Typical Program Layout

- Most of these sections can be left out (depending on your application).
- In the most common cases, main *must* exist
- Function prototypes *can* also include the implementation, but best practice is to only have prototypes before main and implementations after.
- A slightly different layout will be covered later.

Common Libraries

- stdio.h provides most/all of the expected Input/Output routines
  - more often than not, you'll need these to debug your program or monitor its progress
- stdlib.h provides numerous miscellaneous routines from pseudo-random number generation to memory management to sorting/searching and more
- math.h provides [almost] everything you might need to do basic-intermediate math.
  - trig, exponents, rounding, etc.
  - In scientific computing, however, more advanced libraries are usually needed
- mpi.h provides all of the routines in a given Message Passing Interface implementation
  - point-to-point communication, collectives, buffer management, etc.

Canonical Hello World

- Traditional "Hello World!" Includes stdio.h for writing to stdout. Begin the program (main) and print a literal string.
  - As an aside, "puts" is an output function akin to printf with no format processing.
  - As such, most modern compilers are smart enough to turn printf("Hello World!\n") into puts("Hello World!"), for example.

Compiling and Running

- "gcc" is the name of the compiler. Other options include cc, icc, pgCC, CC, mpicc, and many others
- "-o hello_world" specifies that the source is complied into the output file "hello_world"
- "hello_world.c" specifies the name of the source file to compile
- "./hello_world" tells the shell to run the program called "hello_world" located in the current directory

Variables

- C provides many data types, these three classes just provide the basic, most often used types

Arrays

- The examples illustrated here create *static* arrays. That is, their size is fixed and those sizes must be known at compile time (e.g. you can't create an array in this manner based on user input).

- Dynamic arrays are introduced (in part) later in this presentation.

Conditionals and Loops

- Conditionals, a.k.a branching, provides a structure for executing certain sections of code only in certain circumstances. There are a few different methods for branching: *a*) traditional if-else; *b*) ternary, e.g. `y = (x == 0 ? NAN : 1/x)`, where the division by x will only occur if x != 0; *c*) switches (example below); *d*) as well as a few others.

```
switch (var) {
  case 'a':
    puts("Do this only when var == a");
    break;
  case 'b':
  case 'c':
    puts("Do this only when var == b OR c");
    break;
  default:
    puts("Do this in all other cases");
}
```

Conditionals and Loops (cont...)

- Loops, of every kind, are used to execute the same body of code more than once. The most common kinds of loops are **for** loops and **while** loops.
  - For loops are most often used when the body of the loop needs to know which iteration is currently being executed (as in initializing an array), or when the number of iterations is known before-hand.
  - For loops have three "parameters," separated by semicolons: the counter's starting value; the condition that must be true in order to loop; what to do to the counter after each iteration.
  - While loops can be thought of as a for loop's second parameter: there is no notion of a "built-in" counter, only some condition that must be met.

Argument Processing

- The main function traditionally takes two arguments, argc ("ARGument Count") and argv ("ARGument Vector" or "ARGument Values")
- argv always has at least one element (argc is always ≥ 1) which is set to the name of the running program. In our hello_world example above, this means argv[0] is set to "./hello_world".
- All arguments come in a character strings and therefore must be converted to the appropriate types for use later in the program.
  - (int)strtol(...) is the updated version of "atoi". The first argument is the string to convert, the second is an output argument and isn't normally used, the third is the base to convert to (10 is often called "human readable")
  - strtof(...) is identical to strtol but converts to a float whereas (int)strtol converts to a long and then casts it as an int

Pointers and Memory Allocation

Pointers provide special ways of declaring and using variables. They allow you to pass the location (address) of a variable to functions rather than a copy of its value. In this manner, you can create functions with "output arguments," allowing changes to those arguments made *within* the function to persist after the function has returned.

- "int * i" declares a pointer (* i) to a location in memory which stores values of some type (int).
- "malloc(...)" is the function that allocates memory. This memory comes from what's called the *heap*. Normally declared variables (e.g. "int i = 0") are allocated on the *stack*.
  - malloc takes one argument, the number of bytes to allocate. By passing "sizeof(int)", we circumvent the need to know how many bytes a single integer uses on this particular machine.
  - malloc returns a pointer to a chunk of memory of type NULL. By prepending (int*) to the call, we cast it to a pointer to an integer and can use the resulting chunk of memory accordingly.

Pointers and Memory allocation (cont...)

- Allocating memory for single dimensional arrays is straightforward. Arrays are simply pointers to the start of the array.
  - "int * a" creates a pointer to an integer; "(int*)malloc(sizeof(int)*ELEMENTS)" says there should be ELEMENTS number of integers starting at this location in memory.
- Multi-dimensional arrays are trickier. A 2-Dimensional array is actually a pointer to a set of pointers to integers.
  - The first step, therefore, is to allocate a 1-Dimensional meant to hold pointers to integers. This is what (int**)malloc(sizeof(*int)*ROWS) does.
  - Next, each of those pointers must be allocated as if they themselves were also 1-Dimensional arrays; each points to COLUMNS number of integers. This is what the for loop accomplishes.

Functions

This short program calculates the factorial of a specified number. There are a few important things to note here:

1. The function prototype does not need to specify the name of the argument variables, only their type and order/quantity.
   - If you were, however, to provide the implementation here as well, then you *do* need the name of the arguments

2. A more complete version of the code would take the number from the command line, verify that it is a number, otherwise set it to a default value, and do more error checking for overflow issues. For brevity, to fit on a single slide, these features have been omitted.

3. The ternary operator is used here. "`(condition) ? (value if true) : (value if false)`" is roughly equivelant to:

```
if (condition) {
  // commands if true
} else {
  // commands if false
}
```

4. This is a tail-recursive implementation of factorial, meaning the function operates by calling itself and can be rewritten iteratively as such:

```
double factorial(int n) {
  int i;
  double ans = 1;
  for (i = n; i > 1; --i)
    ans *= i;
  return ans;
}
```

Custom Header Files

- Header files provide a way to define custom libraries, reference code in different source files, or simply clean up your primary source file.

- Here, we've placed the factorial function in a header file called fact.h. The double quotes around the filename specifies that fact.h is a custom header rather than a system header. This causes the compiler to look for the header in places like the current directory.

- Header files can simply have more C code in them, containing almost anything except another main.

- Compiling an executable from this example is identical to compiling a single source (`$ gcc -o fact fact.c`)
  - Other methods of using header files may require intermediate steps (e.g. building object files)

Makefiles

This simple Makefile example illustrates one possibility for a Makefile to compile a couple of the source examples used in this presentation.

- Variables can be defined by NAME=VALUE and are later referenced using $(NAME)
  - It is standard practice to define the compiler you're using as a variable as this makes changing it later very easy
  - Common uses for Makefile variables include alternate or additional libraries, build directories, compiler options, and more
- Any text from a '#' to the end of the line is a comment
- Targets specify "jobs" to do, kind of like functions or subroutines. They are defined with `target_name:` at the beginning of the line.
  - Anything after the ':' is a "prerequisite" or "dependency" and corresponds to files that must exist in order for that target to execute any of its commands
  - Lines after the target must begin with a tab and specify commands to run for that target. Commands can be any kind of action and there can be any number of them