Lecture 03
Build System + Pointers

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Fall 2021

Slides adapted from:
Jesse Tov

Northwestern

Administrivia

- Office hours are up and running
  - Be sure to use the office hours queue on Canvas
- Campuswire access (where you ask questions)
  - If you do not have access to campuswire, email me ASAP
- Gradescope access (where you submit code)
  - Be sure to make a Gradescope account ASAP
    - You should have gotten an email

Administrivia

- Homework 1 due on Thursday

- Self-eval for Homework 1 due on Sunday
  - Will be released after the homework is due
  - If you submit homework late, complete self-eval ASAP

  - Goal: make sure you're writing tests for your code
    - We'll ask you about whether you wrote certain tests or not

Gradescope demo

- Submitting code from terminal

- Seeing results in Gradescope
  - Be sure to either follow link or navigate to assignment again

- Can submit as many times as you want
  - We may later rate-limit your submissions
  - Later assignments WILL have hidden tests

  - Use the tests you fail on Gradescope to write your own tests!

Example Gradescope output

Unit test: overlapped_circles 1 (0.0/4.0)

```
#Test: Two circles at different locations that intersect.
#Input:
0 0 2
1 0 2
#Expected Output:
overlapped
#Received Output:
not overlapped

#[X] FAILED
```

- Failure is that Expected and Received Output did not match

- You can duplicate this test locally, which is easier to fix!
  - Create a new test that runs `overlapped_circles()` on `{0,0,2}` and `{1,0,2}`

Be sure to actually test your code locally

- Just running `make` compiles *and* runs tests

- I’ll recompile my code every few lines
  - That way there are never too many bugs to fix at once

- Then I make sure that I’m passing all the tests before uploading
  - And I add new tests whenever I see something weird I’m failing on Gradescope

Today's Goals

- Catch up on various C details
  - Compilation steps
  - Pre-processor
  - Make

- Begin introducing pointers in C
  - Why do they exist?
  - What are they useful for?
  - How do we use them?

Getting files for today's lecture

```
cd ~/cs211/lec/
```
(or wherever you put stuff)

```
tar -xkvf ~cs211/lec/03_pointers.tgz
cd 03_pointers/
```

Outline

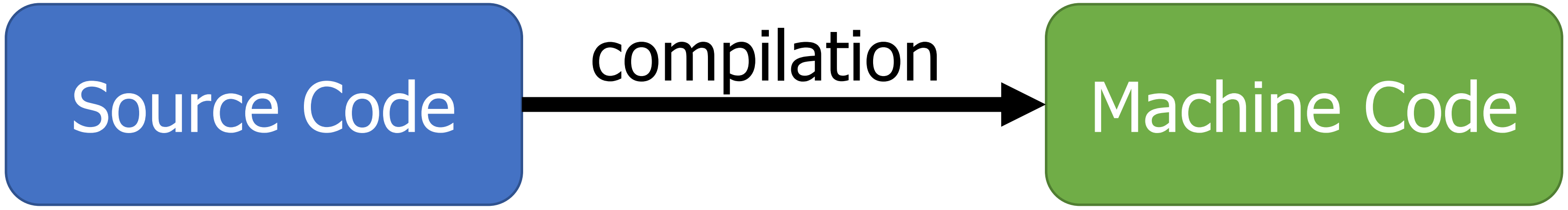
- **Separate Compilation**
- C Pre-Processor
- Makefiles

- What are pointers?
- Why are pointers?
- Variable lifetimes

Problems with compilation

- Two issues
  - Big programs take a very long time to compile
  - How can we reuse our functions in multiple programs?

- Let's focus on that second issue. It would be nice to:
  1. Write some functions in one file
  2. Call those functions from multiple programs (other files)



Solution: multiple C files

- You can write code in any number of different C files
  - And combine them together while compiling

- But we need some way to tell C code in one file about the existence of C code in another file
  - Solution: header files (.h)

  - Header files list all the publicly available functions and variables from a C file
    - Usually, there is a .c and .h file for various libraries

  - Header files are `#include`-ed at the top of your C file

Compiling multiple C files

- Each C file is compiled separately
- Then combine multiple together into a single program

- Compilers have a middle step: object files (.o)
  - Still not human readable
  - Meant to be joined together into a single executable

