G52CPP, 2017/18, Coursework Part 4, working full program

Overview

Coursework part 4 builds on part 3 to produce a working program which actually does something useful, interesting or fun. Please see the Hall of Fame link on the moodle page for examples of previous courseworks, but note that you only need to meet the requirements, no necessarily make something as good as these.

Look at the requirements/marking criteria. Meet the basic requirements. ***Choose the advanced criteria that you think you can do relatively quickly.*** Do not do too much. The framework should make many of these easier to implement than you expect if you understand correctly how to use it.

VERY IMPORTANT: limit the time that you spend on this part of the coursework to a reasonable time for a 20% coursework (e.g. around 20 hours of time). The problem with courseworks is that too many people try to get 100% - this means that ***some people who find it harder spend far too long on it***. Often marking criteria are designed to assess how well you did something and give you a proportion of the marks related to how well you did (e.g. you got 3 out of 5 marks for this criterion). I want to be more transparent in the marking and ensure fairness across markers, so this year I have made the criteria slightly different for this – every criterion is binary (you did it successfully or you didn't). Because you know the criteria and each is fixed it is easy to tell yourself that everyone can get 100%, but in general it is not worth it for everybody. As a marker I find that in exams people who struggle more get lower marks, whereas in courseworks people who struggle just spend too long, get frustrated, use time that they should be spending on other modules, and complain that the coursework took too long. You are strongly encouraged not to do this – it will not be worth it in the long run. You are strongly advised to remember that 70% is a first class and good mark and to not spend too long on the coursework. I have tried to set coursework so that you get the mark you deserve in a reasonable time – which should be possible if you are sensible on how much time you spend on it. In other words the most capable C++ programmer should be able to get full marks in a reasonable time but others would take far too long to do so (some requirements are actually really quick to do, even including debugging time). Note: I could discourage spending too long on it by setting a tight completion deadline, but I dislike doing so as it limits your flexibility to decide what to work on when and to apportion your time between different modules, so please be sensible about the amount of time you spend on this!

Getting started:

1. Complete coursework part 3 first. See the requirements for part 3. Part 3 is due before Easter, part 4 is due afterwards.
2. Experiment with the framework. Try changing which object type is created in the mainfunction.cpp file (change which line is commented out) and look at the BouncingBall example and the Demos 1 to 4. Each should teach you new things that you can do and give examples of how to do them.
3. Read all of the requirements first – both the mandatory and optional ones – since some are a lot easier than others. Pick the optional ones that you want to do (if any) carefully, to be coherent and make the program that you desire. If an optional requirement looks too hard or doesn't really fit your program then do another one instead. This sounds obvious but you will be surprised how many people just do the first few options when given a choice.

4. Decide on the program that you want to do – which will meet enough optional requirements to give you the mark you aim for. It may be a good idea to plan along the lines of "I'll definitely do these ones, and also try these ones if I have time" rather than fixing in your mind at the beginning what you will do, so think about a program idea which allows you to sensibly do that.
5. Start on the requirements below, using what you learned from exercises A and B and coursework part 3.
6. Design your code before you write it. It's a lot easier to get the code right initially than to try to fix it later – even though the design stage can seem tedious and like you are not making progress. It is REALLY EASY to introduce bugs in C++. It is REALLY HARD to debug some errors in C++. I need to think about C++ code a lot more before I write it than I do about Java code.
7. Leave enough time for testing and debugging! Allow for this when choosing what to do. Allow at least as much time for debugging and testing as for designing and writing the code. You don't want a coursework with potentially brilliant features which crashes too often to be able to demo them.

General requirements:

The following general requirements are mostly the same wording as in part 3, and summarises the key requirements for the coursework. Specific marking criteria are shown below.

1. Create a program which runs without crashing or errors and meets the functional requirements listed below and on the following page. Look at the Hall of Fame page to see some examples of previous work and decide on a basic idea for your program. You don't need to implement that much for part 3, but part 4 will involve finishing it.
2. Your program features which are assessed must be different from all of the demos and from exercises A and B.
3. The aim of this coursework is the investigate your ability to understand existing C++ code and to extend it using sub-classing, rather than to use some alternative libraries:
    - Your program MUST use the supplied framework in the way in which it is intended. i.e. the way in which exercises A and B use it to draw the background and moving objects.
    - You must not alter the supplied framework classes.
    - You should not use any external libraries other than standard C++ class libraries, libraries used by the framework code.
4. Demo your work to a lab helper and have it marked. There are 8 compulsory requirements, worth 1% of the module mark each, and you can do up to 12 of the optional ones for 1% each, for a total of 20% of the module mark overall.
5. Zip up your completed project and submit it to part 4 coursework submission.

