Software Development 6: Debugging

1. Introduction
2. Incremental Software Development
3. Incremental Development Example
4. Software Bugs
5. Testing
6. **Debugging**
   - **debugging vs. testing**
   - **testing and programming errors**
   - **tracing**
   - **tracing tools**
   - **instrumentation**
   - **other debugging tools**

Debugging vs. Testing

Testing aims to discover *if* software functions properly or not.

If testing reveals that the software does *not* function properly, this means it contains *bugs* (*errors*).

Debugging is the process of locating the bugs/errors in the code and eliminating them.

The patched program must obviously be thoroughly *retested!*

As we discussed, the primary sources of **software bugs** are:

- **programming errors** (errors in program source code)
- **design errors** (errors in developing the system design)

(We will here assume the specification is correct, but with some software systems, testing may reveal deciencies in the spec!)

Testing and Design Errors

A **design error** occurs when the program code functions as it was designed to, but fails to work properly in some tests.

The discovery of design errors requires changes to the design of the program, and subsequent changes to the program code.

Design errors can lead to very substantial source code changes being required, if major errors were made in the original design.

One the other hand, design errors can be as simple as an overlooked case, requiring no more than an additional `if-else` or `switch` case.

Program design is beyond the scope of this lecture.

Testing and Programming Errors

Programming errors occur when the program source code does not behave according to the program *design*.

We earlier looked at a simple **programming bug taxonomy**.

Programming errors/bugs lead to three common outcomes for test cases that *fail:*

- The program **terminates normally** but produces incorrect output or fails to take correct actions (e.g., prompting).
- The program **terminates abnormally** (i.e., **crashes**).
- The program "**hangs**" (i.e., does not terminate until killed).

Tracing

Tracing refers to printing out (or recording) information about what is happening during a program run.

Tracing is the primary method for locating software bugs.

The primary targets for tracing are:

- program statements executed (branches and loops)
- variable values
- function/method calls (invocations and returns)
- I/O (e.g., file and socket reads/writes)

Tracing tools can provide particular kinds of traces.

Tracing can also be done by **instrumenting** the code—i.e., adding instructions to collect or print out info.

Tracing (contd.)

Tracing tools include **debuggers** and tracing utilities such as `ltrace` and `strace`.

Instrumentation can be done via certain tools such as `gprof` (in conjunction with special GCC flags).

Instrumentation can also be done *manually*, by simply inserting print statements into the code.

C also provides some very basic "tracing" capability for variable values via its `assert()` macro, which validates values.

Tracing: Program Statements

One reason a program may behave improperly is that it does not execute the correct code statements under certain conditions.

In order to identify such errors, it is useful to have a trace of the program statement that get executed under the problem-causing conditions.

Tracing of program statements can be done with debuggers, manual instrumentation, and instrumentation tools like `gprof`.

Often we do not need a *complete* trace of program execution, only information about whether particular branches were taken, whether particular functions were called, and/or how many times certain loop bodies were executed.

Both debuggers and manual instrumentation can provide focused statement tracing.

Tracing: Program Statements (contd.)

Tracing statements to find a control flow error is a first step in debugging the error.

In order to fix the error, the reason for the incorrect control flow must be identified and remedied.

Sometimes such errors can be found by **inspection** of the code responsible for the erroneous control decisions.

Sometimes the reasons will not be obvious, and will require tracing the values being used in the control decisions.

Tracing: Variable Values

Several types of software bugs can lead to variables having incorrect values.

Incorrect variable values may directly result in output errors, or may produce control flow errors that ultimately lead to output errors or other incorrect behavior.

Both debuggers and manual instrumentation can be used to examine—or trace the evolution of—the values stored in program variables.

C's `assert()` macro can be used to identify values that end up outside of the valid range (and stop program execution).

Tracing: Function Calls

Virtually all programs are broken down into functions/methods, so tracing function calls is one way to trace basic control flow.

Generally we will also want to trace the *arguments* given in function calls, as well as any *return values* from the calls.

Debuggers can be used to get this information, though they generally require the user to select each function to be traced, so it is not fast to simply trace all program functions.

Debuggers also allow you to examine the function **call stack** (by doing what is generally called a **backtrace**).

This makes it possible to see the sequence of calls (with arguments) that caused a certain function to be executed, without having to trace each function.

Tracing: Function Calls (contd.)

Stack backtraces are particularly useful when programs *crash*.

Debuggers can use **core dump** files to identify the sequence of function calls (with arguments) that lead to a crash.

The tracing utilities `ltrace` and `strace` allow easy tracing of calls to *library functions* and *system calls* that are made by a program.

Manual instrumentation can also be used to trace function calls.

Tracing: I/O

In order to locate certain program bugs, it is often critical to be able to trace the data that is being read/written by the program.

For example, a program may appear to be making incorrect computations on numbers read from a file, when the problem is actually due to bad assumptions about the file format.

Often, program *"hangs"* are actually due to programs waiting for I/O, which can easily be seen by tracing I/O operations.

Since all I/O is ultimately mediated by the OS, all I/O ultimately involves **system calls** such as `read()` and `write()`.

Because of this, the `ltrace` and `strace` tools are generally a good choice for tracing I/O.

