CMOR 420/520
Computational Science
Programming in C

Skills to develop

- Be able to write your own code
- Be able to read other peoples codes

- Computers are simple and literal. They will do exactly what they are told.
- When reading a code, you must follow what the computer is doing (not what you want it to do).
- If the computer is not doing what you want, you need to be able to edit the code to make the computer do what you want

A brief history of C

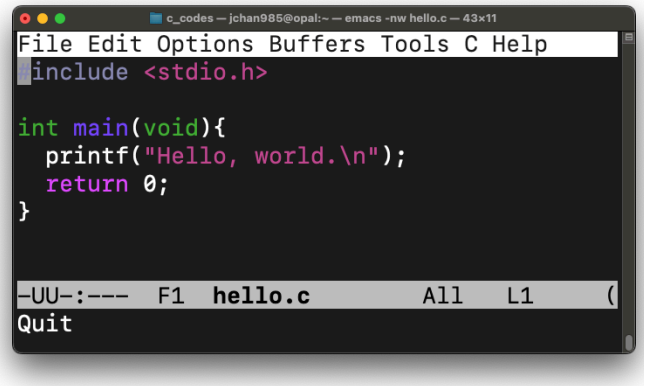
- Created by Dennis Ritchie and Brian Kernighan in the 1970s
- Note that C is a *standard*; there is a <u>document</u> specifying how C and its compilers should behave.
  - Organizations create their own *implementations* of the C standard, e.g., “gcc” vs “clang” vs “icc” vs …
- Simple feature set, hasn’t changed much over 50 years.
- Portability: you can probably run a C code on any machine.
- Versatility: can be used for low level programming (e.g. writing an operating system) to scientific computing.

How to build and run a C program

- C programs need to be *compiled* before being run.
- We need to install the C compiler “gcc” (**G**NU **C c**ompiler)
  - On Linux, “sudo apt install gcc”
  - On Mac, installing Xcode (via “xcode-select —install”) should install gcc as well.
- Usage: “gcc hello.c” will create an executable file “a.out”
  - “gcc hello.c -o hello” will create an executable “hello”

Breakdown of a “hello world” C program

- “stdio.h” is an IO *(header) library*
- The “main” function is what a C program runs.
- “int main(void)” specifies input/output
    - “void” means there are no arguments to this function
    - “int” means that the function returns a integer (often an error code)



Exercise

- Make your first C program!Open VSCode or another test editor
- Make a blank file and add the following lines:

```
#include <stdio.h>

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

- save the file with the name ”hello.c”
- in a terminal, use ”gcc hello.c -o hello” to make an executable
- run the exectutable, use ”./hello”

“printf”

- In the “stdio.h” library, “printf” prints a formatted message to screen.
- Uses format specifiers to parse input into a string; “%d” for ints, “%f” for decimal
  - “%7.3f” means print out 7 spaces with 3 decimals.
- “\n” is a newline character.

```c
#include <stdio.h>

int main(void){
  int x = 3;
  double y = 1.5;
  float z = 1.f;
  printf("x = %d, y = %f, z = %7.3f.\n",
         x, y, z);
}
```

```
(base) jchan985@XXH62CK9GR c_codes % gcc test.c
(base) jchan985@XXH62CK9GR c_codes % ./a.out
x = 3, y = 1.500000, z =   1.000.
```

A slightly more complex C program

- Can define comments in two ways
  - inline comment using both “//”
  - multi-line comment using “/* … */”.
- “argc” and “argv” capture command-line input arguments (we’ll cover “char **” later)
  - “argc” $\geq$ 1 is the number of arguments
  - “argv[i]” is the ith argument (the first argument is the program path/name)
- “atof” is a function in “stdlib.h” which converts a characters to a float value.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char ** argv){

  /*
     In C, one can define, declare,
     and initialize a variable in
     separate steps.
  */

  // Declared
  float x;

  // Initialized
  if (argc > 1){
    x = atof(argv[1]);
  } else {
    x = 1.0f;
  }
  printf("x = %f\n", x);
}
```

Undefined C behavior

- C is pretty permissive; you can perform operations whose behaviors are *undefined.* Some examples:
  - Define “int x;”, then run “printf(“x = %d\n”, x)”? Here, “x” is initialized but not defined.
  - Define “int x = -1”, then set “unsigned int y = x”. What will “y” be?
  - Define “int x = 1;”, then run “printf(“x = %f\n”, x)”? Note we are using the incorrect format here.
- You may get reasonable answers, but these are technically *undefined* behaviors (their behavior, or even whether they compile, may change depending on the computer or compiler).

Administrative stuff

- Some example C codes are available on Canvas now

Try guessing how C behaves

- Some behaviors in C are well-defined (e.g., the C standard specifies what should happen) but can be non-intuitive.
  - Define “char a = ‘5’” and “char b = ‘2’”. What is the value of “int x = a-b”?
  - Define “char a = ‘5’”. What is the value of “x” for “int x = a”?

In C, “char” types are actually 1-bit integers
(you just need to represent enough integers
to enumerate the unique characters)

Data types in C

- 4 main types: “int, float, double, char”
- 4 modifiers: “short, long, unsigned, signed (not typically used)”
  - Some modifiers don’t do anything (“short int”)
  - “long int” most common
- Can use “sizeof” to determine number of bytes in each data type.

```c
#include <stdio.h>
#include <limits.h> // max/min values of different data types
#include <float.h> // info about floating point standards

int main(void){

  // C is simple: 4 data types in C
  int x = 1;
  float y = 1.0; // single precision
  double z = 1.0; // double precision
  char cc = 'a';

  // short + long modifier
  short int a;
  long int b;
  long c; // same as long int
  long double d; // depends on implementation

  printf("size of int = %lu\n", sizeof(int));
  printf("size of float = %lu\n", sizeof(float));
  printf("size of double = %lu\n", sizeof(double));
  printf("size of char = %lu\n", sizeof(char));
  printf("size of long int = %lu\n", sizeof(long int));
  printf("size of long double = %lu\n", sizeof(long double));

  printf("Max value of unsigned int = %u\n", UINT_MAX);
  printf("Max value of unsigned long int = %lu\n", ULONG_MAX);
  printf("Max value of float = %g\n", FLT_MAX);
  printf("Max value of double = %g\n", DBL_MAX);
}
```

data_types.c

Type casting and conversions

- Discussed int, float, double data types. What happens if you do the following:

```
int x = 10;
float y= x;

float x = 10.1;
int y = x;
```

- C will auto convert the given value to the designated variables data type.
  - Fine if making the data type more precise (int $\rightarrow$ float)
  - Making the data type less precise is a problem (float $\rightarrow$ int)

