Programming in C
Monash/NCI Training Week

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, 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++

Why C?

C is the language of HPCs.

- It's used as the interface for most major high performance computing libraries.
  - Linear algebra
  - Fourier transforms
  - Parallel computing
  - GPU acceleration
- It does the heavy lifting behind many Python libraries (and other languages).
- It gives a close representation of hardware-level details necessary for optimisation.

Even if you never write your own C code, the understanding it gives of the underlying hardware can help guide software design and optimisation in other languages.

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. library) 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

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!

List

Goal

Build a basic library of code for working with variable length lists.

In Python, we can do this:

```
a = []
for v in range(6):
    a.append(v)
a.insert(3, 2.5)
a.pop(1)
print(a)
# Prints:
# [0, 2, 2.5, 3, 4, 5]
```

In C, we'd like to be able to do similar:

```
List a = list_init();
for (int v = 0; v < 6; ++v)
    list_append(&a, v);
list_insert(&a, 3, 2.5);
list_pop(&a, 1);
list_print_contents(a);
// Prints:
// [0, 2, 2.5, 3, 4, 5]
```

List - First step

To create an API like this, first we need to understand variables and types

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

```
int my_int = -12345;    // int typically uses 32 bits (4 bytes)
short my_small_int = 32767;      // typically uses 16 bits
long my_large_int = 9999999999999;  // typically 64 bits
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 = 3.14159265358; // up to 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

Data type sizes are implementation-defined! The standard only specifies a minimum size!

For example, `long` only has to be at least 32 bits.

Similarly, `long double` could be as small as 80 bits (with padding to 96 or 128), or it could even be the same size as `double`!

Variables - Examples

Data type sizes are implementation-defined! The standard only specifies a minimum size!

For example, **`long`** only has to be at least 32 bits.

Similarly, **`long double`** could be as small as 80 bits (with padding to 96 or 128), or it could even be the same size as **`double`**!

To get exact size integer types, `stdint.h` is provided:

```
#include <stdint.h>
int32_t my_int = -12345;  // Always 32 bits
uint16_t my_unsigned_short = 32767; // Unsigned 16bit int
```

The full list is available here: https://en.cppreference.com/w/c/types/integer

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 my_int = f;         // information is lost here! my_int = 1

// 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
                                // to use float division.
```

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.:

```c
// 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 lead to unintended bugs, so be careful!
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 */
}
```

Quiz 1

1. How many bits of space are required to store a variable of type `double`?
   a. 8
   b. 16
   c. 32
   d. 64

Quiz 1

1. How many bits of space are required to store a variable of type `double`?
   a. 8
   b. 16
   c. 32
   d. 64 ←

Quiz 1

1. How many bits of space are required to store a variable of type `double`?
   a. 8
   b. 16
   c. 32
   d. 64 ←
2. What is the smallest value that could be stored in a variable of type `unsigned short`?
   a. 0
   b. -128
   c. -32768
   d. -2147483648

Quiz 1

1. How many bits of space are required to store a variable of type `double`?
   a. 8
   b. 16
   c. 32
   d. 64 ←
2. What is the smallest value that could be stored in a variable of type `unsigned short`?
   a. 0 ←
   b. -128
   c. -32768
   d. -2147483648

Operators

- Operators generally fall under one of 3 categories:
  - Mathematical
  - Boolean
  - Bit-wise

Operators

- Operators generally fall under one of 3 categories:
  - Mathematical
  - Boolean
  - Bit-wise

Mathematical

```
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

- Operators 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

- Operators 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

- Operators 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 might store -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.
```

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)

// true evaluates to 1, false evaluates to 0
bool my_bool = true;
int my_int = my_bool; // Store 1 in my_int
```

Quiz 2

1 What value does `a` hold after the code below?

```
float a = 4 / 3;
```

a 1
b 33.33333...
c 33
d 1.33333...

Quiz 2

1 What value does `a` hold after the code below?

```
float a = 4 / 3;
```

a 1 ←
b 33.33333...
c 33
d 1.33333...

Quiz 2

1. What value does `a` hold after the code below?

```
float a = 4 / 3;
```

   a. 1 ←
   b. 33.33333...
   c. 33
   d. 1.33333...

2. What value does `a` hold after the code below?

