Manage C data using the GLib collections

Open source library adds a wide range of useful data utilities

Tom Copeland (tom@infoether.com)
Developer
InfoEther

28 June 2005

In this tutorial, learn how to use the GLib collection data structures to effectively manage data within C programs. In particular, you'll see how to use GLib's built-in data structures/containers -- linked lists, hash tables, arrays, trees, queues, and relations -- to fill the need for them in C.

Before you start

About this tutorial

This tutorial shows you how to use the GLib collections to manage data efficiently and elegantly within your C programs. The GLib collections are the result of many years of refinement and are used by numerous open source programs. These collections provide the more complex data structures/containers (the functions and variables you need to manage data) that are in short supply in the C language.

This tutorial is written for Linux™ or UNIX® programmers whose skills and experience are at a beginning to intermediate level.

Prerequisites

To get the most out of this tutorial, you should be generally familiar with a UNIX-like environment and know how to use a command-line shell.

You also need some basic programming tools to compile the source code examples, such as a compiler like GCC (see the Resources section for downloading GCC); all of the code examples in this tutorial were compiled with GCC 3.4.2.

You also need the GLib runtime and development libraries installed. Most modern Linux distributions come with the GLib runtime installed; for example, the "workstation" installation of Fedora Core 3 comes with two GLib RPMs: glib2-2.4.7-1 and glib2-devel-2.4.7-1.

Organizing data

GLib's scope

First let's review the scope of GLib.

GLib is a lower-level library that provides many useful definitions and functions, including definitions for basic types and their limits, standard macros, type conversions, byte order, memory allocation, warnings and assertions, message logging, timers, string utilities, hook functions, a lexical scanner, dynamic loading of modules, and automatic string completion.

GLib also defines a number of data structures (and their related operations), including:

- Memory chunks
- Doubly-linked lists
- Singly-linked lists
- Hash tables
- Strings (which can grow dynamically)
- String chunks (groups of strings)
- Arrays (which can grow in size as elements are added)
- Balanced binary trees
- N-ary trees
- Quarks (a two-way association of a string and a unique integer identifier)
- Keyed data lists (lists of data elements accessible by a string or integer id)
- Relations and tuples (tables of data which can be indexed on any number of fields)
- Caches

Every program has to manage data

Programs are written to manipulate data. Your program may read in a list of names from a file, prompt a user for some data through a graphical user interface, or load data from an external hardware device. But once the data is in your program, it's up to you to keep track of it. The functions and variables you use to manage data are called *data structures* or *containers*.

If you're writing code in C, you'll find that it's pretty short on complex data structures. There are lots of simple ways to store data, of course:

- The primitive types -- `int`s, `float`s, `char`s, and so forth.
- `enum`, which can hold a series of symbolic names for integers.
- The array, which is C's most flexible data structure.

An array can hold primitives or a series of any type of data or pointers to any type of data.

But arrays have lots of limitations, too. They can't be resized, so if you allocate memory for an array of ten items and find you need to put eleven things in it, you need to create a new array, copy the old items in, and then put in the new item. If you're going to iterate over every item in an array,

you either have to have kept track of how many items are in the array or ensure there's some sort of "end of array" marker at the tail of the array so that you know when to stop.

The problems with keeping track of data in C have been solved many times over by the use standard containers like the linked list and the binary tree. Every freshman computer science major takes a data structures class; the instructor is sure to assign a series of exercises on writing implementations of those containers. While writing these structures, the student gains an appreciation for how tricky they are; dangling pointers and double `free`s wait around every corner to trap the unwary student.

Writing unit tests can help a lot, but overall, rewriting the same data structure for every new program is a thankless task.

Built-in data structures

That's where built-in data structures help. Some languages come with these containers built in. C ++ contains the Standard Template Library (STL), which has a collection of container classes like lists, priority queues, sets, and maps. These containers are also *type-safe*, meaning that you can only put one type of item in each container object that you create. This makes them safer to use and eliminates a lot of tedious casting that C requires. And the STL contains a host of iterators, sorting utilities, and so forth to make working with the containers easier.

The Java programming language also comes with a set of container classes. The `java.util` package contains `ArrayList`, `HashMap`, `TreeSet`, and various other standard structures. It also includes utilities for generically sorting data and creating immutable collections, as well as various other handy bits.

With C, however, there's no built-in container support; you either have to roll your own or use someone else's data structure library.

Fortunately, GLib is an excellent, free, open source library that fills this need. It contains most of the standard data structures and many of the utilities that you need to effectively manipulate data in your programs. And it's been around since 1996, so it's been thoroughly tested with a lot of useful functionality added along the way.

Algorithm analysis in 100 words (or fewer)

Different operations on containers take different amounts of time. For example, accessing the first item in a long list is a lot faster than sorting that same list. The notation used to describe the time to do these operations is called *O-notation*. This topic is worthy of a semester of a computer science major's time, but in a nutshell, O-notation is a worst-case analysis of an operation. In other words, it's a measurement of the *longest* time that an operation will take to complete. It turns out to be a useful way to measure data structure operations since the worst-case operation is frequently encountered (such as when you search a list and don't find the item you were looking for).

The following demonstrates some O-notation examples using things you could do with a set of playing cards that are arranged in a line face down on a table:

- `O(1)` -- Selecting the first card. `O(1)` is also known as "constant time" because picking up the first card takes the same amount of time no matter how many cards are in the list.
- `O(n)` -- Turning over each card. `O(n)` is known as "linear time" because the time to do it increases linearly as the number of cards increases.
- `O(n!)` -- Creating a list of all the possible permutations of all the cards. For every new card that gets added to the list, the number of permutations increases factorially.

Throughout this tutorial you'll see references to the O-notation of operations on various data structures. Knowing the costs of a particular operation on a specific data structure can help you choose containers wisely and maximize your application's performance.

Compiling GLib programs

You'll learn more in this tutorial if you follow along with the examples by compiling and running them. Since they use GLib, you need to tell the compiler where the GLib header files and libraries are so it can resolve the GLib-defined types. This simple program initializes a doubly-linked list and then adds a string of characters to it:

```
//ex-compile.c
#include <glib.h>
int main(int argc, char** argv) {
 GList* list = NULL;
 list = g_list_append(list, "Hello world!");
 printf("The first item is '%s'\n", g_list_first(list)->data);
 return 0;
}
```

You can compile this program by invoking GCC like this:

```
$ gcc -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include
   -lglib-2.0 -o ex-compile ex-compile.c
```

And run it to see the expected output:

```
$ ./ex-compile
The first item is 'Hello world!'
$
```

That's quite a laborious GCC invocation, though. A simpler way to point GCC to the GLib libraries follows.

Using pkg-config

Manually specifying library locations is fragile and tedious, so most modern Linux distributions come with the *pkgconfig* utility to help make this easier. You can use pkgconfig to compile the program above like this:

```
$ gcc 'pkg-config --cflags --libs glib-2.0' -o ex-compile ex-compile.c
```

And the output is the same as before:

```
$ ./ex-compile
The first item is 'Hello world!'
$
```

Note that now you don't have to specify the paths to the GLib header files anymore; pkgconfig's `--cflags` option takes care of that. And the same goes for the libraries that are pointed to by the `--libs` option. Of course, there's no magic involved; pkgconfig just reads the library and header file locations from a configuration file. On a Fedora Core 3 system, the pkgconfig files are located in /usr/lib/pkgconfig, and the glib-2.0.pc file looks like this:

```
$ cat /usr/lib/pkgconfig/glib-2.0.pc
 prefix=/usr
 exec_prefix=/usr
 libdir=/usr/lib
 includedir=/usr/include

 glib_genmarshal=glib-genmarshal
 gobject_query=gobject-query
 glib_mkenums=glib-mkenums

 Name: GLib
 Description: C Utility Library
 Version: 2.4.7
 Libs: -L${libdir} -lglib-2.0
 Cflags: -I${includedir}/glib-2.0 -I${libdir}/glib-2.0/include
```

So all the information is just hidden away by a layer of indirection. And if you happen to have a Linux distribution that doesn't support pkgconfig, you can always just fall back to pointing GCC directly to the header files and libraries.

Real-world GLib usage

Merely enumerating the GLib containers and showing example usages might be a bit dry, so this tutorial also includes real-world usage of GLib in several open source applications:

- Gaim is a popular instant messenger client that is downloaded from SourceForge more than a quarter of a million times each month.
- The GIMP (Graphical Image Manipulation Program) served as the starting point for GLib itself; it's a widely used image-processing program and has been under public development since 1996.
- Evolution is an excellent personal information manager (PIM) that tracks emails, contacts, tasks, and appointments.

Looking at GLib usage in these popular applications also gives you a chance to see some coding idioms; rather than just knowing what the function names are, you can also see how they are commonly used. You'll get a feel for the containers that are being used and maybe you'll even notice some places where someone's picked a container that might not be the best one for the job.

GLib also has many conventions and utility macros. As you go through this tutorial, you'll see many of these used and explained. Rather than try to memorize them all up front, just learn them as you go along and see them in action.

Singly-linked lists

Concepts of singly-linked lists

Perhaps the simplest container in GLib is the singly-linked list; the *GSList*. As its name implies, it's a series of data items that are linked together so that you can navigate from one data item to the next. It's called a singly-linked list because there's only a single link between the items. So, you can only move "forward" through the list, but you can't move forward and then back up.

To drill in a bit further, every time you append an item to the list, a new GSList structure is created. This GSList structure consists of a data item and a pointer. The previous end of the list is then pointed to this new node, which means that now the new node is at the end of the list. The terminology can be a bit confusing because the entire structure is called a GSList and each node is a GSList structure as well.

Conceptually though, a list is just a sequence of lists that are each one item long. It's as if it were a line of cars at a stoplight; even if there were only one car waiting at the stoplight, it'd still be considered a line of cars.

Having a list of items linked together has some usage implications. Determining the length of the list is an O(n) operation; you can't figure out how long the list is unless you count each item. Adding to the front of the list is fast (an O(1) operation) since the list is not a fixed length and doesn't need to be rebuilt once it exceeds a threshold. But finding an item is an O(n) operation since you need to do a linear search over the entire list until you find what you're looking for. Adding an item to the end of the list is also an O(n) operation since to get to the end you need to start at the beginning and iterate until you reach the end of the list.

A GSList can hold two types basic of data: integers or pointers. But this really means that you can put pretty much anything in a GSList. For example, if you wanted a GSList of the "short" data type, you could just put pointers to the shorts in the GSList.

That's enough theory for now; on to actually using GSList!

Creating, adding, and destroying

The following code initializes a GSList, adds two items to it, prints out the list's length, and frees it:

```
//ex-gslist-1.c
#include <glib.h>
int main(int argc, char** argv) {
 GSList* list = NULL;
 printf("The list is now %d items long\n", g_slist_length(list));
 list = g_slist_append(list, "first");
 list = g_slist_append(list, "second");
 printf("The list is now %d items long\n", g_slist_length(list));
 g_slist_free(list);
 return 0;
}

***** Output *****

The list is now 0 items long
The list is now 2 items long
```

A couple of notes on the above code:

- Most GLib functions are of the format `g_(container name)_(function name)`. So to get the length of a GSList, you'd call `g_slist_length`.
- There's no function to create a new GSList; instead, just declare a pointer to a GSList structure and assign it a value of `NULL`.
- `g_slist_append` returns the new start of the list, so you need to hang on to that return value.
- `g_slist_free` doesn't care whether any items have been placed in the list or not; a quick peek at the source code shows that `g_slist_free` just returns immediately if the GSList is NULL. `g_slist_length` also works with an empty list; in that case, it just returns 0.

Adding and then removing data

You can put data in; you'll probably also need to take it out. Here's an example:

```
//ex-gslist-2.c
#include <glib.h>
int main(int argc, char** argv) {
 GSList* list = NULL;
 list = g_slist_append(list, "second");
 list = g_slist_prepend(list, "first");
 printf("The list is now %d items long\n", g_slist_length(list));
 list = g_slist_remove(list, "first");
 printf("The list is now %d items long\n", g_slist_length(list));
 g_slist_free(list);
 return 0;
}

***** Output *****

The list is now 2 items long
The list is now 1 items long
```

Most of this code should look familiar, but there are some points to consider:

- If you call `g_slist_remove` and pass in an item that's not in the list, the list will be unchanged.
- `g_slist_remove` also returns the new start of the list.
- You can see that "first" is added with a call to `g_slist_prepend`. This is a faster call than `g_slist_append`; it's O(1) rather than O(n) because, as mentioned before, doing an append requires a full list traversal. So if it's at all convenient to use `g_slist_prepend`, that's the one you want.