General C project layout

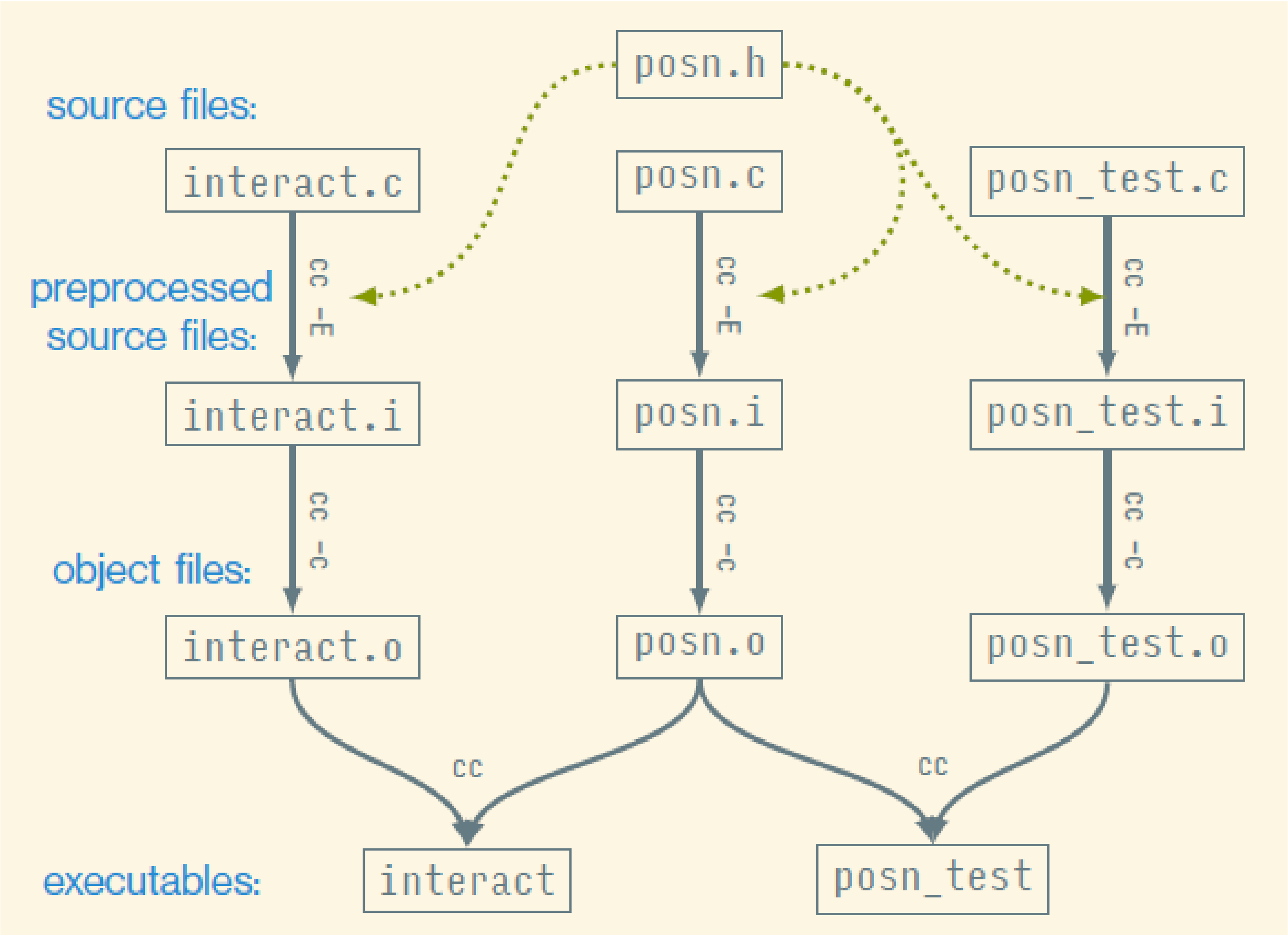
- src/
  - Various code that actually runs your project
- test/
  - Various code that tests your files in src/

- We separate code in src/ into two categories
  - The executable, which has a main() function and not much else
    - Named whatever your executable is, but with a .c
    - Example: overlapped.c

  - Libraries which have both .c and .h files
    - Example: circle.c and circle.h

Example of multiple compilation

example_project/



Outline

- Separate Compilation
- **C Pre-Processor**
- Makefiles

- What are pointers?
- Why are pointers?
- Variable lifetimes

C pre-processor

- Reads in the text of your source code

- Does some initial text-based manipulations to the code
  - Prepares everything for the compiler

C reads files from the top down

- First important thing to know about the pre-processor/compiler
  - They read from the top of the file down
  - Functions that don’t exist when you try to call them are an error

- How would we write this code then?

```
void a(void) {
  b();
}

void b(void) {
  a();
}
```

Function declaration

- You can inform the compiler about functions that will later be defined
  - You are telling the C compiler: “here’s what this other function looks like, you’ll get details about how it works later”
  - Very useful for libraries that you are using

- A function **declaration** in C includes the return type, name, and argument types
  - Examples:

```
void a(int, float);
struct circle read_circle(void);
```

- A function **definition** in C also includes the body of the function

Header files are collections of declarations

- You could manually type out the declaration for each function you want to use at the top of your C file

- Instead, we create “Header files” (.h) that hold all the function declarations for functions in the associated .c file

- `#include`-ing a header file tells the pre-processor to paste its contents
  - The same as if you had typed them in the top of the file yourself
  - Leads to weird errors sometimes if you mess up a header file
  - Be sure to only include header files!

What else can the pre-processor do?

- Macros
  - Text substitutions made by the pre-processor
- Compile-time code inclusion
  - Determine which code is actually compiled based on flags
- Pragma
  - Special commands to the compiler

C macros

```
#define NAME_OF_MACRO value_of_macro
```

- Examples:

  ```
  #define LENGTH 20
  #define FAIL_MESSAGE “There was an error!\n”
  ```

- The pre-processor pastes the text of the “value” wherever it finds the macro “name” in the source code
  - Useful for defining values that will be used in code
  - Again, be careful about weird bugs here!

Macro functions

- Macros can be used as functions as well

```
#define DEBUG(msg) printf(msg)

#define MIN(a, b) ((a < b) ? a : b)
```

- Generally, avoid this
  - You could just write a C function to do the operation instead
    - And the compiler will check this for errors better
  - It can be tricky to get right

Example of macro function trickiness

```
#define ADD(a, b) a+b

int x = ADD(3,4)*5; // Expects 7*5=35
```

- The pre-processor will expand this to:

  ```
  int x = 3+4*5; // Expects 7*5=35
  ```

- Extra parentheses around the macro value prevent this issue

  ```
  #define ADD(a, b) (a+b)
  ```

Ifdef in C

- The pre-processor evaluates the statement before compilation and either includes or removes the text
  - Useful because the code literally does not exist if removed

```
#ifdef DEBUG
  printf("Debug message here\n");
#endif
```

- Ifdef hell: when you can't figure out which C code is actually being compiled due to too many `#ifdef`s

