Annotated First C Programs

Things to know before writing first C program

- Everything you'll learn this semester about C programming language is relevant to the C++ programming language that you'll learn next semester. Since the course is concerned with the C subset of C++ programming language, concepts described in this and other documents are applicable not only to C but also to C++. If you follow lectures, tutorials, and assignments correctly, every C program you'll write this semester will also be a C++ program.
- Everything related to C in this course refers to the C11 standard of the language. Certain code that will be demonstrated throughout the semester might fail to compile in older C standards.
- C programs follow the von Neumann architecture. Both instructions and data of a program reside in memory. Programming is then an endeavor that requires programmers to write appropriate instructions that transform input data into required output data.
- A program consists of *source file(s)*.
- Source file(s) contain functions that communicate with other functions in the program by passing and returning values.
- A *function* encapsulates an algorithm. An *algorithm* is a finite sequence of instructions that operate on data. Think of a function as a black box that takes certain input, performs actions that transform the input, and then returns the transformed data as output to the function's caller.
- A C program is made up of functions which in turn are made up of statements. A *statement* is the atomic unit of a C program. Informally, statements in an algorithm correspond to C statements.
- Data manipulated by a function consists of two types: *literals* and *variables*.
    1. Literals express constant values such as `7` or `123.56` or a character such as `'a'` or a sequence of characters such as `"Hello"`.
    2. Variables identify named memory locations at which data of interest is located. There are two values associated with variables. The first is the physical memory address that variables represent. The second is the contents of these physical memory locations. Unlike other high-level programming languages, both the address and the value stored at the address are accessible to C programmers.
- All data, represented by both constants and variables, is *typed*. A *data type* specifies the set of values and the set of operations that can be applied on these values. The type specified by a programmer for a constant or a variable allows the compiler to translate to the machine the nature of the data stored and how the machine is to interpret the data. If a variable doesn't have a type associated with it, it will be impossible for C compilers to convey to the machine how to correctly interpret the contents of the memory locations associated with that variable. For this reason, both C and C++ are said to be *statically typed* languages. On the other hand, Python is an example of a *dynamically typed* language.
- Using decimal notation is natural for ten-fingered humans, but the language of a computer, called *machine language*, is a sequence of 0s and 1s. Each of the digits, 0 and 1, is called a *binary digit* or *bit*. A bit can be only a 0 or a 1 - never anything else, such as 2 or 3, 5 or 6 or A or B. This is a fundamental concept. Every piece of information stored in a computer or processed by a computer, whether it is your name, or your street address, or the amount you owe on your credit card, is stored as strings of 0s and 1s.

  In isolation, a single bit is not very useful since it can represent only two values. When groups of bits are combined together and some interpretation is applied that gives meaning to the different possible bit patterns, it becomes possible to represent the elements of any finite set. A sequence of bits is referred to as a *binary number*. For example, using a binary number system, groups of bits can be used to encode integers. By using a standard character code such as ASCII, the letters and symbols on a keyboard can be encoded as binary numbers to represent text in a document.

  Although humans can represent arbitrarily large numbers using the decimal, or octal, or some other number system, machines are only capable of representing binary numbers having specific number of bits. Machines don't let you collect together or process binary numbers having an arbitrary number of bits. Instead, machines represent instructions and data using binary numbers having certain fixed number of bits. Since

the early days of computing, a sequence of eight bits, called a *byte*, has become the de facto standard for an unit of digital information. A byte represents the smallest data item and the unit of storage in modern computers.

CPUs have evolved from 8 bits to incorporate binary numbers having 16, 32, and 64 bits. In addition, every CPU has a *word size*, indicating the number of bits in a binary number that can be atomically [that is, as a single unit] processed by the CPU. Most modern CPUs have a $64-$bit word size. Currently, for each specific bit size [8, 16, 32, and 64 bits], computers specify binary numbers using unsigned, signed, and floating-point representations:

- *Unsigned* representation for positive integers greater than or equal to 0 encoded in traditional binary form. For $8-$bits, $2^8 = 256$ unsigned numbers ranging from 0 to 255 can be represented. For $16-$bits, $2^{16} = 65{,}536$ unsigned numbers ranging from 0 to $65{,}535$ can be represented. And so on for $32-$ and $64-$bits.
- *Signed* representation for both positive and negative integers is encoded in *two's-complement* form. For $8-$bits, $2^8 = 256$ signed numbers ranging from -128 to 127 can be represented. For $16-$bits, $2^{16} = 65{,}536$ signed numbers ranging from $-32{,}768$ to $32{,}767$ can be represented. And so on for $32-$ and $64-$bits.
- *Floating-point* representation for rational numbers of the form $V = x \times 2^y$ encoded in the IEEE 754 format.

- The fundamental, numeric data types for $64-$bit C11 compiler used in this course are:

Bit size	Unsigned Integral Types	Signed Integral Types	Floating-Point Types
8-bit	`unsigned char`	`signed char`	-
16-bit	`unsigned short int`	`signed short int`	-
32-bit	`unsigned int`	`signed int`	`float`
64-bit	`unsigned long int`	`signed long int`	`double`
64-bit	`unsigned long long int`	`signed long long int`	-
128-bit	-	-	`long double`

  Note that types `signed short int`, `signed int`, `signed long int`, and `signed long long int` are equivalent to abbreviated types `short`, `int`, `long`, and `long long`, respectively. Similarly, `unsigned short int`, `unsigned long int`, and `unsigned long long int` are equivalent to abbreviated types `unsigned short`, `unsigned long`, and `unsigned long long`, respectively.

