Programming in C
NCI HPC Toolkit

Stephen Sanderson

National Computational Infrastructure, Australia

Acknowledgement of Country

The National Computational Infrastructure acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Acknowledgements

- Fred Fung
- Ben Menadue
- Paul Leopardi
- Rui Yang
- Maruf Ahmed
- Jingbo Wang

Outline

This course presents an introduction to programming in C with little to no assumed prior knowledge.

It covers:

- Syntax
- Control flow
- Data types
- Pointers and memory management
- Common pitfalls and best practices
- Basic compiler options

Not covered: C++, multithreading, profiling, many other things

A great reference: *Modern C*, by Jens Gustedt

Introduction

C is a statically typed systems programming language.

It is:

- Very fast (when used correctly)
- Cross-platform
- Close to the hardware (lots of control)
- Supported by many scientific libraries (e.g. MPI, OpenMP, OpenACC)
- One of the most widely used programming languages (e.g. Linux, Git, VMD, Doom)
- Standardised (ISO **C99**, C11, C17, C23)

Introduction

C is a statically typed systems programming language.

It is:

- Very fast (when used correctly)
- Cross-platform
- Close to the hardware (lots of control)
- Supported by many scientific libraries (e.g. MPI, OpenMP, OpenACC)
- One of the most widely used programming languages (e.g. Linux, Git, VMD, Doom)
- Standardised (ISO **C99**, C11, C17, C23)

It is not:

- Always the best choice for quick development
- The most memory-safe language
- C++

Hello, world!

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Hello, world!

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

- Include standard input/output library
  - Gives access to the `printf` function (and many others)
- `#include` says "Paste the contents of this file here."
  - Use `<>` for external (e.g. system) files, `""` for project files.

Hello, world!

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

- Declare the `main` function
- The compiler looks for this function as the entry point
- `int` is the return type
- `()` marks the input arguments
  - Not taking any for now. We'll come back to this later.
- `{}` marks the body of the function

Hello, world!

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

- Print "Hello, world!" to the terminal
- `'\n'` prints a new line at the end
- `;` marks the end of a code line

Hello, world!

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

- Return a 0 to indicate that program execution succeeded
- Can return other values to indicate various errors

Hello, world!

How do we make it run?

Compiling

- The compiler takes C code and turns it into a binary executable
- Some common C compilers include:
  - `gcc`
  - `icc`
  - `clang`
- This course focuses on the `gcc` compiler, but the core concepts are transferable to others.

```
$ gcc main.c -o hello_world
$ ./hello_world
Hello, world!
$
```

Compiling

- A short summary of some of the most common compiler flags (for `gcc`).

Flag	Purpose
`-o <exe_name>`	Output file (default `a.out`)
`-Wall`	Turn on warnings
`-Wextra`	Turn on extra warnings
`-Wpedantic`	Even more warnings
`-Werror`	Treat warnings as errors
`-g`	Include debugging information in the binary
`-O<level>`	Enable optimisations from 0 (no optimisation) to 3 (includes some potentially unsafe optimisations).
`-D<macro>[=<defn>]`	Define a macro (we'll come back to this later)

Compiling

- A short summary of some of the most common compiler flags (for `gcc`).
- It's a good idea to turn on as many warnings as possible!

Flag	Purpose
`-o <exe_name>`	Output file (default `a.out`)
`-Wall`	Turn on warnings
`-Wextra`	Turn on extra warnings
`-Wpedantic`	Even more warnings
`-Werror`	Treat warnings as errors
`-g`	Include debugging information in the binary
`-O<level>`	Enable optimisations from 0 (no optimisation) to 3 (includes some potentially unsafe optimisations).
`-D<macro>[=<defn>]`	Define a macro (we'll come back to this later)

Compiling

- A short summary of some of the most common compiler flags (for `gcc`).
- It's a good idea to turn on as many warnings as possible!
- In general, aim to get code working with `-O0` and then work up while checking speed and correctness.

Flag	**Purpose**
`-o <exe_name>`	Output file (default `a.out`)
`-Wall`	Turn on warnings
`-Wextra`	Turn on extra warnings
`-Wpedantic`	Even more warnings
`-Werror`	Treat warnings as errors
`-g`	Include debugging information in the binary
`-O<level>`	Enable optimisations from 0 (no optimisation) to 3 (includes some potentially unsafe optimisations).
`-D<macro>[=<defn>]`	Define a macro (we'll come back to this later)

Compiling

- A short summary of some of the most common compiler flags (for `gcc`).
- It's a good idea to turn on as many warnings as possible!
- Beware the `-Ofast` and `-ffast-math` options. They can make code faster, but come at the cost of potential floating point error. Avoid using them when compiling libraries that others will use!

Flag	Purpose
`-o <exe_name>`	Output file (default `a.out`)
`-Wall`	Turn on warnings
`-Wextra`	Turn on extra warnings
`-Wpedantic`	Even more warnings
`-Werror`	Treat warnings as errors
`-g`	Include debugging information in the binary
`-O<level>`	Enable optimisations from 0 (no optimisation) to 3 (includes some potentially unsafe optimisations).
`-D<macro>[=<defn>]`	Define a macro (we'll come back to this later)

Programming is often easiest to learn by doing.

Programming is often easiest to learn by doing.

Let’s build something useful!

Fibonacci Sequence

Goal

Print all Fibonacci numbers less than or equal to some input value.

Example input/output:

```
$ ./fibonacci 12
1 1 2 3 5 8
$ ./fibonacci 0
$ ./fibonacci 1
1 1
$ ./fibonacci asdf
ERROR: invalid input!
$
```

Fibonacci Sequence

Program flow:

- Read input
- Check that it's a positive integer
- Print the Fibonacci numbers

Fibonacci Sequence - Reading input

First we need to read user input and store it somewhere.

To do this, we need to know about variables.

Variables

- C is statically typed, which means variables must be declared with a data type.
- Variable names must begin with a letter or underscore, and can only contain letters, numbers, and underscores.
- The primitive types are:

```
[signed/unsigned] long/short/int
float/[long] double
bool (or _Bool)
[signed/unsigned] char
void
```

- See https://en.wikipedia.org/wiki/C_data_types#Main_types for a full list

Variables - Examples

```c
int my_int = -12345;                 // int uses 32 bits (4 bytes)
short my_small_int = 32767;          // short uses 16 bits
long my_large_int = 9999999999999;   // long uses 64 bits
long long very_large = 99999999999999999999; // long long = 128 bit
unsigned int positive_only = 3200000000;
unsigned char small_positive = 255; // char uses 8 bits
char my_char = 'a';           // Can be numeric or ASCII character

float almost_pi = 3.14;            // 32 bit floating point value
double my_double = 3.141592;       // 64 bit floating point value
long double my_128bit_float = 3.14159265358979;    // 128 bit

_Bool intrinsic_bool = 1; // 0 = false, 1 = true
#include <stdbool.h>
bool my_bool = true; // stdbool.h allows 'bool' and 'true/false'
```

Variables - Examples

```
// Variables can be forward-declared
int a, b, c;   // values currently undefined
a = 5;         // store 5 in a
b = c = 10;    // store 10 in b and c

// types can be implicitly converted, but be careful!
float f = 1.2, fa = a;
int one = f;           // information is lost here! one = 1
double zero = 1/3;  // 1 and 3 are integers. Integer division!
double one_third = 1.0/3; // 3 converted to double to do division
float f_third = one_third; // Less precise data type. Info lost!

// Can explicitly cast types into other types
float not_one_half = a / b;       // Uses integer division!
float one_half = (float) a / b; // a converted to float first
```

Variables - Scope

```
{ // Begin outer scope
    int a = 10;
    { // Begin inner scope
      int b = 5;
      // Variables from outer scope are still accessible
      int c = a + b;
      // Can re-declare 'a' here, but now we can't access
      // the outer 'a'
      int a = 4;
    } // End inner scope
    // outer 'a' is still 10

    // Variables from the inner scope can't be accessed anymore.
    // This is a compile error:
    a = b; // ERROR: use of undeclared identifier 'b'
}
```

Variables - Global

Global variables are those declared in the global scope (outside any functions). E.g.:

```
// Declare a constant to be used anywhere in the program
const int THE_ANSWER = 42;

// Declare a global variable that can be CHANGED anywhere.
// This can be dangerous and lead to unintended bugs!
// Avoid wherever feasible.
int MY_GLOBAL;

// Gain access to a global variable declared in a different
// translation unit
extern int EXTERNAL_GLOBAL;

// Only visible within this translation unit
static int FILE_ONLY = 123;

int main(void) {
  /* Do things with global variables */
}
```

Variables - Pointers & References

- Variables are just named chunks of memory
- Each chunk has an *address*
- References give the address of the memory storing a variable
- Pointers can be used to store the address
- A pointer can be *dereferenced* to access the memory it's pointing to.

