Programming Tools - Opaque Pointers

Authored by: Chris McKillop

One of the most powerful concepts when writing software is abstraction - hiding the details of a system behind a simpler interface. This article shows you how to use a language feature of C (and C++) to provide a very powerful form of abstraction for use in libraries: opaque pointers.

C/C++ has an interesting language feature. If you declare a **typedef** of a structure pointer, you don't need to provide that structure's definition. For example:

```
typedef struct _hidden_struct *handle;
```

This has declared a new type, handle, which is a pointer to a **struct _hidden_struct**. What's interesting about this? Mainly the fact that you can now use this handle type without ever having a definition of the structure it's a pointer to. Why is this powerful? That's what this article will show you (some people might already see why). Please note that the **typedef** above (and those used throughout this article) aren't strictly needed to hide the structure's internal definition - you can just keep using the **struct** keyword as part of the name. However, the **typedefs** are used to make an abstraction away from the fact that there's a structure at all, instead of turning it into a "handle."

Sample problem

To really see the power of opaque pointers, we'll need a problem to solve. Let's suppose we want to make an image library that loads and saves bitmaps. We know that as time goes on this library will need to be able to do more things (new image types, basic transforms) and we want to preserve both compile time and runtime compatibility for this library. This means that if we have an older application, it should work with a new shared library and that if that older application is rebuilt against the new library and headers, it should still build without errors.

What about C++?

Before this goes much further, I need to address the waving hands of all the C++ fans out there, who might be thinking: "C++ already lets me do all of this behind a nice class interface with inheritance and other nice C++ language features". And this is pretty much true in the case of compile-time compatibility. But, because C++ doesn't let you separate the public and private definitions of a class, the way the class is declared changes its runtime behavior (class size, **vtable** offsets, etc.) - you can't provide runtime compatibility without a lot of tender care (and sometimes special compilers). So, this is still useful to C++ people, even if the slant is more towards those of us using C.

Okay, back to the library. When people design libraries like this, they'll often declare a structure that will get filled in by the library and an API for manipulating this structure. For example:

```
typedef struct
{
void *ptr;
int size;
int bytes_per_pixel;
} bitmap_t;
int bitmap_load_file( char *filename, bitmap_t *bitmap );
int bitmap_save_file( char *filename, bitmap_t *bitmap );
```

```
So, to use this library, you would declare a bitmap_t variable and
invoke bitmap_load_file() with a filename and a pointer to the bitmap_t
that was declared. For example:
bitmap_t bitmap;
int ret;
ret = bitmap_load_file( "sample.bmp", &bitmap );
```

Then the user of the library could simply access the pointer inside the structure and, along with the size and **bytes_per_pixel** members, go to work on the image. However, because the contents of **bitmap_t** are known publicly, if the library is updated with new entries in that structure, then the size of the structure will have been changed. And applications that use an updated shared library will probably crash or other bad things when they call into the new library with a structure of a smaller (different) size.

Alternatively, the API could be defined to take a structure size:

```
int bitmap_load_file( char *filename, bitmap_t *bitmap, int size );

int bitmap_save_file( char *filename, bitmap_t *bitmap, int size );
```

and would be used like this:

```
ret = bitmap_load_file( "sample.bmp", &bitmap, sizeof( bitmap_t ) );
```

Which effectively tags the version of the library by using the size of the structure. However, this makes for a terrible support experience inside the library where the structure size has to be checked, and different code paths are taken based on this structure size. This leads to a lot of code bloat.

We could also try to avoid the structure size issue by padding the structure with some "reserved" space:

```
typedef struct
{
void *ptr;
int size;
int bytes_per_pixel;
char reserved[64];
} bitmap_t;
```

But this is only a temporary fix. What if we didn't choose a value that's large enough? Then we're back to the case where a new library causes problems. If we're liberal with our reserved space, then we waste memory.

Another problem common to all of these approaches is what can occur if the layout of the structure changes. Say a new version of the library is built with a structure definition that looks like this:

```
typedef struct
{
int version;
void *ptr;
int size;
int bytes_per_pixel;
} bitmap_t;
```

Then the compiled applications will also get "confused", since what was previously the structure member **ptr** is now version, and so on. The position of a structure member within a structure is important. Seems pretty much impossible to meet our goals? Fear not, loyal readers, the situation isn't that dire!