There are 8 mandatory requirements and a number of optional advanced requirements to choose from (some of which are relatively easy but I didn't want to make them compulsory). **Each of the basic requirements will get you 1 mark for correct completion.** You can also get up to **12 marks for completing up to 12 of the optional advanced criteria** (1 mark each, so there is no need to do more than 12). The aim of the marking scheme is that someone who fully completed courseworks part 1, 2 and 3, did only the mandatory requirements for this part would still get 28/40 for the coursework – the equivalent of a first class mark (70%). For this reason please use your time wisely – if you find this coursework easy or fun then feel free to go for the advanced marks. If it's a struggle and time consuming then consider only going for the mandatory requirements and you can still get a good mark overall on the coursework side while getting some experience of working within a class framework.

Functional requirements overview:

If your program does not work properly, has bugs, or crashes then you will lose marks. (Consider carefully whether to add complex features which may crash it – you could lose more marks than you gain!).

To show that these requirements do not apply only for games, in case you want to create something else, I have given an example for each of what it may mean for a word processor application.

You MUST complete and print a documentation sheet before marking, specifying which requirements you did and giving one or more screenshots for reference. You MAY print the documentation in black and white rather than colour if you wish – or could show off the appearance of the program in colour if you prefer.

Mandatory (compulsory) requirements: (YOU NEED TO DO ALL 8 OF THESE)

You should try to complete all of these requirements. In many cases more advanced versions are in the optional requirements, so I suggest to read those as well before starting on these in case you can pick up easy marks by doing a little bit more at the same time.

1. Appearance looks OK and appropriate

This criterion is all about how the program looks. Does it look attractive or does it look like no effort was put in? Is the background appropriate or just blank?

Marking Criteria: Correctly drawn backgrounds for each stage, which are different to the demos, and are not trivial (e.g. a blank background). This could use an image, a tile manager or be drawn using the fundamental drawing functions on the base engine.

In a word processor application this could mean make it look attractive and show the text correctly.

2. Provide multiple states/stages.

Marking Criteria: You added two or more stages which the game goes through and which have different backgrounds for the screen and different behaviours for moving objects (e.g. one may not have any moving objects).

In a word processor application this could mean making it look attractive and show the text correctly.

3. Use the tile manager appropriately – changing at least one tile

Note that you already displayed the tile manager on the background in the coursework part 3, so this requirement goes beyond that.

Marking Criteria: Have an interesting use of the tiles which makes it attractive and has some purpose. At least one of the tiles should change over time and should be correctly re-drawn. At a bare minimum, change the tile appearances to be different and make at least one of them change and get redrawn at some point – e.g. when clicked on.

In a word processor application you could use a tile manager for a tool bar at the top of the screen.

4. Improve your user-controlled moving object

Note that you provided a basic implementation for this in part 3.

Marking Criteria: Provide a displayable object which can be moved around by the user and has some differences from the demos. i.e. it is not a copy-paste of the demo code without functional modification. This basic implementation should work correctly and there should be some functionality occurring for both mouse and keyboard interaction. As a minimum *both* a key input *and* a mouse input must do something. Minimal examples from the past: 1) move using keyboard but change colour when player clicks on object,

2) move using mouse and change colour when a key is pressed, 3) two objects, one controlled by mouse and one by keyboard. See the bouncing ball demo for an example of using both mouse and keyboard input. Note that the assessment aim is to ensure that you are handling both mouse and keyboard input in the framework so keep this in aim in mind.

In a word processor your moving object would probably be a visible text cursor, which should move to where you click or be movable using the cursor keys.

5. Provide multiple different automated moving objects

Marking Criteria: You have *multiple* automated moving objects which move correctly around the screen. You should use `DisplayableObject` sub-classes for this and will need to have *at least two different subclasses*. The objects should have some basic behaviour, e.g. move back and forth or bounce off the screen edges, but this behaviour must differ in some way between them – i.e. they need some different code which changes their behaviour.

