Introduction to Memory Management in GTK+

Introduction

The purpose of this chapter is to get you to understand how memory is managed in GTK+ so that your applications do not have memory leaks and do not crash because of invalid references. There are only two ways, in essence, that you can foul up a program with respect to how it deals with memory. The first is when it fails to release memory that it no longer needs, causing a *memory leak*. The second is when it tries to reference an object that does not exist at the address that it is using for it. This is either because the object no longer exists, or because it does, but the variable through which it is being accessed has an incorrect address.

You may think that it is not really necessary to prevent memory leaks. After all, there is an awful lot of memory available, and the operating system will clean up after your application when it finishes, so it does not matter whether it is grabbing more physical memory than it really needs while it is running. But if you are trying to be professional about your work, you must not allow your application to grab more memory than it needs, because it will decrease its own performance as well as the performance of the machine on which it runs. And if the leak is bad enough, not only will your program grind to a halt, but the computer will too[1]!

Invalid references inevitably result in crashes and must absolutely be prevented. You never have to worry about this kind of problem if your application never allocates memory dynamically; if every variable that your application uses is either allocated on the run-time stack, i.e., what you would call a *local variable* but is technically called an *automatic variable*, or in the data segment (as when it is declared `static` or is in global scope), then the binding of the address to the variable name is fixed. In GTK+ though, virtually everything that the application creates is through a pointer: widgets, list and tree stores, strings, and so on. For example, creating a button is accomplished with

```
GtkWidget  *button;
button = gtk_button_new();
```

The storage for the button is created by GTK+ and a pointer to it is returned by `gtk_button_new()`. Your application accesses the button widget through the pointer and so there is a possibility that you can foul things up if you do not understand how the memory is managed.

Reference Counting

If you understand reference counting you will not have any problems with memory management. The idea is essentially the same as how UNIX manages disk storage for files, and if you are not familiar with this, then now is as good a time as any to learn about it. In UNIX, a file is represented by an i-node, which is a data structure containing all of the important information about the file, such as who owns it, who has permission to do what to it, how big it is and where its blocks are located, and much more. The i-node also as a link count, which is the number of names that the file has. A file's name is not a part of the i-node. Files can have many names, in the same or different directories, but not spanning multiple file systems. The same file might be named *mydata* in my home directory, and *stewart_data* in some other directory. In this case the link count in the i-node would be 2. If someone else creates a name for the file in a different place,

[1] This is not hypothetical. This exact problem happened once to me.

the link count would go up to 3. Each time that a name is removed, the link count is decremented. If the link count reaches zero, the i-node is deleted as well as the storage for the file.

Objects are reference counted in GTK+. Whereas in UNIX the reference count is the number of names that a file has, in GTK+, it is the number of *owners* of an object. The concept of an owner will be made clear shortly. Objects are created with a reference count of one. Each time that a new reference to an object is created, its reference count is incremented. When a reference is removed, the count is decremented. If the count reaches zero, the object's memory is freed. The act of freeing the memory associated with an object is called *finalizing* the object.

The concept would be fairly simple to understand except for the fact that there are two different types of objects, those that are derived from `GInitiallyUnowned`, and those that are not. Recall that the top of the object hierarchy looks like this:

```
GObject
 +----GInitiallyUnowned
       +----GtkObject
             +----GtkWidget
```

Notice that `GtkWidget` is derived from `GtkObject`, which is derived from `GInitiallyUnowned`, which means that every widget is indirectly derived from `GInitiallyUnowned`. What, you may wonder, is a `GObject` that is not `GInitiallyUnowned`? This list is quite long and includes commonly used objects such as

```
GtkAction
GtkListStore
GtkPrinter
GtkStyle
GtkTextBuffer
GtkTextMark
GtkTextTag
GtkTreeSelection
GtkTreeStore
GtkUIManager
```

Among these are objects that you have encountered already, such as the `GtkTextBuffer`. The full list can be found in the GTK+ object hierarchy at `http://library.gnome.org/devel/gtk/stable/ch01.html`. Objects that are derived directly from `GObject` are where we shall start, for they are easier to understand.