- In programming languages, the term *identifier* means the name given to variables, functions, macros, and so on. Except literals, everything in a C program such as variables and functions must have an identifier so that they can be uniquely *identified* by programmers and compilers.
- Identifiers in C consist of a sequence of Latin characters [`a` through `z`, `A` through `Z`], underscores `_`, and digits `0` through `9`. The first symbol of an identifier must be either a Latin character or an underscore. C identifiers are case sensitive. Valid identifiers are, for example, `monkey`, `Monkey`, `an_id_23`, `big_monkey`, `small_monkey`, `_the_monkey`, or `__special_monkey`. In general, it is never a good idea to name identifiers beginning with two underscore, as in `__bad_monkey` because C standard library uses double underscores and your identifier might clash with a system defined identifier.
- Before a variable or function is used in a program, their identifiers must first be *declared* to the compiler, and presumably to any human reader of the program. More specifically, the compiler must be provided with the type associated with the identifier. This allows the compiler to correctly translate to the machine the values represented by these identifiers. Note that ISO C11 standard specifies that variables can be declared anywhere in a function with the only caveat that they be declared before their first use.
- By default, every program is provided three input/output streams to interact with its environment: *standard input* [`stdin`] for the program to read input from the keyboard device by default, *standard output* [`stdout`] for the program to write output to the computer display screen by default, and *standard error* [`stderr`] for the program to write error or diagnostic messages to the computer display screen by default.
- Before continuing with this document, you must understand how compilers work. Compilation is a multi-stage process involving several tools, including the compiler itself `gcc`, the assembler `as`, and the linker `ld`. The complete set of tools used in the compilation process is known as the *compiler toolchain* or *compiler driver*. The sequence of commands executed by a single invocation of `gcc` consists of the following stages:

preprocessing, *compilation proper*, *assembly*, and *linking*.

- By default, GCC C compiler `gcc` expects source files to have `.c` suffix.
- This course will use the ISO C11 standard. Code presented in this document and throughout this semester may not compile in older C standards.

The minimal C program

Consider the following source file `nothing.c` that - as the name implies - does nothing:

```
1  // this is the minimal C program
2  int main(void)
3  {
4    return 0;
5  }
6
```

1. The double slash `//` on line 1 begins a *single-line comment* that extends to the end of the line. A comment is for consumption by the human reader and not the compiler. The preprocessor will strip away comments and replace them with single space character. This means that the compiler proper will never see comments in a source file.
2. Line 2 declares a function `main`.
    - Line 2 consists of three identifiers `int`, `main`, and `void`. An *identifier* is a sequence of characters used to denote names of variables, functions, and other entities such as macros and types. An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore. This page provides information and rules specific to C identifiers.
    - Not only are `int` and `void` identifiers, they're also C keywords. A *keyword* is a predefined identifier that has special meaning to C compilers.
    - Line 2 declares function `main`. Every C program uses function `main` as the entry point to the program and therefore there can only be *one and only one* function called `main` in a C program.
    - Recall that a function encapsulates an algorithm that transforms input value(s) to output value(s). Inputs to a function are specified between a pair of parentheses `(` and `)` as a sequence of comma-separated values called *function parameters*. Function `main` takes a single parameter of type `void` and returns a value of type `int`. Although functions can take as many parameters as dictated by the author, they can only return a single value.
    - Identifier `void` is a C keyword indicating it is a C data type specifying *no value*. The use of `void` data type in the current context where it is delimited within parentheses `(` and `)` indicates to the compiler and human readers that function `main` won't receive any data, or value, or information from the function that invoked it.
    - C++ standards allow programmers to skip the use of `void` in function parameters. That is, C++ compilers will implicitly assume that function `main` doesn't receive input values and has a `void` parameter if the function is written as:

      ```
      1  int main() // invalid in C - but valid in C++
      2  {
      3    return 0;
      4  }
      5
      ```

      However, C11 requires every function that doesn't receive input values to specify this fact to the compiler using `void`. Therefore, the above code will not compile with the C compiler and must be rewritten as

```
1  int main(void) // this is how this course will declare main
2  {
3    return 0;
4  }
5
```

- `int` is a C keyword indicating a data type that on both 32-bit and $64-$bit machines represents 32-bit signed integers ranging in value from -2147483648 (-2^{31}) to 2147483647 ($2^{31} - 1$). In the current context, the program's author is indicating to the compiler that function `main` will return a value of type `int` to the function that invoked it. Recall that type `int` is an abbreviated and equivalent form of the wordier type `signed int`.
- To summarize, line 2 of the source text *declares* identifier `main` to be a function that takes no values and returns a value of type `int` to the operating system.

3. Line 3:
   - Curly braces are used in C to group together stuff including all statements required to implement an algorithm.
   - Left curly brace `{` starts the *function body* or *code block* of function `main` and will contain statements that implement the algorithm encapsulated by function `main`. The right curly brace on line 5 `}` matches the left curly brace and ends the body of function `main`.
4. Line 4 contains a C *statement*.
   - Notice that statement(s) within a function are indented to easily identify to human readers that these indented statements constitute a code block. Unlike Python, C and C++ are free form and programmers have very few restrictions on how they present source code to compilers.
   - All statements are delimited by a semi-colon `;`. Here, statement `return 0;` will result in function `main` [and therefore the program] returning value `0` of type `int` to the operating system indicating that the program executed successfully. Note that the type of the value returned by the `return` statement [literal `0` has type `int`] matches the return type [`int`] specified on line 2.
   - C programs return nonzero values to indicate failure. Not every operating system makes use of that return value: Linux-based operating systems do, but Windows systems rarely do.
   - Add `return` to your list of C keywords - the others are `int` and `void`.
   - C11 and all standards of C++ allow programmers to skip the explicit `return` statement in function `main`. If there is no explicit `return` statement in function `main`, these C/C++ compilers will implicitly add statement `return 0;` to indicate successful completion. This means that the most minimal C/C++ program will look like this:

     ```
     1  int main(void)
     2  {
     3  }
     4
     ```

     For consistency with older C standards, my documented examples will always contain an explicit `return` statement.
5. Line 5 contains an empty line. According to ISO C standards, every source file must terminate with a newline.
6. C is a free form language implying that programmers have very few restrictions on how they present source code to compilers. For example, the entire source code in `nothing.c` can be written on a single line:

```
1  // this is the minimal C program
2  int main(void) { return 0; }
3
```

Writing code like this decreases the vertical spread of the source code while increasing its horizontal spread. Jamming several things into a single line makes code hard to read and maintain for programmers. Instead, I prefer a compromise that decreases vertical spread a little bit while increasing horizontal spread a little bit. My code will follow this template:

```
// this is the minimal C program
int main(void) {
  return 0;
}

```

7. Open a Windows command prompt. Change your current working directory to `C:\sandbox`. Switch to Linux bash shell using Window shell command `wsl`. Open Code from the bash shell using command: `code nothing.c`. Use Code to enter the code described in `nothing.c`. After saving the file, use GCC C compiler `gcc` to *compile* `nothing.c` (note that the `$` symbol below represents the Linux bash shell command prompt and is not part of the `gcc` command):

