

GUTIL Library Reference

Ben Smith

Introduction

GUTIL is a collection of C++ functions and classes providing technical (i.e. non-scientific) functionality to Lund University's LPJ-GUESS ecosystem modelling framework. It includes templates for a series of dynamic collection classes that function both as linked lists and arrays; a stream input function emulating the FORTRAN read statement; and a class for storing and manipulating character strings. GUTIL can be compiled as a static library or object file and linked in to any C++ program. Compilation should be possible on any platform; it has been tested under Windows, Linux (g++, pgCC, icc) and Unix (Sparc C++). This document provides a synopsis of the functions and classes most likely to be of interest to general users. Additional (and definitive) documentation is provided in the header file `gutil.h`.

Compilation

Linux/Unix: A make file (GNU g++ compiler) is included with the release. This builds `gutil` as a simple object file (`gutil.o`) which may be linked in to any C++ program; e.g.

```
$ g++ xampl.cpp gutil.o
```

Windows: Build `gutil` as a static library (`gutil.lib`). Include the library in the build for any C++ application. Alternatively, include `gutil.cpp` in the build for any C++ application.

All platforms: All of the following `#includes` must appear at the start of any source code file invoking functionality from `gutil`. This includes, of course, the main program. Note that `gutil.h` must appear *last* in the list; the full pathname may be required:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "gutil.h"
```

Stream text input with function `readfor`

```
bool readfor(FILE*& strm, xstring format, ...)
```

Reads in ASCII text data from input stream *strm*, according to a FORTRAN-style format specification given in the string *format*. Addresses of the variables to be assigned to are listed, in order of assignment, in the ellipsis (...) argument list of the function. By default, any characters remaining on the current line are discarded when the statement terminates (this behaviour can be overridden by a \$ specifier in the format string – see below). The function returns true if all specified values could be read in and assigned, or false if an end-of-file

condition prevented some values from being read in and assigned.

Example:

The following code opens for reading a text file called "climate.txt", reads in two values of type double, one integer, and 12 further values of type double, and assigns these values, respectively, to variables lon, lat, elev, and the 12 elements of array mdata. Note that the arguments following the format string are *addresses* of the variables to be assigned to – use the '&' prefix for simple variables; omit the '&' if specifying an array name (a pointer).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <gutil.h>

FILE* in;
double lon,lat;
int elev;
double mdata[12];

in=fopen("climate.txt","rt");
if (!readfor(in,"f6.1,f5.1,i4,12f5.1",&lon,&lat,&elev,mdata))
    printf("Warning: not all values could be read in\n");
```

Format specifiers

A subset of the format specifiers defined for input in the FORTRAN-77 language are supported by function readfor. Note that there are some differences in the way these are implemented compared to FORTRAN. Specifier fields in the format string are normally separated by commas. Each instance of an F, I or A specifier is assumed to correspond to one argument in the ellipsis argument list. Specifier syntax is case-insensitive and spaces and tabs in the format string are ignored.

Important: a value (other than 1) for n (number of items), if specified as part of an F, I or A specification, assumes the values read are to be assigned to consecutive elements of an array, whose starting address is given by a *single* argument in the argument list. Do not use this feature to assign multiple values with the same input format to multiple consecutive arguments.

Floating point specifier: $nFw.dEe$

Reads one or more floating point numbers, assigning each value to a variable of type double.

- n the number of items to read in; if n is not specified, one value is assumed. See cautionary note above.
- w the number of characters to read in for each item; if not specified, characters are read up to the next space, tab, end-of-line, or the next instance on the current line of a separator character appearing immediately after the specifier in the format string (see below).
- d the number of digits in the fractional part of the number; if a decimal point is encountered, this overrides the specified value. If d is omitted and no decimal point is encountered, the input string is interpreted as a whole number.
- e the number of digits in the exponent part of the number; if the character 'E' or 'e' is

encountered, this overrides the specified value. If e is omitted, the exponent 0 (i.e., $10^0 = 1$) is assumed.

