Introduction:

Bit Bucket is a C++ heterogeneous container / hash map, where each cell holds one of the following intrinsic types: bool, char, int, float and std::string. The type of the variant can change during runtime.

Here's a few examples to explain the idea:

Getting data into the bucket:

```
BitBucket bucket;
bucket["int"] = 42; // First an int
bucket["int"] = "rock!"; // Then a string
bucket["f"] = 3.14f;
bucket["string"] = "Soundgarden";
std::cout << bucket["string"] << " " << bucket["int"];
```

Output: "Soundgarden rock!".

Getting data out of the bucket:

```
bucket["int"] = 42;
bucket["float"] = 3.14f;
int x = bucket["int"]; // x is now equal to 42
x = bucket["float"]; // The float value has been truncated to 3
std::string s = bucket["int"]; // s is now equal to "42". Conversion works from strings to numbers and vice versa.

bucket["s"] = "Not a number";
int y = bucket["s"]; // y is equal to 0 and the stack trace will be printed
bool flag = bucket["nonExistant"]; // bool will be equal to false, and the stack trace will be printed
```

The classes:

BitBucket is a class that inherits from std::unordered_map, and adds the following functionality:

1) Construction from a text file. The file must be of the following format:

   ```
   <type> <name> <value>
   <type> <name> <value>
   ...
   ```

   Note: type must be one of the following types: bool, char, int, float, string or auto, where stating auto will automatically deduce the type of the variable according to its value, much like C++11's auto keyword.
2) Output to a text file via BitBucket::serialize
3) Console output via BitBucket::print
4) Printing only the blank cells via BitBucket::printBlank. More on this in the error handling section.

Bit is what the BitBucket holds. This class is essentially a wrapper for boost::variant, with the following added functionality:

1) Constructor that accepts string type, string value (Reading from a file, for example).
2) Output streaming.
3) Conversion from the Bit class to bool, char, int, float or string. Conversions are done according to expected C functionality. For example: A Bit holding 3.14f will convert to an int implicitly by truncation. In addition, strings will also attempted to be converted to the other types.

Error handling:

Two types of errors may commonly occur when working with a Bit Bucket:

1) Access to an invalid cell. For example:

```
int x = bucket["nonExistant"];
```

2) Accessing the variant with a wrong type / An incorrect type cast:

```
bucket["string"] = "Jerusalem";
int x = bucket["string"]; // Can only convert strings representing numbers, not words
```

What do we do in these cases?

In the first case, the cell "nonExistant" will be created and its type will be boost::blank, which is a completely empty struct. x will equal 0. There is no exception or crash. Similarly, in the second case x will be equal to 0 and the stack trace will be printed. So how do we catch such errors?

1) The *stack trace will be printed*, so you'll know exactly which line of code caused either kind of error, the moment it occurs. The stack trace is printed via StackWalker on Windows, and backtrace() on Linux. The stack tracing is conditionally compiled into the library, so if you got this library already compiled keep that in mind.
2) You may use BitBucket::printBlank() to see which cells are blank.
3) You may use BitBucket::isSet(string key) to see if a cell exists before accessing it.

Requirements:

Bit Bucket uses Boost as well as several C++11 features. VS2010 compiles Bit Bucket just fine, as well as VS2012, and the latest GCC and Clang.

Credits and licensing:

 It is currently hosted at: https://github.com/assafmuller/Bit-Bucket. Feel free to fork and pull request!