INILIB 1.0.7b3

http://inilib.sourceforge.net/

March 14, 2005

Brian Barrett (bwbarrett@users.sourceforge.net)
Jeff Squyres (jsquyres@users.sourceforge.net)
Andrew Lumsdaine (lums@lsc.nd.edu)

Contents

List of Figures

List of Tables

1 Introduction

inilib is a C++[2] library which provides a convenient mechanism for saving the "state" of a program in the well-known ".ini file" format used in Microsoft Windows^{TM} [3]. inilib gives the programmer a means of storing a number of arbitrary settings in memory with an easy access interface, as well providing means for saving the information to and loading it from a file on disk. Data is stored in an easy to read format, allowing the user to modify any of the information with a simple text editor (i.e., outside the scope of inilib).

inilib benefits the programmer by providing a simple, intuitive means to store any data that can be expressed in std::string, int, double, or bool types. In addition, inilib handles any conversions that may be necessary to convert from one type to the other. Saving all information stored in inilib to disk or loading information from disk requires only one function call.

inilib provides a hierarchy for access of information, in order to provide the most flexibility for programmers. The top level is called the registry, where each registry corresponds to one file. Inside of a registry, there are zero or more sections. A section contains zero or more attributes, each of which actually store the information. Each section is intended to contain all the information for a particular topic. For example, a text editor program might have a section for window location and size, and another section containing the last documents opened by the user. Figure 1 shows a possible .ini file for such a program.

```
[window settings]
width = 400
height = 500
xpos = 10
ypos = 5

[history]
file1 = /home/bbarrett/document1.txt
file2 = /home/bbarrett/document2.txt
file3 = /home/bbarrett/document3.txt
```

Figure 1: An example file produced by inilib.

Using the information provided in Figure 1 is simple. An example implementation that prints out the last file opened (assuming that file1 always points to the last file opened) is shown in Figure 2. The registry constructor loads the file sample.ini, from Figure 1. The next line demonstrates accessing a piece of information from inilib. The string in the first [] gives the section to access, and the string in the second [] specifies which attribute to return.

2 inilib Overview

The library implements a three-level hierarchy of registry → section → attribute. A registry contains zero or more sections. A section contains zero or more attributes. Each attribute is a single (key, value) pair. To avoid name conflicts, the entire inilib package is in the INI namespace.

The registry is the top-most level of the data structure. It provides methods for accessing the sections it contains, as well as reading and writing its sections from disk. The section class provides access to its attributes. Attributes are classes that actually contain the data to be stored. To access a particular attribute, the programmer would use the syntax (where section_name and attribute_name are both of type std::string):

```
#include <iostream>
#include <string>
#include "inilib.h"

using namespace std;

int
main(int argc, char *argv[])
{
  // Create the object and read the file
  INI::registry information("sample.ini");

  // Output a given attribute
  cout << information["history"]["file1"] << endl;

  return 0;
}
```

Figure 2: Printing out the last file (`file1`) opened in the example text editor.

```
registry_name[section_name][attribute_name]
```

Attributes can store of the following C/C++ types: `double`, `int`, `std::string`, and `bool`. The interface is designed to mimic the STL `map` [4] as closely as possible.

The `registry` class provides an interface to the sections stored in the program. Sections can be added by using `operator[]()` or `insert()`, similar to the way a new element is added to an STL `map`. `operator[]()` can be used to retrieve a section. Iterators are provided to traverse the sections contained in the registry. An STL-like `find()` function is also provided to locate a specific section. Finally, `file_read()` and `file_write()` read and write all of the registry's sections.

The `section` class operates in a manner similar to the `registry` class, with the exception that it does not contain the file functions found in the registry class.

The attributes, which actually store data for the user, provide interfaces for assignment (by overloading of `operator=()`) from any of the C/C++ types `bool`, `double`, `int`, and `std::string`. In addition, attributes can be cast to any of these types. The common unary and binary operators are overloaded for the `attribute` class in order to remove any possible compiler ambiguities that might arise. Section 4.3 provides more information on the overloaded operators.

Figure 3 provides a brief example of the general use of `inilib`. The program will create a `registry` and populate it with the default values for the sample text editor from Section 1. The file `options.ini` is then loaded to import the user's options into the `registry`. The program then demonstrates changing an `attribute`'s value.

Figure 3 also demonstrates the connection between a file and a registry. The constructor associates the filename `user.ini` with `information`, and instructs `inilib` not to load the information in `user.ini` immediately, but to write the `registry` to `user.ini` when the destructor is called. The topic is covered further in Section 3.2.