Hiding the structure's internals

The common thread to all the situations above was that the compiled application was aware of the size of the structure and the location in the structure of the structure members. So we need to hide the internals of the structure and provide access functions to get the important data out of the structure.

Let's try out a new public interface:

```
typedef struct _internal_bitmap * bitmap_t;
int bitmap_alloc( bitmap_t *bitmap );
int bitmap_free( bitmap_t *bitmap );
int bitmap_load_file( bitmap_t bitmap, char *filename );
int bitmap_save_file( bitmap_t bitmap, char *filename );
int bitmap_get_ptr( bitmap_t bitmap, void **ptr );
int bitmap_get_size( bitmap_t bitmap, int *size );
int bitmap_get_bpp( bitmap_t bitmap, int *bpp );
```

And now we can maintain a private interface that applications never get to see or use, only the library:

```
struct _internal_bitmap
{
void *ptr;
int size;
int bytes_per_pixel;
}
```

Did you notice the opaque pointer? Also, notice we've added "access functions" to get the interesting data from the new bitmap "handle"? It's pretty obvious now that we're passing in a **bitmap_t** (a structure pointer) as a handle to the library, but the **alloc** and **free** functions are a little confusing for people.

When we declare a **bitmap_t**, we're really just declaring a pointer to a structure, so we need to provide some memory for that pointer to point at. Here is the main part of the **bitmap_alloc**() function:

```
int bitmap_alloc( bitmap_t *bitmap )
{
struct _internal_bitmap *handle;
handle = ( struct _internal_bitmap * )malloc( sizeof( *handle ) );
if( handle == NULL )
{
return -1;
}
memset( handle, 0, sizeof( *handle ) );
*bitmap = handle;
return 0;
}
```

Since a **bitmap_t** is just a **struct** pointer, we allocate the proper sized **struct** (which we can do - this code is part of the library and it knows how big the structure is). Once we've verified that the **malloc**() didn't fail, we assign the newly allocated structure to the **bitmap_t** pointer. So when the application calls this function, it will get back the proper sized structure to pass into the rest of the library functions.

Here's an example of an "access function" that uses the allocated bitmap handle:

```
int bitmap_get_ptr( bitmap_t bitmap, void **ptr )
{
if( ptr == NULL )
{
return -1;
}
*ptr = bitmap->ptr;
return 0;
}
```

Since the library knows the definition of the **_internal_bitmap** structure, it can directly access its members. If you tried to access the internals of the **bitmap_t** handle in application code, the compiler would return an error, because it has no idea how the structure is organized or what any of its members are named.

For the last bit of code, I'll write a function that loads a bitmap, sets all the pixels to 255, and writes the bitmap back:

```
int turn_bitmap_white( char *filename )
{
int ret, i;
bitmap_t bitmap;
unsigned char *ptr;
int size;
ret = bitmap_alloc( &bitmap );
if( ret )
return ret;
ret = bitmap_load_file( bitmap, filename );
if( ret )
return ret;
ret = bitmap_get_ptr( bitmap, (void **)&ptr );
ret |= bitmap_get_size( bitmap, &size );
if( ret )
return ret;
for( i=0; i<size; i++ )
{
ptr[i] = 255;
}
ret = bitmap_save_file( bitmap, filename );
if( ret )
return ret;
bitmap_free( &bitmap );
return 0;
}
```

Problem solved

So, we've solved our problem. If we change the structure layout, we'll be okay, since the application code can't access the internals of the structure directly and must use "access functions" to get at the internal data.

If we change the size of the structure, we'll be okay, since the library itself allocates the memory for the structure and knows the proper size of the structure to allocate for the given version of the library. You can now replace the library (in shared library form) without having to rebuild any applications. And you can rebuild applications against the library without change. Now, this does assume that you haven't changed the API to your library - opaque pointers are a powerful tool, but they can't perform magic.

Lastly, the use of opaque pointers enforces good programming practice by providing a defined interface (abstraction) between the application and the library. This is usually a good method even when doing your own projects, since it lets you easily separate functionality for future projects!