Pragma examples

- Pragmas tell the C compiler to do something
  - Turn on/off warnings
  - Various compiler tricks that are important for low-level OS code

- Most common example: `#pragma` once at the top of each header
  - Tells the compiler to track this file and only paste it in a given C file once
  - Otherwise could end up with a bunch of different copies

  - Old C code uses `#ifdef` at the top of header files for the same task
    - Paired with an `#endif` at the very bottom of the file

Examples

preprocessor_example/

- The –E flag tells the compiler to only run the pre-processor
- In homework01
  - cc –E src/overlapped.c –o overlapped.i
    - Note that header files are included
    - Note that some functions are only definitions right now
- Example of macro substitution

Break + Administrivia

- Coming soon in class
  - Lecture
    - Pointers (today)
    - Arrays and Strings (Thursday)

  - Lab
    - String manipulation (prep for Homework 2)

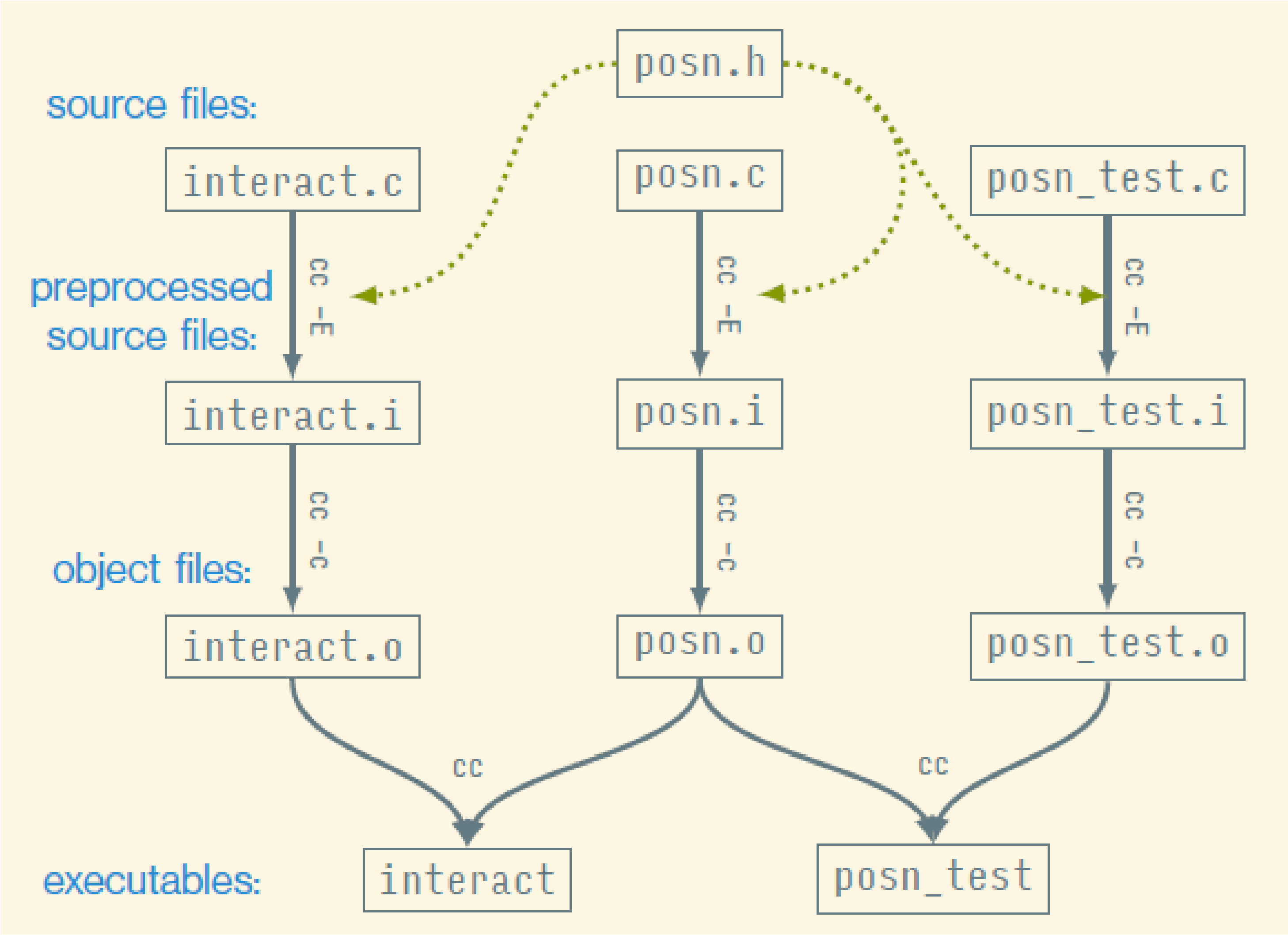
  - Homework
    - Program for text replacement in strings
    - Swaps all instances of a character with another

Outline

- Separate Compilation
- C Pre-Processor
- **Makefiles**

- What are pointers?
- Why are pointers?
- Variable lifetimes

New problem, how do you remember all these steps?



And this doesn't even include various flags we give to the compiler, such as the location of the 211.h library

Simplifying multiple compilation with Make

- Make is a tool for building programs out of multiple source files
  - Allows you to specify goals and requirements as “rules”
  - And then runs the compiler to fulfill those

- To build a file named ⟨`goal`⟩ using make, you run:

  `make ⟨goal⟩`

- `Make` looks around the current directory for a file named Makefile which specifies the various rules
  - We’ll provide the Makefile for you in this class
  - But you’ll have to use `make` to compile your programs

What does a `make` rule look like?