Removing duplicate items

Here's a wrinkle that shows what happens when you have duplicate items in a list:

```
//ex-gslist-3.c
#include <glib.h>
int main(int argc, char** argv) {
 GSList* list = NULL;
 list = g_slist_append(list, "first");
 list = g_slist_append(list, "second");
 list = g_slist_append(list, "second");
 list = g_slist_append(list, "third");
```

```
 list = g_slist_append(list, "third");
 printf("The list is now %d items long\n", g_slist_length(list));
 list = g_slist_remove(list, "second");
 list = g_slist_remove_all(list, "third");
 printf("The list is now %d items long\n", g_slist_length(list));
 g_slist_free(list);
 return 0;
}

***** Output *****

The list is now 5 items long
The list is now 2 items long
```

So if a GSList contains the same pointer twice and you call `g_slist_remove`, only the first pointer will be removed. But you can remove all occurrences of an item with `g_slist_remove_all`.

Last, nth, and nth data

Once a few items are in a GSList, you can pick out items in various ways. Here are some examples, with explanations in the accompanying `printf` statements:

```
//ex-gslist-4.c
#include <glib.h>
int main(int argc, char** argv) {
 GSList* list = NULL;
 list = g_slist_append(list, "first");
 list = g_slist_append(list, "second");
 list = g_slist_append(list, "third");
 printf("The last item is '%s'\n", g_slist_last(list)->data);
 printf("The item at index '1' is '%s'\n", g_slist_nth(list, 1)->data);
 printf("Now the item at index '1' the easy way: '%s'\n", g_slist_nth_data(list, 1));
 printf("And the 'next' item after first item is '%s'\n", g_slist_next(list)->data);
 g_slist_free(list);
 return 0;
}

***** Output *****

The last item is 'third'
The item at index '1' is 'second'
Now the item at index '1' the easy way: 'second'
And the 'next' item after first item is 'second'
```

Note that there are some shortcut functions on GSList; you can simply call `g_slist_nth_data` rather than calling `g_slist_nth` and then dereferencing the returned pointer.

The last `printf` statement is a bit different. `g_slist_next` is not a function call, but rather a macro. It expands to a pointer derefence of the link to the next element in the GSList. In this case, you can see that we passed in the first element in the GSList, so the macro expanded to provide the second element. It's a fast operation too, since there's no function call overhead.

A step back: Working with a user-defined type

So far we've just been working with strings; that is, we've just been putting pointers to characters in the GSList. In the code sample below, you'll define a `Person` struct and push a few instances of that into a GSList:

```
//ex-gslist-5.c
#include <glib.h>
typedef struct {
 char* name;
 int shoe_size;
} Person;
int main(int argc, char** argv) {
 GSList* list = NULL;
 Person* tom = (Person*)malloc(sizeof(Person));
 tom->name = "Tom";
 tom->shoe_size = 12;
 list = g_slist_append(list, tom);
 Person* fred = g_new(Person, 1); // allocate memory for one Person struct
 fred->name = "Fred";
 fred->shoe_size = 11;
 list = g_slist_append(list, fred);
 printf("Tom's shoe size is '%d'\n", ((Person*)list->data)->shoe_size);
 printf("The last Person's name is '%s'\n", ((Person*)g_slist_last(list)->data)->name);
 g_slist_free(list);
 free(tom);
 g_free(fred);
 return 0;
}

***** Output *****

Tom's shoe size is '12'
The last Person's name is 'Fred'
```

A few notes about working with GLib and user-defined types:

- You can see that putting a user-defined type in a GSList is the same as a character string. Note also that you need to do a bit of casting when you're getting the item out of the list.
- This example uses another GLib macro -- the `g_new` macro -- to create the `Fred Person` instance. This macro simply expands to use `malloc` to allocate the correct amount of memory for the given type, but it's a bit cleaner than manually typing the `malloc` function call.
- Finally, if you're going to allocate memory, you need to free it. You can see how the code sample above uses the GLib function `g_free` to do just that for the `FredPerson` instance (since it was allocated with `g_new`). In most cases `g_free` just wraps the usual `free` function, but GLib also has memory pooling functionality that `g_free` and other memory-management functions can use.

Combining, reversing, and all that

GSList comes with some handy utility functions that can concatenate and reverse lists. Here's how they work:

```
//ex-gslist-6.c
#include <glib.h>
int main(int argc, char** argv) {
 GSList* list1 = NULL;
 list1 = g_slist_append(list1, "first");
 list1 = g_slist_append(list1, "second");
 GSList* list2 = NULL;
 list2 = g_slist_append(list2, "third");
 list2 = g_slist_append(list2, "fourth");
 GSList* both = g_slist_concat(list1, list2);
 printf("The third item in the concatenated list is '%s'\n", g_slist_nth_data(both, 2));
 GSList* reversed = g_slist_reverse(both);
```

```
 printf("The first item in the reversed list is '%s'\n", reversed->data);
 g_slist_free(reversed);
 return 0;
}

***** Output *****

The third item in the concatenated list is 'third'
The first item in the reversed list is 'fourth'
```

As expected, the two lists were concatenated head to tail so that the first item in list2 became the third item in the new list. Note that the items aren't copied; they're just hooked on so that the memory needs to be freed only once.

Also, you can see that you can print out the first item in the reversed list using just a pointer dereference (`reversed->data`). Since each item in a GSList is a pointer to a GSList structure, you don't need to call a function to get the first item.

Simple iterating

Here's a straightforward way to iterate over the contents of a GSList:

```
//ex-gslist-7.c
#include <glib.h>
int main(int argc, char** argv) {
 GSList* list = NULL, *iterator = NULL;
 list = g_slist_append(list, "first");
 list = g_slist_append(list, "second");
 list = g_slist_append(list, "third");
 for (iterator = list; iterator; iterator = iterator->next) {
  printf("Current item is '%s'\n", iterator->data);
 }
 g_slist_free(list);
 return 0;
}

***** Output *****

Current item is 'first'
Current item is 'second'
Current item is 'third'
```

The iterator object is just a variable declared as a pointer to a GSList structure. This seems odd, but it's what you would expect. Since a singly-linked list is a series of GSList structs, the iterator and the list should be of the same type.

Note also that this code uses a common GLib usage idiom; it declares the iterator variable at the same time that it declares the GSList itself.

Finally, the `for` loop exit expression checks for the iterator being `NULL`. This works since it will only be `NULL` after the loop has passed the last item in the list.

Advanced iteration with functions

Another way to iterate over a GSList is to use the `g_slist_foreach` function and supply a function to be called for each item in the list.

```
//ex-gslist-8.c
#include <glib.h>
void print_iterator(gpointer item, gpointer prefix) {
 printf("%s %s\n", prefix, item);
}
void print_iterator_short(gpointer item) {
 printf("%s\n", item);
}
int main(int argc, char** argv) {
 GSList* list = g_slist_append(NULL, g_strdup("first"));
 list = g_slist_append(list, g_strdup("second"));
 list = g_slist_append(list, g_strdup("third"));
 printf("Iterating with a function:\n");
 g_slist_foreach(list, print_iterator, "-->");
 printf("Iterating with a shorter function:\n");
 g_slist_foreach(list, (GFunc)print_iterator_short, NULL);
 printf("Now freeing each item\n");
 g_slist_foreach(list, (GFunc)g_free, NULL);
 g_slist_free(list);
 return 0;
}

***** Output *****

Iterating with a function:
--> first
--> second
--> third
Iterating with a shorter function:
first
second
third
Now freeing each item
```

Lots of good stuff in this example:

- A statement like `GSList x = g_slist_append(NULL, [whatever])` lets you declare, initialize, and add the first item to the list all in one fell swoop.
- The `g_strdup` function is handy for duplicating a string; just remember to free it once you're done with it.
- `g_slist_foreach` lets you pass in a pointer, so you can effectively give it any argument along with each item in the list. For example, you could pass in an accumulator and collect information about each item in a list. The only restriction on the iterating function is that it takes at least one `gpointer` as an argument; you can see how `print_interator_short` works even though it accepts only one argument.
- Note that the code frees all the strings using a built in GLib function as an argument to `g_slist_foreach`. All you had to do in this case was cast the `g_free` function to a `GFunc` for this to work. Note that you still have to free the GSList itself with a separate call to `g_slist_free`.

Sorting with GCompareFunc

You can sort a GSList by supplying a function that knows how to compare the items in that list. The following example shows one way to sort a list of strings:

```
//ex-gslist-9.c
#include <glib.h>
gint my_comparator(gconstpointer item1, gconstpointer item2) {
 return g_ascii_strcasecmp(item1, item2);
}
int main(int argc, char** argv) {
 GSList* list = g_slist_append(NULL, "Chicago");
 list = g_slist_append(list, "Boston");
 list = g_slist_append(list, "Albany");
 list = g_slist_sort(list, (GCompareFunc)my_comparator);
 printf("The first item is now '%s'\n", list->data);
 printf("The last item is now '%s'\n", g_slist_last(list)->data);
 g_slist_free(list);
 return 0;
}

***** Output *****

The first item is now 'Albany'
The last item is now 'Chicago'
```

Notice that the `GCompareFunc` returns a negative value if the first item is less than the second, 0 if they're equal, and a positive value if the second is greater than the first. As long as your comparison function conforms to this specification, it can do whatever it needs to internally.

Also, since various other GLib functions follow this pattern, it can be easy to delegate to them. In fact, in the example above, you can just as easily replace the call to `my_comparator` with something like `g_slist_sort(list, (GCompareFunc)g_ascii_strcasecmp)` and you'll get the same results.

Finding an element

There are several ways to find an element in a GSList. You've already seen how you can simply iterate over the contents of the list, comparing each item until you locate the target item. You can use `g_slist_find` if you already have the data you're looking for and just want to get to that location in the list. Finally, you can use `g_slist_find_custom`, which lets you use a function to check each item in the list. `g_slist_find` and `g_slist_find_custom` are illustrated below:

```
//ex-gslist-10.c
#include <glib.h>
gint my_finder(gconstpointer item) {
 return g_ascii_strcasecmp(item, "second");
}
int main(int argc, char** argv) {
 GSList* list = g_slist_append(NULL, "first");
 list = g_slist_append(list, "second");
 list = g_slist_append(list, "third");
 GSList* item = g_slist_find(list, "second");
 printf("This should be the 'second' item: '%s'\n", item->data);
 item = g_slist_find_custom(list, NULL, (GCompareFunc)my_finder);
 printf("Again, this should be the 'second' item: '%s'\n", item->data);
 item = g_slist_find(list, "delta");
 printf("'delta' is not in the list, so we get: '%s'\n", item ? item->data : "(null)");
 g_slist_free(list);
 return 0;
}

***** Output *****

This should be the 'second' item: 'second'
Again, this should be the 'second' item: 'second'
```

```
'delta' is not in the list, so we get: '(null)'
```

Note that `g_slist_find_custom` also takes a pointer to anything as the second argument, so if needed, you can pass in something to help the finder function. Also, the `GCompare` function is the last argument, rather than the second argument, since it is in `g_slist_sort`. Finally, a failing search returns NULL.