Examples:

Input stream: 1234 5.67

<i>format specification</i>	<i>strings read</i>	<i>values assigned</i>
F	"1234"	1234.0
F3	"123"	123.0
F3.1	"123"	12.3
F4.2E1	"1234"	1.23E+04
2F4.2	"1234", " 5.6"	12.34, 5.6

Integer specifier: nIw

Reads one or more integers, assigning each value to a variable of type int.

- n the number of items to read in; if n is not specified, one value is assumed. See cautionary note above.
- w the number of characters to read in for each item; if not specified, characters are read up to the next space, tab, end-of-line, or the next instance on the current line of a separator character appearing immediately after the specifier in the format string (see below).

Examples:

Input stream: 123 4567

<i>format specification</i>	<i>strings read</i>	<i>values assigned</i>
I	"123"	123
I2	"12"	12
2I4	"123 ", "4567"	123, 4567

Character string specifier: nAw

Reads one or more character strings, assigning each to a variable of type xtring.

- n the number of items to read in; if n is not specified, one value is assumed. If reading multiple strings, a width (w) specification must be included (otherwise the rest of the current line is read in and assigned to the first xtring variable; subsequent variables are assigned null strings). See cautionary note above.
- w the number of characters to read for each item; if not specified, characters are read up to the next whitespace character, comma, or the next instance of a separator character appearing immediately after the specifier in the format string (see below).

The read to end-of-line specifier # may be used to read to the end of the current line (see below).

Examples:

Input stream: BERT HIGGINS

<i>format specification</i>	<i>strings read and assigned</i>
A	"BERT"
A2	"BE"
2A6	"BERT H", "IGGINS"
A#	"BERT HIGGINS"

Position specifier: *nX*

Advances one or more characters on the input stream.

n The number of characters to read and discard; if *n* is omitted, a single character is read in and discarded.

End-of-line specifier: /

Advances to the end of the current line on the input stream. Input continues from the start of the next line.

Suppress line feed specifier: \$

If given as the last significant character in the format string, suppresses reading and discarding of the remainder of the current line on the input stream.

Read to end-of-line specifier: #

If specified after a character string specifier, reads to the end of the current line. Use to read in the rest of the current line as a single character string, including white space characters or commas which are otherwise interpreted as separator characters.

Separator character specifier:

Any character other than a space, tab or comma, that cannot be interpreted as part of one of the specifiers above, is interpreted as a separator character, and causes input up to and including the first instance of the specified separator character on the current line. If given immediately following a variable-width F, I or A specification, input continues, for each item, until an instance of the specified separator character is encountered, or the end of the current line is reached (otherwise input continues until a space, tab or the end of the line is reached). If the separator character follows a fixed-width F, I or A specification, or forms a separate specification field, the characters read are discarded.

A comma is interpreted as an *optional* separator character; a variable-width F or I specification followed by a comma results in input of text up to the next space, tab, end-of-

line or comma, *whichever comes first*; a variable-width A specification followed by a comma results in input up to the next comma or end-of-line, whichever comes first.

Examples:

Input stream: 123; Hello World!; 3.142

<i>format specification</i>	<i>strings read</i>	<i>values assigned</i>
I;A;F	"123", " Hello World!", "3.142"	123, " Hello World!", 3.142
F2.1;;I2,A	"12", " 3", ".142"	1.2, 3, ".142"

Input from the keyboard

If input is required from the keyboard instead of a stream attached to a file, function **readfor** can be used specifying `stdin` (one of the default streams defined in `stdio.h`, and normally associated with the terminal keyboard) as the *strm* argument.

Character string manipulation with class `xtring`

Construction and initialisation

The following forms of constructor function are supplied for constructing and initialising new `xtring` objects:

```
xtring()
xtring(char* str)
xtring(char ch)
xtring(int initsize)
```

If a `char*` or `char` argument is supplied, it provides the initial string text; otherwise an empty string ("") is assigned. If the integer argument *initsize* is given, the empty string is assigned to the object, but at least *initsize*+1 bytes of memory are set aside for the object's string buffer (this implies that a string up to *initsize* characters in length can subsequently be assigned to the object without reallocation of memory for the string buffer).