3 Interface

This section describes the interface provided by `inilib`.

Many of the methods provided in the `inilib` package take one of four data types (`bool`,

```
#include <iostream>
#include <string>
#include "inilib.h"

using namespace std;

int
main(int argc, char *argv[])
{
  INI::registry information("user.ini", false, true);

  // Provide default values for many of the fields.
  information["window settings"]["width"] = 300;
  information["window settings"]["height"] = 300;
  information["history"]["file1"] = "";
  information["history"]["file2"] = "";
  information["history"]["file3"] = "";

  // Now, load the user's information
  information.file_read();

  // And pretend the user just changed the screen width
  information["window settings"]["width"] = new_width;

  return 0;
  // Note that the registry is written out upon exit
}
```

Figure 3: A sample program loading data from a file.

`double`, `int` and `std::string`) as an argument. For example, the `section.insert()` has four prototypes:

```
void insert(const std::string& key, bool value);
void insert(const std::string& key, double value);
void insert(const std::string& key, int value);
void insert(const std::string& key, std::string value);
```

For clarity, since `inilib` frequently provides overloaded functions for use with any of the four supported types, the type will be denoted by the Smallcaps font, which denotes one of the four (`int`, `double`, `std::string` and `bool`) supported types:

```
void insert(const std::string& key, DATATYPE value);
```

The term "target" is used below to mean the object upon which the member function is invoked upon. The term "source" typically refers to the argument of the member function.

Additionally, the term "deep copy" is used to denote a copy where all data that is included in an object is copied to the target. This means that after the copy, there are two distinct copies of the data (as opposed to two objects that refer to the same underlying data).

3.1 Namespace

All classes, methods, and operators in the `inilib` library are contained in the `INI` namespace.

3.2 `registry` Class

As described in Section 2, `inilib` is designed such that all access to data must begin with an instance of the registry class.

The `registry` class is directly associated with a filename, which will be used to populate the `registry` or save the information in the `registry` to disk. Constructors are provided to populate a `registry` immediately upon creation. In addition, it is possible to have a `registry` write itself to disk when its destructor is called.

All of the member functions in the `registry` class are summarized in Table 1.

Name	Purpose
Default constructor	Create an empty registry.
Copy constructor	Perform a deep copy.
Other constructor	Associate a filename with the registry, and optionally set read-upon-construction and write-upon-destruction flags.
Destructor	Destroy the registry and all associated data.
`operator=`	Assignment operator; clear the registry and perform deep copy.
`operator+=`	Perform a deep copy/append.
`operator[]`	Return a `section`.
`insert`	Insert a new `section`.
`clear`	Erase all data in the registry.
`find`	Return an iterator to a `section`.
`empty`	Return `true` if there are no sections, `false` otherwise.
`begin`	Return iterator to beginning of the registry.
`end`	Return iterator past the end of the registry.
`file_read`	Read in a specified (or implied) `.ini` file.
`file_write`	Write out a specific (or implied) `.ini` file.
`set_filename`	Associate a filename with the registry.
`get_filename`	Get the filename that is associated with the registry.
`set_write_on_destruct`	Set whether the registry will be written out upon destruction.
`get_write_on_destruct`	Get whether the registry will be written out upon destruction.

Table 1: Member functions in the `registry` class.

3.2.1 Convenience typedefs

Since the `registry` class stores its sections in an STL `map`, iterator access is provided to traverse all the sections in a `registry`. To that end, the following typedefs are provided by the `registry` class:

- `typedef std::map<std::string, section&>::iterator iterator;`
- `typedef std::map<std::string, section&>::const_iterator const_iterator`

These typedefs can be used similar to STL iterators. An example is given in Figure 4.

```
#include <iostream>
#include <string>
#include "inilib.h"

using namespace std;

int
main(int argc, char *argv[])
{
  INI::registry reg("user.ini");
  // Populate the registry
  reg["window settings"]["width"] = 300;
  reg["window settings"]["height"] = 300;
  reg["history"]["file1"] = "";
  reg["history"]["file2"] = "";
  reg["history"]["file3"] = "";

  // print out the name of every section in the
  // registry.
  for (registry::iterator i = reg.begin() ;
       i != reg.end() ;
       ++i)
    cout << (*i).first << endl;

  return 0;
}
```

Figure 4: Using the iterator typedefs provided by the registry class.