- A rule has a goal and pre-requisites for the goal
  - And then specifies commands to create the goal given the pre-requisites

```
⟨goal⟩: ⟨prereqs⟩. . .
    ⟨commands⟩
    ...
```

- Example:

```
hello: hello.c
    cc -o hello hello.c
```

Bonus: Makefile for building interact and posn_test

- Take a look at these if you want to understand the Makefile for the interact and posn_test programs from today's lecture files
  - `~cs211/lec/03_pointers`

Bonus: Makefile for building interact and posn_test

- These rules encode the dependency diagram from a few slides back (but with preprocessing and translation combined)

```
interact: interact.o posn.o
   cc -o interact interact.o posn.o

posn_test: posn_test.o posn.o
   cc -o posn_test posn_test.o posn.o

interact.o: interact.c posn.h
   cc -c -o interact.o interact.c

posn_test.o: posn_test.c posn.h
   cc -c -o posn_test.o posn_test.c

posn.o: posn.c posn.h
   cc -c -o posn.o posn.c
```

Bonus: Makefile for building interact and posn_test

- Good programmers are lazy and hate repetition. So much repetition here!

```
interact: interact.o posn.o
	cc -o interact interact.o posn.o

posn_test: posn_test.o posn.o
	cc -o posn_test posn_test.o posn.o

interact.o: interact.c posn.h
	cc -c -o interact.o interact.c

posn_test.o: posn_test.c posn.h
	cc -c -o posn_test.o posn_test.c

posn.o: posn.c posn.h
	cc -c -o posn.o posn.c
```

Bonus: Makefile for building interact and posn_test

- You don't have to repeat the goal in each recipe
  - It's better to use the special variable $@ instead

```
interact: interact.o posn.o
	cc -o $@ interact.o posn.o

posn_test: posn_test.o posn.o
	cc -o $@ posn_test.o posn.o

interact.o: interact.c posn.h
	cc -c -o $@ interact.c

posn_test.o: posn_test.c posn.h
	cc -c -o $@ posn_test.c

posn.o: posn.c posn.h
	cc -c -o $@ posn.c
```

Bonus: Makefile for building interact and posn_test

- Similarly, $^ is a variable that stands for the prerequisites
  - Or $< when you only want the *first* prerequisite

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

interact.o: interact.c posn.h
	cc -c -o $@ $<

posn_test.o: posn_test.c posn.h
	cc -c -o $@ $<

posn.o: posn.c posn.h
	cc -c -o $@ $<
```

Bonus: Makefile for building interact and posn_test

- Now note that the bottom three compilation rules are the same except for the filename. We can replace them with a pattern rule

```
interact: interact.o posn.o
    cc -o $@ $^

posn_test: posn_test.o posn.o
    cc -o $@ $^

interact.o: interact.c posn.h
    cc -c -o $@ $<

posn_test.o: posn_test.c posn.h
    cc -c -o $@ $<

posn.o: posn.c posn.h
    cc -c -o $@ $<
```

Bonus: Makefile for building interact and posn_test

- This pattern says we can build any .o file from a matching .c file

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

%.o: %.c posn.h
	cc -c -o $@ $<
```

Bonus: Makefile for building interact and posn_test

- That pattern is pretty generic except for the reliance on posn.h
  - Let’s break that out into a separate rule

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

%.o: %.c
	cc -c -o $@ $<

interact.o posn_test.o posn.o: posn.h
```

Bonus: Makefile for building interact and posn_test

- And we really ought to make the compiler used a variable
  - Then others could change it out if desired

