University of Illinois at Urbana-Champaign
Department of Computer Science

MP3: Inheritance and Virtual Functions

CS 225 Data Structures
Spring Semester, 1999

Handed out: Saturday, February 13, 1999
Due date: Friday, February 19, 1999 at 5:00 PM

1 Introduction

This MP will be your third and final MP that deals with mainly C++ syntax. On this MP, you will observe the usage of a handful of classes that are arranged in an inheritance hierarchy. Then, you will add to this inheritance hierarchy yourself.

2 The programming assignment

To begin with, you will need to copy the given files into your own directory. The files are located in:

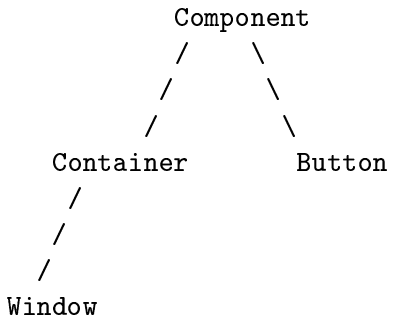
```
~cs225/src/mp3         (MP code)
~cs225/src/mp3/test    (test files)
```

2.1 The given code

The first part of the coding assignment involves reading through and understanding the given code. To start with, there are a few file pairs you have seen already, namely the `asserts.*` files as well as the files for the `Array` and `String` classes.

In addition, a `Point` class has been provided, which basically just provides an object-oriented way in which to deal with integer coordinate pairs.

Finally, there are four classes – `Component`, `Button`, `Container`, and `Window` – arranged in an inheritance hierarchy as follows:



That is, Component is the base class of the hierarchy, Container and Button are derived from Component, and Window is derived from Container.

These classes, which are loosely based on the classes of the same name from the Java AWT (abstract windowing toolkit) hierarchy, are tools with which to deal with GUI elements. Unfortunately, we cannot draw graphics to the screen here – C++ does not have standard graphics built in, and we have no way to test them in handin anyway – so instead we will use output print statements and other statements to represent what would be happening. For example, in the tempMain.C file given with the original MP3 code distribution, we can simulate a user clicking on a point in a GUI window by creating a `Window` object, and then calling the `HandleMouseClick` function off that object. The results get printed to the screen in the form of ``This button has been clicked on!'' types of printed output statements, as can be seen in test.temp.std.

So, what is going on in the actual code? Well, the class `Component` represents a general graphical element. There are various types of `Components` that you can imagine in a graphical system, among them windows and buttons. Each of these components would draw itself in a different way, it would handle mouse clicks in a different way, it could be scaled in size and could perhaps be moved to a different position on the screen, and so on. The goal of the `Component` class is to provide a set of common interface functions that *all* types of `Component`s need to support. So, we know that every type of component – be it a window, or a button, or whatever – allows you to set the component's size, because that is a member function of the `Component` class. Likewise, we know that every component can draw itself, via the function `Draw()` which is in the `Component` class. And so on.

The class `Button` is one specialization of this. We can create a button, with a particular size, at a particular location, and perhaps with a "label" on that button which more or less serves as a name. Then, that button must support the same kinds of behaviors that all `Component` objects must support – `Draw()`, `SetSize()`, etc. We promise in `Component` that these operations can be called on all `Component` objects – including objects of classes derived from `Component` – and so we cannot promise less when we have a derived class. We must support the behavior that we promised we would support in the base class.

In addition, however, the button can add button-specific behavior of its own, and in fact our `Button` class gives you the ability to read the name label of the button. We can't put this ability in `Component`, because not every type of `Component` necessarily has a name label to print. But we know that if we were to create a `Button`, it has at least an "empty" name label, so we can give the `Button` interface this power even if such power is not accessible in general to the entire `Component` hierarchy.

Notice, too, that there were certain functions in `Component` whose declarations ended in `= 0;` and which weren't even defined. Those are *pure virtual functions*. What we are saying is that we aren't going to supply code for these functions in the `Component` class. We want all `Component`-derived classes to support these operations, but they are so specific to the particular class that we could not begin to provide a definition for them here. So, we just say, "derived interfaces must support these functions (for example, `Draw()` somehow" and leave it at that. This makes `Component` an *abstract class* – we cannot actually allocate objects of this class, because its definition is incomplete (three member functions are missing definitions). However, we can use this incomplete class as a half-completed blueprint, from which we derive (using inheritance) a more specific blueprint for more specific components. But, each of these components will have the common behaviors outlined in this half-completed `Component` blueprint.

