

DR ANTON GERDELAN
<GERDELA@SCSS.TCD.IE>

C PROGRAMMING

**LEAP IN AND TRY THINGS. IF YOU SUCCEED,
YOU CAN HAVE ENORMOUS INFLUENCE. IF
YOU FAIL, YOU HAVE STILL LEARNED
SOMETHING, AND YOUR NEXT ATTEMPT IS
SURE TO BE BETTER FOR IT.**

Brian Kernighan

C – THE LANGUAGE OF UNIX

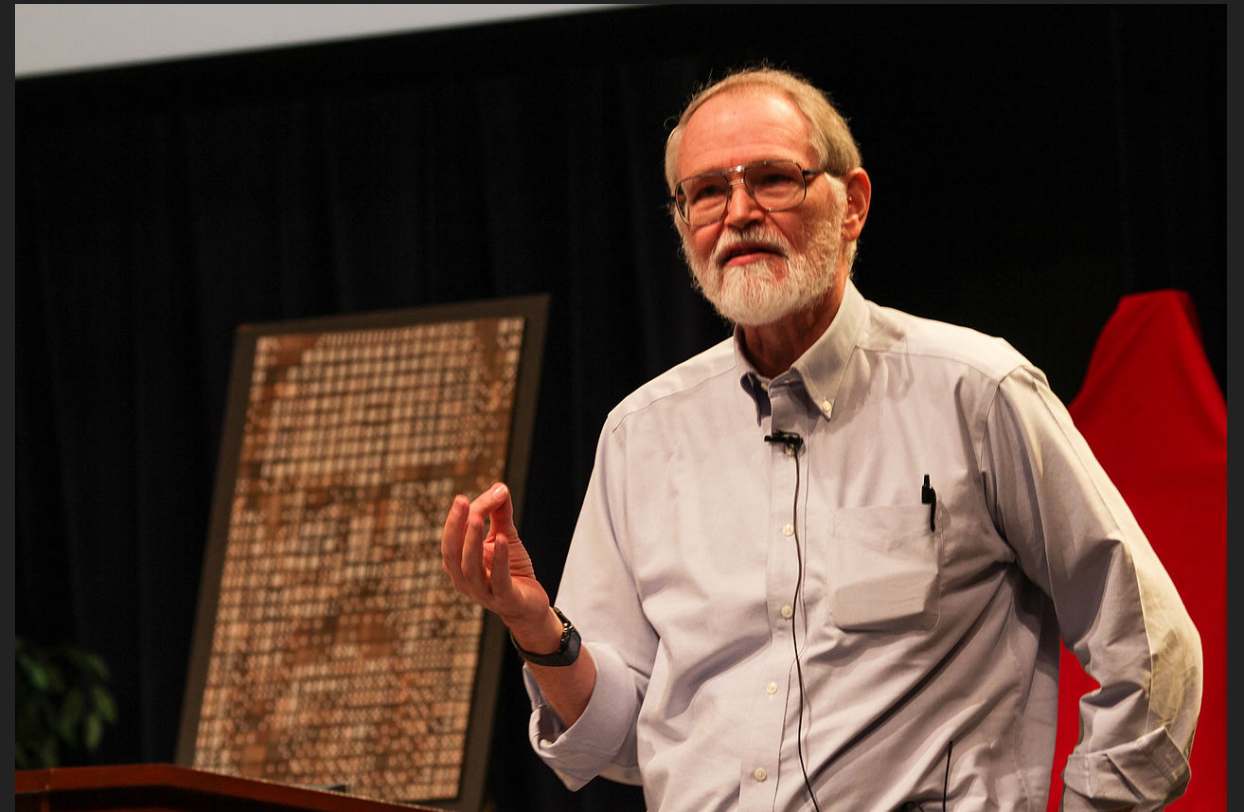
- ▶ Fundamental
- ▶ Easy to learn, simple, powerful
- ▶ DIY memory management
- ▶ Tools
- ▶ Features
- ▶ Plain old data types
- ▶ Pointers and addresses



Unix creators Ken Thompson (left, designed B - 1969, Go - 2007) and Dennis Ritchie (right, designed C - 1972) of Bell Labs. src: ?

