SAND2001–3788
Unlimited Release
Printed December 2001

UTILIB User Manual
Version 1.0

William E. Hart
Optimization and Uncertainty Estimation Dept
Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM
http://www.cs.sandia.gov/~wehart
wehart@sandia.gov

Abstract

This document provides a user manual for the UTILIB software library. UTILIB includes a variety of generic components for C++ software development including abstract data types, I/O management, sorting routines, and random number generators. The UTILIB library has been used by several software projects at Sandia, including DAKOTA, PICO, SGOPT and NETV.

Contents

1 Introduction

UTILIB is a general-purpose C++ library that includes a variety of algorithmic utilities for software development. These utilities define useful datatypes and classes as well as generic routines. In particular, UTILIB provides a variety of services that facilitate the portability of codes, and in particular porting to parallel computing platforms at Sandia. This library has proven useful in the development of several codes at Sandia, including the SGOPT global optimization library, the PICO parallel branch-and-bound library, and the DAKOTA optimization toolkit.

In its current form, this documentation provides more of a reference manual than a user's guide. There are several reasons for this:

- While documenting classes, I realized that there are many inconsistencies that need to be resolved before a polished documentation can be created. For example:
  - Although the format of the dynamic ADTs like heaps and splay trees is very similar, there are subtle differences in the way that they manage keys versus associated data.
  - Some I/O routines developed by Jonathan Eckstein (e.g. `signalError)` need to be resolved with existing UTILIB routines.

- The support for vectors and matrices will change in the near future, as UTILIB provides more support for the serial and parallel vectors and matrices in the Sandia Petra package. If I'm lucky, all numerical vector/matrix operations will be removed from UTILIB.

- Similarly, the STL libraries in ANSI C++ may eliminate the need to support some of the ADT classes.

- Documentation is rather time consuming, and I felt that it was more important to get a reference guide out soon rather than wait a much longer time period for a detailed user's guide.

Consequently, the documentation provided in this document provides a sketch of the detail needed to fully explain the functionality of this software. Further, some of the classes and functions are only partially documented, which reflects the fact that I expect them to be revised in the future. However, I hope it will be sufficient to get started using these libraries.

It is worth noting some points about the design philosophy for the classes in UTILIB:

- **Encapsulation**: One of the chief advantages for using UTILIB data types (e.g. arrays) is the encapsulation of memory allocation that they provide. This feature has been heavily exploited in my subsequent code, and thus memory allocation is generally quite robust. Further, some classes (e.g. `LinkedList)` include mechanisms for efficiently 'reallocating' memory.

- **Robustness**: A related aspect of UTILIB's design is robustness. I have almost always favored design considerations that ensure robustness. For example, the default behavior for `BasicArray's` is to perform bounds checking. In practice, the performance hit that this causes has been far outweighed by the hours saved tracking down obscure errors.

- **Portability**: Portability across many different architectures is another very important aspect of UTILIB. For example, the common definitions for sorting in `sort.h` have proven very effective for defining portable sorting routines. Similarly, the hard-coded template instantions have facilitated the use of UTILIB on a wide range of parallel computing platforms, many of which support rudimentary implementations of C++.

- **Efficiency**: There is generally no *best* way to implement many algorithms and datatypes, since there invariably are performance/utility trade-offs that need to be made. In the design of UTILIB classes, I have generally looked for solutions that admit a reasonably efficient capability while providing the most general possible design. For example, ADT's like splay trees are very general in their capabilities. Still, they include methods that allow the user to track pointers to items in the tree, which can later be used to efficiently remove those items from the tree.

- **Parallelization**: Support for parallelization is an important function for UTILIB. In particular, UTILIB includes mechanisms for managing parallel I/O through the `CommonIO` class, and the `uMPI` class provides wrappers for parallel communication with MPI.

The components of the UTILIB library include

- **Abstract Data Types**: standard abstract data types like trees and arrays

- **Input/Output Routines**: facilities for encapsulating error routines as well as redirecting I/O through user-defined streams

- **Mathematical Routines**: commonly used mathematical routines, especially array operations

- **Random Number Generation**: generators for commonly used probability distributions and a portable random number generator

- **Sorting**: a variety of common sorting routines, as well as a portable definition of sorting methods

- **System Support**: miscellaneous routines, especially to support portability between different operating systems

These components of the libraries are described in greater detail in the following sections.

