Chapter 3: GUI-Kit: Hands-On Training

Prerequisites

This chapter assumes that you are familiar with the C programming language. You should be familiar with using MS-Windows; however, we do not require you to have experience with the Windows API. You should also know how to use your compiler and be familiar with makefiles.

Overview of GUI-based Systems

Windows

Most GUI systems are based on model consisting of a desktop (the display) containing rectangular areas called *windows*. A window is a rectangular area on the display that may be visible or hidden. The system may *stack* the windows so an upper-stack window may entirely or partially obscure a lower-stack window. We sometimes refer to the order of window stacking as the ‘‘Z-order’’.

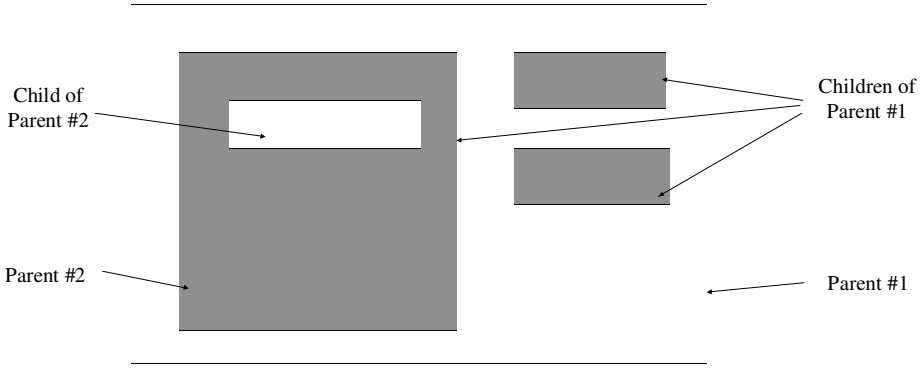

Figure 3-1: Parent/Child Window Relationships

The system creates windows in a hierarchy (see Figure 3-1). The parent of all windows is the display. Child windows always appear in front of their parent window.

A window provides a place for you to present information to the user. You can present information by writing text in the window (using various fonts), drawing geometric figures, rendering bit-mapped images, etc.

Events

You interact with a window using a mouse or keyboard. Mouse interactions are usually associated with the *active* window. The system always sends keyboard input to the window which has the *keyboard focus*. Only one window in the system can have the keyboard focus. The system relays all keyboard and mouse input to the application as *events*. Most events are associated with a particular window.

Input events are not the only types of events your application can receive. You can also receive events such as window size and position changes and exposure events which inform you that a window needs repainting. Some events, such as timer events, are not associated with a particular window.

In a GUI-based system, the user can cause an event to occur on any window. For example, the user may type some text in a field and then click on a button -- the user controls the application. This is the opposite of text-based applications in the past where the application controlled the sequence of events. You must structure your application so that it can respond to these events. This is usually done via an event loop. After creating all of your windows, your application sits back and waits for an event. When you receive an event, your application must determine the event type and handle it appropriately.

GUI-Kit's GUI Model

Although GUI-Kit is based on the GUI model described above, GUI-Kit handles many complexities of GUI-based programming for you. As you'll see in the next section, GUI-Kit defines abstract components from windows. Since the behavior of a particular component is well defined, GUI-Kit can manage all of the standard interactions of the component. Of course, you can override the standard interactions if you wish.

GUI-Kit's Object-Oriented Programming Model

In the past, developers created software from unrelated pieces. Because these pieces didn't understand how to interact with each other, developers generally wrote new programs from scratch. Object-oriented programming organizes the various pieces of software so that they can interact with each other. It also allows one piece of software to reuse the functionality of another piece of software and add new functionality in the process.

Classes

A class defines the behavior and attributes of each piece of software. Scientists often classify things -- for example, you are a member of the class of Humans. In turn, Humans belong to the more general class of Animals, and Animals belong to the class of Living Things. Since Animals have attributes that all Humans share, we call the class of Humans a *subclass* of Animals. We refer to the class of Animals as the *super-class*, or parent class, of Humans. We call the relationship of a specific class to one or more general classes a *class hierarchy*. A class hierarchy makes it easy to *inherit* the *attributes* of an existing class. Let's define some attributes of the Animal class:

- Animals breathe.
- Animals have a heart and blood vessels.
- Animals have eyes, ears, noses, and mouths.

If we now define attributes for the Human class, we can simply say that it is a subclass of Animals and has the following attributes:

- Humans have two arms and two legs.
- Humans have five fingers on each hand and foot.
- Humans walk upright on their two legs.

Note that we didn't have to repeat the fact that Humans breathe because Humans are a subclass of Animals. As you might have expected, GUI-Kit also has a class hierarchy; Figure 3-2 illustrates it.

Objects

In the example above, we defined the class hierarchy for Humans; many people belong to the Human class. In object-oriented programming, we call a member of a class an *instance* or *object*. Thus, a class defines the behavior and attributes of an object. An object is something which your application can manipulate.

Attribute Values

Although classes define attributes, objects specify the *value* for the attribute. For example, in GUI-Kit you can specify the label which appears on a button. Since a button can have a label, ''label'' is an attribute of the button class. The actual text for the label is the attribute's value. Of course, you wouldn't want all buttons to have the same label, so we store the attribute's value in the object, not the class.

Instance Hierarchy

We can create objects so that they are children of other objects. This is often used to define the parent/child window relationship on window-based objects. When we create objects in this fashion, we call the result an *instance hierarchy*. An instance hierarchy should not be confused with a class hierarchy; they are two very different things.

