1 Submission Instructions

Submit to Brightspace on or before the due date a compressed file (.tar or .zip) that includes

1. Header and source files for all classes instructed below.
2. A working Makefile that compiles and links all code into a single executable. The Makefile should be specific to this assignment - do not use a generic Makefile.
3. A README file with your name, student number, a list of all files and a brief description of their purpose, compilation and execution instructions, and any additional details you feel are relevant.

2 Learning Outcomes

In this assignment you will learn to

1. Write an application where we begin to separate into control, view, entity, and collection object classes.
2. Use collection classes instead of raw arrays.
3. Use a UML diagram to implement classes and the interaction between between classes.
4. Implement proper memory management when using dynamic memory.
5. Implement proper encapsulation (using the `const` keyword where appropriate)
   VERY IMPORTANT!!! YOU WILL LOSE MARKS IF YOU DO NOT CONST YOUR FUNCTIONS AND PARAMETERS!!!.

3 Overview

We will continue making our X11 GUI wrapper by making a `FlowPanel`, which is a `Panel` that supports a flow layout of components (defined below), and a `TextArea`, which is a rectangle that displays text.

Instead of storing these components in raw arrays, we will introduce two collection classes, `TAArray` and `PanelArray`, that add an abstraction layer.

We will be putting as much extraneous memory on the heap as possible. As such, pay careful attention to the ownership rules defined for these classes (i.e., rules determining what memory they should delete when they are destroyed). You will also implement a *deep copy* in the `FlowPanel` copy constructor.

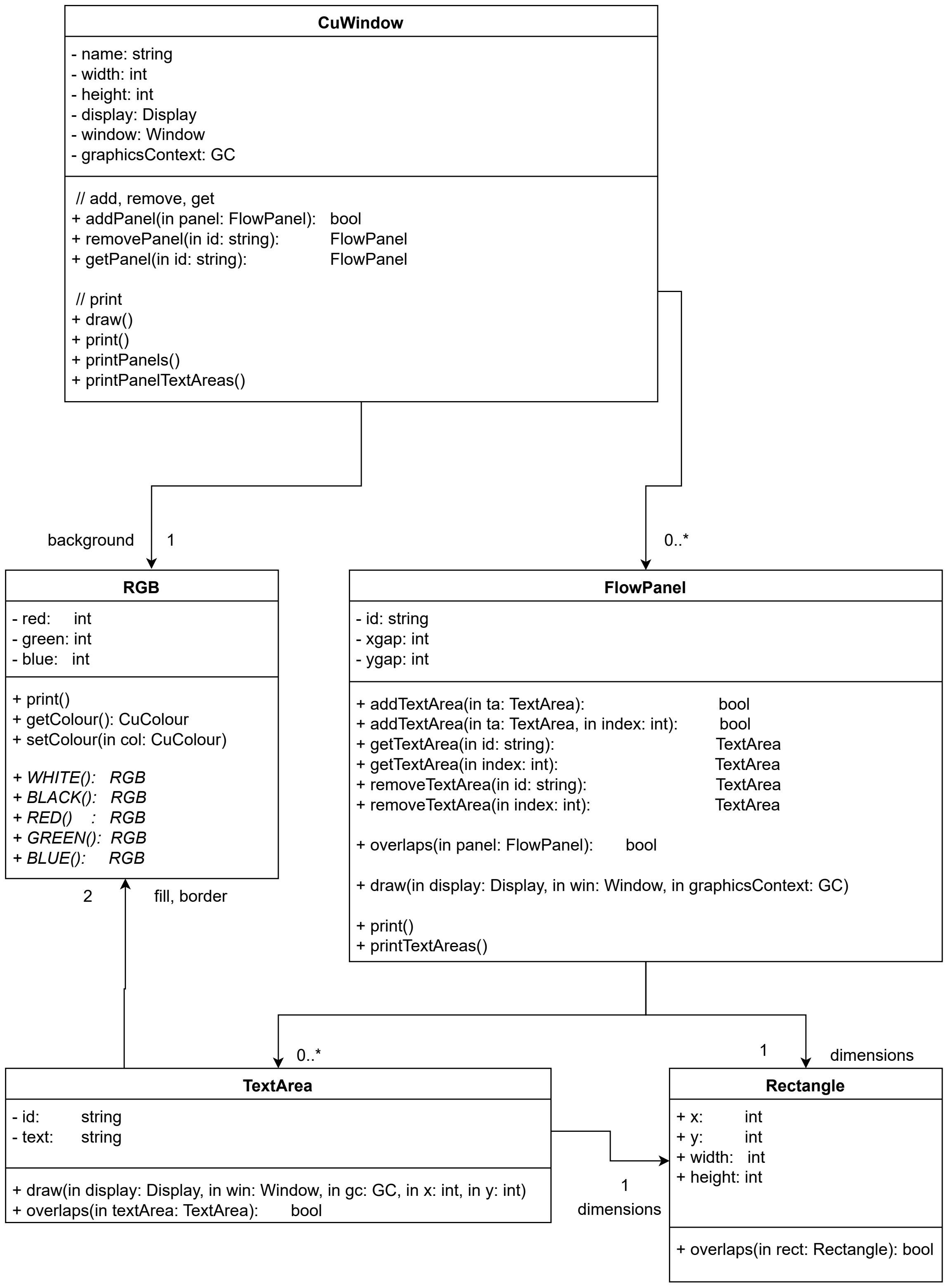
In addition, to all classes except for the `Tester`, `TestControl`, and `View` classes, you should apply the `const` keyword wherever possible.

