TerribleFall: test-driven development of a Physics engine in C++

In this worksheet, we are going to learn about test-driven development in C++.

Set up your environment

We have set up a lab environment for you to work in on the Coursera platform. It has all the tools you need to develop basic C++ code, namely a text editor, a compiler, a linker and a terminal in which to run your program in.

For this lab, we do not support development on your local machine as setting up a development environment is beyond the scope of this lab. We advise you to use the coursera labs environment.

Working in the Coursera lab environment

The Coursera lab environment has already been configured for you with all the essentials to complete this activity. The lab is integrated with Visual Studio Code, an extremely popular code editor optimised for building and debugging modern web and cloud applications.

Start the lab environment application

It is simple to launch a lab exercise. You only need to click on the button "Start" below the activity title to enter a lab environment. Let's explore this lab activity. Go ahead and click on the "Start" button!

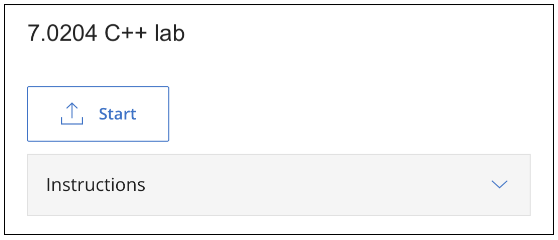


Figure 1: Coursera "Start" button

The Lab environment application

The lab environment application may take some time to load and you will be prompted with the following animation:

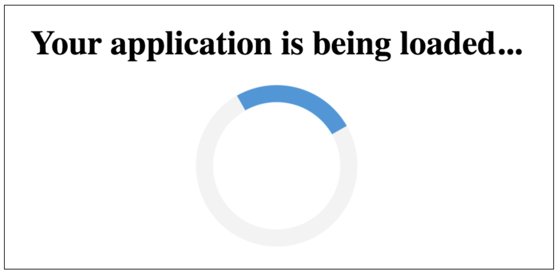
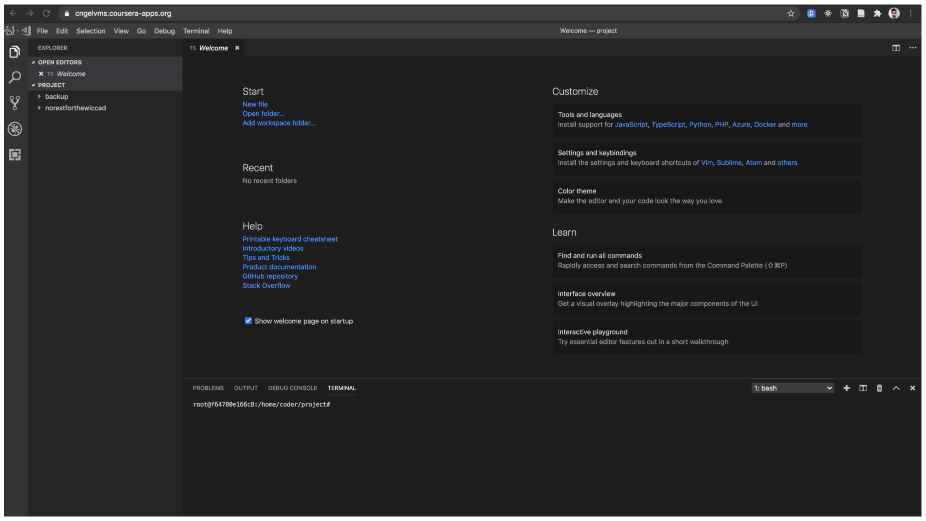

Your application is being loaded...

Figure 2: Coursera loading screen

Just wait a few seconds for your lab activity to start. It should not take longer than thirty seconds. If you experience any issues please load the activity again.