Tracing Tools: GDB

The most popular **debugger** for Linux is the **GNU Debugger** or **GDB** (command `gdb`).

GDB is a *command-line* only tool.

Several *GUI front-ends* have been built for it and several C/C++ IDEs use GDB as their underlying debugger.

The GNU Project's GUI front-end for GDB is **Data Display Debugger** (**DDD**).

GDB/DDD are covered in the **Development Tools** lecture #5.

GDB/DDD allow you to trace statement executions, function calls, and variable values.

Tracing Tools: GDB (contd.)

Unfortunately, using GDB to simply trace calls to some function is not exactly straightforward:

1. Start GDB on your program:
   `gdb ./myprog` (remember to compile with the `-g` flag)
2. Set `myfunc()` as a *breakpoint*:
   ```
   br myfunc
   ```
3. Now set commands to be run at breakpoint:
   ```
   silent
   bt 1
   c
   end
   ```
4. Run program:
   ```
   run
   ```

(Note: GDB can be *"scripted"* via a *commands file*.)

Tracing Tools: ltrace/strace

The tracing utilities `ltrace` and `strace` are also covered in the **Development Tools** lecture #5.

`ltrace` allows you to trace the (dynamic) **library function calls** and **system calls** made by your program as it runs.

`strace` traces only *system calls*.

Note that they cannot be used to trace calls to the functions defined in your own program.

They are particularly useful in programs that make heavy use of library functions or system calls.

Since all program **I/O** ultimately involves system calls, `ltrace`/`strace` are useful for *tracing I/O*.

Tracing Tools: gprof

The **GNU prof** program, `gprof`, can provide information about function calls in a program execution.

Specifically, it can show which functions were called and how many times, and provide a basic call "graph."

To use `gprof`, call `gcc` with the `-pg` flag, e.g.:

```
gcc -Wall -pg -o prog prog.c
```

Now, when you run `prog` it will produce a file `gmon.out` containing function call info for `gprof`.

Use `gprof` to display the info:

```
gprof -b ./prog gmon.out
```

Google will find numerous `gprof` tutorials online.

Manual Instrumentation

Manual instrumentation of a program is as simple as inserting *print statements* (e.g., `printf()`'s) into the program source code.

Printing different messages at different points in the code allows statement execution tracing, printing variable values at different points allows variable value tracing.

While students often dismiss manual instrumentation and immediately turn to debuggers, manual instrumentation can often work better (easily focused, edit-once-run-many).

The major drawback of manual instrumentation is that it requires modifying source code and disrupting normal program output.

Luckily, there are relatively simple ways to minimize these problems.

Manual Instrumentation (contd.)

The key approach for minimizing manual instrumentation issues is to set the instrumentation code up so that it can be easily *turned on and off*.

Accomplishing this by using **C preprocessor directives** that support **conditional compilation** allows instrumentation code to be completely eliminated from executables when desired.

Furthermore, the instrumentation code can be included/not based on options in gcc commands.

Thus, once properly instrumented, the instrumentation code can be activated or deactivated, simply by recompiling.

Manual Instrumentation (contd.)

Control of instrumentation code is accomplished via preprocessor (object) **macros** and the ability to define such macros using `gcc`'s `-D` option.

E.g., we can *define* the macro `DEBUG` by calling `gcc` as:

```
gcc -DDEBUG ...
```

E.g., we can define `DEBUG` with *value 1* by calling `gcc` as:

```
gcc -DDEBUG=1 ...
```

Programs can test `DEBUG` by including directives like:

```
#ifdef DEBUG
```

or

```
#if DEBUG > 0
```

Manual Instrumentation (contd.)

The most basic way to include controllable instrumentation code is by inserting code like the following into your program:

```
#ifdef DEBUG
  fprintf(stderr,"Value of x at point 3 is: %d\n",x);
#endif
```

Debug "levels" can be supported like:

```
#if DEBUG >= 1
  fprintf(stderr,"Value of x at point 3 is: %d\n",x);
#endif
```

(Note that debug output is to `stderr`, so as not to interfere with standard output, and allow *redirection* of debugging output.)

Manual Instrumentation (contd.)

A bit more sophisticated approach is to include something like the following in source files (or use a header file `DTRACE.h`):

```
#if DEBUG
  #define DTRACE(args...) fprintf(stderr, args)
#else
  #define DTRACE(args...)
#endif
```

Tracing statements can then be added to the program with single lines like: `DTRACE("Value of x at point 3 is: %d\n",x);`

(Note: the `DTRACE(args...)` notation is C99!)

Other Debugging Tools

There are a variety of other tools that can be useful in debugging:

- **static code analysis tools**
- **simulators**

Static analysis tools analyze program source code to provide information useful for debugging or to identify potential code errors.

Many of these tools are commercial/proprietary.

FOSS examples include: `ctags`, `cxref`, `cflow`, "lint" versions.

It is best to Google something like "static code analysis tools" to find current information.

Other Debugging Tools (contd.)

One of the most popular *simulator* tools is **Valgrind**.

Valgrind is a very useful tool for locating **memory errors**.

Valgrind is covered in the **Development Tools** lecture #5.