Direct Descendants of `GObject`

Objects descending directly from `GObject` are created with an initial reference count of one. Let us use the `GtkListStore` to demonstrate these ideas. List stores are used with tree view widgets. A tree view widget displays data in a row and column format, much the way a file browser does. The list store is one type of object whose data can be viewed with a tree view. It is essentially a linked list. A `GtkListStore` is created with the function

```
GtkListStore* store = gtk_list_store_new();
```

which creates `store` with a reference count of one. One says that the application *owns the reference* to `store`. The documentation would say that *this particular code owns the reference*, because the application can be large, and it is better to think of the separate parts of it as owning the objects they create.

The reference count of any object can be incremented by calling `g_object_ref()` on the object. Thus, to increment the reference count of `store`, we would call

```
g_object_ref(G_OBJECT( store ) );
```

To decrement the reference count of an object, we use `g_object_unref()`, as in

```
g_object_unref( G_OBJECT( store ) );
```

which decreases the reference count by one.

The method that assigns a `GtkListStore` to a tree view is `gtk_tree_view_set_model()`, as in

```
gtk_tree_view_set_model( GTK_TREE_VIEW( treeview ), GTK_TREE_MODEL( store ) );
```

After the `gtk_tree_view_set_model()` function has returned, the tree view, `treeview`, will own a reference to the list store. Methods such as `gtk_tree_view_set_model()`, which acquire ownership of an object, increment the object's reference count using `g_object_ref()` in their implementations. If they did not do this, then, if an application ever called `g_object_unref()` on the object, its reference count would drop to zero and it would be finalized. At that point, they would hold a reference to invalid memory and a crash would ensue at the next attempt to access the data. So after the list store has been added to the tree view, its reference count is two.

The way that a `GtkTreeView` makes a `GtkListStore` what it displays is similar to the way that a `GtkTextView` widget makes a text buffer what it displays, using `gtk_text_view_set_buffer()`, as in

```
gtk_text_view_set_buffer( GTK_TEXT_VIEW( textview ), GTK_TEXT_BUFFER( buffer ) );
```

Because the text view acquires ownership of `buffer`, `gtk_text_view_set_buffer()` increments the reference count of `buffer`. If `buffer` was created by some other portion of the code that acquired ownership at creation time, the reference count of `buffer` would be two, indicating that `buffer` had two owners. (Text buffers can be shared by multiple text view widgets.)

Let us return to the discussion about the list store. Your application should no longer need to access the list store once it is added to the tree view. All access to the list store takes place through the tree view. Therefore, as soon as it has called `gtk_tree_view_set_model()`, it must relinquish its reference and decrement the reference count with `g_object_unref()` thus:

```
gtk_tree_view_set_model( GTK_TREE_VIEW( treeview ), GTK_TREE_MODEL( store ) );
g_object_unref(G_OBJECT( store ) );
```

This ensures that the reference count is dropped to one, so that when the tree view is destroyed and it unrefs all of its child objects, the store's reference count will become zero and it will be finalized as well. I will talk more about the subject of destruction later.

To summarize, when your application creates an object that descends directly from `GObject`, once it has attached it to an object or widget that assumes ownership, you should call `g_object_unref()` on the object.

Objects Descending from `GInitiallyUnowned`

The story is very different for objects descending from `GInitiallyUnowned`. All such objects are created with a *floating reference count* of one. A floating reference is a reference that is not owned by anyone. This is why the class is called `GInitiallyUnowned`, because initially, objects of this class are un-owned. A floating reference can be thought of as a special kind of reference not associated with any owner. Technically it is a flag in the `GObject` structure that indicates whether or not the initial reference is floating or not; when it is set, the object has a floating reference and when it is clear, the reference is non-floating.

The need for floating references is two-fold. One reason for them is that they are a way to keep objects alive after they are created but before they are attached to parent containers. This will be explained below. The other reason stems from the way that functions can be called in C. Consider the following code:

```
container = create_container();
container_add_child (container, create_child());
```

Suppose that `container` is some type of container widget and that `container_add_child()` adds a child widget to the given container and also adds a reference to the given child, which in this case is the object created by the call to `create_child()`. Suppose that floating references did not exist and that `create_child()` creates an object with a reference count of one. Since the container acquires a reference to the object and increments the reference count, the child object would have a reference count of two after the call. However, because the return value of `create_child()` is not assigned to any object, there is no object on which to call `g_object_unref()` to decrement the count to one. When the container is destroyed, it can "unref" the child object, but the child object's count will drop to one, not zero, and there will be a memory leak since the child's memory can not be freed until the reference count reaches zero.

