Unix Programming

Debugging Tools

Outline

- Debugging
- Debugging: gdb
- Debugging: Visual Tools

Bug Identification & Elimination

1. **Bug reports should contain a test case, output, and the version number of the software.**
2. **Reproduce the bug using the same version the customer used.**
3. **Find the root cause of the bug.**
4. **Check if the bug still occurs with the latest version. If it does, fix it.**
5. **If it doesn't, make sure it is not just masked by other changes to the software.**
6. **Add test cases used to reproduce the bug to the regression test suite.**
7. **Keep Records!**

What is gdb?

- GNU Debugger
- A debugger for several languages, including C and C++
- It allows you to inspect what the program is doing at a certain point during execution.
- Errors like segmentation faults may be easier to find with the help of gdb.
- http://sourceware.org/gdb/current/onlinedocs/gdb toc.html - online manual

Additional step when compiling program

- Normally, you would compile a program like:

```
gcc [flags] <source files> -o <output file>
```

- For example:

```
gcc -Wall -Werror -ansi -pedantic-errors prog1.c -o prog1.x
```

- Now you add a -g option to enable built-in debugging support (which gdb needs):

```
gcc [other flags] -g <source files> -o <output file>
```

For example:

```
gcc -Wall -Werror -ansi -pedantic-errors -g prog1.c -o prog1.x
```

Starting up gdb

- Just try "gdb" or "gdb prog1.x." You'll get a prompt that looks like this:

```
(gdb)
```

- If you didn't specify a program to debug, you'll have to load it in now:

```
(gdb) file prog1.x
```

- Here, prog1.x is the program you want to load, and "file" is the command to load it.

Before we go any further

- gdb has an interactive shell, much like the one you use as soon as you log into the linux grace machines. It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.
- Tip
  - If you're ever confused about a command or just want more information, use the "help" command, with or without an argument:
  - (gdb) help [command]

Running the program

- To run the program, just use:

```
(gdb) run
```

- This runs the program.
  - If it has no serious problems (i.e. the normal program didn’t get a segmentation fault, etc.), the program should run fine here too.
  - If the program did have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum array region (arr=0x7fffc902a270,
r1=2, c1=5, r2=4, c2=6) at sum-array-region2.c:12
```

So what if I have bugs?

- Okay, so you’ve run it successfully. But you don’t need gdb for that. What if the program isn’t working?
- Basic idea
  - Chances are if this is the case, you don’t want to run the program without any stopping, breaking, etc. Otherwise, you’ll just rush past the error and never find the root of the issue. So, you’ll want to step through your code a bit at a time, until you arrive upon the error.
- This brings us to the next set of commands. . .

Setting breakpoints

- Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "break." This sets a breakpoint at a specified file-line pair:

  ```
  (gdb) break file1.c:6
  ```

  This sets a breakpoint at line 6, of file1.c. Now, if the program ever reaches that location when running, the program will pause and prompt you for another command.

- Tip

  You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

More fun with breakpoints

- You can also tell gdb to break at a particular function. Suppose you have a function my func:

```
int my_func(int a, char *b);
```

- You can break anytime this function is called:

```
(gdb) break my_func
```

Now what?

- Once you’ve set a breakpoint, you can try using the run command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing “continue” (Typing run again would restart the program from the beginning, which isn’t very useful.)

```
(gdb) continue
```

- You can single-step (execute just the next line of code) by typing “step.” This gives you really fine-grained control over how the program proceeds. You can do this a lot...

```
(gdb) step
```

Now what? (even more!)

- Similar to “step,” the “next” command single-steps as well, except this one doesn’t execute each line of a sub-routine, it just treats it as one instruction.

  ```
  (gdb) next
  ```

- Tip

  Typing “step” or “next” a lot of times can be tedious. If you just press ENTER, gdb will repeat the same command you just gave it. You can do this a bunch of times.

Querying other aspects of the program

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like the values of variables, etc. This might be useful in debugging. :)
- The print command prints the value of the variable specified:

```
(gdb) print my_var
```

Setting watchpoints

- Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a watched variable's value is modified. For example, the following watch command:

```
(gdb) watch my_var
```

- Now, whenever my_var's value is modified, the program will interrupt and print out the old and new values.
- Tip

  You may wonder how gdb determines which variable named my_var to watch if there is more than one declared in your program. The answer (perhaps unfortunately) is that it relies upon the variable's scope, relative to where you are in the program at the time of the watch. This just means that you have to remember the tricky nuances of scope and extent :(.

Example programs

Other useful commands

- backtrace - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions)
- where - same as backtrace; you can think of this version as working even when you’re still in the middle of the program
- finish - runs until the current function is finished
- delete - deletes a specified breakpoint
- info breakpoints - shows information about all declared breakpoints

More about breakpoints

- Breakpoints by themselves may seem too tedious. You have to keep stepping, and stepping, and stepping. . .
- Basic idea

  Once we develop an idea for what the error could be (like dereferencing a NULL pointer, or going past the bounds of an array), we probably only care if such an event happens; we don’t want to break at each iteration regardless.
- So ideally, we’d like to condition on a particular requirement (or set of requirements). Using conditional breakpoints allow us to accomplish this goal. . .

Conditional breakpoints

- Just like regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger. We use the same break command as before:

```
(gdb) break file1.c:6 if i >= ARRAYSIZE
```

- This command sets a breakpoint at line 6 of file file1.c, which triggers only if the variable i is greater than or equal to the size of the array (which probably is bad if line 6 does something like arr[i]). Conditional breakpoints can most likely avoid all the unnecessary stepping, etc.

Fun with pointers

- Who doesn’t have fun with pointers? First, let’s assume we have the following structure defined:

```
struct entry {
      int key;
      char *name;
      float price;
      long serial_number;
};
```

- Maybe this struct is used in some sort of hash table as part of a catalog for products, or something related.

Using pointers with gdb I

- Now, let’s assume we’re in gdb, and are at some point in the execution after a line that looks like:

```
struct entry * e1 = <something>;
```

- We can do a lot of stu↵ with pointer operations, just like we could in C.
- See the value (memory address) of the pointer:

```
(gdb) print e1
```

- See a particular field of the struct the pointer is referencing:

```
(gdb) print e1->key
(gdb) print e1->name
(gdb) print e1->price
(gdb) print e1->serial number
```

Using pointers with gdb II

- You can also use the dereference (*) and dot (.) operators in place of the arrow operator (->):

```
(gdb) print (*e1).key
(gdb) print (*e1).name
(gdb) print (*e1).price
(gdb) print (*e1).serial number
```

- See the entire contents of the struct the pointer references (you can’t do this as easily in C!):

```
(gdb) print *e1
```

- You can also follow pointers iteratively, like in a linked list:

```
(gdb) print list prt->next->next->next->data
```

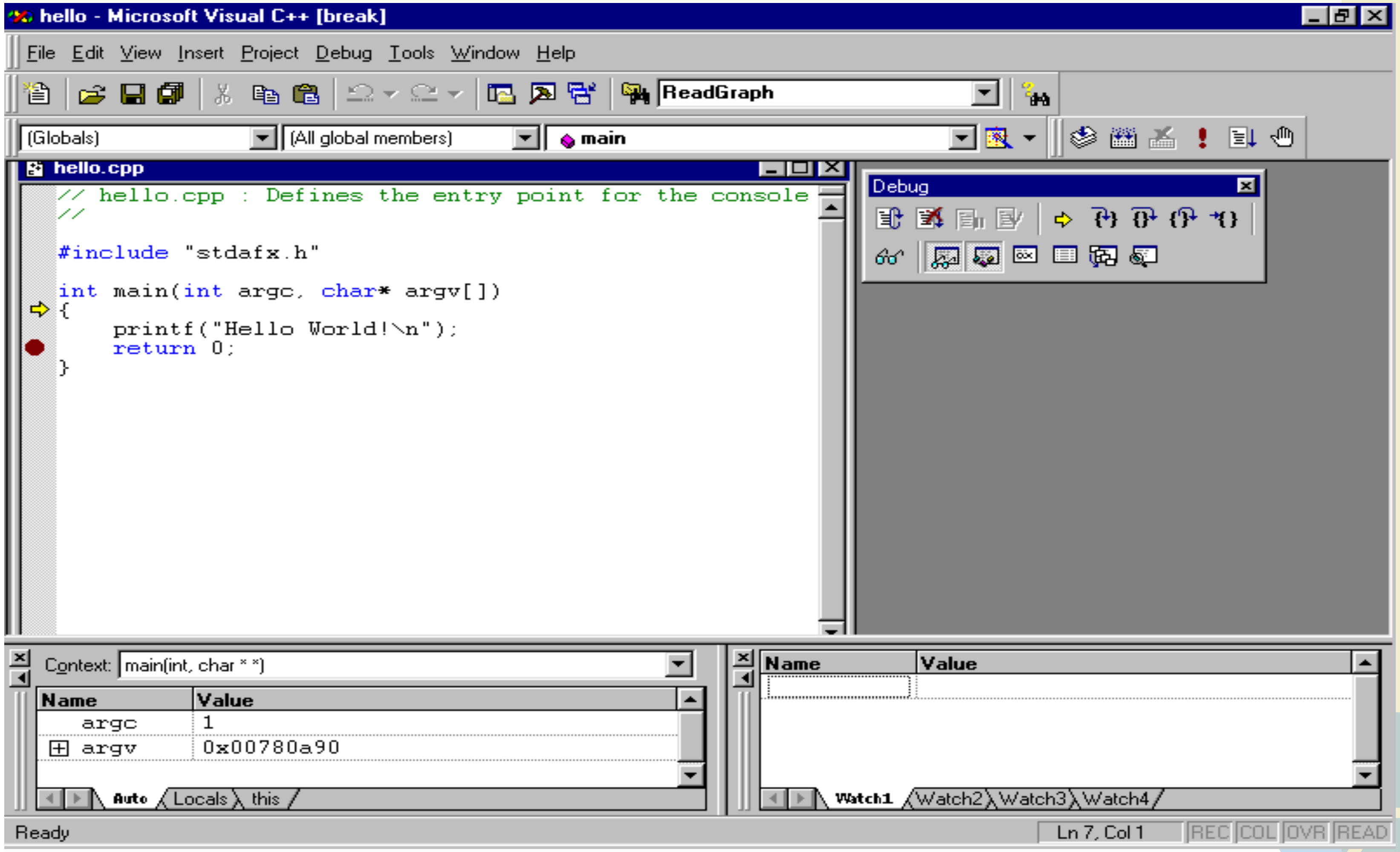
Debuggers

- Debuggers are tools that can examine the state of a running program.
- Common debuggers: adb, dbx, gdb, kdb, wdb, xdb.
- Microsoft Visual Studio has a built-in debugger.
- This talk will focus on the Visual Studio debugger.

Visual Debugger

- Graphically Oriented
- Run from Visual Studio
- Can debug a failed process by selecting the Yes button at “Debug Application” dialog after a memory or other failure occurs
- Can attach to a running process by choosing the Tools->Start Debug->Attach to Process menu option

The Visual Debugger


hello - Microsoft Visual C++ [break]
File Edit View Insert Project Debug Tools Window Help
ReadGraph
(Globals)
(All global members)
main
hello.cpp
Debug
Context: main(int, char * *)
Auto Locals this
Watch1 Watch2 Watch3 Watch4
Ready
Ln 7, Col 1
REC COL OVR READ

