ECE198KL: Introduction to Computer Engineering II **Spring 2013**
Program 12 **due: 10 p.m. on Monday 22 April**

Hello, World! (for WALY)

This week, you will create a screen of your own, applying the basics of C++ class design in the context of the WALY code and beginning to explore the use of event-based programming strategies. The code distributed to you simply launches a screen with a button marked "Start." You will add a second screen, also including a button, and enable the user to drag a small image around the screen with the mouse.

As always, routine details such as how to obtain the code and how to hand in your program can be found in the specification for Program 1. Instructions on targeting and making the Linux and Android versions can be found in the specification for Program 4.

The Pieces

We are using the same package to allow cross-compilation on different platforms as well as support for the graphical user interface. This week, you will develop your own class header and source file, but you may want to take a look at the header and source for the screen already provided (as a model for your own). You will also need to edit part of the dependence information for the program, as stored in the file `Make.inc`. And you will make some modifications to `prog12main.cpp`. Let's discuss each of these in a little more detail:

`jni/BurstScreen.h`	Header file for the burst screen class. You should not change this file, but may find it useful as a prototype for your own header file.
`jni/BurstScreen.cpp`	Source file for the burst screen class. You should not change this file, but may find it useful as a prototype for your own source file.
`jni/Make.inc`	A file that specifies all of the headers and sources to be built into the program. You must add the names of your new files into the lists in this file (see Step 1).
`jni/prog12main.cpp`	A file that contains the analogue of `main` for a WALY program, `WALY_main`. You will need to create an instance of your class and make a couple of additions to this file (see Step 3).

Other files are mentioned as necessary. Feel free to poke through files. Documentation in the WALY code is minimal at this point.

Testing is left to you. In programming studio, we will work on the first three steps. You can go further, but those parts of the code can be done cooperatively. Be sure to credit any collaborators explicitly in your program. **Steps 4 and later should be your own individual work.**

Step 1: Creating a Class

Your assignment this week is to create a second screen with specific behavior. The screen will implemented by a new class that you add to the program. Your class will be similar to the BurstScreen class, and you should read and understand both the header file and the source file for BurstScreen (`BurstScreen.h` and `BurstScreen.cpp`, respectively) before you create your own.

Once you are ready to begin, create a header and a source file for your class. You choose the name, but use the same style as in the rest of the code: class names use initial capitals, constant values use full capitals, and variables use initial lower case; extra words in names also begin with capitals for both class names and variables.

You can decide whether to write your files completely from scratch or to begin with a copy of the BurstScreen class files. If you choose to start with a copy, be sure that you make all necessary changes—most will lead to compilation errors, but some may not. If you do not change the name of the constant (`BURST_SCREEN_H`) used at the beginning and the end of the file to ensure that the content of your header file is only included once, for example, your class definition is likely to be skipped entirely.

Your class should be derived from the WALY Frame class. In your header file, you should include the WALY header file, `WALY/WALY.h`, then tell the compiler that your header uses the WALY namespace with the directive, "`using namespace WALY;`" (no quotes). After that line, the compiler checks for names that are part of WALY before complaining that a name doesn't exist. For example, you can then define your class as follows:

```
class WhateverYouWantToCallYourClass :  public Frame {
};
```

Note that your header and source file names should match your class name (a convention that makes it easy to find the code associated with a class, and is required by some languages, although not by C++). In other words, if your class is "OneClass", your files should be `OneClass.h` and `OneClass.cpp`.

The `Frame` class is part of the WALY namespace. If the compiler has not seen the `using` directive earlier in the file, the class definition must instead give the full name for the base class: `WALY::Frame`.

Create your source file. Include both the WALY header (as in your header) and your own header file. Also add the directive stating that you want to use the WALY namespace, as you did in your header file. See the beginning of `BurstScreen.cpp` for an example.

If you are writing your own source file from scratch, you do not need to have any variable definitions or code until you have written them into your class definition (in your header file). If you have started from a copy of `BurstScreen.cpp`, you will need to change all instances of the class name `BurstScreen` to your class' name.

Before you can compile, you must add your header and source file to the `Make.inc` file. The first few lines are used for building for Android, and you must add your source file name to the list after LOCAL_SRC_FILES. The names are separated by spaces. Order does not matter, but do NOT try to use multiple lines (look back at Program 11 if you really want to know how to format this file more elegantly with long lists of files).

The second portion of `Make.inc` is used for all other platforms. Add your header file to the list after `HEADERS`, and add the object file produced by your source file (replace `.cpp` with `.o`) to the list after `OBJ_FILES`.

After you have created both your header and source files, and have added them both to `Make.inc`, you should be able to compile. Your class is not used at all yet, but your should be able to build for any platform. Try building the program before moving on.

