Chapter 1: GLib

1.1 Introduction

The letter *G* is ubiquitous in the world of open−source software; it stands for GNU (Richard Stallman's "GNU's Not Unix"). You'll see it throughout this book in names like GTK+, GLib, GObject, and GNOME, as well as in many other software packages such as Ghostscript and gcc.

To understand the later chapters in this book, you need to learn about a fundamental library called GLib (*libglib−2.0*). It provides basic data structures and utility functions for the GTK+ widget set and GNOME applications in general. This chapter deals with GLib's architecture and introduces the API. You'll see GLib's object system (GObject) in Chapter 2.

You can't really avoid GLib when using GNOME and GTK+. Other libraries such as ORBit use GLib, and many don't depend on any other libraries. The abstractions and utilities that GLib provides are handy for nearly any programming task and simplify ports to other platforms.

This chapter contains no graphical code. It is a concise, point−by−point guide to the most important GLib functions and data structures. You may find this material a bit dry, so you can go directly to Chapter 3 to get started with GTK+. However, you may find yourself regularly looking back to these first two chapters for reference.

1.2 GLib Naming Conventions

As with many other libraries, GLib has naming rules for consistency and readability:

- Function names are always in lowercase, with an underscore between each part of the name: g_timer_new(), g_list_append(). In addition, all function names begin with g_.
- All functions in a library start with a common prefix. In GLib, this prefix is g_.
- Type names contain no underscores, and each component inside starts with a capital letter: GTimer, GList. The names start with G. The notable exceptions to these rules are the elementary types in Section 1.3.
- If a function operates primarily on a certain type, the prefix of this function corresponds to the type name. For example, the g_timer_* functions work with the GTimer type, and g_list_* functions go with GList.

It sounds more complicated than it is.

1.3 Basic Types

To get started with GLib, you should adjust to its elementary types. You might wonder why it is important to use guchar instead of unsigned char. There aren't any real differences as long as you stay on the same platform. However, if you decide that you want to import, export, and interface your software between, say, Windows and Unix, then you'll be thankful that GLib can abstract the basic data types for you.

For example, if you want to do something unpleasant such as define an unsigned integer variable that is exactly 16 bits wide on any potential platform, things can start to look a little ugly in C. Fortunately, GLib takes care of this so that you don't have to get your hands too dirty. The basic types are listed in the table on the opposite page.

To use GLib and all of its types, include the *glib.h* header file in your source code:

```
#include <glib.h>
```

The gpointer and gconstpointer types appear frequently when interacting with the GLib data structures, because they are ***untyped pointers*** to memory. In GLib, functions that use these pointers take responsibility for verifying the type, not the programmer or the compiler. These can be especially handy for type abstraction in callback functions and equality operators used in sorting and iteration.

The GLib header file defines the constants TRUE and FALSE for the gboolean type. However, it's bad style to use equivalence operators with these constants; that is, use if (my_gboolean), not if (my_gboolean == TRUE).

GLib Type	**Corresponding Type in C**
gchar	char
ugchar	unsigned char
gint	int
guint	unsigned int
gshort	short
gushort	unsigned short
glong	long
gulong	unsigned long
gfloat	float
gdouble	double
gint8	int, 8 bits wide
guint8	unsigned int, 8 bits wide
gint16	int, 16 bits wide
guint16	unsigned int, 16 bits wide
gint32	int, 32 bits wide
guint32	unsigned int, 32 bits wide
gint64	int, 64 bits wide
guint64	unsigned int, 64 bits wide
gpointer	void *, untyped pointer
gconstpointer	const void *, constant untyped pointer
gboolean	Boolean value, either TRUE or FALSE

1.4 Basic Utilities

GLib has a number of utilities that simplify everyday interaction with the C programming language and the system that your program runs on. For functions dealing with GLib data structures, see Section 1.5.

1.4.1 Memory Management

If you employ GLib's memory management routines, you can save yourself some headaches   GLib provides additional error checking and diagnostic functionality. As in C, there isn't much to learn; the table on the next page provides a reference for the C programmer.

Instead of malloc(), realloc, and free(), you can use g_malloc(), g_realloc(), and g_free(); they operate in an identical fashion. To allocate memory and zero out any previous content, use g_malloc0(). Note that its syntax is like malloc, *not* calloc().

The advantage of these functions over those in the standard C library is the built−in error handling. If a problem occurs during runtime, g_error() can step in to examine it (see Section 1.4.6). With the usual C library, you might be faced with a core dump if you fail to check the return codes carefully every time you allocate memory.

GLib Function	**Corresponding C Function**
gpointer g_malloc(gulong n_bytes)	void *malloc(size_t size) with error handling
gpointer g_malloc0(gulong n_bytes)	like malloc(), but initializes memory as in calloc()
gpointer g_try_malloc(gulong n_bytes)	like malloc() without error checking
gpointer g_realloc(gpointer mem, gulong n_bytes)	void *realloc(void *ptr, size_t size) with error checking
gpointer g_try_realloc(gpointer mem, gulong n_bytes)	realloc() without error checking
void g_free(gpointer mem)	void free(void *ptr)

Note If you have some special reason for inspecting the return code by hand, you can do so with the GLib functions *g_try_malloc()* and *g_try_realloc()*, which work just like their C counterparts   that is, they return *NULL* upon failure. You could use this in a place where the allocated memory isn't critical (for example, in extra buffers meant to improve performance) or when you're running some sort of probe.

Naturally, if you choose to override GLib's protection mechanism, you need to know exactly what you're doing. For most applications, the normal functions like g_malloc() can save you a lot of code, frustration, and time.

Normal practice dictates that you do not specify the requested memory block size for functions like malloc() and g_malloc() as a concrete number. Instead, you make the compiler or runtime system figure it out from a type size multiple, usually with sizeof(). To make the data types agree, you must apply a cast to the malloc() return value. All of this makes for a mess of parentheses and stars, so GLib offers the macros g_new(), g_new0(), and g_renew(), as demonstrated in this code fragment:

```
typedef struct _footype footype;
footype *my_data;

/* Allocate space for three footype structures (long version) */
my_data = (footype *) g_malloc(sizeof(footype)*3);

/* The abbreviated version using g_new */
my_data = g_new(footype, 3);

/* To initialize the memory to 0, use g_new0 */
my_data = g_new0(footype, 3);
```

```
/* Expand this block of memory to four structures (long version) */
my_data = (footype *) g_realloc(my_data, sizeof(footype)*4);

/* Shorter version */
my_data = g_renew(my_data, 4);
```

You can clearly see how g_new() abbreviates g_malloc() and g_renew() is a short form of g_recalloc() in this fragment. In addition, g_new0() is a brief form for invoking g_malloc0().

Warning Remember that you need to use a type with *g_new()*, just as you would with *sizeof()*. Something like *b = g_new(a, 1)* (where *a* is a variable) yields a compilation error. It will be an ugly error because *g_new()* is a macro.

Memory Chunks

GUI applications tend to repeatedly allocate memory in identically sized blocks (***atoms***). Furthermore, there are relatively few kinds of atoms. GLib uses a mechanism called ***memory chunks*** (GMemChunk) to provide applications with atoms. A chunk consists of several atoms; its block size is the total byte length of the component atoms. Therefore, the block size must be a multiple of the atom size.

Here is an example of how to use g_mem_chunk_new() to request a new memory chunk:

```
GMemChunk my_chunk;

my_chunk = g_mem_chunk_new("My Chunk",          /* name */
                           42,                  /* atom size */
                           42*16,               /* block size */
                           G_ALLOC_AND_FREE);  /* access mode */
```

The g_mem_chunk_new() function has four arguments: a name for the memory chunk that you can use for diagnostics, the size of each atom (here, 42), the overall block size (it's easiest to write this as a multiple of the atom size), and the access mode (see below). The return value is a pointer to the new GMemChunk structure.

Note A *GMemChunk* isn't a data structure. It's a management system for memory fragments that can contain data structures.

The access mode (or type) gives you control over how to create and deallocate the atoms. There are two modes:

- G_ALLOC_AND_FREE allows you to return individual atoms to the memory pool at any time.
- G_ALLOC_ONLY permits deallocation of atoms only when you dispose of the entire memory chunk. This mode is more efficient than G_ALLOC_AND_FREE.

Here is how an example of how to allocate and free memory atoms in the chunk created in the preceding example:

```
gchar *data[50000];
gint i;

/* allocate 40,000 atoms */
for(i = 0; i < 40000; i++)
{
  data[i] = g_mem_chunk_alloc(my_chunk);
}
```

```
/* allocate 10,000 more atoms and initialize them */
for(i = 40000; i < 50000; i++)
{
  data[i] = g_mem_chunk_alloc0(my_chunk);
}

/* free one atom */
g_mem_chunk_free(my_chunk, data[42]);
```

Here, g_mem_chunk_alloc() and g_mem_chunk_alloc0() make the individual atoms available. They work like g_malloc() and g_malloc0(), returning a pointer to the atom's memory, but they take a GMemChunk structure as the argument instead of a size specification. The g_mem_chunk_free() function takes a pointer to an individual atom, returning it to the unallocated pool of memory.

Warning	Remember that you can use *g_mem_chunk_free()* only on atoms of a memory chunk that was created with the *G_ALLOC_AND_FREE* access mode. In addition, never use *g_free()* to free an atom this will inevitably lead to a segmentation fault, because one of the memory chunk deallocation functions will cause a double *free()*.

Several functions clean and dispose of atoms in memory chunks, working on an entire memory chunk at once. Here is an example of how to use these functions:

```
/* free up any unused atoms */
g_mem_chunk_clean(my_chunk);

/* free all unused atoms in all memory chunks */
g_blow_chunks();

/* deallocate all atoms in a chunk */
g_mem_chunk_reset(my_chunk);

/* deallocate a memory chunk */
g_mem_chunk_destroy(my_chunk);
```

- g_mem_chunk_clean(*chunk*) examines *chunk* and deallocates any unused memory. This procedure gives you some manual control over the underlying memory management. The g_mem_chunk_free() function doesn't necessarily deallocate an atom's memory immediately; GLib does this when convenient or necessary. g_mem_chunk_clean() forces immediate deallocation.
- g_blow_chunks() runs g_mem_chunk_clean() on *all* outstanding memory chunks in your program.
- g_mem_chunk_reset(*chunk*) frees all atoms in *chunk*, including those in use. Be careful when using this function, because you might have a few lingering pointers to previously allocated atoms.
- g_mem_chunk_destroy(*chunk*) deallocates all atoms of *chunk* and the memory chunk itself.

As is the case for general memory management, GLib provides some macros that can save you some typing:

```
typedef struct _footype footype;
GMemChunk *pile_of_mem;
footype *foo;

/* create a memory chunk with space for 128 footype atoms */
pile_of_mem = g_mem_chunk_new("A pile of memory",
                             sizeof(footype),
                             sizeof(footype)*128,
                             G_ALLOC_AND_FREE);
```

```
/* the same thing, with g_mem_chunk_create */
/* the name will be "footype mem chunks (128)" */
pile_of_mem = g_mem_chunk_create(footype, 128, G_ALLOC_AND_FREE);

/* allocate an atom */
foo = (footype *) g_mem_chunk_alloc(pile_of_mem);

/* the same thing, with g_mem_chunk_new */
foo = g_mem_chunk_new(footype, pile_of_mem);

/* the same thing, but zero out the memory */
foo = g_mem_chunk_new0(footype, pile_of_mem);
```

The macros' purposes should be fairly obvious from the code. Note that g_mem_chunk_create() is a shorter way to use g_mem_chunk_new() if you know the atom data type. Note, too, that each macro automatically pieces together the chunk's name. Furthermore, g_mem_chunk_new() and g_mem_chunk_new0() are the memory chunk counterparts of g_new() and g_new0().

If you want to see some statistics on your current memory chunks, use g_mem_chunk_print(*chunk*) for a brief report on one chunk or use g_mem_chunk_info() to see detailed information for all chunks.

1.4.2 Quarks

To label a piece of data in your program, you usually have two options: a numeric representation or a string. Both have their disadvantages. Numbers are difficult to decipher on their own. If you know roughly how many different labels you need beforehand, you can define an enumeration type and several alphanumeric symbols. However, you can't add a label at run time.

On the other hand, you can add or change strings at run time. They're also easy enough to understand, but string comparison takes longer than arithmetic comparison, and managing memory for strings is an extra hassle that you may not wish to deal with.

GLib has a data type called GQuark that combines the simplicity of numbers with the flexibility of strings. Internally, it is nothing more than an integer that you can compare and copy. GLib maps these numbers to strings that you provide through function calls, and you can retrieve the string values at any time.

To create a quark, use one of these two functions:

```
GQuark quark;
gchar *string;

quark = g_quark_from_string(string);
quark = g_quark_from_static_string("string");
```

Both functions take a string as their only parameter and return the quark. The difference between the two is that g_quark_from_string() makes a copy of the string when it does the mapping, and g_quark_from_static_string() does not.

Warning Be careful with *g_quark_from_static_string()*. It saves a tiny bit of memory and CPU every time you call it, but may not be worthwhile because you create an additional dependency in your program that can cause debugging problems later.

If you want to verify that *string* has a quark value, call

```
g_quark_try_string(string)
```

This function returns the string's quark if the program has already defined the string as a quark. A return value of zero (0) means that no quark corresponds to that string (there are no quarks with a numeric value of zero).

To recover the string from a quark, use

```
string = g_quark_to_string(quark);
```

If successful, this function returns a pointer to the *quark* string. Make sure that you don't run any free() calls on that pointer   it isn't a copy.

Here is a short quark demonstration program:

```
GQuark *my_quark = 0;

my_quark = g_quark_from_string("Chevre");

if (!g_quark_try("Cottage Cheese"))
{
  g_print("There isn't any quark for \"Cottage Cheese\"\n");
}
g_print("my_quark is a representation of %s\n", g_quark_to_string(my_quark));
```

Note *GQuark* values are numbers assigned to strings and are efficient when tested for equality. However, they have no set numeric order. You can't use the quark values to test for alphabetic order, and thus, you can't use them as sorting keys. If you want to compare the strings that quarks represent, you must extract the strings with *g_quark_to_string()* and then apply an operation like *strcmp()* or *g_ascii_strcasecmp()* to the result.

1.4.3 C Strings

GLib offers several string functions that interoperate with strings in the standard C library (not to be confused with GString, a GLib−specific string type described in Section 1.5.1). You can use these functions to augment or supplant functions like sprintf(), strdup(), and strstr().

The following functions return a pointer to a newly allocated string that you must deallocate yourself:

- gchar *g_strdup(const gchar **str*)

  Copies *str* and returns the copy.
- gchar *g_strndup(const gchar **str*, gsize *n*)

  Copies the first *n* characters of string and returns the copy. The copy always contains an additional character at the end: a NULL terminator.
- gchar *strnfill(gsize *length*, gchar **fill_char*)

  Creates a string that is *length* characters long and sets each character in the string to *fill_char*.
- gchar *g_strdup_printf(const gchar **format*, ...)

  Formats a string and parameters like sprintf(). However, you don't need to create and specify a buffer as you would in sprintf(); GLib does this automatically.
- gchar *g_strdup_vprintf(const gchar **format*, va_list *args*)

Like the preceding function, but is the analog to vsprintf(), a function that uses C's variable argument facility described on the stdarg(3) manual page.

- gchar *g_strescape(const gchar **source*, const gchar **exceptions*)

  Translates special control characters, backslashes, and quotes in *source* to the normal ASCII range. For example, this function converts a tab to \t. The translations performed are for the backspace (\b), form feed (\f), line feed (\n), carriage return (\r), backslash (\ becomes \\), and double quotes (" becomes \"). Any additional non−ASCII characters translate to their octal representation (for example, escape becomes \27). You can specify any exceptions in the string *exceptions*.
- gchar *g_strcompress(const gchar **source*)

  The reverse of g_strescape(); that is, converts an ASCII−formatted string back to one with real escape characters.
- gchar *g_strconcat(const gchar **string1*, ..., NULL)

  Takes any number of *strings* as parameters and returns their concatenation. You *must* use NULL as the final argument.
- gchar *g_strjoin(const gchar **separator*, ..., NULL)

  Joins a number of strings, adding *separator* between each string. For example, gstrjoin("|", "foo", "bar", NULL) yields "foo|bar". As with g_strconcat(), you must place NULL at the end of the argument list. With a NULL separator, gstrjoin() operates like g_strconcat().

For the following functions, you may need to allocate space for the result; GLib won't make a copy for you. These functions work much like their C counterparts, where one of the arguments contains a buffer large enough for a processed string.

- gchar *g_stpcpy(gchar **dest*, const gchar **src*)

  Copies *src* to *dest*, including the NULL terminator. Upon success, this function returns a pointer to the copy of this terminator in *dest*. This function is useful for efficient string concatenation.
- gint g_snprintf(gchar **string*, gulong *n*, const gchar **format*, ...)

  Like sprintf(); you must ensure that there is enough space for *string* in the result. However, you must specify the length of this buffer with *n*. The return value is the length of the output string, even if the output is truncated due to an insufficient buffer. This is the C99 standard, *not* the traditional snprintf() behavior that your machine's C library may exhibit.
- gint g_vsnprintf(gchar **string*, gulong *n*, const gchar **format*, va_list *list*)

  Like the preceding function, but with variable arguments.
- gchar *g_strreverse(gchar **string*)

  Reverses the order of the characters in *string*. The return value is also *string*.
- gchar *g_strchug(gchar **string*)

  Eliminates whitespace from the beginning of *string*, shifting all applicable characters to the left in *string*. Returns *string*.
- gchar *g_strchomp(gchar **string*)

  Eliminates whitespace from the end of *string*. Returns *string*.

- gchar *g_strstrip(gchar **string*)

  Eliminates whitespace from the beginning *and* end of *string*. Returns *string*.
- gchar *g_strdelimit(gchar **string*, const gchar **delimiters*, gchar **new_delimiter*)

  Changes any characters found in *delimiters* to *new_delimiter*. If *delimiters* is NULL, this function uses "_−|<>."; this is the standard set found in G_STR_DELIMITERS. Returns *string*.
- gchar *g_strcanon(gchar **string*, const gchar **valid_chars*, gchar **substituter*)

  Replaces any character in *string* that isn't in *valid_chars* with *substituter*. Returns *string*. Note that this function complements g_strdelimit().

With the exception of g_ascii_dtostr(), these string functions do not alter their arguments:

- gchar *g_strstr_len(const gchar **haystack*, gssize *haystack_len*, const gchar **needle*)

  Looks through the first *haystack_len* characters of *haystack* for *needle*. This function stops when it finds the first occurrence, returning a pointer to the exact place in *haystack_len*. When this function fails to find *needle*, it returns NULL.
- gchar *g_strrstr(const gchar **haystack*, const gchar **needle*)

  Like g_strstr_len, except that this function returns the *last* incidence of *needle* in *haystack*, and it does not take a size parameter.

- gchar *g_strrstr_len(gchar **haystack*, gssize *haystack_len*, gchar **needle*)

  Identical to g_strrstr, except that it searches only the first *haystack_len* characters of *haystack*.
- gsize g_printf_string_upper_bound(const gchar **format*, va_list *args*)

  Examines *format* and args, returning the maximum string length required to store printf() formatting.
- gdouble g_ascii_strtod(const gchar **nptr*, gchar ***endptr*)

  Converts *string* to a double−length floating−point number. If you supply a valid pointer address for *endptr*, this function sets the pointer to the last character in *string* that it used for the conversion. The difference between this function and strtod() is that this function ignores the C locale.
- gchar *g_ascii_dtostr(gchar **buffer*, gint *buf_len*, gdouble d)

  Converts *d* into an ASCII string, writing into *buffer* of maximum length *buf_len* and ignoring the C locale so that the output format is always the same. The resulting string is never longer than G_ASCII_DTOSTR_BUF_SIZE. This function returns a pointer to *buffer*.

Note Use *g_ascii_strtod()* and *g_ascii_dtostr()* to write to and read from files and data streams, not for anything that people read. Because these functions use a unified, locale−independent format, you'll be protected from certain problems. For example, if someone sets a German locale and runs your program, you won't have to worry about the fact that its locale reverses the meanings of the comma and dot for numbers.

Finally, here are a few functions that handle arrays of strings (gchar **). NULL pointers terminate these arrays.

- gchar **g_strsplit(const gchar **string*, const gchar **delimiter*, gint max_tokens)

Uses *delimiter* as a guide to chop *string* into at most *max_tokens* parts. The return value is a newly allocated array of strings that you must deallocate yourself. If the input string is empty, the return value is an empty array.

- gchar *g_str_joinv(const gchar **separator*, gchar ***str_array*)

  Fuses the array of strings in *str_array* into a single string and returns it as a newly allocated string. If *separator* isn't NULL, g_str_joinv() places a copy of it between each component string.
- gchar **g_strdupv(gchar ***str_array*)

  Returns a complete copy (including each component string) of *str_array*.
- void **g_strfreev(gchar ***str_array*)

  Deallocates the array of strings *str_array* and the strings themselves.

Warning Don't use anything other than *g_strfreev()* to free up an array of strings returned by a function like *g_strsplit()* or *g_strdupv()*.

1.4.4 Unicode and Character Encodings

The traditional C string functions and those in the previous section are byte strings. These functions don't need to worry about the length of an individual character because each gchar is one byte long.

The functions in this section are different, because they work with ***Unicode*** characters and strings. Unicode is an extensive, unified character set that can encode any character in any language using the Universal Character Set (UCS; see ISO 10646). Unicode was originally a 16−bit encoding. GLib supports three different encoding schemes (see also man utf−8):

- **UCS−4** is the full 32−bit UCS−compatible encoding. Every character is 4 bytes long, with the Unicode character occupying the lower 2 bytes (the other 2 are typically zero). The GLib data type for UCS−4 is gunichar. It is 32 bits wide and is the standard Unicode type. Some functions use UCS−4 strings (gunichar *).
- **UTF−16** is the native encoding (UTF stands for Unicode Transformation Format). Every character is 2 bytes wide. GLib uses the gunichar2 type for characters in UTF−16. As with UCS−4, you will see UTF−16 strings (gunichar2 *).
- **UTF−8** is important in practice, because it is compatible with ASCII. Normal ASCII characters use 8 bits, but other characters may require 2 or more bytes. Therefore, every file in ASCII format contains valid UTF−8 text, but not the other way around. A significant disadvantage is that the nonuniform character width of UTF−8 renders random access in a text file next to impossible; to get to a certain part, you must start from the beginning and iterate over the characters. GLib doesn't have a type for UTF−8 because the characters don't have uniform sizes. However, you can encode UTF−8 strings with normal character strings (gchar *).

GLib generally uses the 32−bit UCS−4 gunichar type as its Unicode standard. The following functions test individual Unicode characters:

- gboolean g_unichar_validate(gunichar *c*) returns TRUE if *c* is a valid Unicode character.
- gboolean g_unichar_isdefined(gunichar *c*) returns TRUE if *c* has a Unicode assignment.
- gboolean g_unichar_isalnum(gunichar *c*) returns TRUE if *c* is a letter or numeral.
- gboolean g_unichar_islower(gunichar *c*) returns TRUE if *c* is a lowercase letter.
- gboolean g_unichar_isupper(gunichar *c*) returns TRUE if *c* is an uppercase letter.