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c nothing.c
-o nothing.o
```

   - The six options to `gcc`: `-std=c11`, `-pedantic-errors`, `-Wstrict-prototypes`, `-Wall`, `-Wextra`, `-Werror` are required and necessary every time you compile a source file.
   - Option `-c` indicates that source file `nothing.c` is only to be compiled but not linked. That is, source file `nothing.c` is converted to an *object file* containing binary machine code for a specific CPU but not into an executable program.
   - Option `-o nothing.o` gives the name `nothing.o` to the output object file generated by the compiler. All though compilers will default the object file's name to the name of the source file with an extension of `.o`, I explicitly specify the name of the object file.
   - *Link* object file `nothing.o` with standard library object files to create an executable:

     ```
     $ gcc nothing.o -o nothing.out
     ```

     Since this particular example describes a minimal C program, there are no C standard library object files to be linked to the object file `nothing.o`. If option `-o` is not used, the linker will default the executable file to `a.out`.
   - To run executable program `nothing.out`, type the executable's pathname like this:

     ```
     $ ./nothing.out
     $
     ```

   - Repeat the compile and link steps using *verbose* option `-v`. This option prints the commands that execute the different compilation stages to standard output.

Second C program: Using the C standard library

Consider the code in source file `hello.c` that prints a greeting to `stdout`:

```c
#include <stdio.h>

/*
Print a greeting to the world!!!
*/
int main(void) {
  printf("Hello World!!!\n");

  return 0;
}

```

1. Let's start with line 1 containing unusual characters `#`, `<`, and `>`: `#include <stdio.h>`
   - The *preprocessor* is a program used by the C compiler to provide certain text utility functionalities. Think of the C preprocessor's behavior and capabilities as being similar to a specialized text editor.
   - Identifier `include` is a directive to the preprocessor to replace line 1 with contents of file `stdio.h`.
   - Delimiters `<` and `>` tell the preprocessor to begin searching for file `stdio.h` in standard include paths that were established when the compiler was installed on a computer. The search will conclude in the current working directory of source file `hello.c`.
   - What is the purpose behind including file `stdio.h`? This file is supplied by the compiler vendor and contains *declarations* of input and output functions defined in C standard library. Recall that C itself is a fairly small language and it is up to the C standard library to provide input/output capabilities to C programs. Specific to source file `hello.c`, `stdio.h` contains a *declaration* of function `printf` that prints text to `stdout`. Files such as `stdio.h` are called *header files* because C requires identifiers be declared before their first use and related identifiers are collected in a file which is conveniently added at the *head* or top of the source file so that these identifiers are visible throughout the source file.
   - A *function declaration* introduces a reference to a function defined elsewhere (in this case, in the C standard library) by specifying the function's name, its parameter list, and its return type. A declaration for function `printf` is required in source file `hello.c` for the compiler to ensure the call to `printf` matches the number and types of parameters in the declaration of `printf` and further ensure that the return value from `printf` is used correctly by the caller in `hello.c`. Note that the code in `hello.c` ignores the return value from function `printf`. In short, the compiler will use the function declaration of `printf` to check for the correct use of function `printf` in source file `hello.c`.
   - Function declarations as discussed in this document and throughout this course are known as *function prototypes* to distinguish them from an older style of function declarations in which the parameter list is left empty. This course will leave this bit of ugly history behind and instead follow C++ terminology: the terms *function declaration* and *function prototype* are considered synonyms and the term function declaration will be used in lieu of function prototype.
2. Line 2 is a newline which is left untouched by the preprocessor. The *space*, *tab*, and *newline* are collectively known as *whitespace characters*. These characters are ignored [there are a few cases where they're not ignored but let's not dwell on those obscure details at this early stage] and their main purpose is to provide punctuation.
3. Lines 3, 4, and 5:
   - Lines 3, 4, and 5 specify a *multi-line comment*. A multi-line comment is program text delimited by characters `/*` and characters `*/`. A multi-line comment can be on a line by itself, or it can be on the same line as a statement, or can extend over several lines.
   - Comments are only meant for consumption by human readers - they're supposed to provide the reader with a clear and easy-to-understand description of what is the algorithm and how the algorithm is being implemented by sequences of statements. Although comments are optional, good style requires comments be used throughout a program to improve its readability and to document the algorithm.
   - Since comments don't have meaning to a compiler, C standards require the preprocessor to strip the source file of comments by replacing them with single space characters.
4. Line 6:
   - The declaration of function `main` has been seen in an earlier program. Now, let's expand on the concept of declarations by introducing definitions. Line 6 begins the *declaration* and also the *definition* of function `main`.

- Think of a *function definition* as the physical manifestation of what is described in a function declaration. While a function declaration tells the compiler [and presumably human readers] about the function's name, list of function parameters, and type of value returned by the function, a *function definition* creates memory storage for the instructions necessary to implement the algorithm that is encapsulated by the function and defines its parameters and return value. The following code snippet illustrates the difference between a function declaration and function definition:

```
/*
this is a function declaration (prototype): it tells the
compiler that inc is a function that takes a parameter of
type int and returns a value of type int
*/
int inc(int x);

/*
this is a function definition: it not only tells the compiler
that inc is a function that takes a parameter of type int and
returns a value of type int; in addition a function definition
implements the function using statements.
*/
int inc(int x) {
  return x+1;
}

```

- All C programs must define a single function named `main`. This function will become the entry point of the program - that is, `main` will be the first function executed when the program is started. Returning from this function terminates the program, and the returned value is treated as an indication of program success or failure.
- Any time you want to group things in C you put these things between opening curly brace `{` and closing curly brace `}`. Compilers understand these braces as *punctuator* symbols that enclose these things. With functions, the statements implementing an algorithm are the things that must be enclosed between `{` and `}`. Notice that unlike previous programs, left curly brace `{` is not present in its own line but is instead separated from closing parenthesis `)` by whitespace. All subsequent statements until a corresponding right curly brace `}` specify the function's body. In fact, left curly brace `{` need not be separated from the closing parenthesis `)` by whitespace nor does the first statement have to be whitespace separated from the preceding `{`:

```
#include <stdio.h>
/*Print a greeting to the world!!!*/
int main(void){printf("Hello World!!!\n");return 0;}

```

  C is a free form language that only requires whitespace to provide punctuation. For example, on line 3 of the above code, the compiler will require a whitespace between `int` and `main` to distinguish these two discrete identifiers. Otherwise, `intmain` will be interpreted as a single identifier.

  The following text is also valid C syntax:

```
#include <stdio.h>
/*Print a greeting to the world!!!*/int main
(void){printf("Hello World!!!\n");return 0;}

```

  There doesn't need to be whitespace between `{` and `printf` because the compiler understands `{` to be a *punctuator* symbol that indicates the start of a new block of code and the subsequent identifier `printf` is considered as part of the first statement in this new block of code.

5. Line 7 begins with identifier `printf`.

- Since identifier `printf` is followed by left parenthesis `(`, followed by a bunch of stuff, followed by right parenthesis `)`, the compiler will understand that function `printf` is being called. This function is defined in the C standard library and is used for printing formatted data to standard output.
- Between the left and right parentheses, the text enclosed in a pair of double quotes `"Hello World!!!\n"` is called a string literal. A *string literal* is a sequence of characters delimited by a pair of double quotes `"`.
- The string literal `"Hello World!!!\n"` is the *argument* or value passed to function `printf`.
- You might expect the sequence of characters `Hello World!!!\n` to be printed to `stdout` by function `printf`. However, that is not the case - only the sequence of characters `"Hello World!!!"` are printed to `stdout` on the first line followed by a newline. The backslash `'\'` is called an *escape character* when it is used in a string. The compiler combines it with the character that follows it and then attaches a special meaning to the combination of characters. For example, `\n` represents a skip to newline. The cursor is a moving place marker that indicates the next position in `stdout` [on the screen, for example] where information will be displayed. When executing a `printf` function call, the cursor is advanced to the start of the next line in `stdout` if the `\n` escape sequence is encountered in the string passed to the function. A `printf` string often ends with a `\n` newline escape sequence so that the call to `printf` produces a completed line of output. If no characters are printed on the next line before another newline character is printed, a blank line will appear in the output. For example, the calls

```
printf("today is a good day.\n");
printf("\ntomorrow will be a better day.\n");
```

produce two lines of text with a blank line in between:

```
today is a good day.

tomorrow will be a better day.
```

The first call to `printf` places the cursor at the start of line 2. Since `\n` newline escape sequence is the first character of the string in the argument to the second call to `printf`, the cursor will move to the start of line 3. Because the second call to `printf` also terminates with `\n`, the cursor will print the characters `tomorrow will be a better day.` on line 3 and then shift to the start of line 4.

In addition to `\n`, a number of additional escape characters are recognized. For example, the sequence `\\` is used to insert a single backslash in a string, and the sequence `\"` will insert a double quote in a string. Thus, the call to function `printf`:

```
printf("\"The End.\"\n");
```

will print to `stdout` a line containing

```
"The End."
```

Some of the commonly used escape characters recognized by C are:

Sequence	Character Represented
\a	alert (bell) character
\b	backspace
\n	new line
\t	horizontal tab
\\	backslash
\'	single quote
\"	double quote

- In short, function `main` is calling function `printf` with string literal `"Hello World!!!\n"` as argument that is printed to `stdout` by `printf` as a line containing `Hello World!!!` followed by a newline.

6. Line 8 represents a newline that was introduced by the programmer to make the code more readable to other human readers. As indicated earlier, a newline is left untouched by the preprocessor.
7. Line 9 contains statement `return 0;` which indicates that function `main` is returning a value 0 of type `int`. Since the definition of function `main` on line 6 indicates that `main` is a function that takes no value and returns a value of type `int`, the function ***must*** return a value of type `int`.
8. Line 10 contains right curly brace `}` which matches left curly brace `{` on line 6 indicating the end of the body of function `main`.
9. Line 11 is empty because every C source file is required to terminate with a newline.
10. Compile source file `hello.c` using GCC C compiler with the full suite of options:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c hello.c -o
hello.o
```

11. Link object file `hello.o` with C standard library to create executable file `hello.out`:

```
$ gcc hello.o -o hello.out
```

12. To run executable program `hello.out`, type the executable's pathname like this:

```
$ ./hello.out
Hello World!!!

```

Things to try: Deciphering messages from compiler

1. What happens when you insert characters `/*` at the start of line 4 in source file `hello.c`?
2. What happens when you add characters `/*` at the beginning and characters `*/` at the end of line 4 in the above code?
3. What happens when you add characters `//` at the beginning of line 4 in the above code?

   It is clear that C11 prohibits the nesting of multi-line comments. However, single-line comments can be nested inside multi-line comments.
4. Compile the source file after commenting line 9? Does the compiler generate diagnostic messages? Do you understand why the compiler doesn't generate diagnostic messages?

Third C program: Multiple source files

1. Now, we'll reinforce basic concepts crucial to understanding the structure, configuration, and vocabulary of C programs: *declarations* and *definitions*. This is done by deconstructing the *Hello World!!!* program into a function `hello` that prints `Hello World!!!` to `stdout` and a client function `main` that uses the services of function `hello` by making a call to it. This deconstruction consisting of two functions `main` and `hello` is implemented using two source files `driver.c` and `hello-defn.c`, and a header file `hello-decl.h`. It is further assumed that files `hello-defn.c` and `hello-decl.h` are authored by one programmer while `driver.c` is implemented by a second programmer and neither programmer is aware of the other.
2. Since function `hello` will be used by other parts of the program and function `hello` in turn calls C standard library function `printf`, a header file `hello-decl.h` is created to *declare* these two functions. This is the purpose of header files - to gather together declarations of related entities in a file and provide clients the service of including this header file rather than having to individually declare each entity. More specifically, line 4 in the following header file contains the *declaration* of function `hello`. As described earlier, a *function declaration* specifies the function's name, its parameter list, and its return type. The compiler will use this function declaration to check for the correct use of function `hello` by clients that wish to use this function in their source files. File `hello-decl.h` will look like this:

```c
#include <stdio.h>

// declaration (prototype) of function hello
void hello(void);

```

3. Source file `hello-defn.c` containing the definition of function `hello` will look like this:

```c
#include "hello-decl.h"

// definition of function hello
void hello(void) {
  printf("Hello World!!!\n");
}

```

Notice that line 1 of source file `hello-defn.c` contains a preprocessor directive to include header file `hello-decl.h`. This is a recommended practice to ensure that both the function declarations in the header file and their definitions in the source file match up. Also notice that line 1 has an `include` directive that delimits the name of the header file `hello-decl.h` between a pair of double quotes `"`. When the header file is delimited by angle brackets `<>`, the preprocessor will search for the header file in the standard include paths of the compiler. Now, with a pair of double quotes `"` used as delimiters, the preprocessor will *only* search for the header file in the current directory in which source file `hello-defn.c` is located.