APPLICATIONS OF C

- ▶ Performance
- ▶ Operating systems, drivers
- ▶ Games
- ▶ 3d graphics, imaging
- ▶ Desktop software
- ▶ File i/o, tools
- ▶ Most things



Prof. Brian Kernighan (Princeton U.) - co-author of "K&R C". We will come across his algorithms work later. src: Wikipedia.

PORTABILITY

- ▶ Almost all C also compiles as
 - ▶ C++, objective C
- ▶ Very similar to
 - ▶ C#, Java, D, PHP, JavaScript...
- ▶ Somewhat poss. to compile C++ into C

PRACTISE!

- ▶ How to get better
 - ▶ read functions' instructions in man pages
 - ▶ don't rely on auto-completion and Q&A websites
 - ▶ watch more exp. people coding
 - ▶ collaborate/share/code reviews/help - don't be shy

PRACTISE!

- ▶ Importance of playing around
- ▶ What don't you know?
 - ▶ make a list
- ▶ Ask good questions

POD TYPES

- ▶ "plain old data" types
- ▶ C89 had
 - ▶ unsigned and signed versions
 - ▶ long and short versions
 - ▶ people defined their own boolean type
- ▶ C99 and C++ have `bool`

`int`

`char`

`float`

`double`

`size_t`

`pointers`

`typedef int custom_name;`

A STRUCTURED DATA TYPE – ARRAYS

▶ pros

- ▶ multiple storage
- ▶ simple
- ▶ fast - looping over adjacent memory
- ▶ random access

▶ cons

- ▶ fixed size at compile time - waste space
 - ▶ elements must be same type
 - ▶ insertion requires shuffling
- ▶ often come with a count variable to say how much is used

```
int my_array[2048];
```

```
int sum = 0;
```

```
for (int i = 0; i < 2048; i++){  
    sum += my_array[i];  
}
```

```
printf("%i\n", sum);
```

```
my_array[238] = 10;
```

STRUCT

- ▶ combine variables into single custom variable
 - ▶ useful for passing and returning several variables from function
 - ▶ C++ has classes which are just structs with some extra properties
 - ▶ typedef usually used to shorthand your struct as a data type in C (not needed in C++)
- ```
typedef struct My_Combo_Type{
 int some_variable;
 char some_string[256];
} My_Combo_Type;

My_Combo_Type my_combo;

my_combo.some_variable = 10;
strcpy(my_combo.some_string,
"hello struct");
```
- \* there are various ways to initialise a struct

# ADDRESSES

- ▶ Unique location of each variable
- ▶ Pass by reference
- ▶ Ampersand
- ▶ Dynamic blocks of memory don't have an associated variable
- ▶ Refer to by their address
- ▶ A pointer can store an address

```
void some_func(int* a);

int my_variable = 0;
some_func(&my_variable);
```

# POINTERS, DEREFERENCING

- ▶ Stores memory address (just a number)
- ▶ Looks confusing in C
- ▶ Can point to a pointers address
- ▶ Dereference to get value stored at that address
- ▶ Pointers have a type, for convenience
- ▶ Dereferencing a pointer to a struct

remind me to do a diagram  
on the whiteboard here

and here

and here

# DYNAMIC MEMORY ALLOCATION

- ▶ We will use C malloc rather than C++ new keyword
- ▶ Know size of data in bytes
- ▶ `sizeof( )` function
- ▶ Number of elements
- ▶ Heap