- Type casting is the C conversion feature used to specify when and how to convert a value

```
int x = 10;
x = (float) 10;
```

Control flow in C

- C has the usual control flow: for loops, if/else statements, while loops, break/ continue statements, do-while loops.
- “do-while” loops are like while loops, but execute one iteration before checking the while condition
- “break” and “continue” statements exit the current iteration (either stopping or continuing)
- Note that “0” and “1” are the same as “true” and “false” in C.
- See control_flow.c for examples

```c
// an if loop example
if (argc == 1) {
  printf("No arguments.\n");
} else if (argc == 2){
  printf("One argument.\n");
} else {
  printf("More than one argument.\n");
}

// for loop
int n = 4;
for (int i = 0; i < n; i++) {
  printf("On iteration %d of %d\n", i, n-1);
}

// while loop
int i = -1;
while (i < n) {
  printf("On iteration %d of %d in while loop\n", i, n-1);
  i += 1;
}

// do-while
i = -1;
do {
  printf("On iteration %d of %d in do-while loop\n", i, n-1);
  i += 1;
} while (i < n);

// break and continue statements
while (1) {
  if (i % 2) {
    continue; // exits the current iteration but continues
  }
  printf("On iteration %d out of 10\n", i);
  if (i == 10) {
    break; // exits the current iteration and stops
  }
}
```

More about Control Flow

- $==$ is the conditional 'equal',
  && is the conditional 'and',
  || is the conditional 'or',
  ! is the conditional 'not'
- $i+=1$, $i++$, is the same as $i=i+1$;

- The condition that if, else, for, while, etc check and follow is contained in parentheses
- for loops 'conditions' have 3 parts:
  1. variable that leads the loop,
  2. maximum (or minimum) value attainable by the variable,
  3. increment of the variable

$$(int\ i=0, i<10, i++)$$

Function definitions in C

- We can define additional functions that can be called from our “main” function.
- However, C is sensitive to where function definitions are located.
- What happens if we swap the order of the functions?

```c
#include <stdio.h>

int my_func_1(int x) {
  return x + 1;
}

int my_func_2(int x) {
  return my_func_1(x) * 2;
}

int main(void) {
    int x = 2;
    printf("The output of "
           "`my_func_2(%d) = %d\n",
           x, my_func_2(x));
}
```

multiple_functions.c

Exercise

- include the $< math.h >$ library
- initialize an int x
- Make a function called 'function_1' that computes:

$$y = 1/x^2$$

  (look up online for to computes powers using the math.h library)
- Write a for loop for $i = 1$ to $i = 5$ that computes function_1(i) and prints the results.

C function prototypes and header files

- Functions in a C program must be defined in order from top to bottom. This is hard to manage if you have a lot of functions.
- Solution: introduce “function prototypes”; C doesn’t need to know the entire function, just what the inputs and outputs are.
- “Header files” typically contain a list of function prototypes
  - Can also contain full function definitions; this is common for libraries which are “header-only”.
- See “multiple_functions_with_header.c” and “my header.h”

Exercise

- make a header file for the previous function
- move the function definition

How do C functions treat arguments?

Lets consider the function “foo”:

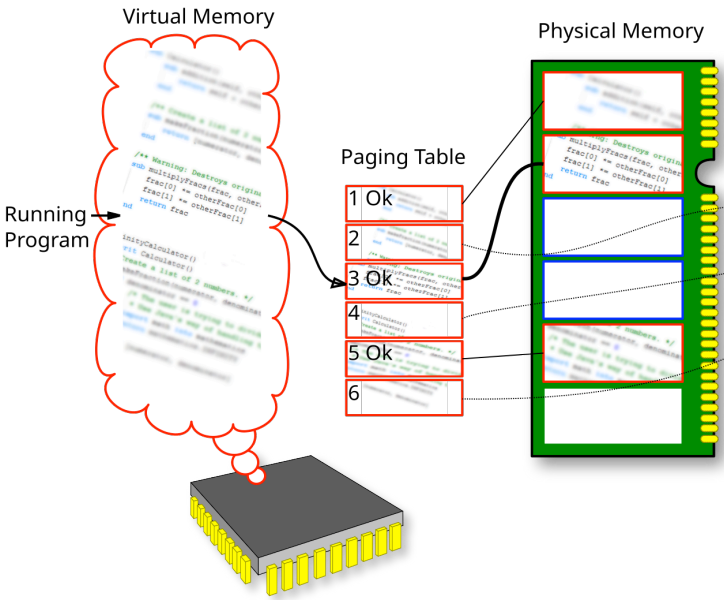
```
void foo(int x){
  x += 1;
}
```

Suppose we ran the code “int x = 1; foo(x);”. What would the value of “x” be afterwards?

C functions: scope and pass-by-copy

- C functions are “pass by copy” or “pass by value”; they create a local copy of whatever variable is passed.
- These local copies are created when the function is called and destroyed when it is done running
- Can be useful to modify function inputs:
  - C doesn’t have a way to return multiple outputs, so you would need to pass in output variables as arguments and modify them. How can we do so?

Pointers



- A pointer address is a **memory address** of a variable in C.
- Provides the variable's *location* but not what's contained inside.
- The address is *not* typed, but pointers in C are typed. Why? C uses the pointer type for array indexing…

https://en.wikipedia.org/wiki/Memory_address

Pointers in C

- Pointers variables store a *memory address.*
- Syntax: specify the variable type and add *****
  - e.g., “int * x”, “float * y”, “int ** z”, …
- You *reference* a variable’s address via “&”: “int* x_ptr = &x;”
  - Referencing converts a variable to its address.
- You *de-reference* a pointer address via ***** : “int x = *x_ptr;”
  - De-referencing converts an address to a value.

Pointer arguments enable “pass by reference” in C

- Recall C functions make copies of arguments (pass by copy)
- Example: suppose a function modifies its input variables.
  - This can happens when you have multiple return arguments or want to save memory.
- Passing a pointer allows you to bypass pass-by-copy and modify a function’s input.

```c
#include <stdio.h>

void modify_args(int x){
  x += 1;
}

void modify_args_ptr(int * x_ptr){
  (*x_ptr) += 1;
}

int main(void){
  int x = 1;
  modify_args(x); // will not modify x
  modify_args_ptr(&x); // will modify x
  printf("The value of x = %d\n", x);
}
```

Creates a local copy of x, updates it, and discards it when the function is done.

