Lecture 02
Introducing C

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Winter 2022

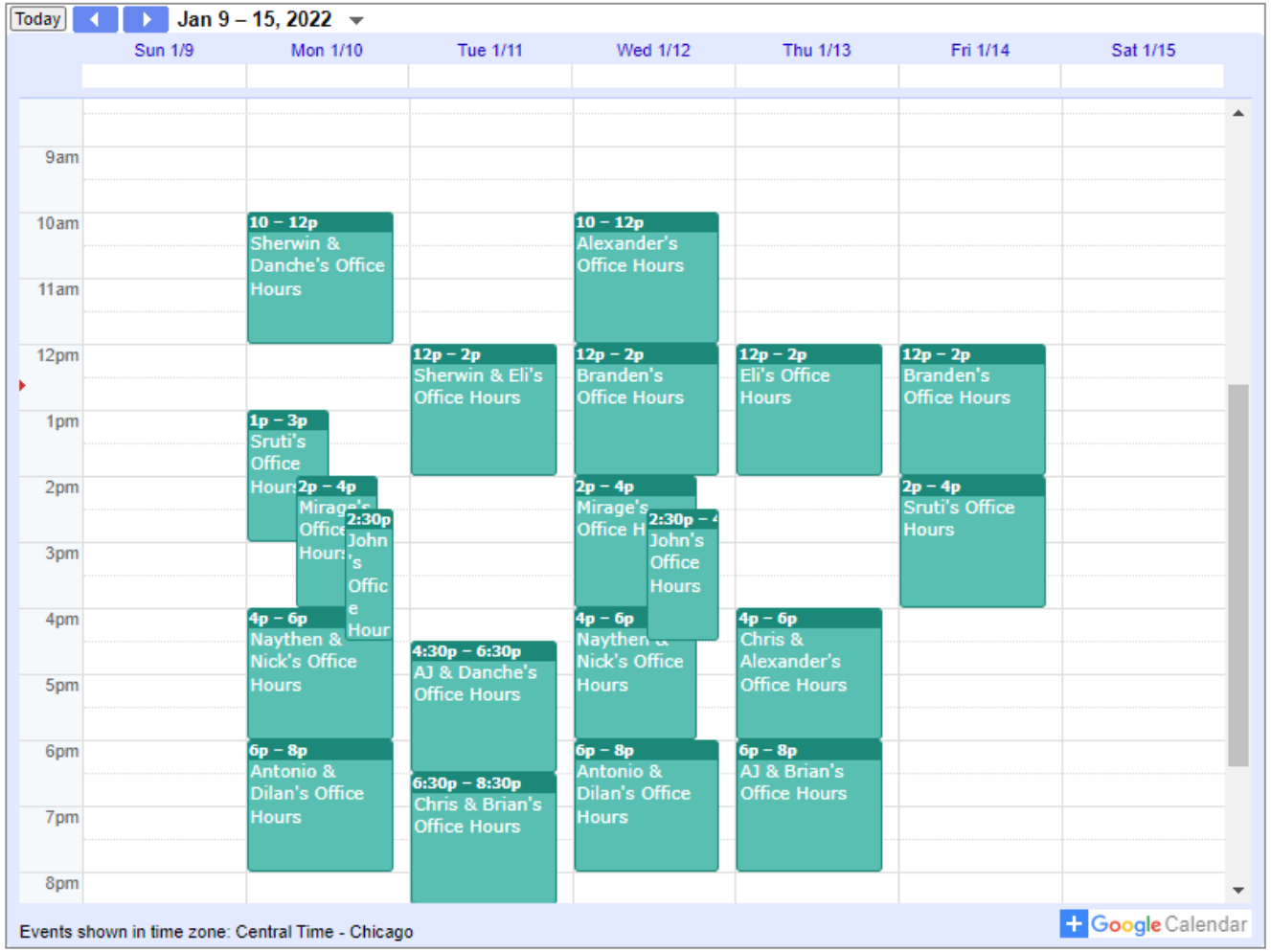
Slides adapted from:
Jesse Tov

Administrivia

- Office hours have started!
- Check Canvas homepage for calendar
- Everyone should have Campuswire access
  - Email me ASAP if you don't

Office Hours:

Office hours at: https://gather.town/app/RP34SKOkOiqVlpqb/NU-CS211-W22-OH
Password: wildcatcomputerscience
Queue signup form - Read-only queue status



Administrivia

- Lab01 is due on **Friday**
  - More than half of you have finished it already 💝
  - Please reach out if you have problems!