- gboolean g_unichar_istitle(gunichar *c*) returns TRUE if *c* is titlecase.

Note Titlecase doesn't appear much in English (or many other languages, for that matter). A titlecase letter is usually some sort of composite character or ligature where the first part of the composite goes to uppercase when the letter is capitalized at the start of a word. An example is the *Lj* in the Croatian word *Ljubija*. [1]

- gboolean g_unichar_isalpha(gunichar *c*) returns TRUE if *c* is a letter.
- gboolean g_unichar_isdigit(gunichar *c*) returns TRUE if *c* is a base−10 digit.
- gboolean g_unichar_isxdigit(gunichar *c*) returns TRUE if *c* is a hexadecimal digit.
- gboolean g_unichar_ispunct(gunichar *c*) returns TRUE if *c* is some sort of symbol or punctuation.

- gboolean g_unichar_isspace(gunichar *c*) returns TRUE if *c* is a form of whitespace, including spaces, tabs, and newlines.
- gboolean g_unichar_iswide(gunichar *c*) returns TRUE if *c* normally requires twice the space of a normal character to draw on the screen.
- gboolean g_unichar_iscntrl(gunichar *c*) returns TRUE if *c* is a Unicode control character.
- gboolean g_unichar_isgraph(gunichar *c*) returns TRUE if you can print *c*; that is, if it's not a control character, format character, or space.
- gboolean g_unichar_isprint(gunichar *c*) is like g_unichar_isgraph(), but also returns TRUE for spaces.

If you have a gunichar character *c* and want to know its classification in Unicode, you can run

```
g_unichar_type(c)
```

This function returns one of the following constants (more information in [TUC]):

- G_UNICODE_LOWERCASE_LETTER: Lowercase letter
- G_UNICODE_UPPERCASE_LETTER: Uppercase letter
- G_UNICODE_TITLECASE_LETTER: Titlecase letter
- G_UNICODE_CONTROL: Unicode control character
- G_UNICODE_FORMAT: Unicode formatting character
- G_UNICODE_MODIFIER_LETTER: Modifier (odd−looking letters that modify pronunciation)
- G_UNICODE_SURROGATE: A composite of two 16−bit Unicode characters that represents one character
- G_UNICODE_UNASSIGNED: Currently unassigned character
- G_UNICODE_PRIVATE_USE: Character reserved for private, internal use
- G_UNICODE_OTHER_LETTER: Any miscellaneous letter
- G_UNICODE_COMBINING_MARK: Mark that may be combined with another letter
- G_UNICODE_ENCLOSING_MARK: Mark that contains another letter
- G_UNICODE_NON_SPACING_MARK: Mark that usually requires no space to print; its position depends on another base character
- G_UNICODE_DECIMAL_NUMBER: Digit
- G_UNICODE_DECIMAL_LETTER_NUMBER: Numeral made from a letter
- G_UNICODE_OTHER_NUMBER: Any other numeral
- G_UNICODE_CONNECTION_PUNCTUATION: Binding punctuation
- G_UNICODE_DASH_PUNCTUATION: Dashlike punctuation
- G_UNICODE_OPEN_PUNCTUATION: Opening punctuation (such as a left parenthesis)
- G_UNICODE_CLOSE_PUNCTUATION: Closing punctuation
- G_UNICODE_INITIAL_PUNCTUATION: Starting punctuation

- G_UNICODE_FINAL_PUNCTUATION: Terminal punctuation
- G_UNICODE_OTHER_PUNCTUATION: Any other punctuation
- G_UNICODE_CURRENCY_SYMBOL: Monetary currency symbol
- G_UNICODE_MODIFIER_SYMBOL: Modifier symbol (for example, an accent)
- G_UNICODE_MATH_SYMBOL: Mathematic symbol
- G_UNICODE_OTHER_SYMBOL: Any other odd symbol
- G_UNICODE_LINE_SEPARATOR: A line break (for example, a line feed)
- G_UNICODE_PARAGRAPH_SEPARATOR: Divides paragraphs
- G_UNICODE_SPACE_SEPARATOR: An empty space

Here are some functions for converting single gunichar characters:

- gunichar g_unichar_toupper(gunichar *c*)

  Converts *c* to uppercase if possible and returns the result. It does not modify the character if it has an uppercase version.
- gunichar g_unichar_tolower(gunichar *c*)

  Converts *c* to lowercase if possible.
- gunichar g_unichar_totitle(gunichar *c*)

  Converts *c* to titlecase if possible.
- gint g_unichar_digit_value(gunichar *c*)

  Returns the numeric equivalent of *c*. If c isn't a numeral, this function returns −1.
- gint g_unichar_xdigit_value(gunichar *c*)

  Same as the preceding function, but with hexadecimal numerals.

Now that you know how to do some interesting things with gunichar characters, you probably want to know how you can get your hands on this kind of data. For the most part, you must extract Unicode characters from UTF−8−encoded strings, and in the process, you'll want make certain that these strings are valid, navigate them, and read the individual characters.

Note In the functions you're about to see, you can provide a *NULL*−terminated string, but this isn't always necessary. If a function takes a *gssize* parameter, you can specify the number of bytes in the UTF−8 string that the function should process. If you want to tell a function to process an entire *NULL*−terminated string, use −1 as the size.

- gboolean g_utf8_validate(const gchar **str*, gssize *max_len*, const gchar ***end*)

  Reads at most *max_len* bytes of the UTF−8 string *str*, returning TRUE if the string is valid UTF−8 text. This function returns FALSE upon failure, and you can also specify an address to a gchar pointer as the *end* parameter. The function sets this **end* to the first invalid character in the string when there is a problem. Otherwise, **end* goes to the end of a valid string.
- gunichar g_utf8_get_char_validated(const gchar **p*, gssize *max_len*)

  Tries to extract the byte sequence at *p* from a UTF−8 character as a gunichar UCS−4 character, making sure that the sequence is valid UTF−8. This function returns the converted character upon success. Upon failure, there are two possible return values: (gunichar) −2 if the function ran out of data to process, or −1 if the sequence was invalid.

- gunichar g_utf8_get_char(const gchar **p*)

  Converts the UTF−8 character at *p* to a gunichar character and returns this result.

Warning *g_utf8_get_char()* doesn't check the validity of its parameters. Use this function for strings that you have already verified with *g_utf8_validate()*. Using these two functions is faster than running *g_utf8_get_char_validated()* on every single character in the string.

The rest of the functions in this section assume that their input is valid UTF−8. Unpleasant circumstances arise if the string is not UTF−8.

- gchar *g_utf8_next_char(gchar **p*)

  Returns a pointer to the character in the UTF−8 string following *p*. Therefore,

  ```
  p = g_utf8_next_char(p);
  ```

  advances a pointer by one character. The pointer should not be at the end of the string. (This is actually a macro in the current GLib implementation, not a function.)
- gchar *g_utf8_find_next_char(const gchar **p*, const gchar **end*)

  Same as the preceding function, but with *end* pointing to the end of the UTF−8 string *p*. If *end* is NULL, this function assumes that *p* ends with a NULL value. The return value is NULL if *p* already points the end of the string.
- gchar *g_utf8_prev_char(gchar **p*)

  Same as the preceding function, but looks back to the previous character. There is no error check, and *p* should not point the start of a string.
- gchar *g_utf8_find_prev_char(const gchar **str*, const gchar **p*)

  Also returns the character previous to *p*, but provides an additional error check when you specify the start of the string with *str*. This function returns NULL upon failure.
- glong g_utf8_pointer_to_offset(const gchar **str*, const gchar **pos*)

  Returns the offset (that is, the character index) of *pos* in the UTF−8 string *str*.
- gchar *g_utf8_offset_to_pointer(const gchar **str*, const gchar **pos*)

  Returns a pointer to the *pos*−th character in the UTF−8 string *str*.

Because the traditional C string library doesn't work on UTF−8 strings, GLib provides some equivalents:

- glong g_utf8_strlen(const gchar **str*, gssize *max*)

  Computes the length (in characters) of *str*. You can specify a maximum byte length with *max*.
- gchar *g_utf8_strncpy(gchar **dest*, const gchar **src*, gsize *n*)

  Copies *n* UTF−8 characters from *src* to *dest*. Note that you must allocate the necessary space at *dest*, and that *n* is the number of characters, *not* bytes.

- gchar *g_utf8_strchr(const gchar **str*, gssize *len*, gunichar *c*)

Returns a pointer to the first occurrence of *c* in *str*, or NULL if the character is not present. Note that *c* must be in the UCS−4 encoding.

- gchar *g_utf8_strrchr(const gchar **str*, gssize *len*, gunichar *c*)

  Same as the preceding function, but looks for the last occurrence of *c* in *str*.
- gchar *g_utf8_strup(const gchar **str*, gssize *len*)

  Returns a new copy of *str*, translated to uppercase. This string could have a different length; characters such as the German scharfes S go from one character (ß) to two (SS) when converted to uppercase. You are responsible for deallocating the new string.
- gchar *g_utf8_strdown(const gchar **str*, gssize *len*)

  Same as the preceding function, but converts uppercase letters to lowercase. Don't expect a string like NUSSDORFER STRASSE to become Nußdorfer Straße, though; your locale software probably isn't smart enough to get this right.
- gchar *g_utf8_casefold(const gchar **str*, gssize *len*)

  Changes the mixed−case version of *str* into a case−independent form and returns the result as a new string. This result isn't suitable for printed output, but works for comparison and sorting.
- gchar *g_utf8_normalize(const gchar **str*, gssize *len*, GNormalizeMode *mode*)

  Produces a *canonical* version of *str*. In Unicode, there are several ways of representing the same character, such as the case of a character with an accent: It can be a single character or a composition of a base character and an accent. To specify *mode*, use one of the following:

  - G_NORMALIZE_DEFAULT: Normalize everything that doesn't affect the text content.
  - G_NORMALIZE_DEFAULT_COMPOSE: Same as the preceding, but attempt to make composed characters as compact as possible.
  - G_NORMALIZE_ALL: Change everything, including text content. For example, this would convert a superscripted numeral to a standard numeral.
  - G_NORMALIZE_ALL_COMPOSE: Same as the preceding, but attempt to make composed characters as compact as possible.

Note Before you compare UTF−8 strings, normalize them with the same mode. The strings might have the same value but slightly different encoding styles that a comparison function won't recognize.

- gint *g_utf8_collate(const gchar **str1*, const gchar **str2*)

  Compares the UTF−8 strings *str1* and *str2* linguistically (at least as much as possible). If *str1* is less than *str2* in the sort order, the return value is −1; if the strings are equal, 0 is the result; and if *str2* comes before *str1*, this function returns 1.

- gchar *g_utf8_collate_key(const gchar **str*, gssize *len*)

  Returns a sorting key for *str*. If you compare two of these keys with strcmp(), the result will be the same as a comparison of their original strings with g_utf8_collate().

Note If you compare a number of UTF−8 strings frequently (for example, if you're sorting them), then you should obtain keys for all of the strings and use *strcmp()* as your comparison function. This approach is much faster than using *g_utf8_collate()* every time, because that function normalizes its parameters every time it runs, and that involves not only a bit of computation, but also memory management time.

Several conversion functions translate strings among the different Unicode encoding schemes. In general, they take a string *str* as input and produce a freshly allocated NULL−terminated string with the same content in a different format. Some of these functions have an *items_read* parameter, which is actually a pointer to an integer; it can write the number of base units converted to this integer (here, base units refers to the number of bytes in UTF−8, 16−bit words in UTF−16, and 32−bit words in UCS−4). Therefore, if you want a function to store this count in a variable i, you would pass &i as the *items_read* parameter to the function. Similarly, you can employ the *items_written* parameter to record the number of characters written to the output stream. You can use NULL for both of these parameters if you don't care about this information.

If the input is in UTF−8 format, and *items_read* is NULL, an error will occur when an incomplete character occurs at the end of the input string (*str*). If you want to find out what this (or any other) error is, use the address of a GError pointer as the *error* parameter (see the "Error Codes" section on the next page; the error class is G_CONVERT_ERROR). These functions return NULL on failure:

- gunichar2 *g_utf8_to_utf16(const gchar ***str*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  Converts a UTF−8 string to a UTF−16 string.
- gunichar *g_utf8_to_ucs4(const gchar **str*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  Converts a UTF−8 string to a UCS−4 string.
- gunichar *g_utf8_to_ucs4_fast(const gchar **str*, glong *len*, glong **items_written*)

  Same as the preceding function, but roughly twice as fast, because it doesn't perform any error checking. Consequently, this function doesn't have the *items_read* and *error* parameters.
- gunichar *g_utf16_to_ucs4(const gunichar2 **str*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  Converts UTF−16 to UCS−4.
- gchar *g_utf16_to_utf8(const gunichar2 **str*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  Converts UTF−16 to UTF−8.
- gunichar2 *g_ucs4_to_utf16(const gunichar **str*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  Converts UCS−4 to UTF−16.

- gchar *g_ucs4_to_utf8(const gunichar **str*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  Converts UCS−4 to UTF−8.
- gint *g_unichar_to_utf8(gunichar *c*, gchar **outbuf*)

  Stores the UCS−4 character *c* as a UTF−8 character in *outbuf*. You must reserve at least 6 bytes in this buffer. The return value is the number of bytes written to the buffer. If *outbuf* is NULL, this function doesn't perform any translation; it simply reports the size of *c* in UTF−8.

Finally, a number of functions perform translations between Unicode and other character encodings:

- gchar *g_locale_to_utf8(const gchar **str*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  Changes a string from your current locale's character encoding to UTF−8. This function (and those that follow) work just like the previous conversion functions.
- gchar *g_filename_to_utf8(const gchar **opsysstring*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  Converts the filename *opsysstring* from your operating system to UTF−8.
- gchar *g_filename_to_uri(const char **filename*, const char **hostname*, GError ***error*)

  Combines *filename* and *hostname* into a UTF−8−encoded URI (Uniform Resource Identifier; URLs are a subset of these). The *filename* string must be a full pathname. You can specify NULL for *hostname*.
- gchar *g_locale_from_utf8(const gchar **utf8string*, glong *len*, glong **items_read*, glong **items_written*, GError ***error*)

  The opposite operation of g_locale_to_utf8(); this function translates *utf8string* into your current locale.
- gchar *g_filename_from_utf8(const gchar **utf8string*, glong *len*, glong * *items_read*, glong **items_written*, GError ***error*)

  Same as the preceding function, but the result is a filename that your operating system understands.
- gchar *g_filename_from_uri(const gchar **uri*, char ***hostname*, GError ***error*)

  Takes a *uri* value in UTF−8 encoding and produces its filename. If there is a hostname in *uri* as well, this function will extract it and set the pointer *hostname* to the hostname. If you don't care about the hostname, you can set this parameter to NULL.

Error Codes

The G_CONVERT_ERROR error class contains the following conditions:

- G_CONVERT_ERROR_NO_CONVERSION: The requested conversion is impossible.
- G_CONVERT_ERROR_ILLEGAL_SEQUENCE: The input string contains an invalid byte sequence.
- G_CONVERT_ERROR_FAILED: The translation failed for an unknown reason.
- G_CONVERT_ERROR_PARTIAL_INPUT: The input string isn't complete.

- G_CONVERT_ERROR_BAD_URI: The input URI isn't valid.
- G_CONVERT_ERROR_NOT_ABSOLUTE_PATH: A path given as input wasn't an absolute path, as required by a function like g_filename_to_uri().

1.4.5 Timer

The Gtimer is nothing more than a stopwatch that is as accurate as your system clock. Here is a demonstration:

```
/* gtimerdemo.c −− demonstration of GTimer */

#include <glib.h>
```

```
#define DURATION 200000

int main(int argc, char **argv)
{
  GTimer *clock = NULL;
  gint i;
  gdouble elapsed_time;
  gulong us;  /* microseconds */

  clock = g_timer_new();
  g_timer_start(clock);
  g_print("Timer started.\n");

  g_print("Loop started.. ");
  for (i = 0; i < DURATION; i++) { ; }
  /* wasting CPU time like this is only allowed in programming examples */

  g_print("and finished.\n");
  g_timer_stop(clock);
  g_print("Timer stopped.\n");

  elapsed_time = g_timer_elapsed(clock, &us);
  g_print("Elapsed: %g s\n", elapsed_time);
  g_print("         %ld us\n", us);

  g_timer_destroy(clock);

  return 0;
}
```

This small program illustrates everything that you need to know about GTimer. The clock variable is a pointer to a GTimer structure, initially set to NULL. To create one of these structures, the program calls g_timer_new(); it sets clock to the return value (a GTimer pointer to the new structure).

The g_timer_start() function starts the stopwatch. The program runs through a loop that does nothing but waste processor time. [2] Afterward, it uses g_timer_stop() to halt the timer.

To retrieve the current state of the timer, call

```
time = g_timer_elapsed(timer, us_ptr);
```

The return value is the elapsed time in seconds, represented as a double−width floating−point number. In addition, you can obtain the fractional part of the elapsed time in microseconds (millionths of a second) with g_timer_elapsed() if you provide a pointer to a gulong variable as *us_ptr*. Because this number does *not* include the whole numbers of seconds, you must multiply the integral part of the second count (obtained by a type cast) by one million and add it if you want a total number of microseconds.

Note If you don't care about microseconds, set *us_ptr* to *NULL*.

You can reset a GTimer structure with g_timer_reset(*timer*), and you can remove one that is no longer needed with g_timer_destroy(*timer*).

1.4.6 Message Logging

To help with runtime error diagnosis, GLib offers several utilities for logging messages to the system console. These are, in order of the priority levels:

1. g_message() for informative messages indicating normal runtime behavior
2. g_warning() for warnings or problems that won't cause errant operation (at least not yet)
3. g_critical() for warnings that likely *are* going to matter
4. g_error() for fatal errors; calling this function terminates your program

These utilities take parameters like printf()   that is, a format string followed by a list of substitution parameters. You don't need to put a newline at the end of the format, though. To ensure that you and your users know that these messages come from your software, you should set the G_LOG_DOMAIN macro when you compile. This can be a short identification string that identifies the application or library. Most GNOME−based programs define G_LOG_DOMAIN with a compiler option like −DG_LOG_DOMAIN=\"*name*\".

This program shows all four message logging types:

```
/* messagedemo.c -- show logging features */

#include <glib.h>

#define NR 42

int main(int argc, char **argv)
{
  g_message("Coffee preparation engaged");
  g_warning("Bean canister #%d empty", NR);
  g_critical("Water flow failure");
  g_error("Heating element incinerated");

  /* this program shouldn't reach this point */
  return 0;
}
```

The output should look something like this:

```
** Message: Coffee preparation engaged

** (process:3772): WARNING **: Bean canister #42 empty

** (process:3772): CRITICAL **: Water flow failure

** ERROR **: Heating element incinerated
aborting...
```

Marking Levels as Fatal

The g_error() call in the preceding program yields an error message telling you that the program is about to abort (and perhaps produce a core dump). You can configure other log priority levels as fatal so that they behave in the same way. For example,

```
g_log_set_always_fatal(G_LOG_LEVEL_WARNING|G_LOG_LEVEL_CRITICAL)
```

sets this behavior for warning and critical messages. This function's argument is a bit mask that you create by applying bitwise OR to any of the following constants:

- G_LOG_LEVEL_CRITICAL
- G_LOG_LEVEL_WARNING
- G_LOG_LEVEL_MESSAGE
- G_LOG_LEVEL_INFO
- G_LOG_LEVEL_DEBUG

Note If an application uses GTK+ or GNOME libraries, you can also supply *−−g−fatal−warnings* as a command−line option to make all warning levels fatal.

Free−Form Messages

If you have have a message that doesn't fit the mold or tone of the preformatted logging utilities, you can send it to the console with g_print() or g_printerr(). The g_print() function works like printf(), sending its output to the standard output (stdout).g_printerr() sends to the stderr, the standard error.

Note Unlike message logging tools like *g_message()*, *g_print()* and *g_printerr()* require that you specify your own line break at the end of your message.

You may be wondering why you can't just use fprintf() with the desired output stream to do this work. Believe it or not, this function may not work well on a Windows system. Another reason is that you can define your own message−processing functions that can alter the log messages and send them output to any place that you want (such as a dialog window or log file). Here is an example:

```
/* printhandlerdemo.c */

#include <stdio.h>
#include <glib.h>

#define N 1

/* print messages in ALL CAPS */
void my_printerr_handler(gchar *string)
{
  GString *msg;

  msg = g_string_new(string);
  msg = g_string_ascii_up(msg);
  fprintf(stderr, "%s\n", msg−>str);
  g_string_free(msg, TRUE);
}

int main(int argc, char **argv)
{
  /* print to stdout */
  g_print("If you lie %d time, no one believes you.\n", N);

  /* print to stderr */
  g_printerr("Ouch.\n");

  /* but if you lie all of the time... */
  g_set_printerr_handler((GPrintFunc)my_printerr_handler);
  g_printerr("%d. Ouch. Ouch. Ouch. (Hey, that really hurts.)", N);

  return 0;
}
```

You'll see how the string functions in my_printerr_handler() work in Section 1.5.1. Here is this program's output:

```
If you lie 1 time, no one believes you.
Ouch.
1. OUCH. OUCH. OUCH. (HEY, THAT REALLY HURTS.)
```

As you can see, you set print handlers with g_set_print_handler() and g_set_printerr_handler(). Their only argument is a GPrintFunc function. The type definition is as follows:

```
typedef void (*GPrintFunc) (const gchar *string);
```

Therefore, your handler must be a void function with a single string argument.

There are two more functions that you should know about when constructing your own error messages: g_strerror() and g_strsignal(). These are platform−independent implementations of strerror() and strsignal(). The g_strerror() function takes an error code such as EBADF or EINVAL and converts it to a slightly more comprehensible message such as "Bad file descriptor" or "Invalid argument." Similarly, g_strsignal() returns the name of a signal when given a numeric signal code.

The advantages of these functions over strerror() and strsignal() are not just that they're platform independent, but that they also create UTF−8 output suitable as input for other libraries, such as GTK+.

1.4.7 Debugging Functions

GLib has several facilities that can help you find bugs in your program. Two of these are macros that take the place of normal return statements. In addition to breaking out of the function, they log a G_LOG_LEVEL_CRITICAL message. Therefore, you can use them in places that your program should not reach in normal operation:

- g_return_if_reached() for void functions.
- g_return_val_if_reached(*val*) for other functions, where you need to return a value *val*.

Two other similar convenience macros are

```
g_return_if_fail(test)
g_return_val_if_fail(test, val)
```

If *test* is false, the function returns, logging a message in the process. You often see these at the beginning of GNOME functions, checking for valid parameters.

There are two more macros that carry out the rudimentary *contract* concept   the assertion, where a certain condition must hold in order for a program to proceed in any meaningful sense:

- g_assert() halts the program with g_error() if its parameter evaluates to FALSE.
- g_assert_not_reached() doesn't take a parameter; it simply stops the program with an error message.