Step 2: Creating Your Screen

Since you derived your class from WALY's Frame class, you can have it drawn to the screen.

Add a constructor to your header file. The constructor must be `public` so that other classes can create instances of your class (objects). The constructor should accept at least two arguments: a pointer to the parent Frame, and a pointer to a function that can be called to switch back to showing the burst screen. The constructor must have the same name as your class, and has no return type.

You should also add a private field in which to store the pointer to the function passed to you (the function that switches back to the burst screen). The field should be private: only your class' code should access it.

Finally, add functions for activating and deactivating an instance of your class. These should be public and should neither take nor return anything (as member functions, they implicitly receive a pointer to an instance of your class). You can call them anything you want, but you will have to match the names when you use them.

So, for example, you might add the following to your class definition:

```
private:
    void (*switchAction) (void);
public:
    WhateverYouWantToCallYourClass (Frame* parent, void (*switchFunc) (void));
    void activate (void);
    void deactivate (void);
```

Next, you need to write the constructor and two functions in your source file. Functions in the source file must include the name of the class as a prefix. Since the constructor also uses the name of the class, the name will appear twice in the constructor definition. Following our example, you should add something like the following:

```
WhateverYouWantToCallYourClass::WhateverYouWantToCallYourClass
        (Frame* parent, void (*switchFunc) (void)) :
        Frame (par), switchAction (switchFunc)
{
    // constructor code goes here
}
```

The list of comma-separated elements between the end of the constructor signature and the start of the code (the open brace) is an initializer list. Any initializers must be given in the order of declaration in the class: base class first, then fields in order of appearance. Fields can be left out, in which case they are initialized using default methods. The constructor code is executed **after** the initializers. For now, you do not need any code in the constructor (but the other parts need to be defined).

The functions are simpler. Other than the class name prefix, they appear exactly like functions in C. Unlike C functions, however, member functions such as these can use functions from your class and fields from the implicit class instance (called `this`, if you want to use the pointer explicitly). Add them to your source file as shown here:

```
void
WhateverYouWantToCallYourClass::activate (void)
{
    setActive (true);
    setVisible (true);
}

void
WhateverYouWantToCallYourClass::deactivate (void)
{
    setVisible (false);
    setActive (false);
}
```

These functions are defined in `WALY/WALYFrame.h`. For both functions, the compiler implicitly casts the pointer to your class instance (the `this` pointer) into a `Frame*`, then invokes the function on the resulting pointer, passing `true` or `false` as the second argument (since `setVisible` and `setActive` are member functions, the first argument, the `Frame*`, is implicit).

Making a frame visible makes it and all of its children visible. Each child frame also has a visible bit. An invisible frame makes all children (and grandchildren, and so forth) invisible, while a visible frame allows but not require children to be visible.

The active bits work the same way: a frame and its descendants only receive events such from the mouse, keyboard, frame updates, and so forth if it is active.

Make sure that these new functions compile before moving on to the next step.

Step 3: Switching to Your Screen

Now we can add code to the `prog12main.cpp` to enable switching to your screen.

You will need to include your class' header file, create a file-scope variable that points to an instance of your class and initialize it to NULL for safety.

In `startGame`, replace the call `burstScreen->activate ()` call with a call to activate your screen instead. Similarly, in `returnToBurst`, add a call to deactivate the instance of your class. (Be sure to either delete or comment out the call to activate the burst screen in this function, or both screens will be active at onces.)

Now, in `WALY_main`, create an instance of your class in the same way that the burst screen is created, but pass `returnToBurst` as the `switchFunc` argument instead of `startGame`. The burst screen calls `startGame` when you press the "Start" button—now when that happens, your class instance will be activated. Be sure to delete the instance of your class, too (just before or after deleting the burst screen instance).

Compile and run the program, then press the button. If you did everything correctly, the screen should appear to freeze, and the button will stay depressed. Although you screen is visible and active, you didn't actually tell the library to draw anything. Frames by default have no content. The burst screen is no longer active, so the display window retains the old content.

Step 4: Adding a Background Color

Let's start by adding a background color. Add the lines below to the constructor function for your class:

```
Rect r;
r.x = r.y = 0;
r.w = 500;
r.h = 800;
setScrollRect (r);

useSolidBackground (SOME_NUMBER);

deactivate ();
```

The scroll rectangle is the portion of the frame's drawing plane that is shown on the screen. If the scroll rectangle is empty (the default), only children are drawn. If the scroll rectangle is set, only children within the rectangle are drawn. However, a scroll rectangle is needed if we want an opaque or solid background color for the frame.

We have chosen to use a 500×800 screen resolution with this program, so your new class instance will fill the whole screen (on Android, the image is scaled up).