The following code could be used in the absence of floating references without causing a memory leak:

```
Child     *child;
container = create_container();         // container ref-count =1
child     = create_child();             // child ref-count = 1
container_add_child (container, child); // child ref-count = 2
g_object_unref (child);                 // child ref-count = 1
```

The difference is that the return value from `create_child()` is stored in `child`, so that `g_object_unref()` can be called on `child`. However, since it is possible to write code in C the first way as well, the `GObject` library was designed to allow such code to work correctly.

The idea of floating references allows the above code to be free of memory leaks, but only if the container has the ability to acquire ownership and convert the floating reference to a *standard reference*, which is what ordinary references are called. The changes are as follows:

- A function that creates a widget, such as `create_child()` above, creates it with a *floating reference count of one*, i.e., one that is *initially unowned*, rather than a standard reference count of one.
- A function that adds a child widget to a container, such as `container_add_child()` above, does not call `g_object_ref()` on the child object because that will not clear the floating reference flag. Instead it calls `g_object_ref_sink()`.

The function `g_object_ref_sink()` exists so that, when a widget is added to a parent container, the container can do two things:

1. remove the floating reference, and
2. acquire its own standard reference to the widget.

The call

```
g_object_ref_sink(object)
```

works as follows. If `object` had a floating reference, then after the call, the code that executed it owns a reference to `object`, the reference count is unchanged, and the floating flag is cleared. If `object` did not have a floating reference, `g_object_ref_sink()` has the same effect as `g_object_ref()`, namely it increments the reference count. Thus, conceptually you should think of the sink operation as

```
if ( was_floating(object) )
    clear( object->floating_flag );
else
    g_object_ref(object);
```

In the example code above,

```
container = create_container();
container_add_child (container, create_child());
```

`create_child()` creates a `Child` widget with a floating reference of one, and the `container_add_child()` function calls `g_object_ref_sink()` on the anonymous child widget (as an object of course), acquiring ownership of it, giving it a standard reference count of one. When the container is destroyed it will unref the child, dropping its count to zero, causing the child to be finalized without any need for the application to unref the child outside of `container_add_child()`.

All widgets except for top-level windows start out life with a floating reference. Top-level windows are different because they are never put into containers – they are the roots of the container trees. When a top-level window is created, the GTK+ library immediately sinks its floating reference and assumes ownership of it, so when it is handed to your application, it has a standard reference count of one.

If an application creates a widget that is not a top-level window, then at some point it will probably be packed into a container. All of the functions that pack widgets into containers automatically sink the widget's floating reference and give the container ownership of the child. This way, when the parent widget eventually receives a destroy signal and unrefs its child objects, their standard reference counts will drop to zero and they will be finalized (unless in the application, for one reason or another, the code calls `g_object_ref()` on any of the children without a corresponding `g_object_unref()`, causing a memory leak!) To be clear, when your application creates a widget of type `foo` using a constructor such as `gtk_foo_new()`, and adds that widget to a container, it never needs to do anything else to manage the widget's memory; GTK+ will take care of freeing the memory when the widget is destroyed.

We can shed more light on this topic with the following example. You might expect that the two instructions

```
GtkWidget *gadget = gtk_gadget_new ();
gtk_widget_destroy (gadget);
```

will simply create `gadget`, allocating memory for it, and then immediately de-allocate its memory. On the face of it that seems to make sense. But in fact, this will cause a memory leak! When `gadget` is created, it has a floating reference only. The call to `gtk_widget_destroy()` is equivalent to

```
gtk_object_destroy(GTK_OBJECT(gadget));
```

The documentation for `gtk_object_destroy` states that

> The memory for the object itself won't be deleted until its reference count actually drops to zero; `gtk_object_destroy()` merely asks reference holders to release their references, it does not free the object.