You'll find assertions throughout this book, as well as in most GNOME applications. In addition, most of the functions in the GNOME platform libraries use these safety features to protect against inappropriate arguments.

If your program is far enough along that you're sure that you don't need any assertions, you can set the G_DISABLE_ASSERT macro when you compile (for example, give the compiler a −DG_DISABLE_ASSERT flag). This disables all assertions and saves a little processor time, because it eliminates the tests.

1.4.8 Exception Handling with Error Reporting

The routines in the previous section help diagnose and eliminate serious runtime errors. However, these won't help you much with nonfatal errors that your program can overcome, ignore, or treat in a special way. For example, if a graphical application can't open a file selected from a pop−up file browser, you normally don't want the whole application to abort. Instead, you prefer it to find out just what the problem was, perhaps put up a dialog box, and do whatever it would do if you clicked the file browser's Cancel button.

People tend to refer to these types of errors as *exceptions* and ways to compensate for them as *exception handling* (the online GLib manual uses the term *error reporting*). The traditional C style is for functions to return special error codes to test after a call; some functions provide additional details (for example, through the errno global variable). Higher−level languages often provide special syntax, such as try{}, throw(), catch(){}, and the like in C++ and Java.

GLib doesn't have any complex features like these because it uses C. However, it does provide a system called GError that's a little easier to use than the usual do−it−yourself method in C. The GError data structure is at the core; how you use this structure is just as important as its implementation.

GError and GError Functions

Functions that use GError take the *address* of a GError pointer as their last parameter. If you want to use a variable err declared with GError *err, you must pass it as &err. In addition, you should set the pointer's value to 0. You can specify NULL as this parameter if you like; in that case, the function will disable its error reporting.

A GError structure has the following fields:

- domain (type GQuark): The domain or class of the error; a label for the module or subsystem where the error occurs. Every error domain must have a macro definition with the format *PREFIX_MODULE*_ERROR (for example, G_FILE_ERROR). The macro expands to a form that returns the quark's numeric value.
- code (type gint): The error code; that is, the specific error inside the error domain. Every possible error code requires a corresponding symbol of the form *PREFIX_MODULE*_ERROR_*CODE* in an enumeration type called *PrefixModule*Error (for example, G_FILE_ERROR_TYPE in GFileError).
- message (type gchar *): A complete description of the error, in plain language.

The following fragment demonstrates how to read an error condition from a function (do_something()) that uses GError:

```
GError *error = NULL;

/* use this GError variable as last argument */
do_something(arg1, arg2, &error);

/* was there an error? */
if (error != NULL)
```

```
{
  /* report the message */
  g_printerr("Error when doing something: %s\n", error->message);

  /* free error structure */
  g_error_free(error);
}
```

You can see from this code that you need to deallocate the error structure with g_error_free() after you're finished. Therefore, if you supply a GError parameter to a function, you should always check it; otherwise, you risk a memory leak.

If you want to do something other than report the error, you'll probably want to know the error domain and code. Instead of checking this by hand, you should use g_error_matches(), a function that matches errors against domains and codes. The first argument is the GError structure, the second is an error domain, and the third is a specific error code. If the error matches the domain and code, the function returns TRUE; otherwise, it returns FALSE. Here is an example:

```
GError *error = NULL;
gchar *filename;
BluesGuitar *fender, *bender;

<< .. >>

filename = blues_improvise(fender, BLUES_A_MAJOR, &error);
if (error != NULL)
{
  /* see if the expensive guitar is broken */
  if (g_error_matches(error, BLUES_GUITAR_ERROR, BLUES_GUITAR_ERROR_BROKEN))
  {
    /* if so, try the cheap guitar */
    g_clear_error(&error);
    filename = blues_improvise(bender, BLUES_A_MAJOR, &error);
  }
}

/* if nothing's working, default to Clapton */
if (error != NULL)
{
  filename = g_strdup("clapton−1966.wav");
  g_error_free(error);
}

blues_play(filename);
```

In this example, blues_improvise() runs, returning a filename if there wasn't a problem. However, if an error occurs, the program checks to see if the code was BLUES_GUITAR_ERROR_BROKEN in the BLUES_GUITAR_ERROR domain. If this was the problem, the program tries one more time with different parameters. Before this attempt, it clears error with g_clear_error(), a function that frees the GError structure and resets the pointer to NULL.

If there is something in error after this second try, indicating that something still isn't working right, the program gives up. Instead of trying any more blues_improvise() calls, it uses a default filename ("clapton−1966.wav") so that blues_play() can do its thing.

Warning After you use a *GError* * structure, immediately deallocate it and reset the pointer. GError−enabled functions can't use the same pointer to several errors at the same time; there's space for only one error. As mentioned earlier, your program will have a memory leak if you do not free the *GError* memory.

Defining Your Own Error Conditions

To use the GError system to report errors in your own functions, do the following:

1. Define an error domain by creating an appropriately named macro that expands to a unique GQuark value.
2. Define all of the error code symbols with an enumeration type.
3. Add a GError ** argument at the end of each of the functions where you want to use GError (that is, this argument is a pointer to a pointer to a GError structure). If the function uses variable arguments, put this parameter just before the va_args list (...).
4. In the places where your function has detected an error, create a fresh GError structure and fill it in accordingly.

Here is a definition of an error domain and some codes:

```
/* define the error domain */
#define MAWA_DOSOMETHING_ERROR (mawa_dosomething_error_quark())

GQuark mawa_dosomething_error_quark(void)
{
  static GQuark q = 0;
  if (q == 0)
  {
    q = g_quark_from_static_string("mawa-dosomething-error");
  }
  return(q);
}

/* and the error codes */
typedef enum {
  MAWA_DOSOMETHING_ERROR_PANIC,
  MAWA_DOSOMETHING_ERROR_NO_INPUT,
  MAWA_DOSOMETHING_ERROR_INPUT_TOO_BORING,
  MAWA_DOSOMETHING_ERROR_FAILED   /* abort code */
}
```

Take a close look at the definition of mawa_dosomething_error_quark() in the preceding example. It creates new quark for the error domain if none exists, but stores the result in a static variable q so that it doesn't have to perform any additional computation on successive calls.

This fragment illustrates how to use the new domain and codes:

```
void mawa_dosomething_simple(GError **error)
{
  gint i;
  gboolean it_worked;

  << do something that sets it_worked to TRUE or FALSE >>

  if (!it_worked)
  {
    g_set_error(error,
                MAWA_DOSOMETHING_ERROR,
                MAWA_DOSOMETHING_ERROR_PANIC,
                "Panic in do_something_simple(), i = %d", i);
  }
}
```

This function "does something," and if it fails, it uses g_set_error() to set the error condition before it returns. This function takes the error pointer address as its first argument, and if that isn't NULL, sets the pointer to a newly allocated GError structure. The g_set_error() function fills the fields of this structure with the third and fourth arguments (the error domain and code); the remaining arguments are the printf() format string and a parameter list that become the GError's message field.

If you want to use the error code from another function, you need to take special care:

```
void mawa_dosomething_nested(GError **error)
{
  gint i;
  gboolean it_worked;
  GError *simple_error = NULL;

  << do something >>

  if (!it_worked)
  {
    g_set_error(error,
                MAWA_DOSOMETHING_ERROR,
                MAWA_DOSOMETHING_ERROR_PANIC,
                "Panic in do_something_nested(), i = %d", i);
    return;
  }

  do_something_simple(&simple_error);
  if (simple_error != NULL)
  {

        << additional error handling >>

        g_propagate_error(error, simple_error);
  }
}
```

In mawa_dosomething_nested(), a similar error initialization occurs if the first part of the function fails. However, this functoin goes on to call do_something_simple() if the first part worked. Because the function can set an error condition, it would make sense to send that error condition back to the original caller. To do this, the function first collects the do_something_simple() condition in simple_error; then it uses g_propagate_error() to transfer the GError structure from simple_error to error.

Warning Never pass a *GError* ** pointer that you got as a parameter to any other function. If it happens to be *NULL*, your program will crash when you try to dereference (access) anything behind it.

To send an error obtained from a function to some other place, use

```
g_propagate_error(error_dest, error_src)
```

Here, error_dest is the destination of the error as a GError **, and error_src is the source as GError *. If the destination isn't NULL, this function simply copies the source to the destination. However, if the destination is in fact NULL, the function frees the source error.

You might have noticed by now that GError tries hard to achieve transparency with respect to NULL, so that you don't have to worry about memory leaks or extra GError pointers when you don't care about the specific nature of the error. In addition, if one of your functions encounters NULL as the error condition, you can take this as a hint that the user doesn't desire any special error treatment and perhaps wants the function to patch up the problem as much as possible.

You should always keep in mind that GError is a fairly workable tool for dealing with exceptions, but only if you stick to the conventions.

[1]Thanks to Roman Maurer for this example.

[2]If you compile this program, disable your optimizer so that it doesn't eliminate this loop.

1.5 Data Structures

GLib has a number of standard implementations for common data structures, including lists, trees, and hash tables. As is the case for other GLib modules, the names for each data type's functions share a common prefix (for example, g_list_ for lists).

1.5.1 Strings

The standard fixed−length, NULL−terminated strings in C are occasionally error prone, not terribly easy to handle, and not to everyone's taste. Therefore, GLib provides an alternative called GString, similar to the length−tracked string in most Pascal implementations. A GString data structure grows upon demand so that there's never a question of falling off the end of the string. GLib manages its length at all times, and therefore, it doesn't need a special terminating character when it contains binary data. GLib functions that use this data type begin with g_string_.

Note The processing functions for GLib lists, arrays, and strings use a pointer to the data structure as their first parameter and return a pointer to the data structure as their return value. A typical statement might look like this:

```
foo = g_string_do_something(foo, bar);
```

You should always remember to reassign the pointer to whatever the function returns, because you can lose track of the data if the function decides to alter its memory location.

This code shows how to create and initialize the GString type:

```
#include <glib.h>

GString *s1, *s2;

s1 = g_string_new("Shallow Green");
s2 = g_string_sized_new(50);
s2 = g_string_assign(s2, "Deep Purple");
g_print("%s\n", s2−>str);
```

Here, g_string_new() takes a normal C string as a parameter and returns a pointer to a new GString string. On the other hand, if you want to assign a C string to a GString string that already exists, use

```
string = g_string_assign(string, c_string);
```

The str field in GString points to the current contents of the string. As illustrated in the preceding example, you can use it just as you would a regular C string. GLib manages the NULL terminator for you.

Warning The *str* field is read only. If you manage to write something to it, don't expect to be able to get it back or that your program will function in any meaningful sense. The value changes between *g_string_*()* calls.

If you have a fairly good idea of how much text you're going to store in a GString structure, use

```
g_string_sized_new(size)
```

to reserve *size* bytes in advance. This can save some time later if you have a string that slowly grows.

As mentioned earlier, a GString string can contain binary data, including NULL bytes. Naturally, when you initialize these strings, you need to specify the length of the data along with the data itself, because there is no universal terminating character. To allocate such a string, use

```
g_string_new_len(initial_data, length)
```

Adding Characters

All of the functions below return GString *:

- g_string_append(GString **gstring*, const gchar **str*)

  Adds a *str* value to the end of *gstring*.
- g_string_append_c(GString **gstring*, gchar *c*)

  Adds *c* to *gstring*.
- g_string_append_unichar(GString **gstring*, gunichar *c*)

  Adds the Unicode character *c* to *gstring*.

If you want to insert something at the very beginning of a string, use the g_string_prepend functions; their names are otherwise as described in the preceding list.

The g_string_insert functions are the same, except that they take an additional index argument (for the index of where to insert in the string; the first index is 0):

- g_string_insert(GString **gstring*, gssize *pos*, const gchar **str*)
- g_string_insert_c(GString **gstring*, gssize *pos*, gchar *c*)
- g_string_insert_unichar(GString **gstring*, gssize *pos*, gunichar *c*)

The following code demonstrates five of these functions on a string called s1.

```
s1 = g_string_assign(s1, "ar");
s1 = g_string_append(s1, "gh");
s1 = g_string_prepend(s1, "aa");
s1 = g_string_prepend_c(s1, 'A');
```

```
s1 = g_string_insert(s1, 4, "rr");

g_print("%s\n", s1->str);           /* prints "Aaaarrrgh" */
```

To insert binary data into a GString string, use these functions:

- g_string_append_len(GString **gstring*, gssize *pos*, const gchar **str*, gssize *length*)
- g_string_prepend_len(GString **gstring*, gssize *pos*, const gchar **str*, gssize *length*)
- g_string_insert_len(GString **gstring*, gssize *pos*, const gchar **str*, gssize *length*)

Removing Characters

You can pull characters out of a GString string at an arbitrary location with

```
g_string_erase(string, index, num_to_remove)
```

To chop a string down to a certain length, use

```
g_string_truncate(desired_length)
```

where *desired_length* is the final length of the string. If you try to truncate a string to a size that is actually larger than the string, nothing happens. If, however, you also want the string's allocated size to grow to that length, this function can truncate and expand:

```
g_string_set_size(desired_length)
```

Note The new data at the end of such a newly expanded string is undefined. This result typically doesn't make a difference in practice, because GLib terminates the original string with a *NULL* byte.

Here are some examples of these functions in action:

```
s1 = g_string_assign(s1, "Anyway");
s1 = g_string_erase(s1, 4, 1);
/* s1 should now be "Anywy" */
s1 = g_string_truncate(s1, 3);

g_print("%s\n", s1->str);          /* prints "Any" */
```

Miscellaneous String Functions

The following are miscellaneous string functions:

- g_string_equal(*string1*, *string2*)

  Compares *string1* and *string2* and returns TRUE if they match. Note that this function is not like strcmp().
- g_string_hash(*string*)

  Returns a 31−bit hash key for *string*. See Section 1.5.5 for more information on hash keys.
- g_string_printf(*string*, *format*, ...)

  Similar to sprintf(), except that it stores the output in GString *string*. The return value is a GString string and like the other string manipulation functions.

- g_string_append_printf(*string*, *format*, ...)

  Same as the preceding function, but appends the result to *string* instead of replacing the previous value.

Deallocating Strings

Deallocate GString *string* with

```
g_string_free(string, free_orig)
```

Here, *free_orig* is a gboolean value that indicates whether the string should be completely deallocated. If you do want to return all of the data to the free memory pool, g_string_free() returns NULL. However, if you want to keep the actual string data in memory, use FALSE as *free_orig*; the function returns the str field of the structure that you just destroyed. Just remember that you're now responsible for deallocating that data as well with g_free(). Here are two examples:

```
gchar *orig_str;

orig_str = g_string_free(s1, TRUE);
/* s1 and all of its fields are now gone;
   orig_str is NULL */

orig_str = g_string_free(s2, TRUE);
/* s1 is gone;
   orig_str points to its old str field */
```

1.5.2 Lists

One of the simplest but most important data structures is the linked list. Implementing linked lists along with their elementary operations is more or less a finger exercise for experienced programmers.

That doesn't mean that their implementations are bug free, though, and who wants to write yet another linked−list library? GLib provides a GList data type and functions for doubly linked lists. (It also provides a GSList data type for singly linked lists, but this book doesn't cover those.)

Creating Lists

Creating a list is easy:

```
#include <glib.h>

GList *list = NULL;
```

In all of the examples that follow, list will be the general pointer to the list or the list's handle. You must initialize all new, empty lists to NULL.

Note There's no special function to create a list; the NULL GList pointer is all you need.

A list consists of linked *elements*, sometimes called *nodes*. Each element is a GList structure that contains an untyped pointer (gpointer) to a block of data. Make sure you always know which pointer goes to an element and differentiate it from the pointer *in* the element that points to the actual data.

You can use any type that you like; the compiler won't care as long as you use the proper type cast. Most lists contain data of only one type; mixing types in lists requires an additional layer of bookkeeping and is somewhat inefficient and problematic.

Note GLib manages the whole list with the pointer to the first node. However, a *GList* pointer also serves as a list iterator. When you program with *GList* structures, make sure that you keep careful track of your pointers.

Adding List Elements

To append a node to the end of a list, use g_list_append():

```
gint *data_ptr;

data_ptr = g_new(gint, 1);
*data_ptr = 42;
list = g_list_append(list, (gpointer)data_ptr);
```

This fragment declares a new pointer data_ptr and sets it to a newly allocated data block. It then sets the memory block to the integer 42. Then g_list_append() takes the list handle as the first argument and the data pointer as the second; note that you must cast the data pointer to gpointer.

Note All of the list examples in this section use integer data types.

As you might suspect from the name, g_list_prepend() operates just like the append function, except that it places the new element at the beginning of the list instead of the end.

Note Keep in mind that *g_list_prepend()* doesn't need to run through the entire list from the list handle to find the end, and therefore is more efficient than its appending counterpart. If you need to add a lot nodes to a list, it is often faster to prepend them and then perhaps use *g_list_reverse()* if they are not in the desired order.

You can also insert elements at any arbitrary place in the list with

```
g_list_insert(list, data, index)
```

Here, *list* and *data* are as usual, but the third parameter is an index. Note that the new element goes into the place just *after* index, not before. Here's an example:

```
GList *tmp;

/* Insert 2003 after the third element */
data_ptr = g_new(gint, 1);
*data_ptr = 2001;
list = g_list_insert(list, data_ptr, 3);

/* Find the list element that was just inserted... */
tmp = g_list_find(list, data_ptr);

/* ...and insert 2000 before that element */
data_ptr = g_new(gint, 1);
*data_ptr = 2000;
list = g_list_insert_before(list, tmp, data_ptr);
```

If you'd rather have a new element put in place before a certain element, try

`g_list_insert_before(`*`list`*`, `*`node`*`, `*`data`*`)`

Notice that the parameters are different; here, the second parameter *node* is an element in list, *not* an index. The third parameter is the new data block; it will occupy a new node preceding *node*.

Navigating a List

The previous example used

`g_list_find(`*`list`*`, `*`data`*`)`

to find the node for *data* in *list*. This function searches the list and returns a pointer to a GList node that contains the same data block address if one happens to exist in the list (upon failure, it returns NULL). This process is not particularly efficient, because a list may require complete traversal before the function finds (or fails to find) a certain node.

There are several other functions for moving around a list. It's perhaps best to illustrate how they work with an example:

```
GList *list, *ptr;
gint *data_ptr;
gint pos, length;

  << create a list in "list" variable >>

/* point to element at position 3 */
ptr = g_list_nth(list, 3);

/* point to the element _before_ position 3 */
ptr = g_list_nth_prev(list, 3);

/* advance to the next element in the list */
ptr = g_list_next(ptr);

/* record current position of ptr in list */
pos = g_list_position(list, ptr);

/* move back one element */
ptr = g_list_prev(ptr);

/* access the data in position 4 */
data_ptr = g_list_nth_data(list, 4);

/* record position of data_ptr */
pos = g_list_index(list, data_ptr);

/* change data in first and last elements to 42 */
ptr = g_list_first(list);
*(gint *)(ptr->data) = 42;
ptr = g_list_last(list);
*(gint *)(ptr->data) = 42;

/* also change the next-to-last element to 42 */
*(gint *)(ptr->prev->data) = 42;

/* record the length of the list */
```

```
length = g_list_length(list);
```

The random−access functions in the preceding program are as follows:

- g_list_nth(*list*, *n*): Returns the node at position *n*.
- g_list_nth_prev(*list*, *n*): Returns the node just before position *n*.
- g_list_nth_data(*list*, *n*): Returns a pointer to the data block of the node at position *n*.
- g_list_first(*list*): Returns the first node in a list.
- g_list_last(*list*): Returns the last node in a list.

Note Keep in mind that a list's first position is 0.
If you have a pointer to *node* in a list, you can use it as the parameter for the following functions:

- g_list_next(*node*): Returns the next node.
- g_list_prev(*node*): Returns the previous node.

These operations pertain to a node's position (or index):

- g_list_position(*list*, *node*): Returns the position of a node in a list.
- g_list_index(*list*, *data*): Returns the position of a node in *list* that contains *data* basically, the reverse of g_list_nth_data().
- g_list_length(*list*): Returns *list* length.

If you know what you're doing, you can move list nodes around by following the pointers that make up a node, as the last few parts of the example code show. In addition to changing the memory behind data, you can follow the next and prev pointers to access adjacent nodes. Notice that the preceding example uses ptr−>prev−>data. You can use this approach only if you are absolutely sure that ptr−>prev isn't NULL.

Removing Elements

Deleting elements from a list is a little different than you might expect. Remember that you are responsible for the management of each node's data, and just as you created space for each data block, you must also free this space when you're done with it. However, you don't have to worry about the actual nodes.

The functions for removing elements from a *list* are

```
g_list_remove(data)
g_list_remove_all(data)
```

Notice that they do not take an index or a pointer to a node as the node to remove; instead, they want a pointer to the target node's *data block*. The idea is that because you probably need to do something with the data block anyway, you should always be able to find it after you delete a node that pointed to it.

This short example shows the functions in action:

```
/* create a 42 */
data_ptr = g_new(gint, 1);  *data_ptr = 42;

/* place three identical 42s at the beginning of a list */
list = g_list_prepend(list, (gpointer)data_ptr);
list = g_list_prepend(list, (gpointer)data_ptr);
list = g_list_prepend(list, (gpointer)data_ptr);
```

```
/* remove the first 42 */
list = g_list_remove(list, (gconstpointer)data_ptr);

/* remove the rest of them */
list = g_list_remove_all(list, (gconstpointer)data_ptr);

/* free the 42 */
g_free(data_ptr);
```

If the list contains more than one instance of the data pointer, g_list_remove() deletes only the first node that contains the pointer; g_list_remove_all() removes all of them.

Iterating Through Lists

You can run a function on every element of a list at once, similar to mapping in a language like Lisp. This is called *iteration*; the GList utility is

```
g_list_foreach(list, func, user_data)
```

It's helpful to see an example first:

```
void print_number(gpointer data_ptr, gpointer ignored)
{
  g_print("%d ", *(gint *)data_ptr);
}

g_list_foreach(list, print_number, NULL);
```

As usual, *list* is a list. *func* is a function with a prototype matching GFunc, and *user_data* is an untyped pointer. The GFunc definition is

```
typedef void (*GFunc) (gpointer data, gpointer user_data);
```

When you put everything in this example together, g_list_foreach() steps through each element *e* of list, running print_number(*e*−>data, NULL). Notice that the GType function takes the data of the element as the data argument, *not* the element itself. The second argument corresponds to the *user_data* parameter of g_list_foreach(). In this example, it is NULL and completely ignored by print_number().

This example *does* involve user_data:

```
void plus(gpointer data_ptr, gpointer addend_ptr)
{
  *(gint *)data_ptr += *(gint *)addend_ptr;
}

gint *num_ptr;
/* Add 42 to each element */
num = 42;
num_ptr = (gpointer)&num;
g_list_foreach(list, plus, num_ptr);

/* Subtract 65 from each element */
num = −65;
num_ptr = (gpointer)&num;
g_list_foreach(list, plus, num_ptr);
```

The only tricky part of this example is that plus accesses the actual addend data in a roundabout way; it takes a pointer in the form of addend_ptr and then dereferences it to get the value of the addend data. The example here uses this approach mostly to avoid loss of sign due to type casting.