You must replace `SOME_NUMBER` above with a 24-bit RGB color value. In other words, an integer with an 8-bit red level, an 8-bit blue level, and an 8-bit green level. For example, the color red is 0xFF0000, green is 0x00FF00, and purple is 0xFF00FF. You can choose any color that suits your fancy for your window.

Your class instances should initially start deactivated (invisible), so the last line ensures that such is the case.

If you compile and run the program, you should now see your screen after pressing the "Start" button.

Step 5: Adding a Label

Next, you will add some words to your screen.

You do not need to manipulate the words later, so you can simply create an instance of the `Label` class, which is derived from the `Frame` class, and make it a child of your screen. When your screen is deleted, all of its children are deleted as well, so you can forget about the label once you have set it up.

In your constructor, add something like the code below just before the call to `deactivate` at the end of your constructor:

```
Label* lbl = new Label (this, 250, 200, "A wise saying...", 0xFFFFFF);
lbl->setAlign (ALIGN_C);
```

The first argument, `this`, refers to the instance of your class being constructed—the new label becomes a child of your class instance (both are derived from `Frame`, which maintains a tree structure with parent-child relationships).

The second and third arguments specify the X and Y coordinates of the alignment point, respectively. You may put your text anywhere on your screen, so long as it is visible.

The fourth argument is a string representing the text to place in the label.

The last argument gives the `24-bit` RGB color number to use for the text. Again, any color is fine, but make sure that we can read it.

Optionally, you can add a sixth argument specifying a background color.

The second function sets the alignment of the label so that the center of the rectangle containing the text (the size depends on the font selected; in the code shown, your label uses the default font) is over the alignment point specified in the parent window. The default alignment is to put the upper left corner of the rectangle over the alignment point.

Compile and run your program, then click "Start" to see your new label.

Step 6: Adding a Button

Now we're ready to add a button that returns us to the burst screen. First, we'll need to declare a function for that purpose in our class header file, as shown below. Add this declaration inside of your class definition, and make sure that it is marked as `private` (no other class should call it).

```
private:
    static void pressButton (Button* b);
```

As you will see shortly, buttons are another class provided by the WALY library. For each button you create, you can provide a callback function that takes a pointer to the button pressed as an argument. We will use the `pressButton` function that you just wrote for this purpose.

Remember that the `static` qualifier in front of the function means that it is not a member function. In other words, it requires no implicit pointer to an instance of your class. The code in the button class has no way to obtain such a pointer, so you must use `static` functions as callbacks for button presses.

You, however, can get a pointer to your class instance from the button pointer. In a moment, you will make your button a direct child of your class instance, allowing you to obtain the parent of the button, then turn it into a pointer to your class, again as shown in the code below.

Once you have a pointer to your class instance (the variable `wywtcyc`), you can invoke the `switchAction` function to return to the burst screen.

Now let's write the function in your source file, as shown below. Notice that you do not repeat the qualifier `static` in the function definition: the compiler knows that this function is not a member function, since it has already seen the class definition.

```
void
WhateverYouWantToCallYourClass::pressButton (Button* b)
{
    WhateverYouWantToCallYourClass* wywtcyc =
        (WhateverYouWantToCallYourClass*)b->getParent ();
    wywtcyc->switchAction ();
}
```

Finally, create the button. Go to your class' constructor and add the following code, again just before the call to `deactivate` at the end of the constructor:

```
    Button* b = new Button (this, 250, 500, "My Button", 0xFFFFFF, 0x008000);
    b->setAlign (ALIGN_C);
    b->setActionFunc (pressButton);
```

The arguments needed to create a button are the same as those needed for a label, except that a background color must be supplied. Alignment is a frame function, so it can be used with buttons as well.

The `setActionFunc` call provides a pointer to the `pressButton` function to the button instance. If you click on the button and release the mouse while still inside the button, whatever function you have provided is called.

New frames appear on top of old ones, so if your button and label overlap, part of your label will be hidden. Place your button so as to avoid this problem.

Compile and run again. Now you can go back and forth between the two screens!

Step 7: Preparing for Dragging

The tasks so far have been primarily tutorial in nature. Soon you will have to design a small state machine on your own and implement it in your class. First, however, you must set up a few more functions that will help you implement the state machine, and create a "rectangle" image that you can move around.

Begin by adding a field to your class. Inside the class definition, declare a field as "`Frame* box;`" (no quotes). Fields should always be `private`.

In your constructor, add the following code just before `deactivate`:

```
    box = new Frame (this, 250, 700);
    box->setAlign (ALIGN_C);
    Rect boxRect;
    boxRect.x = boxRect.y = 0;
    boxRect.w = 80;
    boxRect.h = 50;
    box->setScrollRect (boxRect);
    box->useSolidBackground (0x808080);
```

The code creates a child of your class instance and draws it as a 80×50-pixel grey rectangle with its center aligned at $(x, y) = (250, 700)$.

