Chapter 7: The Debugger

References:

- Online Documentation of Microsoft Visual C++ 6.0 (Standard Edition): MSDN Library: Visual Studio 6.0 release.
- Peter Rechenberg, Gustav Pomberger (Eds.): Informatik-Handbuch (in German). Carl Hanser Verlag, 1997. Kapitel 12: Systemsoftware (H. Mössenböck).
- Brian W. Kernighan / Rob Pike: The Practice of Programming. Addison-Wesley, 1999, ISBN 0-201-61586-X.

Introduction (1)

- Even if a program runs through compiler/linker without error, it still may not work properly.
- Debugging is the process of finding such errors.

  Errors in programs are also called "bugs". In the early days of computers, it might have been possible that small insects got inside the hardware and caused a malfunction. However, already Edison used the word "bug" for a problem in his phonograph (1889), and already at that times the insect was only imaginary.

- "Good programmers know that they spend as much time debugging as writing so they try to learn from their mistakes." [Kernighan/Pike]

Introduction (2)

- Classes of errors:
  - ◇ The program may be aborted because of a run-time error detected by the computer hardware or a library function.

    E.g. dereferencing a NIL-pointer or otherwise accessing an illegal address (unfortunately, only addresses outside the memory allocated to the program cause a run-time error). E.g. a division by 0.
  - ◇ The program may execute without error message, but print wrong results, execute forever (or may in some other way not behave as intended).

Introduction (3)

- “Debugging is hard and can take long and unpredictable amounts of time, so the goal is to avoid having to do much of it. Techiques that help reduce debugging time include good design, good style, boundary condition tests, assertions and sanity checks in the code, defensive programming, well-designed interfaces, limited global data, and checking tools. An ounce of prevention really is worth a pound of cure.” [Kernighan/Pike]

Introduction (4)

- Put assertions in your program (conditions that must be true and cause an error message if not).

  E.g. every procedure should check its parameters.

- Procedures should not crash if called with wrong parameters ("Defensive Programming").

  This is somewhat similar to assertions. E.g. some versions of `printf` print "`(null)`" if a string to be printed is NIL, others crash.

- Consider extreme cases.

  This is what Kernighan/Pike call "boundary condition tests". E.g. if you develop a loop, consider the case that it is never executed. If you write data to an array, consider the case that the array is full.

Introduction (5)

- It might be useful to write procedures that dump data structures in readable form, echo the input read, etc. (debugging output).

  If you put these procedures inside an `#ifdef`, you can easily activate them if something went wrong and you want to see a bit more what is going on internally.

- You should test each piece/module of the program after it is developed, not only the complete program at the end.

  After every change, tests must be repeated. If might be possible to collect test input files and the corresponding output files and write a shellskript/batcg file that runs all the tests.

Introduction (6)

- If you detect a bug, immediately save the current version of the program (source files and binary), the input data, and any other information that might be needed to reproduce the unwanted behaviour.

  If a bug cannot be reproduced, it is very difficult to find it.

- Really clarify the source of each error!

  Sometimes it might be possible to change the program so that the bug no longer appears, but without really understanding why the program did behave in the way it did. From this you don't learn anything and chances are that the bug will reappear in some other form or that the changes you did created a new bug.

Introduction (7)

- It might be good to save older versions of the program.

  Often bugs were introduced by the most recent changes, so it is good to know what these changes were. There are source code control systems that keep track of changes.

- In rare circumstances, library routines contain bugs or even the compiler may not work correctly.

  Do not always think that it is your fault. You should be a bit critical towards other people's code, too. Of course, chances are much bigger that it is your fault, so one has to keep the right balance in searching the error.

Debuggers (1)

- Debuggers are programs that support the task of debugging.
- One can use a debugger in two ways:
  - ◇ One can analyse the main memory after the program crashed ("Post mortem debugging").
  - ◇ One can run the program step by step under the control of the debugger ("dynamic debugger" / "tracing debugger").

Debuggers (2)

- Whereas the compiler translates e.g. C into machine code, an important task of the debugger is to relate the main memory contents (machine code, data bytes) back to the original program.

  It needs the help of the compiler/linker for this task: They might make debugging information available.

- Program development environments usually contain editor, compiler, linker, make, and a debugger.

  Then the debugging information might only be understandable by this debugger. However, there are also standards for debugging information (e.g. in COFF, the common object file format).

Post-Mortem Debugging (1)

- When a program crashes (i.e. causes a run-time error), the operating system might
  - ◇ write the contens of main memory to a file ("core dump") that can be analyzed with a debugger.

    This is the UNIX solution. The file is called "`core`". One can also call the function "`abort()`" in the program to write the core file and terminate program execution. If the program does not terminate, sending it the signal `SIGQUIT` (e.g. by pressing `Ctrl+\`) performs the same function. Common debuggers under UNIX include `adb` (low-level), `dbx`, `sdb`, `xdb`, `gdb`.
  - ◇ directly call a debugger.

    This is done under Windows. The debugger might be "Dr. Watson" or the debugger that comes with VC++.

Post-Mortem Debugging (2)

- If the executable file contains no debugging information, the debugger may only show
  - ◇ the runtime error (e.g. "segmentation fault"),
  - ◇ the current values of registers,

    This includes the instruction pointer/program counter and the stack pointer.
  - ◇ a stack backtrace,

    If the program follows common conventions for stack management, the single activation records may be listed. However, no function names can be listed without debugging information (only return addresses), and the parameters and local variables are probably shown only as bytes (the debugger has no type information).