We are implementing our new design - there a `TestControl` class that implements tests on the classes that you develop, and a `View` class that we will use to interact with the user.

To assist you in writing these classes a UML diagram is provided. You should use this UML diagram to understand the relationships between the classes and to verify function signatures.

The emphasis of this assignment is on **encapsulation** and proper **memory management**.

4 UML Diagram


CuWindow
- name: string
- width: int
- height: int
- display: Display
- window: Window
- graphicsContext: GC
// add, remove, get
+ addPanel(in panel: FlowPanel): bool
+ removePanel(in id: string): FlowPanel
+ getPanel(in id: string): FlowPanel
// print
+ draw()
+ print()
+ printPanels()
+ printPanelTextAreas()
background
1
0..*
RGB
- red: int
- green: int
- blue: int
+ print()
+ getColour(): CuColour
+ setColour(in col: CuColour)
+ WHITE(): RGB
+ BLACK(): RGB
+ RED() : RGB
+ GREEN(): RGB
+ BLUE(): RGB
2
fill, border
FlowPanel
- id: string
- xgap: int
- ygap: int
+ addTextArea(in ta: TextArea): bool
+ addTextArea(in ta: TextArea, in index: int): bool
+ getTextArea(in id: string): TextArea
+ getTextArea(in index: int): TextArea
+ removeTextArea(in id: string): TextArea
+ removeTextArea(in index: int): TextArea
+ overlaps(in panel: FlowPanel): bool
+ draw(in display: Display, in win: Window, in graphicsContext: GC)
+ print()
+ printTextAreas()
0..*
1
dimensions
TextArea
- id: string
- text: string
+ draw(in display: Display, in win: Window, in gc: GC, in x: int, in y: int)
+ overlaps(in textArea: TextArea): bool
1
dimensions
Rectangle
+ x: int
+ y: int
+ width: int
+ height: int
+ overlaps(in rect: Rectangle): bool

5 Classes Overview

This application will consist of 9 classes and one struct. `Rectangle` is a struct provided in `defs.h` to store location and dimension information. It also has an `overlaps` function that you can use. Three complete classes are provided: `View`, `TestControl`, `Tester`. `TextArea` and `CuWindow` are also supplied, but you must complete them. The remaining classes that you will provide are listed in the UML diagram above. In addition there are `PanelArray` and `TAArray` collection classes. All classes are listed below along with their respective categories. You should refer the instructions and the UML diagram to construct your classes.