Warning When iterating over a list, don't add or delete any nodes from the list that is, unless you enjoy dancing with segmentation faults.

Of course, you may find this style of iteration excessive. It's fine to iterate like this instead:

```
GList *l;
gint *data_ptr;

for(l = list; l; l = l->next)
{
  data_ptr = l->data;
  ...
}
```

Sorting Lists

If you have experience with the standard C library function qsort(), you'll have no problems with

```
g_list_sort(list, comp_function)
```

The return value is *list* sorted by *comp_function*. Here's a small example:

```
gint gint_compare(gconstpointer ptr_a, gconstpointer ptr_b)
{
  gint a, b;
  a = *(gint *)ptr_a;
  b = *(gint *)ptr_b;

  if (a > b) { return (1); }
  if (a == b) { return (0); }
  /* default: a < b */
              return (-1);
}

list = g_list_sort(list, gint_compare);
```

Here's the type definition for the comparison function:

```
typedef gint (*GCompareFunc) (gconstpointer a, gconstpointer b);
```

To be specific, it takes two pieces of data as parameters (we'll call them *a* and *b*) and returns one of the following:

- A value less than 0 if *a* is less than *b*
- 0 if *a* is equal to *b*
- A value greater than 0 if *a* is greater than *b*

As was the case with iteration, this function receives the elements' data block pointers as its parameters.

A second list sorting variant allows you to pass additional data to a comparison function. To use it, call

```
g_list_sort_with_data(list, compare_func, user_data)
```

In this case, the comparison function has the GCompareDataFunc type and takes an additional data pointer argument.

Miscellaneous List Operations

Three functions take care of a few odds and ends with respect to lists:

```
GList list2 = NULL;

/* copy list into list2 */
list2 = g_list_copy(list);

/* append list2 to the end of list1 */
list = g_list_concat(list, list2);
list2 = NULL;

/* reverse list */
list = g_list_reverse(list);
```

- g_list_copy(*list*)

  Creates a new copy of *list* and returns the copy.

  Warning This function creates copies of the nodes but does not copy the data blocks. Keep track of that memory.
- g_list_concat(*list*, *list2*)

  Appends *list2* to *list* and returns the result. *This function does not make copies of any nodes; it uses the existing nodes.* Therefore, be careful what you pass, and set the second list to NULL after running this function.
- g_list_reverse(*list*)

  Reverses the order of the nodes *list*.

Deallocating Lists

To return all of a list's nodes to the free memory pool, use g_free_list(*list*). Use this deallocation function *only* on the list's first element.

Warning Keep in mind that GLib has no idea how you created the data blocks of your list elements; you're responsible for them and any memory holes that might have to do with them. If you're sure that each data block appears only once in the list, you can come up with a solution using *g_list_foreach()*. Just make sure you know what you're doing.

1.5.3 Arrays

GLib arrays (GArray) are like their C counterparts, except that they don't have a fixed size. They are much faster than GList structures for random access, but the potential cost of inserting data at various points within the array is higher, because they are just a set of contiguous blocks of memory in the actual implementation.

You can create an array with no particular starting size, or if you have an idea of how much space you need, you can preallocate it:

```
#include <glib.h>

GArray *array, *array2;

/* array of integers, unspecified size */
array = g_array_new(TRUE,               /* use null terminator */
                    FALSE,              /* don't blank memory */
                    sizeof(gint));      /* element size */

/* array of unsigned chars, size 50 */
array2 = g_array_sized_new(FALSE,        /* no null terminator */
                           TRUE,         /* zero memory */
                           sizeof(guchar), /* element size */
                           50);          /* create space for 50 */
```

Create an array with one of these functions:

```
g_array_new(null_terminated, clear, element_size)
g_array_sized_new(null_terminated, clear, element_size, reserved_size)
```

Here, *null_terminated* indicates the use of a NULL terminator, *clear* tells the function to zero out the memory before returning the array, and element_size is the byte length of each element in the array; *reserved_size* in g_array_sized_new() is the array's initial size. Upon success, these functions return the newly allocated array.

Note It doesn't make too much sense to set the first two parameters to *TRUE*, because if you want to terminate your array with *NULL*, you don't want to have any *NULL* bytes that aren't actually the end of the array.

To access an element in an array, use the macro

```
g_array_index(a, type, i)
```

where *a* is the array, *type* is the array's element type, and *i* is the index. Because this is a macro, you can write code like so (for example, to set the element at index 1 to 37):

```
g_array_index(array, gint, 1) = 37;
```

Warning This looks quite wrong in many ways, and you do need to be careful. In particular, you must be absolutely sure that the index exists in your array. Look at the *len* field of an array to check its length (for example, *array−>len* in the preceding example). Also, although *g_array_sized_new()* preallocates space, its initial length is still zero.

To *create* elements in a GArray, you need to add them with a function or use g_array_set_size().

Adding Elements

To add things to your GArray, you need to fill a regular C array with your data. You can add elements to an array in three places:

- At the end: Use g_array_append_vals().
- At the beginning: Use g_array_prepend_vals().

- In the middle: Use g_array_insert_vals().

This code illustrates how these functions work:

```
gint c_array[3];

c_array[0] = 42; c_array[1] = 23; c_array[2] = 69;
/* add the elements in c_array to the end of the GArray array */
array = g_array_append_vals(array, (gconstpointer)c_array, 3);

/* insert 220 and DEADBEEF at index 1 */
c_array[0] = 220; c_array[2] = 0xdeadbeef;
array = g_array_insert_vals(array, 1, (gconstpointer)c_array, 2);
```

There is a way to add a single item to an array, but you must have a variable that contains the data handy. Unfortunately, the names are confusing they are like the three macros in the preceding code, but end in val instead of vals. Because these are macros (and hence not terribly smart), you can't use a constant like 42. In any case, here is a demonstration of g_array_prepend_val():

```
gint tmp;

tmp = 380;
/* insert 380 at the beginning of the array */
array = g_array_prepend_val(array, tmp);
```

Note Of the functions here, only *g_array_append_vals()* has reasonable performance. The others must shift memory around; therefore, the characteristics of *GArray* in this respect are the opposite of *GList*.

If you want to create multiple elements in an array at once, use

```
g_array_set_size(array, size)
```

You can then set the individual elements with g_array_index().

Deleting Elements

You can remove elements in two ways. You can use g_array_remove_index(), which does what you would expect: It pulls an element at a given index out of the array:

```
/* delete element at index 2 */
g_array_remove_index(array, 2);
```

This approach isn't terribly quick, so there is an alternative that replaces the deleted element with the last element in the array. However, if you care about the order of your array, this isn't for you:

```
/* replace index 1 with last element and shorten array */
g_array_remove_index_fast(array, 1);
```

Sorting Arrays

If you perhaps called one too many g_array_remove_index_fast() functions, you can use g_array_sort() and g_array_sort_with_data(). These functions work just like their g_list_sort* counterparts; see Section 1.5.2.

Deallocating Arrays

As was the case with GString (see Section 1.5.1),

```
g_array_free(array, preserve)
```

the *preserve* Boolean value indicates whether you want to preserve the actual data in *array* or not. It returns a pointer to the data (type: gchar *) if you use TRUE as the second parameter; you are responsible for deallocating this later with g_free().

1.5.4 Trees

Another classic data structure is the tree. There are more types of trees than you probably want to know about (splay trees, threaded trees, red−black trees, and so forth), and if you really want to know about them, have a look at [Knuth] and [Cormen]. However, if you just want to use one of them, GLib's GTree type is a complete implementation of balanced binary trees.

Creating Trees

One of the most noteworthy things about GTree is that it doesn't just contain simple elements like GList and GArray. A leaf of a (search) tree not only contains some data, but also a key corresponding to that data. That key is available to help GLib sort through the tree and find the data. For example, you could use the customer number as a key in a customer database, a telephone number as the key for telephone information, the name of a participant spelled phonetically   well, you get the idea.

You must define a comparison relation for your keys (greater or less than) so that the tree can be balanced. If you define it as GCompareFunc (see Section 1.5.2), you can call

```
g_tree_new(compare_func)
```

to create your tree. However, if you opt for GCompareDataFunc instead, use

```
g_tree_new_with_data(comp_func, comp_data)
```

One step further is

```
g_tree_new_full(comp_func, comp_data, key_destroy_func, value_destroy_func)
```

which can also take care of your data's memory management with a pair of GDestroyNotify function definitions. You have to create these functions yourself; GLib calls *value_destroy_func*() when it needs to deallocate the data in a node, and *key_destroy_func* when it needs to free a node's key. It's a simple function prototype   just a void function that takes a single untyped pointer as a parameter:

```
typedef void (*GDestroyNotify) (gpointer data);
```

Note In all of the functions described in this section, when you see something like "this or that will be freed," it means that GLib will do it, and only on the condition that you created the tree with *g_tree_new_full()*. Otherwise, you need to worry about it yourself. You probably don't want to think about that, though, because trees can get complicated.

As usual, GLib manipulates only the keys and data with untyped pointers. Furthermore, you do not reassign your GTree variables after every function call, as with the other types.

Enough talk; let's look at some actual code.

```
#include <glib.h>

GMemChunk *key_chunk;
GTree *tree;

/* compare gints; ignore extra data parameter */
gint key_cmp(gconstpointer a_ptr, gconstpointer b_ptr, gpointer ignored)
{
  gint a, b;
  a = *(gint *)a_ptr;
  b = *(gint *)b_ptr;

  if (a < b)    { return (1); }
  if (a == b)      { return (0); }
  /* if a > b */  return (−1);
}

void free_key(gpointer key)
{
  g_mem_chunk_free(key_chunk, key);
}

void free_value(gpointer value)
{
  g_string_free((GString *)value, TRUE);
}

/* prepare memory for keys and values */
key_chunk = g_mem_chunk_create(gint, 1024, G_ALLOC_AND_FREE);

/* create tree */
tree = g_tree_new_full(key_cmp,
                       NULL,      /* data pointer, optional */
                       free_key,
                       free_value);
```

This program draws storage for the tree's keys from memory chunks and uses GString strings for its values. The three functions are the comparison, key deallocator, and value deallocator. You can see that once you have these three utility functions, you need only run a single function g_tree_new_full() to create the tree.

Adding and Replacing Nodes

Insert a new node into a tree with key *key* and value *value* with

```
g_tree_insert(tree, key, value)
```

If *key* already exists in the tree, this function replaces the old value, and if the tree was created with deallocation functions, returns that value to the memory pool. It does not free the old key; instead, it frees the key that you just passed.

Therefore, you need to be careful if you want to use the key that you passed to g_tree_insert() after the function call. You may want to use this instead:

```
g_tree_replace(tree, key, value)
```

It works just the same as insertion, but when it finds a node with a matching key, it deallocates the old value *and* the old key, replacing them with the new ones. Of course, if you use this version, you must make sure that the old key hasn't wandered somewhere else in your program.

Because both functions have pitfalls, the easiest way to avoid a core dump when using these functions is to reset any pointers to the key and value after placing them in a tree.

Here is an example:

```
gint *key_ptr;
GString *value;

/* insert 42 into the tree */
key_ptr = g_chunk_new(gint, key_chunk);
*key_ptr = 42;
value = g_string_new("forty-two");

g_tree_insert(tree, key_ptr, value);
```

To create the node, you need to get a new memory chunk for the key and then a new GString for the value. Notice how this works in tandem with free_key() and free_value(), discussed earlier.

Finding Nodes

To find a node in *tree* matching *key*, use

```
g_tree_lookup(tree, key)
```

The return value is a pointer to the matching node's value if successful, or NULL if *key* isn't in the tree.

There's a slightly more complicated version:

```
g_tree_lookup_extended(tree, key, key_ptr_addr, value_ptr_addr)
```

Upon a match, this function sets the pointer behind *key_ptr_addr* to the key of the matching node, and likewise with *value_ptr_addr* and the matching value. The return value is TRUE if there's a match, and FALSE otherwise. Use this function only if you need to access the key in the tree for some reason (for example, if you didn't define a function to deallocate keys and need to do it by hand).

Warning — With *g_tree_lookup_extended()*, you can change keys that are in trees. Don't do this; GLib's tree navigation system won't be able to cope with the change.

Here are the functions in action:

```
gint *key_ptr, *key_ptr2;

/* look up 37 in the tree */
key_ptr = g_chunk_new(gint, key_chunk);
*key_ptr = 37;

value = (GString *) g_tree_lookup(tree, key_ptr);
```

```
if (!value)
{
  g_print("%d not found in tree.\n", *key_ptr);
} else {
  g_print("%d found; value: %s.\n", *key_ptr, value->str);
}

/* See if 42 is in there */
*key_ptr = 42;
if (!g_tree_lookup_extended(tree, key_ptr,
                            (gpointer)&key_ptr2,
                            (gpointer)&value))
{
    g_print("%d not found in tree.\n", *key_ptr);
} else {
    g_print("%d found; value: %s.\n", *key_ptr, value->str);
}

g_mem_chunk_free(key_chunk, key_ptr);
```

Warning If you choose not to provide key and value memory management functions when you create the tree, you need to know exactly what the keys and values look like in memory, and it's particularly important to keep track of your keys. For example, keys with pointers to any other data invite memory leaks.

Deleting Nodes

To completely remove a node from a tree, including its key and value, use

```
g_tree_remove(tree, key)
```

However, if you want to preserve the key and value, or you want to remove a node from a tree only temporarily, use

```
g_tree_steal(tree, key)
```

However, make sure that you have pointers to the *original* key and value before you run this, or you'll lose track of them. One way to do this is with g_tree_lookup_extended(); the following code builds on the function call that you saw earlier:

```
/* pull a node from the tree */
g_tree_steal(tree, key_ptr2);

/* key_ptr2 and value contain the key and value (see above)--
   now we'll throw them right back into the tree */
g_tree_insert(tree, key_ptr2, value);

/* this time get rid of the node for good */
g_tree_remove(tree, key_ptr2);
```

Traversing a Tree

As with lists and arrays, you can iterate over a GTree tree. This is called *traversing* the tree, and you typically want to do it in the sort order of the nodes' keys. Use

```
g_tree_foreach(tree, func, data)
```

to call *func* on every node in *tree*. Note that *func* has the GTraverseFunc definition:

```
typedef gboolean (*GTraverseFunc)
           (gpointer key, gpointer value, gpointer data);
```

The traversal goes in the order of the keys, with the smallest element first. The GTraverseFunc can halt the traversal at any time by returning TRUE; otherwise, it returns FALSE to keep things moving (you could use this feature when looking for something). Here's an example that prints every node in the tree:

```
/* print a node in a traversal */
gboolean print_node(gpointer key, gpointer value, gpointer ignored)
{
  g_print("[%d %s] ", *(gint *)key, ((GString *)value)->str);
  return FALSE;
}

g_tree_foreach(tree, print_node, NULL);
```

This example uses the third parameter of the GTraverseFunc:

```
/* add the keys; ignore the value */
gboolean sum_keys(gpointer key, gpointer value_ignored, gpointer sum)
{
    *(gint *)sum += *(gint*)key;
    return FALSE;
}

gint sum = 0;

g_tree_foreach(tree, sum_keys, &sum);
```

Tree Statistics

The following functions report on the size of a tree:

- gint g_tree_nnodes(*tree*)

  Returns the total number of nodes in *tree*.
- gint g_tree_height()

  Returns the height of *tree*.

Removing a Tree

To return a tree and its nodes to the free memory pool, use

```
g_tree_destroy(tree)
```

Notice that this function doesn't end in _free like many of the other functions. If you provided deallocation functions for keys and values, this function also completely frees the keys and values. Otherwise, you're responsible for that memory.

1.5.5 Hash Tables

The last GLib data structure that this book covers is another perennial favorite: the hash table. These tables assign keys to values, and using an efficient internal representation, allow you to quickly access values using the key. The GLib data type for a hash table is GHashTable.

As with trees, you can choose any data type that you like for the keys and values of hash tables. An entry in a hash table consists of two untyped pointers: one for the key and the other for the type. GNOME software makes broad use of GHashTable because it can associate data between any two types. [3]

1.5.6 Creating Hash Tables

Use

```
g_create_hash_table_new(hash_func, equal_func)
```

to create a new hash table, returning the result as GHashTable. The two parameters are functions. The first is the *hash function*, with a type of GHashFunc. Following this is an equality test function (type GEqualFunc) that determines whether two keys are equal. Although you can probably use the built−in default functions, it never hurts to know the types of the following parameters:

```
typedef guint (*GHashFunc)(gconstpointer key);

typedef gboolean (*GEqualFunc)(gconstpointer a, gconstpointer b);
```

The equality function is simple; it takes two keys as parameters and, if they are equal, returns TRUE, or FALSE if they aren't equal.

Hash functions are a little trickier. They take a key as input and (efficiently) return a *hash value*, a guint integer that characterizes the key in some way. This isn't a unique mapping like a quark; you can't get a key back from a hash value. The important part about hash values is that they return values that are well distributed throughout the guint domain, even for similar keys.

If you want to know about the theory of hash values and algorithms, have a look at the algorithms books in the bibliography. For the most part, you will probably find that one of the following default hash functions fits your needs:

- g_str_hash(*string*) processes gchar * *string* into a hash value. If your keys are strings, use this function.
- g_int_hash(*int_ptr*) treats *int_ptr* as a *pointer* to a gint value and generates a hash value from the gint value. Use this function if your keys are of type gint *.
- g_direct_hash(*ptr*) uses *ptr* as the hash value. This function works when your keys are arbitrary pointers.

If you use one of these hash functions, there are corresponding key equality functions at your disposal: g_str_equal(), g_int_equal(), and g_direct_equal().

Here is an example:

```
GHashTable *hash1;
```

```
hash1 = g_hash_table_new(g_direct_hash, g_direct_equal);
```

You are responsible for the memory management with hash tables created with g_hash_table_new(). However, just as in the case of trees in Section 1.5.4, the

```
g_hash_table_new_full(hash_func, equal_func, key_destroy, value_destroy)
```

function can deallocate your hash table entries automatically if you provide it with GDestroyNotify functions:

```
GHashTable *hash2;
hash2 = g_hash_table_new_full(g_str_hash, g_str_equal, g_free, g_free);
```

In this example, the keys and values of the hash table could be dynamically allocated C strings, because g_free() returns these to the free memory pool.

Note You can combine a string and integer hash values with XOR into a hash function like this:

```
struct vpair
{
  gchar *str;
  int value;
};

GHashFunc (struct vpair *p) {
  return(g_str_hash(p->str)^p->value);
}
```

Inserting and Replacing Values

Most GLib programmers add new values to a hash table with this function:

```
g_hash_table_replace(hash_table, key, value)
```

Here is an example:

```
SomeType *key;
AnotherType *value;

key = g_new(SomeType, 1);
value = g_new(AnotherType, 1);

  << ... >>

g_hash_table_replace(hash1,
                     key,              /* key */
                     value);           /* value */

g_hash_table_replace(hash2,
                     g_strdup("foo"),    /* key */
                     g_strdup("bar"));   /* value */
```

As with many other GLib functions, the key and value are untyped pointers. If the key is already in the hash table, this function replaces the value corresponding to that key. However, if you created the table with deallocation functions to manage the key and value memory, g_hash_table_replace() frees the old key's memory because it isn't needed.

There is a seldom−used alternative to g_hash_table_replace():

```
g_hash_table_insert(hash_table, key, value)
```

This function works just like g_hash_table_replace(), except that it deallocates the *new* key if an old key in the hash table matches the new key. Therefore, you must be careful if you still have a pointer to *key* somewhere.

If you want to be completely safe, NULL out any pointers to the key (and value) that you add with either of these functions.

Note The difference between these two functions with respect to the actual content of your keys is an issue only when the key equality function doesn't take all of the data in the key into account. If your keys use a simple data type, such as an integer or string, there is no difference.

To find out how many entries are in a hash table (that is, the number of key−value pairs), use

```
g_hash_table_size(table)
```

Finding Values

The easiest way to find something in a hash table is to use

```
g_hash_table_lookup(table, key)
```

The return value is a pointer to the value, or NULL if the key isn't in any of the hash table's entries.

```
gchar *key, *value;

value = (gchar*)g_hash_table_lookup(hash2, "foo");
if (value)
{
  g_print("hash2{\"foo\"} = %s\n", value);
} else {
  g_print("foo isn't in the hash table\n");
}
```

If you need access to the keys and values in the hash table, try

```
g_hash_table_lookup_extended(table, key, key_addr, ptr_addr)
```

Here's an example:

```
if (g_hash_table_lookup_extended(hash2,
                                 "foo",
                                 (gpointer)&key,
                                 (gpointer)&value))
{
  g_print("hash2{\"%s\"} = %s\n", key, value);
} else {
  g_print("foo isn't in the hash table\n");
}
```

This function takes two pointer addresses as the third and fourth parameters: one for a key and the other for a value. If the given key (the second parameter) is in the hash table, this function sets those pointers to the key and value *inside* the hash table. It returns TRUE upon success.

This function is useful when you need to deallocate your key and value manually.

Warning *g_hash_table_lookup_extended()* gives you direct access to the keys in the table entries, meaning that you also have the ability to change them around. That's a really bad idea you risk inaccessible entries and key duplication.

Deleting Entries

To delete an entry in a hash table, use

```
g_hash_table_remove(table, key)
```

where *table* is the hash table and *key* is the entry's key. When successful, this function returns TRUE, and if you created the hash table with automatic key and value deallocation functions, it also frees the key and value memory. Note that this procedure does not try to deallocate the key that you gave as a parameter, so you can write statements with constants, such as this:

```
g_hash_table_remove(hash2, "this");
```

You can also prevent the GLib hash table from trying to deallocate the key and value with

```
g_hash_table_steal(table, key)
```

Iterating Through Hash Tables

You can call a function on every single entry in a hash table, just as you can for lists, arrays, and trees. There are three ways to iterate:

- void g_hash_table_foreach(GHashTable **table*, GHFunc *func*, gpointer *user_data*)

  Runs *func* on every entry in *table*. This function passes *user_data* as the third parameter to *func*.
- void g_hash_table_foreach_remove(GHashTable **table*, GHRFunc *func*, gpointer *user_data*)

  Same as the preceding function, but removes the entry if *func* returns TRUE. This approach includes running any deallocation functions on the key and value that you might have specified when you created the hash table. This function is useful if you need to filter a hash table.
- void g_hash_table_foreach_steal(GHashTable **table*, GHRFunc *func*, gpointer *user_data*)

  Same as the preceding function, but when removing entries, this function doesn't ever try to deallocate the key and data. This function is useful for moving entries to other hash tables or data structures.