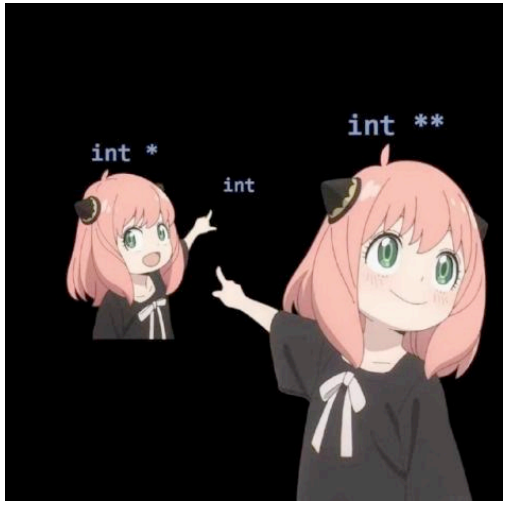
Exercise

- Make a .c file with a function called 'add_one'
  1. takes int data type as an input, and int data type as the output
  2. returns the value of $x + 1$
- include a header file for your new function

- Redo the function with pointers. Dont forget to edit the header file!
  1. takes int* data type as an input, and void as the output
  2. no longer needs a return statement!

Pointers can point to almost anything

- Can have pointers to pointers to pointers to pointers …
  - Pointers (memory addresses) are just another type of variable in C.
  - A pointer to a pointer (e.g., “int**”) is just the memory address of a memory address variable “int*”.
- Flexible: pointers can be used with variables, arrays, custom data types, and even functions.
- Pointers are probably the most influential concept in C.



Pointers and arrays

- In C, arrays and pointers are often conflated.
- You can create an array in C via “int x[4] = {1, 2, 3, 4};”
  - “x[0], x[1], …” access the entries of “x”.
  - “x” is a *pointer to the first element* of the array “x”, so “(*x) = 1” as well.

```c
#include <stdio.h>

void print_vector(int n, int * x){
  for (int i = 0; i < n; ++i){
    printf("x[%d] = %d\n", i, x[i]);
  }
}

int main(void){
  int x[4] = {1, 2, 3, 4};
  print_vector(4, x);
  printf("x's address is %p\n", x);
}
```

See also “array_example.c”

Input arguments are pointers to “char *”

- A single letter is a “char” type. An array of letters (e.g., “char* ” is referred to as a string)
- We can print out elements of “argv” using the “%s” string formatting option

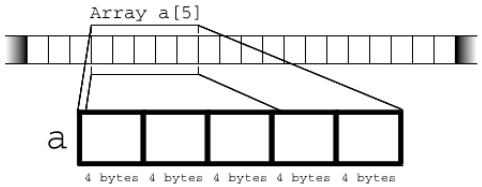
```c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("You have %d values available.\n", argc);
  printf("The first is: %s\n", *argv);
  printf("The second is: %s\n", *(argv+1));
}
```

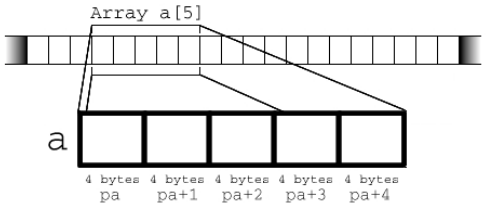
```
> gcc main-args.c
> ./a.out 1.2
You have 2 values available.
The first is: ./a.out
The second is: 1.2
```

- Recall that “(*argv)” is the first element of “argv”.
  - What is “ *(argv+1)” doing here?

Pointers arithmetic and indexing

- Suppose we create an array via “int a[5] = {1, 2, 3, 4, 5};”
  - “a” is a *pointer to the first element* of the array, so “(*a) = 1”
- C defines that “a[1] = * (a + 1)”. What does “(a + 1)” do?
- Recall that “a” is an “(int*)” type; C knows that integers are 4 bytes each. “(a + 1)” returns the memory address *one integer over* from the memory address (first element) of “a”.

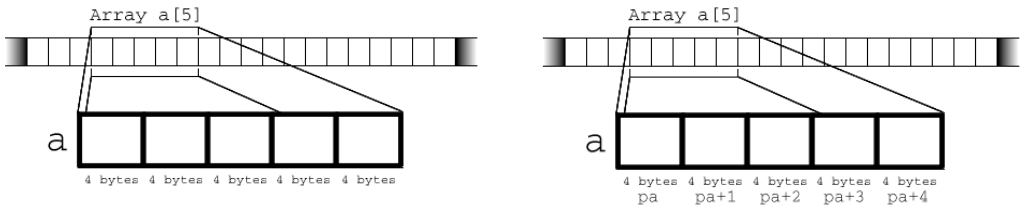




http://www.rc.usf.edu/tutorials/classes/tutorial/c_intro/chapter7.html

Pointers arithmetic and indexing, cont.

- C knows that integers are 4 bytes each. “(a + 1)” returns the memory address *one integer over* from the first element of “a”.



- Note also that arrays defined this way in C are *contiguous in memory*, e.g., **elements that are next to each other in the array are next to each other in memory**.
- Why is memory layout important? Efficiency; take HPC for more!

http://www.rc.usf.edu/tutorials/classes/tutorial/c_intro/chapter7.html

Exercises: pointers and arithmetic

- You can create an array in C via “int x[4] = {1, 2, 3, 4};”. Note that “x[0], …, x[3]” accesses the entries of the array.
  - What will accessing “x[4]” return?
  - What about “*x”?
  - What about “*(x+2)”?
  - What about “*(&x[3]-2)”?

Exercise

- Make an integer array with values 2, 4, 6, 8, 10
- Compute and print the array element '8' using both pointer format $*(x + i)$ and $x[i]$ format.
- Access the array element '4' and add one i.e. you should have the value '5'. Print this.

Exercises: pointers and illegal access

- Suppose you set “int x = 1; int* x_ptr = &x;”.
  - What will “x_ptr[0]” and “x_ptr[1]” return?
  - What happens if you set “x_ptr[1]=10”?

Exercises: pointers and casting

- You can convert one type of variable to another by *casting*
  - Example: “char c = ‘a’; int x = (int) c” (usually done automatically).
  - Can also force conversion of *pointers* from one type to another.
- What happens if you convert one pointer to another type?
  - Suppose you define “int x[4] = {1, 2, 3, 4};”
    - Convert x to “float*” type via “float* y = (float*) x;”
    - What does “printf("%d\n", *((int *)(y+1)))” yield?