```
// hello.cpp : Defines the entry point for the console
//

#include "stdafx.h"

int main(int argc, char* argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

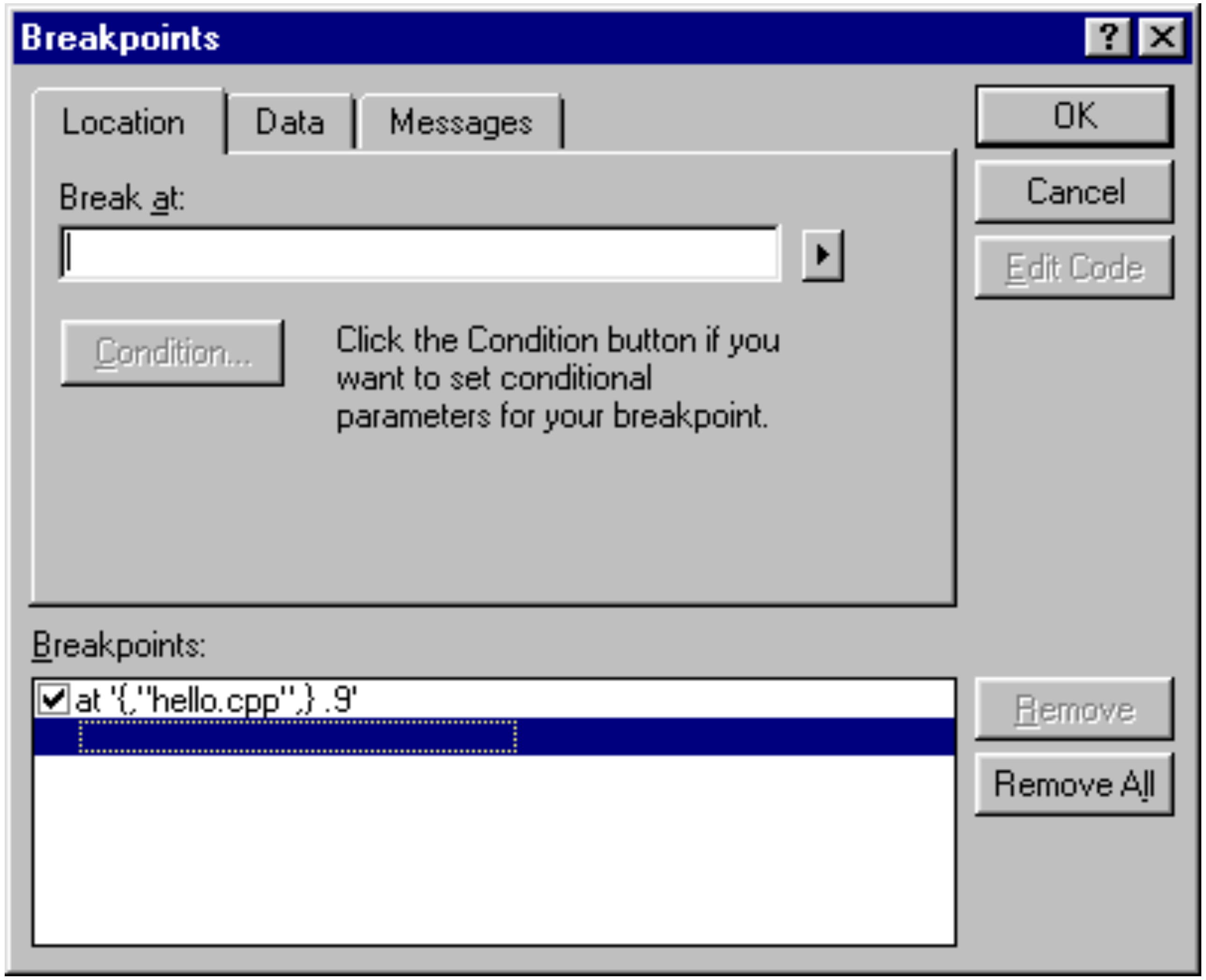
Name	Value
argc	1
⊞ argv	0x00780a90

Name	Value

Breakpoints

- Can stop execution at any line and in any function. (Location)
