# Linked-Lists, Strings Manipulation, & Debugging

## CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

Feb 13, 2023

# Lesson Plan

1. Linked-Lists
2. File I/O (Review)
3. Strings (Manipulation)
4. Debugging Tools: `gdb`, `valgrind`


## Acknowledgements

Part of the slides are borrowed from Andi Bergen.

Section 1

# Linked-Lists

# Linked-Lists

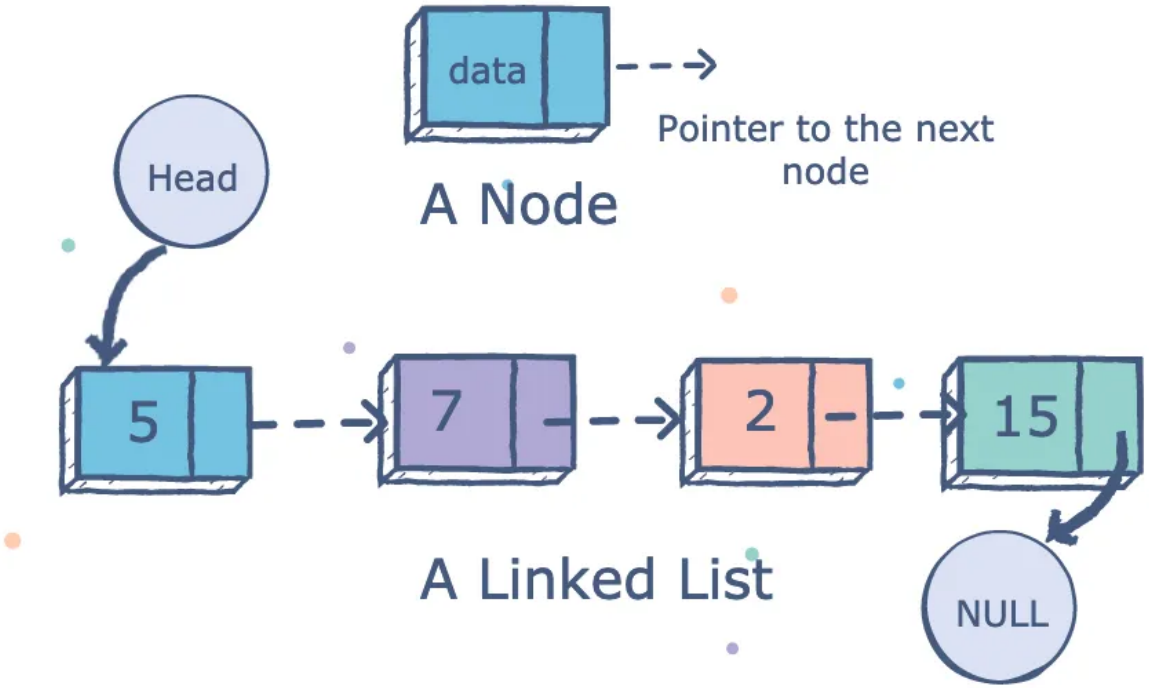


Figure 1: Source: `www.educative.io`

A very common data structure that maintains ordering with easy insertions/deletions is the linked-list (LL).

# Linked-List Nodes (LLNode)

Here's a sample `struct` implementation of a linked-list node.

```
typedef struct llnode {
    struct llnode * next;
    int data;
} LLNode;
```

Each holds a pointer `next` to another `LLNode`, so they can refer to each other.

```
LLNode b = {NULL, 1};  // b doesn't point to any other node.
LLNode a = {&b, 0};    // a.next is a pointer to b
```

`LLNode` structs are often dynamically allocated and referenced by pointers.

```
LLNode *c = malloc(sizeof(LLNode)),
       *d = malloc(sizeof(LLNode)),
       *e = malloc(sizeof(LLNode));
(*c).next = d;
d->next = e;
```

# Linked-Lists: Pointers

The head – representing the first node or entry point – is just like any other node, but its address is the one that's typically passed around to represent the entire linked-lists.

Pointers to `LLNode` objects can be `NULL`, so you may want to check before accessing `next` or `data`. But this can be useful to check for the end of a linked-list!

```
LLNode *node; // suppose it was initialized somewhere...
while (node != NULL) { // iterate until the end
    // suppose you're doing some stuff with node
    node = node->next;
}
```

The tail is the "last" node in the list, and stores a `NULL`-pointer for `next`.

- If you want to stop at a specific node, you can pass the pointer to a *pseudo*-tail.

```
while (node != tail && node != NULL)
    node = node->next;
```

# Linked-Lists: Other Info

1. How big is a linked-list?
2. How can one make a linked-list lightweight?
3. Does its size depend on the objects whose order they represent?
4. Are LLs finite? Does iteration always terminate?
5. What's the complexity of a search? Insertion? Deletion? Update?