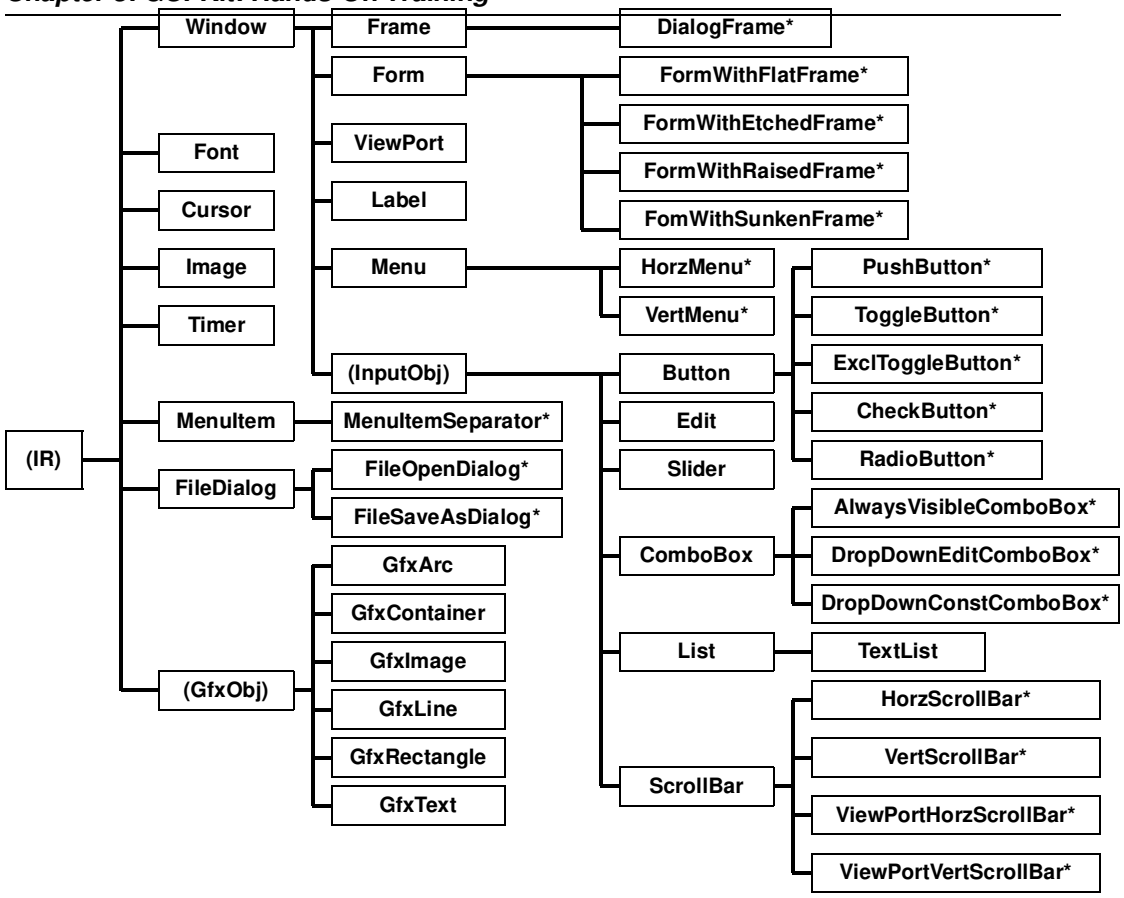

Figure 3-2: GUI-Kit Class Hierarchy

Notification of Actions

When a specific action occurs on an object (e.g., you click a button), the object can notify your application of the action. For example, if you click on the "Quit" button, the object knows how to handle the mouse click, but it does not know what to do in response to the click. Thus, the object tells your application: "someone clicked the button -- do something." Your application might respond to the button click by closing the window.

Cooperating Classes

Certain GUI-Kit classes cooperate with each other. For example, the **Window** class is aware of the existence of the **Cursor** class. Although you create a cursor object separately from a window, you can attach a cursor to a window so that the window has a unique cursor. You'll see later the **GfxObj** (graphics object) class is a highly-cooperative class; it gives GUI-Kit a great deal of flexibility.

Essential Functions

You must understand few key functions to write applications with GUI-Kit; they are:

Function	*Description*
`gkInit`	Initializes GUI-Kit and your application.
`gkCreate`	Creates an object and sets one or more attribute values.
`gkSet`	Sets one or more attribute values on a previously created object.
`gkGet`	Gets the value of an attribute.
`gkFind`	Finds a previously created object.
`gkDestroy`	Destroys an object.
`gkDispatch`	Handles events and dispatches them to the appropriate object.

Note: In this guide, we'll refer to an attribute and value pair as an *AV* (Attribute-Value).

Amazingly, to write a GUI-Kit application, you only need to know how to use these six functions. However, many other functions exist to make writing a GUI-Kit application even easier. We refer to the `gkCreate`, `gkSet`, `gkGet`, `gkFind`, and `gkDestroy` functions as *methods*. We use this term because each class carries out its own method to perform each function.

Let's take a closer look at each function.

gkInit

You will only call `gkInit` once, usually at the beginning of your `main()` function. The prototype for `gkInit` is:

```
int gkInit(..., NULL);
```

As indicated by the prototype, `gkInit` takes a variable number of arguments which terminate with a `NULL`. The variable arguments are AVs which are specific to `gkInit`. Here's a typical example of how you can use `gkInit`:

```
main(int argc, char **argv)
{
    gkInit(GKInit_Args, &argc, argv,
           GKInit_StripArgs,
           NULL);
```

. . .