Once the application loads, you will see an instance of Visual Studio Code IDE as shown below:


cngelvms.coursera-apps.org
File
Edit
Selection
View
Go
Debug
Terminal
Help
Welcome — project
EXPLORER
OPEN EDITORS
TS Welcome
PROJECT
backup
norestforthewiccad
TS Welcome
Start
New file
Open folder...
Add workspace folder...
Recent
No recent folders
Help
Printable keyboard cheatsheet
Introductory videos
Tips and Tricks
Product documentation
GitHub repository
Stack Overflow
Show welcome page on startup
Customize
Tools and languages
Install support for JavaScript, TypeScript, Python, PHP, Azure, Docker and more
Settings and keybindings
Install the settings and keyboard shortcuts of Vim, Sublime, Atom and others
Color theme
Make the editor and your code look the way you love
Learn
Find and run all commands
Rapidly access and search commands from the Command Palette (⇧⌘P)
Interface overview
Get a visual overlay highlighting the major components of the UI
Interactive playground
Try essential editor features out in a short walkthrough
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
1: bash

```
root@f64780e166c8:/home/coder/project#
```

Figure 3: Visual Studio Code editor

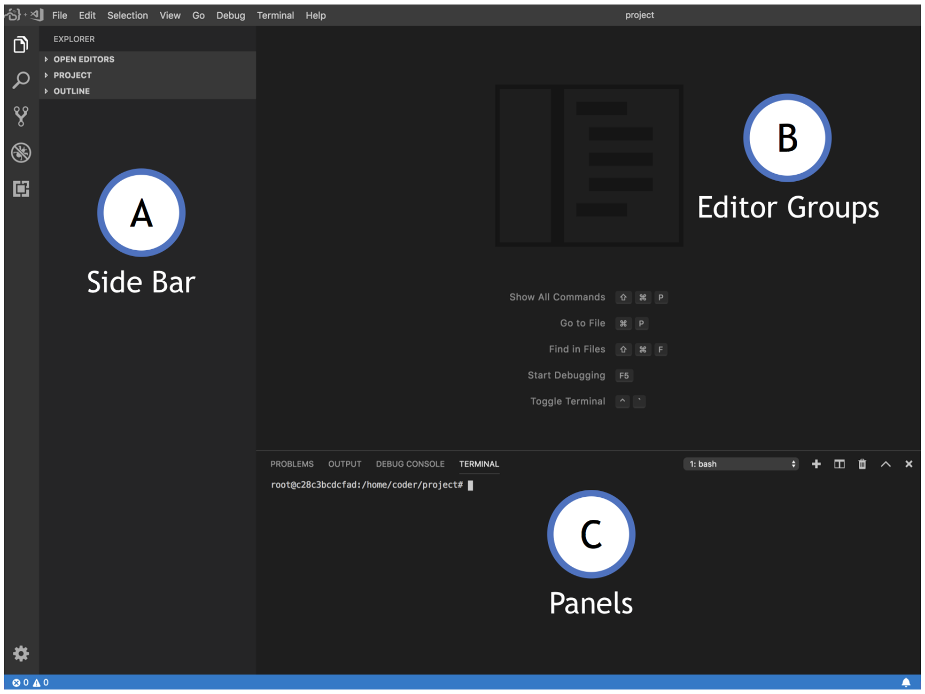
The “Welcome” page illustrates most of the features to get you started with the environment. For now you can close the “Welcome” tab by clicking on the X icon next to the tab name as we will explore the main sections together. The “Welcome” page does not always appear so do not worry as it will not make any difference in the way the IDE works.

The Visual Studio User Interface

The Visual Studio User Interface is very easy to learn and it comes with a great selection of features. We will not explore in details all of the functionalities as it is not in the purpose of this course. We will instead look at the main sections required for you to get started with the labs.

VS Code comes with a simple and intuitive layout that maximises the space provided for the editor while leaving ample room to browse and access the full context of your folder or project.