- Lab02 is released today
  - Practice C programming
  - Due on Monday

- Hw01 will be released tonight
  - Due next week Thursday

Today's Goals

- Introduce the basics of C programming
  - Compilation
  - Variables
  - Conditionals (if)
  - Iteration (while and for)
  - Input and Output (printf and scanf)

- Continue practicing use of the shell

Getting the examples from lecture

- First, make your own cs211 directory to store class stuff in
  - `cd ~/`
  - `mkdir cs211`

- The files for this class are in a zipped tarball (just like a zip file)
  - We can extract them right into your cs211/ directory

  - `cd ~/cs211/`
  - `tar -xvkf ~cs211/lec/02_intro_c.tgz`
  - `cd 02_intro_c`

  - What does that command do?: https://explainshell.com/explain?cmd=tar+-xvkf+%7Ecs211%2Flec%2F02_intro_c.tgz

Outline

- **Quick Shell Note**

- Hello World in C
- Compilation

- Computing Fibonacci Numbers
- Variables
- Iteration

- Other C Syntax
- Input and Output

Reminder: Helpful guides to shell commands

- Great lecture notes on using the shell
  - https://swcarpentry.github.io/shell-novice/
- Tool to explain various shell command syntax
  - https://explainshell.com/
- Tool to explain how to use various shell commands
  - Just type the command into the box at the top
  - https://tldr.ostera.io/

Shell command: sudo

- Superuser do
  - Executes a command with special administrator privilege (superuser)
  - Necessary for installing new programs and modifying the OS

- Run it before a command to execute that command as a superuser
  - Example: `sudo rm -rf /` (don't run this!)

- You can only use `sudo` on computers where you are an admin
  - Only use with caution and care. It can destroy your computer

  - You'll never need it for class stuff
  - You are NOT an admin on the class servers! (neither am I)

sudo example

```
branden@moore:~% sudo echo "Sorry Pred, I'm testing this for CS211."

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for branden:
```

sudo example

```
branden@moore:~% sudo echo "Sorry Pred, I'm testing this for CS211."

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for branden:
branden is not in the sudoers file.  This incident will be reported.
branden@moore:~ [1]%
```

relevant xkcd (but no break, go read this later)

https://xkcd.com/838/

Outline

- Quick Shell Note

- **Hello World in C**
- Compilation

- Computing Fibonacci Numbers
- Variables
- Iteration

- Other C Syntax
- Input and Output

Hello world C program

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

Hello world C program

hello.c

This is the code file where you can find this code!

See slide near beginning on how to get code from today's lecture.

```
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

Hello world C program

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

Hello world C program

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

A function named `main()`

No Arguments (`void`)

Returns an integer

Hello world C program

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

Call to the `printf()` function

One argument to the function, the string “Hello, CS211\n”

Hello world C program

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

The `printf()` function is a part of the standard input/output library, included here

Call to the `printf()` function

One argument to the function, the string “Hello, CS211\n”

Hello world C program

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

Returns a value, 0
(which is of type `int`)

Hello world C program

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

Two special things going on here:

1. main() is a special function name that is called when the program runs

Hello world C program

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, CS 211!\n");

    return 0;
}
```

Two special things going on here:

1. main() is a special function name that is called when the program runs

2. main() returns a number that specifies whether the program succeeded or failed and how
   - 0 means success
   - non-zero means failure
   - specific numbers mean different things to different programs

Outline

- Quick Shell Note

- Hello World in C
- **Compilation**

- Computing Fibonacci Numbers
- Variables
- Iteration

- Other C Syntax
- Input and Output

How do you "run" C code?

- First, the C code needs to be translated
  - From human-readable source code
  - To machine code capable of being executed on a particular machine (definitely not human readable)
- This translation process is called "compiling"
  - The tool that does it is a "compiler"

What does machine code look like?