This is harder for something like a word processor but I'd probably implement it as something like the visible track for the cursor (a moving object which follows the cursor around the screen to help those with accessibility needs to see where the cursor is) or

6. Provide interaction between moving objects, or a moving object and background

Marking Criteria: *either* have (at least) two moving objects which interact appropriately with each other, so that when they collide something happens (e.g. they bounce) *or* have some interaction between the objects and background. Bouncing off the screen edge does not count for this, but drawing a rectangle on the background and having something happen when the object goes over it, or interacting with the tile manager would both count. Note: having a moving object change a tile when it goes over it would meet the relevant part of requirement 3 as well.

In a word processor this could be a tooltip box appearing when the cursor is over the tool bar, or suggestions for alternative words appearing when over a word with a spelling error.

7. Display meaningful changing text on top of any moving object on the screen

Marking Criteria: Note that you had to display some text on the screen for part 3, but this text here must appear on top of moving objects, not behind them. The other difference here is that the information should be meaningful, e.g. showing a context-sensitive timer, or a score. It's not very different though from the part 3 requirement, so should be relatively straightforward.

A word processor could display word count or current line number in a status bar.

8. Program works well and looks good.

Marking Criteria: You did more than the basics and ensured that your program looks good and works well, rather than just 'it has an appearance different to the demos' and 'doesn't always crash'.

This mark is awarded to any program which works well and looks good/appropriate. Basically it means that if you put in the effort to make it work properly and look nice then you get this mark. If the marker looks at it and thinks 'looks nice' *and* when it is used it works properly and well, with no problems, you will get this mark. After using a similar coursework for a number of years it's relatively easy to see whether students put at least some effort into thinking about this. As a minimum this means: you made some effort with the graphical appearance, the background is relevant and not plain or the same as any of the demos, including at least some use of relevant shapes (e.g. separating off a score by putting it in a box and labelling it) and moving objects are not just plain circles or squares. Completing the other requirements will get a long way towards this anyway, so it's not as hard as it may seem.

Optional requirements (CHOOSE UP TO 12 OF THESE TO DO):

A. Load some data: Note that you did some loading of data for Hangman in part 2 so this should not be very new. Load some data correctly from a file, which is then used by the program. The easiest implementation would be to load something which is used as a window title or is displayed on the screen.

B. Advanced data loading: Load some non-trivial data, such as a document that you will appropriately display or the level data for a level of a game. If you meet this criterion then you will automatically also meet requirement A.

C. Data saving: Save some non-trivial data, such as a sorted set of high scores. This should involve writing multiple different values in a format which can be meaningfully used later (e.g. loaded back in).

D. Save/load non-trivial state: This means to provide a method for the user to save/load the current state of the program, so that they can return to the current state later. This requirement means that it correctly saves some non-trivial state – e.g. at least the positions of all objects and information about which state the game is in, score, etc. Note that doing so may also gain you the marks for advanced data loading and data saving above.

E. Advanced (e.g. animated/scrolling) background: A complex background. e.g. a well-designed and appropriately implemented animated or simple scrolling background. Note: this could be relatively hard if you don't fully understand what the surfaces are doing in the BaseEngine framework. You MAY create a new surface in your subclass if you need it. One person last year had flickering torches animating the background and a scrolling background – in that sort of situation I would suggest to count the scrolling background for this and to ask me about the flickering torches as an extra advanced mark (see criteria U and V below).

F. Animated appearance of user controlled object and/or automated objects: one or more of your displayable objects will change appearance in a meaningful way over time. E.g. a ball which grows and shrinks, or an animated character. This must be different from the samples and demos though! Hint: the draw function for the object can look at the current time to determine which frame to draw, how big to make the drawing, etc.

G. Displayable object images: Use an image for at least one of the automated displayable objects. This involves loading the image and drawing it to the foreground (not the background). You should load the image only once and keep it in the DisplayableObject, NOT keep reloading it each frame if you want this mark. If you use multiple images then you could also meet the animated appearance criterion if you cycled them appropriately.

H. Creating new displayable objects during the game: meeting this requirement means that you can dynamically add one or more displayable objects to the game temporarily after it has started and that this works correctly. These could disappear again after a while. For example, add an object which appears and moves for a while for the player to chase if a certain event happens. Adding a bomb that can be dropped, or a bullet that can be fired also would meet this requirement if you implement these as displayable objects. Something like pressing a key to drop a bomb which then blows up later, while displaying a countdown on the bomb, and then changes the tiles in a tile manager would meet a number of requirements in one feature.