1. The `Rectangle` struct (primitive data)
   (a) Contains draw location and size information stored as `ints`.
   (b) Implements an `overlaps` function.
2. The `RGB` class (Entity object):
   (a) Contains colour information.
3. The `TextArea` class (Entity object):
   (a) A rectangular area that contains text.
4. The `FlowPanel` class (Entity object):
   (a) A rectangular area that contains `TextAreas`. When drawn the `TextAreas` are arranged in a flow layout (defined below).
5. The `TAArray` class (Collection object):
   (a) Data structure for `TextAreas`.
6. The `PanelArray` class (Collection object):
   (a) Data structure for `FlowPanels`.
7. The `CuWindow` class (Control, Collection, View object):
   (a) Manages a collection of `FlowPanels`.
   (b) Provides services to add, remove, and access `FlowPanels`, and to print the `FlowPanels` and `TextAreas`.
8. The `View` class (Boundary object):
   (a) Presents a menu, takes input from the user
9. The `TestControl` class (Control object):
   (a) Manages the interaction of the other objects in order to run tests.
10. The `Tester` class (???):
    (a) Provides testing functionality.

6 Instructions

Download the starting code from Brightspace. It includes some global functions that you are to use for testing as well as some classes. All member variables are `private` unless otherwise noted. All member functions are `public` unless otherwise noted. Some return values are not explicitly given. You should use your best judgment (they will often be `void`, but not always). ALL CLASSES MUST HAVE A PRINT FUNCTION (except for `PanelArray` and `TAArray`). This print function should display the metadata of the class using appropriate formatting.

Your finished code should compile into an executable called `a2` using the command `make all` or simply `make`. The Makefile is provided for you. Your submission should consist of a single zip file with a suitable name (e.g., `assignment2.zip`) that contains a folder containing all your files. This folder should also contain a README with your name, student number, a list of all files that are included, a directory structure (if applicable), compiling and running instructions, and any other information that will make the TAs life easier when they mark your assignment.

6.1 The Tester Class

This class is provided for you. You should not modify this class.

6.2 The Rectangle Struct

This is a struct, rather than a class, because we really just want 4 `ints`, but packing them into a struct is more convenient. Contains `x`, `y`, `width`, and `height` stored as `ints`. (Note there is an `XRectangle` struct that does the same thing, but stores these values as `shorts` instead of `ints`. We provide our own implementation to avoid type conversion warnings.) It also has an `overlaps` functions to do intersection testing on rectangles. All members are public.

6.3 The RGB Class

This is the same class as before, but you will add some **static** functions. Rather than use `CuColour` constants like last time for commonly used colours, you will provide static functions that return instances of `RGB` objects for commonly used colours. The functions you must supply are listed below. Each should return an `RGB` object with the appropriate values. Remember, these are **static** functions.

1. `WHITE(), BLACK(), RED(), GREEN(), BLUE()`

6.4 The Array Classes

You have been given a slightly modified version of `StudentArray` from Chapter 8: Object Design Categories. You can use this as a template to make two other collection classes.

6.4.1 PanelArray

This class should use a dynamically allocated array of `FlowPanel` pointers as a backing array.

1. Change the `add` function so that `Channels` are added to the **back** of the array.

6.4.2 TAArray

This class should use a dynamically allocated array of `TextArea` pointers as a backing array.

1. Change the `add` function so that `TextAreas` are added to the **back** of the array
2. Make a second `add` function that takes two parameters, a `TextArea` pointer and an `int index`. Add the `TextArea` at the given index IF IT IS A VALID INDEX, i.e., an index between 0 and the number of `TextAreas` currently in the array. Return true if successful and false if the array is full or the index is not valid.

6.5 The TextArea Class

This is very similar to the Button class from assignment 1, in that it draws a rectangle and places text on it. You may wish to reuse some of that code. The TextArea class has member variables for position and dimensions, and the text that is displayed on the TextArea, and an id unique identifier. A TextArea is stored inside a FlowPanel, rather than in CuWindow directly.

1. Member variables:

   (a) `Rectangle dimensions`: (you can shorten the name from `dimensions` if you wish) the preferred `x` and `y` coordinates of the TextArea within its current Panel (these coordinates are used in a regular Panel with an absolute layout - these are ignored in a FlowPanel), along with the `width` and `height`.

   (b) `string text`: The text that is displayed on the TextArea.

   (c) `string id`: The unique id of the TextArea.

   (d) `RGB fill, border`: the colour of the TextArea and the TextArea border.

