Course
ELEC-A7151
Course materials
Your points
Code
Code Vault
Course Pages
MyCourses
Teams Channel

« 1 Built automation tools | Course materials | 3 Git »
ELEC-A7151 / Software development tools / 2 Software libraries

2 Software libraries¶

Contents

2.1 Overview¶

Software libraries are usually *already compiled* modules of code that are meant to be used by another program(s). Because libraries are already compiled, a library is usually a set of object files, and they are **only meant to be linked** to the executable the programmer is building. The header file of the library defines the interface of a library, as it has all the functions' declarations that are available for use.

Libraries come in two different types:

2.1.1 Static libraries¶

Static libraries are used during the linking stage of the code compilation (*compile time*). They are linked into the final executable, so the advantage of this type of linking is that the library does not need to be loaded when running the final executable. The downside of static linking is the file size; the final executable will be larger as it contains the linked library.

Static libraries usually have **.a** (unix) or **.lib** (Windows) file extension.

Example set of commands for building a static library using gcc:

```
g++ -c examplelib.cpp
ar rcs libexamplelib.a examplelib.o
```

2.1.2 Dynamic libraries¶

Dynamic libraries (also called shared libraries) are only required to be present for use at *runtime*. This means that the code and the final executable can be compiled without having it present at *compile time*. Advantage of dynamic linking is that many different programs can share the same library file. Another advantage is that the library can be updated without recompiling the whole executable that requires the library.

Dynamic libraries usually have **.so** (unix) or **.dll** (Windows) file extension.

Example set of commands of building a dynamic library using gcc:

```
g++ -fPIC -c examplelib.cpp
g++ -shared -o libexamplelib.so examplelib.o
```

2.2 Linking to a library¶

Let's say we have a source file **source.cpp** that uses a function defined inside the **examplelib** we compiled into library before. If we try to compile it with plain

```
g++ source.cpp
```

The linker would complain about the missing symbols we're trying to refer to. Let's also assume that **ONE** of the previously compiled libraries (either the *libexamplelib.so* OR the *libexamplelib.a*) lies in the same directory where the **source.cpp** is being compiled in. To fix the linker issue, we must tell the linker to look for libraries in the current library. We also need to tell the library's name so it will be linked.

```
g++ source.cpp -L. -lexamplelib
```

Now the program will build without issues. The **-L** option tells linker to search for the libraries from the folder "**.**" which is the current directory. This directory can be any directory on the building machine, and if there were many different library directories, the command could contain many L options, e.g. `-L/usr/library1 -L.` The **-l** option tells linker to link to the library with the name given. Pay attention here; the prefix "lib" of the library name is not used here.

« 1 Built automation tools | Course materials | 3 Git »