3.2.2 Constructors

`registry()`

`registry(const registry&)`

`registry(const std::string& file, bool fileread = true, bool filewrite = true)`

Creates an instance of the `registry` class. `file` is a filename to associate with the instance of `registry`. If `fileread` is true, the constructor will call `file_read()`. If `filewrite` is true, the contents of the target `registry` will be written to `file` when the destructor is called (Section 3.2.3). The associated filename can be obtained with the `get_filename()` method (Section 3.2.16) and changed with the `set_filename()` method (Section 3.2.15). The behavior of the write on destruct option can also be controlled with `get_write_on_destruct()` (Section 3.2.18) and `set_write_on_destruct()` methods (Section 3.2.17).

3.2.3 Destructor

`~registry()`

Eliminates the instance of the `registry` class. All `sections` and `attributes` associated with the particular `registry` are destroyed as well. If `get_write_on_destruct()` returns true, the contents of the target `registry` will be saved to disk before the instance is deleted.

If the write fails, no notification will be given to the user.

3.2.4 Assignment Operator

`registry& operator=(const registry& r)`

Assigns `r` to the target registry. This function first deletes all information in the target registry, and makes a deep copy of the information from `r`, including the value of the `write_on_destruct` tag, into the registry.

A reference to the target `registry` is returned.

3.2.5 `operator+=`

`registry& operator+=(const registry& r)`

Adds the information from `r` to the target registry. If a section in `r` does not exist in the target registry, the section will be deep copied into the target (including all attributes in `r`). The value of the `write_on_destruct` tag will not be copied from `r`.

If a section in `r` already exists in the target registry, all attributes from the section in `r` will be deep copied into the section in the registry. If an attribute in the registry exists already, the attribute in `r` will overwrite it.

A reference to the target registry is returned.

3.2.6 `operator[]`

`section& operator[](const std::string& key)`

Returns section referenced by key `key` from the registry. If the key `key` does not already exist in the registry, an empty section is created, stored in the registry (which can be accessed in the future by the key `key`), and a reference to it is returned.

3.2.7 `insert`

`void insert(const std::string& key, const section& sec)`

`void insert(const registry& r)`

Inserts information into the target registry by deep copying the source information.

If `r` is passed to the function, `insert` works in the same way as `operator+=` (see Section 3.2.5).

If `key`, `sec` are passed to the function, `insert` adds the section's information to the target registry. If `key` does not exist in the registry, it will be created. If it does exist, all attributes will be copied into the existing section. If a section already exists with key `key`, `section::operator+=` (Section 3.3.5) is used to combine the two sections.

3.2.8 `clear`

`void clear()`

Removes all information from the registry. All sections (and all attributes in each section) are deleted.

3.2.9 `find`

`iterator find(const std::string& key)`

Returns an iterator pointing to a (name, section) pair with name `key`. The section name is the `first` element of the pair, and is a `std::string`; the `second` element of the pair is a reference to the `section` of that name. If there is no section in the registry with the name `key`, an iterator equal to `registry::end()` (see Section 3.2.12) is returned.

3.2.10 `empty`

`bool empty()`

Returns `true` if there are no sections in the target registry, `false` otherwise.

3.2.11 `begin`

`iterator begin()`

Returns the iterator returned by the underlying `std::map`'s `begin()` function. The iterator points to (name, section) pairs, as described in the `find()` function (Section 3.2.9).

3.2.12 `end`

`iterator end()`

Returns the iterator returned by the underlying `std::map`'s `end()` function.

3.2.13 `file_read`

`bool file_read()`

`bool file_read(const std::string& filename)`

If no argument is provided, attempts to read the file associated with the target registry, either through the constructors (Section 3.2.2) or `set_filename()` (Section 3.2.15).

If an argument is provided, populates the registry with data found in the specified file.

In both cases, information from the file is *appended* to the target registry. Any attributes contained in both the registry and the file will be overwritten with the values from the file. If a section is not already in the target registry, it will be added to the target registry. Likewise, if an attribute is not already in the registry, it will be added. See Section 2 for an in-depth explanation of the underlying implementation of attributes.

Returns `true` on success, `false` otherwise.

3.2.14 `file_write`

`bool file_write()`

`bool file_write(const std::string& filename)`

If no argument is provided, attempt to write to the filename associated with the target registry (either through the constructors (Section 3.2.2) or `set_filename()` (Section 3.2.15).

If the filename argument is provided, output the data contained in the registry to the specified file. Any information already in the file will be overwritten.