2. Constructors:

   (a) A constructor that takes `x, y, width, height, id, label, fill, border` as arguments and initializes the member variables appropriately. `fill` and `border` should be `RGB` objects. `fill` should have a default value of white, and `border` should have a default value of black.

   (b) A constructor that takes `Rectangle, id, label, fill, border` as arguments and initializes the member variables appropriately. `fill` and `border` should be `RGB` objects. `fill` should have a default value of white, and `border` should have a default value of black.

3. Member functions:

   (a) You should make getters and setters as needed.

   (b) A `void draw(Display *display, Window win, GC gc, int x, int y);` function. Draw the TextArea as a (filled) rectangle (see the X11 documentation) at the coordinates given. The `text` is drawn over top of the TextArea, left to right and top to bottom until it finishes the text or runs out of space. The `x` and `y` parameters are the x and y-coordinate **on the window** where this TextArea is being drawn. That is, the FlowPanel that contains the TextArea must first calculate the `x` and `y` coordinates where it is to be drawn on the underlying window, and pass these coordinates in as arguments.

   (c) The `bool overlaps(TextArea& ta)` - return `true` if `ta` overlaps this TextArea. Note there is an overlaps function in the Rectangle struct that you may call.

   (d) A `print` function. This should print (to the console, not to the Window) all the TextArea information. You may wish to `#include <iomanip>` for some formatting tools, but you do not have to. For an *acceptable* print output, see below.

```
TextArea id:        ta1
Preferred location: 10, 10
Size:               80, 50
Text:               This is a TextArea with wrapping text!
```

6.6 The FlowPanel Class

The `FlowPanel` class is similar to the `Panel` class, except that rather than an absolute layout (where the x and y are given explicitly) we will use a flow layout, where elements are laid out left to right, top to bottom.

Note: In the `CuWindow` class, `FlowPanels` themselves will be positioned using an absolute layout like in Assignment 1.

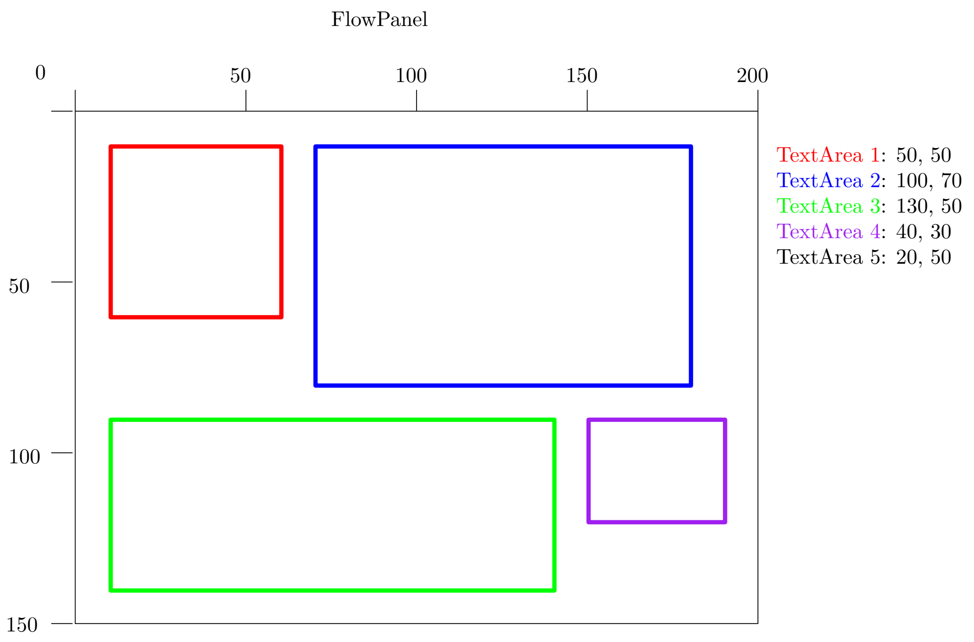
Note that you may wish to have 2 draw functions, one that uses absolute layout and one that attempts to implement the flow layout (described below). That way you can use your absolute layout for testing purposes.