2 Abstract Data Types

The `utilib/src/adt` directory contains definitions for the following abstract datatypes:

- Arrays
  - 1D Array
  - 2D Array
  - 3D Array
  - Bit Array
  - Sparse Matrix
- Character String
- Dynamic Data Types
  - Hash Table
  - Linked List
  - Heap
  - Ordered List
  - Queue
  - Stack
  - Splay Tree

These data types are defined as templates.

2.1 Arrays

The array classes provide a nice level of encapsulation for array data types. The main distinction between the Basic, Simple and Num array types is that the Basic array types do not include I/O operations, the Simple arrays add I/O operations, and the Num arrays add numerical vector operations. The following is a brief tutorial on how to use array-type classes in utilib. This is just what you need to get up and running. I'll touch on implementation where necessary to explain behavior and where useful for debugging. The primary advantage of using these classes is extra safety features such as runtime bounds checking and reference counting. Also bit arrays can save a lot of space.

2.2 One-Dimensional Arrays

The most commonly used 1D arrays are BitArray's, IntVector's and RealVector's. The syntax for doubles and ints is the same, so we only include examples for IntVector's, indicating where BitArray syntax differs. To use these arrays, you need to include the appropriate header files:

```
#include "BitArray.H"
#include "DoubleVector.H"
#include "IntVector.H"
```

You can then declare variables as usual:

```
IntVector IntTester;
BitArray BitTester;
```

All of these ArrayData classes have a pointer to the data in the field `Data` . You can get this pointer using the data() function.

```
int *intarray = IntTester.data();
```

Ordinarily, you will not need to do this, but it can be useful for debugging. Inside a debugger like dbx, typing

```
print *IntTester
```

will return useful information like the data location, size, etc, and

```
print IntTester.Data[i]
```

allows you to view the ith entry in the IntVector.

The declarations above (no arguments) construct an object with a NULL data pointer. One can also specify the initial content of the IntVector:

```
int array-of-ints[20];
IntVector IntTester(20, array-of-ints);
```

The first argument is the length of the array and the second is a pointer to an array of the appropriate type. The data field in the IntVector will point to this array. One can also construct a copy of an existing IntVector:

```
IntVector CopiedVector(IntTester);
```

This will allocate a new integer vector and initialize it with the contents of `IntTester`. The Data field in `CopiedVector` points to the new copy.

Frequently, one will need to use an empty constructor (i.e. start with a size-zero vector) and then put in the true data. More generally, you may want to grow and shrink the vector dynamically:

```
IntTester.resize(100);
```

This will allocate a new array of 100 integers and copy the old data (if any) into the beginning of the array. For example, if a 30-element array is resized to 70, the first 30 elements of the new array will be the values from the old array. For BasicArray and SimpleArray objects, you cannot assume anything about the next 40 values (initialize or set them yourself), but for NumArray objects these array elements will be initialized to zero. If an array of size 70 is resized to 30, then the last 40 elements are truncated.

```
IntTester.size()
```

returns the length of IntTester (in integers [more generally, array elements], not bytes).

The equals (=) operator allocates new space. Thus

```
CopiedVector = IntTester;
```

creates a new integer array and copies the contents of IntTester into that array. The Data field of `CopiedVector` points to the new space. If the vector already exists and you want to reuse the already-allocated space, use the `<<` operator:

```
ExistingIntVector << IntTester;
```

This copies the contents of the Data array from `IntTester` into the array for `ExistingIntVector`. The allocated arrays must be of the same size or you will get an error. To copy by reference, use the `&=` operator. Thus

```
Intvector SharedVector &= IntTester;
```

will have the data of `SharedVector` point to the same array that the data of `IntTester` points to (reference counts are properly updated).

The equals (=) operator is overloaded to allow(re)initialization of a vector that has already been created.

```
intTester = 15;
```

This sets every element of `IntTester` to 15 (or obviously some other integer).

Getting array elements works looks like normal array references (at least for 1D arrays):

```
int index, newvalue;
IntTester[index] = newvalue;
newvalue = IntTester[index];
```

Here's where BitArray syntax varies slightly. Instead of typing

```
BitTester[index] = 1;  // wrong
```

you should use

```
BitTester.set(index);  // turns the (index)th bit on
BitTester.reset(index); // turns the (index)th bit off
```

There are ways to manipulate consecutive subvectors. I haven't needed them, so I'm omitting them for now.