- Can set conditions on breakpoints if you are only interested in specific passes through a piece of code (Location->Condition)
- Conditional breakpoints detached from any one line in the program are also possible, but make program execution very slow (Data).

Breakpoint Window



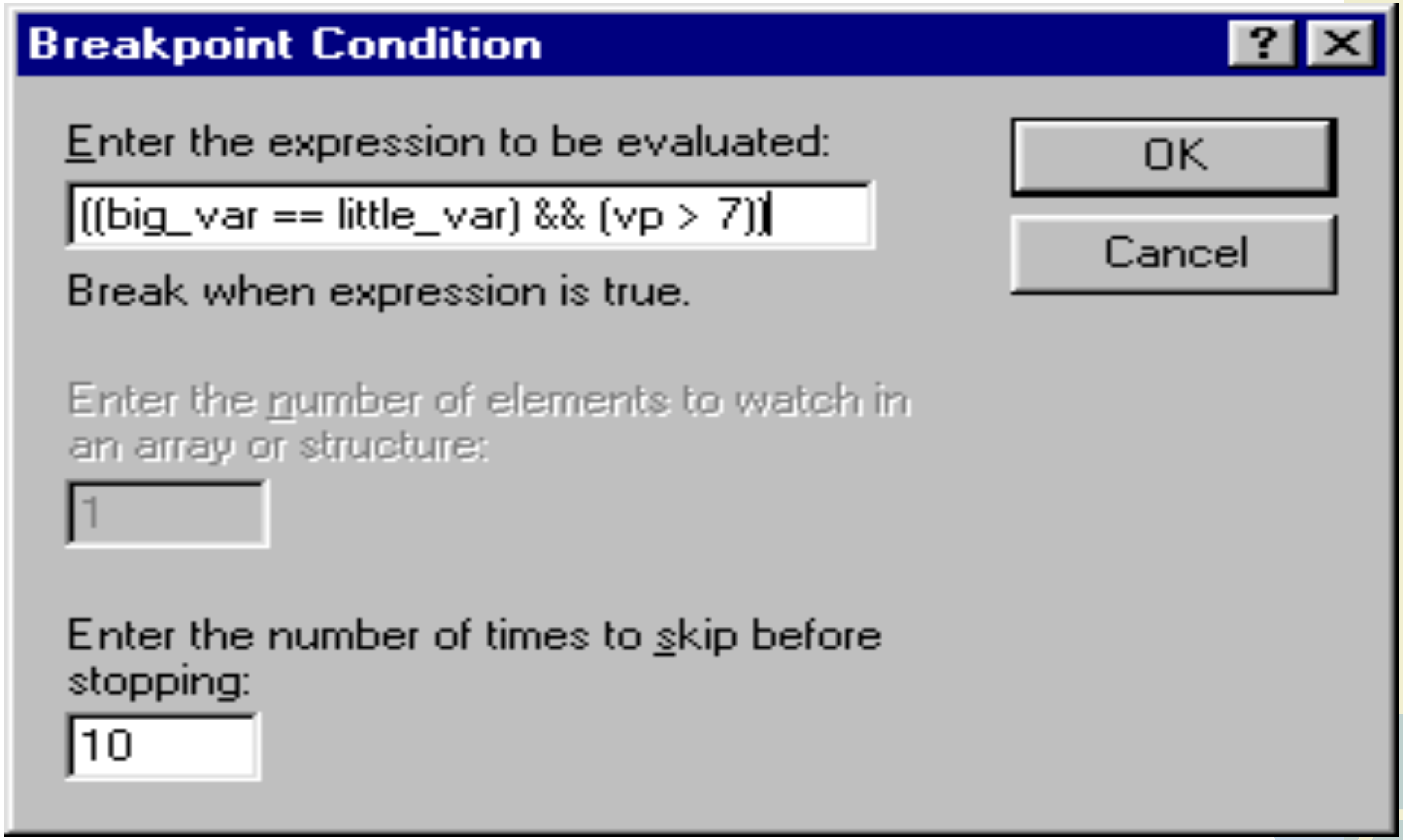
Break at:

Click the Condition button if you want to set conditional parameters for your breakpoint.

Breakpoints:

☑ at '{,"hello.cpp",} .9'

Conditional Window


Breakpoint Condition

Enter the expression to be evaluated:

((big_var == little_var) && (vp > 7))

Break when expression is true.

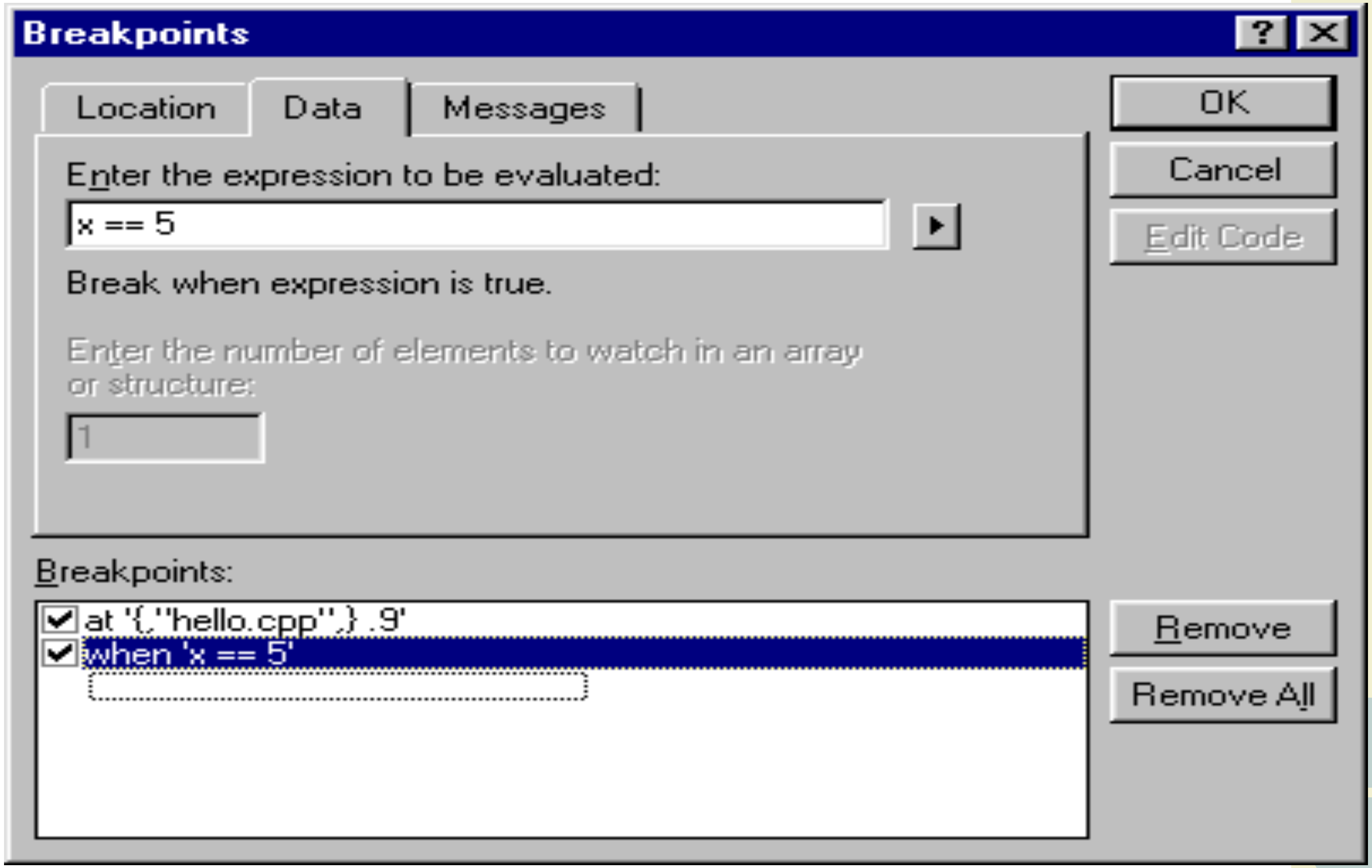
Enter the number of elements to watch in an array or structure:

1

Enter the number of times to skip before stopping:

10

Conditional Data Breakpoint



Enter the expression to be evaluated:

x == 5

Break when expression is true.

Enter the number of elements to watch in an array or structure:

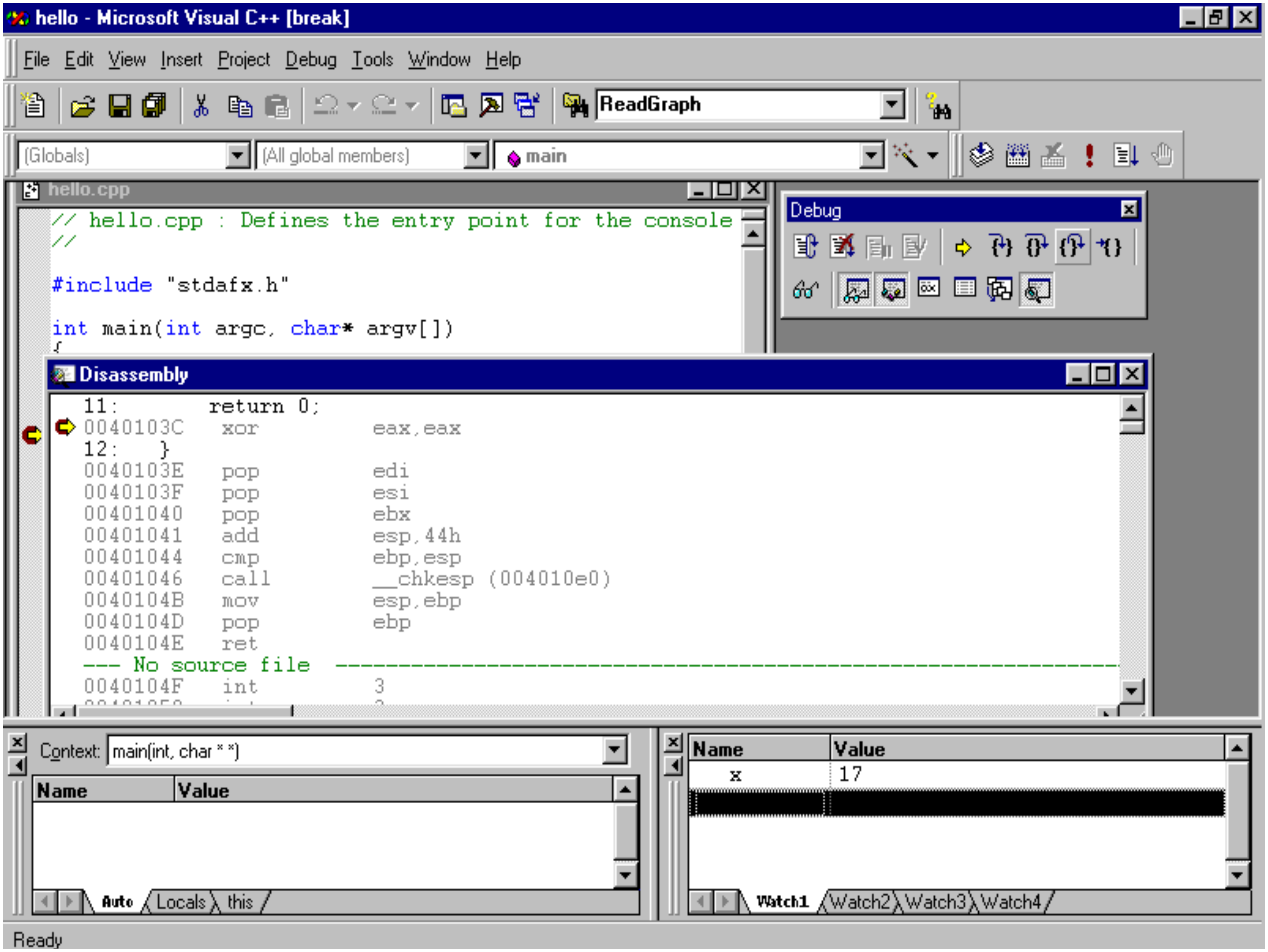
1

Breakpoints:

- [✓] at '{,"hello.cpp",} .9'
- [✓] when 'x == 5'

Examining Program State

- Print and/or Change variable state.
- Walk up/down the stack trace.
- View disassembled code.


hello - Microsoft Visual C++ [break]
File Edit View Insert Project Debug Tools Window Help
ReadGraph
(Globals)
(All global members)
main
hello.cpp
Debug
Disassembly
Context: main(int, char * *)
Auto
Locals
this
Watch1
Watch2
Watch3
Watch4
Ready

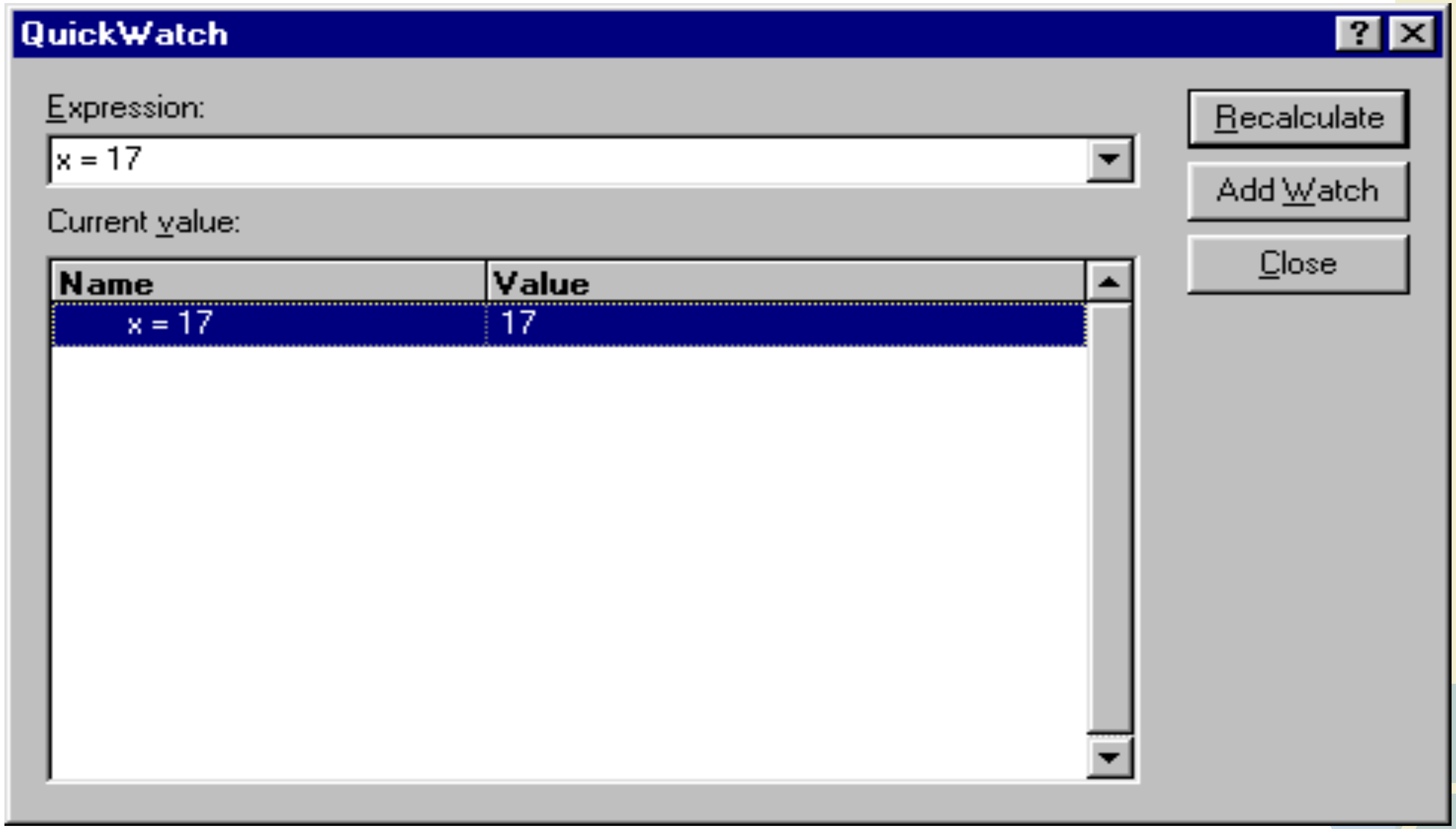
```
// hello.cpp : Defines the entry point for the console
//

#include "stdafx.h"

int main(int argc, char* argv[])
{
```

```
11:        return 0;
0040103C   xor         eax,eax
12:    }
0040103E   pop         edi
0040103F   pop         esi
00401040   pop         ebx
00401041   add         esp,44h
00401044   cmp         ebp,esp
00401046   call        __chkesp (004010e0)
0040104B   mov         esp,ebp
0040104D   pop         ebp
0040104E   ret
--- No source file  ----------------------------------------
0040104F   int         3
```