```
interact: interact.o posn.o
	$(CC) -o $@ $^

posn_test: posn_test.o posn.o
	$(CC) -o $@ $^

%.o: %.c
	$(CC) -c -o $@ $<

interact.o posn_test.o posn.o: posn.h
```

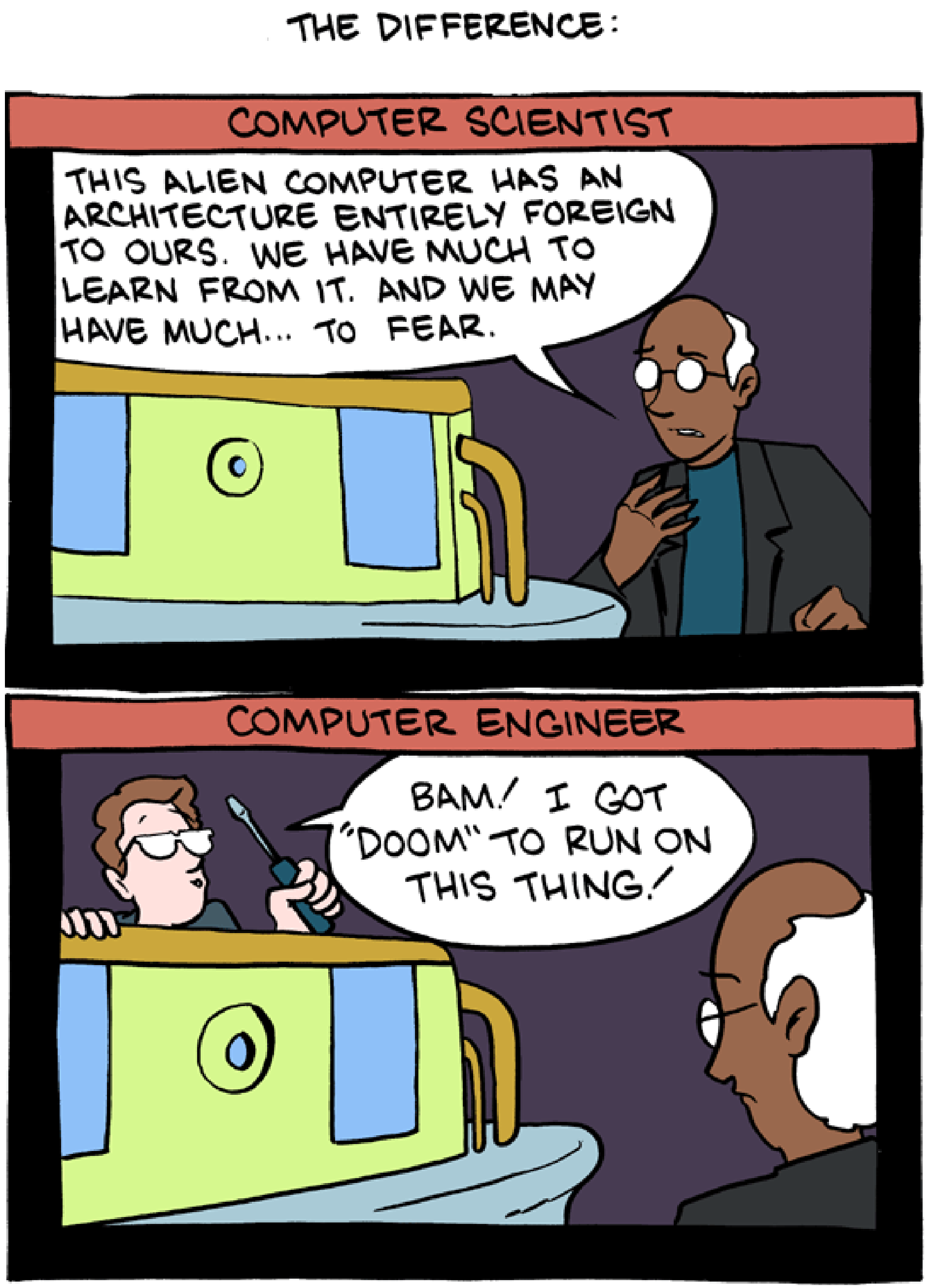
Bonus: Makefile for building interact and posn_test

- Finally, there are often compiler options we want to pass in
  - Here are the standard variables for holding those

```
interact: interact.o posn.o
	$(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

posn_test: posn_test.o posn.o
	$(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

%.o: %.c
	$(CC) -c -o $@ $< $(CPPFLAGS) $(CFLAGS)

interact.o posn_test.o posn.o: posn.h
```

Break + SMBC Comic

- Saturday Morning Breakfast Cereal



https://www.smbc-comics.com/comic/2011-02-17

Outline

- Separate Compilation
- C Pre-Processor
- Makefiles

- **What are pointers?**
- Why are pointers?
- Variable lifetimes

Remember: values, objects, and variables

- **Values** are the actual information we want to work with
  - Numbers, Strings, Images, etc.
  - Example: 3 is an `int` value

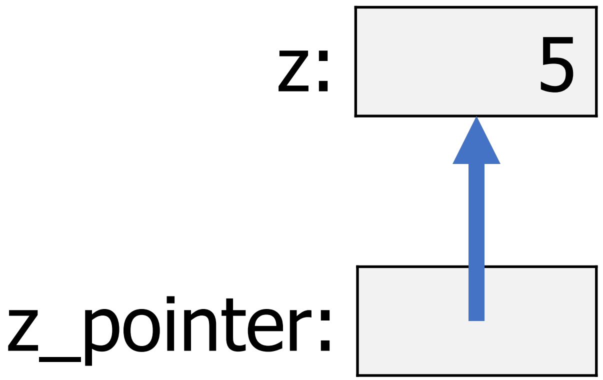
- An **object** is a chunk of memory that can hold a value of a particular type.
  - Example: function `f` has a parameter `int x`
    - Each type `f` is called, a “fresh” object that can hold an int is “created”

- A **variable** is the name of an object

- Assigning to a variable changes the *value* stored in the object named by the variable

Pointers are another type of value

- Values could be a number, like 5 or 6.27
- Or they could be a "pointer" to an **object**
  - Points at the object, not the variable or value
  - It points at the "chunk of memory"
    - Technically, in C it holds the address of that memory



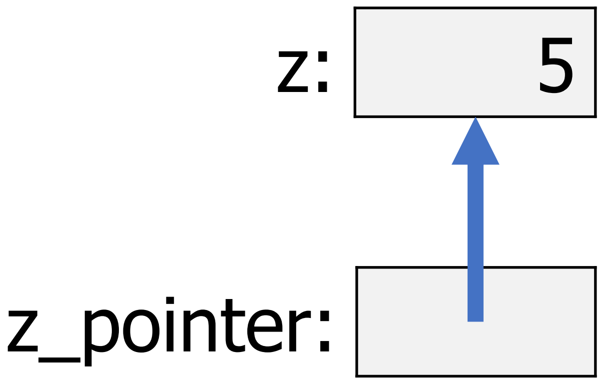
C syntax for pointers

- Pointers are a family of types
  - Each pointer is an existing C type, followed by a *

- To get the pointer to an existing variable, use the & operator
  - Returns the address of that variable

- Example:

```
int z = 5;
int* z_pointer = &z;
```



Longer pointer example

pointers_examples/longer_pointers.c