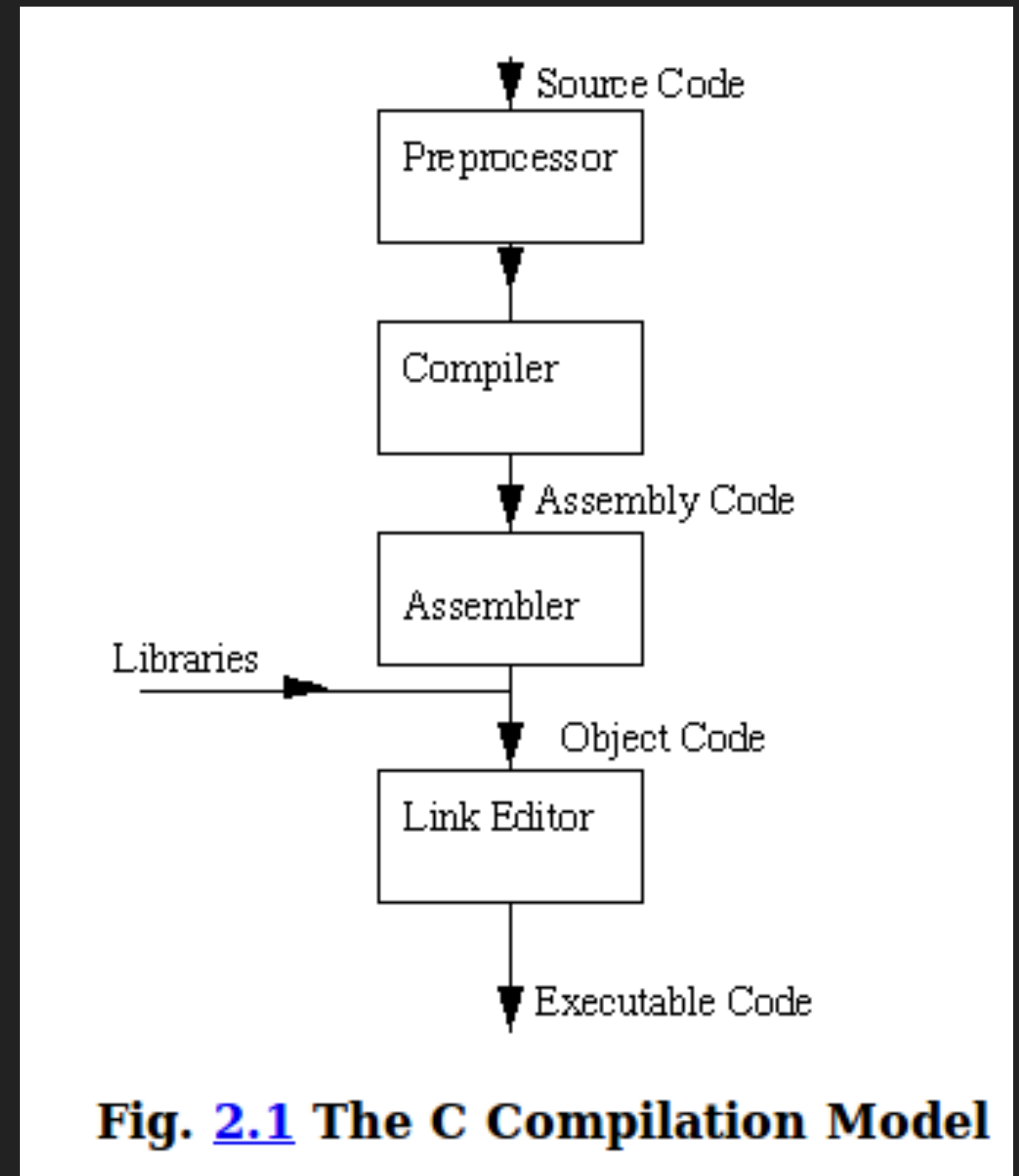
remind me to do some whiteboard here

# SIMPLIFY STRUCTURE

- ▶ Don't get carried away - only solve the problem at hand
- ▶ Time/cost/benefit
- ▶ Clarity, KISS
- ▶ Long functions are fine
- ▶ Blocks are good section separators { }
- ▶ Global variables are fine sparingly
- ▶ Less code, fewer files
- ▶ Quick and dirty is a great start - you can do a better one next time

# COMPILER SEQUENCE

- ▶ C is a human language
- ▶ Pre-processor
- ▶ Compiler
- ▶ Assembler
- ▶ Linker
- ▶ You can stop compiler at each stage and inspect the output
- ▶ Type of issues at each stage



src: <https://www.cs.cf.ac.uk/Dave/C/node3.html>

## ASM INSPECTION

- ▶ Matt Godbolt's gcc explorer  
<https://gcc.godbolt.org/>
- ▶ Insight into what your code compiles into
- ▶ How smart is your compiler?
- ▶ What does optimisation do?
- ▶ How does inline work?
- ▶ How efficient is C++ STL vs. our own data structures? Why?