1. Memory management.
   (a) `TextAreas` are not created in the `FlowPanel`. The user of the library should create a new `TextArea`, make any necessary changes to it, and add the `TextArea` pointer to the `FlowPanel`. At this point, the responsibility for the dynamic memory is transferred from the user to the `FlowPanel`. As such, when a `FlowPanel` is destroyed, it should delete every `TextArea` that it contains.
   (b) If a user calls `getTextArea` (which returns a pointer to a `TextArea` to the user, but does not remove the `TextArea` from the `FlowPanel`), the responsibility for deleting that `TextArea` still lies with the `FlowPanel`.
   (c) If a user calls `removeTextArea` then responsibility for deleting that `TextArea` is transferred to the user.
2. Member variables:
   (a) `string id`: The unique id of the `FlowPanel`.
   (b) `Rectangle dimensions`: the x and y coordinate of the `Panel` within its current `CuWindow`, and the width and height of the `FlowPanel` in pixels.
   (c) `int xgap, ygap`: the minimum amount of space between adjacent `TextAreas` on the x- and y-axes.
   (d) A `TAArray` to keep track of the contained `TextAreas`.
3. Constructors:
   (a) A constructor that takes `x, y, width, height, id, xgap, ygap` as arguments and initializes the member variables appropriately. The `xgap` and `ygap` parameters should have default values of 10.
   (b) A constructor that takes `Rectangle, id, xgap, ygap` as arguments and initializes the member variables appropriately. The `xgap` and `ygap` parameters should have default values of 10.
   (c) A copy constructor that does a DEEP COPY of the `FlowPanel`.
4. Member functions:
   (a) You should make getters and setters as needed.
   (b) See the UML diagram for functions for adding and removing `TextAreas`. These should pass and return `TextAreas` by pointer.
   (c) A `void draw(Display *display, Window win, GC gc)` function. Technically `FlowPanels` should be invisible. However, we will draw them as a(n unfilled) rectangle for debugging purposes (so we can ensure their placement is correct). In addition, draw all the `TextAreas` onto the `FlowPanel` in a flow layout, described below.

   (d) Flow layout: In general, this layout draws all `TextAreas` from left to right and wraps into the next row once it runs out of space. We will demonstrate flow layout with an example - see diagram below. Assume the `FlowPanel` is 200 pixels wide and 150 pixels high. It contains 5 TextAreas, with width and height shown below. We start drawing at 0,0. However, since the `xgap = 10` and `ygap = 10`, we leave space for 10 pixels (both horizontally and vertically), and TextArea 1 is drawn at 10, 10. Since TextArea 1 has a width of 50, then we attempt to draw the next TextArea at `x = xgap + width + xgap` = 10 + 50 + 10 = 70. Since we are still on the first row, the y-coordinate is still 10. TextArea 2 fits, so we draw it next.

At this point, `TextArea 3` will intersect the right side of the FlowPanel, so we start a new row. Out of the two TextAreas drawn on the previous row, the lowest point is at y = 80 (for TextArea 2). Since ygap = 10, the next row is drawn at y = 90. We draw TextArea 3 and 4.

TextArea 5 will intersect the right side of the FlowPanel. We attempt to draw it in the next row, but it will intersect the bottom of the FlowPanel. Since it does not fit, we stop drawing here. If there was TextArea 6, we would not draw it either, even if it fit.



A couple things to note. On each row, the tops of the `TextAreas` should be horizontally aligned. No `TextArea` should be drawn outside of the `FlowPanel`. To make this easier, if you wish, you can make each row the same height. So you would find the `TextArea` with the greatest height, and then every time you want to find the y-coordinate for a new row, you would add this height to the previous y-coordinate. Or figure out your own `TextAreas` layout - as long as it is not absolute layout (which we already did in assignment 1). There is a bonus mark for implementing flow layout correctly, and full marks for any sufficiently novel layout (like a modified flow layout, or grid layout for example). Please detail in your README the behaviour of your layout.

(e) A `print` function. This should print (to the console, not to the Window) all the `FlowPanel` information. You may wish to `#include <iomanip>` for some formatting tools, but you do not have to. For an *acceptable* print output, see below.

```
FlowPanel:     fp1
Position:      10, 10
Size:          20, 50
Num TextAreas: 2
```

(f) A `printTextAreas` function that calls `print` on every `TextArea` contained in the `FlowPanel`.