Declare new `xstring` objects using one of the following forms:

```
xtring s; // equivalent to xtring s=""
xtring s="initial text";
xtring s='c';
xtring s(INITSIZE);
```

You can also cast a string or character literal to an `xtring` the usual way:

```
(xtring)"Cast to a xtring"
(xtring)'c'
```

In general, `xtring` objects can be used in place of standard C `char*` strings without explicit casting, e.g.

```
char copy[100];
xstring original="text";
strcpy(copy,original);
```

However, xstring objects must be explicitly casted to char* when specified as arguments in calls to functions with an ellipsis argument, e.g.

```
xstring name="Ben";
printf("My name is: %s", (char*)name);
```

Casting to char* is useful also if you (unwisely?) choose to write directly to the internal string buffer of the xstring object:

```
xstring name(100);
char* pBuffer=(char*)name;
strcpy(pBuffer, "Ben");
```

Note, however, that some of the member functions of xstring can cause the size and memory position of the internal buffer to change. You must ensure that the buffer is at least *size*+1 bytes (characters) in length if writing a string *size* bytes in length to it. The specified minimum size of the buffer is guaranteed if the xstring object is initialised using the xstring(int) constructor (see above) or following a call to member function reserve (see below).

Other public member functions

unsigned long **len()**

Returns the length of the current string in characters (not including a trailing null character); e.g.

```
xstring s="18 characters long";
int result=s.len();
```

xstring **upper()**

Returns a new xstring equivalent to the current one, but with lower-case alphabets 'a'-'z' converted to upper case; e.g.

```
xstring s="the quick brown fox";
xstring t=s.upper(); // t set to "THE QUICK BROWN FOX"
```

xstring **lower()**

Returns a new xstring equivalent to the current one, but with upper-case alphabets 'A'-'Z' converted to lower case; e.g.

```
xstring s="THE QUICK BROWN FOX";
xstring t=s.lower(); // t set to "the quick brown fox"
```

xstring **printable()**

Returns a new xtring equivalent to the current one, but with non-printable characters (ASCII code 0-31) removed.

xtring **left**(unsigned long *n*)

Returns a new xtring consisting of the leftmost *n* characters of the current xtring. An empty string ("") is returned if $n \leq 0$; if $n >$ length of the current xtring, an identical copy of the current xtring is returned; e.g.

```
xtring s="Ben Smith";
xtring t=s.left(3); // t set to "Ben"
```

xtring **mid**(unsigned long *s*, unsigned long *n*)
xtring **mid**(unsigned long *s*)

If both arguments are given, returns a new xtring consisting of up to *n* characters, starting at character number *s* (zero-based) of the current xtring. If $s < 0$, the new string starts at character 0 of the current xtring; if $s \geq$ length of the current xtring, an empty string is returned. If argument *n* is omitted, returns a new xtring consisting of the rightmost portion of the current xtring, starting at character number *s*. If $s < 0$ an identical copy of the current xtring is returned; e.g.

```
xtring s="Wolfgang Amadeus Mozart";
xtring t=s.mid(9); // t set to "Amadeus Mozart"
xtring q=s.mid(9,7); // q set to "Amadeus"
```

xtring **right**(unsigned long *n*)

Returns a new xtring consisting of the rightmost *n* characters of the current xtring. An empty string is returned if $n \leq 0$; if $n >$ length of the current xtring, an identical copy is returned; e.g.

```
xtring s="Ben Smith";
xtring t=s.right(5); // t set to "Smith"
```

long **find**(char* *s*)
long **find**(char *c*)

Returns the position (zero based) of the specified character string or character if it occurs as a substring of the current xtring. If a char* argument longer than one character is given, the returned value is the position of the first character of the substring, if it is found. If there are several occurrences, the position of the leftmost occurrence is returned. Returns -1 if the string or character is not found; e.g.

```
xtring s="abbcd";
int n=s.find('b'); // n set to 1
int m=s.find("bc"); // m set to 2
int q=s.find("de"); // q set to -1
```

long **findoneof**(char* *s*)

Returns the position in this xtring of the first (leftmost) occurrence any character forming part of the string pointed to by *s*. Returns -1 if there are no occurrences; e.g.

```
xtring s="ababcd";
int n=s.findoneof("edc"); // n set to 3
```

long **findnotoneof**(char* *s*)