```
float a = (0b100 & 0xFF) / 3.;
```

   a. 1
   b. 33.33333...
   c. 33
   d. 1.33333...
   e. 0

Quiz 2

1. What value does `a` hold after the code below?

```
float a = 4 / 3;
```

   a. 1 ←
   b. 33.33333...
   c. 33
   d. 1.33333...

2. What value does `a` hold after the code below?

```
float a = (0b100 & 0xFF) / 3.;
```

   a. 1
   b. 33.33333...
   c. 33
   d. 1.33333... ←
   e. 0

Quiz 2

3 What value does `a` hold after the code below?

```
int b = 10;
int c = 10;
bool a = b < c++;
```

a `true`
b `false`

Quiz 2

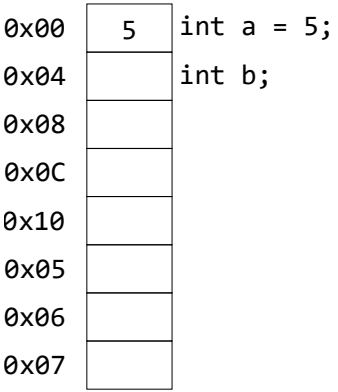
3 What value does `a` hold after the code below?

```
int b = 10;
int c = 10;
bool a = b < c++;
```

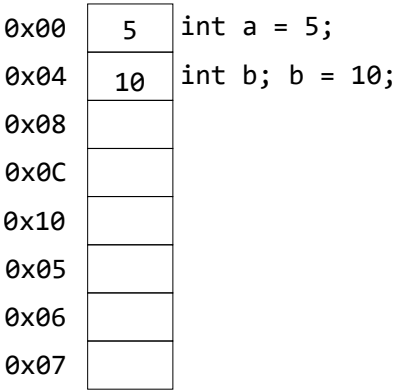
a `true`
b `false` ←

Variables - Pointers & References

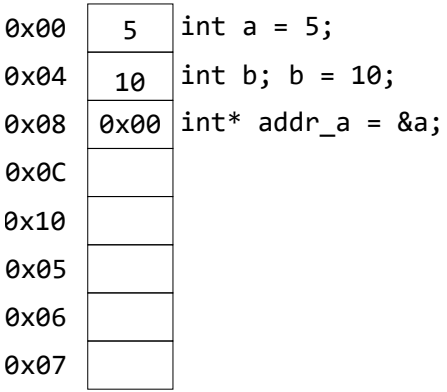
- Variables are just named chunks of memory
- Each chunk has an *address*
- References give the address of the memory storing a variable
- Pointers can be used to store the address
- A pointer can be *dereferenced* to access the memory it's pointing to.



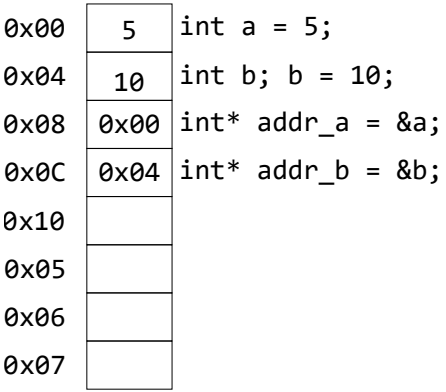
Variables - Pointers & References



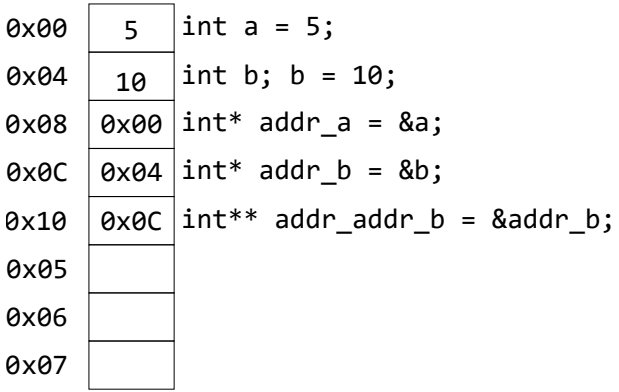
Variables - Pointers & References



Variables - Pointers & References



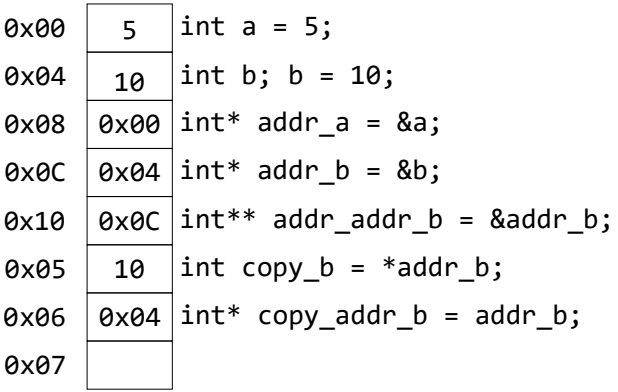
Variables - Pointers & References



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	10	`int copy_b = *addr_b;`
0x06		
0x07		

Variables - Pointers & References



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 $\star$ 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 - Pointers & References

Accessing memory through a pointer of the wrong type is undefined behaviour!