The VS Code layout sections that you will require for this exercises are the Side Bar, the Editor Groups and the Panels.



```
root@c28c3bcdcfad:/home/coder/project#
```

Figure 4: Visual Studio Code layout sections

- **Side Bar (A)** - Contains different views like the Explorer to assist you while working on your project. Mostly used to navigate your folders.
- **Editor Groups (B)** - The main area to edit your files. You can open as many editors as you like side by side vertically and horizontally.
- **Panels (C)** - You can display different panels below the editor region for output or debug information, errors and warnings, or an integrated terminal.

Do not worry if your IDE configuration looks slightly different from the above picture. You can always drag the tabs around to adjust your layout configuration.

Note about the integrated terminal

The integrated terminal panel is normally hidden by default in Visual Studio Code and you need to manually enable it. Click the "View" tab on the top navigation bar and then click the "Terminal" tab to enable the panel window like shown below:

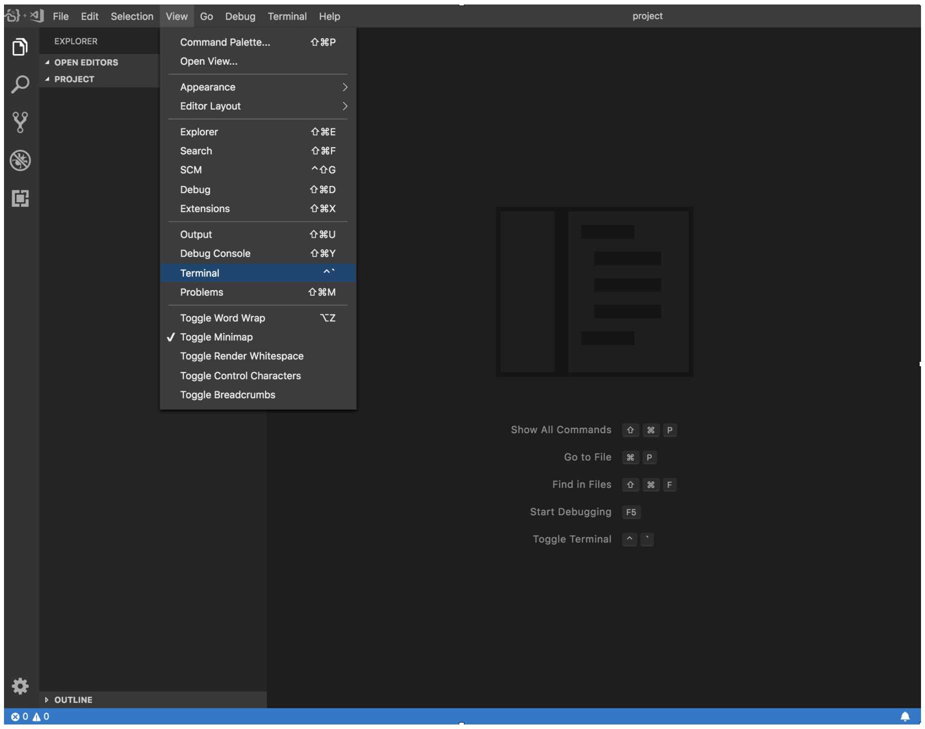


Figure 5: Visual Studio Code Terminal

Final touches

You are now set to work in the Coursera lab environment. Just few other things before you begin this exercise:

- There is a folder called "backup" inside the environment, ignore it for now.
- You can create, edit, and delete files and folders in the Side Bar section.
- The exercise path is /home/coder/project/, open the Terminal and type **cd /home/coder/project/** if you get lost.

Write and run a test program

We are now going to create our first test program. Working in the coursera labs environment, open up a terminal and check you have the cppunit library installed by running this command:

```
apt install libcppunit-dev
```

That installs the cppunit library on the container you are working in. It should already be there.