Here is the type definition for GHFunc (GHRFunc is the same, except that it returns gboolean):

```
typedef void (*GHFunc)(gpointer key, gpointer value, gpointer user_data);
```

The following is a short example. If you need to see something that involves the user_data parameter, check out the example for trees in Section 1.5.4.

```
void print_entry(gpointer key, gpointer data, gpointer user_data)
{                                      /* user_data not used */
    g_print("key: %−10s     value: %−10s\n",
            (gchar *)key, (gchar *)data);
}

g_print("Hash table entries:\n");
g_hash_table_foreach(hash2, print_entry, NULL);
```

Deleting Hash Tables

To completely remove a GHashTable, call

```
g_hash_table_destroy(hash_table)
```

This approach includes the key and value data if you supplied deallocation functions when you created the hash table. Otherwise, you'll have to take care of the key and value in each entry yourself; g_hash_table_foreach() is a handy means of doing this.

[3] This sets GLib hash tables apart from the hash tables (or dictionaries) of many interpreted languages; those typically allow only strings as keys.

1.6 Further Topics

GLib has many capabilities that aren't covered here because there just isn't enough space. These include the following:

- **Date and time functions:** Would you like to know how many days elapsed between the end of the Thirty Year War and your grandmother's birthday? GLib makes quick work of this type of problem with its conversion utilities.
- **Message logging:** Behind g_message(), g_error(), and their friends are several macros that send a log domain and log level along with the usual parameters to the g_log() function. You can define new log domains and log levels, and with g_log_set_handler(), what to do when a message comes through.
- **Quicksort:** g_qsort_with_data() is like C's qsort() function, but accommodates an additional data parameter.
- **Singly linked lists:** GSList saves a few bytes per node if you don't need to be able to navigate backward in the list.
- **Pointer arrays:** If you just want an array of gpointer elements that can grow automatically like GArray, GPtrArray offers a simpler API without all of the element size parameters.
- **Byte arrays:** GByteArray arrays are identical to pointer arrays, but with guint8−size elements.
- **String chunks:** For efficient allocation and cleanup of a large number of C strings (not GString strings), GStringChunk is available with an API like those of memory chunks.

- **N−ary trees:** If you want trees where a node can have more than two children, GLib enables you to build them yourself with the GNode data type.
- **Queues:** The popular first in, first out (FIFO) approach is available with GQueue. There are additional routines for double−ended queues.
- **Shell and file utilities:** GLib has a number of facilities for working with files, pipes, and processes.
- **Threads:** GLib offers portable implementations of threads, thread pools, and interprocess communication (mutual exclusion, async queues, and so on).

- **Dynamic module loader:** GModule is a system for loading shared objects into running processes. If you want your program to support plug−ins, take a look at this.

As you can see, GLib is a powerful tool that can't be completely documented in a book like this. However, an experienced programmer can easily get a good sense of how to use everything else by rooting around in the extensive API reference documentation that comes with GLib.

Chapter 2: GObject

Overview

Contrary to any grousing you may have heard, GNOME is just as object−oriented as other modern GUI platforms. The primary difference between GNOME and its "competitors" is that the GNOME library source is in plain C, so you can also program it in C. Libraries such as GTK+ and GNOME rest on the GObject object system. By contrast, other systems rely on the object−oriented features of their programming languages, as in the case of KDE's C++ implementation.

Note This chapter is dry and dense; it covers several complex topics in a relatively small space. This chapter is the second in the book because it reflects aspects of the entire GNOME API. You don't need to fully comprehend this material to move on to the next chapters, especially because many of the techniques appear throughout the book. At the very least, learn how to create objects, manipulate properties, and install signal handlers (Sections 2.5, 2.5.1, and 2.6.5). You do not need to know how to create your own classes.

If you are already familiar with object−oriented programming and its terminology, you can skip the first section and go right to the implementation details in Section 2.2.

2.1 Object−Oriented Programming Basics

Most programmers agree that programs consist of algorithms and data structures [Wirth]. That's all fine and good, but experience over the years has indicated that algorithms tend to depend on data structures, rather than the other way around.

In other words, nearly every algorithm operates on a specific data structure. If a clearly written C program defines data structures such as struct Flipper and struct Slop, you can expect to see associated functions like flipper_insert(), flipper_shift(), flipper_status(), slop_create(), slop_blocksize(), and slop_destroy(). The functions "belong" to their data structures and always take parameters of a specific data structure. You will not be able to run flipper_status() on a Slop structure.

Take a look at the names above. You should recognize GLib's naming convention from Section 1.2. One tenet of object−oriented programming is that data types, variables of these types, and their algorithms belong together. It's just a way of thinking, but it does help if you have tools that can help you with the day−to−day organizational details. This tool can be the programming language itself; the most prominent object−oriented languages are C++ and Java, and nearly all popular scripting languages offer some sort of object system, even though not everyone chooses to use these features.

C isn't an object−oriented language by any stretch of the imagination, but that's not a problem, because the GLib's GObject library (*libgobject−2.0*) provides object−oriented programming features for C.

2.1.1 Objects as Instances of Classes

Let's get back to Flippers and Slop.

You can think of the Flipper and Slop types as ***classes***, and you can think of variables of these types as ***instances*** of those classes. Because classes are data structures, they contain various data fields (***attributes***; later in this book, you will see the term ***property*** see Section 2.4). A class also has several functions that operate on object attributes. These are called ***methods***, and you can just think of them as functions that are attached to certain classes. For example, the Flipper and Slop classes have methods that start with flipper_ and slop_.

An ***object*** is a data structure in memory that conforms to the class or an *instance* of a class. The process of creating an object from a class is called ***instantiation***; you invoke the ***constructor*** of a class to create objects. You can have as many objects as you like, and you can set their attributes any way you like. For example, you can have two objects named red_car and blue_car of the **Car** class, where the only difference between the two objects is in the color attribute.

GObject manipulates its objects with object references: typed pointers to objects that you create and invalidate with special functions. You may have more than one reference to the same object, and therefore, objects have ***reference counts*** to keep track of their references in the currently running program. If the reference count goes to zero, GObject detects that you no longer need the object and performs the following actions:

1. GObject enters the *dispose* phase to get rid of any references to other objects.
2. GObject *finalizes* the object, marking the memory as ready for reuse.
3. A garbage collector returns the memory to the free pool somewhere down the line.

Constructors and destructors allow custom code that runs when you create or destroy an object. Therefore, an object can be more than just a coupling of data structures and algorithms; it can represent a process. For example, you could define **FilmShort** class to show an animated cartoon. When you create an object from this class, the constructor places the animation on your screen and starts playing the animation. Finalizing the object stops the animation and removes it from your monitor.

2.1.2 Inheritance

Because an object belongs to a class, it has a type. This is sometimes called a *membership relation*; an object my_flipper might belong to the **Flipper** class because you created it with the **Flipper** constructor, and therefore, the class and object have a membership relation.

Now let's assume that you need to something a little trickier; for example, you want to write a program to manage your bloated CD collection. First, you create a **CD** class with several attributes, such as the storage location (location), the title (title), and an inventory number (inv_nr). However, pedant that you are, you have some CD−ROMs on your shelves in addition to audio CDs. Therefore, you decide on two classes, **AudioCD** and **CDROM**. There are some differences in the attributes: **AudioCD** has an artist name and track list; **CDROM** has an operating system and version.

Meanwhile, your CD collection grows so large that you have to store it on different planets, and therefore, you must add a planet attribute to both classes. Isn't it a little clumsy to add that to both classes?

Well, yes. **AudioCD** and **CDROM** share several attributes that were in your original **CD** class. Furthermore, audio CDs and CD−ROMs are both CDs, so they should have a membership relation reflecting this fact. It would make sense if **AudioCD** objects and **CDROM** objects were also **CD** objects.

It's possible to implement this approach. You can dig out your old **CD** class and define **AudioCD** and

CDROM as ***subclasses*** of **CD**. They ***inherit*** attributes from their parent class (or superclass) and add some attributes of their own. The whole system is known as ***inheritance***.

Subclasses also inherit methods from their parents. All of the methods from the **CD** class work on **AudioCD** and **CDROM** objects; a **CD** method doesn't care about the specific subclass membership of an object.

You can create subclasses of subclasses. For example, **CDROM** could have a **SoftwareCD** subclass for CDs containing programs and a **RecordsCD** subclass for your archived documents. These subclasses would inherit their attributes and methods from **CDROM** and add their own. If you had a **RecordsCD** object called my_disc, its membership relations would be as follows:

- my_disc is a **RecordsCD** object.
- my_disc is a **CDROM**, object.
- my_disc is a **CD** object.

Figure 2.1 shows the entire ***class hierarchy*** as a tree.

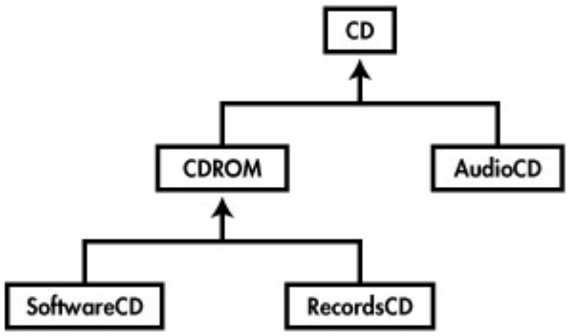

Figure 2.1: CD class hierarchy.

If you implement **RecordsCD** in GObject, Figure 2.1 isn't the complete story. The GObject system has a ***base class*** called **GObject**. Therefore, you have the following membership relations:

- my_disc is a **RecordsCD** object.
- my_disc is a **CDROM** object.
- my_disc is a **CD** object.
- my_disc is a **GObject** object.

See Figure 2.2 for the whole tree diagram.

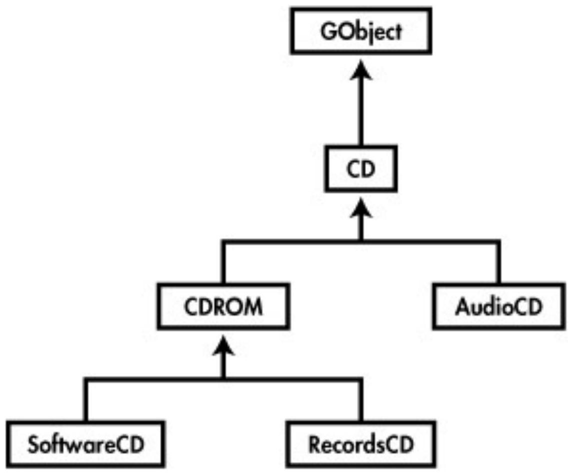

Figure 2.2: CD class hierarchy with GObject base class.
If your collection doesn't contain any exotic CD formats, every CD is a CD−ROM (some subclass of **CDROM**) or an audio CD (**AudioCD**). The **CD** class serves only for the derivation of subclasses; you wouldn't instantiate **CD** objects. Such a class is called an ***abstract class***. Some real−life examples of abstract classes are "building," "vehicle," and "work of art." Nothing is just a work of art, but rather, a painting, installation, sculpture, or whatever.

Interfaces

Let's say that you've finished with your CD inventory program. Now you decide that CDs are too small and so add some tapes and records. Organized person that you are, you decide to expand the inventory system to include some new classes: **Media** as a new abstract class that has your old **CD** as a subclass, as well as two other new subclasses, **Tape** and **Vinyl**. Furthermore, these last two have their own subclasses: **EightTrack**, **StudioTape**, **EP**, and **LP** (see Figure 2.3 for the exact positions).

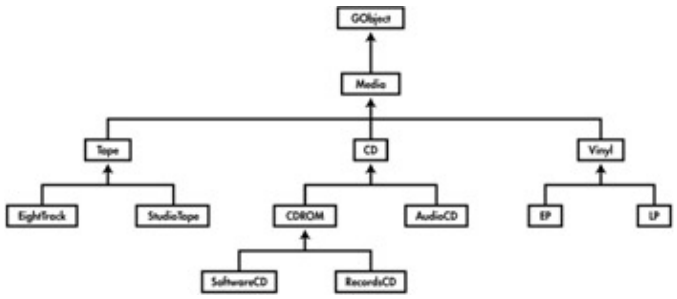

Figure 2.3: Media hierarchy.
With this archival business out of the way, you now want to enable a robot to retrieve audio CDs, 8−track tapes, and records from their storage slots and play them on your designer stereo system. Therefore, the functions that control this system must be able to deal with objects from the **AudioCD**, **Vinyl**, and **EightTrack** classes. This would be practical if they all belonged to a common superclass somewhere, and there is one: **Media**. Unfortunately, this class also includes objects like CD−ROMs and studio tapes, which don't work with your fancy stereo. Therefore, you can't add the support at this level, because the whole idea of a superclass is that *all* of its methods and attributes are supposed to work with its subclasses.

In other object models, it's possible to define classes that inherit from several other classes at once, not just a parent class. This is called ***multiple inheritance***, and it tends to confuse a lot of people.

GObject and many other object systems use a similar concept called an ***interface***. In the ongoing example, all audio CDs, records, and 8−track tapes have one characteristic in common: They fit in the stereo system. In object−oriented terminology, you would say that they all implement the same interface   that is, you can play all of them in a stereo.

Interfaces contain only methods and therefore reflect the capabilities of these objects. Typical interface names often end in *−able*. You could use **Playable** to label the interface to play something on the stereo. Figure 2.4, on the following page, shows the new interface's relationships.

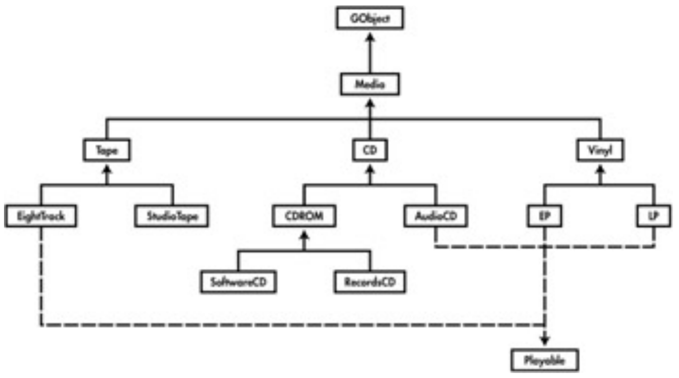

Figure 2.4: Hierarchy of media types, with Playable interface.

All functions that work with **Playable** objects see the methods from only those objects that are defined in the interface. The location of the object classes in the class hierarchy doesn't matter.

What you have just seen sounds easy enough, but now you're about to see the GObject implementation. The rest of this chapter demonstrates how to define classes, create objects, and build interfaces.

2.2 Defining Classes

To put it bluntly, defining a class in GObject isn't easy. If you're used to programming languages like C++, Java, and Smalltalk, this procedure is going to look quite awkward. [1] However, you should try to look at it from an unbiased perspective.

To use GObject, you must include the *glib−object.h* header file and invoke g_type_init() somewhere in your program initialization to set up the GObject type system (GType). Keep in mind that some other function might call g_type_init(), such as gtk_init(), as you'll see in Chapter 3.

2.2.1 Structure Definitions

A GObject class consists of two structures. First, the ***instance structure*** (or *object structure*) holds the class attributes and is the basis for an object in memory. The other structure, the ***class structure***, contains prototypes for certain methods and all signals that the object can provide (you will encounter signals in Section 2.6).

For the **Media** class in Section 2.1.2, you could define Media as the instance structure and MediaClass as the class structure:

```
/* instance structure */
typedef struct _Media
```

```
{
  GObject parent_instance;

  guint inv_nr;
  GString *location;
  GString *title;
  gboolean orig_package;
} Media;

/* class structure */
typedef struct _MediaClass
{
  GObjectClass parent_class;

  /* Signals */
  void (*unpacked)(Media *media);
  void (*throw_out)(Media *media, gboolean permanent);
} MediaClass;
```

The attributes in the instance structure include inv_nr, location, title, and orig_package (this last attribute indicates whether the item is still in the original package). The class structure includes handler prototypes for two signals: **unpacked** and **throw−out**. Ignore them for now; they will reappear in Section 2.6.

Note As you may have noticed by now, there are no prototypes for methods in this class definition. As you will see in Section 2.3, most method prototypes appear outside the class structure.

There is a parent_instance pointer at the beginning of the instance structure, as well as a corresponding parent_class pointer in the class structure. These two definitions are necessary for inheritance; without them, you wouldn't be able do much with your objects (all classes inherit important common features from a base GObject class).

2.2.2 Utility Macros

The following macros are practically essential for smooth operation (you could theoretically write them out by hand each time that you wanted to use these features, but it would test your patience):

```
#define TYPE_MEDIA (media_get_type())

#define MEDIA(object) \
  (G_TYPE_CHECK_INSTANCE_CAST((object), TYPE_MEDIA, Media))

#define MEDIA_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_CAST((klass), TYPE_MEDIA, MediaClass))

#define IS_MEDIA(object) \
  (G_TYPE_CHECK_INSTANCE_TYPE((object), TYPE_MEDIA))

#define IS_MEDIA_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_TYPE((klass), TYPE_MEDIA))

#define MEDIA_GET_CLASS(object) \
  (G_TYPE_INSTANCE_GET_CLASS((object), TYPE_MEDIA, MediaClass))
```

These are somewhat difficult to digest at once, so let's go over them one by one. Note the ongoing difference between the instance and class structures.

- TYPE_MEDIA returns the class type identifier, a GType assignment for the **Media** class. It calls media_get_type()   see Section 2.2.3 for more details. You'll use this macro in any place that calls for an object or type identifier. When you see the term *(class) type identifier* in this section, you need to remember that this is different from a C type.

You will use TYPE_MEDIA when creating **Media** objects.

- MEDIA() is a *casting macro* that casts an *object* (the instance structure) to the **Media** type, or, in other words, the Media structure. This is often simply known as a cast, like its C counterpart.

  Casting macros are useful when calling a method from an object's superclass.
- MEDIA_CLASS() is another casting macro. This one operates on the *class structure*, returning a casted MediaClass structure.
- IS_MEDIA() checks the membership of an object for the Media structure. It is TRUE if the object belongs to the class, and FALSE otherwise.
- IS_MEDIA_CLASS() checks the membership of a class for the MediaClass structure.
- MEDIA_GET_CLASS() yields the class structure of **Media** when given an object (again, an instance) of the Media structure.

As you can see, these macros build on other macros:

- G_TYPE_CHECK_INSTANCE_CAST(*object, type_id, name*) If *object*, a class type identifier *type_id* (for example, TYPE_MEDIA), and the instance structure *name* match, this macro expands to a casted pointer to *object*'s instance structure for *type_id*. Note that matching includes super−classes of *object*.
- G_TYPE_CHECK_CLASS_CAST(*klass, type_id, class_name*) Same as the preceding, but for class structure pointers.

Note *name* and *class_name* are the type names from the C structure definitions. Make sure that you understand that these are very different from *type_id*, a runtime identifier for the entire class.
The preceding macros provide warnings when you attempt to make invalid casts.

- G_TYPE_CHECK_INSTANCE_TYPE(*object, type_id*)

  Returns TRUE if *object* belongs to *type_id*'s class.
- G_TYPE_CHECK_CLASS_TYPE(*klass, type_id*)

  Returns TRUE if *klass* belongs to *type_id*'s class.
- G_TYPE_INSTANCE_GET_CLASS(*object, klass*)

  Returns the class structure for object, casted as the C type class_name.

Note These macros use the parameter *klass* instead of *class*. That's because *class* is a reserved C++ keyword; if you tried to use one of these macros in C++ source code, your program would not compile. The GLib, GTK+, and GNOME source distributions reflect this. However, with parameters to functions in regular C code, this is not a problem, because you wouldn't use a C++ compiler on that. You can always use *class* if you never plan to use C++, or if you just feel like being mean to C++ people.
To keep C++ compatibility at link time, use G_BEGIN_DECLS and G_END_DECLS to encapsulate your C code:

```
G_BEGIN_DECLS

  << header code >>

G_END_DECLS
```

2.2.3 Initializing Type Identifiers

The previous section made many references to the GObject class type identifier, with the macro TYPE_MEDIA expanding to a media_get_type() call to return the **Media** class type identifier. As mentioned before, the type identifier is an actual piece of runtime data that conforms to the GType standards.

This is perhaps easier to understand when you look at the actual definition of media_get_type():

```
GType media_get_type(void)
{
  static GType media_type = 0;

  if (!media_type)
  {
     static const GTypeInfo media_info = {
        sizeof(MediaClass),                /* class structure size */
        NULL,                              /* base class initializer */
        NULL,                              /* base class finalizer */
        (GClassInitFunc)media_class_init,  /* class initializer */
        NULL,                              /* class finalizer */
        NULL,                              /* class data */
        sizeof(Media),                     /* instance structure size */
        16,                                /* preallocated instances */
        NULL,                              /* instance initializer */
        NULL                               /* function table */
     };

     media_type = g_type_register_static(
            G_TYPE_OBJECT,                 /* parent class */
            "Media",                       /* type name */
            &media_info,                   /* GTypeInfo struct (above) */
            0);                            /* flags */
  }

  return media_type;
}
```

The return value of media_get_type() is the C type GType   a type of a type (if that makes any sense). This function returns the same class type identifier every time, because the variable used to store and return the class identifier has a static declaration (media_type). Because of the if statement, media_get_type() does some work to initialize media_type upon first invocation, but needs to return the value of this variable only on subsequent calls.

media_type gets its actual value from

`g_type_register_static(`*`parent_id, name, type_info, options`*`)`

These arguments are as follows:

- *parent_id*: The parent type identifier. In this case, it's the **GObject** base class (G_TYPE_OBJECT).

- *name*: A name (a string) for the type.
- *type_info*: A GTypeInfo structure containing class details (see the following paragraphs).
- *options*: Option flags (using bitwise OR), or zero if you don't need any. For example, G_TYPE_FLAG_ABSTRACT indicates an abstract class.

Warning The type name must be at least three characters long.
The fields of the GTypeInfo structure are as follows (in this order):

- class_size (guint16): Size of the class structure. In this case, it's easy: sizeof(MediaClass).
- base_init (GBaseInitFunc): The base class initializer. When you create a new class, GObject calls the base initializers of all classes in the hierarchy chain leading up from the class. This is normally necessary only if there is some dynamically allocated part of the class (*not* object) that GObject must copy to a derived class, and therefore, you need it only when your class has such data and is the parent of some other class. Normally, it isn't necessary, and you can use NULL.
- base_finalize (GBaseFinalizeFunc): The base class finalizer; essentially, the reverse operation of the base class initializer. You almost certainly need it when you have a base class initializer; otherwise, use NULL.
- class_init (GClassInitFunc): The class initializer. This function prepares class variables, in particular, properties (see Sections 2.4) and signals (Section 2.6). Note that this is not the constructor for the objects; however, the property definitions in the class initializer set up many of the defaults for the constructor. The class initializer appears throughout this chapter; it is instrumental in figuring out how classes work.
- class_finalize (GClassFinalizeFunc): The class finalizer. You normally won't need to do anything when GObject gets rid of a class, unless you have a very complicated mechanism for saving the state over program invocations.