Pointers and n-D arrays

- C matrices are stored as *flat vectors* in either *row major* or *column major* format.
  - Example: the 2D array can use a single initializer list (assumes row major storage)
- Note: syntax for passing statically sized C arrays into functions is a bit odd; we will mainly pass pointers.

```c
#include <stdio.h>

void print_matrix(int x[2][2]) {
  for (int i = 0; i < 2; ++i){
    for (int j = 0; j < 2; ++j){
      printf("%d ", x[i][j]);
    }
    printf("\n");
  }
}

int main(void) {
  int x[2][2] = {{1, 2}, {3, 4}};
  int y[2][2] = {1, 2, 3, 4};
  int z[2][2] = {{0}};

  print_matrix(x);
}
```

Function pointers

- In C, a pointer can point to a function. Some applications:
  - Sorting: to sort something non-standard (e.g., a vector of "(x, y)" pairs), pass a custom "comparison" function pointer.
  - Timing: pass a pointer to the function you want to time.
- Getting a function pointer is the same as referencing any other memory address: " &function_name" returns a pointer to "function_name".
- Taking a function pointer as input is trickier…

Function pointer example

- Here, we define “function_ptr” and use it to add two ints together.
- For a more complex example, look at “function_pointers.c”, which compares entries of two vectors using a pointer to a comparison function.

```c
#include <stdio.h>

int add(int x, int y){
  return x + y;
}

int main(void){
  int x = 1;
  int y = 2;
  int (*function_ptr)(int, int);
  function_ptr = &add;

  printf("output of function ptr = %d\n",
         (*function_ptr)(x, y));
}
```

(void *) pointers

- You can define a (void *) pointer; e.g., a pointer with no type. When is this useful?
- If you want one single function pointer interface for functions with different pointer type arguments, you can use (void *) pointers
  - Requires casting the pointer to the proper type.

```c
#include <stdio.h>
#include <stdlib.h>

void increment_int(void * x_in){
  int * x = (int *) x_in;
  (*x) += 1;
}

void increment_double(void * x_in){
  double * x = (double *) x_in;
  (*x) += 1;
}

int main(void){
  int x = 1;
  double y = 1.0;

  void (*function_ptr)(void *) = &increment_int;
  (*function_ptr)(&x);

  function_ptr = &increment_double;
  (*function_ptr)(&y);

  printf("value of x = %d\n", x);
  printf("value of y = %f\n", y);
}
```

void_pointer.c

Exercise

- Using the 'add_one' function from last time (or another function)
  - modify the function to take a void pointer input
  - type cast within the function to the correct type
- Make a function pointer and reference it to 'add_one'

Pointers != arrays

- Pointers and arrays are strongly connected in C, but they are distinct from each other.
- Example: even if two values are the same, pointers to their addresses will be different.
- Example: if two pointers are the same, then they point to the same memory address.

```c
int main(void) {
  int x = 1;
  int y = 1;
  int * x_ptr = &x;
  int * y_ptr = &y;
  printf("x == y = %d\n", x == y);
  printf("x_ptr == y_ptr = %d\n",
         x_ptr == y_ptr);
}
```

```c
#include <stdio.h>

int main(void) {
  int x = 1;
  int * x_ptr = &x;
  int * y_ptr = &x;
  printf("x_ptr == y_ptr = %d\n",
         x_ptr == y_ptr);

  // change x's value
  *y_ptr = 10;
  printf("x = %d, *x_ptr = %d\n",
         x, *x_ptr);
}
```

Pointers and Arrays

Similarities:

- Array elements can be accessed through pointer arithmetic $*(x + i)$
- a pointer can be accessed using array notation $x[0]$
- These are C defined behaviors

Differences:

- Arrays store data of a specific type in contiguous memory
  - Comparing arrays compares values (addresses dont matter)
- Pointers store addresses
  - Comparing pointers compares addresses (even if the value they store is the same)

Pointers can be dangerous!

- Example: *dangling pointers*.
- Variable scope: variables defined inside a C function *disappear* once the function finishes running!
- Returning a pointer to a variable which disappears can cause weird behavior!

```c
#include <stdio.h>

int* create_ptr_to_int(void) {
  int x = 12;
  return &x;
}

int main(void) {
  int* x_ptr = create_ptr_to_int();
  printf("x_ptr =  %p, x = %d\n",
         x_ptr, *x_ptr);
}
```

See also “dangling_pointer.c”

The “NULL” pointer

- If you have only one input argument for a function but try to access the second argument, you get a “null” pointer.
- If you initialize (but do not assign a value to) a pointer, it is set to a random memory address.
  - Deferencing a random memory address usually returns garbage
  - The NULL pointer returns an **error** when dereferenced.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("You have %d values available.\n", argc);
  printf("The first is: %s\n", *argv);
  printf("The second is: %s\n", *(argv+1));
}
```

```
(base) jchan985@XXH62CK9GR c_codes % ./a.out
You have 1 values available.
The first is: ./a.out
The second is: (null)
```

The “NULL” pointer: a mistake?

- Null pointers are essentially an “invalid” memory address.

```c
#include <stdio.h>

int main(void) {
  int * ptr;
  int * null_ptr = NULL;
  printf("An undefined pointer: %p\n", ptr);
  printf("Its dereferenced value: %d\n", *ptr);
  printf("A pointer defined as NULL: %p\n", null_ptr);
  printf("Its dereferenced value: %d\n", *null_ptr);
}
```

```
(base) jchan985@XXH62CK9GR c_codes % ./a.out
An undefined pointer: 0x103375bd0
Its dereferenced value: -80133272
A pointer defined as NULL: 0x0
zsh: segmentation fault  ./a.out
```

From Tony Hoare, inventor of the null pointer:

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

https://en.wikipedia.org/wiki/Null_pointer#History

Allocating memory: stack vs heap memory

- Computer memory (e.g., RAM) is split into “stack” and “heap” memory by the operating system.
  - The “stack” is faster, but has a smaller fixed size (~8 MB).
    - If each double is 8 bytes, what’s the largest vector that this can store? What about the largest matrix?
  - The “heap” is larger, but is (slightly) slower and has to be manually managed (“allocated” and “free’d” by the user)
- So far, every variable we have defined used “stack” memory, but exceeding stack memory limits leads to a “stack overflow”. We will use “heap” memory for large arrays.

Static vs dynamic memory allocation

- Memory can be dynamically allocated using “malloc”
  - Input: how many bytes you want to allocate.
  - Output: an untyped “(void *)” pointer, which we will *cast* to a pointer of the correct type.
- Allocation “binds” memory to a pointer. You should always “free” a pointer when you’re done with it.