Feel free to change the size, color, alignment, and so forth to suit your taste.

You may want to compile and run now, just to make sure that the rectangle does in fact appear on the screen.

Now go back into your header file and add the following three function declarations (all `private`). These are for handling mouse button and motion events: when the user presses or releases a mouse button or moves the mouse, one of these functions is called, and you can make changes in response to the action by the user.

```
    private:
        static void mouseButtonDown (Frame* f, const Event* e);
        static void mouseButtonUp (Frame* f, const Event* e);
        static void mouseMotion (Frame* f, const Event* e);
```

In your source file, add the three functions. The motion function should appear as follows:

```
void
WhateverYouWantToCallYourClass::mouseMotion (Frame* f, const Event* e)
{
    WhateverYouWantToCallYourClass* wywtcyc =
        (WhateverYouWantToCallYourClass*)f;
    wywtcyc->box->setX (e->motion.x);
    wywtcyc->box->setY (e->motion.y);
}
```

The other two functions are similar, but for now just leave the code empty.

The frame `f` passed to the function is your class instance, so you can use a cast to obtain a pointer to the class instance in a variable, as shown above.

You can then access fields of your function, such as the `box`. The code shown above sets the X and Y coordinates of the box to the coordinates of the mouse, which are provided as part of the event `e`.

In the button functions, these coordinates are available in `e->button.x` and `e->button.y`.

Before your functions are called, you must register them with the WALY library. Add the following code at the end of your constructor (just before `deactivate`):

```
    setCallbackFunc (MOUSE_DOWN, mouseButtonDown);
    setCallbackFunc (MOUSE_UP, mouseButtonUp);
    setCallbackFunc (MOUSE_MOTION, mouseMotion);
```

Compile and run the program. The box should now follow the mouse cursor around the screen.

Step 8: Dragging the Box

Now for your part!

Add fields to your class and change the constructor, mouse button, mouse motion, and activate functions as necessary to create the following behavior.

The box should be at a fixed location when your screen appears (remember that activate is called to show your screen).

When the user presses the mouse button, the box should move to the mouse and follow the mouse around the screen so long as the mouse button is held down.

When the user releases the mouse button, the box should stay at its last position.

Challenges for Program 12

Here are some challenges for this week.

- **(3 points)** Require that the user click inside of the rectangle to begin dragging it.
- **(2 points)** Specialize the font used for your label. Note that there's only one font style available on Nexus, but you can change the size.
- **(2 points)** Go back to the burst screen if the user presses the Escape key. See the code in the burst screen for testing for the key, and use the function passed to your constructor to switch back (instead of minimizing, as does the burst screen on Escape).
- **(2 points)** Let the user drag an image around the screen instead of a square. The burst screen uses an image.
- **(3 points)** Play a sound when a user releases a mouse button on your screen. The burst screen plays a sound–I'm not sure whether you can plug earphones into the EWS machines, but you can hear the sounds on your Nexus.
- **(8 points)** Support multi-touch on the Nexus. The mouse number (from 0 to `(SDL_MAXMOUSE - 1)`) is passed in the `Event`'s `button.which` field for mouse button up/down events and in the `Event`'s `motion.which` field for mouse motion. You will need to add a separate rectangle for each mouse. Make sure that the different mice are distinguishable and predictable (different colored rectangles or different images, for example). You need only support three touches for full credit (Nexus supports at least 10).

Specifics

- Your code must be written in C++ and and must be contained in a header file and source file using a new name selected by you. These names must be included correctly in `Make.inc` to enable compilation. Do not change any other files.
- You must implement all steps. Steps 1, 2, and 3 may be performed together with other students (be sure to credit them), but Steps 4 and onward must be done on your own.
- Your code must be well-commented. You may use either C-style (`/* can span multiple lines */`) or C++-style (`// comment to end of line`) comments, as you prefer. Follow the commenting style of the code examples provided in class and in the textbook.

Grading Rubric

Functionality (60%)

15% - Pressing "Start" successfully launches your screen class with a background color and a label.
20% - A button on your screen allows the user to return to the burst screen.
25% - The user is able to pick up and drag a rectangle around the screen using the mouse. (The rectangle may appear when the user presses down the mouse button, or may already be visible and require that the user click inside of it to begin dragging it, depending on whether you have implemented any challenges.)

Style (20%)

10% - compilation generates no warnings (note: any warning means 0 points here)
5% - does not use global variables (file-scoped exist already)
5% - indentation and variable names are appropriate and reasonably meaningful (index variables can be single-letter)

Comments, clarity, and write-up (20%)

5% - introductory paragraph explaining what you did (even if it's just the required work)
15% - code is clear and well-commented

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.