# Linked-Lists: Other Info

1. How big is a linked-list? What about the head?
   - *Depends on what's stored (pointers, primitives, structs?) in the nodes. The head is the same size as any node.*
2. How can one make a linked-list lightweight?
   - *Just use pointers to anything that is bigger than a pointer!*
3. Does its size depend on the objects whose order they represent?
   - *Depends on how many primitives and pointers you've got.*
4. Are LLs finite? Does iteration always terminate?
   - *LLs themselves are bounded (by application or memory) but iteration can be infinite if nodes pointers form a cycle.*
5. What's the complexity of a search? Insertion? Deletion? Update?
   - *All of these are linear: $\mathcal{O}(n)$! These can be improved by combining LL pointers with other data structures you might see in CSC263/265 and CSC373.*

# Linked-Lists: Operations

Aside from searching, you can insert/delete/update/move nodes relatively easy.

```
LLNode *prev, *new, *target; // suppose they're initialized somewhere...
while (node != NULL) {
    if (node != target) {
        new = node->next; prev = node; node = new;
        continue; }
    switch (op) { // assume node == target
        case INSERT: // insert new after target
            new->next = node->next->next;
            node->next = new; break;
        case DELETE: // remove target
            // how would you handle special case of deleting head?
            prev->next = node->next;
            recursive_free(node); break;
        case UPDATE: node->data = 209; break;
    } break; }
```

# Linked-Lists: Operations

Moving `target` can be done by removing but not freeing `target`, and then re-inserting somewhere else in the LL. There's probably other stuff you might do!

## Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/linkedList.zip`

Section 2

# File Input/Output (I/O)

# File I/O: File Descriptors

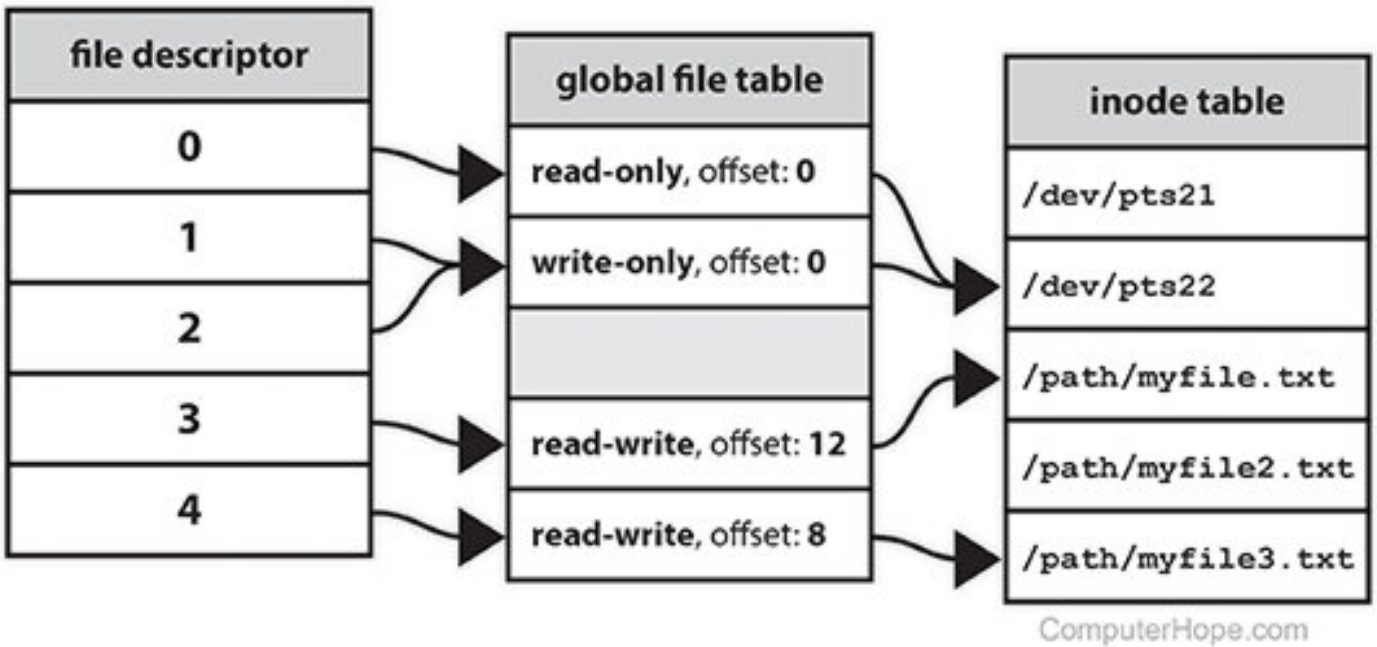


Figure 2: Example of FDs pointing to inodes.

A file descriptor (FD) is a number (non-negative integer) that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.

`www.computerhope.com/jargon/f/file-descriptor.htm`

# File I/O: Streams, File Descriptors & Pointers

Streams are files to which data is written or from which data is read. They're accessed through file pointers (`FILE *` from `<stdio.h>`) that wrap around FDs. The following default streams (FDs) are provided by `<stdio.h>`.

- `stdin` (0): default input; typically from user keyboard or pipes.
- `stdout` (1): default output; usually to terminal screen or pipes.
- `stderr` (2): default error; also to terminal screen.
- Use > to *redirect* `stdout`, and 2> to redirect `stderr`
  - > overwrites the output file, >> appends to it.