```c
#include <stdio.h>

int main(void) {
  int x[3];
}
```

Uses *stack* memory

```c
#include <stdio.h>

int main(void) {
  int * x = (int *) malloc(3 * sizeof(int));
  free(x);
}
```

Uses *heap* memory

Other memory allocation routines

- “calloc”: similar to “malloc” but initializes the memory to zero
- "realloc(ptr, new_size)”: reallocate a chunk of memory of “new_size” bytes to pointer “ptr”
- “memcpy(dest, src, num_bytes)”: copies a block of memory of size “num_bytes” from a pointer “src” to a pointer “dest”

Matrix storage formats

- Contiguous storage formats are row or column major ordering
- Non-contiguous storage formats: elements are not guaranteed to be next to each other.
- Why is this important? It is faster to access an elements *neighbor* than a *random* entry.

Row-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

https://en.wikipedia.org/wiki/Row-_and_column-major_order

Matrices and dynamic memory allocation

- Allocating memory for a vector is easy; however, we have multiple options when allocating memory for a matrix:
  - Format 1: store a matrix as a single “double *” vector (row or column major) and manually index
  - Format 2: store a matrix as a “double **” array
- There will be differences in terms of how fast it is to access entries of each array

How to represent a matrix: format 1

Format 1: store a matrix as a single “double *” vector (row or column major) and manually index (contiguous storage)

- Advantages: simple, contiguous storage, easy to change between row and column major
- Disadvantages: annoying to manually index.

```
// Approach 1: flat array, column major
double * A;
A = malloc(sizeof(double) * m * n);
for (int j = 0; j < n; ++j){
  for (int i = 0; i < m; ++i){
    A[i + j * n] = (double) (i+j);
  }
}
```

How to represent a matrix: format 2

Format 2: store a matrix as a “double **” array, dynamically allocate memory for each row.

- Advantages: convenient, can use multi-dimensional indexing
- Disadvantages: non-contiguous memory

```
// Approach 2: array of arrays (row major)
double ** B;
B = malloc(sizeof(double*) * m);
for (int i = 0; i < m; ++i){
  B[i] = (double*) malloc(sizeof(double) * n);
  for (int j = 0; j < n; ++j){
    B[i][j] = (double) (i+j);
  }
}
```

Any alternatives?

- Would be nice to represent a matrix in a way that is both
  - Easy to access, e.g., can access (i,j)th entry via “A[i][j]”
  - Contiguous memory layout for efficiency.
- Turns out pointers make this fairly straightforward

How to represent a matrix: format 3

- Format 3: store a matrix as a “double **” array, but allocate one single contiguous chunk of memory for the entire array.
- Use pointer arithmetic to assign the pointer location for C[i]

```
// Approach 3: array of arrays (row major)
double ** C = malloc(sizeof(double*) * m);

// allocate a vector of contiguous memory
C[0] = malloc(sizeof(double) * m * n);
for (int i = 1; i < m; ++i){
  C[i] = C[0] + i * n;
}
```

* Each C[i] is a pointer so you can index into the pointer location by C[i][j]

More on Arrays

- This just assigns the memory. You still need to assign the values you want stored in memory.

```
for(int i = 0; i<n; ++i){
    for(int j = 0; j<m; ++j){
        C[i][j] = i * m + j;
    }
}
```

- For all of these, don't forget to free the ptr after use!

Exercise

- Make an array with contiguous memory (format 3)
  1. make it a 2×3
  2. The entries should be $1, 2, 3, \ldots$

Structs

- Since pointers do not store the dimensions of an array, we have to pass the dimensions to most array functions as well.

```
void print_2d_matrix(int m, int n, double ** A){
```

- You can avoid this by defining “structs” or “composite types”

```
struct MyMatrix{
  int m;
  int n;
  double ** A;
};
```

```
void print_MyMatrix(struct MyMatrix A);
```

Struct basics

- See “struct_simple.c”
- Access a structure field via “struct_name.field_name”
- Requires specifying the full type “struct Foo”
- Can initialize using an initializer list

```c
#include <stdio.h>

struct Foo{
  int x;
  char * name;
};

int main(void){

  struct Foo foo;
  foo.x = 1;
  foo.name = "foo";

  // can also use initializer lists
  struct Foo foo_2 = {1, "foo_2"};
}
```

Pointers to structs

- If you have a pointer to a struct, you can use “->” to access the field, e.g., “struct_name -> field_name”
  - In this example, “->” is doing the same thing as “(*foo_ptr).x” and “(*foo_ptr).name”

```c
int main(void){

  struct Foo foo;
  foo.x = 1;
  foo.name = "foo";
  printf("foo.x = %d, foo.name = %s\n",
         foo.x, foo.name);

  struct Foo * foo_ptr = &foo;
  foo_ptr -> x = 10;
  foo_ptr -> name = "foo_ptr";

  printf("After ptr accesses, "
         "foo.x = %d, foo.name = %s\n",
         foo.x, foo.name);
}
```

Structs are new types

- They do most things that data types like “int, double, char, float” do (can define pointers to structs, pass to functions).
- Note: “sizeof(struct)” is *at least* the sum of the sizes of the struct fields, but may be larger
  - C compilers sometimes pad struct memory for alignment purposes.

```c
#include <stdio.h>

struct Foo{
  int x;
  char * name;
};

int main(void) {
  printf("sizeof(Foo) = %lu\n",
         sizeof(struct Foo));
  printf("sizeof(int) + "
         "sizeof(char *) = %lu\n",
         sizeof(int) + sizeof(char *));
}
```

```
~/Desktop/cmor-420-520-f24/c_codes $ ./a.out
sizeof(Foo) = 16
sizeof(int) + sizeof(char *) = 12
```

Type aliases via “typedef”

- Typedef allows you to define type aliases (shortcuts), e.g.,
  - “typedef *type type_alias*”
- Often used to simplify struct syntax.

```c
int main(void) {

  typedef unsigned int uint;
  typedef double darray4[4];

  uint a = 1;
  darray4 b = {1, 2, 3, 4};
}
```

```c
struct MyStruct{
  int x;
  char * name;
};

typedef struct MyStruct MyStruct;

int main(void) {
  struct MyStruct x;
  MyStruct y;
}
```

→

```c
typedef struct{
  int x;
  char * name;
} MyStruct;

int main(void) {
  MyStruct y;
}
```

Structs and header files

- It's common to put struct definitions in header files. However, this can cause some compilation issues.
- What is going on?

- Answer: "#include" just copies and pastes input from a file.
  - "struct_header.h" is included twice! Once directly, and once through "struct_tools.h"

```c
#include <stdio.h>
#include "struct_header.h"
#include "struct_tools.h"

int main(void) {
  MyStruct x = {1, "x"};
  print_info(x);
}
```

struct_header.c

```c
typedef struct{
  int x;
  char * name;
} MyStruct;
```

struct_header.h

```c
#include "struct_header.h"

void print_info(MyStruct x){
  printf("Struct %s has value x = %d\n",
         x.name, x.x);
}
```

struct_tools.h

Structs and header files

- If you include a header file in both "main" and another header file, you get multiple struct definitions
- Solution: *header guards.*
  - This checks if the global variable "_MYSTRUCT_" is defined, and if so, it defines it and the struct.
  - Notice the "#"; just like "#include", this is a C *macro.*