In other words, `gtk_widget_destroy()` does not unref the object; it just asks reference holders to let go of their references. Since `gadget` was not packed into any container, and therefore `g_object_ref_sink()` was not called on it, it is not owned by any reference holder, and so `gtk_widget_destroy()` will not free the memory held by `gadget`. As a result, there is a memory leak. Of course there is no reason to write code like this, but in case for some strange reason you wanted to create a widget and then later destroy it and free the memory that it holds, without ever adding it to a container, you would have to unref it yourself in the code:

```
GtkWidget *gadget = gtk_gadget_new ();
g_object_ref_sink(G_OBJECT(gadget));   // convert floating ref to standard ref
gtk_widget_destroy (gadget);           // break external references
g_object_unref (G_OBJECT(gadget));     // decrease ref count to 0
```

Once again, there is little reason to create a widget that you have no intention of adding to a container other than a top-level window[2]. The purpose of the preceding discussion is to clarify how reference counting works with respect to objects such as widgets that are derived from `GInitiallyUnowned`.

GLib Strings

There are different types of strings that can be used in a GTK+ application written in C:

```
char   *string1;  // The standard C string
gchar  *string2;  // Identical to a standard C string as gchar is a typedef of char
GString string3;  // An extension of C strings that can grow automatically
```

When using standard C string functions, such as `g_strdup()`, `g_strnfill()`, or `g_strdup_printf()`, the general rule is that, if the documentation says that the function returns a newly allocated string, then the returned string should be freed with `g_free()`. Usually the documentation will explicitly state that the returned string must be freed with `g_free()`. If it says neither of these things, then do not call `g_free()` on the string, unless you would like the program to crash.

A `GString` object is a structure

```
typedef struct {
    gchar *str;
    gsize len;
    gsize allocated_len;
} GString;
```

whose memory GLib manages. GLib provides functions to grow and shrink a `GString` object. The internal pointer, `str`, is the address of the buffer, and may be moved around as the string grows and shrinks. A `GString` object is created with one of the `g_string_new()` family of functions. It must be freed, when you are finished with it, using `g_string_free()`. If you want both the structure and the buffer contents within the structure to be freed, then the proper call is

```
gchar* buf = g_string_free(string, TRUE);
```

The second argument controls whether the string data is also freed, or just the wrapping structure. If `TRUE` is passed, the return value is `NULL`. If the second argument is `FALSE`, then the buffer is not freed and the return value is a pointer to the buffer, which must be freed when the code is finished with it, using `g_free()`.

GdkPixbuf Structures

`GdkPixbuf` structures are derived directly from `GObject` and are reference counted. They are created with a reference count of one. An application can share a single pixbuf among many parts of its code. When a part of the program needs to hold a pointer to a pixbuf, it should add a reference to it by calling `g_object_ref()`. When it is finished with the pixbuf, it should call `g_object_unref()` to decrement the reference count. The pixbuf will be destroyed when its reference count drops to zero. The reference count in general needs to be equal to the number of distinct pointers to the object.

[2]If you are thinking that dialogs are not added to containers but that they get destroyed by gtk_widget_destroy(), it is because dialogs are top-level windows and GTK+ owns a reference to them. The destroy signal causes GTK+ to call unref on the dialog, freeing its memory.

The rules described above for C strings generally apply to pixbufs as well. Certain functions that return a `GdkPixbuf` pointer will note in their documentation that they return a newly allocated pixbuf with a reference count of one. If you use such a function, then you must call `g_object_unref()` on it when you are finished using the newly allocated pixbuf. If a pixbuf was used as a source for creating a new pixbuf, as is the case with `gdk_pixbuf_add_alpha()`, which will create a modified version of a pixbuf, and the original is no longer needed, then you should call `g_object_unref()` on it immediately afterward to relinquish ownership and decrement the reference count.

The last thing that you need to be aware of regarding pixbufs is that, if you use `gdk_pixbuf_new_from_data()` to create a pixbuf from data stored internally in memory already, such as an array of pixel values, then you may also need to provide a function that "knows how" to free the memory belonging to that data when the reference count on the pixbuf drops to zero. You should consult the documentation if you plan to use this function.

List and Tree Stores

The `GtkTreeModel` interface defines a generic tree interface for use by the `GtkTreeView` widget. It is an abstract interface that is designed to be usable with any appropriate data structure. The `GtkTreeStore` and the `GtkListStore` are two implemented GTK+ tree models. They provide the data structure as well as all appropriate tree interfaces. Populating them with data requires using either the `gtk_list_store_set()` method or the `gtk_tree_store_set()` method respectively. There are variations on these two functions, but the remarks below are true regardless of which variation is used.