- Just a bunch of numbers
  - Your text editor would interpret those numbers as random characters

- The computer processor reads the numbers to figure out which instruction to run
  - This is a version of assembly code
  - See CS213 for *way* more details

Compiling a C program

- The compiler we'll use is referred to as `cc`
  - Short for C Compiler
  - It takes in C source code and outputs *executable* machine code

- `cc hello.c`
- `ls`
  ```
  a.out hello.c
  ```
- `./a.out`
  ```
  Hello, CS 211!
  ```

Let's go to the shell and try it out!

Compiling a C program

- a.out is the default name, but we probably want to use something more memorable
- The `-o` flag specifies the output filename for the compiler

- `cc -o hello hello.c`
- `ls`
  `hello hello.c`

- `./hello`
  `Hello, CS 211!`

Remember to compile!

- You need to re-compile code every time the source code changes

- You WILL forget to do this at some point
  - And you'll run the program but it'll do the old behavior rather than the new things you've written

- Compile often!
  - Keep multiple windows open to make this easier
  - I write a handful of lines of C code, then compile again
    - Way easier to find one or two mistakes now than deal with MANY later

Break + relevant xkcd

https://xkcd.com/303/

Outline

- Quick Shell Note

- Hello World in C
- Compilation

- **Computing Fibonacci Numbers**
- Variables
- Iteration

- Other C Syntax
- Input and Output

Definition of Fibonacci Function

- $fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$

n	fib(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21

Implementing Fibonacci in C

fib.c

```
long fib(int n){
   if (n < 2) {
      return n;
   } else {
      return fib(n - 2) + fib(n - 1);
   }
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Recursion works in C!

Implementing Fibonacci in C

fib.c

```
long fib(int n){
  if (n < 2) {
    return n;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

```
if (⟨test-expr⟩) { // evaluate ⟨test-expr⟩; then…
  ⟨then-stms⟩ // do these if ⟨test-expr⟩ was true
} else {
  ⟨else-stms⟩ // do these if ⟨test-expr⟩ was false
}
```

Statements can be nested in C

```
if (⟨first-test-expr⟩) {
  if (⟨second-test-expr⟩) {
    ⟨A-stms⟩
  } else {
    ⟨B-stms⟩
  }
} else {
  if (⟨third-test-expr⟩) {
    ⟨C-stms⟩
  } else {
    ⟨D-stms⟩
  }
}
```

C ignores most whitespace

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

```
long fib(int n){
   if (n < 2) {
      return n;
   } else {
      return fib(n - 2) + fib(n - 1);
   }
}
```

C ignores most whitespace

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

```
long fib(int n){
  if (n < 2) {
    return n;
  } else {
    return fib(n - 2) +
           fib(n - 1);
  }
}
```

C doesn't care about whitespace

C ignores most whitespace

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

```
long fib(int n){if(n<2){return n;}else{return
fib(n-2)+fib(n-1);}}
```

C **really doesn't care** about whitespace

C ignores most whitespace

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

```
long fib(int n){if(n<2){return n;}else{return
fib(n-2)+fib(n-1);}}
```

C **really doesn't care** about whitespace

But humans do!

So don't write your code this way!!!!!!!!!!

A note on style

- A lot of things are *possible* in C, but bad ideas
  - They can make things hard to read
  - They can be a source of bugs in code

- We try to provide you with what we think of as "good" C code

- We have a guide to how you should write your C code
  - This is a (small) portion of your grade on each assignment!
  - https://nu-cs211.github.io/cs211-files/cstyle.html

Outline

- Quick Shell Note

- Hello World in C
- Compilation

- Computing Fibonacci Numbers
- **Variables**
- Iteration

- Other C Syntax
- Input and Output

Values, objects, and variables

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

Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

- What happens?

Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

- What happens?
  1. The first statement is a definition.
     It creates an `int` object,
     names it `z`,
     and initializes it to the value `5`

Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

- What happens?
  2. The second statement is an assignment.
     It replaces the value `5`
     stored in the object named by `z`
     with the value `7`.

Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

- What happens?

z: 11

3. The third statement is also an assignment.
   It retrieves the current value of `z` (which is `7`),
   then adds `4` to it,
   and then stores the result back in the object named by `z`.

C: Typed imperative programming

- Imperative programming
  - Each line is a **statement** that changes the program's **state**
  - Usually, the values within a variable

- Type System
  - Variables have a type associated with them

  - The type determines qualities of the *object*
    - Example: how much memory it takes up

  - The type specifies what kind of *value* the variable holds
    - Example: integers, decimal numbers, strings, etc.

Types in C

- Hold an integer number (like 5 or 0 or -3)
  - `char`, `short`, `int`, `long`, `size_t`, `int8_t`, `int16_t`, `int32_t`, etc.
  - These can also specify signedness
    - `unsigned`: only 0 and greater
    - `signed`: negative, 0, or positive

- Hold a decimal number (like 6.238 or 0.00001 or -32566.5)
  - `float`, `double`
  - These are always negative, 0, or positive

- Difference between types: how big of a value they can hold
  - `short`: 0 to 65536 OR `signed short`: -32768 to 32767
  - `int`: 0 to 4294967296 OR `signed int`: -2147483648 to 2147483647

  - We'll have a whole future lecture on *why* the types are like this

Signed vs unsigned variables

- All “integer” types in C can be signed or unsigned
  - char, short, int, long, etc.

  - Unsigned: only zero or positive
  - Signed: negative, zero, or positive

- Signed is the default! If it doesn’t say, it’s usually signed
  - An exception is `size_t` which is unsigned

- Comparing signed and unsigned numbers generates a warning
  - Should make sure they’re the same before comparing

Temporarily changing types while comparing

- You can cast a variable to another type during an expression
  - To cast, put a type in parentheses before the variable name

- Example

```
int i = 0;                  //int is signed by default
size_t length = 5;          //size_t is unsigned

if (i > length) {           // warning here!
   printf("Too big!\n");
}
```

Temporarily changing types while comparing

- You can cast a variable to another type during an expression
  - To cast, put a type in parentheses before the variable name

- Example

```
int i = 0;                  //int is signed by default
size_t length = 5;          //size_t is unsigned

if (i > (int)length) { // no warning anymore!
   printf("Too big!\n");
}
```

`typedef` can be used to make new C type names

- Typedef creates a new type name that is a copy of an existing type

- Typedef keyword is followed by two types
  - First type: the original type name
  - Second type: the new type name

- Example:

```
typedef int x_coordinate_t;
x_coordinate_t my_variable = 5;
```

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev: 

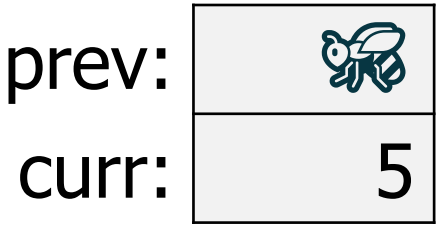
More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```



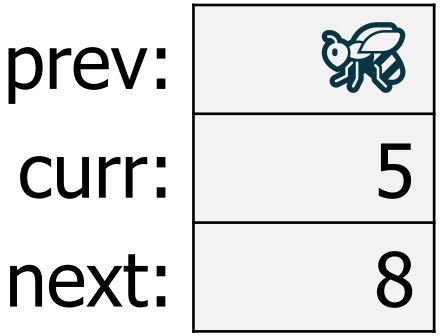
More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```



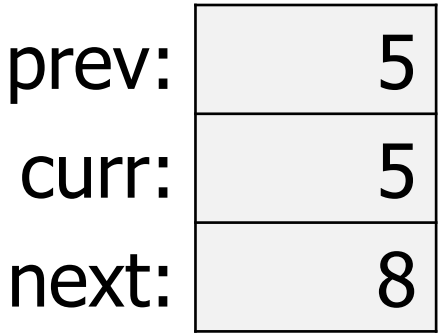
More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```