Advanced adding with insert

Now that you've seen the `GCompareFunc` a few times, some of the more interesting insertion operations will make more sense. Items can be inserted at a given position with `g_slist_insert`, before a specified item with `g_slist_insert_before`, and in a sorted order with `g_slist_insert_sorted`. Here's how it looks:

```
//ex-gslist-11.c
#include <glib.h>
int main(int argc, char** argv) {
 GSList* list = g_slist_append(NULL, "Anaheim "), *iterator = NULL;
 list = g_slist_append(list, "Elkton ");
 printf("Before inserting 'Boston', second item is: '%s'\n", g_slist_nth(list, 1)->data);
 g_slist_insert(list, "Boston ", 1);
 printf("After insertion, second item is: '%s'\n", g_slist_nth(list, 1)->data);
 list = g_slist_insert_before(list, g_slist_nth(list, 2), "Chicago ");
 printf("After an insert_before, third item is: '%s'\n", g_slist_nth(list, 2)->data);
 list = g_slist_insert_sorted(list, "Denver ", (GCompareFunc)g_ascii_strcasecmp);
 printf("After inserting 'Denver', here's the final list:\n");
 g_slist_foreach(list, (GFunc)printf, NULL);
 g_slist_free(list);
 return 0;
}

***** Output *****

Before inserting 'Boston', second item is: 'Elkton '
After insertion, second item is: 'Boston '
After an insert_before, third item is: 'Chicago '
After inserting 'Denver', here's the final list:
Anaheim Boston Chicago Denver Elkton
```

Since `g_slist_insert_sorted` takes a `GCompareFunc`, it's easy to reuse the built-in GLib function `g_ascii_strcasecmp`. And now you can see why there's an extra space at the end of each item; it's so another `g_slist_foreach` example could sneak in there at the end of the code sample, this time with `printf` as the `GFunc`.

Real-world usage of singly-linked lists

You can find lots of GSList usage in all three of the real-world open source applications mentioned earlier. Most of the usage is fairly pedestrian, with lots of inserts and appends and removes and so forth. But here's some of the more interesting stuff.

Gaim uses GSLists to hold the current conversations and for various things in most of the plug-ins:

- gaim-1.2.1/src/away.c uses a sorted GSList to hold the "away messages" (the messages that are received while the client is offline). It uses a custom `GCompareFunc`, `sort_awaymsg_list` to keep those messages sorted by the sender's name.

- gaim-1.2.1/src/protocols/gg/gg.c uses a GSList to hold a list of permitted accounts; it then uses a custom finder to verify that an account is in this list. The custom finder simply delegates to `g_ascii_strcasecmp`, so it's possible that it could be eliminated altogether and `g_ascii_strcasecmp` passed directly to `g_slist_find_custom`.

Evolution uses plenty of GSLists as well:

- evolution-data-server-1.0.2/calendar/libecal/e-cal-component.c uses a GSList to hold meeting attendees. In one case, it builds the GSList by repeatedly calling `g_slist_prepend` and finishing up with a `g_slist_reverse` to get the items in the desired order. As mentioned earlier, this is faster than adding items with `g_slist_append`.
- evolution-2.0.2/addressbook/gui/contact-editor/e-contact-editor.c uses `g_slist_find` in a sort of guard clause; it uses it in a signal handler to ensure that an `EContactEditor` it received in a signal callback is still around before passing it as an argument to a function.

The GIMP uses GSList in some nice ways too:

- gimp-2.2.4/plug-ins/maze/algorithms.c uses a GSList to track cells in the maze-generations algorithm.
- gimp-2.2.4/app/widgets/gimpclipboard.c uses a GSList to hold clipboard pixel-buffer formats (like PNG and JPEG); it passes a custom `GCompareFunc` to `g_slist_sort`.
- gimp-2.2.4/app/core/gimppreviewcache.c uses a GSList as a sort of size-based queue; it holds image previews in a GSList and uses `g_slist_insert_sorted` to insert the smaller images first. Another function in the same file trims the cache by iterating over the same GSList and comparing each item by surface area to find the smallest one to remove.

Doubly-linked lists

Concepts of doubly-linked lists

Doubly-linked lists are much like singly-linked lists, but they contain extra pointers to enable more navigation options; given a node in a doubly-linked list, you can either move forward or backward. This makes them more flexible then singly-linked lists, but it also increases memory usage, so don't use a doubly-linked list unless you're actually going to need this flexibility.

GLib contains a doubly-linked list implementation called a *GList*. Most of the operations in a GList are similar to those in a GSList. We'll review some examples of basic usages and then the added operations that a GList allows.

Basic operations of doubly-linked lists

Here are some of the common operations you can do with a GList:

```
//ex-glist-1.c
#include <glib.h>
int main(int argc, char** argv) {
 GList* list = NULL;
 list = g_list_append(list, "Austin ");
```

```
 printf("The first item is '%s'\n", list->data);
 list = g_list_insert(list, "Baltimore ", 1);
 printf("The second item is '%s'\n", g_list_next(list)->data);
 list = g_list_remove(list, "Baltimore ");
 printf("After removal of 'Baltimore', the list length is %d\n", g_list_length(list));
 GList* other_list = g_list_append(NULL, "Baltimore ");
 list = g_list_concat(list, other_list);
 printf("After concatenation: ");
 g_list_foreach(list, (GFunc)printf, NULL);
 list = g_list_reverse(list);
 printf("\nAfter reversal: ");
 g_list_foreach(list, (GFunc)printf, NULL);
 g_list_free(list);
 return 0;
}

***** Output *****

The first item is 'Austin '
The second item is 'Baltimore '
After removal of 'Baltimore', the list length is 1
After concatenation: Austin Baltimore
After reversal: Baltimore Austin
```

The above code probably looks pretty familiar! All of the above operations are also present in GSList; the only difference for GList is that the function names start with `g_list` rather than `g_slist`. And, of course, they all take a pointer to a GList structure rather than a pointer to a GSList structure.

Better navigation

Now that you've seen some basic GList operations, here are some operations that are possible only because each node in a GList has a link to the previous node:

```
//ex-glist-2.c
#include <glib.h>
int main(int argc, char** argv) {
 GList* list = g_list_append(NULL, "Austin ");
 list = g_list_append(list, "Bowie ");
 list = g_list_append(list, "Charleston ");
 printf("Here's the list: ");
 g_list_foreach(list, (GFunc)printf, NULL);
 GList* last = g_list_last(list);
 printf("\nThe first item (using g_list_first) is '%s'\n", g_list_first(last)->data);
 printf("The next-to-last item is '%s'\n", g_list_previous(last)->data);
 printf("The next-to-last item is '%s'\n", g_list_nth_prev(last, 1)->data);
 g_list_free(list);
 return 0;
}

***** Output *****

Here's the list: Austin Bowie Charleston
The first item (using g_list_first) is 'Austin '
The next-to-last item is 'Bowie '
The next-to-last item is 'Bowie '
```

Nothing too surprising, but a few notes:

- The second argument to `g_list_nth_prev` is an integer indicating how far back you want to navigate. If you pass in a value that goes outside the limits of the GList, prepare for a crash.

- `g_list_first` is an O(n) operation; it starts at the specified node and searches backward until it finds the beginning of the GList. The example above is the worst-case scenario because the traversal starts at the end of the list. `g_list_last` is O(n) for the same reason.

Removing nodes using links

You've already seen how you can remove a node from the list if you have a pointer to the data it contains; `g_list_remove` does that nicely. If you have a pointer to the node itself, you can remove that node directly in a quick O(1) operation:

```
//ex-glist-3.c
#include <glib.h>
int main(int argc, char** argv) {
 GList* list = g_list_append(NULL, "Austin ");
 list = g_list_append(list, "Bowie ");
 list = g_list_append(list, "Chicago ");
 printf("Here's the list: ");
 g_list_foreach(list, (GFunc)printf, NULL);
 GList* bowie = g_list_nth(list, 1);
 list = g_list_remove_link(list, bowie);
 g_list_free_1(bowie);
 printf("\nHere's the list after the remove_link call: ");
 g_list_foreach(list, (GFunc)printf, NULL);
 list = g_list_delete_link(list, g_list_nth(list, 1));
 printf("\nHere's the list after the delete_link call: ");
 g_list_foreach(list, (GFunc)printf, NULL);
 g_list_free(list);
 return 0;
 }

***** Output *****

Here's the list: Austin Bowie Chicago
Here's the list after the remove_link call: Austin Chicago
Here's the list after the delete_link call: Austin
```

So if you have a pointer to a node instead of to a node's data, you can remove that node using `g_list_remove_link`.

After removing it, you'll need to explicitly free it using `g_list_free_1`, which does just what its name implies: it frees one node. As usual, you need to hang on to the return value of `g_list_remove_link` since that's the new beginning of the list.

Finally, if all you want to do is remove a node and free it, you can do that in one step with a call to `g_list_delete_link`.

The same functions exist for the GSList as well; just replace `g_list` with `g_slist` and all the above information applies.

Indexes and positions

If you just want to find the position of an item in a GList, you have two options. You can use `g_list_index`, which looks up an item using the data in it, or you can use `g_list_position`, which uses the pointer to the node. This example illustrates both:

```
//ex-glist-4.c
```

```
#include <glib.h>
int main(int argc, char** argv) {
 GList* list = g_list_append(NULL, "Austin ");
 list = g_list_append(list, "Bowie ");
 list = g_list_append(list, "Bowie ");
 list = g_list_append(list, "Cheyenne ");
 printf("Here's the list: ");
 g_list_foreach(list, (GFunc)printf, NULL);
 printf("\nItem 'Bowie' is located at index %d\n", g_list_index(list, "Bowie "));
 printf("Item 'Dallas' is located at index %d\n", g_list_index(list, "Dallas"));
 GList* last = g_list_last(list);
 printf("Item 'Cheyenne' is located at index %d\n", g_list_position(list, last));
 g_list_free(list);
 return 0;
}

***** Output *****

Here's the list: Austin Bowie Bowie Cheyenne
Item 'Bowie' is located at index 1
Item 'Dallas' is located at index -1
Item 'Cheyenne' is located at index 3
```

Note that `g_list_index` returns a value of `-1` if it can't find the data. And if there are two nodes with the same data value, `g_list_index` returns the index of the first occurrence. `g_list_position` also returns a `-1` if it can't find the specified node.

Again, these methods are also present on GSList under different names.

Real-world usage of doubly-linked lists

Let's look at the GList usage in the previously mentioned open source applications.

Gaim uses plenty of GLists:

- gaim-1.2.1/plugins/gevolution/add_buddy_dialog.c uses a call to `g_list_foreach` to free up references to each contact when closing the "add a buddy" dialog box.
- gaim-1.2.1/src/account.c uses a GList to hold all the accounts; it uses `g_list_find` to ensure an account is not already present before adding it with `g_list_append`.

Evolution GList usage:

- evolution-2.0.2/filter/filter-rule.c uses a GList to hold the parts (such as a subject line to check) of a mail-filtering rule; `filter_rule_finalise` uses `g_list_foreach` to release references to those parts.
- evolution-2.0.2/calendar/gui/alarm-notify/alarm.c uses a GList to hold the alarms; `queue_alarm` uses `g_list_insert_sorted` to insert new alarms in the correct place using a custom `GCompareFunc`.

The GIMP usage:

- gimp-2.2.4/app/file/gimprecentlist.c uses a GList to hold the recently accessed files; `gimp_recent_list_read` reads the file names in from an XML file descriptor and calls `g_list_reverse` before returning the GList.

- gimp-2.2.4/app/vectors/gimpbezierstroke.c uses GLists to hold stroke anchors; `gimp_bezier_stroke_connect_stroke` uses `g_list_concat` to help connect one stroke to another.

Hash tables

Concepts of hash tables

So far this tutorial has covered only ordered containers in which items inserted in the container in a certain order stayed that way. Another type of container is a *hash table*, also known as a "map," an "associative array," or a "dictionary."

Just as a language dictionary associates a word with a definition, hash tables use a *key* to uniquely identify a *value*. Hash tables can perform insertion, lookup, and remove operations on a key very quickly; in fact, with proper usage, these can all be constant time -- that is, O(1) -- operations. That's much better than looking up or removing an item from an ordered list, an O(n) operation.

Hash tables perform operations quickly because they use a *hash function* to locate keys. A hash function takes a key and calculates a unique value, called a *hash*, for that key. For example, a hash function could accept a word and return the number of letters in that word as the hash. That would be a bad hash function because both "fiddle" and "faddle" would hash to the same value.

When a hash function returns the same hash for two different keys, various things can happen depending on the hash table implementation. The hash table can overwrite the first value with the second value, it can put the values into a list, or it can simply throw an error.