Finally, note that the BitArray class is derived from BitArrayBase. This class can be used to define compact representations for user-defined enumeration types. For example, see the TwoBitArray and EnumBitArray classes.

2.3 Dynamic Abstract Data Types

Two different classes of templates have been defined for hash tables, heaps and splay trees. The first is a *simple* template, which uses a simple data type to define the key used by these data structures. The second is a *generic* template, which uses a generic class to define the key. These two classes are derived from from an abstract datatype class that defines the basic operations of the class using abstract operations on the keys.

2.4 Splay Trees

Splay trees, or self-adjusting search trees, are a simple and efficient data structure for storing an ordered set. The data structure consists of a binary tree, with no additional fields. It allows searching, insertion, deletion, deletemin, deletemax, splitting, joining, and many other operations, all with amortized logarithmic performance. Since the trees adapt to the sequence of requests, their performance on real access patterns is typically even better. Splay trees are described in a number of texts and papers [3, 4, 5, 6].

2.5 Heaps

See Cormen, Leiserson and Rivest [1] for more details about heaps. The design of the UTILIB heap classes is similar to the heap classes developed for the PICO software library.

2.6 Hash Tables

See Cormen, Leiserson and Rivest [1] for more details about hash tables. UTILIB provides several default hash functions.

3 Input/Output Routines

The `utilib/src/io` directory contains definitions for the following classes:

- CommonIO
- PackBuffer and UnPackBuffer
- uMPI
- parameter

Additional functions are defined in the following header files:

- comments.h
- nicePrint.h

4 Mathematical Routines

The `utilib/src/math` directory contains definitions for a variety of mathematical and array functions, which are defined in the following header files:

- _math.h
- linpack.h

5 Random Number Generation

The `utilib/src/ranlib` directory contains definitions for datatypes that define random variables. The basic datatype for random number generators is RNG, and two classes are provided for encapsulating linear congruential generators: LCG, which encapsulates the unix random number generator, and PM_LCG, which encapsulates a portable random number generator.

A variety of types are defined for generating random variables using a random number generator. In particular, classes are defined for the Cauchy, Normal, Uniform and Triangular distributions. These distributions are described in detail in a variety of texts (e.g. Evans, Hastings and Peacock [2]). The MNormal and MUniform provide multivariate versions of some of these distributions, and the DUniform class provides the discrete uniform distribution.

6 Sorting Routines

The `utilib/src/sort` directory contains definitions for a variety of sorting and ordering functions, which are defined in the following header files:

- sort.h

6.1 Comparison Semantics

Many routines in UTILIB perform a comparison between two objects and return an integer flag. If we are evaluating how `A` relates to `B`, then the standard comparison semantics for the return value `x` is that `x` is less than zero if `A` is before `B` in the order, `x` is greater than zero if `A` is after `B` in the order, and `x` is zero if they are equal. In the context of numerical values, `A` is before `B` if `A` is less than `B`. Finally, note that if the comparison function is a method of an object, like

```
A.compare(B)
```

then the comparison is evaluating how `A` relates to `B` (and not how `B` relates to `A)`.

7 System Support

The `utilib/src/sys` directory contains definitions for a variety of support functions, including functions that are system-specific. These functions are defined in the following header files:

- _generic.h
- alloc.h
- errmsg.h
- general.h
- real.h
- seconds.h
- signalError.h
- stdlibmpi.h
- utilib_dll.h

8 Installation

8.1 Downloading

The UTILIB software can be downloaded either as a compressed tar file or directly from the UTILIB Concurrent Version System (CVS) repository. The latest release of UTILIB is available at

```
http://www.cs.sandia.gov/~wehart/UTILIB
```

and earlier versions are available in the same directory.