# File I/O: Opening a File with Modes

```
FILE *fopen(const char *filename, const char * mode);
```

A file `filename` is opened with `fopen()` in a `mode {r|w|a}{|+}` to perform the following operations. Returns a file pointer that wraps the FD. The pointer is `NULL` if we fail to open the file (often because the file doesn't exist or your process doesn't have permission).

| action\mode | `r` | `w` | `a` | `r+` | `w+` | `a+` |
|---|---|---|---|---|---|---|
| read | yes | no | no | yes | yes | yes |
| write | no | yes | no | yes | yes | yes |
| append | no | no | yes | no | no | no |
| file exists | ok | ok | ok | ok | truncate | append |
| doesn't exist | fail | create | create | fail | ok | create |

# File I/O: Closing a File: `fclose()`

```
int fclose(FILE *stream);
```

- `stream`: a `FILE *` opened by `fopen()`/`freopen()`.
- returns: `0` if closed properly, `EOF` otherwise.

You should always close files (as soon as possible) when you're done with them.

```
if ((fp = fopen("doesnt_exist.txt", "a")) == NULL) {
    fclose(fp);
    return 1;
}
```

# File I/O: Reading & Writing

## Reading

1. `getchar()`: read a character from `stdin`.
2. `fgetc()`: read a single character from the file.
3. `fgets()`: read strings from files.
4. `fscanf()`: formatted input from a file.
5. `fread()`: block of raw bytes from files; useful for binary files.

Examples: `e1.c`, `e2.c`.

## Writing

You can use `putchar()` to write a character to `stdout`.

```
size_t fwrite(const void *ptr, size_t size,
    size_t nmemb, FILE *stream)
```

Alternatively, use `fwrite()` to write `nmemb` elements (each `size` large) from `*ptr` to `stream`.

Section 3

# Strings Manipulation

# Strings: Just Spicy Arrays

"Strings" in C are actually a special case of `char` arrays: they're **null-terminated**, meaning the last *actual* character is `\0`.

```
char limited[9]; // such a string shouldn't exceed 8
```

- When working with strings, `\0` isn't typically used/printed.
- Instead, it indicates where the string ends.
  - If you're writing a function that doesn't know the exact length of the string, the `\0` might come in handy.
  - Inserting a `\0` in the middle of a `char *` shortens the effective string.

    ```
    char *city = "mississauga";
    city[4] = '\0'; // city is now "miss"
    ```

- Declaring strings with explicit length initializes remainder as `\0`.
- Important for many string-wise functions.

# Strings Manipulation: Some Library Functions (`string.h`)

Many string manipulation functions can be found in `string.h`.

```c
// copies the string pointed to, by src to dest.
char *strcpy(char *dest, const char *src);

// appends the string pointed to, by src to the
// end of the string pointed to by dest.
char *strcat(char *dest, const char *src);

// computes the length of the string str up to
// but not including the terminating null character.
size_t strlen(const char *str);

// compares the string pointed to,
// by str1 to the string pointed to by str2.
int strcmp(const char *str1, const char *str2);
```

# Strings Manipulation: More Library Functions (stdio.h/string.h)

sprintf and gets are provided by stdio.h.

```
// sends formatted output to a string.
int sprintf(char *str, const char *format, ...)

// get string from stdin.
// discontinued in C11/C++14. Why?
char *gets (char *str);
```

**C23: The next generation of the C standard**

In the C23 standard, strdup was introduced. It returns a pointer to a null-terminated byte string on the heap, which is a duplicate of the string pointed to by str1.

```
char *strdup(const char *str1);
```

