CSCI251/CSCI851 Spring-2021 Advanced Programming (**S5b**)

Creating libraries

Outline

- Creating libraries.
  - Static libraries.
  - Dynamic libraries.

Creating libraries

- We have looked at using libraries other people have written.
  - A library is pretty much a concatenation of several object files.
- We can write our libraries for other people to use.
- But we don’t always want them to see the details of our implementation.
  - Not necessarily going to the point of obfuscation, rather using dynamically linked libraries where we provide headers and already compiled code.

- The library may be indexed thus making functions and variables easier to find during the linking phase.
- Any programmer can use the variables or functions defined in a library.
- They are incorporated during the linking stage.

The linker and the library

- When the linker goes to build an executable one of its primary goals is to resolve, unresolved symbols.
- For example if a program calls `cout` …
- … the linker tries to find where `cout` is defined.
- To do this it may ‘link’ in this information, in one of two ways, so when the program is invoked `cout` actually works.

Types of Library

- There are two major types of library, and the type influences linkage:
- Static Libraries:
  - A static library is just a collection of `.o`/`.lib` files concatenated together.
  - When linked with a program the entire contents of the library is brought in thus making the footprint of the final executable larger.

- Shared Libraries:
  - These are a little different.
  - Here when the linker tries to resolve symbols it checks to see if they can either be directly linked in through static libraries or linked at runtime.
  - If the library containing the symbol can be linked dynamically then its contents are not placed into the final executable.
  - At runtime the loader inspects the binary looking at the `.dynamic` section of the executable and loads the appropriate libraries.

- In Unix static libraries are known as `.a` files, where as Shared Libraries are known as `.so` files.
- In Windows and Mac OS X shared libraries are known as `.dll` or `.dynlib`.
- The runtime linker is responsible for linking.

Separation: h vs cpp

- We need to tell “clients” about how to use the functions and classes we write, but they don’t see the details of our implementations.
- So we can put the definition in the header and leave the implementation to the cpp file, and the cpp code can be pre-compiled to save time.

Structure of a library

- A library has two parts:
  - The library itself.
    - Depending on whether it is a static library or shared library it will have a different extension, typically `.a` or `.so`.
  - The header file describing the entry points/ function calls and variables in the library which are publicly accessible.
- A library does not have a main function.

Building a static library …

- Building a static library is straightforward, and requires giving appropriate compiler options.
- To turn `code.cpp` into a library enter:

```
$ CC -c code.cpp
$ CC -xar -o libcode.a code.o
```

- The resultant library is `libcode.a`.
- To use it in other programs you would include the header `code.h` and link against `libcode.a`.
  – Libraries are generally prefixed with `lib`.

- Some people prefer to do it using the `ar` command, the archiver.
- The archiver will combine object files to form a library OR an archive

```
$ CC -c code.cpp
$ ar rc libcode.a code.o
```

- Sometimes people have to run the command `ranlib` to create the index for the library.

```
$ ranlib libcode.a
```

- On some platforms `ranlib` does nothing.
  - It is deprecated on Solaris, `ar` does the functionality.

Using the static library

- If we have generated a static library `libcode.a` we can link it in using …

```
$ CC driver.cpp libcode.a -o program
```

- The .a file doesn't need to be compiled, just linked in after the object files for the cpp files, here just `driver.cpp`, have been generated.

Building a shared library

- Dynamic libraries are built in the same way except we use `-G` (for the GNU compiler use `-shared`) as the link directive - these are passed to `ld`.
  - This tells the compiler we want a shared library.
  - We can replace dynamic content without requiring recompilation of the rest.
- To build the shared library you must use the `-Kpic/-KPIC` (GNU compiler use `-fPIC`) directive which produces position independent output.

- You also need to nominate a name using the –`h` directive.

```
$ CC -Kpic -G -o libcode.so.1.0 -h libcode.so.1 code.cpp
```

- All the previous activity can be done in g++ using the following:

```
$ g++ -fpic -shared -o codecode.so.1.0 -Wl,-soname,libcode.so.1 code.cpp
```

- The rest is the same.
  - `Wl` is an option passed to the linker.

Using a library…

- Let us consider a prebuilt library called `libfcode.so.1.0`.
- The header `code.h` and shared object `code.so.1.0` reside in your current home directory.
- You have a file called `main.cpp` which calls a function in `libcode.so.1.0` called `code`.
- How do you compile it and link it against the library?

- Here is how…

```
$ CC -I. -L. main.cpp -lcode
```

→ The `code` is for the library short name…

- This directive means:

  `-I` headers can be found in the current directory along with the default system directory.

  `-L` libraries can be found in the current directory along with the default system directory.

  `-lcode`, link against `libcode.so.1.0` which is in the current directory.

  Without `-L.` it won't be found.

Using a Shared Library

- Libraries have different names at different stages.
- `libcode.so.1.0` library output from compile stage.
- `libcode.so.1` name of library inside library which is used by the runtime linker.
- `libcode.so` name used by the compiler i.e. `-lcode` directive needs this to work.
- All these MUST be created as symbolic links in order for compilation to work.

- Because this is a shared library, you have to tell the runtime linker where to find `libcode.so` because it is not in the default place of `/usr/lib`.
- To do this we set the environment variable:

```
$ LD_LIBRARY_PATH=.
$ export LD_LIBRARY_PATH
```

- We use "." because the library is in the current directory, normally you would use an absolute path.

- On MAC OS X the environment variable `DYLD_LIBRARY_PATH` can be set.
- You can add libraries to the library cache on a host using `ldconfig` - this will help you with look ups OR you can edit `/etc/ld.so.conf`.
- All these help with runtime finding of libraries.