Name	Value

Name	Value
x	17

Quick Print/Change Variables


QuickWatch

Expression:

```
x = 17
```

Current value:

Name	Value
x = 17	17

Execution Flow

- Step Into - Execute code, step into a function if one is called
- Step Out - Continue execution until N-1’st region of stack frame reached
- Step Over - Execute code, execute any functions that are called without stopping.

Debugging Pointer and Dynamic Data Structure Problems

- Pointers and explicitly allocated dynamic data structures are a central feature in many popular procedural and object-oriented languages
  - Great power - especially in extreme cases (eg C/C++)
  - Can be very painful to debug

Common Pointer Problems

- Pointer to bogus memory
- Corrupt data structure segments
- Data sharing errors
- Accessing data elements of the wrong type
- Attempting to use memory areas after freeing them

Pointers to Bogus Memory

- Uninitialized pointers
- Failing to check memory allocation errors
- Using stomped pointers corrupted by previous memory operations
- Special Case: Indices above/below array space
  - Remedy: index checks

Corrupt Data Structure Segments

- Incorrect Adds/Deletes in trees/lists/etc.
- Stomped pointer values from previous memory operations
  - Remedy 1: Add type info to dynamic data structures
  - Remedy 2: Create routines to check integrity of data structures
  - Remedy 3: Flag deleted memory areas

Data Sharing Errors

- Often share data between logically separate program entities
- Problem 1: Bogus pointer handoff
- Problem 2: Incorrect data format assumptions
- Problem 3: Multiple ownership issues
- Remedy 1: Type info in dynamic data
- Remedy 2: Owner count in memory areas
- Remedy 3: Flag deleted data structures
- Remedy 4: Think through synchronization problems in the design stage

Accessing Elements of Wrong Type

- Access data element of type x, but think you are accessing one of type y
- Can be a source of frequent headaches depending on application/implementation
- Remedy: Include type info in memory allocations

Accessing Data After Freeing It

- Can be a source of many headaches
- Remedy 1: Include freed flag in memory (not a guaranteed solution
- Remedy 2: Create list of “freed” memory, but do not deallocate it. Check list when dereferencing pointers (very expensive in both time and space)
- Big Brother Problem: Accessing data structure after adding it to a “free” list for quick future reuse
- Remedy: Remedy 1 plus a use counter (also not a guaranteed solution)

Final Pointer Comments

- Pointers are powerful, but are often a major source of program errors
- Adding extra state and data structure walk routines can be a big help in debugging (degrades performance/increases memory footprint, but can be removed in release)

Debugging Multitasking Programs

- Multiple process/multi-threaded code ubiquitous in modern programs
- Many debuggers will work with these programs, but it is not always elegant or easy.
- Fallback method: Put new processes to sleep and then attach a debugger to them before they awake.
- Better solution: Read debugger documentation, find better one if it is weak in this area.

A Few Tips

- Pointers and multithreading together can be extremely difficult to debug
- Try to debug parts by themselves before tackling combined system
- Analogous strategies to those used in pointer debugging can be a big help
- Thread/process timing an important concern in the debugging process

Core Dumps

- (Unix) If you run your code outside of the debugger and there is a fault a core file may be generated (depending on your system settings) where the current program state is stored.
- Can debug your code post-mortem via: gdb executable-file core-file

Debug Prompts

- Windows does not use core files.
- If you run your code outside of a debugger and a problem occurs you will be given the option of either debugging the code or killing the executing process.

Abort Signal (Unix)

- You can use the abort signal to help determine the cause of your problem
- SIGBUS: Likely a dereference of a NULL pointer
- SIGSEGV: Likely a dereference of a bogus pointer, an invalid write to code space, or a bad branch to data space
- SIGFPE: Division by zero

Blame the Compiler

- Sometimes software crashes in debugged code but not in optimized code
- The tendency is to blame the compiler and de-optimize the file or function where the bug occurred
- Most often the problem is in the code and is just exposed by the optimizer, typically an uninitialized global variable
- Of course, sometimes it really is an optimizer bug. In that case, please submit a bug report to the compiler vendor with a nice short test program