Introduction to GDB, pointers and pointer arithmetic

Michael Kirkedal Thomsen

CompSys'20, DIKU

September 9, 2020, lecture 4

Today´s plan

- GDB/LLDB background
- GDB/LLDB at a glance
- Memory organisation
- Pointers and pointer arithmetic

GDB background

GDB was first written by Richard Stallman
in 1986 as part of his GNU system

- Richard Stallman, “Debugging with gdb”
www.gnu.org/software/gdb/documentation

LLDB is a newer version debugger with a GDB-like interface, which is part of the LLVM project.

Linux/Windows users can use either, MacOS users use the pre-installed LLDB.

Lest's debug a program

```c
#include <stdio.h>

#define square(x) x*x

int main() {
  int a = square(3+5);
  int b = a*2;
  int c = b/4;
  printf("%d\n", c);
  return 0;
}
```

The is an error in the program. Result is 11 but expected result is 32.

Running GDB

Compile your program with g-flag

```
> gcc -g program
```

Command-line debugging:

```
> gdb [program]*
```

Text-based user-interface debugging:

```
> gdb -tui [program]*
```

Emacs based debugging:

```
M-x gdb
```

Essential commands

- `run [arglist]` : Running you program
- `p(rint) expr` : evaluate and print an expression
- `b(reak) line` : insert a break point
- `c(ontinue)` : continue to next break point
- `s(tep)` : Step one line
- `n(ext)` : Step one line entering function calls
- `q(uit)` : leave GDB again

GDB - breakpoints

Breakpoints are stops in code where you can freeze program operation. Breakpoints are useful in order to stop and evaluate your program.

- `b(reak)` : create breakpoint at current line
- `break n` : create breakpoint at line x
- `break file:n` : create breakpoint at line x in file
- `break function` : create breakpoint at function
- `clear n` : remove breakpoint at line x
- `info break` : list break points

GDB - advanced breakpoints

- `watch expr` : watch an expression
- `disable n` : disable breakpoint
- `enable [once] n` : enable a breakpoins for first visit
- `ignore n count` : Ignore a breakpoint `count` times

GDB - execution

At some point, we need to step through the program and analyze it line by line.

- `n(ext)` : process the next line of code in function
- `s(tep)` : if code is function, go into it
- `c(ontinue)` : run program until next breakpoint or finish

GDB - examination

- `p(rint) [/f] expr` : print the value of an expression; often used for variables
  - You can add special formatting: `d` : signed, `u` : unsigned, `x` : hex, `t` : binary, etc.
- `$n` : use as variables
- `define id cmd end`
- You can use GDB as a interactive interpreter

Example

```
(gdb) define square
print $arg0 * $arg0
end
(gdb) print 3 + 5
$1 = 8
(gdb) square $1
```

Back to my program

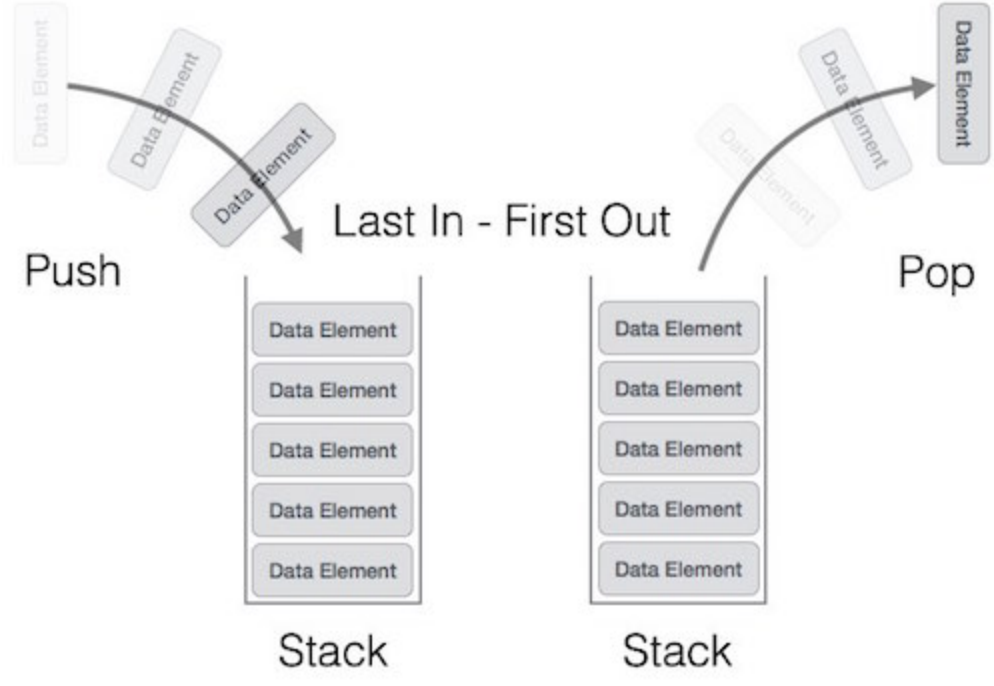
```c
#include <stdio.h>

#define square(x) x*x

int main() {
  int a = square(3+5);
  int b = a*2;
  int c = b/4;
  printf("%d\n", c);
  return 0;
}
```