0x00	5	`int a = 5;`
0x04		`int b;`
0x08		
0x0C		
0x10		
0x05		
0x06		
0x07		

Variables - Pointers & References

0x00	5	`int a = 5;`
0x04	10	`int b; b = 10;`
0x08		
0x0C		
0x10		
0x05		
0x06		
0x07		

Variables - Pointers & References

0x00	5	`int a = 5;`
0x04	10	`int b; b = 10;`
0x08	0x00	`int* addr_a = &a;`
0x0C		
0x10		
0x05		
0x06		
0x07		

Variables - Pointers & References

0x00	5	`int a = 5;`
0x04	10	`int b; b = 10;`
0x08	0x00	`int* addr_a = &a;`
0x0C	0x04	`int* addr_b = &b;`
0x10		
0x05		
0x06		
0x07		

Variables - Pointers & References

0x00	5	`int a = 5;`
0x04	10	`int b; b = 10;`
0x08	0x00	`int* addr_a = &a;`
0x0C	0x04	`int* addr_b = &b;`
0x10	0x0C	`int** addr_addr_b = &addr_b;`
0x05		
0x06		
0x07		

Variables - Pointers & References

Address	Value	Code
0x00	5	`int a = 5;`
0x04	10	`int b; b = 10;`
0x08	0x00	`int* addr_a = &a;`
0x0C	0x04	`int* addr_b = &b;`
0x10	0x0C	`int** addr_addr_b = &addr_b;`
0x05	10	`int copy_b = *addr_b;`
0x06		
0x07		

Variables - Pointers & References

Address	Value	Code
0x00	5	`int a = 5;`
0x04	10	`int b; b = 10;`
0x08	0x00	`int* addr_a = &a;`
0x0C	0x04	`int* addr_b = &b;`
0x10	0x0C	`int** addr_addr_b = &addr_b;`
0x05	10	`int copy_b = *addr_b;`
0x06	0x04	`int* copy_addr_b = addr_b;`
0x07		

Variables - Pointers & References

```
int a = 5;
int* ptr = &a;   // ptr points to a
*ptr = 15;       // a is now 15

int b = 2;
ptr = &b;        // ptr now points to b
*ptr = 4;        // b is now 4. a is still 15.

ptr = 128;       // ptr now points to address 128.
                 // We don't know what's stored there!

b = *ptr;        // This is undefined behaviour!
                 // b could be anything now, or the
                 // program might crash with a
                 // segmentation fault
```

Variables - Pointers & References

Although it's clearer to attach the * to the type, be aware that it actually binds to the variable name!

```
int* p1, i1;      // This declares p1 as a pointer to int,
                  // but i1 as just an int!
int *p2, *p3;     // Use * on each name that should be a pointer
```

Variables - Pointers & References

Although it's clearer to attach the * to the type, be aware that it actually binds to the variable name!

```
int* p1, i1;      // This declares p1 as a pointer to int,
                  // but i1 as just an int!
int *p2, *p3;     // Use * on each name that should be a pointer
```

Beware of dangling pointers!

```
int* my_ptr;
{
    int my_value = 10;
    my_ptr = &my_value;
} // my_value falls out of scope here!

// This is undefined behaviour since
// my_value no longer exists!
int another_value = *my_ptr;
```

Variables - Arrays

```c
int my_array[] = {1, 2, 3, 4, 5};
int another_array[10]; // forward declare a 10 element array

my_array[0] = 6;                // Array is now {6, 2, 3, 4, 5}

my_array[3] = my_array[2];      // Array is now {6, 2, 3, 3, 5}

int* same_array = my_array;     // my_array is just a pointer!
*same_array = 10;               // Array is now {10, 2, 3, 3, 5}
*my_array = 4;                  // Array is now {4, 2, 3, 3, 5}
same_array[4] = 12;             // Array is now {4, 2, 3, 3, 12}
*(same_array+2) = 9;            // Array is now {4, 2, 9, 3, 12}

another_array[10] = my_array[2]; // Undefined behaviour!
```

Arrays of pointers and pointers to arrays

```
// To declare an array of pointers, the syntax is as expected
int* array_of_ptrs[10];  // Store 10 pointers
// array_of_ptrs has the type int*[10]

// For a pointer to an array, use
int (*ptr_to_array)[10]; // Point to an array of length 10
// ptr_to_array has the type int(*)[10]
```

Variables - Strings

```c
// Strings are just arrays of characters
char my_string[] = "Hello, world.";
my_string[12] = '!';
// my_string is now "Hello, world!"

// Note, use single quotes, '', for single chars
// or double quotes, "", for strings.

// We can have a list of strings:
char string_list[][10] = {"Hello", "world", "something"};
// Note, [10] is required, and must be the length of the
// longest element (something\0)
// Also note, string_list is of type char**
```

Variables - Stack vs Heap

- Variables allocated to the stack by default
  - Stack size is limited, so large data may not fit (stack overflow!)
  - In general, stack variable size should be known at compile time.
  - Exception is variable length arrays (e.g. `int vla[my_int];`), but make sure size is checked to be valid!
- Heap used for unknown sizes or large data types
  - Allocate with functions like `malloc()`
  - Need to manually de-allocate (`free()`) when finished, otherwise we cause a memory leak!
- Everything we've seen so far has been stack allocated.

```c
// Allocate space for 1000 ints
int* heap_array = \
    malloc(sizeof(int) * 1000);
if (!heap_array) {
    printf("ALLOCATION FAILED\n");
    // Handle error here
}
// heap_array now points to the
// 1st element of the memory block
heap_array[100] = 1; // Store 1

// de-allocate the memory
free(heap_array); // Don't forget!

// This is undefined behaviour!!
int a = heap_array[100];
```

Variables - Multi-Dimensional Heap Arrays

```c
// Allocate space for a pointer to each row
const int nrows = 3, ncols = 3;
int** heap_3x3_matrix = malloc(sizeof(int*) * nrows);

// Allocate space for all columns of each row
heap_matrix[0] = malloc(sizeof(int) * ncols);
heap_matrix[1] = malloc(sizeof(int) * ncols);
heap_matrix[2] = malloc(sizeof(int) * ncols);

heap_matrix[0][2] = 1;  // Store 1 in row 0, column 2
heap_matrix[2][1] = 5;  // Store 5 in row 2, column 1
```

Variables - Multi-Dimensional Heap Arrays

```c
// Allocate space for a pointer to each row
const int nrows = 3, ncols = 3;
int** heap_3x3_matrix = malloc(sizeof(int*) * nrows);

// A better alternative that keeps the matrix in
// a single chunk of memory
heap_matrix[0] = malloc(sizeof(int) * nrows * ncols);
heap_matrix[1] = &heap_matrix[0][1*3];
heap_matrix[2] = &heap_matrix[0][2*3];

heap_matrix[0][2] = 1;  // Store 1 in row 0, column 2
heap_matrix[2][1] = 5;  // Store 5 in row 2, column 1
```

Variables - Multi-Dimensional Heap Arrays

```c
// Allocate space for 3x3 integers
const int nrows = 3, ncols = 3;
int* heap_3x3_matrix = malloc(sizeof(int) * nrows * ncols);

// Another good alternative is to use a 1D array and some
// clever indexing
heap_matrix[0*ncols + 2] = 1;  // Store 1 in row 0, column 2
heap_matrix[2*ncols + 1] = 5;  // Store 5 in row 2, column 1
```

Back to the Fibonacci Sequence

We need to take an input argument that sets the upper bound of the sequence.

```
$ ./fibonacci 12
```

Back to the Fibonacci Sequence

We need to take an input argument that sets the upper bound of the sequence.

```
$ ./fibonacci 12
```

We can do this with the `main()` function!

Input Arguments

```c
int main(int argc, char* argv[]) {
    // argc tells us how many arguments there were
    // argv is a list of strings - one element for each argument
    // argc will always be >= 1
    // argv[0] will be the name of the executable
    return argc;
}
```

Running this code:

```
$ gcc main.c -o return_argc
$ ./return_argc; echo $?
1
$ ./return_argc hello world "one string"; echo $?
4
```

Back to the Fibonacci Sequence

We want to check whether we were given an input argument, and make sure that argument was an integer.

Back to the Fibonacci Sequence

We want to check whether we were given an input argument, and make sure that argument was an integer.

We need operators and control flow!

Operators

- We've already been using a few operators.
- They generally fall under one of 3 categories:
  - Mathematical
  - Boolean
  - Bit-wise

Operators

- We've already been using a few operators.
- They generally fall under one of 3 categories:
  - Mathematical
  - Boolean
  - Bit-wise

Mathematical

```c
int a, b, c;
// ... set a and b
c = a;         // Assignment operator
c = (a = b);   // Result is the RHS value
c = a = b;     // Equivalent to previous line

c = a + b;   // Addition
c = a - b;   // Subtraction
c = a * b;   // Multiplication
c = a / b;   // Division (rounds down for int)
c = a % b;   // Modulo (remainder)
```

Operators

- We've already been using a few operators.
- They generally fall under one of 3 categories:
  - Mathematical
  - Boolean
  - Bit-wise

Mathematical

```
c += a;        // c = c + a;
c -= a;        // c = c - a;
c *= a;        // c = c * a;
c /= a;        // c = c / a;
c %= a;        // c = c % a;

c++;           // Add 1 to c. Returns original c
c--;           // Subtract 1 from c. As above
++c;           // Add 1 to c. Returns new c value
--c;           // Subtract 1 from c. As above

// Example:
a = 10;        // Store 10 in a
b = a++;       // Store 10 in b, inc. a to 11
c = --a;       // Decrement a to 10. Store 10 in c
```