Note that hash tables aren't necessarily faster than lists. If you have a small number of items -- less than a dozen or so -- you may get better performance by using an ordered collection. That's because even though storing and retrieving data in a hash table takes constant time, that constant time value may be large since computing the hash of an item can be a slow process compared to dereferencing a pointer or two. For small values, simply iterating over an ordered container can be faster than doing the hash computations.

As always, it's important to think about your own application's specific data-storage needs when choosing a container. And there's no reason why you can't switch containers down the road if it becomes clear that your application needs it.

Some basic hash table operations

Here are some examples to put some wheels on the previous theory:

```
//ex-ghashtable-1.c
#include <glib.h>
int main(int argc, char** argv) {
 GHashTable* hash = g_hash_table_new(g_str_hash, g_str_equal);
 g_hash_table_insert(hash, "Virginia", "Richmond");
 g_hash_table_insert(hash, "Texas", "Austin");
 g_hash_table_insert(hash, "Ohio", "Columbus");
 printf("There are %d keys in the hash\n", g_hash_table_size(hash));
 printf("The capital of Texas is %s\n", g_hash_table_lookup(hash, "Texas"));
 gboolean found = g_hash_table_remove(hash, "Virginia");
 printf("The value 'Virginia' was %sfound and removed\n", found ? "" : "not ");
 g_hash_table_destroy(hash);
 return 0;
}

***** Output *****

There are 3 keys in the hash
The capital of Texas is Austin
The value 'Virginia' was found and removed
```

Lots of new stuff there, so some notes:

- The call to `g_hash_table_new` specifies that this hash table will be using strings as the keys. The functions `g_str_hash` and `g_str_equal` are built into GLib since this is a common use case. Other built-in hash/equality functions are `g_int_hash` /`g_int_equal` for using integers as the keys, and `g_direct_hash`/ `g_direct_equal` for using pointers as keys.
- GLists and GSLists have a `g_[container]_free` function to clean them up; a GHashTable can be cleaned up with `g_hash_table_destroy`.
- When you attempt to remove a key/value pair using `g_hash_table_remove`, you get a `gboolean` value in return, indicating whether the key was found and removed. The `gboolean` is a simple cross-platform GLib implementation of a true/false value.
- `g_hash_table_size` returns the number of keys in the hash table.

Inserting and replacing values

When you insert a key using `g_hash_table_insert`, GHashTable will first check to see if that key already exists. If it does, the value will be replaced but not the key. If you want to replace both the key and the value, you need to use `g_hash_table_replace`. It's a subtle difference, so both are illustrated below:

```
//ex-ghashtable-2.c
#include <glib.h>
static char* texas_1, *texas_2;
void key_destroyed(gpointer data) {
 printf("Got a key destroy call for %s\n", data == texas_1 ? "texas_1" : "texas_2");
}
int main(int argc, char** argv) {
 GHashTable* hash = g_hash_table_new_full(g_str_hash, g_str_equal,
  (GDestroyNotify)key_destroyed, NULL);
 texas_1 = g_strdup("Texas");
 texas_2 = g_strdup("Texas");
 g_hash_table_insert(hash, texas_1, "Austin");
 printf("Calling insert with the texas_2 key\n");
 g_hash_table_insert(hash, texas_2, "Houston");
 printf("Calling replace with the texas_2 key\n");
 g_hash_table_replace(hash, texas_2, "Houston");
 printf("Destroying hash, so goodbye texas_2\n");
```

```
 g_hash_table_destroy(hash);
 g_free(texas_1);
 g_free(texas_2);
 return 0;
}

***** Output *****

Calling insert with the texas_2 key
Got a key destroy call for texas_2
Calling replace with the texas_2 key
Got a key destroy call for texas_1
Destroying hash, so goodbye texas_2
Got a key destroy call for texas_2
```

You can see from the output that when `g_hash_table_insert` tried to insert the same string (`Texas`) as an existing key, the GHashTable simply freed the passed-in key (`texas_2`) and left the current key (`texas_1`) in place. But when `g_hash_table_replace` did the same thing, the `texas_1` key was destroyed and the `texas_2` key was used in its place. A few more notes:

- When you create a new GHashTable, you can use `g_hash_table_full` to supply a `GDestroyNotify` implementation to be called when a key is destroyed. This lets you do any resource cleanups for that key or in this case, see what's really happening with a key change.
- You've seen `g_strdup` back in the GSList section; here it's used to allocate two copies of the string `Texas`. You can see that the GHashTable functions `g_str_hash` and `g_str_equal` correctly detected that the actual strings were equivalent even though the variables were pointers to different memory locations. And you had to free the `texas_1` and `texas_2` at the end of the function to avoid leaking memory. Of course, in this case it wouldn't have mattered since the program was exiting, but it's best to be tidy anyhow.

Iterating the key/value pairs

Sometimes you need to iterate over all the key/value pairs. Here's how to do that using `g_hash_table_foreach`:

```
//ex-ghashtable-3.c
#include <glib.h>
void iterator(gpointer key, gpointer value, gpointer user_data) {
 printf(user_data, *(gint*)key, value);
}
int main(int argc, char** argv) {
 GHashTable* hash = g_hash_table_new(g_int_hash, g_int_equal);
 gint* k_one = g_new(gint, 1), *k_two = g_new(gint, 1), *k_three = g_new(gint, 1);
 *k_one = 1, *k_two=2, *k_three = 3;
 g_hash_table_insert(hash, k_one, "one");
 g_hash_table_insert(hash, k_two, "four");
 g_hash_table_insert(hash, k_three, "nine");
 g_hash_table_foreach(hash, (GHFunc)iterator, "The square of %d is %s\n");
 g_hash_table_destroy(hash);
 return 0;
}

***** Output *****

The square of 1 is one
The square of 2 is four
The square of 3 is nine
```

There are a few little twists in this example:

- You can see that using the GLib-provided hashing functions `g_int_hash` and `g_int_equal` lets you use pointers to integers as the keys. And this example uses the GLib cross-platform abstraction for integers: the `gint`.
- `g_hash_table_foreach` is a lot like the `g_slist_foreach` and `g_list_foreach` functions that you've seen already. The only difference is that the `GHFunc` passed to `g_hash_table_foreach` accepts three arguments rather than two. In this case, you passed in a string to be used to format the printing of the key and the value as the third argument. Also, while the keys happen to have been listed in the order they were inserted in this example, there's no guarantee that key-insertion order will be preserved.

Finding an item

To find a specific value, use the `g_hash_table_find` function. This function lets you look at each key/value pair until you locate the one you want. Here's an example:

```
//ex-ghashtable-4.c
#include <glib.h>
void value_destroyed(gpointer data) {
 printf("Got a value destroy call for %d\n", GPOINTER_TO_INT(data));
}
gboolean finder(gpointer key, gpointer value, gpointer user_data) {
 return (GPOINTER_TO_INT(key) + GPOINTER_TO_INT(value)) == 42;
}
int main(int argc, char** argv) {
 GHashTable* hash = g_hash_table_new_full(g_direct_hash, g_direct_equal,
  NULL,
  (GDestroyNotify)value_destroyed);
 g_hash_table_insert(hash, GINT_TO_POINTER(6), GINT_TO_POINTER(36));
 g_hash_table_insert(hash, GINT_TO_POINTER(10), GINT_TO_POINTER(12));
 g_hash_table_insert(hash, GINT_TO_POINTER(20), GINT_TO_POINTER(22));
 gpointer item_ptr = g_hash_table_find(hash, (GHRFunc)finder, NULL);
 gint item = GPOINTER_TO_INT(item_ptr);
 printf("%d + %d == 42\n", item, 42-item);
 g_hash_table_destroy(hash);
 return 0;
}

***** Output *****

36 + 6 == 42
Got a value destroy call for 36
Got a value destroy call for 22
Got a value destroy call for 12
```

As usual, this example introduces `g_hash_table_find` and a few other things as well:

- `g_hash_table_find` returns the first value for which `GHRFunc` returns TRUE. If the `GHRFunc` doesn't return TRUE for any item (such that no suitable item is found), it returns NULL.
- This example demonstrates another set of built-in GLib hashing functions: `g_direct_hash` and `g_direct_equal`. This set of functions lets you use pointers as keys but makes no attempt at interpreting the data behind the pointers. And since you're putting pointers in the GHashTable, you need to use some of the GLib convenience macros (`GINT_TO_POINTER` and `GPOINTER_TO_INT`) to convert the integers to and from pointers.

- Finally, this example creates the GHashTable and gives it a `GDestroyNotify` callback function so you can see when the items are destroyed. Most of the time you'll want to free some memory in a function like this, but for example purposes, this implementation just prints out a message.

Tricky business: Stealing from the table

Occasionally you may need to remove an item from a GHashTable without getting a callback to any `GDestroyNotify` functions the GHashTable has been given. You can do this either on a specific key using `g_hash_table_steal` or for all the keys that match a criteria using `g_hash_table_foreach_steal`.

```
//ex-ghashtable-5.c
#include <glib.h>
gboolean wide_open(gpointer key, gpointer value, gpointer user_data) {
 return TRUE;
}
void key_destroyed(gpointer data) {
 printf("Got a GDestroyNotify callback\n");
}
int main(int argc, char** argv) {
 GHashTable* hash = g_hash_table_new_full(g_str_hash, g_str_equal,
  (GDestroyNotify)key_destroyed,
  (GDestroyNotify)key_destroyed);
 g_hash_table_insert(hash, "Texas", "Austin");
 g_hash_table_insert(hash, "Virginia", "Richmond");
 g_hash_table_insert(hash, "Ohio", "Columbus");
 g_hash_table_insert(hash, "Oregon", "Salem");
 g_hash_table_insert(hash, "New York", "Albany");
 printf("Removing New York, you should see two callbacks\n");
 g_hash_table_remove(hash, "New York");
 if (g_hash_table_steal(hash, "Texas")) {
  printf("Texas has been stolen, %d items remaining\n", g_hash_table_size(hash));
 }
 printf("Stealing remaining items\n");
 g_hash_table_foreach_steal(hash, (GHRFunc)wide_open, NULL);
 printf("Destroying the GHashTable, but it's empty, so no callbacks\n");
 g_hash_table_destroy(hash);
 return 0;
}

***** Output *****

Removing New York, you should see two callbacks
Got a GDestroyNotify callback
Got a GDestroyNotify callback
Texas has been stolen, 3 items remaining
Stealing remaining items
Destroying the GHashTable, but it's empty, so no callbacks
```

Advanced lookups: Finding both a key and a value

GHashTable provides a `g_hash_table_lookup_extended` function for those cases when you need to fetch both a key and its value from a table. It's a lot like `g_hash_table_lookup`, but it accepts two more arguments. These are "out" arguments; that is, they are doubly-indirect pointers that will be pointed to the data when it's located. Here's how they work:

```
//ex-ghashtable-6.c
```

```
#include <glib.h>
int main(int argc, char** argv) {
 GHashTable* hash = g_hash_table_new(g_str_hash, g_str_equal);
 g_hash_table_insert(hash, "Texas", "Austin");
 g_hash_table_insert(hash, "Virginia", "Richmond");
 g_hash_table_insert(hash, "Ohio", "Columbus");
 char* state = NULL;
 char* capital = NULL;
 char** key_ptr = &state;
 char** value_ptr = &capital;
 gboolean result = g_hash_table_lookup_extended(hash, "Ohio",
    (gpointer*)key_ptr, (gpointer*)value_ptr);
 if (result) {
  printf("Found that the capital of %s is %s\n", capital, state);
 }
 if (!g_hash_table_lookup_extended(hash, "Vermont",
    (gpointer*)key_ptr, (gpointer*)value_ptr)) {
  printf("Couldn't find Vermont in the hash table\n");
 }
 g_hash_table_destroy(hash);
 return 0;
}

***** Output *****

Found that the capital of Columbus is Ohio
Couldn't find Vermont in the hash table
```

The code to initialize the variable that will receive the key/value data is a little complicated, but thinking of it as a way of returning more than one value from the function may make it more understandable. Note that if you pass in NULL for either the last two arguments, `g_hash_table_lookup_extended` will still work, it just won't fill in the NULL arguments.