The author of file `hello-defn.c` now has everything required to successfully compile the source file to an object file:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c hello-defn.c -o hello-defn.o
```

4. Independently, the second programmer has defined function `main` in a separate file `driver.c`:

```c
#include "hello-decl.h"

int main(void) {
  hello();

  return 0;
}

```

Similar to source file `hello-defn.c`, `driver.c` has an `include` directive on line 1 that delimits the name of header file `hello-decl.h` between a pair of double quotes `"`. When `hello-decl.h` is included in a source file such as `driver.c`, the preprocessor will replace the line containing `include` directive with contents of file `hello-decl.h`. The transformed `driver.c` will look like this:

```c
#include <stdio.h>

void hello(void);

int main(void) {
  hello();

  return 0;
}

```

The preprocessor will notice that the transformed version of `driver.c` again contains an `include` directive and will further transform the previously transformed file by replacing line 1 of transformed `driver.c` with the contents of file `stdio.h`. This process will recursively continue if `stdio.h` itself contains `include` directives. The transformed version `driver.c` will look like this:

```
contents of stdio.h here ...

void hello(void);

int main(void) {
  hello();

  return 0;
}

```

The author of file `driver.c` now has everything required to create an object file without access to the source file containing the definition of function `hello`:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c driver.c -
o driver.o
```

5. Either of the two programmers or any other programmer can now link together function `main` in object file `driver.o`, function `hello` in object file `hello-defn.o`, and C standard library function `printf` defined in static C standard library `libc.a` into a single executable, say `new-hello.out`:

```
$ gcc driver.o hello-defn.o -o new-hello.out
```

6. To run executable program `new-hello.out`, type the executable's pathname like this:

```
$ ./new-hello.out
Hello World!!!

```

7. One way to think of `hello-decl.h` and `hello-defn.c` is that `hello-decl.h` is an *interface file* while `hello-defn.c` is an *implementation file*. Other programmers include interface files in their source files to call functions declared in these interface files without either programmers nor compilers aware of the implementation details of these functions. Instead programmers and compilers are only interested in whether these declared functions are referenced or called correctly in these other source files. It is the linker that will ensure that the implementation details of these declared functions are integrated with the functions in these other source files to create a synergistic and whole execution file.

Things to try:

1. What happens when you try to compile `driver.c` by commenting out line 1?

   If line 1 of `driver.c` is removed, that is, if `driver.c` doesn't include header file `hello-decl.h`, the compiler will *implicitly* assume that function `hello` is declared as:

```
int hello();

```

   This declaration says that `hello` is a function that takes an *unknown number* of parameters and returns an `int`. That is, if the declaration of function `hello` is not present in the source file, the compiler will implicitly assume that `hello` is declared as `int hello();` and print a diagnostic message to indicate this assumption:

```
$ gcc -std=c11 -Wstrict-prototypes -Wall -Wextra -c driver.c -o driver.o
driver.c: In function ‘main’:
driver.c:4:3: warning: implicit declaration of function ‘hello’ [-Wimplicit-function-declaration]
    4 |   hello();
      |   ^~~~~
```

Notice that source file `driver.c` is successfully compiled into object file `driver.o`. In fact, this object file `driver.o` can be linked with object file `hello-defn.o` and C standard library to create an executable that prints the required greeting to `stdout`:

```
$ gcc driver.o hello-defn.o -o new-hello.out
$ ./new-hello.out
Hello World!!!
```

All though the executable seems to run correctly, remember that the compiler has made a spurious assumption in the absence of an explicit declaration of function `hello` that `hello` is a function that takes an unknown number of parameters and returns an `int` even though `hello` is defined as a function that takes zero parameters and returns nothing. This particular behavior of C means that it is possible to create runtime security holes in the program by authoring functions that call function `hello` with arguments. C++ designers correctly identified this drawback of C as a serious security threat and therefore C++ was designed to flag omissions of function declarations as errors. To ensure C compilers can compile C code developed many decades earlier, C standards are unable to flag omissions of function declarations as errors.

The C code created in this course must compile cleanly with C++ compilers and thus the code must explicitly avoid certain drawbacks of C standards. Therefore, it is important that you always declare every function you use in a source file before its first use so that the compiler is not allowed to make implicit assumptions that can cause runtime bugs and security holes. You enforce this policy using `gcc` compiler option `-pedantic-errors` which will prevent such implicit declarations to be just passed off with diagnostic warnings:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -c driver.c -o driver.o
driver.c: In function ‘main’:
driver.c:4:3: error: implicit declaration of function ‘hello’ [-Wimplicit-function-declaration]
    4 |   hello();
      |   ^~~~~
```

Notice how option `-pedantic-errors` has converted a diagnostic warning to an error, thereby preventing the object file from being created.

2. In the previous question, you saw an example of compiler error. To distinguish between compiler errors and linked errors, compile and link `driver.c` using option `-Werror` like this:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror driver.c
```

What happens when you replace the pair of double quotes delimiters `#include "hello-decl.h"` in line 1 of `driver.c` with angled braces `<` and `>`, as in `#include <hello-decl.h>`?

3. Likewise, what happens when you replace the angled brace delimiters `#include <stdio.h>` in line 1 of `hello-decl.h` with pair of double quotes delimiters `#include "stdio.h"`?

Fourth C program: Mathematical functions and linking with C standard math library

Arithmetic expressions that solve science and engineering problems often require computations beyond basic addition, subtraction, multiplication, and division. Much problem solving requires the use of exponentiation, logarithms, exponentials, and trigonometric functions. This section introduces mathematical functions available in the C standard library.

The process begins with the use of the following preprocessor directive in any source file referencing mathematical functions in the C standard library:

```
#include <math.h>
```

This directive specifies that function prototypes and macros be added to the source file to aid the compiler when it converts calls to mathematical functions in the C standard library.