Returns true on success, false otherwise.

3.2.15 `set_filename`

`void set_filename(const std::string& name)`

Changes the filename associated with the target registry to `name`.

3.2.16 `get_filename`

`std::string get_filename()`

Returns a string containing the filename currently associated with the target registry.

3.2.17 `set_write_on_destruct()`

void `set_write_on_destruct(bool value)`

If `value` is true, the target registry will attempt to write itself to the file associated with it upon destruction. If `value` is false, it will not attempt to write itself on destruction.

3.2.18 `get_write_on_destruct()`

`bool get_write_on_destruct()`

Returns `true` if the target registry will attempt to write itself to the file associated with it upon destruction. Returns `false` otherwise.

3.3 `section` class

The `section` class is designed to be contained in a registry. It contains attributes, which are the actual data stored in a registry. Table 2 summarizes the function members of the `section` class.

Name	Purpose
Default constructor	Create an empty section.
Copy constructor	Perform a deep copy.
Destructor	Destroy the section and all associated data.
`operator=`	Assignment operator; clear the section and perform deep copy.
`operator+=`	Perform a deep copy/append.
`operator[]`	Return a `section`.
`insert`	Insert a new `section`.
`clear`	Erase all data in the section.
`find`	Return an iterator to a `attribute`.
`empty`	Return `true` if there are no attributes, `false` otherwise.
`begin`	Return iterator to beginning of the section.
`end`	Return iterator past the end of the section.

Table 2: Member functions in the `section` class.

3.3.1 Convenience typedefs

Since the section class stores its attributes in an STL `map`, iterator access is provided to traverse all the attributes in a section. To that end, the following typedefs are provided by the `section` class:
`std::map<std::string, attribute&>::iterator iterator`

`std::map<std::string, attribute&>::const_iterator const_iterator`

3.3.2 Constructors

`section()`

`section(const section& s)`

Creates an instance of the `section` class. If a section `s` is given, `s` is deep copied (along with all attributes in `s`) into the target section.

3.3.3 Destructor

`~section()`

Eliminates the instance of the `section` class. All `attribute`s in the section will be destroyed as well.

3.3.4 Assignment Operator

`section& operator=(const section& s)`

Deep copies section `s`. All attributes in the target section before calling `operator=` are deleted from the section. Copies of all attributes in section `s` are added to the target section.

3.3.5 `operator+=`

`section& operator+=(const section& s)`

Deep copies section `s`. All attributes in the section before calling `operator+=` are kept in the section. In the event that there is an attribute in `s` with the same key as an attribute in `this`, the attribute in `s` will overwrite the current attribute.

3.3.6 `operator[]`

`attribute& operator[](const std::string& key)`

Returns the attribute with key `key`. If no such attribute exists in the section, a default attribute is returned (see Section 4.2).

3.3.7 `insert`

`void insert(const std::string& key, DATATYPE value)`

`void insert(const section& sec)`

When `insert` is called with parameter `sec`, `insert` acts like `operator+=` (see Section 3.3.5).

When `insert` is called with parameters `key` and `value`, the attribute is added to the target section. If an attribute with key `key` already exists, it is overwritten.

3.3.8 `clear`

`void clear()`

Removes all information from the `section`. All `attribute`s are deleted.

3.3.9 `find`

`iterator find(const std::string& key)`

Returns an iterator pointing to the (name, attribute) pair with name `key`. The attribute name is the `first` element of the pair, and is a `std::string`; the `second` element of the pair is a reference to the `attribute` of that name. If there is no attribute in the section with the name `key`, an iterator equal to `section::end()` (see Section 3.3.12) is returned.

3.3.10 `empty`

`bool empty()`

Returns `true` if there are no `attribute`s in the target section. Returns `false` otherwise.

3.3.11 `begin`

`iterator begin()`

Returns the iterator returned by the underlying `std::map`'s `begin()` function. The iterator points to (name, attribute) pairs, as described in the `find()` function (Section 3.2.9).

3.3.12 `end`

`iterator end()`

Returns the iterator returned by the underlying `std::map`'s `end()` function.

3.4 `attribute` Class

The `attribute` class is a base class from which four classes are derived (see Section 4.2). The constructors and destructors will rarely be used directly by the programmer – the `registry` and `section` classes are intended to act as the primary interface to attributes.

Table 3 summarizes the members functions on the `attribute` class.

