Command Line Parsing

David Eberly
Magic Software, Inc.
http://www.magic-software.com

Created: July 15, 1999
Modified: September 18, 2002

I used to use Henry Spencer's *getopt* routines for processing command–line parameters. The *getopt* routines are limited in that the option names have to be a single letter. Also, the main routine must contain a loop and a switch statement (of options) which repeatedly gets an argument and decides which option it is and which action to take. I wrote my own command–line parser which allows option names of length greater than 1, and which allows one to get an argument anywhere in the main routine. This tends to keep the parameter processing and actions together in a related block of code.

The definition for the parser class is shown below (and has namespace Mgc):

```
class Command
{
public:
    Command (int iQuantity, char** apcArgument);
    Command (char* acCmdline);
    ~Command ();

    // return value is index of first excess argument
    int ExcessArguments ();

    // Set bounds for numerical arguments.  If bounds are required, they must
    // be set for each argument.
    Command& Min (double dValue);
    Command& Max (double dValue);
    Command& Inf (double dValue);
    Command& Sup (double dValue);

    // The return value of the following methods is the option index within
    // the argument array.

    // Use the boolean methods for options which take no argument, for
    // example in
    //           myprogram -debug -x 10 -y 20 filename
    // the option -debug has no argument.

    int Boolean (char* acName);  // returns existence of option
    int Boolean (char* acName, bool& rbValue);
```

```
    int Integer (char* acName, int& riValue);
    int Float (char* acName, float& rfValue);
    int Double (char* acName, double& rdValue);
    int String (char* acName, char*& racValue);
    int Filename (char*& racName);

    // last error reporting
    const char* GetLastError ();
};
```

The constructor `Command(int,char**)` takes as input the arguments to routine `main`. In a Microsoft Windows application, the constructor `Command(char*)` takes as input the command–line string to `WinMain`.

When parsing options that take numeric arguments, it is possible that upper and lower bounds are required on the input. The bounds for input X are set via calls to `Min` (min $\leq$ X), `Max` (X $\leq$ max), `Inf` (inf $<$ X), or `Sup` (X $<$ sup). These methods return `*this` so that a `Command` object can set bounds and acquire input within the same statement (see examples below).

For now the supported option types are *booleans* (option takes no arguments), *integers*, *reals*, *strings*, or *filenames*. Each type has an associated method whose first `char*` parameter is the option name and whose second parameter will be the option argument (if present). The exceptions are the first boolean method (option with no argument) and filenames (argument with no option). The return value of these methods is the index within the command–line string or zero if the option did not occur.

When all arguments are processed, the method `ExcessArguments` may be called to check for extraneous information on the command–line which does not match what was expected by the program. If extra or unknown arguments appear, then the return value is the index within the command–line string of the first such argument.

The function `GetLastError` returns information about problems with reading command–line parameters. The errors are "option not found", "option requires an argument", "argument out of range", and "filename not found". The user has the responsibility of calling `GetLastError` as desired.

A simple example of command–line parsing is the following. Suppose that you have a program for integrating a function $f(x)$ whose domain is the half–open interval $[a, b)$. The function will be specified as a string and the endpoints of the interval will be specified as real numbers. Let's assume that we want $0 \leq a < b < 1$. Your program will use Riemann sums to do the approximation, so you also want to input the number of partition points as an integer. Finally, your program will write information about the integration to a file.

```
#include "MgcCommand.h"
using namespace Mgc;

// usage: "integrate [options] outputfile"
//        "  -a (float)    :  left endpoint (a >= 0, default=0.0)"
//        "  -b (float)    :  right endpoint (a < b < 1, default=0.5)"
//        "  -num (int)    :  number of partitions (default=1)"
//        "  -func (string):  expression for f(x)"
//        "  -debug        :  debug information (default=none)"
//        "  outputfile    :  name of file for output information"
```

```
int main (int iQuantity, char** apcArgument)
{
    Command kCmd(iQuantity,apcArgument);

    // get left end point (0 <= a required)
    double dA = 0.0f;
    kCmd.Min(0.0).Double("a",dA);
    if ( kCmd.GetLastError() )
    {
        cout << "0 <= a required" << endl;
        return 1;
    }

    // get right end point (a < b < 1 required)
    double dB = 0.5;
    kCmd.Inf(dA).Sup(1.0).Double("b",dB);
    if ( kCmd.GetLastError() )
    {
        cout << "a < b < 1 required" << endl;
        return 2;
    }

    // get number of partition points (1 or larger)
    int iPoints = 1;
    kCmd.Min(1).Integer("num",iPoints);
    if ( kCmd.GetLastError() )
    {
        cout << "num parameter must be 1 or larger" << endl;
        return 3;
    }

    // get function expression (must be supplied)
    char acFunction[128];
    if ( !kCmd.String("func",acFunction) )
    {
        cout << "function must be specified on command line" << endl;
        return 4;
    }

    // get output file name
    char acOutfile[128];
    if ( !kCmd.Filename(acOutfile) )
    {
        cout << "output file must be specified" << endl;
        return 5;
    }

    // want debug information?
```

```
    bool bDebug = false;
    kCmd.Boolean("debug",bDebug);

    // check for extraneous or unknown options
    if ( kCmd.ExcessArguments() )
    {
        cout << "command line has excess arguments" << endl;
        return 6;
    }

    // your program code goes here
    return 0;
}
```