#preprocessor

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

--Brian W. Kernighan

C preprocessor

- Part of the C compilation process; recognizes special # statements, modifies your source code before it is compiled
- This is generally considered to be a "macro processing" tool

function	description
`#include <filename>`	insert a library file's contents into this file
`#include "filename"`	insert a user file's contents into this file
`#define name [value]`	create a preprocessor symbol ("variable")
`#if test`	`if` statement
`#else`	`else` statement
`#elif test`	`else if` statement
`#endif`	terminates an `if` or `if/else` statement
`#ifdef name`	`if` statement; `true` if ***name*** is defined
`#ifndef name`	`if` statement; `true` if ***name*** is *not* defined
`#undef name`	deletes the given symbol name

Constant replacement

- The preprocessor can be used to create constants:

```
#define NUM_STUDENTS  100
#define DAYS_PER_WEEK 7
...

double grades[NUM_STUDENTS];
int six_weeks = DAYS_PER_WEEK * 6;   // 42
printf("Course over in %d days", six_weeks);
```

- When the preprocessor runs before compilation, `7` is literally inserted into the code wherever `DAYS_PER_WEEK` is seen: `DAYS_PER_WEEK` does not exist in the eventual program

```
int six_weeks = 7 * 6;   // 42
```

Optional debugging code

```
#define DEBUG
...

#ifdef DEBUG
    // debug-only code
    printf("Size of stack = %d\n", stack_size);
    printf("Top of stack  = %p\n", stack);
#endif
    stack = stack->next;    // normal code
```

- How is this different from declaring a bool/int named `DEBUG`?

Advanced definitions

- `#define` can be used to modify the C language
- (No different in mechanism than constants)

Good idea? Bad idea? Neutral idea?

```
#define AND    &&
#define EQUALS ==
#define DEREF  ->
...

Point p1 = (Point*) malloc(sizeof(Point));
p1 DEREF x = 10;
p1 DEREF y = 10;
if (p1 DEREF x EQUALS p1 DEREF y AND p1 DEREF y > 0) {
    p1 DEREF x++;
}
```

Preprocessor macros

- Similar to a function, but created inline before compilation

```
#define SQUARED(x)   x * x
#define ODD(x)       x % 2 != 0
int a = 3;
int b = SQUARED(a);
if (ODD(b)) {
    printf("%d is an odd number.\n", b);
}
```

- The above literally converts the code to the following and compiles:

```
int b = a * a;
if (b % 2 != 0) { ...
```

Subtleties

- the preprocessor just replaces tokens with tokens

```
#define foo 42
int food = foo;        // int food = 42;      ok
int foo = foo + foo;   // int 42 = 42 + 42;   bad
```

- preprocessor macros can do a few things functions cannot:

```
#define NEW(t)   (t*) calloc(1, sizeof(t))
...

Node* list = NEW(Node);
```

Caution with macros

- since macros are expanded directly, strange things can happen if you pass them complex values

```
#define ODD(x)       x % 2 != 0
...
if (ODD(1 + 1)) {
    printf("It is odd.\n");    // prints!
}
```

- The above converts the code to the following

```
if (1 + 1 % 2 != 0) {
```

- Fix: Always surround macro parameters in parentheses.

```
#define ODD(x)       (x) % 2 != 0
```

Running the preprocessor

- to define a preprocessor variable, use the `-D` argument

```
$ gcc -D DEBUG -o example example.c
```

- to run only the preprocessor, use the `-E` argument to `gcc`

```
$ gcc -E example.c
int main(void) {
    if ((1 + 1) % 2 != 0) {
        printf("It is odd.\n");
    }
    return 0;
}
```

- rarely used in practice, but can be useful for debugging / learning

gdb

- gdb : GNU debugger. Helps you step through C programs.
  - absolutely essential for fixing crashes and bad pointer code
  - your program must have been compiled with the `-g` flag
- usage

```
$ gdb program
GNU gdb Fedora (6.8-23.fc9)
Copyright (C) 2008 Free Software Foundation, Inc...
(gdb) run parameters
...
```

- redirecting input:

```
$ gdb program
(gdb) run parameters < inputfile
```

gdb commands

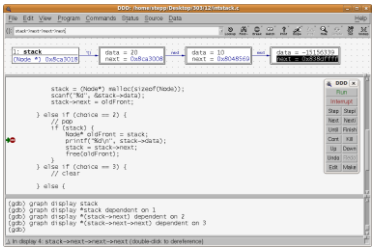
command	description
run or **r** *parameters*	run the program
break or **b** *place*	sets a breakpoint at the given place: - a function's name - a line number - a source file : line number
print or **p** *expression*	prints the given value / variable
step or **s**	advances by one line of code ("step into")
next or **n**	advances by one line of code ("step over")
finish	runs until end of function ("step out")
continue or **c**	resumes running program
backtrace or **bt**	display current function call stack
quit or **q**	exits gdb