Name	Purpose
Default constructor	Create an empty attribute.
Copy constructor	Perform a deep copy.
DATATYPE constructor	Intializer constructor.
Destructor	Destroy the section and all associated data.
`operator=`	Assignment operator; clear the section and perform deep copy.
Casting operators	Convert the attribute to an `bool`, `int`, `double`, or `string`.
Unary operators	As appropriate (see Section 4.3).
`get_type`	Return an enum indicating the underlying attribute's real type.
`get_precision`	Get the precision that will be used to convert doubles to strings.
`set_precision`	Set the precision that will be used to convert doubles to strings.

Table 3: Member functions in the `attribute` class.

3.4.1 Type Enumeration

Each attribute has a type associated with it. The `get_type()` function (see Section 3.4.7) can be used to determine the type of an attribute. The following enumeration is used to determine type:

```
enum attr_type {BOOL, DOUBLE, INT, STRING, NONE};
```

3.4.2 Constructors

`attribute()`

`attribute(attribute*)`

`attribute(const attribute&)`

`attribute(DATATYPE value)`

Creates an attribute. For each of the four attribute types, `value` can only be the same type as the attribute. For example, `bool_attribute` only has a constructor with `value` of type bool.

3.4.3 Destructors

`~attribute()`

Destroys the attribute, freeing any memory associated with it. `wrap_attributes` also destroy the underlying `attribute`, if any (See Section 4.2).

3.4.4 Assignment Operator

`attribute& operator=(DATATYPE value)`

Assigns `value` to the attribute. For more information on how conversions are handled by `inilib`, see Section 4.2.1. `value` will be cast to the type of the underlying attribute, which may cause a loss of data (such as assigning a `double` to any of the other three data types). The precision of a `double` assigned to a `std::string` can be modified using the `set_precision()` and `get_precision()` methods (Section 3.4.8).

3.4.5 Casting Operator

`operator DATATYPE()`

Casts the value of the attribute to the specified type. However, with `operator=()`, certain casts can cause a loss of data (such as casting a `double` to an `int`). Form more information on how conversions are handled during casting, see Section 4.2.1.

As the casting operator is defined for `int`, `double`, `bool`, and `std::string`, compiler ambiguities can arise when using attributes for many common operations (`operator==()`, for instance). To avoid the problem of compiler ambiguities that arise because of the `operator=()` being overloaded, many of the operators have been overloaded for the attribute classes. More information is available in Section 4.3.

3.4.6 Other Overloaded Operators

In order to eliminate compiler ambiguities, many of the overloadable C++ operators are overloaded for the `attribute` classes. See Section 4.3 for more information.

3.4.7 get_type

`attr_type get_type()`

Returns the underlying type of the attribute. If the attribute is a `wrap_attribute`, the result will be the type of the underlying attribute. If an attribute has not yet been assigned a type, `NONE` is returned.

3.4.8 Double Precision Setting

`int attribute::get_precision()`

`void attribute::set_precision(int precision)`

Certain functions require that a `double` be assigned or cast to a `std::string`. The `set_precision()` function is used to set the number of significant figures that will be stored after the conversion. The `get_precision()` function allows access to the current precision for the conversion. Both functions are `static` to the `attribute` class, meaning that the precision level is for the entire class, not specific instances.

`precision` should be no higher than 100. If `precision` is higher than 100, it will automatically be reduced to 100 without warning. The result of `get_precision()` in this case will be 100.

4 Implementation

4.1 registry and section

The `registry` and `section` classes each contain an STL `map` with `(std::string)` type for keys and values of type `(section*)` and `(attribute*)`, respectively. The `operator[]()`, `insert()`, and `clear()` functions provided by `registry` and `section` perform error checking then call the corresponding `map` functions. All functions in the two classes, except for the file access functions `file_read()` and `file_write()`, are inlined to increase the performance of the library.

4.2 attribute

The implementation of attributes is slightly more interesting than that of the `registry` and `section`. There is a base class, `attribute`, from which five other classes are derived:

1. `bool_attribute`
2. `double_attribute`
3. `int_attribute`
4. `string_attribute`

The four derived classes, `bool_attribute`, `double_attribute`, `int_attribute`, and `string_attribute` (page 14), are very similar. They provide casting operators to `bool`, `double`, `int`, and `std::string`, as well as `operator=()` from these four types. This allows the seamless assignment and casting of any attribute type to any other attribute type, similar to the Perl [5] and PHP [1] scripting languages.