GKInit_Args is an attribute which allows gkInit to process the arguments received by main. The GKInit_StripArgs attribute tells gkInit to strip standard GUI-Kit command-line arguments from argc and argv. You might have noticed that we pass argc as a pointer, this is done so that gkInit can modify the argument count when it strips the command-line arguments. The standard GUI-Kit command-line arguments are defined later -- see *Command-Line Arguments* on page 20-1. You should note that some attributes take no values, such as GKInit_StripArgs; these attributes simply perform an action or suggest the presence of a condition. Other attributes can take more than one parameter, as with GKInit_Args; collectively we call these parameters the attribute's value.

Note: when we refer to an attribute by name in this manual, we may refer to it as the C identifier GK*class-name_attribute-name*, or as the resource identifier ***class-name.attribute-name***. We'll use the C identifier (e.g., GKButton_Label) when referring to a code example; we'll use the resource identifier (e.g., **Button.Label**) when referring to the attribute generally. You will learn more about resources later in this chapter.

gkInit performs other initializations besides processing command-line arguments -- it also registers GUI-Kit classes, registers names for standard GUI-Kit functions, and loads resource files. We will cover these topics later in this chapter.

You must always terminate a list of AVs, like the parameters of gkInit, with a NULL. The NULL specifies the end of the AV list.

gkCreate

As stated earlier, gkCreate creates an object and sets its attribute values. The prototype for gkCreate is:

```
GKObject gkCreate(GKObject parent_obj,
                  GKClass *classp, ..., NULL);
```

The parent_obj parameter ties the new object to an instance hierarchy; parent_obj is the parent object in the hierarchy. Not all objects can belong to an instance hierarchy, so sometimes this parameter is NULL. The parent object may also be NULL if you are establishing the top-level object in the hierarchy. The classp parameter is the class of the new object. A list of AVs follows classp. We terminate the AV list with NULL.

We should note here that some attributes are only valid on gkCreate, gkSet, or gkGet, but many attributes are valid on all three methods. Also, gkSet attributes are always valid on gkCreate. See *Class and Attribute Reference* on page 23-1 for each attribute's methods.

gkCreate returns an object identifier which refers to the object proper. If gkCreate fails, it returns NULL. Let's look at an example for gkCreate:

```
GKObject    frame;

frame=gkCreate(NULL, GKFrameClass,
               GKFrame_Title, "gkCreate Example",
               GKWindow_Width, 100,
               GKWindow_Height, 100,
               NULL);
```

In this example, we create a **Frame** object; a **Frame** is a top-level window which is usually used as the main window of your application. Notice that we set the **Frame** attribute GKFrame_Title and two **Window** attributes to specify the width and the height. We can set **Window** attributes on the frame because **Frame** is a subclass of **Window**.

gkSet

gkSet is similar to gkCreate except that you're working with an existing object. The prototype for gkSet is:

```
int gkSet(GKObject obj, ..., NULL);
```

Again, the variable arguments refer to a list of AVs terminated with a NULL. However, this time we pass obj to specify the object that we want to set. We could modify the gkCreate example above to break apart the creation of the object from the setting of the attributes:

```
GKObject    frame;

frame=gkCreate(NULL, GKFrameClass,
               NULL);

gkSet(frame,
      GKFrame_Title, "gkSet Example",
      GKWindow_Width, 100,
      GKWindow_Height, 100,
      NULL);
```

In practice, you would probably check the return code from gkSet to see if the call was successful. gkSet returns GKMETHOD_OK when it succeeds.

gkGet `gkGet` gets the value of a single attribute. The prototype for `gkGet` is:

```
GKVal gkGet(GKObject obj, GKAttr *attr, GKVal arg_val);
```

The `obj` parameter specifies a previously created object. The `attr` parameter specifies the attribute whose value `gkGet` retrieves. Some attributes use `arg_val` to allow additional information for the retrieval. If the attribute does not require `arg_val`, you should set it to `NULL`. Using the previous `gkSet` example, you could retrieve the width of the frame as follows:

```
GKObject    frame;
int         width;

    . . . Create and Set . . .

width=(int)gkGet(frame, GKWindow_Width, NULL);
```

Note that `gkGet` returns a `GKVal` type; `GKVal` is a generic type which can contain any value. Therefore we have to cast the return of `gkGet` to the attribute's type (an `int`). If `gkGet` fails, it returns a value of zero. Depending on the attribute, distinguishing an error of zero from a valid value of zero may be difficult. In these cases, you should use the `gkGetWithStatus` function.

gkFind `gkFind` allows you to find a previously created object. Only two classes, **IR** and **Font**, support the use of `gkFind`. The function prototype is:

```
GKObject gkFind(GKObject parent_obj, GKClass *classp, ..., NULL);
```

The `parent_obj` parameter specifies the parent object in which the desired object must have; it may be `NULL` if you do not want to search on the parent object. The `classp` parameter is the class of the object to be found; note that most classes which allow `gkFind` do not search on this parameter. The arguments following `classp` are pairs of attributes and values; they must be Find attributes. The argument list must terminate with `NULL`.