Multiple values for each key

So far you've seen hashes that have only a single value for each key. But sometimes you'll need to hold multiple values for a key. When this need arises, using a GSList as the value and appending new values to that GSList is often a good solution. It does take a bit more work, though, as you can see in this example:

```
//ex-ghashtable-7.c
#include <glib.h>
void print(gpointer key, gpointer value, gpointer data) {
 printf("Here are some cities in %s: ", key);
 g_slist_foreach((GSList*)value, (GFunc)printf, NULL);
 printf("\n");
}
void destroy(gpointer key, gpointer value, gpointer data) {
 printf("Freeing a GSList, first item is %s\n", ((GSList*)value)->data);
 g_slist_free(value);
}
int main(int argc, char** argv) {
 GHashTable* hash = g_hash_table_new(g_str_hash, g_str_equal);
 g_hash_table_insert(hash, "Texas",
  g_slist_append(g_hash_table_lookup(hash, "Texas"), "Austin "));
 g_hash_table_insert(hash, "Texas",
  g_slist_append(g_hash_table_lookup(hash, "Texas"), "Houston "));
 g_hash_table_insert(hash, "Virginia",
  g_slist_append(g_hash_table_lookup(hash, "Virginia"), "Richmond "));
 g_hash_table_insert(hash, "Virginia",
  g_slist_append(g_hash_table_lookup(hash, "Virginia"), "Keysville "));
 g_hash_table_foreach(hash, print, NULL);
```

```
 g_hash_table_foreach(hash, destroy, NULL);
 g_hash_table_destroy(hash);
 return 0;
}

***** Output *****

Here are some cities in Texas: Austin Houston
Here are some cities in Virginia: Richmond Keysville
Freeing a GSList, first item is Austin
Freeing a GSList, first item is Richmond
```

The "insert a new city" code in the example above takes advantage of the fact that `g_slist_append` accepts NULL as a valid argument for the GSList; it doesn't need to check if this is the first city being added to the list for a given state.

When the GHashTable is destroyed, you have to remember to free all those GSLists before freeing the hash table itself. Note that this would be a bit more convoluted if you weren't using static strings in those lists; in that case you'd need to free each item in each GSList before freeing the list itself. One thing this example does show is how useful the various `foreach` functions can be -- they can save a fair bit of typing.

Real-world usage of hash tables

Here's a sampling of how GHashTables are being used.

In Gaim:

- gaim-1.2.1/src/buddyicon.c uses a GHashTable to keep track of "buddy icons." The key is the buddy's user name, and the value is a pointer to a `GaimBuddyIcon` structure.
- gaim-1.2.1/src/protocols/yahoo/yahoo.c is the only location in these three applications to use `g_hash_table_steal`. It uses `g_hash_table_steal` as part of a section of code that builds an account name to buddy list mapping.

In Evolution:

- evolution-2.0.2/smime/gui/certificate-manager.c uses a GHashTable to track S/MIME certificate roots; the key is the organization name, and the value is a pointer to a `GtkTreeIter`.
- evolution-data-server-1.0.2/calendar/libecal/e-cal.c uses a GHashTable to track time zones; the key is a time zone ID string, and the value is a string representation of an `icaltimezone` structure.

In GIMP:

- gimp-2.2.4/libgimp/gimp.c uses a GHashTable to track temporary procedures. In the only usage of `g_hash_table_lookup_extended` in the entire GIMP codebase, it uses a call to `g_hash_table_lookup_extended` to find a procedure so that it can first free the hash key's memory before removing the procedure.
- gimp-2.2.4/app/core/gimp.c uses a GHashTable to hold images; the key is a image ID (an integer), and the value is a pointer to a `GimpImage` structure.

Arrays

Concepts of arrays

So far we've covered two types of ordered collections: GSList and GList. These are rather similar in that they depend on pointers to link from one element to the next item, or in the case of the GList, to the previous item. But there's another type of ordered collection that doesn't use links; instead it works more or less like a C array.

It's called the *GArray* and it provides an indexed ordered collection of a single type that grows as necessary to accommodate new items.

What's the advantage of an array over a linked list? For one thing, indexed access. That is, if you want to get the fifth element in the array, you can simply call a function to retrieve that item in constant time; there's no need to iterate up to that point manually, which would be an O(n) operation. An array knows its own size, so to query the size is an O(1) operation rather than O(n) operations.

Basic operations of arrays

Here are some of the primary ways to get data in and out of a GArray:

```
//ex-garray-1.c
#include <glib.h>
int main(int argc, char** argv) {
 GArray* a = g_array_new(FALSE, FALSE, sizeof(char*));
 char* first = "hello", *second = "there", *third = "world";
 g_array_append_val(a, first);
 g_array_append_val(a, second);
 g_array_append_val(a, third);
 printf("There are now %d items in the array\n", a->len);
 printf("The first item is '%s'\n", g_array_index(a, char*, 0));
 printf("The third item is '%s'\n", g_array_index(a, char*, 2));
 g_array_remove_index(a, 1);
 printf("There are now %d items in the array\n", a->len);
 g_array_free(a, FALSE);
 return 0;
}

***** Output *****

There are now 3 items in the array
The first item is 'hello'
The third item is 'world'
There are now 2 items in the array
```

Some points to ponder:

- There are several options to consider when creating a GArray. In the example above, the first two arguments to `g_array_new` indicate whether the array should have a zero element as a terminator and whether new elements in the array should be automatically set to zero. The third argument tells the array which type it is going to hold. In this case the array is created to hold the type `char*`; putting anything else in the array would lead to segfaults.
- `g_array_append_val` is a macro designed so that it does not accept literal values, so you can't call `g_array_append_val(a, 42)`. Instead, the value needs to be placed in a variable

and that variable passed in to `g_array_append_val`. As a consolation for the inconvenience, `g_array_append_val` is very fast.
- A GArray is a structure with a member variable `len`, so to get the size of the array, you can just reference that variable directly; no need for a function call.
- A GArray grows in powers of two. That is, if a GArray contains four items and you add another, it will internally create another eight-element GArray, copy the four existing elements into it, and then add your new element. This resizing process takes time, so if you know you're going to have a certain number of elements, it's more efficient to create the GArray pre-allocated to the desired size.

More new/free options

In this example you'll see a few different ways to create and destroy a GArray:

```
//ex-garray-2.c
#include <glib.h>
int main(int argc, char** argv) {
 GArray* a = g_array_sized_new(TRUE, TRUE, sizeof(int), 16);
 printf("Array preallocation is hidden, so array size == %d\n", a->len);
 printf("Array was init'd to zeros, so 3rd item is = %d\n", g_array_index(a, int, 2));
 g_array_free(a, FALSE);

 // this creates an empty array, then resizes it to 16 elements
 a = g_array_new(FALSE, FALSE, sizeof(char));
 g_array_set_size(a, 16);
 g_array_free(a, FALSE);

 a = g_array_new(FALSE, FALSE, sizeof(char));
 char* x = g_strdup("hello world");
 g_array_append_val(a, x);
 g_array_free(a, TRUE);

 return 0;
}

***** Output *****

Array preallocation is hidden, so array size == 0
Array was init'd to zeros, so 3rd item is = 0
```

Note that since GArrays grow by powers of two, it's inefficient to size an array to something close to a power of two, like fourteen. Instead, just go ahead and bump it up to the closest power of two.

More ways to add data

Thus far you've seen data added to the array with `g_array_append_val`. But there are other ways to get data into an array, as shown below:

```
//ex-garray-3.c
#include <glib.h>
void prt(GArray* a) {
 printf("Array holds: ");
 int i;
 for (i = 0; i < a->len; i++)
  printf("%d ", g_array_index(a, int, i));
 printf("\n");
}
int main(int argc, char** argv) {
```

```
 GArray* a = g_array_new(FALSE, FALSE, sizeof(int));
 printf("Array is empty, so appending some values\n");
 int x[2] = {4,5};
 g_array_append_vals(a, &x, 2);
 prt(a);
 printf("Now to prepend some values\n");
 int y[2] = {2,3};
 g_array_prepend_vals(a, &y, 2);
 prt(a);
 printf("And one more prepend\n");
 int z = 1;
 g_array_prepend_val(a, z);
 prt(a);
 g_array_free(a, FALSE);
 return 0;
}

***** Output *****

Array is empty, so appending some values
Array holds: 4 5
Now to prepend some values
Array holds: 2 3 4 5
And one more prepend
Array holds: 1 2 3 4 5
```

So you can append multiple values to an array, you can prepend one value, and you can prepend multiple values. Be careful with prepending values, though; it's an O(n) operation since the GArray has to push all the current values down to make room for the new data. You still need to use variables when appending or prepending multiple values, but it's fairly straightforward since you can append or prepend an entire array.

Inserting data

You can also insert data into an array in various places; you're not limited to simply appending or prepending items. Here's how it works:

```
//ex-garray-4.c
#include <glib.h>
void prt(GArray* a) {
 printf("Array holds: ");
 int i;
 for (i = 0; i < a->len; i++)
  printf("%d ", g_array_index(a, int, i));
 printf("\n");
}
int main(int argc, char** argv) {
 GArray* a = g_array_new(FALSE, FALSE, sizeof(int));
 int x[2] = {1,5};
 g_array_append_vals(a, &x, 2);
 prt(a);
 printf("Inserting a '2'\n");
 int b = 2;
 g_array_insert_val(a, 1, b);
 prt(a);
 printf("Inserting multiple values\n");
 int y[2] = {3,4};
 g_array_insert_vals(a, 2, y, 2);
 prt(a);
 g_array_free(a, FALSE);
 return 0;
}
```

```
***** Output *****

Array holds: 1 5
Inserting a '2'
Array holds: 1 2 5
Inserting multiple values
Array holds: 1 2 3 4 5
```

Note that these `insert` functions involve copying the current elements in the list down to accommodate the new items, so using `g_array_insert_vals` is much better than using `g_array_insert_val` repeatedly.

Removing data

There are three ways to remove data from a GArray:

- `g_array_remove_index` and `g_array_remove_range`, both of which preserve the existing order
- `g_array_remove_index_fast`, which doesn't preserve the existing order

Here are examples of all three:

```
//ex-garray-5.c
#include <glib.h>
void prt(GArray* a) {
 int i;
 printf("Array holds: ");
 for (i = 0; i < a->len; i++)
  printf("%d ", g_array_index(a, int, i));
 printf("\n");
}
int main(int argc, char** argv) {
 GArray* a = g_array_new(FALSE, FALSE, sizeof(int));
 int x[6] = {1,2,3,4,5,6};
 g_array_append_vals(a, &x, 6);
 prt(a);
 printf("Removing the first item\n");
 g_array_remove_index(a, 0);
 prt(a);
 printf("Removing the first two items\n");
 g_array_remove_range(a, 0, 2);
 prt(a);
 printf("Removing the first item very quickly\n");
 g_array_remove_index_fast(a, 0);
 prt(a);
 g_array_free(a, FALSE);
 return 0;
}

***** Output *****

Array holds: 1 2 3 4 5 6
Removing the first item
Array holds: 2 3 4 5 6
Removing the first two items
Array holds: 4 5 6
Removing the first item very quickly
Array holds: 6 5
```

If you're wondering about a usage scenario for `g_array_remove_fast`, you're not alone; none of the three open source applications use this function.

Sorting arrays

Sorting a GArray is straightforward; it uses the `GCompareFunc`, which you already seen at work in the GList and GSList section:

```
//ex-garray-6.c
#include <glib.h>
void prt(GArray* a) {
 int i;
 printf("Array holds: ");
 for (i = 0; i < a->len; i++)
  printf("%d ", g_array_index(a, int, i));
 printf("\n");
}
int compare_ints(gpointer a, gpointer b) {
 int* x = (int*)a;
 int* y = (int*)b;
 return *x - *y;
}
int main(int argc, char** argv) {
 GArray* a = g_array_new(FALSE, FALSE, sizeof(int));
 int x[6] = {2,1,6,5,4,3};
 g_array_append_vals(a, &x, 6);
 prt(a);
 printf("Sorting\n");
 g_array_sort(a, (GCompareFunc)compare_ints);
 prt(a);
 g_array_free(a, FALSE);
 return 0;
}

***** Output *****

Array holds: 2 1 6 5 4 3
Sorting
Array holds: 1 2 3 4 5 6
```

Note that the comparing function gets a pointer to the data items, so in this case you need to cast them to a pointer to the correct type and then dereference that pointer to get to the actual data item. GArray also includes an alternate sorting function, `g_array_sort_with_data`, that accepts a pointer to an additional piece of data.