# Strings Manipulation: Memory Errors

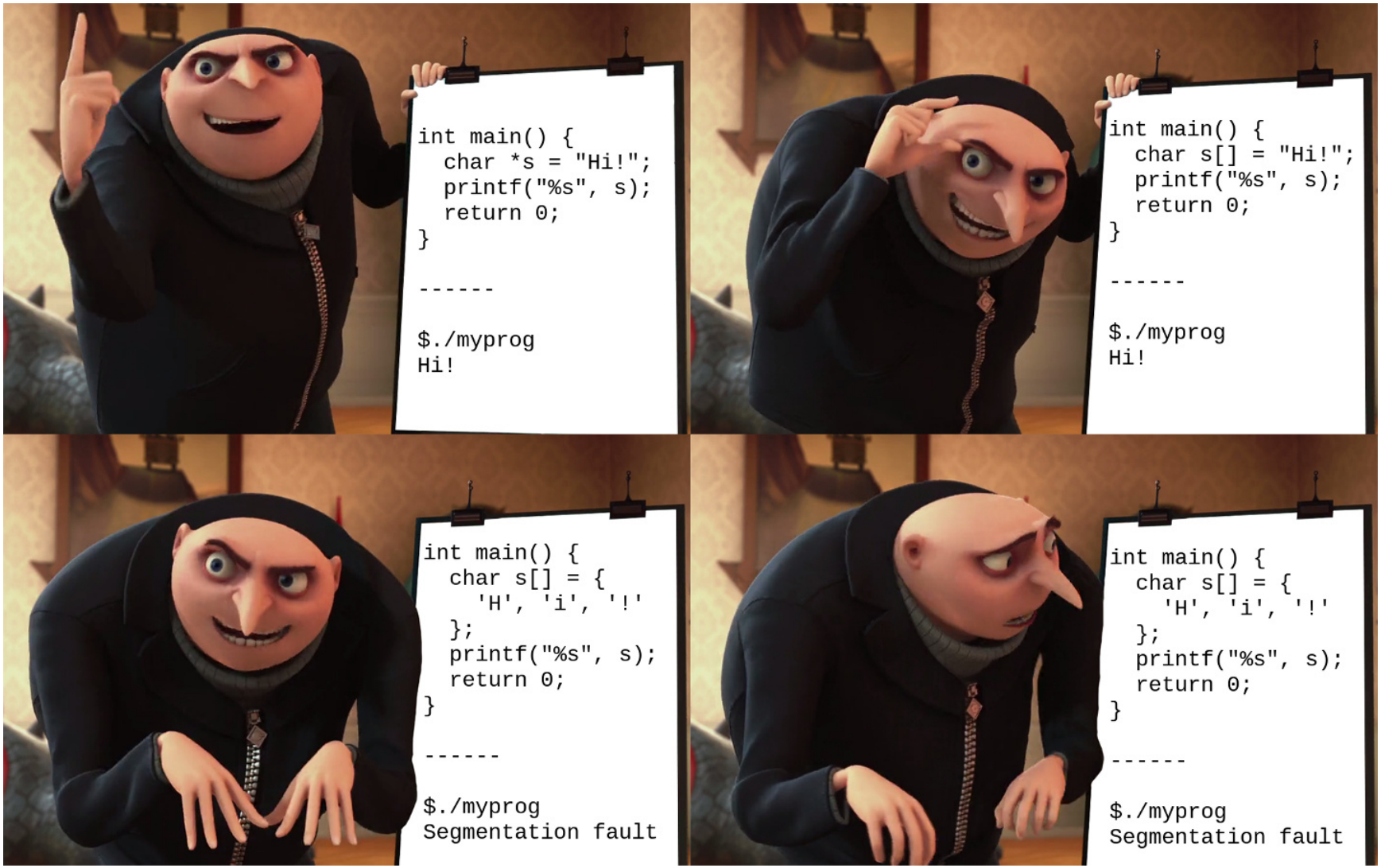


Figure 3: you hate to see it

# Strings Manipulation: Manipulation & Memory Safety

- String manipulation is a major cause of memory errors (e.g., *buffer overflow*)
- The C standard library includes both *safe* and *unsafe* string functions
  - **Some** unsafe functions can be used safely if the string is **guaranteed** to be NULL-terminated:

```
// unsafe, only safe if argv is guaranteed NULL-terminated
strlen(argv[0]);
```

  - But even so-called "safe" functions can cause memory errors if used improperly:

```
// safe, but vulnerable to improper use
char x[2]; strncpy(x, "blabla", 7);
```

# Strings Manipulation: Safety Manual

From `man gets` (Linux):

> *Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.*

Also from `man gets` (Mac):

> *The gets() function cannot be used securely. Because of its lack of bounds checking, and the inability for the calling program to reliably determine the length of the next incoming line, the use of this function enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. It is strongly suggested that the fgets() function be used in all cases.*

# Strings Manipulation: Unsafe Functions & Security

- *Question*: How can an attacker exploit a buffer overflow to break a system's security?
  - *Answer*: One way: `samsclass.info/127/proj/p3-lbuf1.htm`
- *Real example*: WhatsApp vulnerability from May 2019.
  - `arstechnica.com/information-technology/2019/05/whatsapp-vulnerability-exploited-to-infect-phones-with-israeli-spyware/`

## Lessons

- Only use C when necessary, and be mindful of safe programming practices.
- Otherwise, be responsible and use the right language for your task.