Operators

- We've already been using a few operators.
- They generally fall under one of 3 categories:
  - Mathematical
  - Boolean
  - Bit-wise

Boolean

```
bool w, x, y, z;
x = a > b;          // Greater than
x = a < b;          // Less than
y = a >= b;         // Greater than or equal to
y = a <= b;         // Less than or equal to
z = a == b;         // Equal to
w = a != b;         // Not equal to

y = !x;             // Not
z = x || y;         // Or (short-circuit)
z = x && y;         // And (short-circuit)
z = !(x && !y || w); // Can be combined

// Ternary
c = x ? a : b;      // if (x) c = a; else c = b;
```

Operators

- We've already been using a few operators.
- They generally fall under one of 3 categories:
  - Mathematical
  - Boolean
  - Bit-wise

Bit-wise

```
a = 0b010 | 0b001;   // Bit-wise or. a == 0b011
a = 0b110 & 0b011;   // Bit-wise and. a == 0b010
a = 0b110 ^ 0b011;   // Bit-wise xor. a == 0b101
// NOTE: bit-wise operators don't short-circuit
a = 0b100 >> 1;      // Right shift. a == 0b010
a = 0b001 << 1;      // Left shift. a == 0b010
a = ~0b011;          // Bit-wise not.
// Since 0b011 is 32bit int, this evaluates to
//  a = 0b11111111111111111111111111111100

a |= b;      // a = a | b;
a &= b;      // a = a & b;
a ^= b;      // a = a ^ b;
a <<= 1;     // a = a << 1; Equivalent to a*2
a >>= 2;     // a = a >> 2; Equivalent to a/4
```

Operators - Use with care!

Beware of implicit type conversions!

```
int a = 2147483648;             // This stores -2147483648 in a!
unsigned int b = 1 - 2;         // This stores 4294967295 in b!
unsigned short c = 0x10000;     // This stores 0 in c!

float d  = 1/3;                 // This stores 0 in d!
float d1 = 1.0/3;               // This works as expected
float d2 = 1/3.0;               // So does this
float d3 = (float)1/3;          // And this (equivalent to 1.0f/3)

float e_f = 1.0;
int e_i = (int)e_f;             // This stores 1 in e_i.
int* e_i_ptr = (int*)&e_f;      // This creates a pointer of type int*
                                //   that points to data of type float!
int i_from_ptr = *e_i_ptr;      // This stores 1065353216 in i_from_ptr!
```

Operators - Use with care!

Check operator precedence!

```
// Addition binds more tightly than shift operators
unsigned int a = 1 << 1 + 1; // This stores 4 in a!
// == binds more tightly than bit-wise operators
unsigned int b = 1 ^ 1 == 0; // This stores 1 in b!
```

Be aware of boolean evaluation.

```
// Values equal to 0 result in false, everything else gives true
bool a = 0x100000000;                // This stores true in e
bool b = (unsigned int)0x100000000; // This stores false!
bool c = (float)1.0; // This stores true
bool d = (float)0.0; // This stores false (0.0 == 0)

int* ptr = 0;        // This is a null pointer
bool e = ptr;        // This stores false in e
bool* e_ptr = &e;    // This is a pointer to e
bool f = e_ptr;      // This stores true in f
```

Onto control flow

Control Flow - `if` statements

```c
bool condition, other_condition;
// ... code that sets condition to something

if (condition) {
    printf("Condition was true!\n");
} else {
    printf("Condition was false!\n");
}

// { } can be omitted if body is a single line
if (other_condition)
    printf("other_condition was true!\n");
// else statement can also be omitted if not required
```

Control Flow - `if` statements

```c
int a, b, c;
// ... code that sets a, b, and c

// Statements can be chained for more complex logic
if (a > b) {
    printf("a greater than b!\n");
} else if (a > c) {
    printf("a greater than c, but not b!\n");
} else printf("a is the smallest!\n");
```

Note

It's good practice to indent the body of control flow statements to make the code clearer and easier to read.

Control Flow - `if` statements

```c
int a, b, c;
// ... code that sets a, b, and c

// Statements can also be nested!
if (a > b) {
    printf("a greater than b, ");
    if (a > c)
        printf("and also c!\n");
    else
        printf("but not c!\n");
} else if (a > c) {
    printf("a greater than c, but not b!\n");
} else printf("a is the smallest!\n");
```

Control Flow - `switch` statements

Suppose we want to handle various values of some variable.

With an if statement, we would write:

```
if (var == 0) {
    // do something
} else if (var == 1) {
    // do something else
} else if (var == 3) {
    // do a different thing
} else {
    // do the default thing
}
```

With a switch statement, we would write:

```
switch (var) {
    case 0:
        // do something
        break;
    case 1:
        // do something else
        break;
    case 3:
        // do a different thing
        break;
    default:
        // do the default thing
}
```

Control Flow - `switch` statements

Switch statements can "fall through" to cover multiple options.

```
switch (var) {  // Note that var must be an integer type
    case 0: // and the values must be constant expressions
        // do something for 0 and fall through
    case 1:
    case 3:
        // do this for values of 0, 1, and 3
        break;
    case 5:
        // do a separate thing
        break;
    default:
        // do the default thing
}
```

Control Flow - `while` and `do` loops

```
// Loop until condition is false
while (condition) {
    // Do something that might change 'condition'
    // Body won't execute if condition is false at the start
}

// Loop at least once until condition is false
do {
    // Body will always be executed at least once
} while (condition);
// Note, ; is required after a do loop
```

Control Flow - `while` and `do` loops

Example: Print "Hello" *N* times.

```c
int counter = 0;
int N;
// ... set N to something ...

while (counter < N) {
    printf("Hello\n");
    // This is equivalent to
    // counter = counter + 1
    counter++;
}
```

```c
int counter = 0;
int N;
// ... set N to something ...

// A do loop is incorrect here,
// since it would still print
// once if N is 0 or lower
do {
    printf("Hello\n");
    counter = counter + 1;
} while (counter < N);
```

`while` loops are more useful when we don't know how many iterations are needed. When we know the number of iterations, a `for` loop is more convenient (but equivalent).

```c
int N = 10;
for (int counter = 0; counter < N; ++counter) {
    printf("Hello\n");
}
```

Control Flow - `for` loops

Each of the three statements are called at specific times, and all are optional

```
for (/* called once at start */;
     /* called before each loop through */;
     /* called after each loop through */)
{
    /* Body of loop */
}
```

This can be transformed to a `while` loop:

```
{ // scope of equivalent for loop
    /* called once at start */;
    while (/* called before each loop through */) {
        /* Body of loop */
        /* called after each loop through */)
    }
}
```

Control Flow - `for` loops

A `for` loop without a condition statement will loop infinitely

```
for (;;) {
    // Print "Hello" until the program is killed
    printf("Hello\n");
}
```

Equivalent to:

```
while (true) {
    // Print "Hello" until the program is killed
    printf("Hello\n");
}
```

Control Flow - `break` and `continue`

- **`break`**
  - Break out of the current loop or **`case`**
  - Only breaks out of immediately surrounding loop if loops are nested
- **`continue`**
  - Skip the rest of the loop body and begin the next iteration
  - Only applies to surrounding loop if loops are nested

```c
// This only prints once
while (true) {
    printf("Hello\n");
    break;
}
```

```c
// This only prints 5 times
for (int i=0; i < 10; i++) {
    if (i > 4) continue;
    printf("Hello\n");
}
```

Control Flow - labels and goto

goto jumps to a label *anywhere* in the code. This makes it very easy to introduce memory leaks or undefined behaviour. **It should be used only with extreme caution!**

Main use cases are for error handling, and to exit early from a nested loop where **break** statements would be awkward:

```
// This prints "Hello" 500,000 times. NOT 100,000,000 times
for (int i=0; i < 10000; i++) {
    for (int j=0; j < 10000; j++) {
        if (i*j > 500000) goto my_label;
        printf("Hello\n");
    }
}
my_label:
printf("Finished with loop\n");
```

Back to the Fibonacci sequence.