`Button`, on the other hand, is a *concrete class*. In addition to whatever other member functions it has, and in addition to whatever other defined member functions from `Component` it may override (actually, `Button` does not override any already-defined `Component` mmember functions, though `Container` does), it fills in the definitions for the pure virtual functions. Now, even though we may use this blueprint (the `Button` class) as the base class of an even more detailed blueprint (for example, a "Colored Button" class), we at least have a complete blueprint now and thus we can

allocate objects of `Button`. This, again, was unlike `Component` – we could not allocate objects of type `Component`, we could only use that class to derive more specific classes from.

We can still declare *pointers* to `Component` objects, though. It is just that, ultimately, those pointers will need to point to objects of concrete and more specific types, such as `Button`.

Now, let's look at the third class, `Container`. This class has the ability to store many `Component` objects within itself, yet we can also view it as a `Component` in its own right. So, we could perhaps have a `Container` of 5 slots, and in those slots hold three `Button` objects, and 2 objects that were the hypothetical "Colored Button" objects we spoke of earlier above, or perhaps instead we could use those last two slots to hold "checkbox" objects or "information box" objects (where you have space to type in a value) or perhaps just simple tiles that do nothing.

This class overrides `SetSize()` and `SetLocation()`, because if you have an array of, say, five objects, and you shrink the width of the array by halving it, then it is necessary for all the components stored in the array to have their widths halved as well. The `Container` is in a way the "cell border" class, and what we actually put *in* the cells are other `Component` objects (or, more specifically, objects of `Component`-derived classes). But, if we were to shrink the cell borders by squeezing everything closer together, then the items inside the cell borders would have to shrink as well. We cannot very well say, "give me five 10x10 cells and inside each one, place a 100x100 button". If the cells are only 10x10, the buttons can be only 10x10.

So, the `Container` functions will at times call the size adjustment functions on its respective container objects in order to force them to conform with the newer dimensions of the container. We can do this because *all* components have these size adjustment functions, because we declared the interfaces to these functions inside the `Component` class.

And, likewise, if you wish to `Draw` a container, you must peform any special drawing work the container itself needs (for example, drawing the cell borders, or perhaps not doing anything), and then drawing the individual components that the container contains. This is why we do not provide a definition for a `Draw` function in `Container`. We have decided that `Container` itself is an incomplete blueprint, until a more specific container class provides some information about how to draw that specific type of container. However, we do know how to handle the drawing of the individual components of the container, so we put that info inside a protected helper function that we can call from later derived classes.

Finally, we have the `Window` class. This is simply a concrete container object. We know exactly how to draw a window – first you need to actually draw the window border and menu bar, and then you can draw the individual components. So, since we have now completed the information about how to draw this container, we have a complete blueprint and so we can allocate objects of type `Window`.

The `tempMain.C` file shows a small program that illustrates the use of some of these things. We are creating a window and setting its size and number of components, and then one by one we actually add those components. When we are done, we can click in various places of the window, and the click is "transferred" to the component that is in that spot in the window. When we move the window, the components inside come with it, so that when we click in a new spot in the moved window, we are still "transferring" the click to an internally-held component.

So, take a close look at the file syntax of the given code, and try and understand how the tempMain.C file generates the results that it does, and how easily we are able to transfer control from one component (the window) to another component (the button) by using functions common to both (since they are both of the `Component` type).

2.2 Your task

You are going to write two classes of your own. The first will be called `Easel`, and will be derived directy from `Component`. The second will be called `Panel` and will be derived from `Container`. (The class `Panel` exists in Java, but `Easel` is completely made up for this MP.)

The specification for `Panel`, the easier of the two, is as follows:

`Panel`

This class is derived from `Container`. A `Panel` in Java is a "window" that cannot be a stand-alone window. For example, when you write an applet and run it in a web page, that is a panel. It has space set aside to draw and write things in, as a window does, but the panel cannot stand-alone; it needs to be inside something else. We are going to tweak things slightly, in order to make our stripped-down version a bit more interesting, but that is the basic idea.