# Strings Manipulation: What's Safe? What isn't?

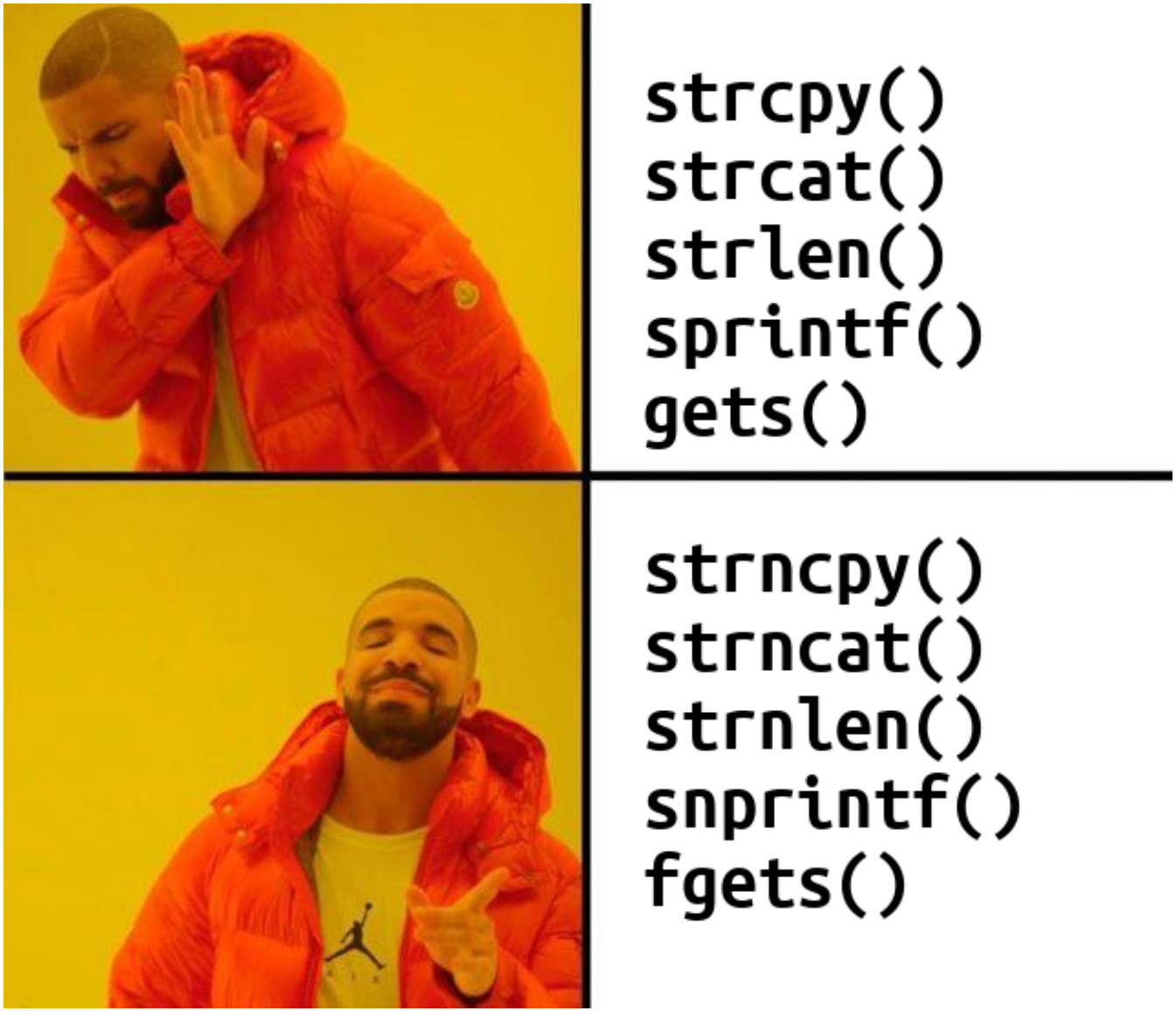


Figure 4: What Drake thinks about `strings.h`/`stdio.h`.

# Strings Manipulation: Safe(r) Library Functions

Just like with normal arrays, it's better practice to use functions that restrict or explicitly state the maximum number of characters.

```c
// from string.h
char *strncpy(char *dest, const char *src, size_t num);
char *strncat(char *dest, const char *src, size_t num);
size_t strnlen(const char *s, size_t num);
char *strndup( const char *str, size_t size ); // C23

// from stdio.h
int snprintf(char *restrict s, size_t n, const char *restrict fmt, ...);
char *fgets(char *restrict s, int n, FILE *restrict stream);
```

## Safety is Always Evolving

Newer versions and languages sometimes (re-)implement safer functions. For example, C11 Annex K introduces `strnlen_s`, whose distinction from `strnlen` is that it checks if `const char *s` is `NULL` and returns 0 if so; what would `strnlen` do?

# Strings Manipulation: Exercises in Safety (1)

```
strcpy(dest,src);
```

```
#define MAXS 100
char dest[MAXS];

strncpy(dest,
    src, MAXS);
```

```
#define MAXS 100
char dest[MAXS];

strncpy(dest,
    src, MAXS-1);
```

```
#define MAXS 100
char dest[MAXS];
dest[0] = '\0';
```

# Strings Manipulation: Exercises in Safety (1)

**Is this safe?**

```
char *dest;
strcpy(dest, src);
```

**What about these?**

```
#define MAXS 100
char dest[MAXS];

strncpy(dest, src, MAXS);

strncpy(dest, src, MAXS - 1);

dest[0] = '\0';
strncat(dest, src, MAXS - 1);
```

# Strings Manipulation: Exercises in Safety (1)

**Is this safe?** *No.*

```
char *dest;
strcpy(dest, src); // unsafe
```

**What about these?** *No, yes, and yes.*

```
#define MAXS 100
char *dest = malloc(sizeof(char) * MAXS); // same issues, stack or heap

strncpy(dest, src, MAXS); // unsafe: maybe no null-terminator

strncpy(dest, src, MAXS - 1); // safe

dest[0] = '\0'; // it gets overwritten
strncat(dest, src, MAXS - 1); // safe, same as above
```

# Strings Manipulation: Exercises in Safety (2)

**How about these?**

```
char str1[20] = "BeginnersBook";
printf("Length of string str1 %d\n", strnlen(str1, 30));
printf("Length of string str1 %d\n", strnlen(str1, 10));

char str2[6] = "csc209";
// some other code here.
int b;
scanf("%d", &b);
printf("Length of string str2 %d\n", strnlen(str2, b));

char buf[209] = {'\0'};
printf("Enter your name and press <Enter>\n");
gets(buf);
```

# Strings Manipulation: Exercises in Safety (2)

**How about these?** *Yes, no, and no.*

```c
char str1[20] = "BeginnersBook";
printf("Length of string str1 %d\n", strnlen(str1, 30)); // safe
printf("Length of string str1 %d\n", strnlen(str1, 10)); // safe

char str2[6] = "csc209";
// some other code here; might've removed the \0 from str2!
int b;
scanf("%d", &b); // unsafe; never trust external input
printf("Length of string str2 %d\n", strnlen(str2, b));

char buf[209] = {'\0'};
printf("Enter your name and press <Enter>\n");
gets(buf); // unsafe; accepts infinite stream
```

# Strings Manipulation: Example (`minigrep.c`)

Let's implement a miniature version of `grep` called `minigrep`. It'll simply search for lines matching `pattern` and print out match positions. More precisely, for each `line` in your `file`, our implementation will see if there exists some `size_t i` such that `strcmp(line+i, pattern)`. No flags, and exactly a `filename` and `pattern` are provided as arguments. Compile and run it on its own source code:

```
$ gcc -o minigrep minigrep.c
$ minigrep minigrep.c \#define
```

You'll find that the given code doesn't actually work! Replace `strcmp` with `strncmp` if the `line` is expected to be longer than `pattern`.