If `gkFind` succeeds, it returns the desired object; otherwise, it returns `NULL`. An alternate function, `gkFindOrCreate`, attempts to find an object using `gkFind`, if it fails, it creates a new object. `gkFindOrCreate` takes the same parameters as `gkFind` and it also accepts any Create attribute. This function is especially useful with the **Font** class.

gkDestroy `gkDestroy` simply destroys an object. The prototype is:

```
int gkDestroy(GKObject obj);
```

gkDestroy returns GKMETHOD_OK if it successfully destroyed the object. If you have attached obj to another object (such as a cursor attached to a window), gkDestroy will not destroy the object, instead it returns GKMETHOD_OBJINUSE to indicate that the object is in use.

gkDispatch

gkDispatch is the main loop for your application. After you have created all of you objects, you simply call gkDispatch. gkDispatch takes care of dispatching events to the appropriate objects; in turn, the objects will notify your application of relevant events. We'll talk more about gkDispatch later.

A Complete GUI-Kit Program

You'll find our example program for this section in the \gui-kit\samples\prog1 directory (\gui-kit is where you installed GUI-Kit on your hard disk). Copy all of the files from this directory to a working directory. To compile the example, set the VENDOR environment variable to your compiler's vendor, which should be:

- MSC for Microsoft C/C++ 8.00 or greater (included with the Win32 SDK), or Microsoft Visual C++ 32-bit edition.
- BOR for Borland C++ 4.0 or greater.
- WAT for Watcom C/C++ 32 9.5 or greater.
- SC for Symantec C++ Professional 6.1 or greater.

You may also need to set the GKBASE_DIR environment variable, see *Installing GUI-Kit* on page 2-1 for more information. If you're using Microsoft C/C++, you would compile the example by typing the following in a Windows DOS session:

```
set VENDOR=MSC
dmake APP=prog1
```

After the compilation is complete, run prog1.exe from the File Manager or from the File menu on the Program Manager. You should see one button labeled "Press Me" and another labeled "Quit". If you click on the "Press Me" button, its label changes to "Thank You!". Notice that the program automatically resizes the button and moves the "Quit" button to the right. If you click on the "Quit" button, the program will exit.

Let's take an in-depth look at prog1.c (we have numbered the lines later explanation):

```
 1  #define GKInclAll
 2  #include "gk.h"
 3
 4  /* Prototypes for internal notification procedures  */
 5  static void LabelChange(GKObject obj);
 6  static void Quit(GKObject obj);
 7
 8  static GKObject frame;
 9
10  /*-------------------------------------------------------------*/
11  int main(int argc, char **argv)
12  {
13      GKObject    form;
14
15      gkInit(GKInit_Args, &argc, argv,
16             GKInit_StripArgs,
17             NULL);
18
19      frame=gkCreate(NULL, GKFrameClass,
20                     GKFrame_Title, "GUI-Kit 101",
21                     NULL);
22
23      form=gkCreate(frame, GKFormClass,
24                    GKForm_AutoLeftToRight,
25                    NULL);
26
27      gkCreate(form, GKButtonClass,
28               GKButton_Label, "Press Me",
29               GKButton_NotifyProc, LabelChange,
30               NULL);
31
32      gkCreate(form, GKButtonClass,
33               GKButton_Label, "Quit",
34               GKButton_NotifyProc, Quit,
35               NULL);
36
37      gkDispatch();
38      return(0);
39  }
40
41  /*-------------------------------------------------------------*/
42  /* NotifyProc for "Press Me" button.
43   */
44  static void LabelChange(GKObject obj)
```

```
45  {
46      gkSet(obj,
47            GKButton_Label, "Thank You!",
48            NULL);
49  }
50
51  /*-------------------------------------------------------*/
52  /* NotifyProc for "Quit" button - destroy the frame -- this
53   * will cause gkDispatch to exit.
54   */
55  static void Quit(GKObject obj)
56  {
57      gkDestroy(frame);
58  }
```

Lines 1 - 2: The definition of `GKInclAll` causes gk.h to include all auxiliary includes files. You will usually define `GKInclAll` unless you are writing a GUI-Kit class.

Lines 4 - 6: Prototypes for notification procedures (also called "NotifyProcs") defined by `prog1`. Certain GUI-Kit objects can notify your application when specific events occur. In this example, we have defined two NotifyProcs; these NotifyProcs are associated with the buttons. The `LabelChange` NotifyProc causes the "Press Me" label to change to "Thank You!" when the first button is pressed. The `Quit` NotifyProc causes the program to exit when the "Quit" button is pressed.

Line 8: Declares the variable `frame` which is a "handle" to a GUI-Kit **Frame** object. `frame` is global to this module. You'll see later that there are methods of avoiding global variables for larger applications -- eliminating global variables will allow you to use notification procedures for several objects.

Lines 10 - 12: The first routine called in any GUI-Kit application is `main`, just as you would write in any standard C program. Note that although this is a Windows program, you don't have a `WinMain` procedure. Using a `WinMain` procedure wouldn't be portable across platforms.

Line 13: Declares the variable `form` that is a "handle" to a GUI-Kit **Form** object.

Lines 15 - 17: `gkInit` initializes GUI-Kit and your application. You must always call `gkInit` prior to creating any objects. You can only call a handful of GUI-Kit routines prior to `gkInit`; two examples are `gkRegisterNamedProcedure` and `gkRegisterNamedClass`. You'll note that we pass two AVs to `gkInit`:

- `GKInit_Args` allows GUI-Kit to see `main`'s arguments. Note that we pass the address of `argc` as the first attribute value. This allows GUI-Kit to modify the value of `argc` if it strips any arguments from `argv`.
- `GKInit_StripArgs` tells `gkInit` to strip-out any standard GUI-Kit command-line arguments from `argv` and to modify `argc` accordingly. You would normally use this attribute if your application has command-line arguments -- this way you don't have to process the GUI-Kit command-line arguments.