The CVS repository for UTILIB can be accessed by executing

```
cvs -d :ext:GEUutili@gaston.cs.sandia.gov:/usr/local/cvs/cvsroot checkout utilib
```

The password for this repository is 'anonymous'. The developer's password for this repository is restricted; please contact Bill Hart at `wehart@sandia.gov` to request the password to commit changes to this repository. If you are accessing this repository throught a firewall (e.g. Sandia's SRN firewall), or you expect to checkout updates frequently, then the script `cvs-u` can be used to encapsulate the access to the CVS repository. The `cvs-u` script can be downloaded at

```
ftp://ftp.cs.sandia.gov/pub/papers/wehart/src/cvs-shells.tar
```

Note that this script uses the `ssh` command, version 1.x.

8.2 Installation on Unix

Installation of UTILIB on UNIX systems is performed by the following steps:

1. Unpack the archive, unless you have already done that

   ```
   gunzip utilib-$VERSION.tar.gz    # uncompress the archive
   tar xf utilib-$VERSION.tar       # unpack it
   ```

2. Move into the `utilib` directory and run the configure script.

   ```
   ./configure
   ```

   The `configure` script automates much of the setup activity associated with building large suites of programs like UTILIB on various hardware platforms. This includes

   (a) making symbolic links so that files used for configuration can be accessed from one location

   (b) generating Makefiles so that objects, libraries, executables and other 'targets' can be created for specific and unique hardware platforms

   (c) calling itself recursively so that sub-directories can also be configured

   By default, the configure script does not assume that UTILIB relies on any other software libraries. There are a number of configuration options that can be used to customize the installation. The full parameter list for the `configure` script is:

```
configure hosttype [--target=target] [--srcdir=dir] [--rm]
                   [--site=site] [--prefix=dir] [--exec-prefix=dir]
                   [--program-prefix=string] [--tmpdir=dir]
                   [--with-package[=yes/no]] [--without-package]
                   [--enable-feature[=yes/no]] [--disable-feature]
                   [--norecursion] [--nfp] [-s] [-v] [-V | --version]
                   [--help]
```

Many of these options are not necessary since system information can be often acquired from your local machine. Refer to the Cygnus `configure` documentation for complete information (see utilib/doc/configure.ps). The following options are either commonly used or specific to UTILIB (examples of arguments are provided):

[–with-compiler=<gcc,CC>]	Sets up a specific compiler; The native compiler is the default.
[–target=<solaris>]	Optional flag to specify the target machine that you are cross-compiling for.
[–site=<snl980>]	Specifies the site-specific locations for MPI, etc.
[–with-debugging]	Turns on the OPTIMIZATION macro (code is compiled with the -g flag).
[–with-mpi]	Turns on the use of the MPI package.
[–with-mpe]	Turns on the use of the MPE package.
[–with-swig]	Enables the use of swig to wrap UTILIB for use with the Python scripting language.
[–with-static]	Enables the compilation of statically linked libraries (the default).
[–with-insure]	Enables the compilation with the insure++ debugging tool.
[–with-shared]	Enables the compilation of dynamically linked libraries, which can be shared.
[–with-optimization=<level>]	Sets the optimization level used when compiling the source files.
[–with-ansi]	Sets up the compiler to use ANSI standard constructs for C++. (the default)
[–with-ansiheaders]	Creates flags that force the use of ANSI standard C++ header conventions. (the dfault)

The configure script creates Makefiles from `Makefile.in` template files, which outline the basic 'targets' that need to get built. Variables that are package, site or hardware dependent are stored in individual 'fragment' files. These 'fragment' files are added to the custom created Makefiles when users and code developers (recursively) configure this repository with specific host, target, package and/or site parameters.

Running `configure` takes a while, so be patient. Verbose output will always be displayed unless the

user/developer wishes to silence it by specifying the parameter, '–silent'. If you wish to configure only one level/directory, remember to use the option '–norecursion'. All generated "config.status" files include this parameter as a default for easy makefile re-generation; after editing a Makefile.in file, you can construct the associate Makefile file by typing `config.status`.

After the `configure` command is completed, three files will be generated in each configured directory (specified by the file, 'configure.in').

(a) Makefile-${target}
The suffix, ${target}, will depend on the target specified. Native builds have identical host and target values.

(b) Makefile
This will be a symbolic link to the file mentioned above. A user or developer will simply type `make` and the last generated Makefile-${target} will then be referenced.