6.7 The CuWindow Class

The `CuWindow` class handles the X11 logic for making a display, opening a window and getting a graphics context for drawing. It also maintains a statically allocated array of (non-overlapping) `Panel` objects.

1. Memory management.
   (a) `FlowPanels` are not created in the `CuWindow`. A user should create a new `FlowPanel`, make any necessary changes to it (like adding `TextAreas`), and add the `FlowPanel` pointer to the `CuWindow`. At this point, the responsibility for the dynamic memory is transferred from the user to the `CuWindow`. As such, when a `CuWindow` is destroyed, it should delete every `FlowPanel` that it contains.
   (b) If a user calls `getPanel` (which returns a pointer to a `FlowPanel` to the user, but does not remove the `FlowPanel` from the `CuWindow`), the responsibility for deleting that `FlowPanel` still lies with the `CuWindow`.
   (c) If a user calls `removePanel` then responsibility for deleting that `FlowPanel` is transferred to the user.
2. Member variables:
   (a) `int width, int height`: the current width and height of the window in pixels.
   (b) `string name`: The name of the window (which should be displayed at the top)
   (c) A `PanelArray` to hold `FlowPanels`.
   (d) An `RGB` member for the background colour of the window.
3. In addition, these member variables are necessary to maintain and draw on an X11 window:
   (a) `Display* display`: Connection to the X server.
   (b) `Window window`: To store the XID of the window that we opened.
   (c) `GC gc`: A graphics context (so we can draw on the window).
4. Constructors:
   (a) A constructors that take `name, width, height, background` as arguments and initializes the member variables appropriately. The background should be an `RGB` object. The constructor should also open a display, a window, and create a graphics context.
   (b) You should also have a destructor. This should free the graphics context, destroy the window, and close the display, and clean up any memory necessary.
5. Member functions:
   (a) You should make getters and setters as needed.
   (b) `addPanel` - This function should take a `FlowPanel` pointer as an argument. If this `FlowPanel` does not overlap any other `FlowPanel` in the `CuWindow`, AND it does not extend outside of the `CuWindow` boundaries, then add this `FlowPanel` to the `PanelArray`. Return true if the `FlowPanel` is added, and false otherwise. Note that this is the same behaviour as in Assignment 1.
   (c) `removePanel` - This function should find the `FlowPanel` with the given id, remove it from the `PanelArray`, and return the pointer. Return `nullptr` if no such `FlowPanel` exists.
   (d) `getPanel`: This function should return a pointer to the `FlowPanel` with the given `id`, or else `nullptr` if no such `FlowPanel` exists.

(e) A `draw` function. You should first fill the window with a rectangle to "blank" everything out and provide a background colour (using the `RGB` member variable for the colour). Then you should draw all the `Panels` and their contents onto the window.

    `NOTE:` X11 does not synchronize by default. Thus, if there are changes being made to `CuWindow` and we attempt to draw, it may not be rendered properly. It is HIGHLY recommended that at the top of this draw function, before doing anything else, you sleep a bit so that any changes to `CuWindow` can be completed. Run the command `usleep(100000)` as the very first line of the `draw` function. You will also need `#include <unistd.h>` at the top of your file.

(f) A `print` function. This should print (to the console, not to the window) the name of the window and the number of `FlowPanels`.

(g) A `printPanels` function. This should print out all the `FlowPanels`.

(h) A `printPanelTextAreas` function. This should print out all the `TextAreas` from all the `FlowPanels`.

6.8 The TestControl Class

This class has been done for you. It interacts with your classes and the `View` to run a series of tests and output your mark.

6.9 The View Class

This class has been done for you. It interacts with the user and returns values to the control object

6.10 The `main.cc` File

These have been done for you. `main.cc` is compiled into an executable `a2`. The `a2` executable runs tests and gives you your mark.