Fibonacci Sequence - Reading Input

```c
int main(int argc, char* argv[]) {
    // Make sure we were given an input argument
    if (argc != 2) {
        printf("ERROR: expected one input argument\n");
        return 1;
    }
    // TODO: convert argv[1] to an integer
    return 0;
}
```

Fibonacci Sequence - Reading Input

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
    // Make sure we were given an input argument
    if (argc != 2) {
        printf("ERROR: expected one input argument\n");
        return 1;
    }
    int fib_max = atoi(argv[1]);          // Read user input
    printf("Input was: %i\n", fib_max); // Print it back out
    return 0;
}
```

For standard library documentation, https://en.cppreference.com/w/c is a good resource.

`atoi` takes a `char*` as input, and outputs an `int`

`printf` format specification can be found here: https://en.cppreference.com/w/c/io/fprintf

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument
$ ./fibonacci 12
Input was: 12
$ ./fibonacci -3
Input was: -3
```

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument
$ ./fibonacci 12
Input was: 12
$ ./fibonacci -3
Input was: -3
$ ./fibonacci 1.2
Input was: 1
```

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument
$ ./fibonacci 12
Input was: 12
$ ./fibonacci -3
Input was: -3
$ ./fibonacci 1.2
Input was: 1
$ ./fibonacci asdf
Input was: 0
```

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument
$ ./fibonacci 12
Input was: 12
$ ./fibonacci -3
Input was: -3
$ ./fibonacci 1.2
Input was: 1
$ ./fibonacci asdf
Input was: 0
```

These last two cases should return an error!
Negative numbers also don't really make sense.

Try to implement this yourself!

Useful information:

You can access individual characters of the input argument using `argv[1][i]`, where `i` is an integer.

The last character in a C string is `'\0'`. Use this to know when the loop should end.

The `isdigit(char)` function from `ctype.h` returns true if the input character is a digit (0-9).

Fibonacci Sequence - Reading Input

```c
// ~snip~ include stdio.h for printf() and stdlib.h atoi()
#include <ctype.h> // for isdigit()
int main(int argc, char* argv[]) {
    if (argc != 2) {  // Make sure we were given an input argument
        printf("ERROR: expected one input argument\n"); return 1;
    }
    // Check for invalid input
    for (int i = 0; argv[1][i] != '\0'; ++i) {
        if (!isdigit(argv[1][i])) {
            printf("ERROR: invalid input!\n"); return 2;
        }
    }
    int fib_max = atoi(argv[1]);         // Read user input
    printf("Input was: %i\n", fib_max); // Print it back out
    return 0;
}
```

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument
```

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument
$ ./fibonacci 12
Input was: 12
```

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument

$ ./fibonacci 12
Input was: 12

$ ./fibonacci -3
ERROR: invalid input!
```

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument

$ ./fibonacci 12
Input was: 12

$ ./fibonacci -3
ERROR: invalid input!

$ ./fibonacci 1.2
ERROR: invalid input!
```

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument

$ ./fibonacci 12
Input was: 12

$ ./fibonacci -3
ERROR: invalid input!

$ ./fibonacci 1.2
ERROR: invalid input!

$ ./fibonacci asdf
ERROR: invalid input!
```

Now it works as expected!

Fibonacci Sequence - Reading Input

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci
ERROR: expected one input argument
$ ./fibonacci 12
Input was: 12
$ ./fibonacci -3
ERROR: invalid input!
$ ./fibonacci 1.2
ERROR: invalid input!
$ ./fibonacci asdf
ERROR: invalid input!
```

Now it works as expected!

The code almost doesn't fit on the slide though. Let's fix that.

Functions

- Functions allow you to separate sections of code that perform a particular task
- They allow for easy code re-use (avoid copy-pasting!)
- They also allow for easy testing
  - Unit tests can be compiled for debugging
  - Call function with various valid and invalid inputs and check for expected result
- We've already used a few functions from the standard library:
  - `printf()`
  - `atoi()`
  - `isdigit()`
- `main()` is also just a function. The C compiler knows that the `main()` function will be the main entry point for the program

Functions

- Functions allow you to separate sections of code that perform a particular task
- They allow for easy code re-use (avoid copy-pasting!)
- They also allow for easy testing
  - Unit tests can be compiled for debugging
  - Call function with various valid and invalid inputs and check for expected result
- We've already used a few functions from the standard library:
  - `printf()`
  - `atoi()`
  - `isdigit()`
- `main()` is also just a function. The C compiler knows that the `main()` function will be the main entry point for the program
- Let's add a `input_valid()` function to handle input validation
  - Take input string as input (`argv[1]`)
  - Return `true` if input is valid, `false` otherwise

Functions

```c
// ~snip~ #includes omitted
#include <stdbool.h>
bool input_valid(const char*);          // Forward declare function
int main(int argc, char* argv[]) {
    if (argc != 2) {  // Make sure we were given an input argument
        printf("ERROR: expected one input argument\n");
        return 1;
    }
    if (!input_valid(argv[1])) {      // Make sure input is valid
        printf("ERROR: invalid input!\n");
        return 2;
    }
    int fib_max = atoi(argv[1]);           // Read user input
    printf("Input was: %i\n", fib_max); // Print it back out
    return 0;
}
```

Functions

```
bool input_valid(const char* input) {
    for (int i = 0; input[i] != '\0'; ++i) {
        if (!isdigit(input[i])) return false;
    }
    return true;
}
```

Note

The `const` keyword specifies that the variable cannot change.

Note this is a pointer to a `const char` (i.e. `(const char)*`). The address pointed to can still change, but that won't affect the data. For constant data *and* address, use `const char*const`.

`const` applies to the left unless it's the leftmost keyword. (i.e. `const int` and `int const` are equivalent).

Pointers as function arguments

- Functions only have access to the local scope.
- Only a single variable can be returned with a `return` statement.
- Large input data can be expensive to copy.

Pointers as function arguments

- Functions only have access to the local scope.
- Only a single variable can be returned with a `return` statement.
- Large input data can be expensive to copy.
- This can all be worked around by using pointers as input arguments.

Pointers as function arguments

```c
void return_two_values(int* val1, double* val2) {
    *val1 = 10;
    *val2 = 4.5;
}

int main(void) {
    int my_int;
    double my_double;
    get_two_values(&my_int, &my_double);
    printf("my_int = %i and my_double = %g\n", my_int, my_double);
    return 0;
}

// Prints:
// my_int = 10 and my_double = 4.5
```

Pointers as function arguments

```c
// Taking pointer to const as promise that data won't be modified
int sum_of_array(const int* array, const int array_length) {
    int sum = 0;
    for (int i = 0; i < array_length; ++i)
        sum += array[i];
    return sum;
}

int main(void) {
    int my_array[] = {5, 3, 4, 0, 1, 10};
    printf("Sum of my_array is: %i\n", sum_of_array(my_array, 6);
    return 0;
}
// Prints:
// Sum of my_array is: 23
```

Pointers as function arguments

```
// Even better, if working with arrays, make obvious
// that we expect array to be initialised to a particular size
int sum_of_array(const int len, const int array[len]) {
    int sum = 0;
    for (int i = 0; i < len; ++i)
        sum += array[i];
    return sum;
}
int main(void) {
    int my_array[] = {5, 3, 4, 0, 1, 10};
    // Can avoid hard-coding length of stack array
    // by using sizeof(). Doesn't work for heap pointers
    printf("Sum of my_array is: %i\n", \
        sum_of_array(sizeof(my_array)/sizeof(int), my_array);
    return 0;
}
```

Pointers as function arguments

```c
// Taking pointer to const as promise that data won't be modified
int sum_of_array(const int len, const int array[len]) {
    int sum = 0;
    for (int i = 0; i < len; ++i)
        sum += array[i];
    return sum;
}

int main(void) {
    int my_int = 12;
    // Can take address of a variable to pass in a pointer to it
    printf("Sum of my_array is: %i\n", sum_of_array(1, &my_int));
    return 0;
}
// Prints:
// Sum of my_array is: 12
```

Pointers as function arguments

Notice that we used `int main(void)` on the previous slide to express that the `main` function doesn't take any arguments.

In C, a function declaration of `int my_func()` just says that `my_func` returns an integer, without providing any information about input arguments, and hence it is valid to call `my_func` with any number of arguments.

To explicitly declare a function as taking zero input arguments, it can be declared as `int my_func(void)`. In this case, it will be a compile error to call `my_func` with anything other than 0 arguments.

Static variables

Static variables inside a function have a lifetime of the program, but live in the local scope.

```c
int count_up() {
  // Static variable persists between function calls
  static int value = 0;
  return ++value;
}

int main(void) {
  printf("%d, ", count_up());
  printf("%d, ", count_up());
  printf("%d\n", count_up());
  return 0;
}
// Prints: 1, 2, 3
// If value wasn't declared as static, this would print: 1, 1, 1
```

Static functions

Functions declared as **`static`** can be interpreted in the same way as global **`static`** variables. That is, they're only visible within the current translation unit.

`file1.c`:

```
void log_value(const int value) {
    printf("Value is: %d\n", value);
}
```

`file2.c`:

```
void log_value(const int value, const int step) {
    printf("Got value = %d on step %d\n", value, step);
}
```

This will fail to link since there are two versions of `log_value`:

```
$ gcc file1.c file2.c
/usr/bin/ld: /tmp/ccbaHZyr.o: in function `log_value':
file.c:(.text+0x0): multiple definition of `log_value';
/tmp/cc2aYUjz.o:file1.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Static functions

Functions declared as `static` can be interpreted in the same way as global `static` variables. That is, they're only visible within the current file.

`file1.c`:

```
static void log_value(const int value) {
    printf("Value is: %d\n", value);
}
```

`file2.c`:

```
static void log_value(const int value, const int step) {
    printf("Got value = %d on step %d\n", value, step);
}
```

This will work, since both versions are local to their own file.
Declaring all but one as `static` would also work.