If your algorithms involve extensive math operations then you should look up the wide variety of functions prototyped in `math.h` to see which functionality is already implemented and what you may have to build from scratch. There's an important detail relating to trigonometric functions that stymies beginner programmers: trigonometric functions assume that their argument is in radians. For example, if you've a variable `theta` containing a value in degrees, that angle must be converted to radians (recall from trigonometry that $180^\circ = \pi$ radians). The following code fragment does the trick:

```
#define PI          (3.141593)
#define DEG_TO_RAD  (PI/180.0)
...
double theta_rad = theta * DEG_TO_RAD;
double x = sin(theta_rad);
// conversion can also be specified within function reference
double y = sin(theta * DEG_TO_RAD);
```

Distance between two points

Recall the problem statement:

$$\text{Compute straight line distance between two points in a plane.}$$

and the algorithm that was devised to solve this problem:

$$\begin{aligned}&\text{Input:} \quad P = (P.x, P.y) \text{ and } Q = (Q.x, Q.y)\\ &\text{Output: distance } d(P,Q) \text{ between points } P \text{ and } Q\end{aligned}$$

$$\begin{aligned}\textbf{Algorithm } d(P,Q)\quad &\textbf{1. } width = (Q.x - P.x) \quad [\text{Compute width of right triangle generated by points } P \text{ and } Q]\\ &\textbf{2. } height = (Q.y - P.y) \quad [\text{Compute height of right triangle generated by points } P \text{ and } Q]\\ &\textbf{3. } d(P,Q) = \sqrt{(width)^2 + (height)^2} \quad [\text{Compute hypotenuse of right triangle}]\end{aligned}$$

First, an `include` preprocessing directive for `math.h` and function prototype for function `distance` are provided in header file `distance.h`:

```
#include <math.h>

/*!
@author pghali
@brief  Computes the distance between two points.

This function takes coordinates of point (px, py) and
point (qx, qy) and returns the distance between P and Q.

@param px - double-precision floating-point value specifying px.
@param py - double-precision floating-point value specifying py.
@param qx - double-precision floating-point value specifying qx.
@param qy - double-precision floating-point value specifying qy.
@return   - a double-precision floating-point value measuring the
distance between P and Q.
```

```
*//*_____________________________________________________________*/
double distance(double px, double py, double qx, double qy);

```

The following code in source file `distance.c` defines function `distance`:

```
#include "distance.h"

double distance(double px, double py, double qx, double qy) {
  // compute sides of right triangle formed by two points
  double width  = qx - px;
  double height = qy - py;
  return sqrt(width*width + height*height);
}

```

Source file `distance.c` is [only] compiled in the usual manner:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c distance.c -o
distance.o
```

Function `distance` can be tested by calling it from function `main` which is implemented in file `test-dist.c`:

```
#include <stdio.h>
#include "distance.h"

int main(void) {
  double px = 0.0, py = 0.0, qx = 3.0, qy = 4.0; // input portion
  // compute distance from P(0, 0) to Q(3, 4)
  double dist = distance(px, py, qx, qy);
  printf("Distance is %f\n units", dist); // output portion
  return 0;
}

```

Source file `test-dist.c` is [only] compiled in the usual manner:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c test-dist.c -o
test-dist.o
```

However, the linker throws an error when an attempt is made to generate an executable:

```
$ gcc test-dist.o distance.o -o dist.out
usr/bin/ld: distance.o: in function `distance':
distance.c:(.text+0x59): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status

```

The problem is that function `sqrt` is not defined in the default C standard library `libc.a`. Instead, it is defined in external math library `libm.a` and because of historical reasons the compiler does not link to file `libm.a` unless it is explicitly selected. This is done using `-lm` option to the linker stage of `gcc`:

```
$ gcc test-dist.o distance.o -o test-dist.out -lm
```

Option `-lm` is a shorthand for *link object files with library file* `/usr/lib/x86_64-linux-gnu/libm.a`.

Run executable program `test-dist.out` to test function `distance`:

```
$ ./test-dist.out
Distance is 5.000000 units

```

The two source files `distance.c` and `test-dist.c` can be individually compiled and then linked together along with the C standard library with a single call to `gcc`:

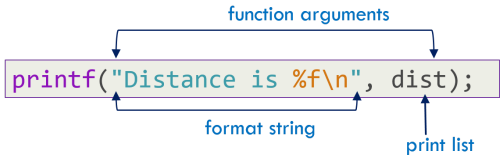
```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror distance.c test-dist.c -o test-dist.out
```

Printing output from program

The `printf` function was first introduced in the *Hello World!!!* program to print a sequence of characters comprising a greeting to `stdout` [the computer's screen by default]. In addition to printing explanatory text, the `printf` function was also used in the *distance* program to print values to `stdout`. Consider the following statement that prints the value of a `double` variable named `dist`:

```
printf("Distance is %f units\n", dist);
```

This `printf` statement contains two arguments: a *format string* and a *print list* containing an identifier to specify the value to be printed.



A format string is enclosed in a pair of double quotes `"`, and can contain text, format specifiers, or both. A *format specifier* begins with the character `%` and describes the format to use in printing the value of a variable. If the format string doesn't contain format specifiers, the characters in the format string will be printed to `stdout` as is. In this example, the format string specifies that the characters `Distance is` are to be printed. The next group of characters `%f` represents a format specifier that indicates that a floating-point value is to be printed next, which will then be followed by the characters `units`. The next combination of characters `\n` represents a newline indicator; it causes a skip to a newline on `stdout` after the information has been printed. The second argument in the `printf` statement is a variable `dist`; it is matched to the format specifier `%f` in the format string. Function `printf` is authored to a print a `float` or a `double` when the format specifier is `%f` [floating-point form]. Thus, in this example, since the value of `dist` is 5.0, this value will be printed to `stdout`:

```
Distance is 5.000000 units

```

Why does `printf` print `5.000000` and not `5.0`? This matter will be explained in the next section.

Printing floating-point values

Floating-point values are displayed in one of several formats as indicated in the following table. Please note that you don't need to memorize any of these details. They're only being provided here so that you know that there many ways in which floating-point values can be printed. And, when you need to print floating-point values a certain way, you should look up a reference page to know what your options are.