As with all functions that accept an AV list, we terminate the list with `NULL`.

Lines 19 - 21: Creates a **Frame** object and assigns the object handle to `frame`. A **Frame** is a top-level window which allows the user to control size, placement, and other aspects of the application. By default, GUI-Kit hides a frame until we call the `gkDispatch` routine, after which it displays the frame. The first parameter to `gkCreate` is the parent object. **Window**-based objects accept a parent object which specifies the parent window. Since a **Frame** is a top-level window, it has no parent window; thus, we use `NULL` for the parent object.

The second parameter to `gkCreate` is the class of the object we're creating. Since we are creating a **Frame**, we have specified `GKFrameClass`. When you specify a GUI-Kit class name in C, we always prefix the class name with "`GK`" and suffix it with "`Class`".

The rest of the parameters to `gkCreate` comprise the attribute-value list. This list tells `gkCreate` how it should create the object. Again, we always terminate the attribute-value list with NULL.

The only attribute specified on the creation of the **Frame** sets the frame's title to "GUI-Kit 101". All attributes which allow string values duplicate the string internally, unless explicitly stated otherwise. This allows you format a locally allocated string, pass it as an attribute's value, and not have to worry about the allocation of the string. GUI-Kit manages a copy of the string for you.

Of course, a thorough GUI-Kit application would check the return value. A NULL tells you an error has occurred.

Lines 23 - 25: Creates a **Form** object and assigns the object handle to `form`. A **Form** object manages the size and position of its children's windows (which will be the buttons created in the following steps). A form relieves you of the burden of having to specify the position of each child window. To create complex layouts, a form may contain another form as a child window. The creation of the form is similar to the creation of the frame, except that it takes `frame` as the parent object. The `GKForm_AutoLeftToRight` attribute specifies that child objects should be automatically laid-out from left to right unless specified otherwise.

Lines 27 - 30: Creates a **Button** object as a child window of the **Form** object. We have assigned a label of `"Press Me"` to the button. GUI-Kit automatically adjusts the size of the button to fit the label. Also, the procedure `LabelChange` is set as the NotifyProc for the button. This procedure will change the button's label when the button is pressed.

Lines 32 - 35: Creates another **Button** object similar to the previous one. GUI-Kit will call the `Quit` procedure when this button is pressed.

Line 37: We call `gkDispatch` to handle execution of the remainder of the application. `gkDispatch` will display the previously created **Frame** object; in turn, this will cause the frame to display all its children's windows. The main purpose of `gkDispatch` is to dispatch events to the appropriate objects. `gkDispatch` returns to the caller when no more frames displayed.

Line 38: Exits the application after `gkDispatch` has returned.

Lines 41 - 49: This is the NotifyProc for the Press Me button. When the button is pressed, this routine changes the button's label to `"Thank You"`. The `obj` object handle passed to the routine is the object on which the event occurred (i.e., the button was pressed). Note that we didn't have to change the button size or change the position of the Quit button -- the parent **Form** object handled this automatically.

Lines 51 - 58: This is the NotifyProc for the Quit button. It simply destroys the frame object -- this will cause `gkDispatch` to return back to `main`. When you call `gkDestroy` from within a NotifyProc, it does not destroy the object immediately. Instead, the object is placed on a ''pending destruction'' list and GUI-Kit destroys it when the NotifyProc returns. We call this behavior ''safe destruction'' because it allows you to destroy the object which is invoking the NotifyProc. If GUI-Kit immediately destroyed the object, the object's code which invoked the NotifyProc may try to access the object after the NotifyProc returns -- this would be invalid. Safe destruction is automatic -- the only thing you need to remember is that GUI-Kit delays the actual destruction of the object until the NotifyProc returns.

Attached Objects

Attaching Objects

When using objects in GUI-Kit, you may associate one object with another -- we call this ‘‘attaching’’ one object to another. For example, you might want a custom mouse cursor to appear in a window. To accomplish this, you would create a cursor object and attach it to a window object. You can attach to an object more than one object. In the previous example, you could have attached the cursor to three different windows.

Detaching Objects

When you detach (disassociate) an attached object, GUI-Kit automatically attempts to destroy the object that you attached. You can detach an object by specifying a `NULL` attribute value in place of the object. Detachment of an object also takes place if you replace an attached object with another. For example, if you created a second cursor and attached it to a window, GUI-Kit will detach and attempt to destroy the current cursor. Detachment does not occur if you try to attach the same object a second time. If we also attach the attached object to another object (e.g., you attach a cursor to a second window), the object is considered ‘‘in use’’ and GUI-Kit will not destroy it. GUI-Kit keeps track of an object’s in use status with a reference count. Each time you attach an object to another, GUI-Kit increments the reference count. When you detach an object, GUI-Kit decrements the reference count and attempts to destroy the object. If an object has a reference count greater than zero, `gkDestroy` does not destroy the object and instead returns `GKMETHOD_OBJINUSE`.

Preventing Destruction

You can artificially increment or decrement an object’s reference count by specifying the **IR.ReferenceObj** or **IR.DereferenceObj** attributes, respectively. If you reference an object (i.e., increment the reference count) when you create it, GUI-Kit will never destroy it until you explicitly dereference it and call `gkDestroy`. This method is useful if you want to create a cursor that you may attach and detach periodically and you do not want GUI-Kit to destroy it during a detachment.