`Private Data Members`

There are two point objects that are data members of `Panel`. These point objects will conceptually hold the lower left and upper right corner points of whatever object – Netscape window, some type of panel-viewer, or whatever – holds this panel.

`Public Functions`

There are five public functions to write for `Panel`

- You first need a default constructor. The internal points in this case can be both initalized to (0, 0).
- You also want a second constructor, which will accept two `Point` objects as parameters and use them to initialize the internal points.
- Next, you want a function `ParentResized` which takes two point objects (we are assuming the parent calls this function off its `Panel` object) and sets the `Panel` points to these points. There is nothing to return.
- Since this is to be a concrete class, you will need to fill in the definition for the remaining two pure virtual functions of the `Component` class `HandleMouseClick` having been defined in the `Container` class). The `Draw()` function should print out the line:

```
This panel is held by an application window
  with the following corner points:
```

  followed by the points, in the format:

```
(1st x coord, 1st yCoord), (2nd x coord, 2nd y coord)
```

  Next, you want the line

```
This panel holds the following components:
```

  followed by a printing of the components held by the panel.

- The `Clone()` function should behave as the `Clone()` functions did in the two given concrete classes. Note from those two classes that you can indeed call the compiler-supplied copy constructor. (If that last sentence didn't make any sense to you, look carefully at the `Clone()` functions for the two given concrete classes and think carefully about what we said about copy constructors and when we do and don't have to write them and what happens when we don't.)

The specification for `Easel`, is as follows:

`Easel`

This class is derived directly from `Component`.

`Private Data Members`

There is one private data member for `Easel` that you must have, and that is an `Array` of `Point` objects. (Note that since it is an `Array` of *objects* and not `pointers`, you will not have any dynamic memory to worry about and thus will not need to write "the Big 3". If you find that a few additional data members would be helpful – some extra integers, perhaps – you can add those as well. That will be left up to you as a design decision. But the one you *have* to have is the `Array`.

`Public Functions`

There are four public functions to write for `Easel`.

- You need a default constructor. This constructor should set up your `Array` to be of size 20 initially, plus of course you should make the appropriate initializations for whatever other data members (if any) that you added yourself.

- Since this is to be a concrete class, you will need to fill in the definition for the three pure virtual functions of the `Component` class. The first one you want to handle is the `HandleMouseClick` class. If there is a "mouse click" on this object, you add a point to the easel at the place where the "mouse click" occured. So, for example, if the easel has corner points (0, 0) and (20, 30), and you have a "mouse click" at (10, 10), that will be on this easel, and so you should store a point internally that is at (10, 10).

  We will not go into the specifics of how to accomplish this; give it some thought (and remember you are storing an `Array` of `Points` and work out the details. One small note, though – as we have mentioned in class, if you should fill the current dimensions of your array, and you want to increase the size of the array to hold more data, you should at that time *double* the size of the array, rather than simply increasing the size by one or two.

- The `Draw()` function should print out the line:

  ```
  This easel contains the following points:
  ```

  followed by all the points you have added to the easel via mouse clicks, in the order that you added those points to the easel. If we could run some graphics, the picture we would draw is a blank white rectangle with black dots wherever the mouse had been clicked. That is the kind of thing you are trying to describe in text. The format of each point will be

  ```
  (x-coordinate, y-coordinate)
  ```

on its own line. Of course, "x-coordinate" and "y-coordinate" are to be replaced by the appropriate numbers.

- The `Clone()` function should behave as the `Clone()` functions did in the two given concrete classes. Note from those two classes that you can indeed call the compiler-supplied copy constructor. (If that last sentence didn't make any sense to you, look carefully at the `Clone()` functions for the two given concrete classes and think carefully about what we said about copy constructors and when we do and don't have to write them and what happens when we don't.)

That is all!

2.3 Testing the new classes

Later today, an extensive test class group will appear that will test things in interesting ways to teach you more about virtual functions. As long as the above classes are working as we have described, everything should be fine. Details will appear on the NG.

3 Handing in your code

To handin your MP3 code, use the command

```
handin cs225 mp3 easel.h easel.C panel.h panel.C}
```