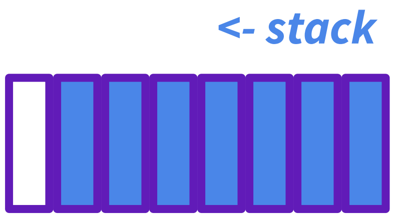
CSE 374 Lecture 11

Debugging *DGB*

The stack



Stack stores active functions & <u>local</u> variables

Each function gets a frame, moving down in memory

Last frame is completed, deleted

then the next most recent frame.

(Last in-first out)

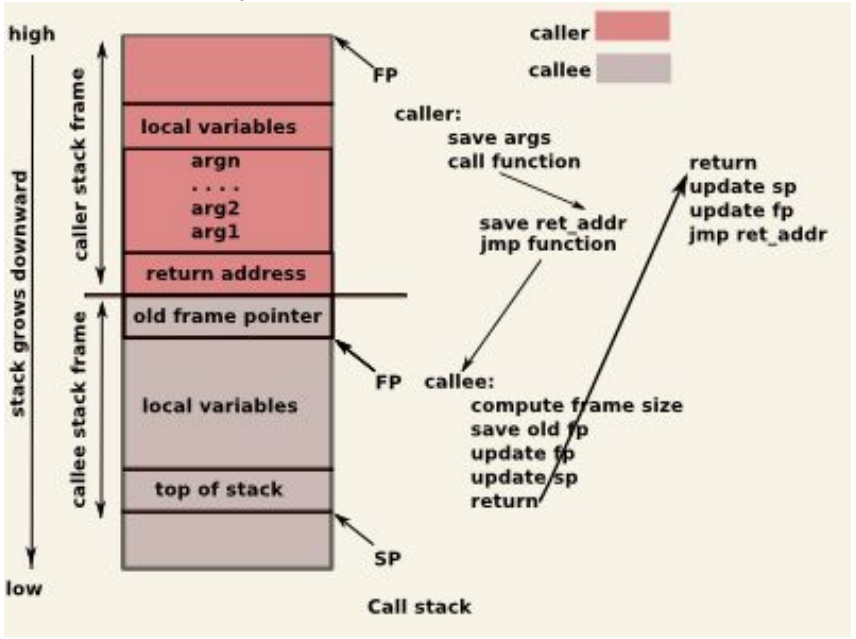
Each function call creates a frame

Containing:

Arguments, return address,

Pointer-to-last-frame,

local variables



How to avoid debugging

Avoid writing code

How to avoid debugging

- Don’t put bugs in the program!!
- Think before typing – design before coding
- Write down design (comments) as you go
  - Functions: declaration+comments should be complete spec
  - Significant data: declaration + comments should be complete spec
  - If someone has to read the code to figure out how to use something or understand data structures, comments are bad
- Review/check comments and compare to code as you work; Will catch errors before you run the program
- Follow coding conventions
- Turn on compiler warnings (-Wall); use assert; get the computer to find problems for you.

How to make debugging less painful

1. Don't Panic
2. Be systematic
3. Test theories
4. Practice
5. Test early and often

Debugging techniques (basic)

- Comment out (or delete) code
  - tests to determine whether removed code was the source of the problem
- Test one function at a time
  - Like you commented out most of the code….
- Add print statements
  - Says ‘I got here’, or ‘my variable value is 52’
- Test the edges
  - Code often breaks at the beginning or end of a loop, or at the entry or exit of a function; double check your logic in these places
  - Double check your logic in the odd / rare exceptional cases

Can a debugger help?

A “debugger” is a tool that lets you stop running programs, inspect (sometimes set) values, etc.

Instead of relying on changing code (commenting out, printf) interactively examine variable values, pause, and progress step-by-step.

(So, yes.)

Don’t expect the debugger to do the work; use it as a tool to test your theories

Most modern IDEs have built in debugging functionality

What about gdb

‘Gdb’ => gnu debugger (standard part of linux development, supports many languages)

https://courses.cs.washington.edu/courses/cse374/19sp/refcard.pdf

Techniques are the same as in most debugging tools

Can examine a running file

Can also examine ‘core’ files of previous crashed programs…. Neat!

Takes some time to get facile; worth it.

Run gdb

1. Compile code with ‘-g’ flag (saves human readable info)
2. Open the program with: gdb <executable file>
3. Start or restart the program: run <program args>
    a. Quit the program: kill
    b. Quit gdb: quit
4. Reference information: help
5. Most commands have short abbreviations
6. <return> often repeats the last command
    a. Particularly useful when stepping through code

GDB summary - looking around

- bt – stack backtrace
- up, down – change current stack frame
- list – display source code (list n, list <function name>)
- print expression – evaluate and print expression
- display expression
  - (re-)evaluate and print expression every time execution pauses.
  - undisplay – remove an expression from this recurring list.
- info locals – print all locals (but not parameters)
- x (examine) – look at blocks of memory in various formats

What are breakpoints?

- Temporarily stop program running at given points
  - Look at values in variables
  - Test conditions
- break function (or line-number or ...)
- conditional breakpoints (break XXX if expr)
  - to skip a bunch of iterations
  - to do assertion checking
- going forward: continue, next, step, finish
  - Some debuggers let you “go backwards” (typically an illusion)
- Also useful for learning program structure (e.g., when is a function called)

Breakpoints

- break – set breakpoint.
    - break <function name>, break <linenumber>, break <file>:<linenumber>
- info break – print table of currently set breakpoints
- clear – remove breakpoints
- disable/enable – temporarily turn breakpoints off/on
- continue – resume execution to next breakpoint or end of program
- step – execute next source line
- next – execute next source line
    - But treat function calls as a single statement and don't step into them
- finish – execute to the conclusion of the current function
    - How to recover if you meant “next” instead of “step”

Debugging thoughts

- There are two major kinds of debugging

Getting code to run without crashing **Getting code to do what you want**

- Gdb (and other tools) can help with both
- The former is harder in C than Java, because the compiler does less work for C
- But, a ‘working’ program is not necessarily a ‘correct’ program
- The logic of the latter is often the same
  - a running program is not necessarily a correct program

Preview of coming attractions

Dynamically allocating memory - assigns memory from the heap

```
void* malloc (size_t size);
```

Allocate memory block

Allocates a block of size bytes of memory, returning a pointer to the beginning of the block.
The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.

If size is zero, the return value depends on the particular library implementation (it may or may not be a *null pointer*), but the returned pointer shall not be dereferenced.

Preview of coming attractions

Dynamically allocating memory

```
void free (void* ptr);
```

Deallocate memory block

A block of memory previously allocated by a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.
If ptr does not point to a block of memory allocated with the above functions, it causes *undefined behavior*.

If ptr is a *null pointer*, the function does nothing. Notice that this function does not change the value of ptr itself, hence it still points to the same (now invalid) location.