7 Grading

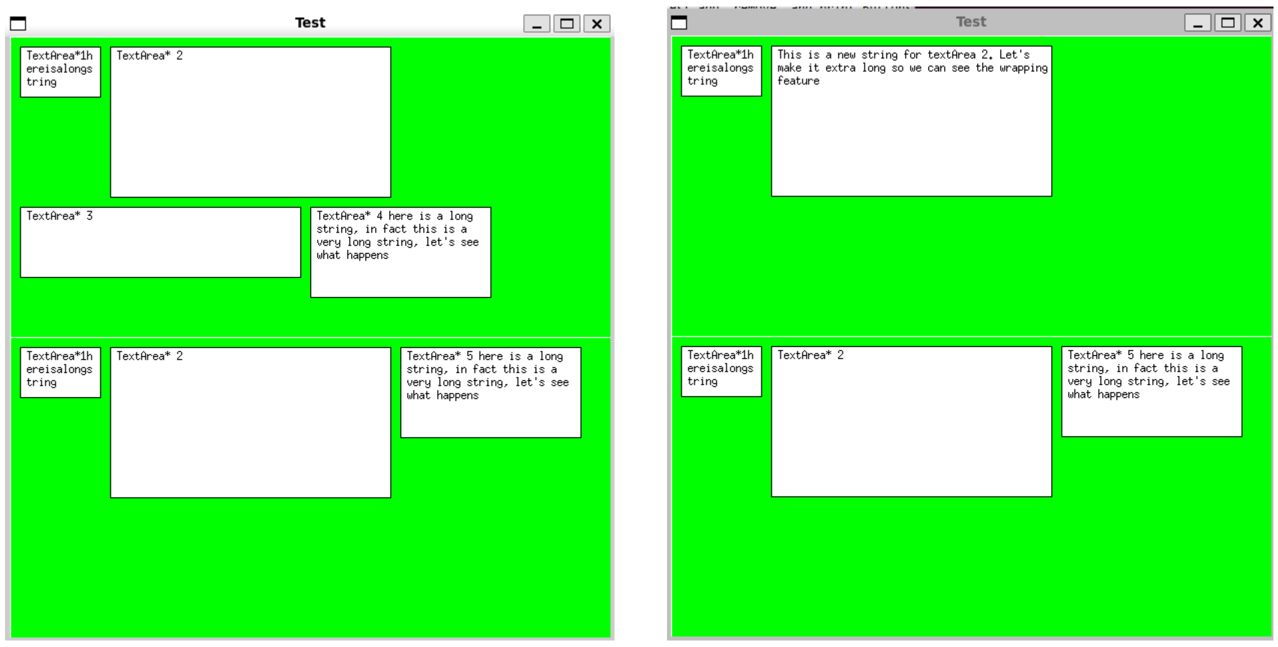
The marks are divided into three main categories. The first two categories, **Requirements** and **Constraints** are where marks are earned. The third category, **Deductions** is where you are penalized marks.

7.1 Specification Requirements

These are marks for having a working application (even when not implemented according to the specification, within reason). The test suite will automatically allocate the marks, though they can be adjusted by the marking TA if some anomaly is found.

General Requirements

- All marking components must be called and execute successfully to earn marks.
- All data handled must be printed to the screen to earn marks (make sure the various `print` functions print useful information).
- 1. [2 marks] TextArea print test.
- 2. [3 marks] FlowPanel print test.
- 3. [8 marks] FlowPanel and TextArea integration test.
- 4. [5 marks] CuWindow, FlowPanel and TextArea integration test.



```
TextArea*1h
ereisalongs
tring
TextArea* 2
TextArea* 3
TextArea* 4 here is a long
string, in fact this is a
very long string, let's see
what happens
TextArea*1h
ereisalongs
tring
TextArea* 2
TextArea* 5 here is a long
string, in fact this is a
very long string, let's see
what happens
```

```
TextArea*1h
ereisalongs
tring
This is a new string for textArea 2. Let's
make it extra long so we can see the wrapping
feature
TextArea*1h
ereisalongs
tring
TextArea* 2
TextArea* 5 here is a long
string, in fact this is a
very long string, let's see
what happens
```

Figure 1: Properly rendered window. Note that in the second figure, TextArea 3 is resized and no longer fits in the top `FlowPanel`.