Returns the position in this xtring of the first (leftmost) character *not* forming part of the string pointed to by *s*. Returns -1 if no such character is found; e.g.

```
xtring s="ababcd";
int n=s.findnotoneof("abc"); // n set to 4
```

double **num**()

Returns the numerical value of the current xtring, if it is a valid representation of a double precision floating point number in C++. Call function `isnum()` to test whether the returned value is meaningful; e.g.

```
xtring pi="3.142";
double v=pi.num(); // v set to 3.142
```

char **isnum**()

Returns 1 if the current xtring is a valid representation of a double precision floating point number in C++, 0 otherwise; e.g.

```
xtring good="3.142";
xtring bad="three point one four two";
double val;
if (good.isnum()) val=good.num();
else val=bad.num();
```

void **printf**(char* *fmt*,...)

A printf-style function for writing formatted data to this xtring object. Equivalent to function `sprintf` in the standard C stream input/output library (`stdio.h`). See any standard C/C++ reference manual for full documentation of printf-style output functions; e.g.

```
xtring out;
char* name[]="pi";
double dval=3.142;
int ndec=3;
out.printf("%s has the value %g to %d decimal places",name,dval,ndec);
// out set to "pi has the value 3.142 to 3 decimal places"
```

void **reserve**(unsigned long *n*)

Expands or contracts the memory allocation to the current xtring object to accommodate a string at least *n* characters in length (not including the trailing null byte). The currently stored string may be copied to a new location in memory but is not deleted. This function may be useful if you intend to write directly to the internal string buffer, whose address is returned by casting the xtring object to `char*`; e.g.

```
xtring s;
s.reserve(100);
strcpy(s, "A string up to 100 characters long");
```

In general, however, there should be no reason to write directly to the internal string buffer of an xtring object; use the assignment (=) operator (see below) to assign a new value to the object.

Overloaded operators

The following operators are defined for xtring objects:

Assignment: =, +=

Concatenation: +, +=

Comparison: ==, !=, <, >, <=, >=

Array subscript: []

Operator functionality is described here mainly by code examples. The examples below assume the following data types for variables:

```
xtring x1,x2,x3;
char* s;
char c;
unsigned long n;
```

Assignment:

```
x1="Ben";
x1+=" Smith"; // x1 set to "Ben Smith"
```

Concatenation (appends a char* string, xtring or character to the end of an xtring string):

```
x1="Wolfgang ";
x2=" Mozart";
c='A';
x3=x1+c;
x3+=x2+" (composer)";
// x3 set to "Wolfgang A Mozart (composer)"
```

Concatenation of, for example, two char* strings, or a xtring to the end of a char* string, is possible by casting one of the operands to an xtring:

```
x1=(xtring)"Wolfgang "+"Mozart";
x1=(xtring)"Wolfgang"+x2;
```

Comparison (== and != compare string identity; <, >, <=, >= compare "alphabetic" rank):

```
x1==x2 && x2!=x3 || x1<s || x1>c || x2<=x3 || x3>=x1
```

Note that the left hand operand must be a xtring (or casted to xtring)

Array subscript ($x[n]$ retrieves a reference to the n th character [zero-based] within xtring x):

```
c=x1[n];
```

```
x1[n]=c;
```

The size of the internal string buffer is expanded if necessary to ensure that the specified subscript is valid (points to a character position within the internal string buffer). However, the string itself is not expanded (i.e. the position of the trailing null byte, signifying the end of the string, is not changed).

Collection class **ListArray**

ListArray is provided as a template that may be used to produce a dynamic collection of any C++ object type. The collection behaves both as a double linked list and an array. The only restriction is that the object class to be stored must have a valid default constructor.

Declare a type using this template as either:

```
typedef ListArray<MyObjectType> MyCollectionClass;
```

or (for collection classes containing additional members, apart from the ones inherited from ListArray):

```
class myclass : public ListArray<MyObjectType>
{ ... };
```

where `MyObjectType` is any C++ class, struct, union, or simple type (int, float, double etc.). Either of the above declarations will produce a class including the following public functions and member variables:

void **initarray**(unsigned int *nitem*)

Clears the list array (if not empty) and fills it with *nitem* `MyObjectType` objects.

