

Building a Reusable Scheme for Debugging

Travis Spencer

CS Tutors – Portland State University

May 3, 2005

1 Introduction

As students, we are constantly moving from one assignment to the next at a rapid pace. One week we're working on a scanner for compilers class and the next we're writing a shell for operating systems and the next we're creating a revision control system for software engineering. In an environment like this, it is helpful to have some general tools that can move with us from one project to the next. One thing that is universal to all CS projects is their need to be debugged. For this reason, it makes a lot of sense to take the time to create a general purpose debugging scheme that can be reused in various projects. One aspect of such a tool is the automation of user input.

2 Automating User Input

Often assigned projects are supposed to take input from a user and perform some processing on it. Debugging this sort of program can be tiresome; it means that you have to constantly reenter fictions, test information. It doesn't have to be so tedious. One simple solution is to redirect standard in from the command line like this:

```
./a.out < testcase01
```

While this works, it isn't very flexible. You will have to enter that every time you start a new instance of gdb and each time you run the program from the command line.

A better solution is one that reads input from a default file and allows the default test case to be easily overridden. Also, the program must read input interactively after submitting it to be graded. Also, it would be nice if nothing has to be changed before turning in the assignment. Here is one way to solve this problem.

2.1 Setting up the Solution

Create a header file that will be included in all of your C++ files (say `globals.h`) with the following code.

```
#ifndef DEBUG
#   define printDebugMsg(msg) ((void)0)
#   define printDebugValue(x) ((void)0)

#   include <iosfwd>

        extern std::istream& input;
#else
#   define DEFAULT_INPUT "/u/tspencer/school/cs321/Scanner/tests/test01"

#   ifdef VERBOSE_DEBUG
#       define printDebugMsg(msg) std::cout << (msg) << std::endl
#       define printDebugValue(x) std::cout << #x " = " << (x) << std::endl
#   else
#       define printDebugValue(x) ((void)0)
#       define printDebugMsg(msg) ((void)0)
#   endif

#   include <fstream>

        extern std::ifstream input;
#endif
```

This code solves the problem stated above through the use of conditional compilation, by creating an alternative input stream, and by defining a default file to read a test case from.

Firstly, when the `DEBUG` symbol is not defined the preprocessor will substitute all print statements with a cast of zero to a void type, a typical

way of creating a noop; however, when `DEBUG` and `VERBOSE_DEBUG` are defined (e.g., in your makefile), the print statements can be used to test variable values and output informative messages. Since the grader won't define a `DEBUG` symbol when grading your work, the print statements can be left in the code after submission. By adding the extra symbol `VERBOSE_DEBUG`, you can flip the print statements on and off while you are debugging.

Secondly, this snippet defines an external, global variable called `input`. When debugging is *not* defined, it is declared as an `istream`. While debugging, on the other hand, it is declared as a file stream (`fstream`). Through the wonders of inheritance, this difference will allow you to use the variable to read from a file while debugging and standard input after submitting the project.

The last thing to note about this code is the symbol `DEFAULT_INPUT` which is only defined when debugging is enabled. This symbol will be used as the default test case unless another is provided (see below).

It is tempting to use this global file to include a lot of project-specific information; however, doing so isn't advisable because every source file will include it. This means that any change to `globals.h` will force a recompilation of the entire project. If you are using `make`, this will subvert its dependency checking.

2.2 Using the Proposed Solution

Now that we have this global definition, how can it be used? The first thing that you have to do is create a variable called `input` whose existence was announced in the global header file. I often put this in my project's main file. The code to do this looks something like this:

```
#ifndef DEBUG
#   include <iostream>

    std::istream& input = std::cin;
#else
    std::ifstream input;
#endif
```

Essentially, this creates an alias for `std::cin` when `DEBUG` is undefined using a reference. When `DEBUG` is defined, however, it declares an

actual variable. Note how `iostream` was included since the header file only included `iosfwd`. The latter just forward declares `istream` while `iostream` includes the actual definitions of `iostream`'s members and methods.

Now, whenever you want to read from the user or from a file, you use `input` — not `cin` or some other file input stream. This way nothing in the code will need to be changed irregardless of whether `DEBUG` is defined or not.