An Example - prog1a

To illustrate attached objects, let’s modify the `prog1` example that we used previously. If you prefer, these modifications already exist in `prog1a.c` and you can simply compile it with:

```
dmake APP=prog1a
```

If you’re changing `prog1.c` by hand, replace the `LabelChange` procedure with the following code:

```
static void LabelChange(GKObject obj)
{
    GKObject    gfxobj;
    int         radius;
```

```
    /* A random radius 1 through 30 */
    radius=rand() % 30 + 1;
    gfxobj=gkCreate(NULL, GKGfxArcClass,
                    GKGfxArc_Circle, radius, radius, radius,
                    GKGfxObj_FillStyle, GKFS_SOLID,
                    GKGfxObj_ForegroundColor, "Blue",
                    NULL);

    gkSet(obj,
          GKButton_LabelGfxObj, gfxobj,
          NULL);
}
```

Now when the Press Me button is pressed, this procedure will change the button's label to a solid blue circle with a random radius. The label is a **GfxArc** object, which is a subclass of **GfxObj** (we'll talk more about the **GfxObj** class in just a moment). When the `gfxobj` object is set via the `GKButton_LabelGfxObj` attribute, GUI-Kit detaches and destroys the previous label and it attaches the new label represented by `gfxobj`. If you press the button several times, the circle will change in size. Each time the circle changes, GUI-Kit detaches and destroys the previous **GfxObj** and it attaches the new one.

The GfxObj Class

The **GfxObj** class and its subclasses (**GfxArc**, **GfxContainer**, **GfxImage**, **GfxLine**, **GfxRectangle**, and **GfxText**) comprise a set of graphics objects which allow you to attach graphics to various types of windows. Let's say you attached a **GfxObj** to a **Button** and now the button needs to be redrawn. The **Button** class automatically handles the rendering of the graphics object. Also, if you make a change to the **GfxObj**, the **GfxObj** automatically notifies the button and the button resizes and redraws itself to fit the change. In the `prog1a.c` example, changing the circle's radius would have been more efficient than creating a new graphics object each time the circle changed. As you can see, **GfxObj** handles all of the hairy repainting details for you; you need not write one line of code to handle the repainting of the button's label.

You should note that GUI-Kit treats text like any other graphics object. The **GfxText** class can handle complex formatting of text similar to a word processor. When we created the Press Me button, we specified the attribute `GKButton_Label` -- this attribute is simply a convenient way of specifying a **GfxText** object for `GKButton_LabelGfxObj`. Internally, GUI-Kit takes

the specified label string, creates a **GfxText** object, and attaches the object to `GKButton_LabelGfxObj`.

GUI-Kit uses graphics objects in many classes. When GUI-Kit allows you to specify the visual appearance of an object, it usually accepts a **GfxObj**. Here are brief explanations of **GfxObj** and its subclasses:

- **GfxArc** - Draws arcs, chords, pies, ellipses, and circles.
- **GfxContainer** - Allows you combine one or more graphics objects together as that they appear as one graphics object. This class can also lay out the graphics objects.
- **GfxImage** - Draws, and optionally tiles, bit mapped images.
- **GfxLine** - Draws connected lines and filled or unfilled polygons.
- **GfxRectangle** - Draws filled and unfilled rectangles.
- **GfxText** - As mentioned before, this class formats and draws complex text. It also allows you to change the color and font of the text.

The Instance Registry

The **IR** (Instance Registry) class is the base class for all GUI-Kit classes. **IR** maintains basic information about every object; such as the reference count, the parent object in the instance hierarchy, and the instance's name. IR automatically maintains the reference count and parent object for you; however, it contains several attributes which you can use to write more robust GUI-Kit applications.

Instance Names

GUI-Kit allows you to name any object within your application. To name an object, you specify the `GKIR_InstanceName` attribute on `gkCreate` or `gkSet`. You can retrieve an object's instance name using `gkGet`. Instance names are convenient for eliminating global object variables. Each instance name must be unique within your application; GUI-Kit will generate an error message if you attempt to assign a duplicate name.

To illustrate instance names, make the following modifications to prog1a.c (the source for this is in `prog1b.c`). We have marked lines that we modified with '|' and lines added with '>'.

```
1    #define GKInclAll
2    #include "gk.h"
3
4    /* Prototypes for internal notification procedures  */
5    static void LabelChange(GKObject obj);
```

```
 6    static void Quit(GKObject obj);
 7
 8    /*static GKObject   frame;*/
 9
10    /*-------------------------------------------------------------*/
11    int main(int argc, char **argv)
12    {
13        GKObject    form, frame;
14
15        gkInit(GKInit_Args, &argc, argv,
16               GKInit_StripArgs,
17               NULL);
18
19        frame=gkCreate(NULL, GKFrameClass,
20                       GKIR_InstanceName, "Prog1Frame",
21                       GKFrame_Title, "GUI-Kit 101",
22                       NULL);
23
24        form=gkCreate(frame, GKFormClass,
25                      GKForm_AutoLeftToRight,
26                      NULL);
27
28        gkCreate(form, GKButtonClass,
29                 GKButton_Label, "Press Me",
30                 GKButton_NotifyProc, LabelChange,
31                 NULL);
32
33        gkCreate(form, GKButtonClass,
34                 GKButton_Label, "Quit",
35                 GKButton_NotifyProc, Quit,
36                 NULL);
37
38        gkDispatch();
39        return(0);
40    }
41
42    /*-------------------------------------------------------------*/
43    /* NotifyProc for "Press Me" button.
44     */
45    static void LabelChange(GKObject obj)
46    {
47        GKObject    gfxobj;
48        int         radius;
49
```

```
50        /* A random radius 1 through 30 */
51        radius=rand() % 30 + 1;
52        gfxobj=gkCreate(NULL, GKGfxArcClass,
53                        GKGfxArc_Circle, radius, radius, radius,
54                        GKGfxObj_FillStyle, GKFS_SOLID,
55                        GKGfxObj_ForegroundColor, "Blue",
56                        NULL);
57
58        gkSet(obj,
59              GKButton_LabelGfxObj, gfxobj,
60              NULL);
61    }
62
63    /*-------------------------------------------------------------*/
64    /* NotifyProc for "Quit" button - destroy the frame -- this
65     * will cause gkDispatch to exit.
66     */
67    static void Quit(GKObject obj)
68    {
69        gkDestroy( gkFindInstance("Prog1Frame") );
70    }
```