I. Allow user to enter text which appears on the graphical display: For example, when entering a name for a high score table, capture the letter key presses and pressing of delete key, and show the current string on the screen (implementing delete at well may be important). This needs thought but is useful to demonstrate your understanding of strings. I added this optional requirement because some people did it anyway last year for entering high score tables and I wanted the marking scheme to reflect that it was done by giving a mark for it. Entering text into the console window does not count for this.

J. Display text aligned with moving objects: This means something like labelling the moving objects with text (or a tooltip on user controlled object?), e.g. a health score floating over a displayable object. The key features are that the text moves when the object does and it is redrawn appropriately without leaving a mess on the screen.

K. Complex intelligence on an automated moving object. As a minimum this criterion would involve something more than moving randomly or homing in on a player. E.g. it could involve something like 'if player is on the same column then home in, otherwise move randomly', or 'keep the same direction until I bump into something then change direction towards player and repeat'.

L. Impressive intelligence on an automated moving object: A really good implementation of the intelligence of a moving object, e.g. to use a shortest path algorithm to find the shortest way through a maze to get to a player, or predicting a player's path and moving towards that rather than the player itself, or showing some apparent intelligence, would get both marks K and L (2 marks rather than 1). Note that the important thing for marking here is the skill you show in your C++ ability by implementing this.

M. More complex tile manager interaction: This means tiles which change appropriately according to specific conditions and interactions with displayable objects, rather than just the simple tile changing which was in requirement 3. Previous examples include things like 'player presses a button in (i.e. moves into) a tile and a door opens in another tile' (the screen should show the change in the door, for example).

N. Implement a hierarchy of moving object classes: This means create a number of displayable object subclasses where some are sub-classes of sub-classes of displayable object classes. The aim of this requirement is for you to demonstrate your understanding of adding features using subclassing so that different subclasses will add features. It should be clear why intermediate sub-classes exist and what functionality they add that is shared by their own subclasses (i.e. the hierarchy is meaningful). E.g. in the past I saw a hierarchy where there were there were Player, Bullet and Enemy subclasses of DisplayableObject, and there were then various different enemy sub-classes which added different features for different types of enemies, but some behaviour applied to all enemies (e.g. ability to be shot).

O. Non-trivial pixel-perfect collision detection: If you implement some collision detection which is accurate to the pixel level for something other than just rectangles interacting with rectangles then you get one or more advanced marks. Circle-circle interaction would give this mark if it is not just equivalent to rectangle-rectangle interaction (i.e. not 'bounding rectangle' overlaps).

P. More complex collision detection: If you do something more complex than circle-circle interaction, such as Circle-Rectangle interaction (edge of circle interacting with any part of rectangle – which is harder than you think due to having to consider both corners and edges) then you get mark O and mark P (i.e. 2 marks).

Q. Really complex collision detection: If you do something really complex, such as complex outline interactions (e.g. someone did bitmap-bitmap interaction in the past, checking for coloured pixels interacting, and someone else split complex shapes into triangles which could be collision detected) then you get marks O, P and Q (i.e. **3 marks**). Note that this is hard to get, so if your implementation is not solving a really complex task then you probably will get marks O & P instead of all three.

R. Polymorphic state structure: Implement a polymorphic class structure for the state model (sub-type polymorphism). Look up the 'State Pattern' to see what this means and think about it. It's an advanced mark so we will not explain how to do this beyond the following: there will be a basic state base class and a subclass for each of the different states. Your BaseEngine sub-class will need to know which state object is currently valid and the different methods will call virtual methods on this object. The different behaviour will therefore occur due to having a different object being used for each state rather than having a switch in each of the methods. If you have if or switch statements specifying what to do in different states then you won't have done in properly so you won't get this mark. You DO NOT need more than the one sub-class of BaseEngine. If you had to create multiple sub-classes of BaseEngine then the implementation is wrong (and there will be other issues since you will have more than one window as well). If you did requirement N, then thinking about what that actually did may help you with this requirement.