Both `gtk_list_store_set()` and `gtk_tree_store_set()` accept a variable number of arguments, which are essentially ordered pairs, as in

```
void gtk_tree_store_set (GtkTreeStore *store,
                         GtkTreeIter *iter,
                         ..., -1);
```

where the missing arguments are pairs of the form (*column_id*, *value*) in which *column_id* is an integer and *value* may be any value whatsoever, such as strings, integers, pixbufs, or pointers to arbitrarily complex structures. What you need to know is whether the data passed to this function is copied into the store's row, or accessed by reference. For example, in the call

```
gchar  name[] = "Groucho";
gtk_tree_store_set (store, &iter, NAME_ID, name, -1);
```

the question is whether the actual string data stored in name is copied into the row referenced by the iterator, or whether a pointer to name is copied into the row.

The short answer is that you do not need to worry about allocating and freeing memory for the data to store because the underlying GLib/GObject GType and GValue system takes care of most memory management automatically. For example, if you store a string in a column of a row, the model will make a copy of the string and store the copy. If you later change the column to a new string, the model will automatically free the old string and again make a copy of the new string and store the copy.

The long answer is that when data is added to a list or tree store using any of the `gtk_*_store_set()` functions, how it is handled by GTK+ depends on which of three categories of data it falls into.

1. If the data is a `GObject` (i.e., an object derived directly from `GObject`) , then the store takes ownership of it by calling `g_value_dup_object()` on the object and relinquishing ownership of the previously held object, if it is replacing an old value.

2. If the data is a simple scalar data type such as a numeric, Boolean, or enumerated type or a pointer, then the store makes a copy of the data. Note that the pointer is copied, not the data to which it points.

3. If the data is a string or a boxed structure, then the store duplicates the string or the boxed structure and stores a pointer to it. If the data is replacing existing data, then in this case the string or boxed structure is freed first using `g_free()` or `g_boxed_free()` respectively. (`GBoxed` is a wrapper mechanism for arbitrary C structures. They are treated as opaque chunks of memory.)

This may leave you wondering about questions such as how GTK+ handles pixbufs. As noted above, pixbufs are derived directly from `GObject` and fall under category 1 above. When retrieving data from a store with `gtk_tree_model_get()` or any of its variants, you have to be aware of how GTK+ handles the different types of data, so that you know whether or not to free its memory when you are finished with it:

1. If the data being obtained is a `GObject`, the function increments its reference count, since it is providing another reference to it for us.

2. If the data is a simple scalar data type such as a numeric, Boolean, or enumerated type or a pointer, the function makes a copy of it and delivers the copy.

3. If the data is a string or a boxed type, the function copies it and returns a pointer to the data.

This implies that we need to call `g_object_unref()` on objects acquired from the store when we are finished with them, and free the data we retrieved from store if it is string or boxed type data (using `g_free()` or `g_boxed_free()` respectively) when finished with them. Focusing on pixbufs again, they are category 1 and so it is necessary to relinquish ownership of a pixbuf acquired from a store using `gtk_tree_model_get()` by unref-ing it. No other data needs to be memory managed.

Closing Thoughts

When you are debugging your program, it is a good idea to turn on a system monitor and watch the virtual and physical memory usage of the program. If you see it steadily climbing as the application runs, then you have a leak somewhere. My suggestion is that you put the program through repeated actions such as clicking the same menu item repeatedly, to see which is the culprit.

It will be obvious if you have freed memory that you were not supposed to free because the application will crash. You should look at the bug report issued by your window manager (e.g. Gnome) and look at the stack trace. Starting at the top of the stack, descend it until you find the first function whose code is yours, as opposed to a GTK+ library. Therein lies the code that killed the beast. Work backwards through it with the help of what this document provides and you will find the mistake.

There are more formal methods of finding leaks, but the preceding approach is relatively easy to do and does not require learning how to use a new tool. If you do want to learn how to use a good tool, get a copy of Valgrind (`http://valgrind.org/`) which is open source, free software for memory management analysis.