```
1. double alpha;
```

alpha: ???

What is the initial value of `alpha`?

Longer pointer example

pointers_examples/longer_pointers.c

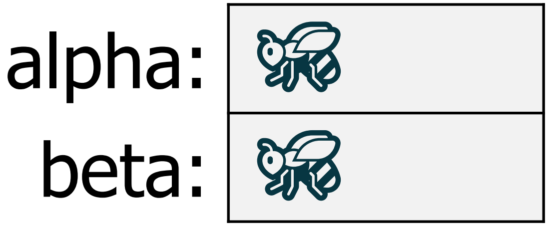
```
1. double alpha;
```

alpha: 

Longer pointer example

pointers_examples/longer_pointers.c

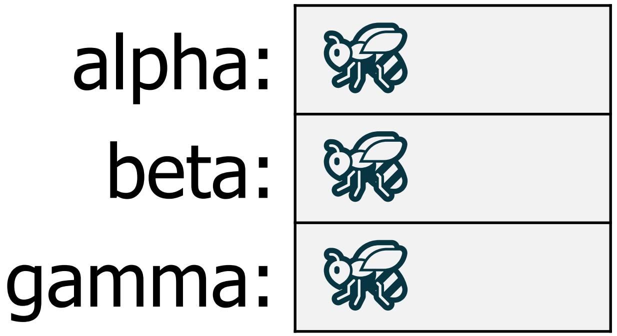
```
1. double alpha;
2. double* beta;
```



Longer pointer example

pointers_examples/longer_pointers.c

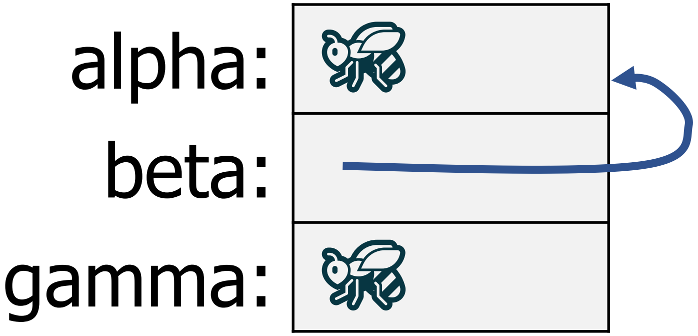
```
1. double alpha;
2. double* beta;
3. double* gamma;
```



Longer pointer example

pointers_examples/longer_pointers.c

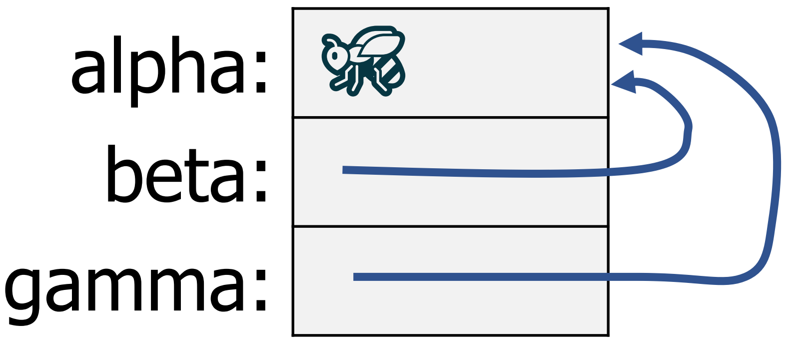
```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
```



Longer pointer example

pointers_examples/longer_pointers.c

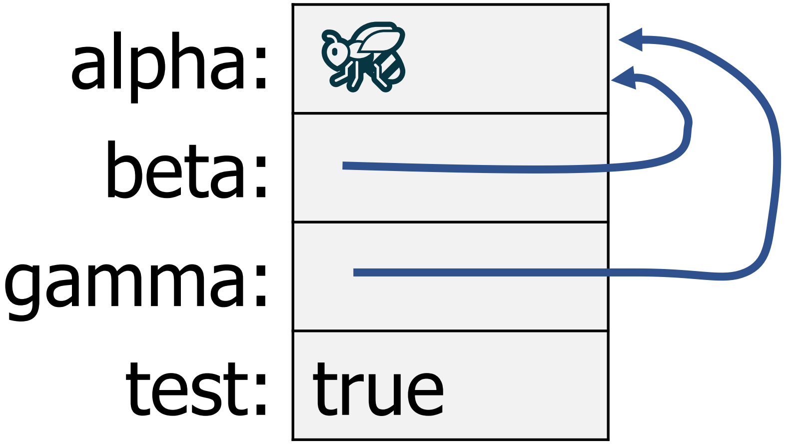
```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
```



Longer pointer example

pointers_examples/longer_pointers.c

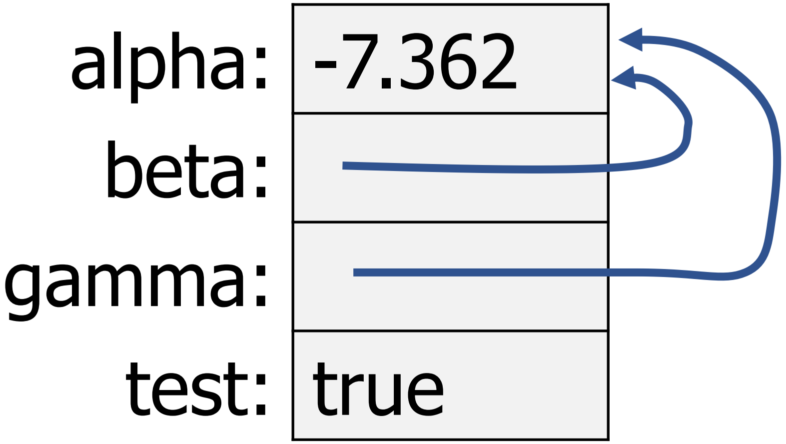
```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
6. bool test = (beta == gamma && beta == &alpha);
```



Longer pointer example

pointers_examples/longer_pointers.c