More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev:	5
curr:	8
next:	8

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev:	5
curr:	8
next:	13

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev:	8
curr:	8
next:	13

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev:	8
curr:	13
next:	13

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev:	8
curr:	13
next:	21

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev:	13
curr:	13
next:	21

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev:	13
curr:	21
next:	21

More complicated example

```
int prev;
int curr = 5;
int next = 8;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;

prev = curr;
curr = next;
next = prev + curr;
```

prev:	13
curr:	21
next:	34

Outline

- Quick Shell Note

- Hello World in C
- Compilation

- Computing Fibonacci Numbers
- Variables
- **Iteration**

- Other C Syntax
- Input and Output

Statements and Conditions aren’t enough

- Those lines of code were actually implementing Fibonacci!
  - And they were doing it without requiring any recursion

- Problem: it’s really repetitive to have to write out the same lines of code again and again
- Solution: Iteration

Iteration with the While Statement

- Syntax

```
while (⟨test-expression⟩) {
  ⟨body-statements⟩
}
```

- Semantics
  1. Evaluate ⟨test-expression⟩ to a `bool`
  2. If the `bool` is *false* then skip to the statement after the `while` loop
  3. Execute ⟨body-statements⟩ (if the `bool` was true)
  4. Go back to step 1

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        n = n - 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

For loops

- For loops allow you to combine iteration and incrementing
  - When you write a for statement like this:

```
for (⟨start-decl⟩; ⟨test-expr⟩; ⟨step-expr⟩) {
  ⟨body-stms⟩
}
```

  - It's as if you'd written this while statement:

```
{
  ⟨start-decl⟩;
  while (⟨test-expr⟩) {
    ⟨body-stms⟩
    ⟨step-expr⟩;
  }
}
```

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        n = n - 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;
    int i = 0;

    while (i < n) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        i = i + 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;
    int i = 0;

    for ( ; i < n; ) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        i = i + 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;
    // int i = 0;

    for (int i = 0; i < n; ) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        i = i + 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    for (int i = 0; i < n; i = i + 1) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        //i = i + 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    for (int i = 0; i < n; i = i + 1) {
        long prev = curr;

        curr = next;
        next = prev + curr;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Break + Question

- What value will this code return when called as:
  - loop_function(6)
  - loop_function(5)
  - loop_function(3)

```
int loop_function(int test) {
   int retval = 0;
   while (test < 5) {
      retval = retval + 1;
      test = test + 1;
   }
   return retval;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6) **returns 0**
  - loop_function(5)
  - loop_function(3)

```
int loop_function(int test) {
   int retval = 0;
   while (test < 5) {
      retval = retval + 1;
      test = test + 1;
   }
   return retval;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6) **returns 0**
  - loop_function(5) **returns 0**
  - loop_function(3)

```
int loop_function(int test) {
   int retval = 0;
   while (test < 5) {
      retval = retval + 1;
      test = test + 1;
   }
   return retval;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6) **returns 0**
  - loop_function(5) **returns 0**
  - loop_function(3) **returns 2**