```
float f = 1.0;
int* i_ptr = (int*)&f;      // This creates a pointer of type int*
                            //   which points to data of type float!

int i_from_ptr = *i_ptr;    // Undefined behaviour!
                            // Could store 1065353216 in i_from_ptr,
                            // but not guaranteed!
```

Variables - Arrays

```
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]
// Example:
int a = 10, b = 5;
int* abab[4] = {&a, &b, &a, &b};
*(abab[0]) = 2;     // Now a == 2, and *(abab[2]) == 2

// 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]
// Example:
int abcd[4] = {1, 2, 3, 4};
int (*ptr_to_abcd)[4] = &abcd;
```

Variables - Strings

```
// 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**

// Alternatively, without specifying length of longest string
char* my_strings[] = {"Hello", "world", "something"};
```

Variables - Stack vs Heap

- Variables are 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!

```
// 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 - Stack vs Heap

- Variables are 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.

```
// 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];
```

Quiz 2

1 If `ptr` is a pointer to an integer (declared as `int* ptr;`), what is the type of `&ptr`?

a `int`
b `int*`
c `int**`
d `int*&`

Quiz 2

1 If `ptr` is a pointer to an integer (declared as `int* ptr;`), what is the type of `&ptr`?

a `int`
b `int*`
c `int**` ←
d `int*&`

Quiz 2

1. If `ptr` is a pointer to an integer (declared as `int* ptr;`), what is the type of `&ptr`?
   a. `int`
   b. `int*`
   c. `int**` ←
   d. `int*&`

2. After the code below, what is the value of `a`?

```
int a = 4;
int b = 10;
int* ptr = &a;
*ptr = 1;
ptr = &b;
a = *ptr + a;
```

a. 1
b. 4
c. 10
d. 11
e. 14

Quiz 2

1 If `ptr` is a pointer to an integer (declared as `int* ptr;`), what is the type of `&ptr`?

a `int`
b `int*`
c `int**` ←
d `int*&`

2 After the code below, what is the value of `a`?

```
int a = 4;
int b = 10;
int* ptr = &a;
*ptr = 1;
ptr = &b;
a = *ptr + a;
```

a 1
b 4
c 10
d 11 ←
e 14

Quiz 2

3 In the line below, where is the memory pointed to by `arr` stored?

```
int* arr = malloc(sizeof(int) * 50000);
```

a The stack.
b The heap.
c This line will cause a compile error.

Quiz 2

3 In the line below, where is the memory pointed to by `arr` stored?

```
int* arr = malloc(sizeof(int) * 50000);
```

a The stack.
b The heap. ←
c This line will cause a compile error.

We can use pointers as backing storage for our `List` type

We can use pointers as backing storage for our `List` type

How can we create the type itself?

User-Defined Datatypes

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

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

// Now we can use u64 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.

```
// 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.

```
// 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:

```
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_double {
    int i;
    double f;
};
// sizeof(union int_or_double) == sizeof(double) since double is the large

// Initialise as for struct, but only set one member!
union int_or_double my_int_or_double = {.i = 10};
// Store 1.0 (a double). The integer value is overridden!
my_int_or_float.f = 1.0;

// This *interprets* bits of the double as an int! (Potential UB!)
int some_int = my_int_or_float.i;
```

A pointer to a **union** is the correct way to reinterpret memory when needed (e.g. packing multiple types into a buffer).