The attribute class is used in situations where the type of the attribute is not known at the time the attribute is created, for the return type of many of the overloaded operators, and for storage in the `section` class. The `attribute` class provides the same interface as the four classes that derive from it. However, instead of containing an actual data value, the `attribute` class contains a pointer to one of the other four classes. The interaction between the `attribute` class and the derived classes is completely transparent to the programmer.

Casting to	Casting from bool	double	int	std::string
bool	NA	(bool) value	(bool) value	false if string is empty, true otherwise.
double	(double) value	NA	(double) value	atof(value)
int	(int) value	(int) value	NA	atoi(value)
std::string	"true" if value is true, empty string otherwise	sprintf(ret, ''%.PRECISIONlf'', value)	sprintf(ret, ''%.PRECISIONlf'', value)	NA

Table 4: Conversion behavior in inilib

The library is intended to be used in a situation where the programmer provides a set of default attributes and values within the program. When reading attributes from a file, entries from the file override the values in memory. However, the file value's type is assumed to be the same as the attribute in the memory. If an attribute exists in the file being loaded, but is not already in memory, the parser must make a best-guess as to the type of attribute to create. If the value is whitespace, followed by an optional + or -, a series decimal digits, a decimal point, then a series of decimal digits, followed by any amount of whitespace, inilib will consider the value to be of type double. If the value is any amount of whitespace, followed by one or more decimal digits, followed by any amount of whitespace, inilib will consider the value to be of type int. Otherwise, inilib will consider the value to be of type std::string.

4.2.1 Data Type Conversion

Whenever possible, conversion from one data type to another is handled by casting to the desired data type. For example, the conversion between from an int to bool is handled by calling (bool) value. In certain circumstances, a cast will not produce the desired result, such as casting a std::string to a double. Table 4 details the conversions used in inilib.

4.3 Attribute Operator Overloading

Due to the overloading of casting operators for the attribute classes (allowing an attribute to be cast to one of bool, double, int, or std::string), compiler ambiguities can arise. Two solutions to the problem of compiler ambiguties exist. The first is to require the programmer to explicitly cast the attribute in any operation that may result in a compiler ambiguity. This option requires more work for the programmer, which is not what a library should do. The second option is to overload the most common operators for the attribute class, eliminating any compiler ambiguities that may arise. The second option is obviously a better solution for the programmer, and is therefore implemented in inilib. Table 5 provides a listing of the operators overloaded in inilib.

4.3.1 Binary Operators

The overloaded binary operators fall into two categories: attribute/DATATYPE functions and attribute/attribute functions. The attribute/DATATYPE functions will cast the attribute to the same type as the other argument and then perform the operation.

The attribute/attribute binary operators do not have an implied type to cast to, as an attribute can be cast to any one of bool, double, int, or std::string. Therefore, the

pre increment	++lvalue
pre decrement	--lvalue
not	!expr
unary minus	-expr
multiply	expr * expr
divide	expr / expr
modulo	expr % expr
add (plus)	expr + expr
subtract (minus)	expr - expr
less than	expr < expr
less than or equal	expr <= expr
greater than	expr > expr
greater than or equal	expr >= expr
equal	expr == expr
not equal	expr != expr
multiply and assign	lvalue *= expr
divide and assign	lvalue /= expr
module and assign	lvalue %= expr
add and assign	lvalue += expr
subtract and assign	lvalue -= expr

Table 5: Overloaded operators provided by `inilib`.

`attribute`/`attribute` functions use the following hierarchy for casting (listed from lowest to highest):

1. `std::string`
2. `bool`
3. `int`
4. `double`

The `attribute` with the higher type acording to the hierarchy will be cast to the type of the other `attribute`. For example, if the comparison operator is called on a `string_attribute` and `double_attribute`, the casting operator will be used to cast both attributes to type `double`, then the operation will be performed. Although an arbitrary hierarchy, it is used because it is similar to the one used by Perl and PHP to control casting from one type to another.

The `attribute`/`attribute` binary arithmetic operators create a temporary attribute of the appropriate type (as discussed in the previous paragraph) and use the arithmentic operation and assign member functions to perform the arithmetic operation. Therefore, the behavior of the `attribute`/`attribute` binary arithmetic operators is determined by the operation of the member arithmetic operation.

4.3.2 Unary Operators

The unary operators are all member functions of the `attribute` class. Some of the unary operators (pre increment, for instance), are not defined in C/C++ for some data types (double, in the case of the pre increment operator). However it would be inconvienent not to have the operators defined for all four types used by `inilib`. Table 6 describes the result of many unary operators when called on an `attribute`. The action of these operators is intended to mimic Perl and PHP as closely as possible.[1]

The post-increment and post-decrement operators are not provided, as it is not possible to return an object of type `attribute`, which would be required for the post-increment and post-decrement operators.