**Full code**

`github.com/rhubarbwu/csc209/blob/master/lectures/lec06/minigrep.c`

Section 4

# Debugging Tools: `gdb` and `valgrind`

# Debugging Tools for C

Thanks to the widespread use of C/C++, there are lots of tools to help you analyse, debug, and optimize your code.

## Static Analysis

Static code analyzers parse through source code and libraries to ensure that types, imports, and pointers make sense before you even execute the code.

- They include the compilers you're already familiar with (`gcc`/`g++`/`clang`/`clang++`).

## Dynamic Analysis

- Common tools for dynamic analysis (on variables/objects during execution) include debuggers like `gdb` and `lldb` (`lldb.llvm.org/`).
- Some integrated development environments (IDEs) like Visual Studio (Code) and JetBrains CLion and have their own or wrappers around existing debuggers.
- Another useful tool for analysing memory structures and integrity is `valgrind`.

# Why Use a Debugger?

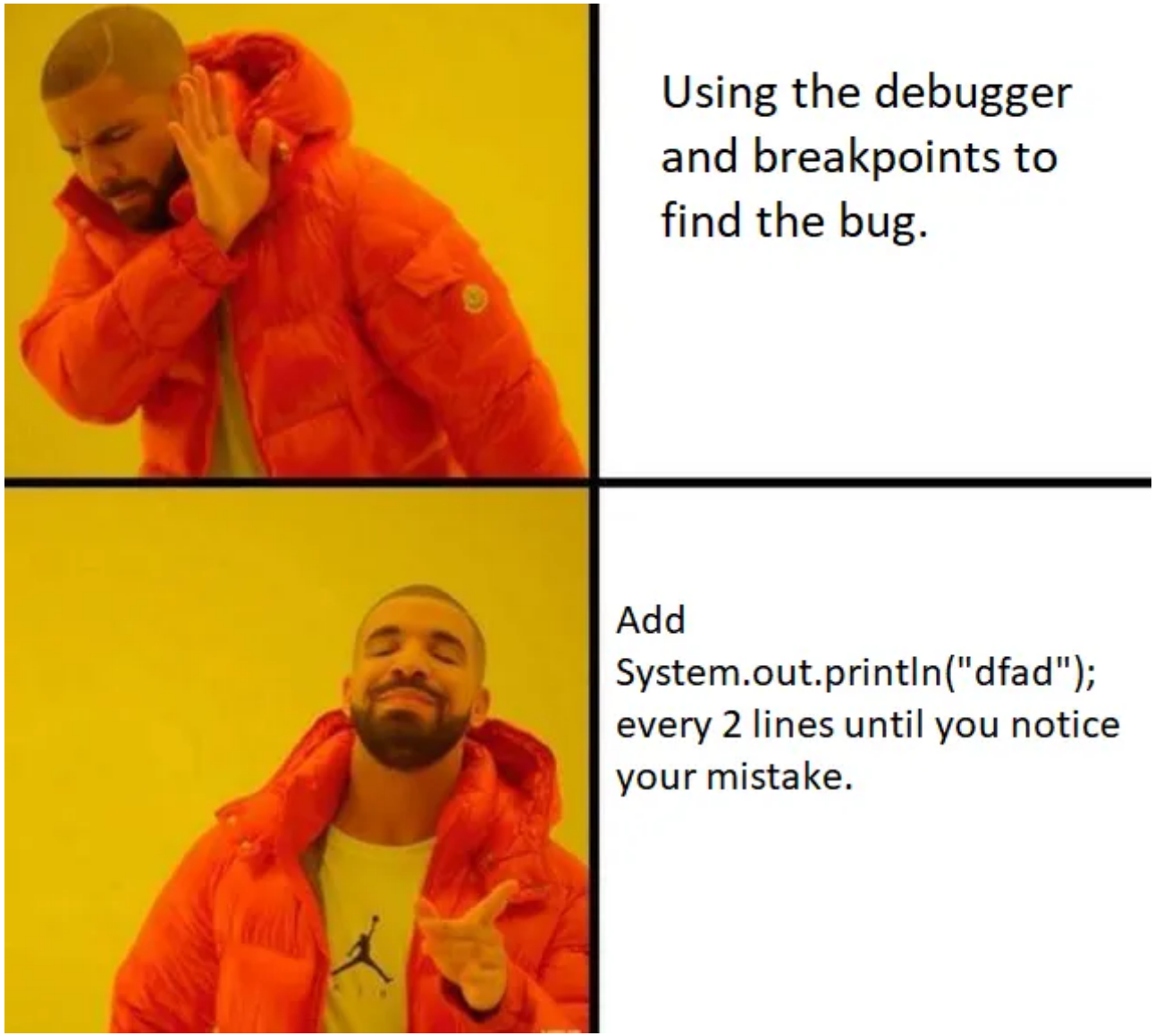


Figure 6: admit it, we all do this

# Why Use a Debugger?


Figure 7: debugging can be time-consuming

# Why Use a Debugger?

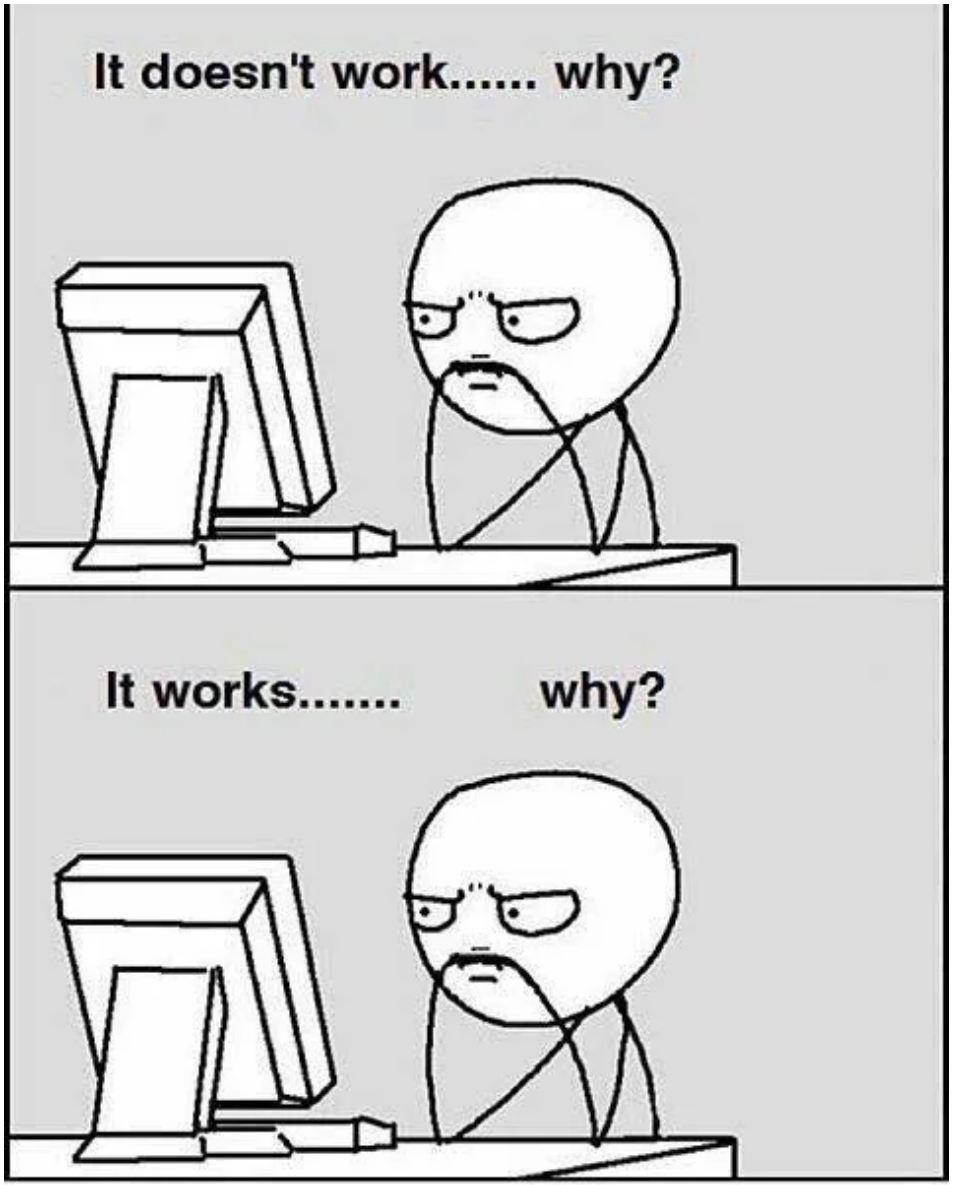


Figure 8: sometimes you actually want to understand

# GDB: The GNU Project Debugger

The GNU Project debugger (GDB) allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Programs might be executing on the same machine as GDB (native), on another machine (remote), or on a simulator. GDB can run on most popular UNIX and Windows variants, as well as on Mac OS X. In addition to C/C++, GDB supports:

- Assembly, Ada, D, Fortran, Go, Objective-C, OpenCL, Modula-2, Pascal, Rust

Source: `sourceware.org/gdb/`

# GDB: How to Get Started

Firstly, include the -g flag in gcc/clang. And then prepend gdb to the command.

```
$ gcc -g -o main main.c # compile with -g
$ gdb ./main # run a simple command
$ gdb --args ./main arg1 arg2 ... # run with arguments to main
```

## Breakpoints & Run

At certain points in the program you want to see the value of certain variables. You can use break (b) to set suchs breakpoints at non-blank/comment lines.

```
(gdb) break 5
Breakpoint 1 at 0x126e: file main, line 5.
(gdb) break 128
Breakpoint 2 at 0x1304: file main, line 129.
```

And finally, use run to start the debugger.

```
(gdb) run
Starting program: ./main arg1 arg2
```

# GDB: Stepping Through the Program

Once you start, you have several commands for stepping. Here are some basics.

| commands | what it does |
|---|---|
| backtrace (bt) | Print backtrace of all stack frames, or innermost frames. |
| break (b) | Set breakpoint at specified (line) location. |
| dump | Dump target code/data to a local file. |
| exit/quit (q) | Exit the debugger. |
| next (n) | Step program, proceeding through subroutine calls. |
| nexti (ni) | Step one instruction, but proceed through subroutine calls |
| print (p) | Print value of a variable or expression. |
| step (s) | Step program until it reaches a different source line. |
| stepi (si) | Step one (lower-level) instruction exactly. |

You can use `help` to see more instructions and `help all` for a list of commands.