In this example, we've have eliminated the `frame` global variable and added an instance name of `"Prog1Frame"` to the frame. In the `Quit` procedure (line 69), we use the `gkFindInstance` function to get the frame object by name.

Key Data

The **IR** class also allows you associate arbitrary data with any object -- we call this "key data." You use a key to identify each piece of data that you associate with an object. The key can be any value which C can convert to a pointer; common keys are integers, function pointers, and addresses of variables. Let's consider the following example (the source for this example is in `prog1c.c`):

```
1    #define GKInclAll
2    #include "gk.h"
3
4    /* Prototypes for internal notification procedures  */
5    static void LabelChange(GKObject obj);
6    static void Quit(GKObject obj);
7
8    /*static GKObject   frame;*/
9
```

```
10  /*-------------------------------------------------------------*/
  11    int main(int argc, char **argv)
  12    {
| 13        GKObject    form, frame;
  14
  15        gkInit(GKInit_Args, &argc, argv,
  16               GKInit_StripArgs,
  17               NULL);
  18
  19        frame=gkCreate(NULL, GKFrameClass,
  20                       GKFrame_Title, "GUI-Kit 101",
  21                       NULL);
  22
  23        form=gkCreate(frame, GKFormClass,
  24                      GKForm_AutoLeftToRight,
  25                      NULL);
  26
  27        gkCreate(form, GKButtonClass,
  28                 GKButton_Label, "Press Me",
  29                 GKButton_NotifyProc, LabelChange,
  30                 NULL);
  31
  32        gkCreate(form, GKButtonClass,
  33                 GKButton_Label, "Quit",
  34                 GKButton_NotifyProc, Quit,
> 35                 GKIR_KeyData, (GKVal)Quit, (GKVal)frame,
  36                 NULL);
  37
  38        gkDispatch();
  39        return(0);
  40    }
  41
  42    /*-------------------------------------------------------------*/
  43    /* NotifyProc for "Press Me" button.
  44     */
  45    static void LabelChange(GKObject obj)
  46    {
  47        GKObject    gfxobj;
  48        int         radius;
  49
  50        /* A random radius 1 through 30 */
  51        radius=rand() % 30 + 1;
  52        gfxobj=gkCreate(NULL, GKGfxArcClass,
  53                        GKGfxArc_Circle, radius, radius, radius,
```

```
54                      GKGfxObj_FillStyle, GKFS_SOLID,
55                      GKGfxObj_ForegroundColor, "Blue",
56                      NULL);
57
58          gkSet(obj,
59                GKButton_LabelGfxObj, gfxobj,
60                NULL);
61      }
62
63      /*-------------------------------------------------------*/
64      /* NotifyProc for "Quit" button - destroy the frame -- this
65       * will cause gkDispatch to exit.
66       */
67      static void Quit(GKObject obj)
68      {
69          gkDestroy( (GKObject)gkGet(obj, GKIR_KeyData, (GKVal)Quit) );
70      }
```

On line 35 we associate some key data with the Quit button. We use the `Quit` function pointer as our key and `frame` for the key data. We must cast the key and key data to a `GKVal` type to force them to become generic pointers. On line 69, we use `gkGet` to retrieve the key data associated with the key `Quit`. `gkGet` will return the **Frame** object (the key data) created in `main`.

The `Quit` function pointer is a good key because it will always be unique and we know it probably won't be used as a key by some other function. It is also better than an arbitrarily selected integer; since the compiler must assign a unique address to each function, we always get a unique value. If your entire application follows this convention, you can be certain of always having unique keys. See *Key Data* on page 4-1 for more uses of key data.

Resource Files

If you have a larger application, it can be a hassle to change and recompile the code every time you want to make a minor change to the user interface. GUI-Kit resource files solve this problem by allowing you declare class and object attribute defaults, define groups of objects that you can create on demand, and define symbolic font and color names. If you declare most of your objects in a resource file, you can change your user interface without recompiling your application. This means that your application will contain mostly NotifyProcs which handle the interaction of the user interface.

When your application calls `gkInit`, GUI-Kit attempts to load a resource file with the same base name as your application and a suffix of "`.gkr`". For example, if the program name is `prog1`, `gkInit` attempts to load a resource file with the name `prog1.gkr`. If no such file exists, it loads `gui-kit.gkr` -- the default GUI-Kit resource file. `gkInit` also attempts to load a resource file called `default.gkr` which contains user-specific defaults. Whenever GUI-Kit attempts to load a resource file, it looks for the file in the current directory. If the file is not found, it searches the directories specified by the `PATH` environment variable. This means you can place your application in the same directory as the executable program and GUI-Kit will find it.