```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
6. bool test = (beta == gamma && beta == &alpha);
7. alpha = -7.362;
```



Dereferencing a pointer

- Pointers can be used to read or modify the value in the object pointed at

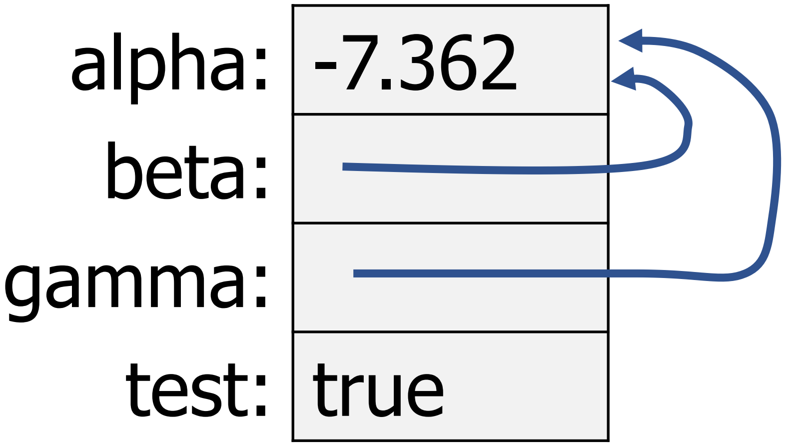
- The * operator is used for getting/setting the value in the object
  - This is called "dereferencing" the pointer
  - Not multiply in this context

- Examples:

```
printf("%d\n", *my_int_pointer);

*my_int_pointer = 15;
```

Longer pointer example

pointers_examples/longer_pointers.c

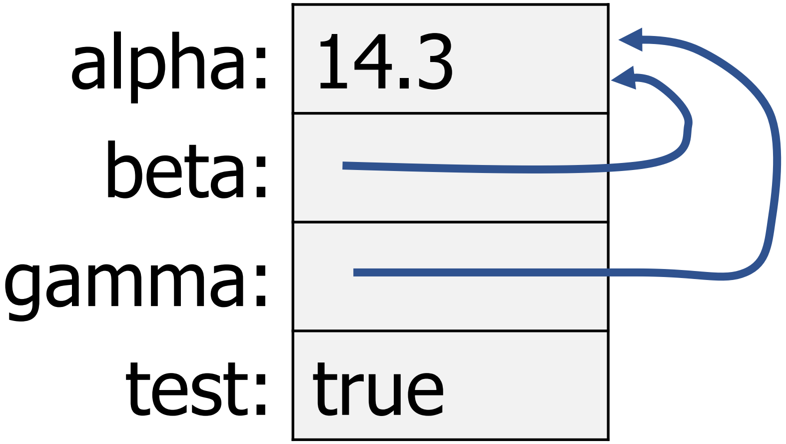



```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
6. bool test = (beta == gamma && beta == &alpha);
7. alpha = -7.362;
8. test = (*beta < 0); // still true!
```

Longer pointer example

pointers_examples/longer_pointers.c

```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
6. bool test = (beta == gamma && beta == &alpha);
7. alpha = -7.362;
8. test = (*beta < 0);
9. *gamma = 14.3
```



Possible pointer values

- Uninitialized

```
unsigned long* zeta;
```

- Pointing at an existing object

```
char* letter_ptr = &my_char;
```

- Null (explicitly pointing at nothing)

```
int* p = NULL;
bool* b = NULL;
double* d = NULL;
```

  - NULL works for any pointer type
  - NULL is NOT the same as uninitialized (🐝)
  - Dereferencing a null pointer is an error (segfault)

Some things to remember about pointers

1. Remember that a pointer is a type
   - int*, char*, short*, bool*, double*, size_t*, etc.

2. Think carefully about whether the pointer is being modified or the value in the object it points to
   - my_pointer = &x; // modifies which object we are pointing at
   - *my_pointer = x; // modifies the value in the object we are pointing at

3. Remember that pointer variables are themselves variables
   - They have values: the address of the object being pointed at
   - They name objects: memory is allocated to hold the address

C things that make pointers annoying

- For pointer types, the * doesn't have to be next to the type
  - These three all mean exactly the same thing:

```
1. int*        x; // I strongly recommend you use this

2. int    *    x;

3. int        *x;
```

C things that make pointers annoying

- For pointer types, the * doesn't have to be next to the type
  - These three all mean exactly the same thing:

```
1. int*         x; // I strongly recommend you use this

2. int    *     x;

3. int         *x;
```

- The * operator also means multiplication

```
signed long w = *t * *v; // multiply values referenced
                         // by the pointers t and v
```

Never define multiple variables at once

- You can define multiple variables at once in C

```
double x, y, radius;
```

Equivalent code:

```
double x;
double y;
double radius;
```

Never define multiple variables at once

- But this breaks when you're using pointers

```
double* x, y, radius;
```

Equivalent code:

```
double* x;
double y;
double radius;
```

} Not pointers!!! 😱 (applies to `double y;` and `double radius;`)

- To write that line correctly, you need to write:

`double *x, *y, *radius;` OR `double * x, * y, * radius;` (spacing doesn't matter)

- Or just never ever declare multiple variables in the same line!
  - That's the CS211 style rule

Break + Question

```
int a = 15;
int* b = &a;
int* c = b;
*c = 7;
```

What are the values of:

```
a   =
*b  =
c   =
```

Break + Question

```
int a = 15;
int* b = &a;
int* c = b;
*c = 7;
```

What are the values of:

```
a    = 7      // set by *c=7
*b   =
c    =
```

Break + Question

```
int a = 15;
int* b = &a;
int* c = b;
*c = 7;
```

What are the values of:

```
a    = 7      // set by *c=7
*b   = 7      // points to value of a
c    =
```

Break + Question

```
int a = 15;
int* b = &a;
int* c = b;
*c = 7;
```

What are the values of:

```
a    = 7      // set by *c=7
*b   = 7      // points to value of a
c    = &a     // holds the address of a
```

Outline

- Separate Compilation
- C Pre-Processor
- Makefiles

- What are pointers?
- **Why are pointers?**
- Variable lifetimes

Pointers functions directly modify values inside variables

- Normally, functions get a copy of the value inside the variable

- With pointers, functions can directly modify the variable
    - The function gets a copy of the pointer to the variable

Adding two to a variable WITHOUT pointers

pointers_examples/
add_without_pointers.c

```
int add_two(int n) {
  return n+2;
}

int main(void) {
  int x = 15;
  x = add_two(x);
  printf("%d\n", x);
  return 0;
}
```

Adding two to a variable WITH pointers

pointers_examples/
add_with_pointers.c

```
void add_two(int* n) {
  *n += 2;
}

int main(void) {
  int x = 15;
  add_two(&x);
  printf("%d\n", x);
  return 0;
}
```

Side-by-side comparison of without/with pointers

```c
int add_two(int n) {
  return n+2;
}

int main(void) {
  int x = 15;
  x = add_two(x);
  printf("%d\n", x);
  return 0;
}
```

```c
void add_two(int* n) {
  *n += 2;
}

int main(void) {
  int x = 15;
  add_two(&x);
  printf("%d\n", x);
  return 0;
}
```

Another example: what if we want to pass a struct

pointers_examples/
struct_with_pointers.c

```
typedef struct plants {
  bool is_watered;
  double height;
  int num_leaves;
} plant_t;
```

```
void initialize_oak_tree(plant_t* plant){
  (*plant).is_watered = true;
  (*plant).height = 10;
  (*plant).num_leaves = 100000;
}

int main(void){
  plant_t plant_a;
  initialize_oak_tree(&plant_a);
  return 0;
}
```

Shortcut for pointers to structs

- C programs end up using pointers to structs A LOT

- It's annoying to type (*struct).field all the time
  - So we made a shortcut. These two mean exactly the same thing:

```
(*struct).field

struct->field
```

(that's dash and greater than)

  - This is known as "syntactic sugar"
    - Bonus syntax to make common things easier

Adding a function to print the struct

pointers_examples/
struct_with_pointers.c

```
typedef struct plants {
  bool is_watered;
  double height;
  int num_leaves;
} plant_t;
```

```
void initialize_oak_tree(plant_t* plant){
  (*plant).is_watered = true;
  (*plant).height = 10;
  (*plant).num_leaves = 100000;
}

void print_plant(plant_t* plant){
  printf("Plant is %d meters tall and "
         "has %d leaves.\n",
         plant->height, plant->num_leaves);

  if (!plant->watered) {
    printf("\tIt needs to be watered!\n");
  }
}
```

Scanf example

- `scanf()` uses pointers to write to the variables you pass it

```
int x = 0;
int count = scanf("%d", &x);
```

- Pointers allow `scanf()` to read results directly into your variable
- Pointers also `scanf()` to simultaneously return the number of arguments matched

Outline

- Separate Compilation
- C Pre-Processor
- Makefiles

- What are pointers?
- Why are pointers?
- **Variable lifetimes**

When is a pointer "valid"?

1. If it is initialized

2. If the variable it is referencing still has a valid lifetime
   - Variables "live" until the end of the scope they were created in
   - Scopes are defined by { }

   - Example:

```
void some_function(void) {
    int a = 5;
}
```

← `a` goes "out of scope" here
The variable stops being "alive"

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```c
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

~~a~~:

- Variable `a` is no longer "alive" at this point
  - It "poofs" out of existence
  - The variable is no longer valid

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

n: 17

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n:	17
a:	5

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n:	17
a:	5

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

n:	17
a:	5
b:	16

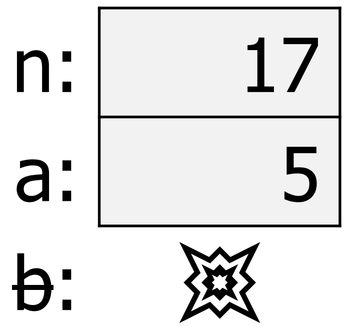
Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

n:	17
a:	5
b:	16

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```



Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

n:	17
a:	5

Referring to variable `b` at this point would be a compilation error

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

~~n~~:

~~a~~:

Variable lifetimes are what makes loops work

- Variables created inside of loops only exist until the end of that iteration of the loop
  - i.e. they only exist until the next end curly brace }

```
while (n < 5) {
    int i = 1;
    n += i;
}
```

A new variable `i` is created each time the loop repeats

Dangling pointers reference invalid objects

pointers_examples/
dangling_pointer.c

```
int* get_pointer_to_value(void) {
  int n = 5;
  return &n;
}

int main(void) {
  int* x = get_pointer_to_value();
  printf("%d\n", *x);
  return 0;
}
```

Dangling pointers reference invalid objects

pointers_examples/
dangling_pointer.c

```
int* get_pointer_to_value(void) {
  int n = 5;
  return &n;
}
```

n goes out of scope at the end of this function

So what does the pointer point to???

```
int main(void) {
  int* x = get_pointer_to_value();
  printf("%d\n", *x);
  return 0;
}
```

Dangling pointers are especially dangerous

- Accessing a dangling pointer is *undefined behavior*
  - Anything could happen!
- If you are lucky: segmentation fault (a.k.a. SIGSEGV)
  - The OS kills your program because it accesses invalid memory
- If you are unlucky: *anything at all*
  - Including returning the correct result the first time you run it and an incorrect result the second time

Outline

- Separate Compilation
- C Pre-Processor
- Makefiles

- What are pointers?
- Why are pointers?
- Variable lifetimes