**Space Age**

In Practice Mode

Introduction

Given an age in seconds, calculate how old someone would be on:

* Earth: orbital period 365.25 Earth days, or 31557600 seconds
* Mercury: orbital period 0.2408467 Earth years
* Venus: orbital period 0.61519726 Earth years
* Mars: orbital period 1.8808158 Earth years
* Jupiter: orbital period 11.862615 Earth years
* Saturn: orbital period 29.447498 Earth years
* Uranus: orbital period 84.016846 Earth years
* Neptune: orbital period 164.79132 Earth years

So if you were told someone were 1,000,000,000 seconds old, you should be able to say that they're 31.69 Earth-years old.

If you're wondering why Pluto didn't make the cut, go watch [this youtube video](http://www.youtube.com/watch?v=Z_2gbGXzFbs).

Exercises without stub implementations

Like the majority of C programs you will write, this exercise comes without any header file (\*.h) or source file (\*.c).

Add the required files

You will need to create these yourself as part of the exercise. This is so that you get to practice deciding on and creating the program's interface yourself, which is an important part of programming in C. It does mean that when you first try to run the tests, they won't compile. They will give you an error similar to:

make: \*\*\* No rule to make target 'src/exercise\_name.c', needed by 'tests.out'. Stop.

This error occurs because the makefile attempts to use a file that hasn't been created yet (exercise\_name.c). To resolve this error you will need to add a matching file to the src directory. For example, for the error above you would add a file called exercise\_name.c.

When you try to run the tests again you will get a slightly different error.

make: \*\*\* No rule to make target 'src/exercise\_name.h', needed by 'tests.out'. Stop.

Again the makefile still needs another file that has yet to be created (exercise\_name.h). The solution to this error is similar to the last, add the matching file to the src directory. For example, for the above error you would add a file called exercise\_name.h

Add the required declarations

Running the tests a third time you see you will get new errors, similar to:

Compiling tests.out

src/exercise\_name.c:1:0: error: ISO C forbids an empty translation unit [-Werror=pedantic]

cc1: all warnings being treated as errors

test/test\_exercise\_name.c:13:1: error: unknown type name ‘bar\_t’; did you mean ‘\_\_bar\_t’?

bar\_t b = foo(a, 2.5);

^~~~~~

\_\_bar\_t

test/test\_exercise\_name.c:15:14: error: storage size of fizz isn’t known

struct baz fizz;

^~~~

test/test\_exercise\_name.c:29:20: error: implicit declaration of function ‘foo’ [-Werror=implicit-function-declaration]

bar\_t b = foo\_function(fizz, 2.5);

This error means that you need to add function, type and variable decarations to the header file to match those the test file is attempting to use. In doing this you will need to look at the build errors and the test code to determine what needs to be declared in the header file.

For example, for the error above you should declare a type named bar\_t, a struct called baz and a function named foo(). Additionally we can tell from looking at the above errors that function foo() has two parameters.

The first parameter is passed an argument of fizz and thus should have a type of struct baz. The second parameter is passed 2.5 and so could have one of two types, either float or double. You will need to look at the test code to determine which.

The *names* of the parameters are not determined by the test code and so are left up to you to decide, though in keeping with C programming practices they should be descriptive but not overly long.

Further, we can see that the the return type expected by the test code is of the type bar\_t.

Putting this all together we end up with a function declaration that looks like the following:

// The parameter names are not very good here they should be more descriptive in a real exercise.

// We have decide on a double for the second parameter in this hypothetical example.

bar\_t foo(struct baz b, double d);

You should continue to do this for any further similar errors. To check that you have correctly determined the required declaration, just run the tests again and analyse any errors similarly.

Additionally, remember to add any includes to the header file any headers that it itself requires.

Include guards

Before you are finished with the header file, you should add include guards. Include guards are used to help prevent the file from being included multiple times by accident. If the file *was* included multiple times then the functions and other items would be declared twice; In C this is an error because the compiler thinks you are trying to declare another *different* function that has the same name.

To add include guards, add something similar to the following as the first two lines of the header:

#ifndef EXERCISE\_NAME\_H

#define EXERCISE\_NAME\_H

And on the very last line:

#endif

Add the required definitions

Once the header file is complete you may still have build errors similar to the following:

Compiling tests.out

src/exercise\_name.c:1:0: error: ISO C forbids an empty translation unit [-Werror=pedantic]

cc1: all warnings being treated as errors

makefile:24: recipe for target 'tests.out' failed

make: \*\*\* [tests.out] Error 1

This is because although you have *declared* all the items you need to, they have not yet been *defined*. To define the needed items you need to add their implementation to the exercise\_name.c file. For the function foo() from the previous example, this would look similar to:

#include "exercise\_name.h"

bar\_t foo(struct baz b, double d)

{

// Your exercise code here

}

After having resolved these errors you should be ready to start making the tests pass!

Getting Started

Make sure you have read the "Guides" section of the [C track](https://exercism.io/my/tracks/c) on the Exercism site. This covers the basic information on setting up the development environment expected by the exercises.

Passing the Tests

Get the first test compiling, linking and passing by following the [three rules of test-driven development](http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd).

The included makefile can be used to create and run the tests using the test task.

make test

Create just the functions you need to satisfy any compiler errors and get the test to fail. Then write just enough code to get the test to pass. Once you've done that, move onto the next test.

As you progress through the tests, take the time to refactor your implementation for readability and expressiveness and then go on to the next test.

Try to use standard C99 facilities in preference to writing your own low-level algorithms or facilities by hand.