Floating-Point Format Specifiers for `printf`	
Conversion specifier	**Description**
`f` or `F`	Display floating-point value using *fixed-point notation*
`e` or `E`	Display floating-point value using *exponential notation*
`g` or `G`	Display floating-point value using either *fixed-point* or *exponential notation* depending on value's magnitude
`L`	Place before any floating-point format specifier to display `long double` value

- Format specifiers `f` and `F` always prints *at least* one digit to the left of the decimal point. Values displayed with format specifiers `f` and `F` print 6 digits of precision to the right of the decimal point by default. This is the reason why `printf` prints `5.000000` and not `5.0` in the previous example.
- Format specifiers `e` and `E` display floating-point values in exponential notation - the computer equivalent of scientific notation used in math. For example, the value 123.4567 is represented in scientific notation as 1.234567×10^2 and in exponential notation as $1.234567\text{e}{+}02$ by the computer. This notation indicates that 1.234567 is multiplied by 10 raised to the second power ($\text{e}{+}02$) where e stands for *exponent*. Format specifiers `e` and `E` print lowercase `e` and uppercase `E`, respectively, preceding the exponent, and print *exactly* one digit to the left of the decimal point. Values displayed with format specifiers `e`, and `E` print 6 digits of precision to the right of the decimal point by default.
- Format specifier `g` [or `G`] prints in either `e` (or `E`) or `f` format with no trailing zeros. For example, 1.234000 is printed as `1.234`. The details of which format is picked are too detailed for beginners and the interested reader can look up the details here.

The following code illustrates the various floating-point format specifiers:

```c
#include <stdio.h> // contains function prototype of printf

int main(void) {
  double d  = 1234567.89;
  float f  = d;

  // format specifier f to print float value
  printf("Print float  using format specifier %%f: %f\n", f);
  // format specifier f to print double value
  printf("Print double using format specifier %%f: %f\n", d);
  // format specifier F to print float value
  printf("Print float  using format specifier %%F: %F\n", f);
  // format specifier F to print double value
  printf("Print double using format specifier %%F: %F\n", d);
  // format specifier e to print float value
  printf("Print float  using format specifier %%e: %e\n", f);
  // format specifier e to print double value
  printf("Print double using format specifier %%e: %e\n", d);
  // format specifier E to print float value
  printf("Print float  using format specifier %%E: %E\n", f);
  // format specifier E to print double value
  printf("Print double using format specifier %%E: %E\n", d);
  // format specifier g to print float value
  printf("Print float  using format specifier %%g: %g\n", f);
  // format specifier g to print double value
  printf("Print double using format specifier %%g: %g\n", d);
  // format specifier G to print float value
  printf("Print float  using format specifier %%G: %G\n", f);
  // format specifier G to print double value
  printf("Print double using format specifier %%G: %G\n", d);

  return 0;
}

```

The `printf` statements print the following text to `stdout`:

```
Print float  using format specifier %f: 1234567.875000
Print double using format specifier %f: 1234567.890000
Print float  using format specifier %F: 1234567.875000
Print double using format specifier %F: 1234567.890000
Print float  using format specifier %e: 1.234568e+06
Print double using format specifier %e: 1.234568e+06
Print float  using format specifier %E: 1.234568E+06
Print double using format specifier %E: 1.234568E+06
Print float  using format specifier %g: 1.23457e+06
Print double using format specifier %g: 1.23457e+06
Print float  using format specifier %G: 1.23457E+06
Print double using format specifier %G: 1.23457E+06

```

Notice that the `E`, `e`, `g`, and `G` format specifiers cause the value to be rounded when printed to `stdout` while format specifier `f` does not.

Printing integer values

Integer values are displayed in one of the several format specifiers shown in the following table:

Integer Format Specifiers for `printf`	
Conversion specifier	**Description**
`d`	Display as *signed integer*
`i`	Display as *signed integer*
`u`	Display as *unsigned integer*
`o`	Display as *unsigned octal integer*
`x` or `X`	Display as *unsigned hexadecimal integer* with hexadecimal digits printed as `a-f` or printed as `A-F`
`h` or `l` or `ll`	**Length modifiers** – place ***before*** any integer format specifier to display `short int` or `long int` or `long long int` values

The following code illustrates the various integer format specifiers:

```c
#include <stdio.h> // contains function prototype of printf

int main(void) {
  int val = 345;

  // format specifier d to print signed integer value
  printf("Format specifier %%d:   %d\n", +val);
  // format specifier d to print signed integer value
  printf("Format specifier %%d:   %d\n", -val);
  // format specifier i to print signed integer value
  printf("Format specifier %%i:   %i\n", -val);
  // format specifier u to print unsigned integer value
  printf("Format specifier %%u:   %u\n", val);
  // format specifier u to print unsigned integer value
  printf("Format specifier %%u:   %u\n", -val);
  // format specifier o to print unsigned integer value in octal
  printf("Format specifier %%o:   %o\n", val);
  // format specifier o to print unsigned integer value in octal
  printf("Format specifier %%o:   %o\n", -val);
  // format specifier x to print unsigned integer value in hexadecimal
  printf("Format specifier %%x:   %x\n", val);
  // format specifier x to print unsigned integer value in hexadecimal
  printf("Format specifier %%x:   %x\n", val);
  // format specifier x to print unsigned integer value in hexadecimal
  printf("Format specifier %%x:   %x\n", -val);
```

```
  // length modifier h used with format specifier d to
  // print unsigned integer value as short int
  printf("Format specifier %%hd:  %hd\n", val);
  // length modifier h used with format specifier d to print signed int
  // value 65535 which is -1 as signed short int
  printf("Format specifier %%hd:  %hd\n", 65535);
  // length modifier l used with format specifier d to print
  // signed integer value as long int
  printf("Format specifier %%ld:  %ld\n", -20000000L);
  // length modifier l used with format specifier u to print
  // unsigned integer value as unsigned long int
  // UL suffix means constant is of type unsigned long int
  printf("Format specifier %%lu:  %lu\n", 20000000UL);
  // length modifier L used with format specifier u to print
  // signed long int value as unsigned long int
  // L suffix means constant is of type signed long int
  printf("Format specifier %%lu:  %lu\n", -20000000L);
  // length modifier LL used with format specifier d to print
  // signed long long int value as signed long long int value
  // LL suffix means constant is of type signed long long int
  printf("Format specifier %%lld: %lld\n", -20000000LL);
  // length modifier ll used with format specifier d to print
  // unsigned long long int value as unsigned long long int value
  // LLU suffix means constant is of type unsigned long long int
  printf("Format specifier %%llu: %llu\n", 20000000LLU);

  return 0;
}