Post-Mortem Debugging (3)

- Output of the post-mortem debugger without debugging information (continued):
  - ◇ the machine code of the program,

    The debugger might be able to disassemble it, i.e. show it in symbolic form (but still as machine instructions).
  - ◇ the contents of any main memory address.

    Again, since there is no symbolic information, one does not know exactly where the global variables are.
- This information is basically useless unless one has access to the program source.

Post-Mortem Debugging (4)

- If the linker has preserved the global names in the executable file, the debugger is able to show
  - ◇ function names in the stack backtrace (especially the function in which the crash happened),

    This is not quite reliable since the names and addresses of functions declared as "`static`" might be missing. If there are standard procedure prologues and epilogues, the debugger might at least be able to detect this.
  - ◇ the contents of global variables.

    The debugger now knows their addresses. It does not know their types, so the user must himself/herself specify the type in the print command.

Post-Mortem Debugging (5)

- Today, compilers have options to put debugging information in the object code files.

  And the linker has an option to copy it into the executable program. Program files that do not include symbolic information are sometimes called "stripped" (this may also mean that the relocation information was removed).

- MS VC++ puts the debugging information into the files that are separate from the executable program.

  `VC60.pdb` contains information collected by the compiler for `.obj` files, e.g. type information. ⟨Project⟩`.pdb` is created by the linker, and includes symbol information, function prototypes, and the information from `VC60.pdb`. The absolute path of ⟨Project⟩`.pdb` is contained in the executable program.

Post-Mortem Debugging (6)

- Debugging information includes:
  - ◇ The names and machine code addresses of all functions (including static ones).
  - ◇ The correspondence between machine code addresses and source file lines.
  - ◇ Names, offsets, and types for function parameters and local variables.
  - ◇ Names, types, and addresses for global variables.
  - ◇ Names, types, and offsets of `struct` components.
  - ◇ Names and values of enumeration type constants.

Post-Mortem Debugging (7)

- Now the debugger is able to show:
  - ◇ In which line in which `.c`-file the error occurred.
  - ◇ A stack backtrace, i.e. which line in which file contained the function call for the function in which the error occurred (and recursively all calling functions up to `main`).
  - ◇ Values of parameters, and, if requested local and global variables.

    These are formatted according to the known types. It is also possible to browse complicated linked structures, i.e. to follow pointers and show all components of structures.

Post-Mortem Debugging (8)

- If a program causes an error under Windows, one gets the dialog box "Prog has caused an error in PROG.EXE. Prog will now close. If you continue to experience problems, try restarting your computer."

  In good operating systems, user programs can never confuse the operating systems such that a reboot is necessary.

- Then one can click on "Debug" to start the debugger.

  Alternatively, one can execute the program with `Debug`→`Go` (F5). Then the program is directly executed under the debugger.

Post-Mortem Debugging (9)

- E.g., the stack backtrace may look as follows:

```
p(int 0x00000001) line 11 + 3 bytes
main() line 13 + 7 bytes
mainCRTStartup() line 206 + 25 bytes
KERNEL32! bff7b9e4()
KERNEL32! bff7b896()
KERNEL32! bff7a24f()
```

- This means that the error was detected in procedure `p` at line `11`.

  The line was compiled into several machine instructions and it probably happend in the second one. This explains the offset of 3 bytes.

Post-Mortem Debugging (10)

- VC++ will open the source file and an puts a yellow arrow in front of the line that caused the error.
- In the example, the procedure `p` was called with one integer parameter that had the value `1`.
- The procedure `p` was called from the procedure `main` (line 13 in `main` contains the procedure call).

  One can double-click on `main` in the stack backtrace and VC++ opens the source file for it (this position is marked with a green triangle).
- `main` itself was called from `mainCRTStartup`.

Post-Mortem Debugging (11)

- There are three more return addresses on the stack, but no symbolic information is available for them.
- In the "Watch" windows, one can enter the names of variables and see their values.

  In order to see local variables, the right function must be selected on the stack backtrace (double-click on it if necessary). Pointers can be dereferenced by clicking on the "+" sign near to them.
- Alternatively, one can open the "Variables" window to see all variables of the selected function.

  Debugger windows are opened under `View`→`Debug Windows`. There are also windows that show registers, memory, or machine code.

Dynamic Debugger (1)

- A dynamic debugger (or tracing debugger) can execute the program step by step. In each step, it is possible to view variable values.

  In MS VC++, one can start the program under the debugger with `Build`→`Start Debug`→`Step Into`. (the "`Build`" menu then changes to "`Debug`"). By repeatedly selecting this (or pressing F10) one can execute every line of the program. However, the debugger will also "step into" library procedures. If one wants to execute an entire procedure call without stopping, one selects "`Step Over`". If one wants to execute the rest of the current procedure and stop only at the next statement of the calling function, one selects "`Step Out`".

Dynamic Debugger (2)

- One can also execute longer portions of the program without stopping, e.g. everything until the critical point.
- One does this by setting a breakpoint on the line where one wants execution to be temporarily suspended.

  In MS VC++, one clicks with the right mouse button in front of the line where one wants the breakpoint to be set. The context menu contains "Insert/Remove Breakpoint". The program is started with `Build`→`Start Debug`→`Go` (then the `Build` menue changes to `Debug`, which also contains `Stop Debug`).

Dynamic Debugger (3)

- A program can contain several breakpoints.

  Execution then stops at each breakpoint that is reached.

- Instead of setting a breakpoint, one can also place the cursor at the line where one wants to stop and then select "`Run to Cursor`".

- Experience shows that executing a program in this way in a debugger can take a lot of time.

- It is better to first get the post mortem stack back-trace and think about possible reasons for the error.