- Macro distributes over the expression

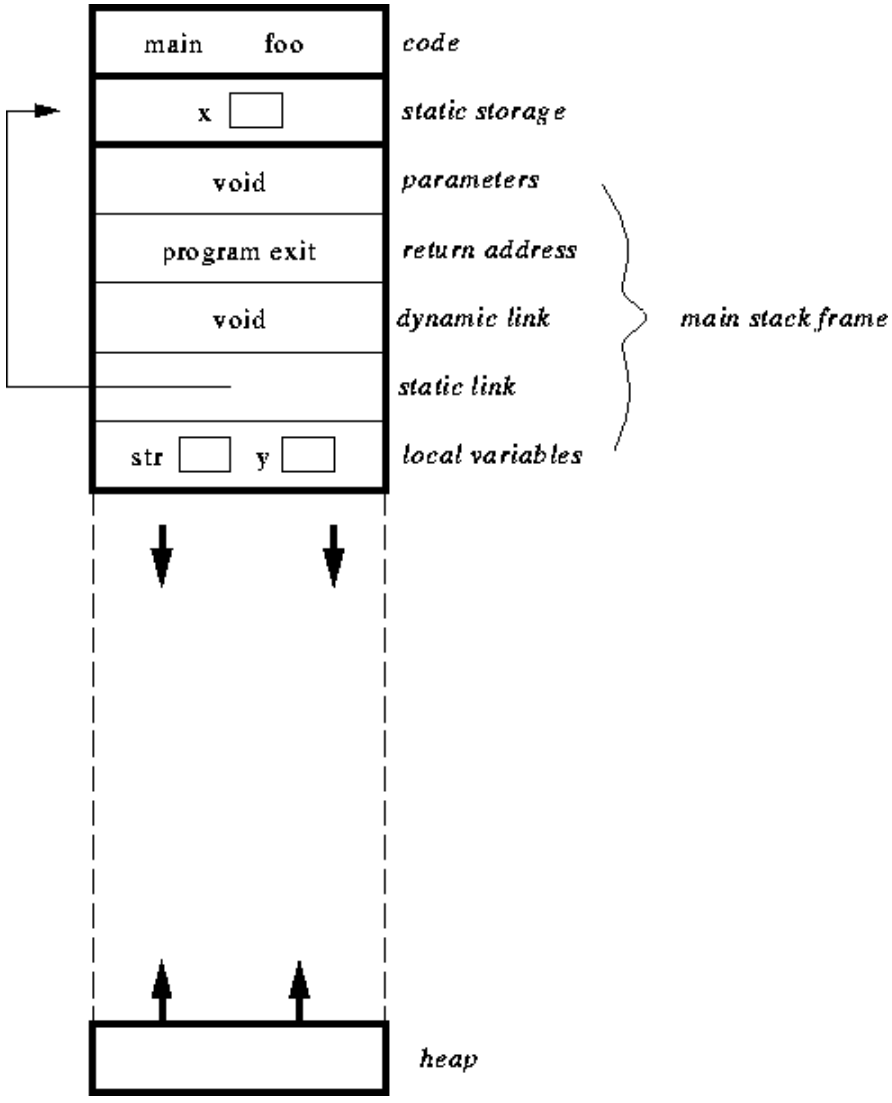
The stack at a glance

The stack at a glance

The stack at a glance



The program stack



Convention dictates that stacks grows downwards. // Thus the bottom of a stack is

Example

```c
int tester(int* c, int k);

int main() {
  int i, k;
  int x[1000];

  for(i = 0; i < 10000; ++i){
    x[i] = i;
  }

  printf("Enter integer in 0..9999: ");
  scanf("%d", &k);

  tester(x, k);
}

int tester(int* c, int k) {
  printf("x[%d] = %d\n", k, c[k]);
  return 1;
}
```

GDB - stack frames and addresses

Segmentation faults are difficult as it can be annoying and difficult to step until a failure. Rather than step through code, just let the program fail and run a backtrace.

- `backtrace` : print the function calls leading up to the current line
- `up n` : move up the call stack
- `down n` : move down the call stack
- `info frame` : information about the current frame
- `info args` : information on arguments to function call of frame
- `info locals` : information about local variables
- `x expr` : examine memory about expression

GDB scripts - automisation

What GDB Does During Startup

1. Executes all commands from system init file
2. Executes all the commands from `~/.gdbinit`
3. Process command line options and operands
4. Executes all the commands from `./.gdbinit`
5. Reads command files specified by the `-x` flag
6. Reads the command history recorded in the history file.

Automate by updating `.gdbinit`.

Questions?