Write the code

Now we are going to create a new C++ file and put a test into it. Create a file called test.cpp and put the following code in it:

```
#include <cppunit/TestCase.h>
#include <cppunit/TestCaller.h>
#include <cppunit/ui/text/TestRunner.h>

class FixtureTests : public CppUnit::TestFixture
{
  public:

    void setUp() override
    {
    }
    void tearDown() override
    {
    }
    void testAddition()
    {
      CPPUNIT_ASSERT(2+3 == 4);
      CPPUNIT_ASSERT(2+3 == 6);
    }
};

int main()
{
  CppUnit::TextUi::TestRunner runner{};
  runner.addTest(new CppUnit::TestCaller<FixtureTests>{
      "test the addition operator",
      &FixtureTests::testAddition
  });
  runner.run();
  return 0;
}
```

Notes:

- The main function is the entry point for the program. It will run that first.
- main creates a TestRunner, adds a test to it wrapped in a TestCaller, then runs the test.

Compile and run the code

Compile and link this program by running the following command in the terminal:

```
g++ test.cpp -lcppunit -o testfall
```

Notes:

- g++ is the GNU C++ compiler and linker tool
- test.cpp is the name of the file you just created
- -lcppunit means link to the cppunit library
- -o testfall means once you have built the executable program, call it testfall

Run the program by running this command in the terminal:

```
./testfall
```

You should see some output like this:

```
.F


!!!FAILURES!!!
Test Results:
Run:  1   Failures: 1   Errors: 0


1) test: test the addition operator (F) line: 18 test.cpp
assertion failed
- Expression: 2+3 == 4
```

Pass the test

Can you figure out why the test failed?

Change the code for the test so that it passes. Clue - look for the line that comapres 2+3 to 4. Write the least code possible to pass the test.

Backup your work in the Coursera development environment

It is always a good idea to backup your work and indeed a best practice in the cycles of the test-driven development workflow. In order to backup your work inside the Coursera development environment do the following:

- Run this command inside the terminal: /home/coder/project/backup/script/backup.sh
- Visit **https://your-lab-identifier**/download/ to download your backup.
- **https://your-lab-identifier** is your lab URL, simply add /download/ at the end of it and visit the combined URL in a new browser tab.

Start testing the physics engine

Now we are ready to do the real work which is to test the physics engine.

Create a folder for the physics engine code

Create a folder called src in your project - use the VSCode add folder function by right clicking on the file browser area of the interface.

Create the files for the physics engine

Create two files in the src folder called physics.h and physics.cpp. Leave the files empty for now. Add this line to the top of test.cpp to include the code in physics.h:

```
#include "src/physics.h"
```

- Add a new test to test.cpp, below testAddition:

  ```
  void testThingPosition()
  {
      Thing thing{5.0f, 10.0f, 1.0f};
      CPPUNIT_ASSERT(thing.getX() == 5.0f);
  }
  ```

- Add the test to the testrunner by adding this code to the main function:

  ```
  runner.addTest(new CppUnit::TestCaller<FixtureTests>
  {
      "test the thing position is correct",
      &FixtureTests::testThingPosition
  });
  ```

- For reference, your complete test.cpp file should now contain this:

```
#include <cppunit/TestCase.h>
#include <cppunit/TestCaller.h>
#include <cppunit/ui/text/TestRunner.h>
#include "src/physics.h"

class FixtureTests : public CppUnit::TestFixture
{
    public:

        void setUp() override
        {
        }
        void tearDown() override
        {
        }
        void testAddition()
        {
            CPPUNIT_ASSERT(2+2 == 4);
        }


        void testThingPosition()
        {
            Thing thing{5.0f, 10.0f, 1.0f};
            CPPUNIT_ASSERT(thing.getX() == 5.0f);
        }
};

int main()
{
    CppUnit::TextUi::TestRunner runner{};
    runner.addTest(new CppUnit::TestCaller<FixtureTests>{
        "test the addition operator",
        &FixtureTests::testAddition
    });

    runner.addTest(new CppUnit::TestCaller<FixtureTests>{
        "test position",
        &FixtureTests::testThingPosition
    });

    runner.run();
    return 0;
}
```