Compiler Explorer - C++

Source: Examples ▾  
Name: max array ▾  
Load Save Save as... Full link Short link

x86-64 gcc 6.2 ▾ compiler options

Binary Unused labels Directives Comment-only lines Intel syntax Colourise

Code editor

```
1 // Type your code here, or load an example.
2 int square(int num) {
3 return num * num;
4 }
5
```

Assembly output

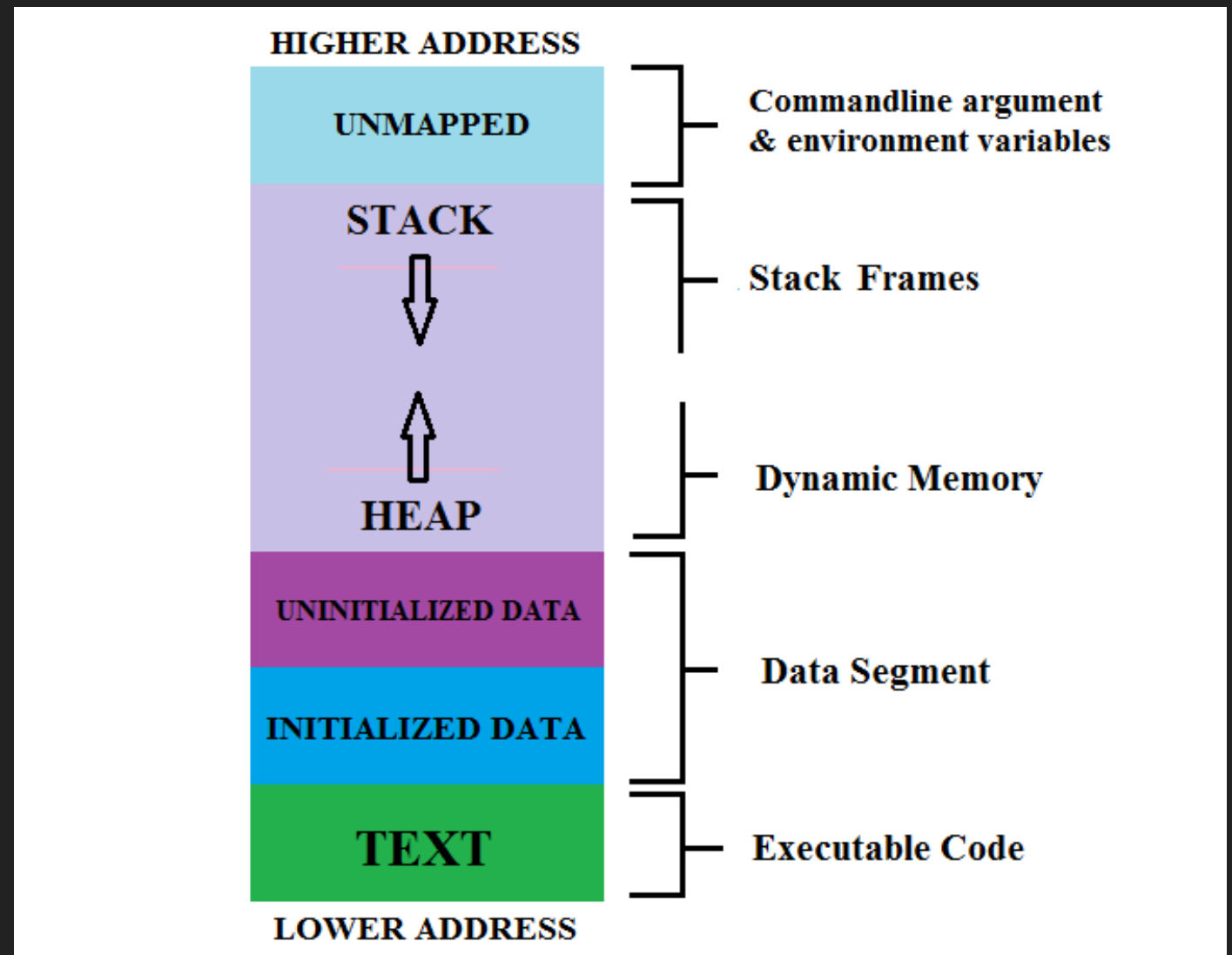
```
1 square(int):
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -4(%rbp)
5 movl -4(%rbp), %eax
6 imull -4(%rbp), %eax
7 popq %rbp
8 ret
9
```