# GDB: Additional Info

- Enabling gdb with -g makes the binary bigger. Feel free to leave it out if you're trying to produce the slimmest possible application that you feel has been debugged enough.
- gdb itself is a command-based application, but there exist front-ends and wrappers ranging in complexity from ddd and cgdb all the way to Visual Studio.

## Back to minigrep

Try debugging minigrep.c to fix it!

```
# gdb is enabled in some other compilers too!
$ clang -g -o minigrep minigrep.c
# debug in searching for while loops or macros
$ gdb --args minigrep minigrep.c while
$ gdb --args minigrep minigrep.c \#define
```

- What breakpoints might be interesting?
- Which variables/expressions?

/small

Full code:

github.com/rhubarbwu/csc209/blob/master/lectures/lec06/minigrep.c

# GDB: Additional Info

- Enabling gdb with -g makes the binary bigger. Feel free to leave it out if you're trying to produce the slimmest possible application that you feel has been debugged enough.
- gdb itself is a command-based application, but there exist front-ends and wrappers ranging in complexity from ddd and cgdb all the way to Visual Studio.

## Back to minigrep

Try debugging `minigrep.c` to fix it!

```
# gdb is enabled in some other compilers too!
$ clang -g -o minigrep minigrep.c
# debug in searching for while loops or macros
$ gdb --args minigrep minigrep.c while
$ gdb --args minigrep minigrep.c \#define
```

- What breakpoints might be interesting? `break 18`, `break 29`.
- Which variables/expressions? At 29: `line`, `line+i`, `pattern` and/or `result`.

Full code: `github.com/rhubarbwu/csc209/blob/master/lectures/lec06/minigrep.c`

# Valgrind: What is it?

Figure 9: Depiction of Saint George and the dragon.

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

Source: `valgrind.org/`

# Valgrind: What Kinds of Build Tools?

Figure 10: Relevance? Look up the green/red/purple "dragon book".

The Valgrind distribution currently includes seven production-quality tools:

- a memory error detector (Memcheck, the *default*)
- two thread error detectors
- a cache and branch-prediction profiler
- a call-graph generating cache and branch-prediction profiler
- two different heap profilers

It also includes an experimental SimPoint basic block vector generator.

Source: `valgrind.org/`

# Valgrind: In Action (Memory Permissions)

```c
#include <stdlib.h>
void f(void) {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;  // problem 1: heap block overrun
}
int main(void) {
    f();
    return 0; }
```

Running `valgrind ./example` might show you something like this.

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:4)
==19182==    by 0x80483AB: main (example.c:7)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:3)
==19182==    by 0x80483AB: main (example.c:7)
```

# Valgrind: In Action (Memory Permissions)

What does this stuff mean?

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:4)
==19182==    by 0x80483AB: main (example.c:7)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:3)
==19182==    by 0x80483AB: main (example.c:7)
```

- The process ID (PID) is 19182.
- The first line indicates an "Invalid write": you wrote out of bounds on the heap.
- Below the first line is the stack trace; read it bottom-up.
- The code addresses (eg. 0x804838F) are *usually* unimportant.
- Some error messages have a second component describing the memory address involved.
  - This one shows that the written memory is just past the end of a block allocated with malloc() on line 5 of example.c.

# Valgrind: In Action (Memory Leaks)

```c
#include <stdlib.h>
void f(void) {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;  // problem 1: heap block overrun
}               // problem 2: memory leak -- x not freed
```

You can include `--leak-check=[yes|full]` for detailed info on memory leaks.
Running `valgrind --leak-check=yes ./main` might show you something like this.

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:4)
==19182==    by 0x80483AB: main (example.c:7)
```

- "definitely lost": your program is leaking memory – fix it!
- "probably lost": your program is leaking memory, unless you have some way of backtracking on the heap.

Source: `valgrind.org/docs/manual/quick-start.html`

# Debugging: Correctness vs. Optimization

Because using -g to enable gdb (and others) adds debugging information to the application in the end, the binary will be larger and less efficient.

```
$ gcc -o example example.c
$ gcc -g -o example-g example.c
$ wc -l example example-g && du -h example example-g
```

- You can use `wc -l` or `du -h` to count number of lines or bytes a binary takes up.

Debugging is for the purpose of correctness, which is the most important thing.

- Once you're satisfied that your code is correct (as can be), remove -g to compile more compact and efficient code.

## Optimization: gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Conversely, the more optimization you include (-O1, -O2, -O3, etc.), the less helpful or reliable valgrind and other tools can become (especially for complex programs).

## Example: Leaky Linked-List

Full code: github.com/rhubarbwu/csc209/blob/master/lectures/lec06/leakyll.c

# Assignment 2

- Released on Saturday Feb 11, due Friday Feb 24.
- About implementing dictionaries.
- Using arrays, pointers, dynamic memory, strings manipulation.
- Graded on correctness, memory integrity and efficiency.
- Please use the debugging tools to help!