Incidentally, none of the three sample applications use either `g_array_sort` or `g_array_sort_with_data`. But as always, it's good to know that they're available.

Pointer arrays

GLib also provides `GPtrArray`, an array designed specifically to hold pointers. This can be a bit easier to use than the basic `GArray` since you don't need to specify a particular type when creating it or adding and indexing elements. It looks very much like GArray, so we'll just review some examples of the basic operations:

```
//ex-garray-7.c
#include <glib.h>
#include <stdio.h>
int main(int argc, char** argv) {
 GPtrArray* a = g_ptr_array_new();
 g_ptr_array_add(a, g_strdup("hello "));
```

```
 g_ptr_array_add(a, g_strdup("again "));
 g_ptr_array_add(a, g_strdup("there "));
 g_ptr_array_add(a, g_strdup("world "));
 g_ptr_array_add(a, g_strdup("\n"));
 printf(">Here are the GPtrArray contents\n");
 g_ptr_array_foreach(a, (GFunc)printf, NULL);
 printf(">Removing the third item\n");
 g_ptr_array_remove_index(a, 2);
 g_ptr_array_foreach(a, (GFunc)printf, NULL);
 printf(">Removing the second and third item\n");
 g_ptr_array_remove_range(a, 1, 2);
 g_ptr_array_foreach(a, (GFunc)printf, NULL);
 printf("The first item is '%s'\n", g_ptr_array_index(a, 0));
 g_ptr_array_free(a, TRUE);
 return 0;
}

***** Output *****

>Here are the GPtrArray contents
hello again there world
>Removing the third item
hello again world
>Removing the second and third item
hello
The first item is 'hello '
```

You can see how using a pointer-only array makes for a more straightforward API. This may explain why in Evolution, `g_ptr_array_new` is used 178 times, whereas `g_array_new` is only used 45 times. Most of the time a pointer-only container is good enough!

Byte arrays

Another type-specific array provided by GLib is the `GByteArray`. It's mostly like the types you've already seen, but there are a few wrinkles since it's designed for storing binary data. It's very handy for reading binary data in a loop since it hides the "read into a buffer-resize buffer-read some more" cycle. Here's some example code:

```
//ex-garray-8.c
#include <glib.h>
int main(int argc, char** argv) {
 GByteArray* a = g_byte_array_new();
 guint8 x = 0xFF;
 g_byte_array_append(a, &x, sizeof(x));
 printf("The first byte value (in decimal) is %d\n", a->data[0]);
 x = 0x01;
 g_byte_array_prepend(a, &x, sizeof(x));
 printf("After prepending, the first value is %d\n", a->data[0]);
 g_byte_array_remove_index(a, 0);
 printf("After removal, the first value is again %d\n", a->data[0]);
 g_byte_array_append(g_byte_array_append(a, &x, sizeof(x)), &x, sizeof(x));
 printf("After two appends, array length is %d\n", a->len);
 g_byte_array_free(a, TRUE);
 return 0;
}

***** Output *****

The first byte value (in decimal) is 255
After prepending, the first value is 1
After removal, the first value is again 255
After two appends, array length is 3
```

You're also seeing a new GLib type used here: `guint8`. This is a cross-platform 8-bit unsigned integer that is helpful for representing bytes accurately in this example.

Also, here you can see how `g_byte_array_append` returns the GByteArray. So if you want to nest a couple of appends similar to the way you might do method chaining, this makes it possible. Doing more than two or three of those is probably not a good idea, though, unless you want your code to start looking LISP-ish.

Real-world usage of arrays

The various GLib array types are used in the sample applications, although not as widely as the other containers you've seen.

Gaim uses only GPtrArrays and only in one or two cases. gaim-1.2.1/src/gtkpounce.c uses a GPtrArray to keep track of several GUI widgets that can be triggered when various events (like a buddy logging in) occur.

Evolution uses mostly GPtrArrays, although a number of GArrays and GByteArrays appears as well:

- evolution-2.0.2/widgets/misc/e-filter-bar.h keeps several types of search filters in GPtrArrays.
- evolution-2.0.2/camel/providers/imap4/camel-imap4-store.c uses a GPtrArray to track items in an IMAP folder; it uses g_ptr_array_sort with a GCompareFunc that delegates to `strcmp`.

The GIMP uses a fair number of GArrays and only a very few GPtrArrays and GByteArrays:

- gimp-2.2.4/app/tools/gimptransformtool.c uses a GArray to track a list of `GimpCoord` instances.
- gimp-2.2.4/app/base/boundary.c fills a GArray with points as part of a nifty `simplify_subdivide` function; a doubly-indirect pointer to the GArray gets recursively passed around as part of a boundary simplification routine.

Trees

Concepts of trees

Another useful container is called a *tree*. A tree consists of a root node that can have children, each of which can have more children, and so forth.

An example of a tree structure is a filesystem or perhaps an email client; it has folders that have folders that can contain more folders. Also, remember the end of the hash table section where you saw an example of multivalued keys? (For example, a String for a key and a GList for the value.) Since those GList values could have contained more GHashTables, that was an example of a tree structure trapped inside a GHashTable. It's a lot simpler to just use GTree rather than fighting another container to make it act like a tree.

GLib includes two tree containers: GTree, a *balanced binary tree* implementation, and GNode, a *n-ary* tree implementation.

A binary tree has a special property that each node of the tree has no more than two children; a *balanced binary tree* means that the elements are kept in a specified order for faster searching. Keeping the elements balanced means that removal and insertion can be slow since the tree may need to internally rebalance itself, but finding an item is a O(log n) operation.

A *n-ary tree* node, by contrast, can have many children. This tutorial focuses mostly on binary trees, but you'll see some examples of n-ary trees, too.

Basic tree operations

Here are some basic operations you can perform on a tree:

```
//ex-gtree-1.c
#include <glib.h>
int main(int argc, char** argv) {
 GTree* t = g_tree_new((GCompareFunc)g_ascii_strcasecmp);
 g_tree_insert(t, "c", "Chicago");
 printf("The tree height is %d because there's only one node\n", g_tree_height(t));
 g_tree_insert(t, "b", "Boston");
 g_tree_insert(t, "d", "Detroit");
 printf("Height is %d since c is root; b and d are children\n", g_tree_height(t));
 printf("There are %d nodes in the tree\n", g_tree_nnodes(t));
 g_tree_remove(t, "d");
 printf("After remove(), there are %d nodes in the tree\n", g_tree_nnodes(t));
 g_tree_destroy(t);
 return 0;
}

***** Output *****

The tree height is 1 because there's only one node
Height is 2 since c is root; b and d are children
There are 3 nodes in the tree
After remove(), there are 2 nodes in the tree
```

A few notes on that code:

- You can see how each node in a GTree consists of a key-value pair. The key is used to ensure the tree is balanced, the node is inserted in the proper place, and the value is a pointer to the "payload" that you want to track.
- You have to supply a `GCompareFunc` to `g_tree_new` so the GTree knows how to compare the keys. This can be a built-in function as shown above, or you can roll your own.
- The tree "height" is simply the number of nodes from top to bottom, inclusive. To execute this function, the GTree has to start at its root and move downward until it hits a leaf node. The `g_tree_nnodes` function is even more expensive; it requires a full traversal of the entire tree.

Replace and steal

You've seen the `replace` and `steal` function names before on GHashTable; the ones on GTree work in much the same way. `g_tree_replace` replaces both the key and the value of a GTree entry, as opposed to `g_tree_insert`, which replaces only the value if the key inserted is a duplicate. And `g_tree_steal` removes a node without calling any `GDestroyNotify` functions. Here's an example:

```
//ex-gtree-2.c
#include <glib.h>
```

```
void key_d(gpointer data) {
 printf("Key %s destroyed\n", data);
}
void value_d(gpointer data) {
 printf("Value %s destroyed\n", data);
}
int main(int argc, char** argv) {
 GTree* t = g_tree_new_full((GCompareDataFunc)g_ascii_strcasecmp,
  NULL, (GDestroyNotify)key_d, (GDestroyNotify)value_d);
 g_tree_insert(t, "c", "Chicago");
 g_tree_insert(t, "b", "Boston");
 g_tree_insert(t, "d", "Detroit");
 printf(">Replacing 'b', should get destroy callbacks\n");
 g_tree_replace(t, "b", "Billings");
 printf(">Stealing 'b', no destroy notifications will occur\n");
 g_tree_steal(t, "b");
 printf(">Destroying entire tree now\n");
 g_tree_destroy(t);
 return 0;
}

***** Output *****

>Replacing 'b', should get destroy callbacks
Value Boston destroyed
Key b destroyed
>Stealing 'b', no destroy notifications will occur
>Destroying entire tree now
Key d destroyed
Value Detroit destroyed
Key c destroyed
Value Chicago destroyed
```

In this example, you create the GTree using `g_tree_new_full`; just like with a GHashTable, you can register for notifications for any combination of key or value destruction. The second argument to `g_tree_new_full` can contain data to be passed to the `GCompareFunc`, but there's no need for it here.

Looking up data

GTree provides ways to look up the key only or both the key and the value. This is just like you've seen before in GHashTable; there's a `lookup` and a `lookup_extended`. Here's an example:

```
//ex-gtree-3.c
#include <glib.h>
int main(int argc, char** argv) {
 GTree* t = g_tree_new((GCompareFunc)g_ascii_strcasecmp);
 g_tree_insert(t, "c", "Chicago");
 g_tree_insert(t, "b", "Boston");
 g_tree_insert(t, "d", "Detroit");
 printf("The data at 'b' is %s\n", g_tree_lookup(t, "b"));
 printf("%s\n",  g_tree_lookup(t, "a") ?
    "My goodness!" : "As expected, couldn't find 'a'");

 gpointer* key = NULL;
 gpointer* value = NULL;
 g_tree_lookup_extended(t, "c", (gpointer*)&key, (gpointer*)&value);
 printf("The data at '%s' is %s\n", key, value);
 gboolean found = g_tree_lookup_extended(t, "a", (gpointer*)&key, (gpointer*)&value);
 printf("%s\n", found ? "My goodness!" : "As expected, couldn't find 'a'");

 g_tree_destroy(t);
 return 0;
```

```
}

***** Output *****

The data at 'b' is Boston
As expected, couldn't find 'a'
The data at 'c' is Chicago
As expected, couldn't find 'a'
```

Here you see the doubly-indirect pointer technique again. Since multiple values need to be provided by `g_tree_lookup_extended`, it accepts two pointers to pointers, one to the key and one to the value. And note that if `g_tree_lookup` can't find the key, it returns a NULL `gpointer`, whereas if `g_tree_lookup_extended` can't find the target, it returns a `gboolean` value of FALSE.

Listing the tree with foreach

GTree supplies a `g_tree_foreach` function to iterate over the nodes of the tree in the sorted order. Here's an example:

```
//ex-gtree-4.c
#include <glib.h>
gboolean iter_all(gpointer key, gpointer value, gpointer data) {
 printf("%s, %s\n", key, value);
 return FALSE;
}
gboolean iter_some(gpointer key, gpointer value, gpointer data) {
 printf("%s, %s\n", key, value);
 return g_ascii_strcasecmp(key, "b") == 0;
}
int main(int argc, char** argv) {
 GTree* t = g_tree_new((GCompareFunc)g_ascii_strcasecmp);
 g_tree_insert(t, "d", "Detroit");
 g_tree_insert(t, "a", "Atlanta");
 g_tree_insert(t, "c", "Chicago");
 g_tree_insert(t, "b", "Boston");
 printf("Iterating all nodes\n");
 g_tree_foreach(t, (GTraverseFunc)iter_all, NULL);
 printf("Iterating some of the nodes\n");
 g_tree_foreach(t, (GTraverseFunc)iter_some, NULL);
 g_tree_destroy(t);
 return 0;
}

***** Output *****

Iterating all nodes
a, Atlanta
b, Boston
c, Chicago
d, Detroit
Iterating some of the nodes
a, Atlanta
b, Boston
```

Note that when `iter_some` returned TRUE, the iteration stopped. This makes `g_tree_foreach` useful for searching up to a point, accumulating the first 10 items that match a condition, or things of that sort. And, of course, you can just traverse the entire tree by returning FALSE from the `GTraverseFunc`.

