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, however, strong typing is kept during run-time. Meaning, if the variant is equal to 5, it's type is currently an int. If you try to yank it out of the bucket as a float, an error will occur. Error handling is done gracefully and is explained better below.

An example would explain the intended functionality best:

```
BitBucket bucket;
bucket["int"] = 42;
bucket["int"] = "rock!";
bucket["f"] = 3.14f;
bucket["string"] = "Soundgarden";

std::string s = bucket["int"]; // Remember, bucket["int"] was assigned
42, and then "rock!"
std::cout << bucket["string"] << " " << s;
```

Output: "Soundgarden rock!".

```
bucket["int"] = 42;
int x = bucket["int"]; // x is now equal to 42
float y = bucket["int"]; // Oops! y is now equal to 0, and the stack
trace will be printed to the console so you may fix the error as soon as
possible
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) operator() bool, char, int, float, string - Successful if the current type of the variant is the type being cast to.

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.

```
int x = bucket["float"]; // One possibility

Bit bit = bucket["float"]; // Another
int x = bit;
```

What do you 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.

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.

Credits and licensing:
Bit Bucket was written by Assaf Muller. The code is under the MIT license, which means you can use it for closed, open source, free and commercial products. It is currently hosted at: https://github.com/assafmuller/Bit-Bucket. Feel free to fork and pull request!