Ph.D. in operations research and statistics from Rensselaer Polytechnic Institute and the M.B.A. degree from Rochester Institute of Technology. Prior to joining Texas Tech, he was a member of the faculty at North Carolina State University and Rensselaer. He has been employed as an industrial/manufacturing engineer by Xerox and Raytheon.

Sathyakumar Selvaraj is a graduate student in the Department of Industrial Engineering at Texas Tech University pursuing his Ph.D. degree. He received his B.E. degree in mechanical engineering from Anna University (Madras, India) and the M.S.I.E. degree from Texas Tech. His research interests include computer simulation and factory automation.