Also, note that you shouldn't modify the tree while iterating over it using `g_tree_foreach`.

There's a deprecated function, `g_tree_traverse`, that was intended to provide other ways to traverse the tree. For example, you could visit the nodes in *post order*, that is visiting a tree from the bottom up. This has been deprecated since 2001, though, so the GTree documentation suggests that any usages of it be replaced with `g_tree_foreach` or a n-ary tree instead. None of the open source applications surveyed here use it, which is a good thing.

Searching

You can find items using `g_tree_foreach` and, if you know the key, `g_tree_lookup`. But for more complicated searches, you can use the `g_tree_search` function. Here's how it works:

```
//ex-gtree-5.c
#include <glib.h>
gint finder(gpointer key, gpointer user_data) {
int len = strlen((char*)key);
 if (len == 3) {
  return 0;
 }
 return (len < 3) ? 1 : -1;
}
int main(int argc, char** argv) {
 GTree* t = g_tree_new((GCompareFunc)g_ascii_strcasecmp);
 g_tree_insert(t, "dddd", "Detroit");
 g_tree_insert(t, "a", "Annandale");
 g_tree_insert(t, "ccc", "Cleveland");
 g_tree_insert(t, "bb", "Boston");
 gpointer value = g_tree_search(t, (GCompareFunc)finder, NULL);
 printf("Located value %s; its key is 3 characters long\n", value);
 g_tree_destroy(t);
 return 0;
}

***** Output *****

Located value Cleveland; its key is 3 characters long
```

Note that the `GCompareFunc` passed to `g_tree_search` actually determines how the search proceeds by returning 0, 1, or -1 depending on which way the search should go. This function could even change the conditions as the search proceeded; Evolution does just that when it uses `g_tree_search` to manage its memory usage.

More than binary: n-ary trees

The GLib n-ary tree implementation is based on the `GNode` structure; as mentioned before, it allows for many child nodes for each parent node. It seems to be rarely used, but for completeness, here's a usage flyover:

```
//ex-gtree-6.c
#include <glib.h>
gboolean iter(GNode* n, gpointer data) {
 printf("%s ", n->data);
 return FALSE;
}
int main(int argc, char** argv) {
 GNode* root = g_node_new("Atlanta");
 g_node_append(root, g_node_new("Detroit"));
 GNode* portland = g_node_prepend(root, g_node_new("Portland"));
```

```
 printf(">Some cities to start with\n");
 g_node_traverse(root, G_PRE_ORDER, G_TRAVERSE_ALL, -1, iter, NULL);
 printf("\n>Inserting Coos Bay before Portland\n");
 g_node_insert_data_before(root, portland, "Coos Bay");
 g_node_traverse(root, G_PRE_ORDER, G_TRAVERSE_ALL, -1, iter, NULL);
 printf("\n>Reversing the child nodes\n");
 g_node_reverse_children(root);
 g_node_traverse(root, G_PRE_ORDER, G_TRAVERSE_ALL, -1, iter, NULL);
 printf("\n>Root node is %s\n", g_node_get_root(portland)->data);
 printf(">Portland node index is %d\n", g_node_child_index(root, "Portland"));
 g_node_destroy(root);
 return 0;
}

***** Output *****

>Some cities to start with
Atlanta Portland Detroit
>Inserting Coos Bay before Portland
Atlanta Coos Bay Portland Detroit
>Reversing the child nodes
Atlanta Detroit Portland Coos Bay
>Root node is Atlanta
>Portland node index is 1
```

You can see that `GNode` allows you to put nodes pretty much anywhere you want to; it's up to you to access them as you see fit. It's very flexible, but it may be so flexible that it's a bit hard to pin down an actual usage scenario. In fact, it's not used in any of the three open source applications surveyed here!

Real-world usage of trees

GTree is a complex structure and doesn't get as much usage as the other containers we've looked at so far. Gaim doesn't use it at all. The GIMP and Evolution have some usages, though.

The GIMP:

- gimp-2.2.4/app/menus/plug-in-menus.c uses a GTree to hold plug-in menu entries. It uses `g_tree_foreach` and a custom GTraverseFunc to traverse the GTree to add the plug-in procedure to the GimpUIManager. It uses the standard C library function `strcmp` as its `GCompareData` function.
- gimp-2.2.4/plug-ins/script-fu/script-fu-scripts.c uses a GTree to hold hold "script-fu" scripts. Each value in the GTree is actually a GList of scripts.

Evolution's evolution-2.0.2/e-util/e-memory.c uses a GTree as part of an algorithm that calculates unused memory chunks. It uses a custom GCompareFunc, `tree_compare`, to order the `_cleaninfo` structures, which point to freeable chunks.

Queues

Concepts of queues

Another handy data structure is a queue. A *queue* holds a list of items and is usually accessed by adding items to the end and removing items from the front. This is useful when you have things

that need to be processed the order in which they arrived. A variation on the standard queue is the "double-ended queue", or *dequeue*, which allows items to be added to or removed from either end of the queue.

There are times when it's good to avoid a queue, though. Queue searching is not particularly fast (it's O(n)), so if you'll be searching frequently, a hash table or tree might be more useful. The same applies to a situation where you'll be needing to access random elements in the queue; if you do that, you'll be doing a lot of linear scans of the queue.

GLib provides a dequeue implementation with `GQueue`; it supports the standard queue operations. It's backed by a doubly-linked list (the GList), so it supports many other operations, as well, such as insertion and removal from the middle of the queue. But if you find yourself using those functions frequently, you may want to rethink your container choice; perhaps another container might be more suitable.

Basic queue operations

Here are some basic GQueue operations using the "ticket line" as the model:

```
//ex-gqueue-1.c
#include <glib.h>
int main(int argc, char** argv) {
 GQueue* q = g_queue_new();
 printf("Is the queue empty?  %s, adding folks\n",  g_queue_is_empty(q) ? "Yes" : "No");
 g_queue_push_tail(q, "Alice");
 g_queue_push_tail(q, "Bob");
 g_queue_push_tail(q, "Fred");
 printf("First in line is %s\n", g_queue_peek_head(q));
 printf("Last in line is %s\n", g_queue_peek_tail(q));
 printf("The queue is %d people long\n",  g_queue_get_length(q));
 printf("%s just bought a ticket\n", g_queue_pop_head(q));
 printf("Now %s is first in line\n", g_queue_peek_head(q));
 printf("Someone's cutting to the front of the line\n");
 g_queue_push_head(q, "Big Jim");
 printf("Now %s is first in line\n", g_queue_peek_head(q));
 g_queue_free(q);
 return 0;
}

***** Output *****

Is the queue empty?  Yes, adding folks
First in line is Alice
Last in line is Fred
The queue is 3 people long
Alice just bought a ticket
Now Bob is first in line
Someone's cutting to the front of the line
Now Big Jim is first in line
```

Most of the method names are fairly self-descriptive, but some of the finer points:

- The various methods to push and pop items from the queue don't return anything, so you need to keep that pointer returned from `g_queue_new` to use the queue.
- Either end of the queue is usable both for adding and removing. If you want to simulate a ticket line with the folks in the back peeling off to buy from another line, that's quite doable.

- There are nondestructive `peek` operations to check the item at the head or tail of the queue.
- `g_queue_free` doesn't accept a function to assist in freeing each item, so you'll need to do that manually; this is the same as with `GSList`.

Removing and inserting items

While a queue is usually only modified by adding/removing items from the ends, GQueue allows you to remove arbitrary items and insert items in arbitrary locations. Here's how that looks:

```
//ex-gqueue-2.c
#include <glib.h>
int main(int argc, char** argv) {
 GQueue* q = g_queue_new();
 g_queue_push_tail(q, "Alice");
 g_queue_push_tail(q, "Bob");
 g_queue_push_tail(q, "Fred");
 printf("Queue is Alice, Bob, and Fred; removing Bob\n");
 int fred_pos = g_queue_index(q, "Fred");
 g_queue_remove(q, "Bob");
 printf("Fred moved from %d to %d\n", fred_pos, g_queue_index(q, "Fred"));
 printf("Bill is cutting in line\n");
 GList* fred_ptr = g_queue_peek_tail_link(q);
 g_queue_insert_before(q, fred_ptr, "Bill");
 printf("Middle person is now %s\n", g_queue_peek_nth(q, 1));
 printf("%s is still at the end\n", g_queue_peek_tail(q));
 g_queue_free(q);
 return 0;
}

***** Output *****

Queue is Alice, Bob, and Fred; removing Bob
Fred moved from 2 to 1
Bill is cutting in line
Middle person is now Bill
Fred is still at the end
```

Lots of new functions there:

- `g_queue_index` scans the queue for an item and returns the index; if it can't find the item, it returns -1.
- To insert a new item in the middle of the queue, you need a pointer to the place where you want to insert it. As you can see, you can get a handle on this by calling one of the "peek link" functions: `g_queue_peek_tail_link`, `g_queue_peek_head_link`, or `g_queue_peek_nth_link`, which returns a GList. Then you can insert an item either before or after that GList.
- `g_queue_remove` lets you remove an item from anywhere in the queue. Continuing the "ticket line" model, this means that folks can abandon the line; they're not stuck in it once they join up.

Finding items

In previous examples, you've seen how you can get an item if you have a pointer to the data it contains or if you know its index. But like the other GLib containers, GQueue also includes several find functions: `g_queue_find` and `g_queue_find_custom`:

```
//ex-gqueue-3.c
```

```
#include <glib.h>
gint finder(gpointer a, gpointer b) {
 return strcmp(a,b);
}
int main(int argc, char** argv) {
 GQueue* q = g_queue_new();
 g_queue_push_tail(q, "Alice");
 g_queue_push_tail(q, "Bob");
 g_queue_push_tail(q, "Fred");
 g_queue_push_tail(q, "Jim");
 GList* fred_link = g_queue_find(q, "Fred");
 printf("The fred node indeed contains %s\n", fred_link->data);
 GList* joe_link = g_queue_find(q, "Joe");
 printf("Finding 'Joe' yields a %s link\n", joe_link ? "good" : "null");
 GList* bob = g_queue_find_custom(q, "Bob", (GCompareFunc)finder);
 printf("Custom finder found %s\n", bob->data);
 bob = g_queue_find_custom(q, "Bob", (GCompareFunc)g_ascii_strcasecmp);
 printf("g_ascii_strcasecmp also found %s\n", bob->data);
 g_queue_free(q);
 return 0;
}

***** Output *****

The fred node indeed contains Fred
Finding 'Joe' yields a null link
Custom finder found Bob
g_ascii_strcasecmp also found Bob
```

Note that if `g_queue_find` can't find the item, it returns null. And you can pass either a library function, like `g_ascii_strcasecmp` , or a custom function like `finder` in the above example as the `GCompareFunc` argument to `g_queue_find_custom` .

Working the queue: Copy, reverse, and foreach

Since GQueue is backed by a GList, it supports some list-manipulation operations. Here's an example of how to use `g_queue_copy` , `g_queue_reverse` , and `g_queue_foreach` :

```
//ex-gqueue-4.c
#include <glib.h>
int main(int argc, char** argv) {
 GQueue* q = g_queue_new();
 g_queue_push_tail(q, "Alice ");
 g_queue_push_tail(q, "Bob ");
 g_queue_push_tail(q, "Fred ");
 printf("Starting out, the queue is: ");
 g_queue_foreach(q, (GFunc)printf, NULL);
 g_queue_reverse(q);
 printf("\nAfter reversal, it's: ");
 g_queue_foreach(q, (GFunc)printf, NULL);
 GQueue* new_q = g_queue_copy(q);
 g_queue_reverse(new_q);
 printf("\nNewly copied and re-reversed queue is: ");
 g_queue_foreach(new_q, (GFunc)printf, NULL);
 g_queue_free(q);
 g_queue_free(new_q);
 return 0;
}

***** Output *****

Starting out, the queue is: Alice Bob Fred
After reversal, it's: Fred Bob Alice
```

```
Newly copied and re-reversed queue is: Alice Bob Fred
```

`g_queue_reverse` and `g_queue_foreach` are fairly straightforward; you've seen them both working on various other ordered collections already. `g_queue_copy` requires a bit of care though, since the pointers are copied but not the data. So when freeing the data, make sure not to do a double-free.