```
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval = retval + 1;
    test = test + 1;
  }
  return retval;
}
```

Outline

- Quick Shell Note

- Hello World in C
- Compilation

- Computing Fibonacci Numbers
- Variables
- Iteration

- **Other C Syntax**
- Input and Output

C comments

- `//` means a single-line comment
- `/*` starts a multiline comment, which continues until `*/`

- How to use comments effectively
  - Comment "blocks" of code with their purpose
    - Every line is too much
    - Often helpful to write the comments before the code as planning
  - Comment tricky bits of code so you know what it means
    - You + several weeks = "what does that code mean?!"

Logical operators

- || &&
  - Logical OR, and Logical AND
  - `a < 5 && b > 12`

- !
  - Logical NOT
  - `!(a < 5)` equivalent to `(a >= 5)`

- ==
  - Equality test
  - `5 == 5` -> TRUE
  - `16 == -3` -> FALSE

  - Don’t mix it up with assignment (single equals sign)
    - Really common new C programmer mistake

Other operators you'll see around

- `+=  *=  -=  /=`
  - Perform the action of `VAR = VAR operator ARG`
  - `a += 5` -> `a = a + 5`
  - `a *= b` -> `a = a * b`

- `%`
  - Modulus operator
  - Returns the remainder of division
  - `12 % 10` -> 2

- `~  |  &  ^`
  - Bitwise NOT, OR, AND, and XOR (you'll learn these in CS213)
  - Importantly, ^ is not exponentiation!!!

Adding and Subtracting one

- `++  --`
  - Shorthand for plus 1 or minus 1
  - `++a  -> a += 1  -> a = a + 1`

- The auto-increment/decrement operators can go before or after the variable
  - (`--x`) subtracts one and returns the new value of x from the expression
  - (`x--`) subtracts one but returns the *old* value of x from the expression

  - Usually, this doesn't matter, unless you write complicated statements that combine assignment and conditions
  - `if (--x > 0)` … (please just never do this)

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    for (int i = 0; i < n; ++i) {  // i++ also works
        long prev = curr;

        curr = next;
        next = prev + curr;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Ternary Operator

- ? :
  - Shorthand version of an if statement, determining result of expression
  - Example:
    - `return (a < 5) ? a : b;`

      equivalent to

    - ```
      if (a < 5) {
        return a;
      } else {
        return b;
      }
      ```
  - You won't need to use this. Usually, it just makes code harder to read.
      ```

Outline

- Quick Shell Note

- Hello World in C
- Compilation

- Computing Fibonacci Numbers
- Variables
- Iteration

- Other C Syntax
- **Input and Output**

printf() function

- The usual way to print in C is the `printf()` function
  - Takes a *format string* followed by arguments to *interpolate* in place of the string's format specifiers

```
printf("(%d, %d)\n", x, y);
```

`%d` format specifier means the argument is an `int`

Prints "`(`" + the value of x + "`, `" + the value of y + "`)\n`"

- `printf()` is in the stdio.h library, which needs to be `#include`-ed

Example: formatted output

output.c

```c
#include <stdio.h>
int main(void){
  int x = 5;
  double f = 5.1;
  printf("sizeof x: %zu bytes\n", sizeof(x));
  printf("sizeof f: %zu bytes\n", sizeof(f));
  printf("x: %d\nf: %.60e\n", x, f);
}
```

- A format specifier gives the argument's type and maybe some options
  - `%zu` type: `size_t` (the return result of `sizeof`)
  - `%d` type: `int`
  - `%.60e` type: `double`, include 60 digits of precision

How do you learn format specifiers?

- You look them up in a guide!
  - Even I don't have them memorized…

- `man 3 printf`
  - Runs in the terminal
  - Shows details about printf

- google "printf format specifiers" (this is what I do)
  - cplusplus.com is a good resource
  - https://www.cplusplus.com/reference/cstdio/printf/

Reading user input

- To input numbers in C, use the `scanf()` function

- scanf reads keyboard input, converts it to the require type, and stores it in an existing variable:

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

  - Like `printf()`, `scanf()` uses a format string to determine what type to convert the input into
  - `&x` means to pass x's location, not its value (more on this next week)
  - Careful: `scanf()` directives aren't exactly the same as `printf()`

Example: reading input

input.c

```c
#include <stdio.h>

double sqr_dbl(double n){
  return n * n;
}

int main(void){
  double d = 0.0;
  scanf("%lf", &d);
  printf("%lf squared is %lf\n", d, sqr_dbl(d));
}
```

Example: reading multiple items

multi_input.c

```c
#include <stdio.h>
int main(void){
   int x;
   int y;
   printf("Enter two integers: ");
   scanf("%d%d", &x, &y);
   printf("%d * %d = %d\n", x, y, x * y);
}
```

What if scanf() has an error?

check_input.c

- `scanf()` returns the number of successful conversions

```
#include <stdio.h>
int main(void){
   int x
   int y;
   printf("Enter two integers: ");
   if (scanf("%d%d", &x, &y) != 2) {
      printf("Input error\n");
      return 1;
   }
   printf("%d * %d == %d\n", x, y, x * y);
}
```

You now know the basics of C programming!

- We’re missing a few simple things
  - You’ll practice those in Lab02 and Hw01
  - Structs!

- We’re missing some advanced features
  - We’ll cover those next week

Outline

- Quick Shell Note

- Hello World in C
- Compilation

- Computing Fibonacci Numbers
- Variables
- Iteration

- Other C Syntax
- Input and Output