Table 6: Behavior of unary operators overloaded in `inilib`.

	`attribute` type			
	`bool`	`double`	`int`	`std::string`
pre increment	sets the value of the attribute to `true` and returns the new value	adds `1` to the value of the attribute and returns the new value	adds `1` to the value of the attribute and returns the new value	If the value of the attibute is an int or double,[2] the value is cast to that type and `1` is added to the value. The result is converted back to a string. The resulting string is assigned to the attribute and returned.
pre decrement	sets the value of the attribute to `false` and returns the new value	subtracts `1` from the value of the attribute and returns the new value	subtracts `1` from the value of the attribute and returns the new value	if the value of the attibute is an int or double, the value is cast to that type and `1` is subtracted from the value. The result is converted back to a string. The resulting string is assigned to the attribute and returned.
not	returns `true` if value is `false`, returns `false` if the value is `true`	returns `true` if the value is `0`, returns `false` otherwise	returns `true` if the value is `0`, `false` otherwise	returns `true` if the string is empty, `false` otherwise
unary minus	same as `not`	returns value, but with sign reversed	returns value, but with sign reversed	if the string contains only a number (integer or decimal), returns the value as a string, but with a '-' prepended. Otherwise, returns the current value

continued...

[1]The one exception is integer and string addition. The incrementing of the alphanumerical string is not supported in `inilib`

[2]Meaning it contains a string of the form dddd or ddd.ddd, where d is a decimal digit, with an arbitrary number of digits.

Table 6: (cont.)

	`bool`	`double`	`int`	`std::string`
multiply and assign	casts argument to `bool` and performs a logical `and`, returning the result and assigning it to the attribute	casts argument to `int` and multiplies, returning the result and assigning it to the attribute	casts argument to `double` and multiplies, returning the result and assigning it to the attribute	If the argument is a `bool`, `double`, or `int`, casts attribute to that type and performs the multiplication. The result is converted back to a string, returned, and assigned to the attribute. If the argument is a string, the `std::string`'s + operator is used. If the argument is an attribute, the action is as above, based on the underlying type.
divide and assign	casts argument to `bool` and performs a logical `or`, returning the result and assigning it to the attribute	casts argument to `int` and divides, returning the result and assigning it to the attribute	casts argument to `double` and divides, returning the result and assigning it to the attribute	If the argument is a `bool`, `double`, or `int`, casts attribute to that type and performs the division. The result is converted back to a string, returned, and assigned to the attribute. If the argument is a string, the operation returns the current string. If the argument is an attribute, the action is as above, based on the underlying type.
				continued...

Table 6: (cont.)

	`bool`	`double`	`int`	`std::string`
modulo and assign	casts argument to `bool` and performs a logical `and`, returning the result and assigning it to the attribute	casts argument to `int` and performs modulo operation, returning the result and assining it to the attribute.	casts argument and value to `int` and performs modulo operation. Result is cast back to a `double`	If the argument is a `bool`, `double`, or `int`, casts attribute to that type and performs the modulo, using the rules specified for the `bool`, `double`, and `int` attributes. The result is converted back to a string, returned, and assigned to the attribute. If the argument is a string, the operation returns the current string. If the argument is an attribute, the action is as above, based on the underlying type.
add and assign	casts argument to `bool` and performs a logical `or`, returning the reuslt and assigning it to the attribute	casts argument to `int` and adds, returning the result and assigning it to the attribute	casts argument to `double` and adds, returning the result and assigning it to the attribute	If the argument is a `bool`, `double`, or `int`, casts attribute to that type and performs the addition. The result is converted back to a string, returned, and assigned to the attribute. If the argument is a string, the operation returns the current string. If the argument is an attribute, the action is as above, based on the underlying type.

continued…

Table 6: (cont.)

	bool	double	int	std::string
subtract and assign	casts argument to bool and performs a logical and, returing the result and assigning it to the attribute	casts argument to int and subtracts, returning the result and assigning it to the attribute	casts argument to double and subtracts, returning the result and assigning it to the attribute	If the argument is a bool, double, or int, casts attribute to that type and performs the subtraction. The result is converted back to a string, returned, and assigned to the attribute. If the argument is a string, the operation returns the current string. If the argument is an attribute, the action is as above, based on the underlying type.