- class_data (gconstpointer): GObject passes the class data to the class initializer and class finalizer. One example of a situation in which you might use this function is when you want to use the same initializer for several classes. Of course, that could make things even more complicated than they already are.
- instance_size (guint16): The size of the instance structure is an allocated object structure's memory requirement; sizeof(Media) shown earlier in the code is typical.
- n_preallocs (guint16): The number of instances to preallocate depends on the number of instances that you think a typical program will use. If you specify 0, GObject allocates memory when it needs to. Be careful when using this function, because you can waste a lot of memory if you go nuts with many different kinds of classes.
- instance_init (GInstanceInitFunc): The instance initializer runs every time you create a new object of the class. It's not strictly necessary, because the constructor is also at your disposal. This field is present because it is a part of GType; GObject doesn't need it.
- value_table (const GTypeValueTable *): This function table can map certain kinds of values to functions, and it is really important only if you're doing something very involved with the type system, such as creating your own object system. This book does not cover that topic, and it's probably best if you forget that this function even exists.

Here are the type definitions for the various function prototypes you just saw:

```
typedef void (*GBaseInitFunc)      (gpointer g_class);
typedef void (*GBaseFinalizeFunc)  (gpointer g_class);
typedef void (*GClassInitFunc)     (gpointer g_class, gpointer class_data);
typedef void (*GClassFinalizeFunc) (gpointer g_class, gpointer class_data);
typedef void (*GInstanceInitFunc)  (GTypeInstance *instance, gpointer g_class);
```

Note You can get rid of the *GTypeInfo* structure as soon as you're finished with it *g_type_register_static()* makes a complete copy for itself. So in the media example, you could free up any dynamically allocated parts of *media_info* that you like.

Try not to get tangled up in the initializers and finalizers. Normally, you won't be dealing with dynamically allocated data in the classes themselves, so you won't need anything except a class initializer; you'll see that in Section 2.4.1. You should not need to bother with the instance initializer, because you can use properties.

2.2.4 The Base Class: GObject

This section illustrates the makeup of the base class, **GObject**. All GObject classes inherit from this class. Not only do you need some of the utility macros to create new classes, but it helps to know the methods that you'll come across later.

Macros

In Section 2.2.2, you defined several utility macros for the **Media** class. Here are the **GObject** versions:

- G_TYPE_OBJECT returns **GObject**'s type identifier. Don't confuse this with G_OBJECT_TYPE.
- G_OBJECT(*object*) casts *object* to the GObject instance structure.
- G_OBJECT_CLASS(*klass*) casts an object class structure *klass* to the GObjectClass class structure.
- G_IS_OBJECT(*object*) returns TRUE if the *object* parameter is an instance of a **GObject**. This should return TRUE for any object that you define with GObject, unless you're very daring and decide to make your own base object.
- G_IS_OBJECT_CLASS(*klass*) returns TRUE if *klass* is a class structure. It should return TRUE for any class structure within the GObject system.
- G_OBJECT_GET_CLASS(*object*) returns the class structure (GObjectClass) corresponding to any instance structure.

Of these, you will encounter G_TYPE_OBJECT and G_OBJECT() the most: G_TYPE_OBJECT when you need to know the type identifier of **GObject** (for instance, when defining a class like **Media**), and G_OBJECT() when you need to pass an object instance as a GObject instance to a function.

Note Many common functions that take **GObject** parameters don't require class type casts, but rather just expect an untyped *gpointer* pointer and do the casting work on their own. Although this approach isn't terribly consistent, it can save you an awful lot of typing. The functions that do this include *g_object_get(), g_object_set(), g_object_ref(), g_object_unref()*, and the entire *g_signal_connect()* family.

Base Class Methods

Here is the part of the **GObject** class structure that contains public methods and a signal handler prototype. You will see some of these time and again. (This definition comes straight from the *gobject.h* header file.)

```
typedef struct _GObjectClass GObjectClass;
typedef struct _GObjectConstructParam GObjectConstructParam;

  << ... >>

struct _GObjectClass
{
  GTypeClass g_type_class;
  << ... >>

  /* public overridable methods */
  GObject* (*constructor) (GType type,
                           guint n_construct_properties,
                           GObjectConstructParam *construct_properties);

  void (*set_property) (GObject *object,
                        guint property_id,
                        const GValue *value,
                        GParamSpec *pspec);

  void (*get_property) (GObject *object,
                        guint property_id,
                        GValue *value,
                        GParamSpec *pspec);

  void (*dispose) (GObject *object);

  void (*finalize) (GObject *object);

  << ... >>

  /* signals/handlers */
  void (*notify) (GObject *object, GParamSpec *pspec);

  << ... >>

};

struct _GObjectConstructParam
{
  GParamSpec *pspec;
  GValue     *value;
};
```

The methods are as follows:

- constructor: This is the object's constructor, called when you create an instance of a class. It takes a type identifier parameter (type), the number of properties to create (n_construct_properties), and an array of property descriptions (construct_properties). You'll read about properties and the GValues that they employ in Section 2.4.

  The constructor creates the object and initializes its data. If you want to create your own constructor, you should always run the constructor of the parent class first and then extend the resulting parent class object by yourself, so that your object isn't missing anything. Keep in mind, though, that you do not usually need to make your own constructors; there is a further example of inheritance in Section 2.7 that provides an alternative.
- set_property: Writer function for properties.
- get_property: Reader function for properties.
- dispose: The destructor; GObject calls this when removing an object that is no longer in use (see Section 2.5.2). Destructors take a single parameter: the object to destroy. Destructors clean up after signal handlers and sort out other internal matters.
- finalize: GObject calls the finalizer when an object's reference count goes to zero (again, see Section 2.5.2), before it gets around to calling the destructor. Use this for housekeeping functions that require immediate attention, such as removing dynamically allocated memory. You should always call an

object's parent finalizer at the end of your own finalizers. After running the finalizer, the object is officially dead, and you should do nothing more with the data.
- notify: GObject calls this special method for property change signals (see Section 2.6). There is no reasonable default, and you should not touch this (that is, inherit notify from the parent).

[1] It might be some consolation that C++ and most scripting languages have GNOME bindings.

2.3 Methods

Although you haven't seen the class initializer for **Media** yet, you can now see how to define a simple method. Here are some important things to remember about methods:

- Methods usually do *not* appear in class structure. Instead, method prototypes usually appear somewhere soon after the class structure.
- A method's name should reflect the class name (for example, media_*() for **Media**).
- A method's first parameter is always an object (a structure of the instance class). Any remaining parameters are up to you.
- In public methods, always check that the first parameter is actually a valid object of the method's class.
- In addition, cast this object parameter after you do the check, because the object you get could be in a subclass.
- Be careful about setting an object's attributes. Standard GTK+/GNOME practice dictates that all attributes are properties (see Section 2.4); use that system for setting attributes.

These considerations sound like a lot of fuss, but this example shows that it doesn't amount to much:

```
void media_print_inv_nr(Media *object)
{
  Media *media;

  g_return_if_fail(IS_MEDIA(object));

  media = MEDIA(object);
  g_print("Inventory number: %d\n", media->inv_nr);
}
```

Most public methods contain everything here but the last line (g_print(...);).

2.4 Properties

You should set and retrieve attribute data on your GObject instances using the ***property*** system so that other programmers can get at data through a uniform interface, rather than going through a series of custom access functions.

Properties have names and descriptions, so they are self−documenting to a certain extent. In addition, exposing a GObject's data with properties allows you to employ an object design tool (such as Glade, see Chapter 5).

You'll encounter properties *ad infinitum* in GTK+ when you read Chapter 3, primarily when manipulating widget settings. [2]

2.4.1 Declaring Parameters

You must define each property in a class as a GObject *parameter*. To get started, you should obtain a GParamSpec structure in your class initialization function (described in Section 2.2.3).

You'll need the following information to create a GParamSpec structure:

- An **identifier**. A short string will do, such as **inventory−id**.
- A **nickname**. The full name of the parameter, such as inventory number.
- A **description**. A concise explanation, such as number on the inventory label.
- **Options**, such as read−write access.
- Type−specific information, such as
    - ♦ Minimum value
    - ♦ Maximum value
    - ♦ Default value
    - ♦ A secondary type if you're encapsulating parameters (for example, G_TYPE_BOXED, G_TYPE_ENUM, G_TYPE_FLAGS, G_TYPE_OBJECT, G_TYPE_PARAM).
    - ♦ Sizes of arrays.

Because it is somewhat complicated, you don't create a GParamSpec structure by hand. Instead, use one of the g_param_spec_ functions in the following table to allocate the structure and set its fields.

Function	**Type**
g_param_spec_boolean()	gboolean
g_param_spec_boxed()	GBoxed
g_param_spec_char()	gchar
g_param_spec_double()	gdouble
g_param_spec_enum()	GEnumClass, GEnumValue
g_param_spec_flags()	GFlagsClass
g_param_spec_float()	gfloat
g_param_spec_int()	gint
g_param_spec_int64()	gint64
g_param_spec_long()	glong
g_param_spec_object()	GObject
g_param_spec_param()	GParamSpec
g_param_spec_pointer()	gpointer
g_param_spec_string()	gchar *
g_param_spec_uchar()	guchar
g_param_spec_uint()	guint
g_param_spec_uint64()	guint64
g_param_spec_unichar()	gunichar

g_param_spec_value_array()	Array of some other type

Typically, you place the call to a g_param_spec_ function in your class initializer function. You may recall that the class initializer for the **Media** example is media_class_init(). Therefore, using parameters for **inventory−id** and **orig−package**, media_class_init() would look something like this:

```
static void media_class_init(MediaClass *class)
{
  GParamSpec *inv_nr_param;
  GParamSpec *orig_package_param;

  << ... >>

  /* create GParamSpec descriptions for properties */
  inv_nr_param = g_param_spec_uint("inventory−id",     /* identifier */
                                   "inventory number", /* nickname */
                                   "number on inventory label",
                                                       /* description */
                                   0,                  /* minimum */
                                   UINT_MAX,           /* maximum */
                                   0,                  /* default */
                                   G_PARAM_READWRITE); /* flags */

  orig_package_param = g_param_spec_boolean("orig−package",
                                            "original package?",
                                            "is item in its original package?",
                                            FALSE,
                                            G_PARAM_READWRITE);

  << ... >>

}
```

Although the actual inv_nr and orig_package fields from the Media instance structure aren't in this function, you will need to come back to them when you actually install the property in the class. The GParamSpec structure serves to *describe* the property: its purpose, type, and permissible values.

The last parameter to a g_param_spec_ function is a bit mask that you can specify with a bitwise OR of any of the following:

- G_PARAM_CONSTRUCT indicates that GObject will assign a value to the property upon object instantiation. At the moment, this works only in conjunction with G_PARAM_CONSTRUCT.

- G_PARAM_CONSTRUCT_ONLY indicates that the property may take on a value *only* when an object is instantiated.
- G_PARAM_LAX_VALIDATION disables type checks when you write to this property. Set this only if you know exactly what you're doing.
- G_PARAM_READABLE allows read access to the property.
- G_PARAM_WRITABLE allows write access to the property.
- G_PARAM_READWRITE is shorthand for G_PARAM_READABLE|G_PARAM_WRITABLE.

Note As this chapter progresses, the *media_class_init()* function code in this section will grow.

2.4.2 Tangent: Generic Containers for Values

Before you get to Section 2.4.3, where you see how to activate a property in the class initializer, you need to familiarize yourself with how GObject moves the property values from place to place.

Untyped gpointer−style pointers normally take on the task of setting and retrieving function parameters of an arbitrary type. However, you need to store and check the type information at run time so that you don't try to do something disastrous, like attempting to copy a string into a memory location that corresponds to a random integer.

GObject has a mechanism for holding a value along with its type into a single "container," so that you can pass the container along as a parameter and pull its value and type out when necessary. This system is called GValue, with function names that start with g_value_.

The actual container is the GValue data structure. If you need to create one, you can do it with GLib's elementary memory management utilities; there aren't any special functions just for GValues.

Warning That said, you must use *g_malloc0()*, *g_new0()*, or some other similar allocation function that sets all of the new memory bytes to zero. You'll get an error when you try to initialize a *GValue* structure with random bytes.

After creating a GValue structure gvalue, initialize it with

```
g_value_init(gvalue, type)
```

where *type* is an identifier such as G_TYPE_INT (see page 78 for a full list). For each type identifier, the following are available:

- A verification macro, G_VALUE_HOLDS_*TYPE*(*gvalue*), that returns TRUE when *gvalue* contains the type.
- A writer function, g_value_set_*type*(*gvalue*, *value*), to place *value* into the *gvalue* container.
- A reader function, g_value_get_*type*(*gvalue*), to fetch the value from *gvalue*.

Note If you can't decide on how often to verify the type inside a container, err on doing it too often instead of one time too few when you go to read or write a value.

The standard types are listed below. Here is a small GValue demonstration:

```
/* gvaluedemo.c -- demonstrate GValue */

#include <glib-object.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  GValue *value;

  g_type_init(); /* initialize type system */

  /* allocate new GValue, zero out contents */
  value = g_new0(GValue, 1);

  /* initialize GValue type to gint */
  g_value_init(value, G_TYPE_INT);
  /* set the value to 42 and read it out again */
  g_value_set_int(value, 42);
  g_print("Value: %d\n", g_value_get_int(value));

  /* is the type of the GValue a GObject? */
  if (G_VALUE_HOLDS_OBJECT(value))
  {
     g_print("Container holds a GObject\n");
  } else {
     g_print("Container does not hold a GObject\n");
  }
  /* expect "Container does not hold a GObject" */

  g_free(value);

  return 0;
}
```

There are two special access functions for GValue structures of G_TYPE_STRING: g_value_set_static_string(*gvalue*, *str*) transfers a string pointer *str* into a *gvalue*, and g_value_dup_string(*gvalue*) returns a copy of a string in *gvalue* (you will need to deallocate that copy when you're done with it, though).

Otherwise, the names are uniform. For each of the types in the following list, there is a verification macro and a reader and writer access function, as described earlier. For example, G_TYPE_CHAR comes with G_VALUE_HOLDS_CHAR(), g_value_get_char(), and g_value_set_char().

G_TYPE_BOOLEAN	G_TYPE_FLOAT	G_TYPE_POINTER
G_TYPE_BOXED	G_TYPE_INT	G_TYPE_STRING
G_TYPE_CHAR	G_TYPE_INT64	G_TYPE_UCHAR
G_TYPE_DOUBLE	G_TYPE_LONG	G_TYPE_UINT
G_TYPE_ENUM	G_TYPE_OBJECT	G_TYPE_UINT64
G_TYPE_FLAGS	G_TYPE_PARAM	G_TYPE_ULONG

It might be prudent to note that GObject also defines a gchararray type that stands for gchar *. The advantage of using gchararray over a simple gchar pointer is in name only; when you have a gchararray array, you can be certain that it's a string and not some other type that you've cast to gchar *.

To reset a GValue to its original state   that is, the zeroed memory that you had just before you ran g_value_init()   use g_value_unset(gvalue). This frees up any extra memory that gvalue is currently using and sets its bytes in memory to zero. At that point, it's ready to be used again.

2.4.3 Installing Properties

Now you're ready to install some properties. Recall that in Section 2.4.1 you came up with the GParamSpec structures for the *Media* class properties in the media_class_init() function. The property installation continues in that function, with a call to

```
g_object_class_install_property(class, id, param)
```

where *class* is the class structure, *param* is the property's GParamSpec structure, and *id* is a unique identifier obtained with an enumeration type. This identifier should begin with PROP_.

Warning The property identifier must be greater than zero, so you will need to place a dummy value like *PROP_MEDIA_0* at the head of your enumeration type.

The code for media_class_init() should make things clear:

```
enum {
  PROP_MEDIA_0,
  PROP_INV_NR,
  PROP_ORIG_PACKAGE
};

static void media_class_init(MediaClass *class)
{
  GParamSpec *inv_nr_param;
  GParamSpec *orig_package_param;
  GObjectClass *g_object_class;

  /* get handle to base object */
  g_object_class = G_OBJECT_CLASS(class);

  << param structure setup from Section 2.4.1 >>

  /* override base object methods */
  g_object_class->set_property = media_set_property;
  g_object_class->get_property = media_get_property;
```

Warning Notice that you must set the *set_property* and *get_property* methods before installing the class properties.

```
  << ... >>

  /* install properties */
  g_object_class_install_property(g_object_class,
                                  PROP_INV_NR,
                                  inv_nr_param);

  g_object_class_install_property(g_object_class,
                                  PROP_ORIG_PACKAGE,
                                  orig_package_param);

  << ... >>
}
```

Note The code in Section 2.4.1 and here accesses *GParamSpec* structures with the temporary variables *inv_nr_param* and *orig_package_param* before installation with *g_object_class_install_property()*. Programmers normally omit these temporary variables, using the entire call to *g_param_spec_boolean()* as a parameter when they install the property.

To use the property system in the new class, media_class_init() must cast the new **Media** class structure (class) pointer to a **GObject** class structure named g_object_class. When you get a handle from a cast like this, you can override the set_property and get_property methods in the **GObject** base class. This is exactly what you must do to install the new properties such as **inventory−id** and **orig−package**. Remember that the base object knows nothing about the properties and instance structures of its subclasses.

The example overrides the base class methods with media_set_property() and media_get_property(), and therefore, you must supply prototypes before media_class_init(). These are straightforward, and because they are replacements for methods in the base class structure, they must conform to the prototypes in the base class structure:

```
static void media_set_property(GObject *object,
                               guint prop_id,
                               const GValue *value,
                               GParamSpec *pspec);

static void media_get_property(GObject *object,
                               guint prop_id,
                               GValue *value,
                               GParamSpec *pspec);
```

Note You can avoid prototypes for static functions by placing the actual functions (described below) before *media_class_init()*.

Now that you've set up all of this infrastructure to handle the properties, you can write the functions that actually deal with the inv_nr and orig_package fields in the instance structure. The implementations consist of some busywork; to set a field, media_set_property() does the following:

1. Determines the property to set.
2. Removes the new property value from the GValue container from its parameter list.
3. Sets the field in the instance structure (*finally!*).

Here is the actual code:

```
static void media_set_property(GObject *object,
                               guint prop_id,
                               const GValue *value,
                               GParamSpec *pspec)
{
  Media *media;
  guint new_inv_nr;
  gboolean new_orig_package;

  media = MEDIA(object);

  switch(prop_id)
  {
     case PROP_INV_NR:
        new_inv_nr = g_value_get_uint(value);
        if (media->inv_nr != new_inv_nr)
        {
           media->inv_nr = new_inv_nr;
        }
        break;

     case PROP_ORIG_PACKAGE:
        new_orig_package = g_value_get_boolean(value);
        if (media->orig_package != new_orig_package)
        {
           media->orig_package = new_orig_package;
        }
        break;

     default:
        G_OBJECT_WARN_INVALID_PROPERTY_ID(object, prop_id, pspec);
        break;
  }
}
```

The media_get_property() code is similar, except that it needs to put one of the instance structure fields into the container rather than the other way around. No notification is necessary. (It would be very nosy of your program to tell the object every time someone looked at a property, but you can do it if you really want to.)

```
static void media_get_property(GObject *object,
                               guint prop_id,
                               GValue *value,
                               GParamSpec *pspec)
  {
   Media *media;

   media = MEDIA(object);

   switch(prop_id)
   {
      case PROP_INV_NR:
         g_value_set_uint(value, media->inv_nr);
         break;

      case PROP_ORIG_PACKAGE:
         g_value_set_boolean(value, media->orig_package);
         break;

      default:
         G_OBJECT_WARN_INVALID_PROPERTY_ID(object, prop_id, pspec);
      break;
  }
}
```

Take a close look at the default case in each of these functions. The default comes into play only when the function encounters an invalid property, and it runs G_OBJECT_WARN_INVALID_PROPERTY_ID() on the object, invalid property, and parameter structure. The resulting warning message will hopefully be strong enough to get you to check your property access function calls.

Why Properties?

You may be wondering why you need such a complicated system just to set a bunch of fields in your instance structure. After all, you could write access methods to do this, or you could even just tell the programmer to set the fields. However, you want a uniform system such as properties for the following reasons:

- You need a *dynamic* system. Subclasses can add their own properties with little effort, as you will see in Section 2.7.
- You want to define behavior for when properties change. This capability is extremely important in GUI programming, where you want the program to react to changes in buttons, check boxes, and other elements. If you haphazardly set instance structure fields instead of using properties, you would need to define a "reaction" function and make sure that any code that sets a field also calls that function. Even with access methods, this can get out of hand very quickly, especially if your "reaction" function needs to set other fields.
- You want a system that's easy to document. It's easy to list property names with their possible values and descriptions. Access method APIs are considerably harder to describe, especially if the methods aren't uniform.

But What About Those Access Methods That I Keep Seeing?

In practice, you might see access methods for a class that correspond to properties in the class, especially with older code. Functionally, there is no difference; it's just one more layer of indirection.

Let's say that you have this to set the **Media orig−package** property:

```
void media_set_orig_package(Media *object, gboolean new_value)
{
  Media *media;
  g_return_if_fail(IS_MEDIA(object));
  media = MEDIA(object);

  if (media−>orig_package != new_value)
  {
     media−>orig_package = new_value;
     g_object_notify(G_OBJECT(media), "orig−package");
  }
}
```

This does all of the work that you see under case PROP_ORIG_PACKAGE: in media_set_property() from page 81. Therefore, you can rewrite that part:

```
    case PROP_ORIG_PACKAGE:
       new_orig_package = g_value_set_boolean(value);
       media_set_orig_package(media, new_orig_package);
       break;
```

This is primarily a matter of convention and a function of the age of the code. When reading API documentation, you may see an object with more properties than access function pairs, indicating that someone may have added more properties to some older code without bothering to use access functions (developers don't usually remove access functions for fear of breaking third−party applications). In that case, *class*_set_*property*() contains a mix of access function calls and field assignments/notifications.

Direct access function calls are slightly faster because they do not have to look up the property identifier and encapsulate any values. However, this speedup usually doesn't matter.

[2]Unfortunately, some GTK+ classes do not have property interfaces; hopefully, that's not a permanent situation.

2.5 Using Objects

So far, you have seen a quite a bit of preparatory work with objects. This work is admittedly complex, and you are probably hoping that *using* the objects is easier. Thankfully, it is.

To create an object, use g_object_new(*type*, ..., NULL), where *type* is the object's class type identifier. For example, to create a **Media** object with the default property values, use

```
Media *media;

media = g_object_new(TYPE_MEDIA, NULL);
```

If you want to set some of the properties when you create the object, try something like the following instead:

```
/* create an object, setting some properties */
media = g_object_new(TYPE_MEDIA,
                     "inventory−id", 42,
                     "orig−package", FALSE,
                     NULL);
```

Note The property list always ends with *NULL*.
If you decide to follow tradition and write a generator function to create an object for you, conventions dictate that the generator name should be type_new(): for example, media_new(). You can always define a macro for this, but be careful with the variable arguments; not all C preprocessors support variable arguments.

If you did a tidy job of programming your classes   in particular, if every attribute in your instance structure corresponds to a property, with the appropriate code in your initialization functions   you don't need a generator function. If you stick to g_object_new(), you can specify properties in any order that you wish, easily add and delete properties, and in general, will not need to remember an additional set of function names. Furthermore, if you avoid generator functions, you will have a much easier time creating bindings for other programming languages.