Resource File Example

The following example is equivalent to `prog1.c`, only it uses a resource file (source is in `prog1d.gkr` and `prog1d.c`).

```
(File: prog1d.gkr)
 1  #
 2  # Resource file for prog1d (prog1d.gkr)
 3  #
 4
 5  !include "gui-kit.gkr"
 6
 7  ObjectGroup "Prog1Group" {
 8      Object Frame "Prog1Frame" {
 9          Title   "GUI-Kit 101"
10
11          Object Form - {
12              AutoLeftToRight -
13
14              Object Button "Button1" {
15                  Label       "Press Me"
16                  NotifyProc  LabelChange
17              }
18
19              Object Button "Button2" {
20                  Label       "Quit"
21                  NotifyProc  Quit
22              }
23          }
24      }
25  }

(File: prog1d.c)
26  #define GKInclAll
```

```
27  #include "gk.h"
28
29  /* Prototypes for internal notification procedures  */
30  static void LabelChange(GKObject obj);
31  static void Quit(GKObject obj);
32
33  /*-------------------------------------------------------------*/
34  int main(int argc, char **argv)
35  {
36
37      gkRegisterNamedProcedure("LabelChange",
38                               (GKNamedProcedure)LabelChange);
39
40      gkRegisterNamedProcedure("Quit", (GKNamedProcedure)Quit);
41
42      gkInit(GKInit_Args, &argc, argv,
43             GKInit_StripArgs,
44             NULL);
45
46      if (gkCreateObjectGroup("Prog1Group") == -1)
47          gkError("Couldn't create Prog1Group");
48
49      gkDispatch();
50      return(0);
51  }
52
53  /*-------------------------------------------------------------*/
54  /* NotifyProc for "Press Me" button.
55   */
56  static void LabelChange(GKObject obj)
57  {
58      gkSet(obj,
59            GKButton_Label, "Thank You!",
60            NULL);
61  }
62
63  /*-------------------------------------------------------------*/
64  /* NotifyProc for "Quit" button - destroy the frame -- this
65   * will cause gkDispatch to exit.
66   */
67  static void Quit(GKObject obj)
68  {
69      gkDestroy( gkFindInstance("Prog1Frame") );
70  }
```

Lines 1 through 25 are the resource file for `prog1d`. Lines 26 through 70 are the corresponding C source code.

Line 5 includes the GUI-Kit default resource file -- all applications that have resource files should do this so that they define the standard GUI-Kit defaults.

Line 7 starts an object group definition called `Prog1Group`. An object group is a method of associating several objects under one name. As we'll see later, this lets us create all of the objects in the group with one function call.

Line 8 starts the declaration of the frame and assigns it an instance name of `"Prog1Frame."` You should remember that these are simply declarations at this point -- GUI-Kit does not create any objects until you explicitly create the object group.

Line 9 declares the frame's title. Since **Title** is an attribute of the **Frame** class (the class we're creating), specifying **Frame.Title** is not necessary. If we wanted to specify the window's width here, we'd have to specify the **Window.Width** attribute explicitly.

Line 11 starts the declaration of the form. Since the form is a child window of the frame, we include its definition within the frame's definition. Nesting objects automatically create the parent-child relationships necessary for the instance hierarchy. A dash '-' for the instance name says that the form does not have an instance name. Line 12 specifies that the form will be laid-out from left to right; since **AutoLeftToRight** has no value associated with it, we specify a dash '-'.

Lines 14 through 17 declare the Press Me button as a child window of the form and assigns it an instance name of `"Button1."` Similarly, lines 19 through 22 declare the Quit button.

Lines 37 through 40 register named procedures for `LabelChange` and `Quit`. GUI-Kit needs to know the names for these procedures prior to loading the resource file -- this allows you to specify the procedures by name in the resource file.

Line 42: Since we called our resource file `prog1d.gkr` (the same name as the program), `gkInit` will load `prog1d.gkr`. When the resource file is loaded, the definition for the `Prog1Group` object group will also be loaded (but GUI-Kit will not create the objects).

Line 46 creates all the objects defined by the `Prog1Group` object group by using the `gkCreateObjectGroup` function. You can use the `gkDestroyObjectGroup` function to destroy all of the objects created by `gkCreateObjectGroup`. This would allow you to create and destroy a dialog box on demand, for example.

The rest of the code is similar to the previous examples we've seen. Line 69 destroys the frame using its instance name.

If you have compiled `prog1d`, you can experiment with changing various things such as the button labels. For example, try changing `"Press Me"` to `"Click Me"` and then run `prog1d`. Note that you can make changes to the user interface without recompiling `prog1d`.

Where to Go from Here

Now that you have a basic understanding of GUI-Kit, you can compile and run some other GUI-Kit sample programs. The samples can be found in the `\gui_kit\samples` directory.

Further Reading

The proceeding chapters explain each GUI-Kit class in more detail. The following chapters will be useful as references:

- *GUI-Kit Resources* on page 5-1
- *Class and Attribute Reference* on page 23-1
- *Function Reference* on page 24-1
- *Data Type Reference* on page 25-1
- *GUI-Kit Event Reference* on page 26-1
- *GUI-Kit Key Code Reference* on page 27-1

Good luck and don't be afraid to experiment with GUI-Kit! GUI-Kit functions are robust and provide plenty of integrity checking; so if you make a mistake, GUI-Kit will usually catch it.