List

What should we use to create our `List` type?

a
```
typedef double* List;
```

b
```
typedef enum { /* ... */ } List;
```

c
```
typedef struct { /* ... */ } List;
```

d
```
typedef union { /* ... */ } List;
```

List

We need to keep track of both the backing storage and the length, so a `struct` makes the most sense.

List

We need to keep track of both the backing storage and the length, so a **`struct`** makes the most sense.

It could be done in a few ways. We will start with the following:

```c
#include <stddef.h> // for size_t
typedef struct {
    size_t len;
    double* data;
} List;
```

List

We need to keep track of both the backing storage and the length, so a `struct` makes the most sense.

It could be done in a few ways. We will start with the following:

```c
#include <stddef.h> // for size_t
typedef struct {
    size_t len;
    double* data;
} List;
```

Next we need to create the `list_init()` function, so let's take a look at functions in C.

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 seen an function definition for `main()`, and used the `malloc()` standard library function.
- In the case of `main()`, the C compiler knows to look for it as the main entry point to 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 seen an function definition for `main()`, and used the `malloc()` standard library function.
- In the case of `main()`, the C compiler knows to look for it as the main entry point to the program.
- Let's add a `list_init()` function to create an instance of our `List` type.

Functions

```c
#include <stddef.h>      // For size_t
#include <stdio.h>       // For printf()
typedef struct {size_t len; double* data;} List;
List list_init(void);    // Forward declare function. (void) means no args
int main(void) {
    List test_list = list_init();   // Create instance of List & print it
    printf("test_list: {\n\tlen: %d\n\tdata: %p\n}\n",
           test_list.len, test_list.data);
    return 0;
}
List list_init(void) {
    return (List) {.len = 0, .data = NULL};
// OR: List out = {.len = 0, .data = NULL};   return out;
}
```

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

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

Functions

Functions that take input arguments must specify the type of each argument.

```
int add_ints(int a, int b) {
    return a + b;
}
```

Input arguments are local variables, so changing them DOESN'T change the variable passed in:

```
void change(int a) {
    a = 10;
}
int main(void) {
    int a = 5;
    change(a);
    return a;    // Returns 5, since change(a)
                 // only changes its local copy
}
```

Functions

Functions can call other functions, and the result of a function call can be passed directly as an argument:

```
int square(int a) {
    return a * a;    // NOTE: `a^2` would be `a bitwise xor 2`!
}

int do_stuff(void) {
    int a = 10;
    int b = add_ints(a, square(a));
    return b * 4;
}
```

Function arguments can be marked as **const** like other variable declarations:

```
void do_things(const int a, const float b) {
    /* No chance of accidentally changing a or b */
}
```

Functions - 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.

Recursion

- Recursion is when a function calls itself.
- 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.

```
// 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);
}
```

Hands-on: Try creating a `list_print()` function to print the list info

```
// This code should print the same thing
// as the test_list code from earlier
List test_list = list_init();
printf("test_list: ");
list_print(test_list);
```

```
// Prints:
// test_list: List {
//     len: 0
//     data: (nil)
// }
```

Functions

The next function to implement is `list_append()`. For that, we need to be able to change the instance of `List` passed to the function.

Functions

The next function to implement is `list_append()`. For that, we need to be able to change the instance of `List` passed to the function.

To change things outside of a function, use input arguments which are pointers to the things you want to change:

```
void change(int* a) {
    *a = 10;
}
int main(void) {
    int a = 5;
    change(&a);       // NOTE: passing in ADDRESS of `a`!
    return a;         // Returns 10
}
```

Functions

Pointers as function arguments can also allow multiple return values:

```
void return_two_things(int* out_a, float* out_b) {
    *out_a = 10;
    *out_b = 5.0;
}
void do_things(void) {
    int a; float b;
    return_two_things(&a, &b);
    // Now a == 10 and b == 5.0
}
```

Functions

When a pointer needs to be taken but its contents won't be changed, it's good practice to take it as a pointer to `const` as a promise to the caller (and to help the compiler make optimisations):

```
int add_first_three(const int* elements) {
    return elements[0] + elements[1] + elements[2];
}
void array_is_unmodified(void) {
    int array[] = {1, 2, 3, 4, 5};
    int sum = add_first_three(array); // sum == 6
}
```

To also make the pointer itself a `const` as well, use:

```
void func(const char*const some_const_string);
```

Unless it comes before the type, `const` applies to the left, so `const int` and `int const` are exactly equivalent.