`MyObjectType&` **createobj**()

Creates a new object of type `MyObjectType` and returns a reference to it.

bool **firstobj**()

Causes the internal object pointer to point to the first `MyObjectType` object in the list array. Returns false if the list array is empty.

bool **isobj**

This variable (NB: not a function) is true whenever the internal object pointer points to a `MyObjectType` object, false otherwise (including when the list array is empty).

bool **nextobj**()

Causes the internal object pointer to point to the next `MyObjectType` object in the list array. Returns false if the last item has already been reached.

`MyObjectType&` **getobj**()

Returns a reference to the object currently pointed to by the internal object pointer. Do not call this function unless the pointer is pointing to a valid object (**isobj**=true).

unsigned int **nobj**

The number of objects currently stored in the list array.

void **killobj()**

Removes the MyObjectType object currently pointed to by the internal pointer.

void **killall()**

Clears the entire list array, releasing dynamic memory.

MyObjectType& **operator[]**(unsigned int *i*)

Overload of the array brackets operator which returns a reference to the *i*'th MyObjectType item in the list array. Do not call unless the internal object pointer is pointing to a valid object. Note that this function does NOT affect the value of the internal pointer.

It is possible to loop sequentially through all items in the list array using code like the following:

```
mycollection.firstobj();
while (mycollection.isobj) {
    MyObjectType& obj=mycollection.getobj();
    // NB: '&' required unless query only
    /* query or modify object obj here */
    mycollection.nextobj();
}
```

or alternatively:

```
for (i=0;i<mycollection.nobj;i++) {
    MyObjectType& obj=mycollection[i];
    /* query or modify object obj here */
}
```

Estimating elapsed and remaining time with class Timer

The computer model for which gutil was developed can sometimes take many hours to complete a simulation. It is desirable for users to obtain an ongoing estimate of progress and remaining simulation time. This class provides the necessary functionality.

The following public member functions are provided:

void **init()**

Should be called before any other operations on a Timer object.

void **settimer**([double *duration*])

Commences timing: optional parameter duration gives number of seconds to "finished" status.

void **setprogress**(double *progress*)

Specifies fractional progress (in range 0-1) towards "finished" status (modifies time remaining to "finished")

double **getprogress**()

Returns fractional progress (in range 0-1) towards "finished" status

The following public data members are available (should be *queried* only):

int elapsed.hours	elapsed full hours since timer set
int elapsed.minutes	elapsed full minutes past the hour since timer set
int elapsed.seconds	elapsed full seconds past the minute since timer set
int elapsed.milliseconds	elapsed milliseconds past the second since timer set
char* elapsed.str	elapsed time in the form "hh:mm:ss"
int remaining.hours	full hours remaining to "finished" status
int remaining.minutes	full minutes past the hour remaining
int remaining.seconds	full seconds past the minute remaining
int remaining.milliseconds	milliseconds past the second remaining
char* remaining.str	elapsed time in the form "hh:mm:ss"

Sample program:

```
void slow(int i) {
    /* Do some number crunching */
}

int main() {
    const int STEPS=1e10;
    const int REPORT=1e9;
    int i;

    Timer t;
    t.init();

    t.settimer();

    for (i=0;i<STEPS;i++) {
        t.setprogress(double(i)/double(STEPS));

        if (!(i%REPORT)) {
            printf("Elapsed time: %s\n",t.elapsed.str);
            printf("Remaining time: %s\n",t.remaining.str);
        }

        slow(i);
    }

    return 0;
}
```

Function `unixtime`

void `unixtime`(xstring& *result*)

Writes the date and time as a character string to *result*; e.g.

```
Tue Nov 06 10:17:47 2001
```

Contact information

Direct any queries to:

Dr Ben Smith
Dept of Physical Geography and Ecosystems Analysis
Lund University
Geocentrum II
22362 Lund
Sweden

E-mail: ben.smith@nateko.lu.se

URL: www.nateko.lu.se/personal/benjamin.smith

22 July 2006.