Compiler output — x86-64 gcc 6.2 (g++ (GCC-Explorer-Build) 6.2.0)  
Compiled ok



# A PROGRAM'S MEMORY MODEL

- ▶ reserved system stuff
- ▶ Stack (frame per function)
- ▶ Heap (malloc)
- ▶ BSS (uninitialised statics)
- ▶ .data (initialised statics)
- ▶ Text (code)
- ▶ +dyn libraries loaded in-between stack and heap



src: <http://www.firmcodes.com/memory-layout-c-program-2/>

# THE [FUNCTION CALL] STACK

- ▶ each function launched is awarded a **frame** of memory for its local variables
- ▶ if that function calls another function inside it - **push** a new frame on the stack
- ▶ when a function ends - **pop** its frame from stack remind me to draw again
- ▶ most debuggers show you the call stack
  - ▶ on crash can get a "**backtrace**" or "stack trace"
- ▶ are huge call stacks of tiny functions bad? - recursive vs loop?
- ▶ <https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html>

# HEADERS AND >1 FILE

- ▶ Weakness of C - no packages
- ▶ `#include` (cut-pastes in)
- ▶ to share declarations between files (otherwise type them at top of every file)
- ▶ Header guards to avoid "*symbol already defined*" circular includes
- ▶ Using `extern` and `static` to share or hide between files
- ▶ Usually don't put code instructions in headers
  - ▶ exceptions

```
#include <assert.h>
```

```
int g_global_counter_thing;
```

```
int my_other_function(int* addr_of_thing) {
 assert(addr_of_thing);
 *addr_of_thing = *addr_of_thing + 1;
 g_global_counter_thing++;
 return *addr_of_thing;
}
```

---

```
// anton.h - header for anton.c
```

```
// written in C99 - Anton Gerdelan - date
```

```
#pragma once
```

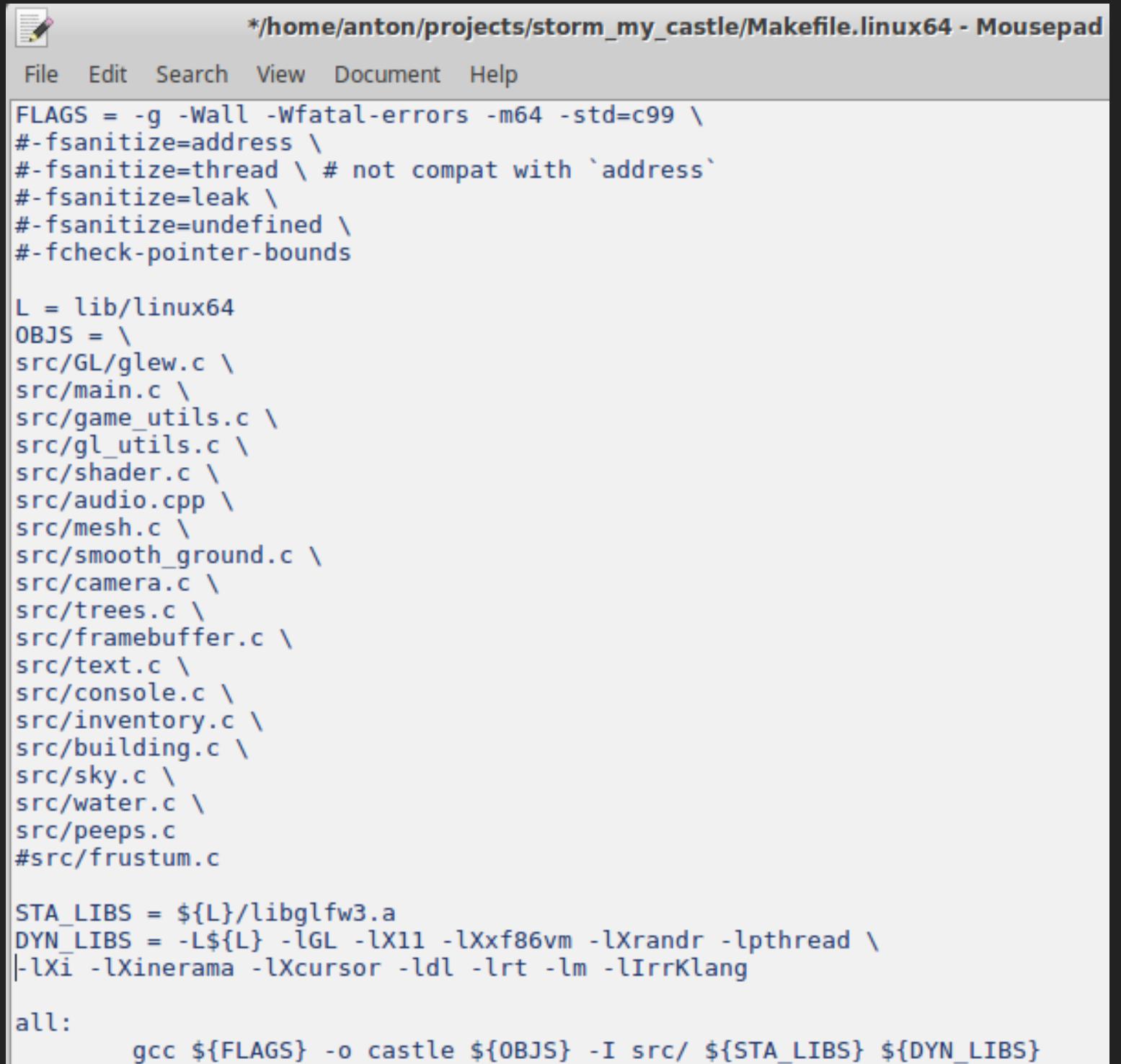
```
#include <stdbool.h>
```

```
// concise explanation
```

```
int my_other_function(int* addr_of_thing);
extern int g_global_counter_thing;
```