Finally, you need to open a file for the input stream to read from while debugging. This should be done in a way that is flexible enough to read from a default test case while providing a simple way to override it. To accomplish this, in your project's main function, include code such as this:

```
int main(int argc, char** argv)
{
#ifdef DEBUG
    if (argc > 1)
        input.open(argv[1]);
    else input.open(DEFAULT_INPUT);

    if (! input.is_open()) {
        cerr << "The input file couldn't be opened.\n";
        return EXIT_FAILURE;
    }
#endif

    return EXIT_SUCCESS;
}
```

This snippet expects the first argument to be a path to a test case file. If the program is called without any arguments, it will use the default test case that was defined as `DEFAULT_INPUT` in the global header file. Now as you are developing your program, you can start it without any arguments or by simply typing `run` in `gdb` when you are debugging. When you need to test the application with different inputs, you can simply pass them on the command line or as a parameter to `gdb`'s `run` command. You wont have to redirect standard in and you wont have to type anything.

2.3 Putting it all Together

To see how this works overall, I've put together a simple program that consists of a makefile, a global header file, and a single source code file.

2.3.1 Makefile

```
debug:
    g++ -g -DDEBUG -o myproj myproj.cpp

noisy:
    g++ -g -DVERBOSE_DEBUG -DDEBUG -o myproj myproj.cpp

release:
    g++ -o myproj myproj.cpp
```

2.3.2 globals.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

#ifndef DEBUG
#   define printDebugMsg(msg) ((void)0)
#   define printDebugValue(x) ((void)0)

#   include <iosfwd>

    extern std::istream& input;
#else
#   define DEFAULT_INPUT "input"

#   ifdef VERBOSE_DEBUG
#       define printDebugMsg(msg) std::cout << (msg) << std::endl
#       define printDebugValue(x) std::cout << #x " = " << (x) << std::endl
#   else
#       define printDebugValue(x) ((void)0)
#       define printDebugMsg(msg) ((void)0)
#   endif
#endif
```

```

#   include <fstream>

        extern std::ifstream input;
#endif

#endif // GLOBALS_H

2.3.3 myproj.cpp
#include "globals.h"

#include <iostream>

using namespace std;

void foo(int x, int y);

#ifndef DEBUG
        istream& input = cin;
#else
        ifstream input;
#endif

int main(int argc, char** argv)
{
#ifdef DEBUG
        if (argc > 1)
                input.open(argv[1]);
        else input.open(DEFAULT_INPUT);

        if (! input.is_open())
        {
                cerr << "The input file couldn't be opened.\n";
                return EXIT_FAILURE;
        }
#endif
        int i, j;

```

```

    // Note how I'm using 'input' and not cin and not some file stream.
    input >> i >> j;

    printDebugMsg("About to call foo.");
    foo(i, j);
    printDebugMsg("Returned from foo.");

    return EXIT_SUCCESS;
}

void foo(int x, int y)
{
    printDebugValue(x);
    printDebugValue(y);

    input >> x >> y;

    printDebugValue(x);
    printDebugValue(y);

    cout << x << endl
         << y << endl;
}

```

2.3.4 input

```
1 2 3 4
```

2.3.5 input2

```
5 6 7 8
```

2.3.6 Compiling and Running the Sample

When you compile and run this program using the default target (by simply typing `make`), you get this output:

```
$ ./myproj
```

```
3
4
```

Notice how you don't get prompted four times for input. Now compile it by running `make release`. When you run the program, you are prompted for four numbers and the last two are output like this:

```
$ ./myproj
4 3 2 1
2
1
```

Now, compile the program using the `noicy` target. When you run it, you will see the output of all print statements. Here is a sample run:

```
$ ./myproj
About to call foo.
x = 1
y = 2
x = 3
y = 4
3
4
Returned from foo.
```

Finally, you can override the default input file by building the program with the `debug` or `noicy` targets and passing a different test case to the program like this:

```
$ make noicy
g++ -g -DVERBOSE_DEBUG -DDEBUG -o myproj myproj.cpp
$ ./myproj input2
About to call foo.
x = 5
y = 6
x = 7
y = 8
7
```

```
8
Returned from foo.
$ make
g++ -g -DDEBUG -o myproj myproj.cpp
$ ./myproj input2
7
8
```

3 Conclusion

Hopefully, after reading this tutorial, you will see that you don't have to constantly enter input while writing your programs. You can create a simple scheme that is flexible enough to be used from one programming assignment to the next. By freeing you from the burden of constantly entering input, you can write better test cases, focus more on the programming assignment, and hopefully get better grades.