Books

Advanced C Techniques & Applications The C Programming Language

C/C++

Callbacks Extern Memory Management References and Pointers

Dynamic Libraries

Dynamic library loading is a technique that load classes and modules during runtime rather than having them statically linked during compile time. This allows developers to have more flexibility and extensibility in their design without sacrificing robustness of the design.

In Unix based operating systems, there is already a library that allows one to load shared objects (.so files) into a program. These commands are as follows:

```
void* dlopen(const char);
void* dlsym(void* handle, char* symbol);
const char* dlerror();
int dlclose(void* handle);
```

which give us the ability to dynamically open and load classes.

However, in C there are no classes; Therefore this function is used to load functions rather than classes. This means that the loader is looking for prototypes in order to properly instantiate the function. In C++ this brings a few problems: how do you locate the symbols that are needed in the library, how can you create objects belonging to the class that is loaded, and how can you access them in a useful manner?

The answer is found in polymorphism. Because the main program does not directly create the object one can use a **factory class** or more directly using a single function. For simplicity of the example, a single function will be used in this example.

Using a single function as a simple example. Lets see how you would load a library for creating shapes.

```
shape* maker()
{
  return new square;
}

int main()
{
...
```

```
  void* handle = dlopen("libnewshapes.so", RTLD_NOW);

  if (handle == NULL)
  {
    cerr << dlerror() << endl;
    exit(-1);
  }

  void* mkr = dlsym(handle, "maker");
  shape* myshape = static_cast<shape* ()>(mkr)();
...
}
```

This code snippet opens a shared library called "libnewshapes.so" and make makes a handler. The `dlsym()` command takes the handler to the shared library and passes it to the maker function that instantiates the object and returns the memory location to `mkr`. We then cast `mkr`, a void pointer, into a shape pointer allowing the compiler to properly interpret the data in the given memory location.

However, now we have a new problem. This will most likely fail to compile because `maker()` function cannot be resolved. This problem can be resolved by telling the compiler to use C-style linkage using the extern "C" qualifier.

```
#ifndef __CIRCLE_H
#define __CIRCLE_H
#include "shape.hh"
class circle : public shape
{
public:
   void draw();
};
#endif // __CIRCLE_H
#include <iostream> #include "circle.hh"
void circle::draw()
{
   // simple ascii circle<\n>
   cout << "\n";
   cout << "      ****\n";
   cout << "    *      *\n";
   cout << "   *        *\n";
   cout << "   *        *\n";
   cout << "   *        *\n";
   cout << "    *      *\n";
   cout << "      ****\n";
   cout << "\n";
}
```

```
extern "C"
{
shape *maker()
{
   return new circle;
}
class proxy
{
public:
   proxy()
   {
      // register the maker with the factory
      factory["circle"] = maker;
   }
};
// our one instance of the proxy
proxy p;
}
```

Auto Registration

If we would like to create a list of these maker functions, an easy and efficient way is to use the standard map functionality of C++. We can map a maker function to a string associated with what is being created. We can even make things more automated by using "self-registering objects". This is done by using a "proxy" class. The registration occurs in the constructor class, so we need to create only one instance of the proxy class to register the maker. The prototype looks as follows:

```
class proxy
public:
  proxy()
  {
    factory["shape name"] = maker;
  }
};
```

Where `factory[]` is a **global map exported by the main program**. Now in main if we declare `proxy p;` and then instantiate a shape, when the object is created it will place itself into the proxy map.

More About Extern C qualifier

`extern C` makes a function name in C++ have C linkage, or in other words it does not mangle the name. Names can be mangled in C++ because of object overloading, but C does not understand how to interpret that. That is why `extern C` must be used when loading in objects dynamically to make sure that the names are compatible with the C functions being used to load the libraries (`dlopen()`).

C linkages can be used to specify a single or sequence of functions.

```
extern "C" void foo(int);
extern "C"
{
   void g(char);
   int i;
}
```

References

Article ManPage

Function Pointers

Just like there can be data pointers in C++, there are also pointers to functions. The following example shows a function pointer.

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
    void (*fun_ptr)(int);
    fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

The reason for the extra parenthesis around `fun_ptr` is as follows: if we were to write `void* fun_ptr(int)`, that would declare a function that returns a void pointer. Writing it as `void (*fun_ptr)(int)` changes the precedence of the operation to show that this is a pointer to a function.

Interesting Facts about Function Pointers

1. A function pointer points to code, not to data. Typically function pointers store the start of executable code.
2. Unlike normal pointers, we do not allocate/de-allocate memory using function pointers.
3. A function's name can also be used to gets the functions' address.

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
  printf("Value of a is %d\n", a);
}

int main()
{
  void (*fun_ptr)(int) = fun; // & removed

  fun_ptr(10); // * removed

  return 0;
}
```

4. Like normal pointers, you can make an array of function pointers.
5. Like normal data pointers, a function pointer can be passed as an argument and can be returned from a function. (Callbacks)