- Compile and run:

  g++ test.cpp -lcppunit -o testfall then ./testfall

You should see some output like this:

```
test.cpp: In member function ‘void FixtureTests::testThingPosition()’:
test.cpp:24:7: error: ‘Thing’ was not declared in this scope
       Thing thing{5.0f, 10.0f, 1.0f};
       ^~~~~
In file included from /usr/include/cppunit/TestCase.h:6:0,
                 from test.cpp:1:
test.cpp:25:22: error: ‘thing’ was not declared in this scope
       CPPUNIT_ASSERT(thing.getX() == 5.0f);
                      ^
test.cpp:25:22: note: suggested alternative: ‘this’
```

That is because you have not declared the Thing class in physics.h. We'll do that next.

Add the basic Thing class definition

Add this code to physics.h:

```
#pragma once

class Thing
{
    public:
        /** create a thing with the sent properties
         * @param x - the horizontal position
         * @param y - the vertical position
         * @param radius - the radius (we assume it is circular)
        */
        Thing(float x, float y, float radius);
        /**
         * set the position of the thing
         * @param x - the horizontal position
         * @param y - the vertical position
         */
        void setPosition(float x, float y);
        /**
         * Get the horizontal position
         * @return float
         */
        float getX();
        /**
         * Get the vertical position
```

```
         * @return float
         */
        float getY();
        /**
         * Apply a force of the sent magnitude to the
         * thing
         * @param xForce - horizontal force
         * @param yForce - vertical force
         */
        void applyForce(float xForce, float yForce);
        /** update the position of the thing
         * New position  = x + dx, y + dy
         */
        void update();
    private:
        float x;
        float y;
        float dX;
        float dY;
        float radius;
};
```

Now run the command again:

```
g++ test.cpp -lcppunit -o testfall
```

More errors:

```
/tmp/ccYhkL5e.o: In function `FixtureTests::testThingPosition()':
test.cpp:(.text._ZN12FixtureTests17testThingPositionEv
[_ZN12FixtureTests17testThingPositionEv]+0x45): undefined reference to
`Thing::Thing(float, float, float)'
test.cpp:(.text._ZN12FixtureTests17testThingPositionEv
[_ZN12FixtureTests17testThingPositionEv]+0x116): undefined reference to
`Thing::getX()'
collect2: error: ld returned 1 exit status
```

That is because we have promised to implement all those things in physics.h but we've not actually done it!

Add the implementation of the Thing class

Let's provide an implementation by adding this code to physics.cpp (note it is physics.cpp not physics.h):

```
#include "physics.h"
// Thing constructor
Thing::Thing(float x, float y, float radius)
{
}
// end of Thing constructor

void Thing::setPosition(float x, float y)
{
}
float Thing::getX()
{
    return this->x;
}
float Thing::getY()
{
    return this->y;
}
void Thing::applyForce(float xForce, float yForce)
{
}
void Thing::update()
{
}
```

Now run the compile and link command again:

```
g++ test.cpp src/physics.cpp -lcppunit -o testfall
```

Note that we have added src/physics.cpp to the list of cpp files. Now run the binary:

```
./testfall
```

You should see this:

```
..F


!!!FAILURES!!!
Test Results:
Run:  2   Failures: 1   Errors: 0


1) test: test position (F) line: 25 test.cpp
assertion failed
- Expression: thing.getX() == 5.0f
```

Now we are ready to start working on this code to pass the test.

Write the code to pass the first test

Add this code into src/physics.cpp, in place of the part labelled as Thing constructor:

```
// Thing constructor
Thing::Thing(float x, float y, float radius)
    : x{x}, y{y}
{
}
// end of Thing constructor
```

Re-run the command:

```
g++ test.cpp src/physics.cpp -lcppunit -o testfall
```

and

```
./testfall
```

Now you should be passing the tests:

```
..


OK (2 tests)
```

More features!