(c) config.status
A 'recording' of the configuration process (i.e., what commands were executed to generate the makefile). It can be used by the custom makefile to re-generate itself with a command such as this

```
make Makefile.
```

Fragment files exist so that `configure` can support multi-platform environments. UTILIB can be configured for code development and execution on the following platforms :

```
SPARC-SUN-SOLARIS2.5.1   (Sun ULTRAsparc)
MIPS-SGI-IRIX6.4         (SGI Octane)
HPPA1.1-HP-HPUX9.05      (HP 9000/700 series)
PENTIUM-INTEL-COUGAR     (Intel TFLOP supercomputer at SNL)
i686-UNKNOWN-LINUX       (Red Hat 7.1)
```

The fragment files for these platforms and for the packages that UTILIB relies on are located in the `utilib/config` directory. There are five types of files in this directory:

```
mf-<host>-<target>-<site>
Automatically generated by the configure scripts.

mh-<host>
Fragments that define the utilities provided by the host (e.g. the
definition of MAKE.

mp-<target>-<site>
Fragments that define information for the packages that are used by
UTILIB (e.g. MPI).

ms-<site>
Fragments that define the site-specific general configuration
information. If this does not exist for a given site, then the
default ms-default fragment is used.

mt-<target>
Fragments needed to specfy how to compile code for a target
architecture (e.g. compiler name/location).
```

3. Compile the program by running make.

```
make
```

Note that the makefiles in UTILIB may not be portable to all `make` commands. However, they do work with the GNU `gmake` command. The latest file `Makefile-${target}` generated by `configure` will be referenced by this command. The target directory for the library is created for the particular target platform as a subdirectory of utilib/lib.

Prior to making object files header files are linked into the directory `utilib/include`.

4. Optional: Generate the html library documentation.

   ```
   make html
   ```

   This requires the `doxygen` utility.

5. Optional: Generate the postscript version of the user manual.

   ```
   make ps
   ```

   This requires the `doxygen`, `latex`, and `dvips`.

6. Optional: Generate the PDF version of the user manual.

   ```
   make pdf
   ```

   This requires the `doxygen`, `latex`, `dvips` and ghostscript packages.

8.3 Installation on Windows

UTILIB was originally developed under UNIX, but it has been ported to Windows NT using Microsoft's Visual C++ (version 6.0). A MSVC++ project is provided in `utilib/src/vcpp`. This project defines a DLL that will be compiled for UTILIB, and it can be easily included in a user's workspace. The project file relies on the environmental variable 'UTILIB', which is defined from the MS Windows Control Panel under `System/Environment`. This variable should be set to the path of the `utilib` directory. Note: this project file is out of date.

9 Acknowledgements

The genesis of the UTILIB library is in the BBUMS library developed by Bill Hart and Brian Bartell while graduate students at U.C. San Diego. Although Brian would probably not recognize any othe the UTILIB software, the design of some of the most widely used software, like array classes, is due to him. The BBUMS library was subsequently reorganized and renamed the SGOPT library, which focuses on a methods for global optimization. UTILIB was the `stdlib` subdirectory in SGOPT, which was extracted from SGOPT when it became clear that several groups at Sandia would be interested in using the UTILIB components without the additional baggage of the optimizers in SGOPT.

I would like to thank Cindy Phillips, Jonathan Eckstein and Mike Eldred for their input on this software. Each of them has identified numerous bugs, and refinements in the configuration process are largely due to the demands that their uses of UTILIB have made.

The UTILIB library includes several files taken from the RANLIB.C library of C routines for random number generation, which was developed by Barry W. Brown and James Lovato, Dept of Biomathematics at the University of Texas, Houston.

This document was prepared using the Doxygen software documentation tool, developed by Dimitri van Heesch, copyright 1997-2001.

This work was supported in part by the Mathematics, Information and Computational Science program, U.S. Department of Energy, Office of Energy Research. This work was performed at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1996.

[2] M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions, Second Ed.* John Wiley and Sons, Inc., New York, 1993.

[3] Lewis and Denenberg. *Data Structures and Their Algorithms.* Harper Collins, 1991.

[4] Sleator and Tarjan. Self-adjusting binary search trees. *JACM*, 32(3):652–686, July 1985.

[5] M. Weiss and B. Cummins. *Data Structure and Algorithm Analysis.* 1992.

[6] D. Wood. *Data Structures, Algorithms, and Performance.* Addison-Wesley, 1993.