5 Using inilib

This section contains information on obtaining inilib, installing it, and getting help if something goes wrong. In addition, this section describes the coding standards maintained in the creation of inilib.

5.1 Obtaining inilib

The source code for inilib is available from the project's homepage:

http://inilib.sourceforge.net/

Packages containing the official, supported releases are available from the web page, as well as CVS access for the most current code. The CVS code – unless otherwise noted – should be considered unstable, and may not function as intended, and may therefore break your applications.

The official inilib distribution contains the source code in src/, as well as a comprehensive test suite that can be found at contrib/test_suite/ and usage examples in contrib/examples/. The test suite is discussed in Section 5.4. The usage examples contain extensive comments on the usage of inilib. In addition, the documentation for inilib is available in the doc/ directory of the distribution and on the project's home page.

5.2 Installing inilib

inilib uses the GNU autoconf and automake utilities to create a build process that works across all tested platforms. In addition, it may work on platforms that are not tested or supported. For most cases, the build process is as simple as:

```
% ./configure
% make
% make examples
% cd contrib/test_suite ; ./inilib_test ; cd ../..
% make install
```

The make examples and ./inilib_test steps are optional, but it is recommended you use them. For file access reasons, you must be in the same directory as the inilib_test binary and test.ini file in order to run the test suite.

C++ does not enjoy the same standard methodology of building static libraries across different platforms and compilers like C does. Indeed, for C libraries, the use of ar(1) and (sometimes) ranlib(1) is all that is required. However, C++ functions (particularly where templates are involved) may require multiple passes from the compiler before a usable library can be produced. There is currently no uniform manner to produce C++ libraries across platforms/compilers. It is hoped that the next release of the GNU libtool project will address this issue. Until then, only the platforms and compilers listed in Table 7 are supported for use with inilib.

If your compiler is not listed, it is quite possible that your it will work properly with inilib – it should be a fairly straightforward task to modify configure.in for the right library compilation hooks for your C++ compiler.

5.3 Getting Help

Thanks to SourceForge, bug tracking and reporting software and mailing lists are available from the inilib homepage. The authors of inilib are Brian Barrett and Jeff Squyres. To contact the authors about a problem with inilib, please use the support listserv, inilib-support@lists.sourceforge.net.

There is also a development list, inlib-devel@lists.sourceforge.net.

5.4 Coding Standards

inilib was developed under a fairly stringent set of standards that should ensure proper functionality.

The primary development environment for inilib is Sun Solaris 2.6 (UltraSparc) using the Sun Workshop 5.0 compilers. All releases are verified to be free of memory leaks,[3] using bcheck, a memory checker that is part of the Workshop compiler suite.

In addition to being free of memory leaks, inilib releases will compile without warnings on all supported platforms using all supported compilers. GNU extenstions and non-standard extensions to the STL will not be used, to increase portability.

A comprehensive test suite is included with the inilib source. It can be found in the contrib/test_suite directory. This test suite is intended to test the entire inilib product for proper functionality. All releases canidates will be verified using the test suite on all supported platform/compiler combinations (See Section 5.5) before being released.

5.5 Supported Platforms

Table 7 shows the platform/compiler combinations have been tested and are supported by inilib. Other platforms may work, but may require some modifications to the build script. g++ 2.95.2 may provide the easiest porting, as its C++ library build process is the same as the standard C library build process.

[3] Free from avoidable memory leaks. The Workshop 5.0 STL implementation has a few small memory leaks in the iostream implementation, so bcheck will find some memory leaks in inilib.

Platform	Compiler
Solaris 2.6 (Sparc)	Workshop 5.0 CC KCC 3.4f g++ 2.95.2
Solaris 7 (Sparc)	Workshop 5.0 CC KCC 3.4f g++ 2.95.2
x86 Linux (RedHat 6.2)	KCC 3.4f g++ 2.95.2
MIPS Irix	Irix CC KCC 3.4f g++ 2.95.2

Table 7: Platform / compiler combinations supported by `inilib`.

References

[1] PHP. http://www.php.net/.

[2] C++ Forum. ANSI/ISO standard, programming language C++. Technical report, American National Standards Institute, 1998.

[3] Microsoft. Microsoft Windows^{TM}. http://www.microsoft.com/.

[4] SGI Inc. Standard Template Library Programmer's Guide. http://www.sgi.com/Technology/STL/, 1996.

[5] Larry Wall. Perl. http://www.perl.com/.