```
#ifndef _MYSTRUCT_
#define _MYSTRUCT_

typedef struct{
  int x;
  char * name;
} MyStruct;

#endif
```

Exercise

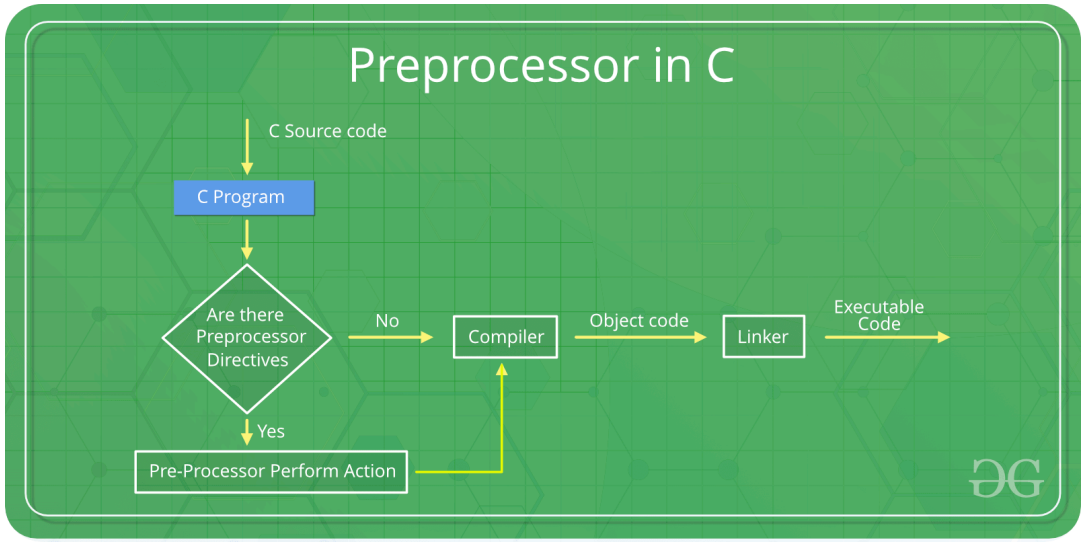
- Make a struct for the contiguous array from last time
- Include: int rows, int columns, double ** data
- typedef this struct, and place in a header file

- Work on the homework!

Macros and the “C preprocessor”

- The “#” actually refers to a *C macro*, otherwise known as a C preprocessor command. So what is the C preprocessor?
- Compilation is actually a series of steps, and the first step is resolving preprocessor directives. e.g.,
  - “#include <stdio.h>” resolves to the contents of “stdio.h”
  - “#define PI 3.14159” resolves by replacing every instance of “PI” with 3.14159.
- Can also define macro functions and expression, but beware…

The C preprocessor in context of compilation



https://www.geeksforgeeks.org/cc-preprocessors/

Pros and cons of macros

- Macros can be convenient: no type restrictions, global scope.
- Things to beware of with macros:
  - They look like functions but can behave very differently.
  - Hard to debug due to global effects
  - They don’t respect namespaces (which is why they are typically all-caps)

Compiling more complex projects

- Compiling across multiple files (simple version)
  - “gcc first_file second_file third_file …”
- What about if the include files are in a different directory?
  - “gcc -I./include first_file second_file …”
  - The flag “-I*path_to_include_directory*” specifies where gcc should look for header files.

Good coding habits

- It is expected, especially for large projects, that a project consist of multiples files of code
- Each file should contain code that achieves a specific 'function' or are closely related
- Using multiple files makes the overall program easy to read and easy to locate a desired portion of the code.
  No one wants to open and read a file with thousands of lines of code!

- Some codes can have dozens of files (or more)
  - Example: Legacy codes as labs, any code your professor uses

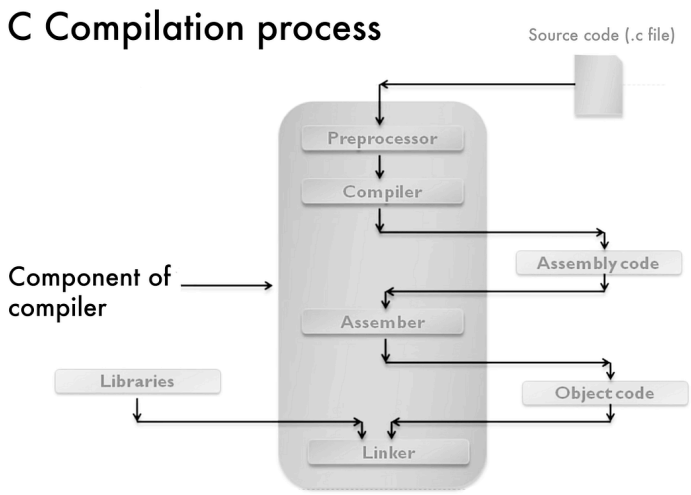
Good coding habits

- functions have descriptive names
- variables should have appropriate names (usually follow convention),
  - Example: To solve $\Delta u = f$, one would use the variable u and forcing f in the code.
- Even considering scope, try not to reuse important variable names unless the context is appropriate.
  - Example: To solve $\Delta u = f$, one would not use the variable u and f to mean anything other than the solution u and forcing f.

Good coding habits

- Use notes to describe the code
  - Describe what file does
  - Describe what a function does
  - Describe what the important variables denote
  - Leave notes to yourself or others about important parts of code
- Helps you find/recall what a file or function does without reading it line by line!
- Helps your coworker (or advisor) read the code

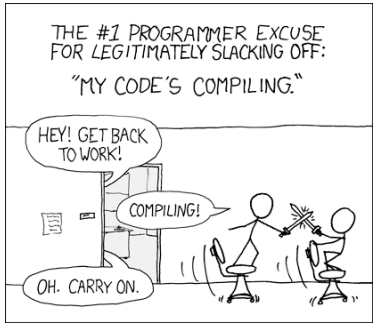
What actually happens during compilation?



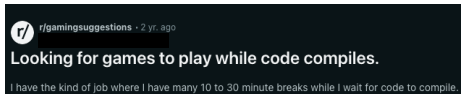
https://agustinespinoza.medium.com/c-program-compilation-process-6452628f275d

Compiling in steps

- Can separate compilation of individual files (e.g., creating object files) and combining compiled files together (e.g., linking)
- Why break up compilation steps?
  - Compilation can take a **very** long time, especially for large codebases.
  - Using language features like C++ templates increases compile times.
  - Separate steps means you can skip recompiling a file if it hasn't changed.



https://xkcd.com/303/



Example: a simple compilation workflows

- Compile .c files to .o files via “gcc -I*include_dir* -c *file_name*” (makes one .o file)
- Linking via “gcc *first_file_name.o second_file_name.o …”*

* .o files are object files, not executables
* Exectuable is made using “gcc -o desired_name first_file.o second_file.o …”

- New problem: if you split up compilation commands, you now have multiple compile commands to run. Some fixes:
  - Put the compile commands in a shell script
  - Makefiles (and other “build systems”)

Exercise

- take one of the previous exercise codes, move the functions (not main) into a seperate .c file
- make the .o exectuables for both the main .c file and new .c file
- link them i.e make the exectuable and run it

Makefiles and other build systems

- It's hard to scale individual compile commands to larger codebases, and shell scripts don't detect changes to files.
- "Build systems" address both of these issues:
  - "Make" is the oldest (1976, part of GNU Autotools). Some more "recent" ones include CMake (2000), SCons (2001), Meson (2013), etc.
  - Many newer projects have been scrapped to focus on supporting more mature projects.

What does a Makefile do?

- “make” is a terminal program (should be installed by default on WSL, clear.rice.edu, Mac, etc) which runs a “Makefile”
  - “make” tracks *if a specific file has been edited* since the last time the project was compiled and built
  - A “Makefile” specifies *dependencies,* e.g., what files need to be compiled in order to build a project.
- Specifying what depends on what in a Makefile can be somewhat tricky syntactically.

A very simple program

- We'll use this very simple program to illustrate how to construct a Makefile.
  - Will go through several versions to incrementally add features.
- To start, the Makefile must be called "Makefile". We'll put it in the same directory as all the files.