- 5. [7 marks, manual inspection] Test render window, see example figures.
  - a) [1 mark] FlowPanels are drawn correctly
  - b) [2 marks] TextAreas are drawn in the correct location (1 mark each FlowPanel), any layout except absolute layout is acceptable.
  - c) [2 marks] Text is correct - at least one line should appear - see first diagram (1 mark each FlowPanel).
  - d) [2 marks] Text is correct - note the change in TextArea 2. At least one line should appear - see second diagram (1 mark each FlowPanel).
- 6. [7 marks, memory management]
  a) [3 marks] FlowPanel memory test
  b) [4 marks] CuWindow memory test

Possible bonus marks:

- 1. [1 mark] Correctly implement flow layout for `TextAreas` in the `FlowPanel`.
- 2. [1 mark] Proper text wrapping in the `TextArea`, does not go outside the rectangle either to the side or below.
- 3. [1 mark] Have the window redraw when exposed or resized (perhaps in its own thread). You may use global functions or variables for this - it just needs to work. Please detail the design in your README.

Requirements Total: 32 marks (plus 3 bonus)

7.2 Constraints

The previous section awards marks if your program works correctly. In this section marks are awarded if your program is written according to the specification and using proper object oriented programming techniques. This includes but is not limited to:

- Apply "`const`"-ness to your program.
  - Print statements, getters, and any member function that does not change the value of any member variables should be `const`.
  - Any parameter object (passed by reference) that will not be modified should be `const`.
- Proper declaration of member variables (correct type, naming conventions, etc).
- Proper instantiation of member variables (statically or dynamically)
- Proper instantiation of objects (statically or dynamically)
- Proper constructor and function signatures.
- Proper constructor and function implementation.
- Proper use of arrays and data structures.
- Passing objects by *reference* or by *pointer*. Do not pass by value.
- Reusing existing functions wherever possible *within reason*. There are times where duplicating tiny amounts of code makes for better efficiency.
- Proper error checking - check array bounds, data in the correct range, etc.

7.2.1 Constraints: 12 marks

1. 2 marks: Proper implementation and `const`-ing of the `RGB` class.
2. 2 marks: Proper implementation and `const`-ing of the `TextArea` class.
3. 2 marks: Proper implementation and `const`-ing of the `TAArray` class.
4. 2 marks: Proper implementation and `const`-ing of the `FlowPanel` class.
5. 2 marks: Proper implementation and `const`-ing of the `PanelArray` class.
6. 2 marks: Proper implementation and `const`-ing of the `CuWindow` class.

Constraints Total:	**12 marks**
Requirements Total:	**32 marks**
Assignment Total:	**44 marks**

7.3 Deductions

The requirements listed here represent possible deductions from your assignment total. In addition to the constraints listed in the specification, these are global level constraints that you must observe. For example, you may only use approved libraries, and your programming environment must be properly configured to be compatible with the virtual machine. This is not a comprehensive list. Any requirement specified during class but not listed here must also be observed.

7.3.1 Documentation and Style

1. Up to 10%: Improper indentation or other neglected programming conventions.
2. Up to 10%: Code that is disorganized and/or difficult to follow (use comments when necessary).

7.3.2 Packaging and file errors:

1. 5%: Missing README
2. 10%: Missing Makefile (assuming this is a simple fix, otherwise see 4 or 5).
3. up to 10%: Failure to use proper file structure (separate header and source files for example), but your program still compiles and runs
4. up to 50%: Failure to use proper file structure (such as case-sensitive files and/or Makefile instructions) that results in program not compiling, but is fixable by a TA using reasonable effort.
5. up to 100%: Failure to use proper file structure or other problems that severely compromise the ability to compile and run your program.

As an example, submitting Windows C++ code and Makefile that is not compatible with the Linux VM would fall under 4 or 5 depending on whether a reasonable effort could get it running.

7.3.3 Incorrect object-oriented programming techniques:

- Up to 10%: Substituting C functions where C++ functions exist (e.g. don't use `printf` or `scanf`, do use `cout` and `cin`).
- Up to 25%: Using smart pointers.
- Up to 25%: Using global functions or global variables other than the `main` function and those functions and variables expressly permitted or provided for initialization and testing purposes.

7.3.4 Unapproved libraries:

- Up to 100%: The code must compile and execute in the default course VM provided. It must NOT require any additional libraries, packages, or software besides what is available in the standard VM.
- Up to 100%: Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*