More fun with links

You've seen a few examples of links; here are some handy link removal functions. Recall that each item in the GQueue is actually a GList structure with the data stored in a "data" member:

```
//ex-gqueue-5.c
#include <glib.h>
int main(int argc, char** argv) {
 GQueue* q = g_queue_new();
 g_queue_push_tail(q, "Alice ");
 g_queue_push_tail(q, "Bob ");
 g_queue_push_tail(q, "Fred ");
 g_queue_push_tail(q, "Jim ");
 printf("Starting out, the queue is: ");
 g_queue_foreach(q, (GFunc)printf, NULL);
 GList* fred_link = g_queue_peek_nth_link(q, 2);
 printf("\nThe link at index 2 contains %s\n", fred_link->data);
 g_queue_unlink(q, fred_link);
 g_list_free(fred_link);
 GList* jim_link = g_queue_peek_nth_link(q, 2);
 printf("Now index 2 contains %s\n", jim_link->data);
 g_queue_delete_link(q, jim_link);
 printf("Now the queue is: ");
 g_queue_foreach(q, (GFunc)printf, NULL);
 g_queue_free(q);
 return 0;
}

***** Output *****

Starting out, the queue is: Alice Bob Fred Jim
The link at index 2 contains Fred
Now index 2 contains Jim
Now the queue is: Alice Bob
```

Note that `g_queue_unlink` doesn't free the unlinked GList structure, so you'll need to do that yourself. And since it is a GList structure, you'll need to use the `g_list_free` function to free it -- not the simple `g_free` function. Of course, it's simpler to call `g_queue_delete_link` and let that take care of freeing the memory for you.

Sorting queues

Sorting a queue seems a bit odd, but since various other linked-list operations are allowed (like `insert` and `remove`), so is this one. It could be handy too, if you wanted to occasionally reorder the queue to move higher priority items to the front. Here's an example:

```
//ex-gqueue-6.c
#include <glib.h>
typedef struct {
 char* name;
 int priority;
 } Task;
```

```
Task* make_task(char* name, int priority) {
 Task* t = g_new(Task, 1);
 t->name = name;
 t->priority = priority;
 return t;
}
void prt(gpointer item) {
 printf("%s   ", ((Task*)item)->name);
}
gint sorter(gconstpointer a, gconstpointer b, gpointer data) {
 return ((Task*)a)->priority - ((Task*)b)->priority;
}
int main(int argc, char** argv) {
 GQueue* q = g_queue_new();
 g_queue_push_tail(q, make_task("Reboot server", 2));
 g_queue_push_tail(q, make_task("Pull cable", 2));
 g_queue_push_tail(q, make_task("Nethack", 1));
 g_queue_push_tail(q, make_task("New monitor", 3));
 printf("Original queue: ");
 g_queue_foreach(q, (GFunc)prt, NULL);
 g_queue_sort(q, (GCompareDataFunc)sorter, NULL);
 printf("\nSorted queue: ");
 g_queue_foreach(q, (GFunc)prt, NULL);
 g_queue_free(q);
 return 0;
}

***** Output *****

Original queue: Reboot server   Pull cable   Nethack   New monitor
Sorted queue: Nethack   Reboot server   Pull cable   New monitor
```

Now you have a GQueue to model your workload and occasionally you can sort it, remaining happy in the knowledge that Nethack will be promoted to its rightful position at the front of the queue!

Real-world usage of queues

GQueue isn't used in Evolution, but the GIMP and Gaim use it.

The GIMP:

- gimp-2.2.4/app/core/gimpimage-contiguous-region.c uses a GQueue to store a series of coordinates in a utility function that finds contiguous segments. As long as the segment stays contiguous, new points are pushed onto the end of the queue and then popped off to be checked during the next loop iteration.
- gimp-2.2.4/app/vectors/gimpvectors-import.c uses a GQueue as part of a Scalable Vector Graphics (SVG) parser. It's used as a stack; items are both pushed onto and popped off of the head of the queue.

Gaim:

- gaim-1.2.1/src/protocols/msn/switchboard.c uses a GQueue to track outgoing messages. New messages are pushed on to the tail and when sent, are popped off the head.
- gaim-1.2.1/src/proxy.c uses a GQueue to track DNS lookup requests. It uses the queue as a temporary holding area between the application code and a DNS child process.

Relations

Concepts of relations

A GRelation is like a simple database table; it consists of a series of records, or *tuples*, each of which consists of several fields. Each tuple must have the same number of fields, and you can specify an index on any field to allow lookups on that field.

As an example, you could have a series of tuples holding names with the first name in one field and the last name in the second field. Both fields could be indexed, so that fast lookups could be done using either the first name or the last name.

GRelation shows a bit of a weakness in that each tuple can contain a maximum of two fields. Thus, using it as an in-memory database table cache won't work well unless your table is rather thin. I searched the `gtk-app-devel-list` mailing list for notes on this and found that a patch had been discussed back in February of 2000 that would have expanded this to four fields, but it never seems to have made it into the distribution.

The GRelation seems to be a little-known structure; none of the open source applications that are surveyed in this tutorial are currently using it. A bit of poking around the Web found an open source email client (Sylpheed-claws) that uses it for a variety of purposes, including for tracking IMAP folders and message threads. So it may just need a bit of publicity!

Basic operations of relations

Here's an example of creating a new GRelation with two indexed fields and then inserting a few records and running some basic informational queries:

```
//ex-grelation-1.c
#include <glib.h>
int main(int argc, char** argv) {
 GRelation* r = g_relation_new(2);
 g_relation_index(r, 0, g_str_hash, g_str_equal);
 g_relation_index(r, 1, g_str_hash, g_str_equal);
 g_relation_insert(r, "Virginia", "Richmond");
 g_relation_insert(r, "New Jersey", "Trenton");
 g_relation_insert(r, "New York", "Albany");
 g_relation_insert(r, "Virginia", "Farmville");
 g_relation_insert(r, "Wisconsin", "Madison");
 g_relation_insert(r, "Virginia", "Keysville");
 gboolean found = g_relation_exists(r, "New York", "Albany");
 printf("New York %s found in the relation\n", found ? "was" : "was not");
 gint count = g_relation_count(r, "Virginia", 0);
 printf("Virginia appears in the relation %d times\n", count);
 g_relation_destroy(r);
 return 0;
}

***** Output *****

New York was found in the relation
Virginia appears in the relation 3 times
```

Note that the indexes are added right after calling `g_relation_new` and before calling `g_relation_insert` . That's because other GRelation functions, like `g_relation_count` , depend on an index existing and will fail at runtime if it doesn't exist.

The above code contains a call to `g_relation_exists` to see if "New York" is in any GRelation. This requires an exact match on each field in the relation; you can match on any one indexed field using `g_relation_count` .

You've seen `g_str_hash` and `g_str_equal` functions before in the GHashTable section; they're used here to enable fast lookups of indexed fields in the GRelation.

Selecting tuples

Once data is in a GRelation, it can be fetched using the `g_relation_select` function. The result is a point to a `GTuples` structure, which can be further queried to get the actual data. Here's how to use it:

```
//ex-grelation-2.c
#include <glib.h>
int main(int argc, char** argv) {
 GRelation* r = g_relation_new(2);
 g_relation_index(r, 0, g_str_hash, g_str_equal);
 g_relation_index(r, 1, g_str_hash, g_str_equal);
 g_relation_insert(r, "Virginia", "Richmond");
 g_relation_insert(r, "New Jersey", "Trenton");
 g_relation_insert(r, "New York", "Albany");
 g_relation_insert(r, "Virginia", "Farmville");
 g_relation_insert(r, "Wisconsin", "Madison");
 g_relation_insert(r, "Virginia", "Keysville");
 GTuples* t = g_relation_select(r, "Virginia", 0);
 printf("Some cities in Virginia:\n");
 int i;
 for (i=0; i < t->len; i++) {
  printf("%d) %s\n", i, g_tuples_index(t, i, 1));
 }
 g_tuples_destroy(t);
 t = g_relation_select(r, "Vermont", 0);
 printf("Number of Vermont cities in the GRelation: %d\n", t->len);
 g_tuples_destroy(t);
 g_relation_destroy(r);
 return 0;
}

***** Output *****

Some cities in Virginia:
0) Farmville
1) Keysville
2) Richmond
Number of Vermont cities in the GRelation: 0
```

A few notes on selecting and iterating tuples:

- The records in the `GTuples` structure returned from the `g_relation_select` are in no particular order. To find out how many records have been returned, use the `len` member of the `GTuple` structure.
- `g_tuples_index` accepts three arguments:
  - The GTuple structure
  - The index of the record you're querying
  - The index of the field you wish to retrieve

- Note that you need to call `g_tuples_destroy` to properly free up memory allocated during a `g_relation_select` . This works fine even if no records are actually referenced by the GTuples object.

Wrapup

Summary

In this tutorial you've seen how to use the data structures found in the GLib library. You've seen how you can use these containers to effectively manage your program's data, and you've seen how several popular open source projects use these containers as well. Along the way you've also gotten familiar with many of the GLib types, macros, and string handling functions.

GLib contains a lot of other neat functionality: it's got a threading-abstraction layer, a portable-sockets layer, message-logging utilities, date and time functions, file utilities, random-number generation, and much more. Exploring any of these modules would be worthwhile. And if you're feeling generous, you could even improve some of the documentation -- for example, the documentation for the lexical scanner includes a comment about how it needs some example code and more detail. If you've benefited from open source code, don't forget to lend a hand in improving it!

Acknowledgments

Many thanks to Sven Neumann, Simon Budig, Tim Ringenbach, and Michael Meeks for their helpful feedback on the "real world" GLib usages shown in this tutorial.

Resources

Learn

- The GLib Reference Manual is the definitive source for GLib documentation.
- Several open source applications are discussed in this tutorial. Evolution is a personal information management client; Gaim is an instant-messenger client; and The GIMP is an image manipulation program. All three programs are popular and use the GLib containers extensively.
- For a historical look at GLib, check out George Lebl's two-part series "GNOMEnclature: The wonders of GLib" on developerWorks: Part 1 (developerWorks, April 2000) and Part 2 (developerWorks, June 2000).
- "Better programming through effective list handling" (developerWorks, January 2005) looks at extending singly-linked lists to handle arbitrary data types as an effective tool for processing data.
- " Data structures: Make the right choice" (developerWorks, September 2004) examines a real-world problem and performance-enhancing solution for choosing the best data structure.
- The developerWorks Linux zone has more resources for Linux developers.
- Purchase Linux books at discounted prices in the Linux section of the Developer Bookstore.

Get products and technologies

- Download the complete C source code of all the examples used in this tutorial.
- Download the GNU Compiler Collection (GCC) from the GCC releases page.
- Download GLib 2.4 from the GTK FTP site; there's also work available on GLib v2.6.
- Build your next Linux development project with IBM trial software, available for download directly from developerWorks.
- Purchase Linux books at discounted prices in the Linux section of the Developer Bookstore.

Discuss

- The GTK application developers mailing list is a helpful resource for asking questions and, better still, for researching archives for existing answers. This message was mentioned in the GRelation discussion.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Tom Copeland

Tom Copeland started programming in BASIC on a TRS-80 Model III, but demand for that skill has waned and he now mostly writes Ruby, C, and Java code. He contributes to various open source projects, including PMD and GForge, and he helps administer RubyForge, an open source Ruby project repository. He and his wife, Alina, have five children (Maria, Tommy, Anna, Sarah, and Steven) and live in Virginia. You can contact Tom at tom@infoether.com.