```c
#include "hello_make.h"

int main() {
  // call a function in another file
  my_hello();
}
```

hello_make.c

```c
/*
example include file
*/

void my_hello(void);
```

hello_make.h

```c
#include <stdio.h>
#include "hello_make.h"

void my_hello(void) {
  printf("Hello makefiles!\n");
}
```

hello_func.c

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

A very simple Makefile (version 1)

- The main thing in a Makefile are “targets” and “dependencies”

```
hello_make: hello_make.c hello_func.c
        gcc -o hello_make hello_make.c hello_func.c
```

  - Here, the executable “hello_make” is the *dependency*
  - Everything to the right of “hello make: …” are *targets*
  - *Below* the target/dependency is the compile command.
- Keep in mind that Makefiles are sensitive to whitespace vs tabs, etc! The space before “gcc” must be a **tab.**

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

A very simple Makefile (version 1), cont.

- What happens when we run this Makefile via “make”?

```
hello_make: hello_make.c hello_func.c
        gcc -o hello_make hello_make.c hello_func.c
```

- What happens when we
  - Modify hello_make.c or hello_func.c?
  - Delete “hello_make”?
  - Run “make hello_make”?

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

A simple Makefile (version 2)

- We recompile everything every time a single file changes. Can we improve this?
- Solution: add “hello_func.o” and “hello_make.o” as targets, and have “hello_make” depend on those.

```
hello_make: hello_make.o hello_func.o
        gcc -o hello_make hello_make.o hello_func.o

hello_make.o: hello_make.c
        gcc -c hello_make.c

hello_func.o: hello_func.c
        gcc -c hello_func.c
```

```
CC = gcc
hello_make: hello_make.o hello_func.o
        $(CC) -o hello_make hello_make.o hello_func.o
```

Note that this Makefile does the same as the above due to the *implicit rules* in “make”

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Makefile's default and implicit behavior

- Two things to note:
  - "make" runs the *first* target by default only.
  - "make" has *implicit behavior* for ".o" and ".c" files - they will be compiled using the *C compiler* "$(CC)" implicit variable.
- "CC" defaults to the "cc" program. We can override this by setting "CC = gcc" and referencing it via "$(CC)".
  - This sets "gcc" as the default C compiler.
- We will try to not to rely on implicit Makefile behavior in this class.

A simple Makefile (version 3)

- One flaw of our current Makefile: things don't get updated if we make changes to "hello_make.h"!
  - Header files aren't compiled targets, so how do we fix this?
  - Solution: just add "hello_make.h" as a dependency explicitly

```
CC = gcc
DEPS = hello_make.h
hello_make: hello_make.o hello_func.o $(DEPS)
        $(CC) -o hello_make hello_make.o hello_func.o
```

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

A simple Makefile (version 4)

- One flaw of our current Makefile: things don't get updated if we make changes to "hello_make.h"!
  - Header files aren't compiled targets, so how do we fix this?
  - Solution: just add "hello_make.h" as a dependency explicitly

```
CC = gcc
CFLAGS = -I./
DEPS = hello_make.h
hello_make: hello_make.o hello_func.o $(DEPS)
        $(CC) -o hello_make hello_make.o hello_func.o $(CFLAGS)
```

We also set the "CFLAGS" variable so we can specify the include path.

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Fancier Makefiles: “make clean” and objects

- Can the Makefile “clean up” or “reset” the project?

```
INC_DIR = ./
CC = gcc
CFLAGS = -I$(INC_DIR)
DEPS = hello_make.h
OBJ_FILES = hello_make.o hello_func.o
hello_make: $(OBJ_FILES) $(DEPS)
        $(CC) $(CFLAGS) -o hello_make $(OBJ_FILES)

clean:
        rm -f hello_make *.o
```

- Here, we also moved “*.o” files into an “OBJ_FILES” variable

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Pattern-matching with “make”

- If you add files to a project, it would be nice if your Makefile could track them automatically. Some examples:
  - Can we automatically add all header files in some “include” directory as dependencies (e.g., to the “DEPS” list)?
  - What if we wanted to automatically tell the Makefile to compile all “.c” files in a specific directory?
  - What if we wanted to clean up compilation by moving all the compiled “.o” files to an “obj” directory?

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Special symbols for Makefiles

- Makefiles can perform pattern recognition, wildcards (similar to a Linux terminal).
  - Instead of specifying individual files, we can specify general rules so that *any file* that gets added to a directory is *automatically compiled*.
- Makefile has special symbols (e.g., $@, $^, $%)
  - “$@”, “$^”, “$<” refer to the target, all dependencies, and only the first dependency.

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Special symbols/keywords for Makefiles

- Example: “$@” refers to the target:

```
hello_make: $(OBJ_FILES) $(DEPS)
        $(CC) $(CFLAGS) -o hello_make $(OBJ_FILES)
```

↓

```
hello_make: $(OBJ_FILES) $(DEPS)
        $(CC) $(CFLAGS) -o $@ $(OBJ_FILES)
```

- The “wildcard” keyword in Make behaves like a Linux wildcard

```
INC_DIR = ./
INC_FILES := $(wildcard $(INC_DIR)/*.h)
DEPS = $(INC_FILES)
```

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Pattern substitution in Makefiles

- Can use the “wildcard” keyword to determine all “.c” files, but we need to use the list of “.c” files to determine “.o” files too.