2.5.1 Using Properties

Now that you know how to create an object and initialize its properties, you probably want to know how to set and retrieve those properties. The two functions for this are g_object_set() and g_object_get(); their parameters are very similar to those of g_object_new(), as you can see in this example:

```
guint nr;
gboolean is_unpacked;

<< ... >>

/* set new values */
g_print("Setting inventory−id = 37, orig−package = TRUE\n");
g_object_set(media,
             "orig−package", TRUE,
             "inventory−id", 37,
             NULL);

/* double−check those new values */
g_print("Verifying.. ");
g_object_get(media,
             "orig−package", &is_unpacked,
             "inventory−id", &nr,
             NULL);
g_print("inventory−id = %d, orig−package = %s\n",
        nr, is_unpacked ? "TRUE" : "FALSE");
```

To retrieve parameters with g_object_get(), you need to specify the address of the target memory location; g_object_get() fills the memory with the new values. Examples in the preceding code are &is_unpacked and &nr.

Note

> Be careful not to mix up your types when accessing properties. That is, don't code something like *g_object_get(media, "my−double", &my_int)*. You may also need to cast certain constants when using them in conjunction with *g_object_set()*. This is one area where access functions may be a somewhat preferable option.

Here are a few more functions that work with properties:

- g_object_set_property(*object*, *name*, *value*)

  Sets a single property *name* in *object* to *value*.
- g_object_get_property(*object*, *name*, *addr*)

  Stores the property *name* from *object* in the memory at *addr*.
- g_object_set_valist(*object*, *name*, *varargs*)

  Like g_object_set_property(), but operates with variable arguments.
- g_object_get_valist(*object*, *name*, *varargs*)

  Same as the preceding function, but retrieves the arguments instead of storing them.

2.5.2 Strong and Weak Object References

GObject uses a reference count to keep track of its objects. When you call g_object_new(), you're actually getting a reference to the object, and GObject makes a note with the reference count. If you want another reference to the same object, use g_object_ref(*object*) to return another reference.

To remove a reference, call g_object_unref(*object*). All references to a single object are equal; there is no special treatment for an original reference, and as mentioned before, GObject removes an object when the reference count goes to zero.

Such references are sometimes known as *strong* references, because they determine when GObject destroys an object. However, there are also *weak* references that can be present when GObject performs its garbage collection. GObject manages these references to a certain extent; it has a list of the object pointers in memory. To create a weak reference, use

```
g_object_add_weak_pointer(object, weak_ptr_addr)
```

Here, *object* is a casted existing reference to an object, and *weak_ptr_addr* is the *address* of the new weak pointer. If GObject removes the object behind the weak reference, it sets the weak reference to NULL. Use g_object_remove_weak_pointer() with the same arguments to remove a weak pointer.

Note You still must assign the weak pointer by hand.

Here are some examples of how to use references that build on the examples in the previous section:

```
Media *media2, *media_weak;

  << ... >>

media_weak = NULL;

/* set media2 to a strong reference from media */
media2 = g_object_ref(media);
```

```
/* set media_weak to a weak reference to media2 */
media_weak = media2;
g_object_add_weak_pointer(G_OBJECT(media2), (gpointer) &media_weak);

/* remove one strong reference */
g_object_unref(media2);

/* see if media_weak is NULL, meaning that object is gone */
if (media_weak == NULL)
{
   g_print("media_weak is NULL; object is gone\n");
} else {
   g_print("media_weak is not NULL; object is still in memory\n");
}

/* remove another reference */
g_object_unref(media);

/* check the weak pointer again */
if (media_weak == NULL)
{
   g_print("media_weak is NULL; object is gone\n");
}
```

Don't confuse g_object_*_weak_pointer() with g_object_weak_*ref(). The latter function enables you to call a notification function when the GObject is destroyed, but will not be covered in this book. The weak pointer functions here just set the object pointer to NULL.

2.6 Signals

Signals are events that can happen to an object during the course of the object's life. Signals serve as a means of communication between objects; when an object gets a signal, it can react in some way. The signal system in GObject is called GSignal.

Note In written language, a signal can be the act of sending a message or the message itself. In GObject, a signal is the sender's version of the message. A GObject signal has one source and multiple potential recipients.

A signal **handler** is a callback function with a prototype that you declare when you initialize a class (for example, media_class_init() from Section 2.4.1). When some other part of a program **emits** a signal, GSignal calls the handler on the object. You can alter the characteristics of an object with a signal handler. You can specify the order of the handler invocations on a per−signal basis.

Note You add signal handlers on a per−object basis. When you add a handler to one object, it does not apply for any other objects in the class.

You can pass parameters along with signals and receive return values from handlers with the ***marshalling*** [3] mechanism. Each signal usually has a function pointer in an object's class structure. The function behind this pointer is called the *default handler*. You can link additional functions into a signal, so that if there is more than one return value per signal emission, the object may process all of the return values back with the help of an ***accumulator***. Otherwise, GSignal returns the value of only the last signal handler.

2.6.1 Defining Signals and Installing Handlers

A signal isn't a data structure; it's just a guint identifier managed by GObject. You should define some names for these identifiers with an enumeration type (for caching purposes later); a good place to do this is somewhere before your class initialization function (for example, media_class_init()). Here is an example for two signals: one that removes an item from its package, and another that throws out an item.

```
/* Signal indices */
enum {
  UNPACKED,
  THROW_OUT,
  LAST_SIGNAL
};
```

As you can see, these names correspond to indices. Furthermore, common practice says that you should create a cache array as a static variable. Later, you will set each array element to a GObject signal identifier.

```
/* Signal identifier map */
static guint media_signal[LAST_SIGNAL] = {0, 0};
```

Note how LAST_SIGNAL indicates the size of the array.

Now you need to think about the signal handlers. You may recall from Section 2.2.1 that you already provided some infrastructure in the MediaClass class structure for signal handlers: function pointer fields called unpacked and throw_out. The actual functions that correspond to these are media_unpacked() and media_throw_out(), so you need to provide their prototypes:

```
/* Prototypes for signal handlers */
static void media_unpacked(Media *media);
static void media_throw_out(Media *media, gboolean permanent);
```

With this, you are close to completing the media_class_init() function first started in Section 2.4.1. Continue by setting the function pointers in the class structure to the actual signal handlers as shown on the next page:

```
/* Initialize the Media class */
static void media_class_init(MediaClass *class)
{
  GObjectClass *g_object_class;

  << parameter/property code >>

  /* set signal handlers */
  class->unpacked = media_unpacked;
  class->throw_out = media_throw_out;
```

Then install the **unpacked** signal and its default handler with g_signal_new():

```
  /* install signals and default handlers */
  media_signal[UNPACKED] =
     g_signal_new("unpacked",                     /* name */
                  TYPE_MEDIA,                     /* class type identifier */
                  G_SIGNAL_RUN_LAST|G_SIGNAL_DETAILED, /* options */
                  G_STRUCT_OFFSET(MediaClass, unpacked), /* handler offset */
                  NULL,                           /* accumulator function */
                  NULL,                           /* accumulator data */
```

```
                g_cclosure_marshal_VOID__VOID,  /* marshaller */
                G_TYPE_NONE,                    /* type of return value */
                0);
```

That's a mouthful, to say the least, so here's how the code breaks down:

- The **return value** of g_signal_new() is GObject's identifier. You should store it in the mapping array from earlier.
- The **name** is a short string to identify the signal.
- The **type identifier** is the GObject class type identifier macro.
- **Options** may include one or more of the following as a bitwise OR:
    - G_SIGNAL_DETAILED: The signal supports details (see Section 2.6.6).
    - G_SIGNAL_NO_HOOKS: You may not use emission hooks with the signal (see Section 2.6.7).
    - G_SIGNAL_NO_RECURSE: If GSignal gets another signal emission for this signal handler when the handler is still active, the signal handler restarts   it does not call the signal handler from within the signal handler.
    - G_SIGNAL_RUN_FIRST: Signal emission has several stages. This flag indicates that the handler should run in the first stage (see Section 2.6.2).
    - G_SIGNAL_RUN_LAST: The signal handler runs in the third stage (use this if you're not sure what to do).
    - G_SIGNAL_RUN_CLEANUP: The signal handler runs in the last stage.
    - G_SIGNAL_ACTION: If some code emits an **action signal**, it doesn't need to do any extra housecleaning around the target object. You can use this to interconnect code from different sources.
- The **offset** is an ugly way to tell g_signal_new() where the class signal handler function is. It is an offset from the memory location of the class structure, and luckily, you have the G_STRUCT_OFFSET macro to do the work for you.
- The **accumulator** is a callback function that collects various return values (see Section 2.6.4).
- The **accumulator data** is where to put the accumulator's data.
- The C **Marshaller** for the signal is described in Section 2.6.3.
- The **return value** is the return value of the marshaller.
- The **number of parameters** specifies how many extra parameters to pass along with the marshaller. If this number is greater than zero, you must specify the GValue types (see the **throw_out** example that follows).

Having learned all of this, you can install the signal and default handler for **throw_out** (the difference is that **throw_out** takes a gboolean parameter) and *finally* put media_class_init() to rest.

```
  media_signal[THROW_OUT] =
     g_signal_new("throw_out",
                  TYPE_MEDIA,
                  G_SIGNAL_RUN_LAST|G_SIGNAL_DETAILED,
                  G_STRUCT_OFFSET(MediaClass, throw_out),
                  NULL, NULL,
                  g_cclosure_marshal_VOID__BOOLEAN,
                  G_TYPE_NONE,
                  1,
                  G_TYPE_BOOLEAN);
```

The signal handlers are fairly simple:

```
/* unpacked signal handler */
static void media_unpacked(Media *media)
{
  if (media->orig_package)
  {
     g_object_set(media, "orig-package", FALSE, NULL);
     g_print("Media unpacked.\n");
  } else {
     g_print("Media already unpacked.\n");
  }
}

/* throw_out signal handler */
static void media_throw_out(Media *media, gboolean permanent)
{
  if (permanent)
  {
     g_print("Trashing media.\n");
  } else {
     g_print("Media not in the dumpster quite yet.\n");
  }
}
```

Notice the additional parameter to media_throw_out(), and that these functions have no return values.

2.6.2 Emitting Signals

One way to emit a signal is with

```
g_signal_emit_by_name(object, name [, parms ..] [, return])
```

The arguments are as follows:

- *object* (gpointer): The target object.
- *name* (const gchar *): The signal identifier (for example, "unpacked").
- *parms*: Signal handler parameters (if any).
- *return*: Location of return value (if any).

Therefore, if you have a signal with a signature of VOID:VOID, you need only two parameters; otherwise, you need at least three. Here are some examples with the signals defined in Section 2.6.1:

```
g_signal_emit_by_name(media, "unpacked");
/* expect "Media unpacked." */

g_signal_emit_by_name(media, "unpacked");
/* expect "Media already unpacked." */

g_signal_emit_by_name(media, "throw-out", TRUE);
/* expect "Trashing media." */
```

Many programmers prefer to emit signals based on the numeric signal identifier to avoid a lookup on a string. This function is the manual equivalent of g_signal_emit_by_name():

```
g_signal_emit(gpointer object, guint signal_id, GQuark detail, ...)
```

You should use this function in conjunction with cached signal identifiers. Recall from Section 2.6.1 that media_signal[] holds the cache for the ongoing media example. Therefore, this example sends the **unpacked** signal to media:

```
g_signal_emit(media, media_signal[UNPACKED], 0);
```

Note This book primarily uses *g_signal_emit_by_name()* because it requires less coding baggage. However, if you continuously emit signals, you should consider caching the signal identifiers as described above.

If you set the **throw−out** handlers to return gboolean, the following code would retrieve that value and place it into a return_val variable:

```
gboolean return_val;

  << ... >>
g_signal_emit_by_name(media, "throw−out", TRUE, &return_val);

if (return_val)
{
   g_print("Signal (throw−out): returned TRUE.\n");
} else {
   g_print("Signal (throw−out): returned FALSE.\n");
}
```

When you emit a signal, the GSignal runs through the following stages of handler calls:

1. **Default handlers** installed with the G_SIGNAL_RUN_FIRST option
2. **Emission hooks** (see Section 2.6.7)
3. **User−defined handlers** installed *without* the after option (see Section 2.6.5)
4. **Default handlers** installed with the G_SIGNAL_RUN_LAST option
5. **User−defined handlers** installed *with* the after option
6. **Default handlers** installed with the G_SIGNAL_RUN_CLEANUP option

Here are a few additional functions for emitting signals:

- g_signal_emitv(const GValue *object_and_parms*, guint *signal_id*, GQuark *detail*, GValue **result*)

  To use this function, you must store the target object and signal parameters in a GValue array *object_and_parms* and provide a place for the return value at result.
- g_signal_emit_valist(gpointer *object*, guint *signal_id*, GQuark *detail*, va_list*va_args*)

  This function works just like g_signal_emit, but with a previously prepared variable argument list *va_list* for the handler arguments and return value. With this call, you can create your own signal emission functions that take variable arguments.

2.6.3 Marshallers

When some code emits a signal, GSignal uses a marshaller to transport a list of parameters to the signal handler and to collect and propagate any return values.

Marshallers have names that reflect the parameter types and return values. The format is:

```
prefix_RETURNTYPE__PARM1TYPE[_PARM2TYPE_...]
```

For example, the marshaller for media_unpacked() was g_cclosure_marshal_VOID__VOID because this handler takes no parameters other than the object and returns nothing.

GObject comes with a number of marshallers for one parameter and no return value, as shown in the table on the next page.

Marshaller	**Parameter Type**
g_cclosure_marshal_VOID__BOOLEAN	gboolean
g_cclosure_marshal_VOID__BOXED	GBoxed*
g_cclosure_marshal_VOID__CHAR	gchar
g_cclosure_marshal_VOID__DOUBLE	gdouble
g_cclosure_marshal_VOID__ENUM	gint (enumeration types)
g_cclosure_marshal_VOID__FLAGS	guint (options)
g_cclosure_marshal_VOID__FLOAT	gfloat
g_cclosure_marshal_VOID__INT	gint
g_cclosure_marshal_VOID__LONG	glong
g_cclosure_marshal_VOID__OBJECT	GObject*
g_cclosure_marshal_VOID__PARAM	GParamSpec* or derived
g_cclosure_marshal_VOID__POINTER	gpointer
g_cclosure_marshal_VOID__STRING	gchar* or gchararray
g_cclosure_marshal_VOID__UCHAR	guchar
g_cclosure_marshal_VOID__ULONG	gulong
g_cclosure_marshal_VOID__UINT	guint
g_cclosure_marshal_VOID__VOID	void (no parameters)

Warning Using the wrong marshaller will probably cause your program to crash.
If you don't see the marshaller you need in the preceding list (that is, your signal handler returns a value and/or takes more than a single parameter), you have to provide your own. Your marshaller names should resemble the following:

```
_my_marshal_INT__VOID
_my_marshal_VOID__OBJECT_INT
_my_marshal_UINT__BOOLEAN_BOOLEAN
```

where _my_marshal is your prefix.

Thankfully, you don't have to actually write the marshaller code; there's a utility called glib−genmarshal to do the dirty work for you. For example, to create the marshallers above, put the following in a file called my_marshaller.list:

```
INT:VOID
VOID:OBJECT,INT
UINT:BOOLEAN
```

The file format is fairly obvious; each line is a ***signature*** defining a new marshaller, starting with the return type. After a colon, you list the parameter types. You should be able to determine the valid types from the table earlier in this section.

To create the actual code, run these two commands:

```
glib-genmarshal --prefix _my_marshal --header my_marshaller.list > my_marshaller.h
glib-genmarshal --prefix _my_marshal --body my_marshaller.list > my_marshaller.c
```

You now have a new source file, *my_marshaller.c*, and a *my_marshaller.h* header file.

Warning You don't have to supply a prefix. The default is *g_cclosure_user_marshal*, but if you choose to accept this, be aware that you risk duplicate symbols at link time, especially if you are combining several different pieces of code.

You must include *my_marshaller.h* in the source file that includes your class initialization function (or any other place where you install signal handlers). The *my_marshaller.h* file should look something like this:

```
#ifndef ___my_marshal_MARSHAL_H__
#define ___my_marshal_MARSHAL_H__

#include <glib-object.h>

G_BEGIN_DECLS

/* INT:VOID (my_marshaller.list:1) */
extern void _my_marshal_INT__VOID
    (GClosure     *closure,
     GValue       *return_value,
     guint         n_param_values,
     const GValue *param_values,
     gpointer      invocation_hint,
     gpointer      marshal_data);

/* VOID:OBJECT,INT (my_marshaller.list:2) */
extern void _my_marshal_VOID__OBJECT_INT
    (GClosure     *closure,
     GValue       *return_value,
     guint         n_param_values,
     const GValue *param_values,
     gpointer      invocation_hint,
     gpointer      marshal_data);

/* UINT:BOOLEAN (my_marshaller.list:3) */
extern void _my_marshal_UINT__BOOLEAN
    (GClosure     *closure,
     GValue       *return_value,
     guint         n_param_values,
     const GValue *param_values,
     gpointer      invocation_hint,
     gpointer      marshal_data);

G_END_DECLS

#endif /* ___my_marshal_MARSHAL_H__ */
```

If you're building a Makefile, rules for creating the marshallers would look something like this:

```
my_marshaller.h: my_marshaller.list
        glib-genmarshal --prefix _my_marshal --header \
        my_marshaller.list > my_marshaller.h

my_marshaller.c: my_marshaller.list
```

```
        glib−genmarshal −−prefix _my_marshal −−body \
        my_marshaller.list > my_marshaller.c
```

Note Remember that the whitespace in the preceding listing actually consists of tabs.
You may also want to add a dependency for glib−genmarshal, but this is probably best done with the help of GNU autoconf (see Chapter 6).

2.6.4 Signal Accumulators

If GSignal runs several signal handlers for one signal, the handler calls run in succession, and the marshaller propagates the return value from the last handler back to the code that emitted the signal.

In rare cases, though, you may want to know what all of the handlers returned. You can define an accumulator to collect and process all of the return values.

To install an accumulator along with a signal, supply a GSignalAccumulator callback function as the fifth argument to g_signal_new(). Here is the callback type definition:

```
typedef struct _GSignalInvocationHint GSignalInvocationHint;

  << ... >>

typedef gboolean (*GSignalAccumulator)
     (GSignalInvocationHint *ihint,
      GValue                *return_accu,
      const GValue          *handler_return,
      gpointer               data);

  << ... >>

struct _GSignalInvocationHint
{
  guint         signal_id;
  GQuark        detail;
  GSignalFlags  run_type;
};
```

GSignal calls your accumulator right after it runs each signal handler. As you can see from the preceding code, accumulator functions have four arguments:

- ihint (GSignalInvocationHint *): A structure containing the signal identifier signal_id, a detail (see Section 2.6.6) and the signal options from g_signal_new().
- return_accu (GValue *): The accumulator that GSignal eventually returns to the code that emitted the signal. You can do anything you like with this container.

- handler_return (const GValue *): Contains the return value from the last signal handler.
- data (gpointer): Any accumulator data that you set up with g_signal_new().

Your accumulator should return TRUE if you want GSignal to continue calling signal handlers for this particular signal emission, or FALSE if it should stop.

An accumulator typically may be used to look over Boolean values that signal handlers return. As soon as one of the handlers returns TRUE, the accumulator propagates TRUE as a return value and stops the emission process.

This book doesn't have an example of an accumulator (there's only so much space), but it's easy enough to find one: Unpack the GTK+ source code, change to the distribution's top−level directory, and run this command:

```
grep GSignalInvocationHint */*
```

This command prints lines from the source files that have accumulators.

2.6.5 Attaching Handlers to Signals

As you will see with widget objects in Chapter 3, you want to be able attach different signal handlers to the same kind of object (for example, if you have two button objects, you don't want the buttons to do the exact same thing).

This code attaches a new handler called meep_meep() to the **unpacked** signal on media, using g_signal_connect():

```
static void meep_meep(Media *media)
{
  guint nr;
  g_object_get(media, "inventory−id", &nr, NULL);

  g_print("Meep−meep! (Inventory number: \%d)\n", nr);
}
  << ... >>

gulong handler_id;

/* connect new handler */
handler_id = g_signal_connect(media,
                              "unpacked",
                              (GCallback) meep_meep,
                              NULL);

/* test the new handler */
g_signal_emit_by_name(media, "unpacked");
/* expect "meep−meep" message, plus output of other handler(s) */
```

In this example, GSignal calls meep_meep() before the default signal handler, because the default was not installed with G_SIGNAL_RUN_FIRST.

The most common way to attach a handler is to use

```
gulong handler_id;

handler_id = g_signal_connect(object, name, function, data);
```

- *object* (gpointer): The target object.
- *name* (const gchar *): The signal name.
- *function* (GCallback *): The new signal handler. This callback function must have the same prototype as in the instance structure, but you may need to cast to get a fit as an argument.
- *data* (gpointer): Optional data for the signal handler.

You'll see plenty of uses for the optional data pointer later in this book, such as the one described in Section 3.3. Normally, GSignal attempts to pass the data to the signal handler as the last argument. However, if you want to use a handler that takes a data pointer as its *first* parameter, use

```
g_signal_connect_swapped(object, name, function, data)
```

You might want to do this if your handler is a library function that takes only one argument. An example is in Section 3.6.10.

```
g_signal_connect_after(object, name, function, data)
```

is nearly identical to g_signal_connect(), except that function runs in stage 5 listed in Section 2.6.2 rather than in stage 3. The idea is that you can make the handler run after the default handler, but as you can see from that section, you can also override that behavior when you install the default handler.

All g_signal_connect*() calls return an identifier for the handler binding. If you store the identifier as *handler_id*, you can check the status of a binding with

```
g_signal_handler_is_connected(object, handler_id)
```

To remove a binding, invoke

```
g_signal_handler_disconnect(object, handler_id)
```

Here are some examples:

```
/* test and disconnect handlers */
if (g_signal_handler_is_connected(media, handler_id))
{
   g_print("meepmeep is connected to media. Detaching...\n");
}

g_signal_handler_disconnect(media, handler_id);

if (!g_signal_handler_is_connected(media, handler_id))
{
  g_print("meepmeep no longer connected:\n");
  g_signal_emit_by_name(media, "unpacked");
}
```

Note Remember that any handlers that you connect to an object are on a per−object basis and do not apply for the rest of the class. You can also connect, disconnect, and block signal handlers during emission.

2.6.6 Details

Signal details are further subdivisions of signals. To specify a detail in a signal name, append two colons and the detail name (for example, **unpacked::ding**).