Next we are going to work on some more features - let's look at movement. The movement of thing should be controlled by the Thing.update function and its dX and dY variables. Each time we call update on a thing, it should change its x and y position by dX and dY.

Verify that the Thing starts off still

A thing should start off being still. Let's create a test which creates a thing, calls update, then checks it has not moved. In test.cpp:

```
void testThingStartsStill()
{
  Thing thing{5.0f, 10.0f, 1.0f};
  thing.update();
  CPPUNIT_ASSERT(thing.getX() == 5.0f);
}
```

In test.cpp::main:

```
runner.addTest(new CppUnit::TestCaller<FixtureTests>{
  "test starts still",
  &FixtureTests::testThingStartsStill
});
```

Re-run the command:

```
g++ test.cpp src/physics.cpp -lcppunit -o testfall
```

and

```
./testfall
```

That looks ok… but I'm not convinced update actually did anything. Put this code in Thing.update, in physics.cpp:

```
void Thing::update()
{
    x += dX;
    y += dY;
}
```

Now we are failing:

```
...F


!!!FAILURES!!!
Test Results:
Run:  3   Failures: 1   Errors: 0


1) test: test starts still (F) line: 31 test.cpp
assertion failed
- Expression: thing.getX() == 5.0f
```

To pass this one, we need to initialise dX and dY as in C++, your variables might contain garbage if you do not initialise them. Update the thing constructor in physics.cpp:

```
Thing::Thing(float x, float y, float radius)
: x{x}, y{y}, dX{0}, dY{0}
{
}
```

Now you should pass. This is a really common error in C++ - not initialising class data members (the private fields). Especially if you come from a language like Java or Javascript.

A failing test for applyForce

Now let's try applying forces to a Thing. Applying a force to a still object should cause it to move. Here is a test for that. In test.cpp:

```
void testThingMoves()
{
  Thing thing{5.0f, 10.0f, 1.0f};
  thing.applyForce(1, 1);
  thing.update();
  // we'll assume a simple force model
  // where applying a force means it
  // adds that dX each time we update
  CPPUNIT_ASSERT(thing.getX() == 6.0f);
}
```

and in main in test.cpp:

```
runner.addTest(new CppUnit::TestCaller<FixtureTests>{
      "test force",
      &FixtureTests::testThingMoves
  });
```

Run the commands:

g++ test.cpp src/physics.cpp -lcppunit -o testfall

and

```
./testfall
```

Does this test fail? Can you write the code to make it pass by fixing the Thing::applyForce function?

Further tests

Now we have run a few cycles of the test-driven workflow in C++. If you want to continue working on the physics engine, here are some ideas:

didCollide function

Add a didCollide function to Thing in physics.h. It should have the following signature:

bool didCollide(Thing& otherThing);

If the two things are close enough to eachother, it should return true, otherwise, false.

World class

Can you work on a World class that has the following basic design? Put this in physics.h, then work on the implementation in physics.cpp:

```
class World
{
    public:
        /** Create a world with the sent properties
         * @param width
         * @param height
         * @param gravity
        */
        World(float width, float height, float gravity);
        /**
         * Add a thing to the world.
         * @param thing - a pointer to a thing (pass this in with &aThing)
         */
        void addThing(Thing* thing);
        /**
         * Checks if the two things collided,
         * true if distance < radius1 + radius2
         * @param thing1: a pointer to a thing (&thing)
         * @param thing2: a pointer to another thing (&thing)
         * @return bool
         */
        bool didThingsCollide(Thing* thing1, Thing* thing2);
        /**
         * Returns the number of things in the world
         * @return int
         */
        int countThings();
        /**
         * Update the things in the world, including
         * applying gravitational forces
         * computing collisions
         * and applying edge of world bounces
         */
        void update();
    private:
        float width;
        float height;
        float gravity;
};
```

Here are some things you might test:

- Can you add things to World and does it correctly store them?
- Does World apply gravitational forces to things correctly?
- Does World calculate if things have collided and apply basic repulsion forces?