```

Notice that the minus sign prints on line 9, while the plus sign is suppressed on line 7. Also, format specifier `i` on line 11 behaves the same as format specifier `d` on line 9. Also, on line 15, format specifier `u` interprets value -345 as unsigned value 4294966951. The `printf` statements print the following text to `stdout`:

```
Format specifier %d:   345
Format specifier %d:   -345
Format specifier %i:   -345
Format specifier %u:   345
Format specifier %u:   4294966951
Format specifier %o:   531
Format specifier %o:   37777777247
Format specifier %x:   159
Format specifier %x:   159
Format specifier %x:   fffffea7
Format specifier %hd:  345
Format specifier %hd:  -1
Format specifier %ld:  -20000000
Format specifier %lu:  20000000
Format specifier %lu:  18446744073689551616
Format specifier %lld: -20000000
Format specifier %llu: 20000000

```

Printing multiple values

It is possible to print multiple values to `stdout` using a single `printf` statement with a format string containing multiple format specifiers. The *distance* program is rewritten to print the coordinates of the source and destination points in addition to the computed distance:

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

int main(void) {
  double px = 0.0, py = 0.0, qx = 3.0, qy = 4.0; // input portion
  double dist = ; // compute distance
  printf("Distance from (%f, %f) to (%f, %f) is %f\n units",
                  px, py, qx, qy, distance(px, py, qx, qy));
  return 0;
}

```

Since there are five format specifiers in the format string, five corresponding variables or expressions must follow in the print list. The first format specifier corresponds to variable `px`, the second specifier corresponds to variable `py`, and so on. Because values displayed with format specifier `f` print 6 digits of precision to the right of the decimal point by default, the text printed to `stdout` will be:

```
Distance from (0.000000, 0.000000) to (3.000000, 4.000000) is 5.000000 units

```

Reading input from user

Suppose the programmer wishes to test the *distance* program with various position coordinates other than $P = (0, 0)$ and $Q = (3, 4)$. The programmer would have to change values of variables `px`, `py`, `qx`, and `qy`, and then recompile `test-dist.c`, relink, and reexecute the program to obtain the distance for a different set of points. Alternatively, if `scanf` function from C standard library is used to read position coordinates, there is no need to recompile and relink the program; the program only needs to be reexecuted.

The new version of function `main` in source file `test-dist-new.c` looks like this:

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

int main(void) {
  // input portion
  double px, py, qx, qy;
  printf("Enter point P: ");
  scanf("%lf %lf", &px, &py);
  printf("Now, enter point Q: ");
  scanf("%lf %lf", &qx, &qy);

  // call to function distance
  double dist_pq = distance(px, py, qx, qy);

  // output portion
  printf("Distance from P(%.3f, %.3f) to Q(%.3f, %.3f) is %.3f\n",
         px, py, qx, qy, dist_pq);
  return 0;
}

```

Consider the call to `scanf` function on line 8: `scanf("%lf %lf", &px, &py)`. The first argument of the `scanf` function is a *format string* `"%lf %lf"` that specifies the types of variables whose values are to be entered from `stdin` [keyboard]. The entire (and complex) list of type specifiers is provided here. A simpler list of type specifiers is shown in the following table:

Variable Type	Specifier
Integer Values	
int	%i, %d
short	%hi, %hd
long int	%li, %ld
unsigned int	%u
unsigned short	%hu
unsigned long	%lu
Floating-Point Values	
float	%f, %e, %E, %g, %G
double	%lf, %le, %lE, %lg, %lG
long double	%Lf, %Le, %LE, %Lg, %LG

From the table, the specifiers for an `int` variable are `%i` or `%d`; the specifiers for a `float` variable are `%f` , `%e`, and `%g`; and the specifiers for a `double` variable are `%lf`, `%le`, and `%lg`. It is critical to use the correct specifier - don't expect help from the compiler if you use `%f` specifier to read the value for a `double` variable - your program will fail miserably!!!

The remaining two arguments in `scanf` function are memory locations that correspond to the specifiers in the control string. These memory locations are indicated with *address operator* `&`. This operator is a *unary* operator [meaning it requires a single operand] that determines the memory address of the operand with which it is associated. A common error is to omit the address operator for identifiers.

Since the values to be entered through `stdin` (the keyboard) are double-precision floating-points to be stored in variables `px` and `py`, the two arguments are `&px` and `&py`. Since the program must read two values, the numbers must be separated by at least one whitespace character which could be a space character or a tab or a newline. Note that the number may contain a decimal point, but doesn't have to.

In order to prompt the user to enter the values, a `scanf` statement is preceded by a `printf` statement that describes the information that the user should enter from the keyboard:

```
printf("Enter point P: ");
```

Compile this source file, link with `distance.o` and the math library to create an executable:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c test-dist.c -o
test-dist.o
$ gcc test-dist-new.o distance.o -o test-dist-new.out -lm
```

Run the executable program and the interaction with the user looks like this:

```
$ ./test-dist-new.out
Enter point P: 10.12 12.10
Now, enter point Q: 13.45 45.13
Distance from P(10.120, 12.100) to Q(13.450, 45.130) is 33.197
$
```

Things to review

1. What is an *identifier*? What is the legal way to write an identifier?
2. What is a *keyword*? List the keywords you've come across in this document.
3. What is a *function*?
4. What is the difference between a *function declaration* [or *function prototype*] and a *function definition*?
5. What are *function parameters*?
6. How do functions specify that they will not take any parameters? How do functions specify that they're not returning values?
7. What is the purpose of function `main` in a C program?
8. What is a C *comment*? How many different ways can you comment a C source file?
9. What is the *preprocessor*?
10. What are *whitespace characters*? What purpose do whitespace characters serve?
11. What is a *data type* in the context of a programming language?
12. What is a *variable*?

13. What is a *literal value*? How does it differ from a variable? What is a *string literal*? What are the other literals in this tutorial?
14. Using function `printf` how will you print values of type `int` and values of type `double`?
15. Using function `scanf` how will you print values of type `int` and values of type `double`?
16. What does a *compiler* do? What is the input to a compiler and what is its output?
17. What is the purpose of a *linker*? What are its inputs and what does it generate as output?