You can add detail information when you connect a handler or emit a signal. When you emit a signal with a detail, GSignal calls the handlers with that detail and those completely without details. GSignal does not call a handler with a detail that does not match the given emission. In addition, if you emit a signal *without* a detail, GSignal will not call any handler connected *with* a detail. You should get the idea from the following example:

```
static void ding(Media *media)
{
  g_print("Ding.\n");
}

static void dong(Media *media)
{
  g_print("Dong.\n");
}

  << ... >>

/* connect handlers with ding and dong details */
g_signal_connect(media, "unpacked::ding", (GCallback)ding, NULL);

g_signal_connect(media, "unpacked::dong", (GCallback)dong, NULL);

g_signal_emit_by_name(media, "unpacked::ding");
/* expect "Ding," then "Media ... unpacked" */

g_signal_emit_by_name(media, "unpacked::dong");
/* expect "Dong," then "Media ... unpacked" */

g_signal_emit_by_name(media, "unpacked");
/* expect only "Media ... unpacked" */
```

Note Signal details work only when you install a signal with the *G_SIGNAL_DETAILED* option (see Section 2.6.1).

2.6.7 Emission Hooks

User−defined signal handlers connect only to single objects. However, you can also define ***emission hooks*** that apply to a GSignal identifier instead of an object. When you emit a signal that has a hook, GSignal calls the hook regardless of the target object. Therefore, you can make user−defined hooks at run time that apply to all objects in a class rather than just one object. You can attach details to hooks, just as you did with regular signals and their handlers.

Hook functions have the GSignalEmissionHook type and look a bit different than normal signal handlers. Here is the function type definition:

```
typedef gboolean (*GSignalEmissionHook) (GSignalInvocationHint *ihint,
                                         guint n_param_values,
                                         const GValue *param_values,
                                         gpointer data);
```

As with the accumulators in Section 2.6.4, each hook receives a GSignalInvocationHint structure and a user−defined untyped data pointer. The other parameters are as follows:

- n_param_values is the number of signal emission parameters.
- param_values is an array of GValues, each containing the parameters from the signal emission. The first parameter is the target object. To get to the others, you need to do some pointer arithmetic (as the example in this section will imply).
- data is a pointer to any user−defined data.

Hooks return gboolean. If a hook returns FALSE, GSignal removes the hook from the signal emission sequence. Make sure to return TRUE if you want your hook to run more than once.

The following example is a little difficult to read at first, but it does very little. Essentially, the hook verifies that its parameter is a Media object and prints the inventory number. After GSignal calls the hook for the third time, the hook returns FALSE and therefore requests removal from the signal.

```
static gboolean my_hook(GSignalInvocationHint *ihint,
                        guint n_param_values,
                        const GValue *param_values,
                        gpointer *data)
{
  static gint n = 0;
  guint inv_nr;
  Media *m;
  GObject *obj;

  g_print("my_hook(): ");

  /* check for a valid Media object */
  if (n_param_values > 0)
  {
     obj = g_value_get_object(param_values + 0);
     if (IS_MEDIA(obj))
     {
        m = MEDIA(obj);
        g_object_get(m, "inventory-id", &inv_nr, NULL);
        g_print("inventory number = %d.\n", inv_nr);
     } else {
        g_print("called with invalid object\n");
     }
  } else {
      g_print("called with invalid parameters\n");
  }

  n++;
  g_print("my_hook(): invocation #%d", n);

  if (n == 3)
  {
     g_print(" (last time)\n");
     return(FALSE);
  } else {
     g_print("\n");
     return(TRUE);
  }
}

  << ... >>

gulong hook_id;
Media *m2, *m3;

  << create one more media object, m2 >>

/* add an emission hook */
  hook_id = g_signal_add_emission_hook(media_signal[UNPACKED],
                                       0,
                                       (GSignalEmissionHook)my_hook,
                                       NULL, NULL);

/* test the hook on three different objects */
g_signal_emit_by_name(media, "unpacked");
g_signal_emit_by_name(m2, "unpacked");
g_signal_emit_by_name(media, "unpacked");

/* this time, the hook should no longer be active */
g_signal_emit_by_name(media, "unpacked");
```

Notice that g_signal_add_emission_hook() uses the signal identifier map from Section 2.6.1.

To remove a hook, run g_signal_hook_remove() on the hook identifier that g_signal_add_emission_hook() returns.

Note You won't find too much use for hooks in common practice. Before you install a hook into a class, you might ask yourself if it's really necessary.

2.6.8 More Signal Utilities

A number of tools are available to monitor and control signals and their emissions. For example, you can get the options specified at installation, the after flag, and handler bindings. In addition, you can block signal handlers and interrupt a signal emission.

Blocking Signal Handlers

- g_signal_handler_block(gpointer *object*, gulong *handler_id*) Disables a signal handler temporarily. GSignal will not call the handler for object until further notice.
- g_signal_handler_unblock(gpointer *object*, gulong *handler_id*) Enables a signal handler.

Note You can disable a signal handler as many times as you like; the effect is like putting an extra latch on a door. Therefore, if you block a handler three times in succession, you must enable it three times to get it working again.

Aborting Signal Emissions

- g_signal_stop_emission_by_name(gpointer *object*, const gchar **signame*) Ends the current signal emission. Note that signame is the signal's name, including any detail. If there is no such signal emission, this function prints a warning message.
- g_signal_stop_emission(gpointer *object*, guint *signal_id*, GQuark *detail*) Same as the preceding function, but uses a signal identifier and a separate detail name.

Identifier Functions

To help you manage signals, names, and identifiers, GSignal provides the following:

- guint g_signal_lookup(gchar **name*, GType *class*) returns the signal identifier corresponding to *name* for the *class* class type identifier.
- gchar *g_signal_name(guint *signal_id*) returns the signal name for *signal_id*.
- guint *g_signal_list_ids(GType *class*, guint **num_sigs*) returns an array of signal IDs for *class*, writing the number of signals in *num_sigs*. You must deallocate this memory by yourself.

Here is a small demonstration:

```
guint i, nr, *sigs;

sigs = g_signal_list_ids(TYPE_MEDIA, &nr);
g_print("ID    Name\n");
g_print("----  -------------\n");
i = 0;
while (i < nr)
{
  g_print("%-4d %s\n", *sigs, g_signal_name(*sigs));
  i++;
  sigs++;
}
g_print("\nTotal signals: %d\n", nr);
g_free(sigs);
```

Miscellaneous Functions

Here are several more functions that work with signals that you might find useful. Refer to the online API documentation for a detailed list of the parameters.

- g_signal_newv() is like g_signal_new(), except that here you supply the handler parameter types in an array.
- g_signal_valist() wants a va_list of the handler parameter types. This function is suitable for building your own signal installers.
- g_signal_connect_data() is the full−blown function for installing signal handlers. The rest of the g_signal_connect functions in this chapter are macros based on this function.
- g_signal_query() asks for detailed information about a signal and fills a GSignalQuery structure with the information. If the utilities in the previous subsection weren't enough for you, check out this one.
- g_signal_handlers_block_matched() blocks all signal handlers that match criteria in a given GSignalMatchType structure.
- g_signal_handlers_unblock_matched() is like the preceding function, but it enables the signal handlers.
- g_signal_handlers_disconnect_matched() is like the preceding function, but it removes the signal handlers from their objects.
- g_signal_handler_find() looks for a signal that matches the criteria in a GSignalMatchType structure.
- g_signal_handlers_block_by_func() disables signal handlers based on a pointer to the handler function.
- g_signal_handlers_unblock_by_func() is the opposite of the preceding function.
- g_signal_handlers_disconnect_by_func() is like the preceding function, but removes the handler.

[3] There are two spellings of this word: *marshaling* and *marshalling*. The text in this book uses the double−l variants, but you may see files and API elements with a single l.

2.7 Inheritance

After digesting the material in the previous sections, you should be quite familiar with the creation and use of classes and signals. This chapter's final topic is inheritance.

In principle, you already saw inheritance when you created the **Media** class, because it is a subclass of **GObject**. Here are the details of how to build a subclass:

1. Define the **instance structure** with a pointer for the parent instance structure at the beginning.
2. Define the **class structure** with a pointer to the parent class structure at the beginning.

3. In the function that returns the new subclass, use the **parent class type identifier** as the first argument to g_type_register_static().
4. In the **class initializer**, install any new properties and signals. You may also install new default handlers for inherited signals.

The greater part of this section illustrates these steps, creating a **CD** class from **Media**. There is only one additional property, **writable**. A small demonstration of how to work with the new subclass follows the definitions.

To start, you need the usual instance and class structure definitions introduced in Section 2.2.1, as well as the macros from Section 2.2.2. Notice that writable is in the instance structure, but you need nothing else. Items such as set_property and get_property come from the parent structures.

```
/******** CD (class derived from Media) **********/

typedef struct _CD {
  Media media_instance;
  gboolean writable;
} CD;

typedef struct _CDClass {
  MediaClass media_class;
} CDClass;

#define TYPE_CD (cd_get_type())
#define CD(object) \
   (G_TYPE_CHECK_INSTANCE_CAST((object), TYPE_CD, CD))

#define CD_CLASS(klass) \
   (G_TYPE_CHECK_CLASS_CAST((klass), TYPE_CD, CDClass))

#define IS_CD(object) \
   (G_TYPE_CHECK_INSTANCE_TYPE((object), TYPE_CD))

#define IS_CD_CLASS(klass) \
   (G_TYPE_CHECK_CLASS_TYPE((klass), TYPE_CD))

#define CD_GET_CLASS(object) \
   (G_TYPE_INSTANCE_GET_CLASS((object), TYPE_CD, CDClass))

static void cd_class_init(CDClass *class);
```

Now you must provide a function to return the new class type identifier (TYPE_CD), as in Section 2.2.3. Note the parent class type identifier from the **Media** class (shown in boldface):

```
GType cd_get_type(void)
{
  static GType cd_type = 0;
  const GInterfaceInfo cleanable_info;

  if (!cd_type)
  {
    static const GTypeInfo cd_info = {
       sizeof(CDClass),
       NULL,
       NULL,
       (GClassInitFunc)cd_class_init,
       NULL,
       NULL,
       sizeof(CD),
       16,
       NULL
    };

    const GInterfaceInfo cleanable_info = {
       cd_cleanable_init, NULL, NULL
    };

    /* Register type, use ID of parent class TYPE_MEDIA */
    /* "CD" is too short, use "CompactDisc" instead */
    cd_type = g_type_register_static(TYPE_MEDIA, "CompactDisc", &cd_info, 0);

    /* add interface */
    g_type_add_interface_static(cd_type, TYPE_CLEANABLE, &cleanable_info);
  }
  return cd_type;
}
```

Now you are almost ready to write the cd_class_init() class initializer, but first, you must provide some dependencies:

```
/* CD constants and prototypes for properties */
enum {
  PROP_0_CD,
  PROP_WRITABLE
};

static void cd_get_property(GObject *object,
                            guint prop_id,
                            GValue *value,
                            GParamSpec *pspec);

static void cd_set_property(GObject *object,
                            guint prop_id,
                            const GValue *value,
                            GParamSpec *pspec);
```

Note By now, the pedantic C programmer may be wondering why you would ever make prototypes for static functions. The prototypes just shown are not necessary if you define the functions before the *cd_class_init()*. In this book, it's primarily a matter of organization   we didn't want to go into detail about properties before explaining the role of the class initializer.

For the sake of demonstration, this subclass replaces **Media**'s default signal handler for **unpacked** with this:

```
/* a new default signal handler for unpacked */
static void unpacked_cd()
{
  g_print("Hi!\n");
}
```

The class initializer is fairly straightforward; notice how replacing the default signal handler is a simple assignment after you get the parent class structure:

```
/* CD class initializer */
static void cd_class_init(CDClass *class)
{
  GObjectClass *g_object_class;
  MediaClass *media_class;

  media_class = MEDIA_CLASS(class);
  media_class->unpacked = unpacked_cd;

  g_object_class = G_OBJECT_CLASS(class);
  g_object_class->set_property = cd_set_property;
  g_object_class->get_property = cd_get_property;

  g_object_class_install_property(
     g_object_class,
     PROP_WRITABLE,
     g_param_spec_boolean("writable", "Writable?",
                          "Is the CD writable?", FALSE,
                          G_PARAM_READWRITE|G_PARAM_CONSTRUCT_ONLY));
}
```

You set and retrieve **writable** as described in Section 2.5.1, but you may wonder how this works. After all, the preceding code overrides the set_property() and get_property() methods for the base class, and there is no mention of the parent's properties in the functions the follow.

The key to understanding this is that GObject initializes parent classes first. When a class installs its properties, GObject associates those properties with the class, and therefore, it can also look up the appropriate *property() functions based on that class.

```
static void cd_set_property(GObject *object,
                            guint prop_id,
                            const GValue *value,
                            GParamSpec *pspec)
{
  gboolean writable;
  CD *cd = CD(object);

  switch(prop_id)
  {
    case PROP_WRITABLE:
       writable = g_value_get_boolean(value);
       if (cd->writable != writable)
       {
          cd->writable = writable;
       }
       break;

    default:
       G_OBJECT_WARN_INVALID_PROPERTY_ID(object, prop_id, pspec);
       break;
  }
}

static void cd_get_property(GObject *object,
                            guint prop_id,
                            GValue *value,
                            GParamSpec *pspec)
{
  CD *cd = CD(object);

  switch(prop_id)
  {
    case PROP_WRITABLE:
       g_value_set_boolean(value, cd->writable);
       break;

    default:
       G_OBJECT_WARN_INVALID_PROPERTY_ID(object, prop_id, pspec);
       break;
  }
}
```

Now we're ready to use the new subclass. For the purposes of demonstration, assume that you also created another new subclass of **Media** called **EightTrack** for 8−track tapes. It adds a **minutes** property to **Media**, representing the total playing time of a tape.

You create objects and access properties as you would expect:

```
Media *media;
CD *cd;
EightTrack *eighttrack;
guint nr;
gboolean is_unpacked;

  << create a new media object >>

/* create a new CD object */
cd = g_object_new(TYPE_CD,
                  "inventory−id", 423,
                  "writable", FALSE,
                  NULL);

/* verify data in the object */
g_object_get(cd,
             "inventory−id", &nr,
             "writable", &is_unpacked,
             NULL);

g_print("cd: writable = %s, inventory−id = %d\n",
        is_unpacked? "true":"false", nr);

/* create an EightTrack object */
eighttrack = g_object_new(TYPE_EIGHTTRACK, "minutes", 47, NULL);
```

The following tests the signal handlers. Remember that the **unpacked** handler for **CD** is different now.

```
/* EightTrack's unpacked handler; same as Media's */
g_signal_emit_by_name(eighttrack, "unpacked", NULL);

/* CD's unpacked handler; expect "Hi!" instead */
g_signal_emit_by_name(cd, "unpacked", NULL);
```

Finally, you can test various objects for membership in classes:

```
/* is cd in Media? (subclass in parent; expect true) */
g_print("cd is %sMedia object\n", IS_MEDIA(cd)? "a " : "not a ");

/* is eighttrack in Media? (subclass in parent; expect true) */
g_print("eighttrack is %sMedia object \n", IS_MEDIA(eighttrack)? "a " : "not a ");

/* is media in CD? (parent in subclass; expect false) */
g_print("media is %sCD object\n", IS_CD(media)? "a " : "not a ");
```

```
/* is cd in EightTrack? (expect false) */
g_print("cd is %sEightTrack object\n", IS_EIGHTTRACK(cd)? "an " : "not an ");
```

Note You sometimes need access to the internals of the parent object in your object initialization and manipulation functions. If you want the **Media** parent object of *cd*, use the *MEDIA()* casting macro to get a *Media* object. Likewise, you can get at the parent class with *MEDIA_CLASS()*. The class initializer function *cd_class_init()* used this to override the ***unpacked*** signal handler.

2.7.1 Interfaces

In principle, an interface is nothing but a class with no objects and no regard for class hierarchy. Interfaces consist only of methods, and an object implements the interface when its class has all of these methods.

An interface's infrastructure includes an abstract interface type with a class structure, but no instance structure. Interfaces inherit all of their base characteristics from a base interface **GTypeInterface** (type identifier G_TYPE_INTERFACE), much like a regular class inherits from **GObject**.

Defining an Interface

This section illustrates an interface called **Cleanable** that **CD** and **EightTrack** implement. **Cleanable** will include only one method: void clean(Cleanable *object).

The structures are fairly trivial   the instance structure is empty, and the class structure contains the parent interface and a pointer to the clean method:

```
/* empty declaration for instance structure */
typedef struct _Cleanable Cleanable;

/* Cleanable class structure */
typedef struct _CleanableClass {
  GTypeInterface base_interface;
  void (*clean) (Cleanable *object);
} CleanableClass;
```

Next, you must define a type identifier, casting, membership, and interface macros for **Cleanable**. Following the naming conventions in Section 2.2.2, TYPE_CLEANABLE() returns the result of cleanable_get_type(), IS_CLEANABLE() verifies that an object implements the interface, and CLEANABLE() casts an object to a cleanable type. CLEANABLE_GET_CLASS() returns the interface class, *not* the class of the object.

Another deviation from normal classes and objects is that you don't need CLEANABLE_CLASS or IS_CLEANABLE_CLASS, again, because there are no objects that belong strictly to the **Cleanable** interface.

```
GType cleanable_get_type() G_GNUC_CONST;

#define TYPE_CLEANABLE (cleanable_get_type())

#define CLEANABLE(object) \
  (G_TYPE_CHECK_INSTANCE_CAST((object), TYPE_CLEANABLE, Cleanable))

#define IS_CLEANABLE(object) \
  (G_TYPE_CHECK_INSTANCE_TYPE((object), TYPE_CLEANABLE))

#define CLEANABLE_GET_CLASS(object) \
  (G_TYPE_INSTANCE_GET_INTERFACE((object), TYPE_CLEANABLE, CleanableClass))
```

The type initializer (cleanable_get_type()) is very similar to that of a class (see Section 2.4.1). However, there are a few differences:

- You need only three fields in the GTypeInfo structure.
- Base initializer and base finalizer functions are not NULL.

The code you're about to see also includes the base initializer and finalizer for the **Cleanable** interface. These do nothing more than manipulate a global variable that serves as a reference counter. When the counter goes from 0 to 1, or from 1 to 0, you may want to do something special with the interface. In the following examples, empty code blocks indicate where to place this code.

This process is somewhat clumsy, but it's necessary because interfaces are not derived from **GObject** and thus have no common class initializer.

```
static guint cleanable_base_init_count = 0;

static void cleanable_base_init(CleanableClass *cleanable)
{
  cleanable_base_init_count++;

  if (cleanable_base_init_count == 1)
  {
     /* "constructor" code, for example, register signals */
  }
}

static void cleanable_base_finalize(CleanableClass *cleanable) {
  cleanable_base_init_count--;

  if (cleanable_base_init_count == 0)
  {
     /* "destructor" code, for example, unregister signals */
  }
}

GType cleanable_get_type(void)
{
  static GType cleanable_type = 0;

  if (!cleanable_type)
  {
     static const GTypeInfo cleanable_info = {
        sizeof(CleanableClass),
        (GBaseInitFunc) cleanable_base_init,
        (GBaseFinalizeFunc) cleanable_base_finalize
     };

     cleanable_type = g_type_register_static(G_TYPE_INTERFACE,
                                             "Cleanable",
                                             &cleanable_info,
                                             0);
  }

  return cleanable_type;
}
```

Implementing and Installing an Interface

Every class that implements an interface must advertise the implementation. The relevant function call here is

```
g_type_add_interface_static(class_type_id, interface_type_id, info)
```

Place this call in your type registration function (for example, cd_get_type()). The arguments here are as follows:

- *class_type_id* (GType): The implementing class type identifier.
- *interface_type_id* (GType): The interface type identifier.
- *info* (const GInterfaceInfo *): Contains these fields:
    - interface_init (GInterfaceInitFunc): GObject calls this function to initialize the interface when you cast an object from the implementing class to the interface.
    - interface_finalize (GInterfaceFinalizeFunc): GObject runs this function when the interface is no longer needed.
    - interface_data (gpointer): Optional data that you may pass to either of the preceding functions.

The code for installing the interface in cd_get_type() follows; eighttrack_get_type() is similar.

```
static void cd_cleanable_init(gpointer interface, gpointer data);

GType cd_get_type(void)
{
  static GType cd_type = 0;

  if (!cd_type)
  {
     const GInterfaceInfo cleanable_info = {
        cd_cleanable_init, NULL, NULL
     };

       << type initializing code >>

     /* add interface */
     g_type_add_interface_static(cd_type, TYPE_CLEANABLE, &cleanable_info);
  }
  return cd_type;
}
```

Here are the type definitions for GInterfaceInitFunc and GInterfaceFinalizeFunc from the GLib header files:

```
typedef void(*GInterfaceInitFunc) (gpointer g_iface, gpointer iface_data);

typedef void(*GInterfaceFinalizeFunc) (gpointer g_iface, gpointer iface_data);
```

Normally, this *class−specific* interface initialization function does nothing other than verify that the interface is ready and then install the actual interface implementation function. For the **CD** class, this function is named cd_clean().

When verifying the interface, recall from the previous section that the **Cleanable** class used a cleanable_base_init_count global variable to keep track of reference counts. If the interface is ready, that count is greater than zero:

```
void cd_clean(Cleanable *cleanable);

static void cd_cleanable_init(gpointer interface, gpointer data)
{
  CleanableClass *cleanable = interface;

  g_assert(G_TYPE_FROM_INTERFACE(cleanable) == TYPE_CLEANABLE);
  /* is the interface ready? */
  g_assert(cleanable_base_init_count > 0);

  cleanable->clean = cd_clean;
}
```

Here is the implementation of cd_clean():

```
void cd_clean(Cleanable *cleanable)
{
  IS_CD(CD(cleanable));

  g_print("Cleaning CD.\n");
}
```

The interface code for **EightTrack** is nearly identical.

One issue remains: how to bring the **Cleanable** implementations for **CD** and **EightTrack** into a single method called clean(). This method takes one Cleanable * object argument (an untyped pointer, of sorts) and runs the implementation that matches the class behind the object. The general procedure is as follows:

1. If the argument doesn't implement the interface, abort.
2. Retrieve the class−specific interface from the object.
3. Add a reference to the object, so that GObject doesn't delete the object in the process of running the interface.
4. Call the class−specific interface function.
5. Remove the extra reference to the object.

Here is the code:

```
void clean(Cleanable *object)
{
  CleanableClass *interface;

  g_return_if_fail(IS_CLEANABLE(object));

  interface = CLEANABLE_GET_CLASS(object);
  g_object_ref(object);
  interface->clean(object);
  g_object_unref(object);
}
```

This short survey of the inner workings of interfaces is probably far more than you will ever need to know. In practice, you will use standard interfaces, not build your own. Therefore, this section concludes with a short demonstration of how to use the new interface on the cd and eighttrack objects created earlier in this chapter:

```
clean(CLEANABLE(cd));
clean(CLEANABLE(eighttrack));
```

2.8 Further Topics

As with GLib, the topic of GObject and its API is rather broad. Here are some more topics that aren't covered in this chapter:

- GBoxed: A data type and API that allows you to wrap C structures into opaque data types. The most prominent boxed typed is GValue.
- GValueArray packs several GValue elements into an array. Its API is similar to that of GList.
- GClosure: GSignal calls unnamed functions with the help of ***closures***; GClosure functions start with g_closure_.
- g_enum_*, g_flags_* are several functions and types that involve finite sets such as enumeration types and bitwise options.