# MAKEFILES AND BUILD SYSTEMS

- ▶ Gross
- ▶ Worth learning Makefile
- ▶ IDEs have custom project files
- ▶ Meta-build systems exist
- ▶ Linking libraries is painful



```
*/home/anton/projects/storm_my_castle/Makefile.linux64 - Mousepad
File Edit Search View Document Help

FLAGS = -g -Wall -Wfatal-errors -m64 -std=c99 \
#-fsanitize=address \
#-fsanitize=thread \ # not compat with `address`
#-fsanitize=leak \
#-fsanitize=undefined \
#-fcheck-pointer-bounds

L = lib/linux64
OBSJ = \
src/GL/glew.c \
src/main.c \
src/game_utils.c \
src/gl_utils.c \
src/shader.c \
src/audio.cpp \
src/mesh.c \
src/smooth_ground.c \
src/camera.c \
src/trees.c \
src/framebuffer.c \
src/text.c \
src/console.c \
src/inventory.c \
src/building.c \
src/sky.c \
src/water.c \
src/peeps.c
#src/frustum.c

STA_LIBS = ${L}/libglfw3.a
DYN_LIBS = -L${L} -lGL -lX11 -lXxf86vm -lXrandr -lpthread \
-lXi -lXinerama -lXcursor -ldl -lrt -lm -lIrrKlang

all:

gcc ${FLAGS} -o castle ${OBSJ} -I src/ ${STA_LIBS} ${DYN_LIBS}
```

# LINKING

- ▶ Dynamic vs. static libraries
- ▶ Operating systems all have different formats
  - ▶ dynamic: `.so` `.dll` `.dylib`
  - ▶ static or stubs: `.a` `.lib`
- ▶ System libraries vs. local libraries
- ▶ We will try to avoid this topic
- ▶ Linux/Apple may need to link math library explicitly
  - ▶ only if you use functions from `math.h`
  - ▶ `clang -o my_prog main.c -lm`

# WHAT TO DO THIS WEEK

- ▶ Make sure that you can log in to lab computers
- ▶ Find an easy/working build env.
  - ▶ Visual Studio or another IDE?
  - ▶ Do you know how to step through code with a debugger?
  - ▶ GCC or Clang?
- ▶ Make sure that you can compile a few simple C examples
- ▶ Do the warm-up assignment
  - ▶ Let me know if it's too easy/too hard

# TUTORIALS

### ▶ Tutorial

- ▶ analysing some code, discussion, solving problems
- ▶ bring pen+paper (or laptop if you want)

### ▶ We can modify tutorials to suit needs by request

- ▶ e.g. what are your concept / tools knowledge gaps?

### ▶ Assignments

- ▶ 2-3 weeks each (2x 3hr lab sessions for help/grading)
- ▶ Know how to do everything - work individually, but not in isolation
- ▶ Starter code or example in a lecture or tutorial