S. Implement full pause facility: Implement a full pause facility. This should mean that all moving objects stay stationary and then carry on normally (from where they paused) when the program is unpaused without suddenly jumping to new positions. Hint: you need to not count pause time when working out positions – look at the methods of BaseEngine.

T. Sellable quality: This is supposed to be hard to get. It basically means that the marker and Jason both went 'wow' when they saw this and thought that people would easily pay money to buy this program.

Probably not one to aim for since any program meeting this will already get the advanced marks elsewhere I would expect, but it gives us the opportunity to comment 'wow!' about your program. In some cases it is possible that a program could be sellable while not implementing too many of the other features, since some app store programs are relatively simple, for example.

U. Another advanced feature I didn't think of: I hope that this is rare and that most features fit the earlier criteria, but this gives you the opportunity to do something else really complex that I didn't think of and still get a mark. You should discuss any advanced feature with Jason in advance and get approval for it to be able to gain a bonus mark if appropriately implemented. Any feature which is not pre-approved will not be eligible for this mark. Note that to be eligible the feature needs to illustrate C++ knowledge and/or ability to work within the supplied framework and should not be simple/trivial to do. E.g. an extraordinarily complex (in terms of C++ code) feature to do something I didn't think about. *I repeat: to get this mark you MUST have agreed it with Jason in advance and he will not always agree things that you suggest if they do not show the required skills/abilities.*

V. A second advanced feature: This is even less likely to be needed that requirement U, but it allows you to do a second, completely different, thing which also matches the criteria for requirement U. In past years sometimes students have done things which we thought were brilliant but were not in the marking scheme. Adding this as well as requirement U basically allows up to two of these to be agreed in advance to get marks. *Again you need to agree this with Jason in advance so that the criteria can be determined and marking is feasible and fair.*

Final comments:

The coursework changed this year. It is a lot shorter than last year (in terms of what you actually need to do – noting that you should not do more than 12 of the optional requrirements) and most of the complex things are now options rather than compulsory, so you can avoid things that you really hate to debug (e.g. read/write files) if you wish. I have also emphasized to spend reasonable time on the coursework rather than far too much time. These changes are in response to student suggestions last year to add some short courseworks earlier in the module, so some of the marks went there, and that it would be better with more options rather than having all compulsory requirements, so that people could pick the appropriate ones for their program.

Take advantage of the labs to find out how to implement things.

Take advantage of the Friday time to ask me questions about how to do things – so I can show everybody. Please look on this time as an office hour as well as a demo hour.

Start early. *There is a lot of time allocated so that you have time to think about and fix bugs, not so you can start late*. Sometimes you need to take a break and come back the next day when you get a bug – C++ is HARD to debug! Allow for this.

Documentation sheet: you need to complete one of these for your coursework

Your name/id: <What is your name and student Id – so we can identify you>

Coursework name: <Pick a name for your program>

Summary: <summarise what your program does. E.g racing game, word processor, pacman, bomberman, etc and list any key features you want to highlight which are not covered below>

<INSERT ONE OR MORE SCREENSHOTS HERE TO ILLUSTRATE YOUR GAME>

Mandatory (compulsory) requirements: tick them and add a few words of comment if it helps you/us to know what to demo/mark

1. **Appearance looks OK and appropriate**
2. **Provide multiple states/stages**
3. **Use the tile manager appropriately – changing at least one tile**
4. **Improve your user-controlled moving object**
5. **Provide multiple different automated moving objects**
6. **Provide interaction between moving objects, or a moving object and background**
7. **Display meaningful changing text on the screen**
8. **Program works well and looks good**

Optional requirements – which did you do and what did you do to complete these? Add a few keywords to remind you what to demo.

A. Load some data
B. Advanced data loading
C. Data saving
D. Save/load non-trivial state
E. Advanced (e.g. animated/scrolling) background
F. Animated appearance of user controlled object and/or automated objects
G. Displayable object images
H. Creating new displayable objects during the game
I. Allow user to enter text which appears on the graphical display
J. Display text aligned with moving objects
K. Complex intelligence on an automated moving object
L. Impressive intelligence on an automated moving object
M. More complex tile manager interaction
N. Implement a hierarchy of moving object classes
O. Non-trivial pixel-perfect collision detection
P. More complex collision detection
Q. Really complex collision detection
R. Polymorphic state structure
S. Implement full pause facility
T. Sellable quality
U. Another advanced feature
V. A second advanced feature