Functions - Good practices

In general, it is good practice for functions to:

- Take a pointer to input arguments that could be large and don't need to be copied
  - For small data types that fit in registers, copying is better unless using as an extra return variable.
- Take a `const` pointer to input data you promise not to change.
  - This helps users of your function debug their code, and helps the compiler stop you from making mistakes.
- Declare functions as `static` if they're not needed in other files.
  - This avoids potential naming conflicts.
- Use array notation for input arrays where possible to indicate the expected size.
  - `void my_func(const int len, const int array[len]);` is clearer than `void my_func(const int len, const int* array);`, especially if the function takes in multiple arrays, or arrays with multiple dimensions.
  - E.g. matrix-matrix multiplication on the stack:

```
void mat_mul(const int m, const int k, const int n,
    const double A[m][k], const double B[k][n], double C[m][n]);
```

  - **This won't work with raw heap pointers, but there is a workaround!**

Arrays on the heap

```c
// Some function that takes an a x b array as input
void some_function(const int a, const int b, int arr[a][b]);

// ...

// Declare heap_array as a pointer to an m x n array of ints
int m = 100, n = 200;
int (*heap_array)[m][n];

// Allocate space on the heap for an m x n array of ints
heap_array = malloc(sizeof(*heap_array));

// Call the function by passing it the array (on the heap)
// pointed to by heap_array. This works because *heap_array
// has the type int[m][n], not just int**
some_function(m, n, *heap_array);
```

Back to the Fibonacci sequence

Back to the Fibonacci sequence

We have our input. Now we need to do something with it.

```
// ~snip~ #includes omitted
bool input_valid(const char*);          // Forward declare function
int main(int argc, char* argv[]) {
    if (argc != 2) {  // Make sure we were given an input argument
        printf("ERROR: expected one input argument\n");
        return 1;
    }
    if (!input_valid(argv[1])) {      // Make sure input is valid
        printf("ERROR: invalid input!\n");
        return 2;
    }
    int fib_max = atoi(argv[1]);            // Read user input
    printf("Input was: %i\n", fib_max); // Print it back out
    return 0;
}
```

Back to the Fibonacci sequence

```c
// ~snip~ #includes omitted
bool input_valid(const char*);
void print_fibonacci(const int);
int main(int argc, char* argv[]) {
    if (argc != 2) {  // Make sure we were given an input argument
        printf("ERROR: expected one input argument\n");
        return 1;
    }
    if (!input_valid(argv[1])) {    // Make sure input is valid
        printf("ERROR: invalid input!\n");
        return 2;
    }
    int fib_max = atoi(argv[1]);   // Read user input
    print_fibonacci(fib_max);      // Print the fibonacci numbers
    return 0;
}
```

Now we just need a function that prints the Fibonacci numbers

Now we just need a function that prints the Fibonacci numbers

Four main options: `while` loop, `do` loop, `for` loop, recursion

Now we just need a function that prints the Fibonacci numbers

Four main options: `while` loop, `do` loop, `for` loop, recursion

Try to implement this yourself!

Solutions

print_fibonacci() - while loop

```c
void print_fibonacci(const int fib_max) {
    if (fib_max <= 0) {
        printf("\n");
        return;
    }
    int prev = 1, current = 1;
    printf("1");
    while (next <= fib_max) {
        printf(" %i", next);
        int next = prev + current
        prev = current;
        current = next;
    }
    printf("\n");
}
```

```c
void print_fibonacci(const int fib_max) {
    if (fib_max <= 0) {
        printf("\n");
        return;
    }
    int prev = 0, current = 1, next = 1;
    printf("1");
    do {
        printf(" %i", next);
        prev = current;
        current = next;
        next = prev + current
    } while (next <= fib_max);
    printf("\n");
}
```

print_fibonacci() - for loop

```
void print_fibonacci(const int fib_max) {
    if (fib_max <= 0) {
        printf("\n");
        return;
    }
    int prev = 1, current = 1, next = 2;
    printf("1 1");
    for (; next <= fib_max; next = prev + current) {
        printf(" %i", next);
        prev = current;
        current = next;
    }
    printf("\n");
}
```

print_fibonacci() - Recursion

```c
void print_fibonacci(const int fib_max, \
    const int prev, const int current)
{
    if (current > fib_max) {
        printf("\n");
        return;
    } else if (current == 1) printf("1 1");
    else printf("%i", current);

    // Functions can call themselves!
    print_fibonacci(fib_max, current, prev + current);
}
```

print_fibonacci() - Recursion

```c
// ~snip~ #includes omitted
bool input_valid(const char*);
void print_fibonacci(const int, const int, const int);
int main(int argc, char* argv[]) {
    if (argc != 2) {  // Make sure we were given an input argument
        printf("ERROR: expected one input argument\n");
        return 1;
    }
    if (!input_valid(argv[1])) {     // Make sure input is valid
        printf("ERROR: invalid input!\n");
        return 2;
    }
    int fib_max = atoi(argv[1]);  // Read user input
    print_fibonacci(fib_max,1,1); // Print the fibonacci numbers
    return 0;
}
```

Recursion

- Function calls itself with new arguments.
- Can be very useful when working with recursive data structures (eg. binary trees)
- Need to be careful about overflowing the call stack!
  - When a function is called, a new *stack frame* is created to save the current state of the calling code.
  - If function calls are nested too deeply, this can overflow the stack.
  - Can be avoided by using tail recursion if compiler can do tail call optimisation. In this case, the recursive function call is the last operation, so it can replace the current stack frame instead of beginning a new one.

```c
// Need to add the result of the
// function call to `N`, so
// stack frame must be preserved
int sumto(int N) {
    if (N > 1)
        return N + sumto(N-1);
    return N;
}

// Function call is final
// operation, so tail call
// optimisation is possible
int sumto(int N, int cur) {
    if (N == 0) return cur;
    return sumto(N-1,cur+N);
}
```

End of session 1.

Session 2

Yesterday we learned about the basic features of C and used them to implement a simple Fibonacci program.

We concluded by developing four different Fibonacci sequence implementations.

Session 2

Yesterday we learned about the basic features of C and used them to implement a simple Fibonacci program.

We concluded by developing four different Fibonacci sequence implementations.

Let's pick up from there and learn some new features to clean up our code.

The Preprocessor

The Preprocessor

- The preprocessor operates before compilation begins.
- Preprocessor directives begin with a `#`
- `#include` is a preprocessor directive that says "Insert the contents of this file here."
  - We can create our own header files to clean up our code!
- `#define` can be used to define a compile-time constant or macro that gets inserted wherever its symbol appears.
- `#if .. #elif .. #else .. #endif` can be used to conditionally compile sections of code based on constants available from `#define` macros.
- Similarly, `#ifdef .. #elifdef .. #else .. #endif` and `#ifndef .. #elifndef .. #else .. #endif` for conditional compilation based on whether a symbol has been `#defined`
  - `#elifdef` and `#elifndef` are C23 features! For older compilers, use `#elif defined` instead.
- To see exactly what the preprocessor is doing, you can compile with the `-E` flag, and open the resultant output text file (typically a `.i` file by convention).

The Preprocessor

- The preprocessor operates before compilation begins.
- Preprocessor directives begin with a `#`
- `#include` is a preprocessor directive that says "Insert the contents of this file here."
  - We can create our own header files to clean up our code!
- `#define` can be used to define a compile-time constant or macro that gets inserted wherever its symbol appears.
- `#if .. #elif .. #else .. #endif` can be used to conditionally compile sections of code based on constants available from `#define` macros.
- Similarly, `#ifdef .. #elifdef .. #else .. #endif` and `#ifndef .. #elifndef .. #else .. #endif` for conditional compilation based on whether a symbol has been `#defined`
  - `#elifdef` and `#elifndef` are C23 features! For older compilers, use `#elif defined` instead.
- To see exactly what the preprocessor is doing, you can compile with the `-E` flag, and open the resultant output text file (typically a `.i` file by convention).
- **Let's try some of these out in our Fibonacci code.**

The Preprocessor - #include

main.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include "input_valid.h"
#include "fibonacci.h"
int main(int argc, char* argv[]) {
    if (argc != 2) {  // Make sure we were given an input argument
        printf("ERROR: expected one input argument\n"); return 1;
    }
    if (!input_valid(argv[1])) {    // Make sure input is valid
        printf("ERROR: invalid input!\n"); return 2;
    }
    int fib_max = atoi(argv[1]);  // Read user input
    print_fibonacci(fib_max);     // Print the fibonacci numbers
    return 0;
}
```

The Preprocessor - #include

input_valid.h:

```c
#ifndef INPUT_VALID_H
#define INPUT_VALID_H
#include <stdbool.h>
bool input_valid(const char* input);
#endif
```

input_valid.c:

```c
#include "input_valid.h"
#include <ctype.h>
bool input_valid(const char* input) {
    for (int i = 0; input[i] != '\0'; ++i) {
        if (!isdigit(input[i])) return false;
    }
    return true;
}
```

The Preprocessor - `#include`

fibonacci.h:

```c
#ifndef FIBONACCI_H
#define FIBONACCI_H
#include <stdbool.h>
bool print_fibonacci(const int fib_max);
#endif
```

The Preprocessor - #include

fibonacci.c:

```
#include "fibonacci.h"
#include <stdio.h>
void print_fibonacci(const int fib_max) {
    if (fib_max <= 0) { printf("\n"); return; }
    int prev = 1, current = 1, next = 2;
    printf("1 1");
    while (next <= fib_max) {
        printf(" %i", next);
        prev = current;
        current = next;
        next = prev + current
    }
    printf("\n");
}
```

The Preprocessor - `#include`

Try compiling like before:

```
$ gcc main.c -o fib
```

The Preprocessor - `#include`

Try compiling like before:

```
$ gcc main.c -o fib
```

```
/usr/bin/ld: /tmp/ccSudkIr.o: in function `main':
main.c:(.text+0xb6): undefined reference to `valid_input'
/usr/bin/ld: main.c:(.text+0xfd): undefined reference to `print_fibonacci'
collect2: error: ld returned 1 exit status
```

The Preprocessor - #include

Try compiling like before:

```
$ gcc main.c -o fib
```

```
/usr/bin/ld: /tmp/ccSudkIr.o: in function `main':
main.c:(.text+0xb6): undefined reference to `valid_input'
/usr/bin/ld: main.c:(.text+0xfd): undefined reference to `print_fibonacci'
collect2: error: ld returned 1 exit status
```

This is a linker error.

Since `main.c` only includes the header, the compiler doesn't know anything about the implementation, so when it tries to link the function call to a the compiled function, it can't find it.

The Preprocessor - `#include`

Try compiling like before:

```
$ gcc main.c -o fib
```

```
/usr/bin/ld: /tmp/ccSudkIr.o: in function `main':
main.c:(.text+0xb6): undefined reference to `valid_input'
/usr/bin/ld: main.c:(.text+0xfd): undefined reference to `print_fibonacci'
collect2: error: ld returned 1 exit status
```

This is a linker error.

Since `main.c` only includes the header, the compiler doesn't know anything about the implementation, so when it tries to link the function call to a the compiled function, it can't find it.

This is easily fixed by telling the compiler about the implementation:

```
$ gcc main.c valid_input.c fibonacci.c -o fib
$ ./fib 10
1 1 2 3 5 8
$
```

Multi-Step Compilation

- Files can also be compiled into an object file (`.o` file) individually by calling the compiler with the `-c` flag.
  - `-c` tells the compiler to compile without linking.
  - For a single input `.c` file, the default output is a file with the same name, with a `.o` extension.
  - `.o` files can be listed as input files in a future compile step the same way as `.c` files.
- This can allow small parts of a large project to be re-compiled without having to compile the whole thing from scratch. Only the changed code and code that depends on it needs to be recompiled.
  - For example, if we updated `fibonacci.c`, then `fibonacci.o` would need to be recompiled, as would the final executable, `fib`, but `valid_input.o` can remain unchanged.

```
$ gcc fibonacci.c -c
$ gcc valid_input.c -c
$ gcc main.c fibonacci.o valid_input.o -o fib
$ ./fib 10
1 1 2 3 5 8
$
```

But what if we want to choose between different Fibonacci implementations?

Fibonacci Sequence - Choice of Implementation

One option is to define four different versions of the function and just call the particular one we want:

fibonacci.h:

```c
#ifndef FIBONACCI_H
#define FIBONACCI_H
void print_fibonacci_while(const int fib_max);
void print_fibonacci_do(const int fib_max);
void print_fibonacci_for(const int fib_max);
void print_fibonacci_recursive(
    const int fib_max,
    const int prev,
    const int current
);
#endif
```

Fibonacci Sequence - Choice of Implementation

This would work, but what if we want to choose an implementation at compile time without re-writing the code?

Fibonacci Sequence - Choice of Implementation

This would work, but what if we want to choose an implementation at compile time without re-writing the code? We can do this with `#if` inside our `main()` function.

`main.c`:

```
    /* ... set fib_max from user input */
#if FIB_IMPL==0
    print_fibonacci_while(fib_max);
#elif FIB_IMPL==1
    print_fibonacci_do(fib_max);
#elif FIB_IMPL==2
    print_fibonacci_for(fib_max);
#elif FIB_IMPL==3
    print_fibonacci_recursive(fib_max,1,1);
#else
#error Unknown Fibonacci implementation
#endif
    /* ~snip~ */
```

Fibonacci Sequence - Choice of Implementation

Now if we compile as before, we get our error message:

```
$ gcc main.c valid_input.c fibonacci.c -o fib
main.c: In function 'main':
main.c:20:2: error: #error "Unknown Fibonacci implementation"
   20 | #error Unknown Fibonacci implementation
      |  ^~~~~
```

Fibonacci Sequence - Choice of Implementation

Now if we compile as before, we get our error message:

```
$ gcc main.c valid_input.c fibonacci.c -o fib
main.c: In function 'main':
main.c:20:2: error: #error "Unknown Fibonacci implementation"
   20 | #error Unknown Fibonacci implementation
      |  ^~~~~
```

We need to provide a choice:

```
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=1
$ ./fib 10
1 1 2 3 5 8
$
```

Fibonacci Sequence - Choice of Implementation

Now if we compile as before, we get our error message:

```
$ gcc main.c valid_input.c fibonacci.c -o fib
main.c: In function 'main':
main.c:20:2: error: #error "Unknown Fibonacci implementation"
   20 | #error Unknown Fibonacci implementation
      |  ^~~~~
```

We need to provide a choice:

```
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=1
$ ./fib 10
1 1 2 3 5 8
$
```

This chooses the `do` implementation, but it's not very clear!

Fibonacci Sequence - Choice of Implementation

One option is to use `#define` to label our magic numbers. This is like declaring a global **const**, but its value is available to the preprocessor. Downside is that it has no type information.

`main.c`:

```c
#define IMPL_WHILE 0
#define IMPL_DO 1
#define IMPL_FOR 2
#define IMPL_RECURSIVE 3

#include <stdio.h>
#include <stdlib.h>
#include "valid_input.h"
#include "fibonacci.h"
int main(int argc,
     char* argv[]) {
/* ~snip~ */
```

```c
#if FIB_IMPL==IMPL_WHILE
    print_fibonacci_while(fib_max);
#elif FIB_IMPL==IMPL_DO
    print_fibonacci_do(fib_max);
#elif FIB_IMPL==IMPL_FOR
    print_fibonacci_for(fib_max);
#elif FIB_IMPL==IMPL_RECURSIVE
    print_fibonacci_recursive(fib_max,1,1);
#else
#error "Unknown Fibonacci implementation"
#endif
    /* ~snip~ */
```

Fibonacci Sequence - Choice of Implementation

Now we can compile with

```
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=IMPL_DO
$ ./fib 10
1 1 2 3 5 8
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=IMPL_WHILE
$ ./fib 15
1 1 2 3 5 8 13
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=SOMETHING
main.c: In function 'main':
main.c:20:2: error: #error "Unknown Fibonacci implementation"
   20 | #error Unknown Fibonacci implementation
      |  ^~~~~
```

Fibonacci Sequence - Choice of Implementation

Now we can compile with

```
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=IMPL_DO
$ ./fib 10
1 1 2 3 5 8
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=IMPL_WHILE
$ ./fib 15
1 1 2 3 5 8 13
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=SOMETHING
main.c: In function 'main':
main.c:20:2: error: #error "Unknown Fibonacci implementation"
   20 | #error Unknown Fibonacci implementation
      |  ^~~~~
```

This has a flaw though, which is that if `SOMETHING` were `#defined` elsewhere in the code, and it happened to evaluate to 0, 1, 2, or 3, then we'd choose that implementation instead of hitting the error!

Fibonacci Sequence - Choice of Implementation

Another option is to use `#ifdef` instead so that there aren't any magic numbers.

`main.c`:

```
    /* ~snip~ */
#ifdef FIB_IMPL_WHILE
    print_fibonacci_while(fib_max);
#elifdef FIB_IMPL_DO
    print_fibonacci_do(fib_max);
#elifdef FIB_IMPL_FOR
    print_fibonacci_for(fib_max);
#elifdef FIB_IMPL_RECURSIVE
    print_fibonacci_recursive(fib_max,1,1);
#else
#error "Unknown Fibonacci implementation"
#endif
    /* ~snip~ */
```

Fibonacci Sequence - Choice of Implementation

In this case, we compile with:

```
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL_DO
$ ./fib 10
1 1 2 3 5 8
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL_WHILE
$ ./fib 15
1 1 2 3 5 8 13
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL_SOMETHING
main.c: In function ‘main’:
main.c:20:2: error: #error "Unknown Fibonacci implementation"
   20 | #error Unknown Fibonacci implementation
      |  ^~~~~
```

Fibonacci Sequence - Choice of Implementation

This makes our `main()` function quite messy though. Let's add a wrapper function.

`fibonacci.h`:

```
#ifndef FIBONACCI_H
#define FIBONACCI_H
void print_fibonacci(
    const int fib_max);
#endif
```

`main.c`:

```
/* ~snip~ */
print_fibonacci(fib_max);
/* ~snip~ */
```

`fibonacci.c`:

```
#include "fibonacci.h"
/* ~snip~ various implementations ~snip! */
void print_fibonacci(const int fib_max) {
#ifdef FIB_IMPL_WHILE
    print_fibonacci_while(fib_max);
#elifdef FIB_IMPL_DO
    print_fibonacci_do(fib_max);
#elifdef FIB_IMPL_FOR
    print_fibonacci_for(fib_max);
#elifdef FIB_IMPL_RECURSIVE
    print_fibonacci_recursive(fib_max,1,1);
#else
#error Unknown Fibonacci implementation
#endif
```

Fibonacci Sequence - Choice of Implementation

We could also do this with a macro instead of a wrapper function. `fibonacci.h`:

```
#ifndef FIBONACCI_H
#define FIBONACCI_H
#ifdef FIB_IMPL_WHILE
#define PRINT_FIBONACCI(FIB_MAX) print_fibonacci_while(FIB_MAX)
#elifdef FIB_IMPL_DO
#define PRINT_FIBONACCI(FIB_MAX) print_fibonacci_do(FIB_MAX)
#elifdef FIB_IMPL_FOR
#define PRINT_FIBONACCI(FIB_MAX) print_fibonacci_for(FIB_MAX)
#elifdef FIB_IMPL_RECURSIVE
#define PRINT_FIBONACCI(FIB_MAX) \
    print_fibonacci_recursive(FIB_MAX,1,1)
#else
#error "Unknown Fibonacci implementation"
#endif
// contd...
```

Fibonacci Sequence - Choice of Implementation

```c
// contd...

void print_fibonacci_while(const int fib_max);
void print_fibonacci_do(const int fib_max);
void print_fibonacci_for(const int fib_max);
void print_fibonacci_recursive(const int fib_max,
    const int prev, const int current);
#endif
```

main.c:

```c
/* ~snip~ */
PRINT_FIBONACCI(fib_max);
/* ~snip~ */
```

Based on which implementation is defined at compile time, the correct function call is put in place of `PRINT_FIBONACCI`, and `fib_max` is passed through in place of `FIB_MAX`.

Fibonacci Sequence - Choice of Implementation

```c
// contd...

void print_fibonacci_while(const int fib_max);
void print_fibonacci_do(const int fib_max);
void print_fibonacci_for(const int fib_max);
void print_fibonacci_recursive(const int fib_max,
    const int prev, const int current);
#endif
```

`main.c`:

```c
/* ~snip~ */
PRINT_FIBONACCI(fib_max);
/* ~snip~ */
```

Based on which implementation is defined at compile time, the correct function call is put in place of `PRINT_FIBONACCI`, and `fib_max` is passed through in place of `FIB_MAX`.

Fibonacci Sequence - Choice of Implementation

There's one other trick we could use that gives an even cleaner solution.

```
// The ## operator concatenates two symbols into one, so it
// will fill IMPL with whatever is passed in, and then join it to
// 'print_fibonacci_'.
// But calling PRINT_FIB_HELPER(FIB_IMPL) directly would just
// give us 'print_fibonacci_FIB_IMPL', since that's what IMPL
// would expand to. Instead, we call it from another macro so
// that FIB_IMPL gets expanded first (e.g. to 'while', 'for', etc.)
#define PRINT_FIB_HELPER(IMPL) print_fibonacci_ ## IMPL
#define PRINT_FIB(IMPL) PRINT_FIB_HELPER(IMPL)

void print_fibonacci(const int fib_max) {
    PRINT_FIB(FIB_IMPL)(fib_max);
}
```

Then compile with (for example):

```
$ gcc main.c valid_input.c fibonacci.c -o fib -DFIB_IMPL=while
```

What if we want to choose the implementation at runtime?

What if we want to choose the implementation at runtime?

A short diversion into custom datatypes

User-Defined Datatypes

We can define our own datatypes based on the primitive types.

```
// This defines size_t as an alias for unsigned long
typedef unsigned long u64;

// Now we can use size_t as a more convenient name
u64 my_unsigned_long = 0x100000000;

// This can be used to convey meaning, and provide
// single points of change.
typedef double Meters;
typedef double Seconds;
```

User-Defined Datatypes - Enumerations

Enumerations can be used to represent different states.

```c
// Two ways to declare. Tagged:
enum Colours {
    RED,
    GREEN,
    BLUE
};  // <- NOTE semicolon is required!
enum Colours my_colour = RED;

// Untagged (typedef colours to alias an anonymous enum)
typedef enum {RED, GREEN, BLUE} Colours;
Colours my_colour = BLUE;

// Can combine both to allow both declaration types:
typedef enum Colours {RED, GREEN, BLUE} Colours;
```

User-Defined Datatypes - Enumerations

Enumerations can be used to represent different states.

```c
// Two ways to declare. Tagged:
enum Colours {
    RED,
    GREEN,
    BLUE
};  // <- NOTE semicolon is required!
enum Colours my_colour = RED;

// Untagged (typedef Colours to alias an anonymous enum)
typedef enum {RED, GREEN, BLUE} Colours;
Colours my_colour = BLUE;

// Can combine both to allow both declaration types:
typedef enum Colours {RED, GREEN, BLUE} Colours;
```

User-Defined Datatypes - Enumerations

Enumerations evaluate to integers by default.

```
enum Colours {RED, GREEN, BLUE};
enum Colours my_colour = RED;
int red_value = my_colour;   // Stores 0
int blue_value = BLUE;       // Stores 2

// Custom values can be assigned
enum Colours {RED = 4, GREEN = 2, BLUE = 123};

// Can be useful for bit masks:
enum Options {OPTA = 1 << 0, OPTB = 1 << 1, OPTC = 1 << 2};
int bitmask = OPTA | OPTC; // Stores 0b101
if (bitmask & OPTA) { /* Do something if OPTA bit is set */ }
bitmask |= OPTB;     // Set the OPTB bit without altering others
bitmask &= ~OPTC;    // Unset the OPTC bit. bitmask == 0b011
```

User-Defined Datatypes - Structures

Multiple values can be stored together in a structure:

```c
struct SomeData {
     int a;
     double b;
     float c;
};
// As for enums, must use struct tag, or declare with typedef
struct SomeData my_data1 = {1, 1.0, 2.0f};
// The below is equivalent, but much clearer. Prefer this notation.
struct SomeData my_data2 = {.a = 1, .b = 1.0, .c = 2.0f};

// Use . to access members
my_data1.a = 10;      // Store data in my_data1's a value
// Use -> in place of . to access members through a pointer
struct SomeData* data_ptr = &my_data2;  // Get pointer to my_data2
data_ptr->b = 3.14; // Store 3.14 in my_data2's b
```

User-Defined Datatypes - Unions

Unions allow multiple datatypes to be stored in a single memory location.
Only one member is valid at a time! Use with care!

```
union int_or_float {
    int i;
    float f;
};
// Initialise as for struct, but only set one member!
union int_or_float my_int_or_float = {.i = 10};
// Store 1.0f. The integer value is overridden!
my_int_or_float.f = 1.0;

// This *interprets* the bits of the float as an int!
int some_int = my_int_or_float.i;
// Be careful not to access through the wrong member!
```

Fibonacci Sequence - Choice of Implementation

With more knowledge about datatypes, now we can switch between Fibonacci implementations at run time using function pointers. `fibonacci.h`:

```c
typedef enum Algorithm {WHILE, DO, FOR, RECURSIVE} Algorithm;
// Declare print_algorithm as a type alias for any function
// that takes a const int as input and returns void
typedef void print_algorithm(const int);

// Function to choose algorithm from user input
Algorithm choose_algorithm(int argc, const char** argv);
// Function to get the required function pointer
print_algorithm get_algorithm(Algorithm);
```

Fibonacci Sequence - Choice of Implementation

With more knowledge about datatypes, now we can switch between Fibonacci implementations at run time using function pointers. `fibonacci.h`:

```
typedef enum Algorithm {WHILE, DO, FOR, RECURSIVE} Algorithm;
// Declare print_algorithm as a type alias for any function
// that takes a const int as input and returns void
typedef void print_algorithm(const int);

// Function to choose algorithm from user input
Algorithm choose_algorithm(int argc, const char** argv);
// Function to get the required function pointer
print_algorithm get_algorithm(Algorithm);
```

Note, another option to declare a function pointer without a `typedef` is:

```
// declare my_fn_pointer with same signature as above
void (*my_fn_pointer)(const int);
my_fn_pointer = print_while;    // Point to print_while function
```

Fibonacci Sequence - Choice of Implementation

fibonacci.cpp:

```
// Need a print_recursive with same signature as others
void print_recursive_wrap(const int fib_max) {
    print_recursive(fib_max, 1, 1); }
print_algorithm get_algorithm(Algorithm algo) {
    switch (algo) {
      case WHILE:
        return print_while; // Point to the print_while function
      case DO:
        return print_do;
      case FOR:
        return print_for;
      case RECURSIVE:
        return print_recursive_wrap;
    }
    fprintf(stderr, "ERROR: unknown algorithm!\n"); exit(255); }
```

Try implementing runtime algorithm selection yourself

(with or without function pointers)

Getting back on track, compilation is becoming more complicated with the extra files.

Getting back on track, compilation is becoming more complicated with the extra files.

How can we manage this for large projects?

Build Systems

- Build systems provide an abstraction over compilation
- They can be used to manage (and validate) compile options
- They can also handle complex dependency trees
  - E.g. code depends on a separately compiled library of other code in the project
  - Can have a compile target for each, with one dependent on the other
  - Compilation steps automatically re-run if required when build system is run
- There are a few to choose from, but the most common are `make` and `cmake`
- These both have a lot of complexity, so we'll just cover the basics here

Build Systems - make

Makefile

```
cflags := -Wall -Wextra -Wpedantic -Werror -O3
CC := gcc
sources := $(wildcard *.c)
all: fib_while fib_do fib_for fib_recursive

fib_while: ${sources}
	${CC} ${cflags} $^ -o $@ -DFIB_IMPL_WHILE

fib_do: ${sources}
	${CC} ${cflags} $^ -o $@ -DFIB_IMPL_DO

fib_for: ${sources}
	${CC} ${cflags} $^ -o $@ -DFIB_IMPL_FOR

fib_recursive: ${sources}
	${CC} ${cflags} $^ -o $@ -DFIB_IMPL_RECURSIVE
```

Now we can build the various versions with:

```
$ make fib_while -j4 # -j flag for number of parallel build tasks
$ ./fib_while 10
1 1 2 3 5 8
$ make # running this will build all four versions
```

There is a lot more that can be done with `make` for more advanced configuration. For example, individual build steps can have their own target that produces a `.o` file, and then `make` can automatically detect which `.o` files (and subsequent steps that depend on them) need to be recompiled based on file modification times.

See https://makefiletutorial.com/ for a more detailed C-oriented guide, or for a more in-depth but general guide.

Build Systems - `cmake`

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(Fibonacci C)

set(SRCS main.c valid_input.c fibonacci.c)
add_executable(fib_while ${SRCS})
add_executable(fib_do ${SRCS})
add_executable(fib_for ${SRCS})
add_executable(fib_recursive ${SRCS})

# Apply compiler flags
foreach(EXE IN ITEMS fib_while fib_do fib_for fib_recursive)
  target_compile_options(${EXE} PRIVATE -Wall -Wextra
      -Wpedantic -Werror)
endforeach()
# contd...
```

Build Systems - cmake

```
# contd...
target_compile_definitions(fib_while PRIVATE FIB_IMPL_WHILE)
target_compile_definitions(fib_do PRIVATE FIB_IMPL_DO)
target_compile_definitions(fib_for PRIVATE FIB_IMPL_FOR)
target_compile_definitions(fib_recursive PRIVATE FIB_IMPL_RECURSIVE)
```

Can configure the project with:

```
$ cmake -B build -DCMAKE_BUILD_TYPE=Release
```

This says "Build in a directory called 'build', and use the 'Release' configuration (defaults to `-O3 -DNDEBUG`)"

Can then build all targets with

```
$ cmake --build build -j4 # Build all targets. Accepts -j like make
$ cmake --build build --target fib_while  # Just fib_while
$ build/fib_while 10
1 1 2 3 5 8
```

Build Systems - `cmake`

`cmake` is a little harder to get started with than `make`, but has the benefit of being cross-platform (hence the `c`), and tends to be better at handling complex projects.

For a more detailed `cmake` tutorial, see the documentation, https://cmake.org/cmake/help/latest/guide/tutorial/index.html, or other tutorials such as https://riptutorial.com/cmake.

Build Systems - `cmake`

`cmake` is a little harder to get started with than `make`, but has the benefit of being cross-platform (hence the `c`), and tends to be better at handling complex projects.

For a more detailed `cmake` tutorial, see the documentation, https://cmake.org/cmake/help/latest/guide/tutorial/index.html, or other tutorials such as https://riptutorial.com/cmake.

If you're using Linux or MacOS, then compilation with `gcc`, `make`, and `cmake` should work exactly the same way on your own computer. For Windows users, you can either use the Windows Subsystem for Linux to compile/test in a Linux environment, or install Microsoft Visual Studio to compile natively with the `msvc` compiler (check the documentation for details).

Some performance considerations

As you begin writing bigger projects, performance will inevitably become a factor to consider.

Some performance considerations

As you begin writing bigger projects, performance will inevitably become a factor to consider.

Some general good practices that will help:

- Avoid deeply nested loops (i.e. loops inside loops inside loops etc.)
  - If you find yourself writing a `for` loop over a large range inside another loop that also goes over a large range, think about whether you can achieve your goal in a more efficient way.

Some performance considerations

As you begin writing bigger projects, performance will inevitably become a factor to consider.

Some general good practices that will help:

- Avoid deeply nested loops (i.e. loops inside loops inside loops etc.)
  - If you find yourself writing a `for` loop over a large range inside another loop that also goes over a large range, think about whether you can achieve your goal in a more efficient way.
- Flatten multi-dimensional arrays.
  - Consider a 2D array, e.g.
    `double** mat_2D = malloc(sizeof(double*)*nrows);`
  - Instead of calling `mat_2D[i] = malloc(sizeof(double)*ncols)` for each row, consider using `mat_2D[0] = malloc(sizeof(double)*nrows*ncols);`, and then just assigning each row pointer as `mat_2D[i] = mat_2D[i-1] + ncols;`, or equivalently, `mat_2D[i] = &mat_2D[0][i*ncols];`
  - This means the whole array is stored in one block of memory, which takes less time to allocate and is better for CPU caching when iterating over it.
  - Using a 1D array with indices calculated as `[r*ncols+c]` has similar benefits.

Some performance considerations

- Prefer a struct of arrays to an array of structs.
  - When iterating over an array of structs, often you only need one or two of the members.
  - If the struct is large, there will often be a lot of unused data being fetched, so computation may become limited by memory bandwidth.
  - By using a struct of arrays instead, you can iterate over the arrays of the members you care about without loading unnecessary data.

Some performance considerations

- Prefer a struct of arrays to an array of structs.
  - When iterating over an array of structs, often you only need one or two of the members.
  - If the struct is large, there will often be a lot of unused data being fetched, so computation may become limited by memory bandwidth.
  - By using a struct of arrays instead, you can iterate over the arrays of the members you care about without loading unnecessary data.
- Use a profiler to find optimisation targets.
  - Profilers like `gprof` can help find which parts of your code take the longest to run.
  - Similarly, for debugging, tools like `valgrind` are useful for finding memory bugs (e.g. leaks, out of range accesses, etc.).

Some performance considerations

- **Don't try too hard to pre-optimise.**
  - Start by writing good, readable code that is easy to debug and produces correct results.
  - Once it's working, use a profiler to find the best targets for optimisation.
  - In many cases, code that is easy for a human to read is also easy for a compiler to understand (and optimise for you).

Some performance considerations

- **Don't try too hard to pre-optimise.**
  - Start by writing good, readable code that is easy to debug and produces correct results.
  - Once it's working, use a profiler to find the best targets for optimisation.
  - In many cases, code that is easy for a human to read is also easy for a compiler to understand (and optimise for you).
- **Don't reinvent the wheel.**
  - There are many existing libraries of code that are very fast. For example, BLAS and LAPACK for common matrix operations, and FFTW for fast Fourier transforms.
  - Compiled libraries can be linked with the `-l` compiler flag. For example `-lblas` will look for `libblas.a` (among other potential filenames) and link to it.
  - Can add extra directories to look for libraries in with the `-L` flag.

Practice makes perfect

You now have all the tools you need to begin your own C projects.

Practice makes perfect

You now have all the tools you need to begin your own C projects.

There's a lot more that can be learned about C, but that's best discovered as you need it.

- For some example problems to solve, try https://projecteuler.net/archives

Practice makes perfect

You now have all the tools you need to begin your own C projects.

There's a lot more that can be learned about C, but that's best discovered as you need it.

- For some example problems to solve, try https://projecteuler.net/archives
- To learn more about the C's capabilities and standard library features, have a look at https://en.cppreference.com/w/c/. For example:
  - Handling of complex numbers: https://en.cppreference.com/w/c/numeric/complex
  - Memory management functions: https://en.cppreference.com/w/c/memory
  - Reading terminal input: https://en.cppreference.com/w/c/io/fscanf
- Particularly for large projects, becoming familiar with source control (e.g. Git) for change tracking is essential. You can learn more about it here:
  - Online Git tutorial: https://www.tutorialspoint.com/git/index.htm
  - MIT Git lecture: https://www.youtube.com/watch?v=2sjqTHE0zok

Practice makes perfect

You now have all the tools you need to begin your own C projects.

There's a lot more that can be learned about C, but that's best discovered as you need it.

- For some example problems to solve, try https://projecteuler.net/archives
- To learn more about the C's capabilities and standard library features, have a look at https://en.cppreference.com/w/c/. For example:
  - Handling of complex numbers: https://en.cppreference.com/w/c/numeric/complex
  - Memory management functions: https://en.cppreference.com/w/c/memory
  - Reading terminal input: https://en.cppreference.com/w/c/io/fscanf
- Particularly for large projects, becoming familiar with source control (e.g. Git) for change tracking is essential. You can learn more about it here:
  - Online Git tutorial: https://www.tutorialspoint.com/git/index.htm
  - MIT Git lecture: https://www.youtube.com/watch?v=2sjqTHE0zok
- A common method for debugging is to print out the value of key variables with `printf()`. There are much more powerful options though, such as `gdb`: http://www.gdbtutorial.com/
  - Once you've learned the basics of `gdb`, see http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf for a handy reference.

One last practice problem

Practice Problem

Using `scanf()` to read move coordinates, write a tic-tac-toe game:

```
$ ./tictactoe        Player 2: 1 2        Player 2: 2 1
   |   |                | O |              X | O |
---+---+---          ---+---+---          ---+---+---
   |   |                | X |              O | X |
---+---+---          ---+---+---          ---+---+---
   |   |                |   |                |   |

Player 1: 2 2        Player 1: 1 1        Player 1: 3 3
   |   |              X | O |              X | O |
---+---+---          ---+---+---          ---+---+---
   | X |                | X |              O | X |
---+---+---          ---+---+---          ---+---+---
   |   |                |   |                |   | X
                                          Player 1 wins!
```