Functions - Application

With this information, now we can write `list_append()`:

```
void list_append(List* list, const double value) {
    // Reallocate the backing storage to have space for the new element.
    // realloc() is like malloc(), but will grow the memory pointed to
    // when possible for speed.
    // If list->data == NULL it will just allocate new memory.
    list->data = (double*)realloc(
        list->data, sizeof(double) * (list->len+1));

    // Add the new element
    list->data[list->len] = value;

    // Update the length
    list->len++;

    // Alternatively, the above two lines could be combined as:
    // list->data[list->len++] = value;
}
```

Functions - Application

We should also add a `list_destroy(List*)` function to make sure memory is cleaned up!

```
void list_destroy(List* list) {
    free(list->data);
    list->data = NULL;  // NULL out to make errors easier to find
    list->len = 0;
}
```

Functions - Application

We should also add a `list_destroy(List*)` function to make sure memory is cleaned up!

```
void list_destroy(List* list) {
    free(list->data);
    list->data = NULL;  // NULL out to make errors easier to find
    list->len = 0;
}
```

But this has a bug!
If we just write

```
List my_list = list_init();
list_destroy(&my_list);
```

we'll be calling `free()` on a `NULL` pointer!

If statements

We can fix this simply with an **if** statement:

```
void list_destroy(List* list) {
    if (list->data != NULL) free(list->data);
    list->data = NULL;
    list->len = 0;
}
```

If statements

We can fix this simply with an **if** statement:

```
void list_destroy(List* list) {
    if (list->data != NULL) free(list->data);
    list->data = NULL;
    list->len = 0;
}
```

Note that `NULL` pointers evaluate to false, while non-null pointers evaluate to true, so this could equivalently (but less clearly) be written as

```
if (list->data) free(list->data);
```

Control Flow - `if` statements

An **`if`** statement takes an expression, and conditionally executes the next block.

```
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

Unlike Python, indentation is not required, but it's still good practice for readability.

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 a variable (often an **`enum`**).

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);
```

Control Flow - `for` loops

`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).

```
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

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

```
// 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");
```

Quiz 3

1 What will be printed by the code below?

```
int a = 10;
switch (a) {
    case 2:
        printf("Hello, ");
        break;
    case 10:
        printf("Hi, ");
        if (a % 2) break;
    case 5:
        printf("goodbye, ");
    default:
        printf("goodnight.");
}
```

a `"Hello, "`

b `"Hi, "`

c `"Hi, goodnight."`

d `"Hi, goodbye, "`

e `"Hi, goodbye, goodnight."`

f It won't print anything.

g This code will cause a compiler error.

Quiz 3

1 What will be printed by the code below?

```
int a = 10;
switch (a) {
    case 2:
        printf("Hello, ");
        break;
    case 10:
        printf("Hi, ");
        if (a % 2) break;
    case 5:
        printf("goodbye, ");
    default:
        printf("goodnight.");
}
```

a `"Hello, "`

b `"Hi, "`

c `"Hi, goodnight."`

d `"Hi, goodbye, "`

e `"Hi, goodbye, goodnight."` ←

f It won't print anything.

g This code will cause a compiler error.

Quiz 3

2 How many times will "hello" be printed?

```
for (int i = 10; i > 0; i -= 2) {
    for (; i > 4; --i) {
        printf("hello\n");
    }
}
```

a 4

b 5

c 6

d 20

e This code will cause a compiler error.

Quiz 3

2 How many times will "hello" be printed?

```
for (int i = 10; i > 0; i -= 2) {
    for (; i > 4; --i) {
        printf("hello\n");
    }
}
```

a 4

b 5

c 6 ←

d 20

e This code will cause a compiler error.

Hands-on

The code below stores some values in a list.

Write a `list_print_contents` function to print the contents of the list:

```
void list_print_contents(List);      // TODO: Implement this
int main(void) {
    List test_list = list_init();    // Initialise list

    // Store square numbers
    for (int i = 0; i <= 10; ++i)
        list_append(&test_list, (double)(i*i));

    // Should print out:
    // [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
    list_print_contents(test_list);

    list_destroy(&test_list);      // Clean up
    return 0;
}
```

Input Arguments

Let's change our program so that it takes the number of square numbers to print out as a runtime argument:

```
$ ./squares 5
[0, 1, 4, 9, 16, 25]
$ ./squares 3
[0, 1, 4, 9]
$
```

Input Arguments

Let's change our program so that it takes the number of square numbers to print out as a runtime argument:

```
$ ./squares 5
[0, 1, 4, 9, 16, 25]
$ ./squares 3
[0, 1, 4, 9]
$
```

To read the input argument, we need to use the full signature of the `main` function:

```
int main(int argc, char* argv[]) {
    /* ... */
}
```

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
    printf("Called with \"%s\"\n", argv[0]);
    printf("Number of arguments: %i\n", argc - 1);
    return 0;
}
```

Running this code:

```
$ gcc main.c -o call_info
$ ./call_info
Called with "call_info"
Number of arguments: 0
```

```
$ ./call_info hello world "one string"
Called with "call_info"
Number of arguments: 3
$
```

Input Arguments

We can handle the input argument with some basic error checking with:

```
#include <stdio.h>      // For printf()
#include <stdlib.h>     // For atoi() and isdigit()
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;
    }
    // Make sure it's a positive integer
    for (int i = 0; argv[1][i] != '\0'; ++i) {
        if (!isdigit(argv[1][i])) {   // isdigit() returns true for [0-9]
            printf("ERROR: expected a positive integer\n");
            return 2;
        }
    }
    int N = atoi(argv[1]);   // Read input. atoi() converts char* to int
    /* ... construct and print list as before ... */
```

How can we make our `List` type accessible in other code through an `#include`?

The Preprocessor

- The preprocessor operates before compilation begins, and directives begin with #

The Preprocessor

- The preprocessor operates before compilation begins, and directives begin with `#`
- `#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!

The Preprocessor

- The preprocessor operates before compilation begins, and directives begin with `#`
- `#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.
  - The `-D` compiler flag can also be used to `#define` macros (e.g. `gcc ... -Dmy_const=42`)

The Preprocessor

- The preprocessor operates before compilation begins, and directives begin with `#`
- `#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.
  - The `-D` compiler flag can also be used to `#define` macros (e.g. `gcc ... -Dmy_const=42`)
- `#if .. #elif .. #else .. #endif` can be used to conditionally compile sections of code based on constants available from `#define` macros.

The Preprocessor

- The preprocessor operates before compilation begins, and directives begin with `#`
- `#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.
  - The `-D` compiler flag can also be used to `#define` macros (e.g. `gcc ... -Dmy_const=42`)
- `#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 `#define`d
  - `#elifdef` and `#elifndef` are C23 features! For older compilers, use `#elif defined(name)` and `#elif !defined(name)` instead.

The Preprocessor

- The preprocessor operates before compilation begins, and directives begin with `#`
- `#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.
  - The `-D` compiler flag can also be used to `#define` macros (e.g. `gcc ... -Dmy_const=42`)
- `#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 `#define`d
  - `#elifdef` and `#elifndef` are C23 features! For older compilers, use `#elif defined(name)` and `#elif !defined(name)` 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, and directives begin with `#`
- `#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.
  - The `-D` compiler flag can also be used to `#define` macros (e.g. `gcc ... -Dmy_const=42`)
- `#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 `#define`d
  - `#elifdef` and `#elifndef` are C23 features! For older compilers, use `#elif defined(name)` and `#elif !defined(name)` 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 List code.**

The Preprocessor

list.h:

```
#ifndef LIST_H
#define LIST_H
#include <stddef.h>
typedef struct {
    size_t len;
    double* data;
} List;
List list_init(void);
void list_destroy(List*);
void list_append(List*, double);
void list_print(List);
void list_print_contents(List);
#endif
```

The Preprocessor

list.c:

```
#include "list.h"
/* ... Implementations of List methods go here ... */
```

main.c:

```
#include "list.h"
int main(int argc, char* argv[]) {
    List test_list = list_init();
    list_append(&test_list, 42.);
    list_print(test_list);
    list_print_contents(test_list);
    list_destroy(&test_list);
    return 0;
}
```

The Preprocessor - `#include`

Try compiling like before:

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

The Preprocessor - #include

Try compiling like before:

```
$ gcc main.c -o test_list
/usr/bin/ld: /tmp/ccoU3ljs.o: in function `main':
main.c:(.text+0x1f): undefined reference to `list_init'
/usr/bin/ld: main.c:(.text+0x3f): undefined reference to `list_append'
/usr/bin/ld: main.c:(.text+0x52): undefined reference to `list_print'
/usr/bin/ld: main.c:(.text+0x65): undefined reference to `list_print_contents'
/usr/bin/ld: main.c:(.text+0x71): undefined reference to `list_destroy'
collect2: error: ld returned 1 exit status
```

The Preprocessor - `#include`

Try compiling like before:

```
$ gcc main.c -o test_list
/usr/bin/ld: /tmp/ccoU3ljs.o: in function `main':
main.c:(.text+0x1f): undefined reference to `list_init'
/usr/bin/ld: main.c:(.text+0x3f): undefined reference to `list_append'
/usr/bin/ld: main.c:(.text+0x52): undefined reference to `list_print'
/usr/bin/ld: main.c:(.text+0x65): undefined reference to `list_print_contents'
/usr/bin/ld: main.c:(.text+0x71): undefined reference to `list_destroy'
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 test_list
/usr/bin/ld: /tmp/ccoU3ljs.o: in function `main':
main.c:(.text+0x1f): undefined reference to `list_init'
/usr/bin/ld: main.c:(.text+0x3f): undefined reference to `list_append'
/usr/bin/ld: main.c:(.text+0x52): undefined reference to `list_print'
/usr/bin/ld: main.c:(.text+0x65): undefined reference to `list_print_contents'
/usr/bin/ld: main.c:(.text+0x71): undefined reference to `list_destroy'
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 list.c -o test_list
$
```

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 `list.c`, then `list.o` would need to be recompiled, as would the final executable, but other `.o` files from code that doesn't depend on `list.c` can be reused.

```
$ gcc list.c -c
$ gcc main.c -c
$ gcc main.o list.o -o my_list_exe
$
```

Macros

`#define` isn't just for defining constants. It can also be used for macros!

We could use it to create a convenient macro for iterating over the list:

list.h:

```
// ...
// NOTE: \ is required for line continuation in #defines
#define list_foreach(it, list) \
    for (double* it = (list).data; it < (list).data + (list).len; ++it)
// ...
```

main.c:

```
// ...

// This copy-pastes the macro contents with `list` filled as `my_list`
// and `it` filled as `val`
list_foreach(val, my_list) {
    printf("%d\n", *val);
}
```

As your project grows, you'll find it useful to manage compilation with a build system.

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

all: test_list

list.o: list.c
  ${CC} ${CFLAGS} $^ -c

test_list: main.c list.o
  ${CC} ${CFLAGS} $^ -o $@
```

Build Systems - `make`

Now we can build with:

```
$ make -j4 # -j flag for number of parallel build tasks
$
```

There is a lot more that can be done with `make` for more advanced configuration.

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 list.c)
add_executable(test_list ${SRCS})

# Apply compiler flags
target_compile_options(test_list PRIVATE -Wall -Wextra
  -Wpedantic -Werror)
```

Build Systems - cmake

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
$
```

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.

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.

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).

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
  - File IO example: https://en.cppreference.com/w/c/io/fopen

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
  - File IO example: https://en.cppreference.com/w/c/io/fopen
- 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

Build your mountain

As you solve common problems in C, you can build your own library of useful code, including your own `List` type!
This `List` implementation is quite simplistic, but we'll work on improving it tomorrow.

Build your mountain

As you solve common problems in C, you can build your own library of useful code, including your own `List` type!
This `List` implementation is quite simplistic, but we'll work on improving it tomorrow.

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 in for libraries with the `-L` flag.

One last practice problem

Practice Problem

Write a program that reads an integer, *N*, from the command line, populates a `List` with the first *N* Fibonacci numbers, and then prints out that list.
You should be able to use many of the `list_*` functions we created today.

Example output:

```
$ ./fib 5
[1, 1, 2, 3, 5]
$ ./fib 10
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
$ ./fib -1
ERROR: Invalid input!
$ ./fib 0
$
```

As an extension, try taking a file name as the command line input, where each line of the input file is a number, *N*, for which the first *N* Fibonacci numbers should be printed.