```
INC_DIR = ./
SRC_DIR = ./
INC_FILES := $(wildcard $(INC_DIR)/*.h)
SRC_FILES := $(wildcard $(SRC_DIR)/*.c)
OBJ_FILES = hello_make.o hello_func.o # ???
```

- Can use the “patsubst” (pattern substitution) keyword:

```
OBJ_DIR = ./
OBJ_FILES := $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRC_FILES))
```

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

A more advanced Makefile

- Automatically detects and compiles “.c”, “.o” files

```
CC = gcc
INC_DIR = ./
SRC_DIR = ./
OBJ_DIR = ./
INC_FILES := $(wildcard $(INC_DIR)/*.h)
SRC_FILES := $(wildcard $(SRC_DIR)/*.c)
OBJ_FILES := $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRC_FILES))

DEPS = $(INC_FILES)
CFLAGS = -I$(INC_DIR)

hello_make: $(OBJ_FILES) $(DEPS)
        $(CC) $(CFLAGS) -o $@ $(OBJ_FILES)

clean:
        rm -f hello_make *.o
```

- However, Make can’t tell that “.c” and “.o” files depend on $(DEPS)

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

When to avoid implicit rules in Makefiles

- Implicit rules in Makefiles are convenient, but if you have header dependencies, they aren't taken into account.
- It's not hard to replace an implicit rule with an explicit rule using Makefile special symbols, but it is ugly…

```
# "@" refers to the target (left hand side)
# "<" refers to the first prerequisite.
# If we use "^", we build all prereqs, including $(DEPS).
# If we use "<", we don't build the $(DEPS) prereq.
$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c $(DEPS)
        $(CC) -c $< -o $@ $(CFLAGS)
```

This specifies that all ".o" files are targets, depend on both the corresponding ".c" file and DEPS, and uses a custom build command.

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

First vs final makefile (only specifies one thing)

```
CC = gcc
INC_DIR = ./
SRC_DIR = ./
OBJ_DIR = ./
INC_FILES := $(wildcard $(INC_DIR)/*.h)
SRC_FILES := $(wildcard $(SRC_DIR)/*.c)
OBJ_FILES := $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRC_FILES))
```

```
hello_make: hello_make.c hello_func.c
        gcc -o hello_make hello_make.c hello_func.c
```

```
hello_make: $(OBJ_FILES) $(DEPS)
        $(CC) $(CFLAGS) -o $@ $(OBJ_FILES)

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c $(DEPS)
        $(CC) -c $< -o $@ $(CFLAGS)

clean:
        rm -f hello_make $(OBJ_DIR)/*.o
```

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Final makefile with directory structure

```
CC = gcc
INC_DIR = ./include
SRC_DIR = ./src
OBJ_DIR = ./obj
INC_FILES := $(wildcard $(INC_DIR)/*.h)
SRC_FILES := $(wildcard $(SRC_DIR)/*.c)
OBJ_FILES := $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRC_FILES))

DEPS = $(INC_FILES)
CFLAGS = -I$(INC_DIR)

hello_make: $(OBJ_FILES) $(DEPS)
        $(CC) $(CFLAGS) -o $@ $(OBJ_FILES)

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c $(DEPS)
        $(CC) -c $< -o $@ $(CFLAGS)

clean:
        rm -f hello_make $(OBJ_DIR)/*.o
```

Example adapted from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

More on ".o" files, creating and linking libraries

- Users can *link* external libraries during the linking process
  - Static libraries are just bundled collections of ".o" files
  - Can run "ar r libsource_name.a source_name.o" to create a library called "libsource_name"
- Can link a library with "gcc … -L. -lsource_name
  - "-L" specifies the library path, "-lsource_name" tells gcc to link a library called "libsource_name.a" (the "lib" and ".a" are implied)

* the header for the library must be included in your code in order to use it

Linear algebra libraries: BLAS and LAPACK

- BLAS: **B**asic **L**inear **A**lgebra **S**ubroutines
  - Low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication.
  - Often 10-100x faster than naive implementations.
- LAPACK: **L**inear **A**lgebra **Pack**age
  - Routines for solving systems, linear least squares, eigenvalue problems, SVD, matrix factorizations such as LU, QR, Cholesky and Schur decomposition.

* Originally made for Fortran so using routines/functions in practice takes extra work

Downloading Libraries and Packages

- Homebrew is a package manager for Linux/MacOS
  - Simplifies downloading and managing packages (such as lapack)
  - Installs packages to their own directory and then symlinks their files into /opt/homebrew (on Apple Silicon).

- For Linux/MacOS one can use 'sudo apt-get lapack-dev libblas-dev'
- For Linus/MacOS with homebrew one can use 'brew install lapack'
- Windows is more complicated and involves building from source code

Using Lapack in C

- The appropriate header file is included, '# include <lapacke.h>'
- Compiled using 'gcc *filename*.c -I/*header-file-path* -llapack -o *exectuablename*'
- Depending on installation, you may also need to tell it the path to the library

- All of this could be in a MakeFile

- List and description of member function/routines can be found in the online documentation: https://www.netlib.org/lapack/explore-html/topics.html

Miscellaneous: the “const” keyword

- C allows you to specify that a variable is “const”, e.g., cannot be changed (e.g., for safety).
- Example: “const int x = 1” means you cannot assign a new value to “x”. If you try, you get a compiler error.
- Const is a bit weird with pointers; you can have a pointer to a constant variable, or a constant pointer.

Rules for “const”, application to pointers

- You can put “const” before or after the type
  - In general, “const” refers to the first type to its left.
  - Exception: if const is first, it applies to the first part.
- What do these do? See “const_keyword.c” for a demo.
  - “**int * const ptr**”: a constant pointer to an integer
  - “**const int * ptr**”: a pointer to a constant int
  - “**const int * const ptr**”: a constant pointer to a constant int.

Rules for “const”, application to pointers

- Which commands are allowed by “const” and which aren’t?
- In practice, “const” is often used to specify function inputs that should not be modified.
  - This adds an extra layer of safety when passing pointers in to functions.

```c
int x = 1;
int y = 2;

const int * ptr_1 = &x;
*ptr_1 = 2; // which works?
ptr_1 = &y;
printf("*ptr_1 = %d\n", *ptr_1);

// will this work?
const int * z = malloc(sizeof(int) * 10);
z[0] = 10;

int * const ptr_2 = &x;
*ptr_2 = 2; // which works?
ptr_2 = &y;
printf("*ptr_2 = %d\n", *ptr_2);
```