A gdb session

```
$ gdb intstack
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
(gdb) b 34
Breakpoint 1 at 0x4010ea: file intstack.c, line 34.
(gdb) r
Starting program: /home/user/intstack
Breakpoint 1, main () at intstack.c:34
34                      Node* oldFront = stack;
(gdb) p stack
$1 = (Node *) 0x4619c0
(gdb) n
35                      printf("%d\n", stack->data);
(gdb) n
36                      stack = stack->next;
(gdb) n
37                      free(oldFront);
(gdb) p stack
$4 = (Node *) 0x462856
(gdb) p oldFront
$2 = (Node *) 0x4619c0
(gdb) p *oldFront
$3 = {data = 10, next = 0x462856}
(gdb) c
Continuing.
```

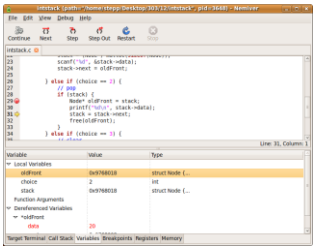
ddd

- ddd (Data Display Debugger): Graphical front-end for gdb
  - allows you to view the values of your variables, pointers, etc.
  - **`$ ddd programName`**



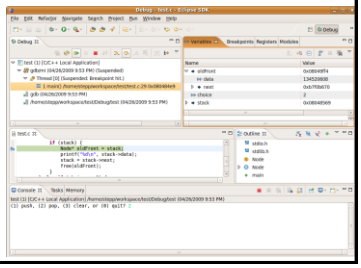
nemiver

- nemiver : Another graphical debugger front-end
  - design goal: Be usable even if you don't know gdb commands
  - **`$ nemiver programName arguments`**



Other debuggers

- Eclipse CDT (C/C++ Development Toolkit)
  - create a new Managed Make C Project
  - right-click project name, choose Debug As, Local C/C++ Application



valgrind

- valgrind : A memory-leak detector and debugging tool
  `valgrind programName arguments`

```
(1) push, (2) pop, (3) clear, or (0) quit? 2
==3888== Conditional jump or move depends on uninitialised value(s)
==3888==    at 0x80484E7: main (intstack.c:28)
==3888==
==3888== Use of uninitialised value of size 4
==3888==    at 0x80484F2: main (intstack.c:30)
-15156339
==3888==
==3888== Use of uninitialised value of size 4
==3888==    at 0x8048507: main (intstack.c:31)
==3888==
==3888== Invalid free() / delete / delete[]
==3888==    at 0x4025DFA: free (vg_replace_malloc.c:323)
==3888==    by 0x8048517: main (intstack.c:32)
==3888==  Address 0x8048569 is in the Text segment of
   /home/stepp/intstack
```

Valgrind: leaks, too

- stats about leaked memory on program exit

```
(1) push, (2) pop, (3) clear, or (0) quit? 1
Number to push? 10
(1) push, (2) pop, (3) clear, or (0) quit? 1
Number to push? 20
(1) push, (2) pop, (3) clear, or (0) quit? 2
20
(1) push, (2) pop, (3) clear, or (0) quit? 2
10
(1) push, (2) pop, (3) clear, or (0) quit? 0

==5162== LEAK SUMMARY:
==5162==    definitely lost: 16 bytes in 2 blocks.
==5162==      possibly lost: 0 bytes in 0 blocks.
==5162==    still reachable: 0 bytes in 0 blocks.
==5162==         suppressed: 0 bytes in 0 blocks.
```

lint / splint: checks for possible errors

- famously picky (sometimes should be ignored)
- but good for helping you find potential sources of bugs/errors
- not installed on attu, but can install it on your Linux:
  `$ sudo apt-get install splint`

```
$ splint *.c
Splint 3.1.2 --- 07 May 2008
part2.c: (in function main)
part2.c:8:2: Path with no return in function declared to return int
 There is a path through a function declared to return a value on which
   there
 is no return statement. This means the execution may fall through
   without
 returning a meaningful result to the caller. (Use -noret to inhibit
   warning)

use_linkedlist.c:5:5: Function main defined more than once
 A function or variable is redefined. One of the declarations should
   use
 extern. (Use -redef to inhibit warning)

part2.c:8:1: Previous definition of main
```

Questions?