1. Crash Logs

Mercury comes with a crash log generator. It, like everything else, is compatible with both Linux and Windows. It provides information about the current crashing issue, as well as information that was on the stack, like static-time code.

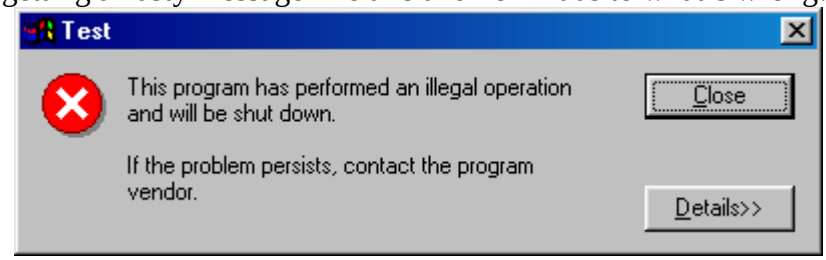
When you run code like this:

Main.cpp

```
int MakeMess()
{
	int i;
	memcpy( NULL, &i, 100000 );
	return 5;
}

int main(int argv, char* argc[]) {
	StartCrashHandler(argc, argv);
	MakeMess();
	return 1;
}
```

Instead of getting a nasty message like this and no hint as to what's wrong…



You get a nice file like:

CrashLog.txt

```
CVS Build crash report (build 1000)
-----------------------------------

Crash reason: Access Violation

0043abf3: memcpy()
00402d34: void MakeMess(void)
00402dfa: main()
7c910732: ntdll!RtlAllocateHeap [7c900000+105d4+15e]
7c910732: ntdll!RtlAllocateHeap [7c900000+105d4+15e]
7c911596: ntdll!wcsncpy [7c900000+10a8f+b07]
0043b30e: _onexit()
0043cbf3: mainCRTStartup()

--End of Crash Report
```

Obviously, copying 100000 bytes to memory starting at address zero isn't allowed, but we don't have to scratch our heads when we get that little box. The program can either exit cleanly or you can run custom code to present the user with a dialog box saying that the program crashed.

In code you can force a crash, say for an assert or known failure. You can do this using three methods (all of which can be found in MercuryUtil.h).

1) mercury_crash(message);
2) ASSERT(condition);
3) ASSERT(condition, message);

The crash handler will provide the user with the reason on the first line like the following after you execute the command: `ASSERT( 5 < 4 );`

CrashLog.txt

```
CVS Build crash report (build 1000)
-----------------------------------

Assertion 5 < 4 failed

00439aae: void debug_crash(void)
00439c02: void ForceCrashHandler(char const *)
004326d5: void mercury_crash(char const *)
00402d36: int MakeMess(void)
00402d73: main()
           .
           .
           .
```

2. MercuryINI

Mercury contains an easy-to-use comprehensive class designed to process and deal with INI files. MercuryINI is a class that is a C++ representation of an INI file. It supports both loading and saving. PREFSMAN is an example of a global singleton instantiation of MercuryINI.

MercuryINI can either run parallel to a file (use `Open()` and `Save()` or load from and save to plain strings (use `Load()` and `Dump()`). You can use the non-default constructor if you want the file to open in the constructor and optionally save in the destructor.

INI files are broken up into different keys and each key has a set of values. Each value can be whatever fits on one line. You can use `ConvertToCFormat()` and `ConvertToUnformatted()` from MercuryUtil.h if you want to store multi-line or non-text data in the INI file.

Access to MercuryINI is obtained by using the various `GetValue()` functions. `GetValue()` functions with a postfix, like `GetValueS()`, `GetValueI()`, `GetValueF()` retrieve the corresponding string, integer, or floating-point value. If the value does not exist it will return to you whatever is put into the default value and if the `bMakeValIfNoExist` flag is set, it will make the value based on the default value. The `GetValue()` function that does not have a postfix will give you the value by parameter and return whether or not it was able to retrieve the element.

3. Mercury Utility Functions

Mercury contains some general utility functions for general use. It is suggested you use these functions because at a later time, the way in which files are used may change. If you use these functions, you will not have to perform any conversions.

- `CString ssprintf( const char * unformatted, … )` – Perform formatted text output, however do not send it to the console, instead return the string of its output. This function acts exactly like `printf()` for strings. It is suggested over stringstreams because it is slightly faster and allows for one-line conversions, unlike stringstreams which require clearing, and reading of the string.
- `CString ConvertToCFormat( const CString & ncf )` – Convert string containing binary characters to C-style string. This means convert newlines to \n, back slashes to \\, tabs to \t, etc.
- `CString ConvertToUnformatted( const CString & cf )` – Convert a C-style formatted string into it's binary string equivalent. This will process the \t's, \n's, \0's, etc. and convert them into the characters they are meant to represent.
- `long DumpFromFile(const CString & filename, char *& data)` – Dump data from a file into a char * and return the size of data. If -1 then there was an error opening the file.
- `bool DumpToFile( const CString & filename, const char * data, long bytes )` – Dump data into the file filename. bytes specifies the length of data. Returns true if successful, false else.
- `void FileToCString( const CString & filename, CString & data )` – Dump the contents of a file to data.
- `void CStringToFile( const CString & filename, const CString & data )` – Dump data into file.
- `bool FileExists( const CString & filename )` – Return true if the file exists, false if not.
- `long BytesUntil( const char* strin, const char * termin, long start, long slen, long termlen )` – Return the number of bytes until a desired terminal where `strin` is the input string, `termin` is the tokens to look for (one per character) start is where to begin looking in `strin`, `slen` is the total length of `sterin`, and `termlen` is the number of terminal tokens to check for.

- `long BytesNUntil( const char* strin, const char * termin, long start, long slen, long termlen )` – Same as above, except returns number of bytes until it reaches a character that is not in the list of tokens.
- `void SplitStrings( const CString & in, vector < CString > & out, const char * termin, const char * whitespace, long termlen, long wslen )` – Split given string `in` into a vector of strings based on separation of the string by whitespace and terminals. `termlen` is the number of terminals to look for, `wslen` is the number of whitespace chars to look for. Example: you can split apart (5,6,8,9) by saying `SplitStrings( "5, 6, 7, 8, 9", out, ",", " ", 1, 1 );`
- `inline void  Clamp( float & in, float min, float max )` – Clamp in within the boundaries min and max.

4. Mercury Log

Rather than just clogging up the out console window with tons of debug information and trying to scatter around debug output files, Mercury provides a simple log system. MercuryLog.h provides a simple logging class which is instantiated as a singleton at static time called LOG. If you wish to have outputted format, use `ssprintf` to format your output.

Example:

Input Code

```
1  LOG.Info( "This is being outputted to the info file." );
2  LOG.Log( ssprintf( "This is formatted output %f, %d.",
3           55.22f, 5 ) );
4  LOG.Warn( "You really messed something up now!" );
```

Output (to console)

```
1  This is being outputted to the info file.
2  00:00.150: This is formatted output 55.220000, 5.
3  00:00.150: ********************************************
4  00:00.150: WARNING:You really messed something up now!
5  00:00.150: ********************************************
```