Chapter 64

External libraries

If you have a C++ compiler, you can write as much software as your want, by yourself. However, some things that you may need for your work have already been written by someone else. How can you use their software?

64.1 What are software libraries?

In this chapter you will learn about the use of *software libraries*: software that is written not as a standalone package, but in such a way that you can access its functionality in your own program.

Software libraries can be enormous, as is the case for scientific libraries, which are oftet multi-person multi-year projects. On the other hand, many of them are fairly simple utilities written by a single programmer. In the latter case you may have to worry about future support of that software.

64.1.1 Using an external library

Using a software library typically means that

- your program has a line

  ```
  #include "fancylib.h"
  ```

- and you compile and link as:

  ```
  icpc -c yourprogram.cxx -I/usr/include/fancylib
  icpc -o yourprogram yourprogram.o -L/usr/lib/fancylib -lfancy
  ```

You will see specific examples below.

If you are now worried about having to do a lot of typing every time you compile,

- if you use an IDE, you can typically add the library in the options, once and for all; or
- you can use *Make* for building your program. See the tutorial.

64.1.2 Obtaining and installing an external library

Sometimes a software library is available through a *package manager*, but we are going to do it the old-fashioned way: downloading and installing it ourselves.

A popular location for finding downloadable software is `github.com`. You can then choose whether to

- clone the repository, or
- download everything in one file, typically with `.tgz` or `.tar.gz` extension; in that case you need to unpack it

  ```
  tar fxz fancylib.tgz
  ```

  This usually gives you a directory with a name such as

  ```
  fancylib-1.0.0
  ```

  containing the source and documentation of the library, but not any binaries or machine-specific files.

Either way, from here on we assume that you have a directory containing the downloaded package.

There are two main types of installation:

- based on *GNU autotools*, which you recognize by the presence of a *configure* program;

  ```
  cmake ## lots of options
  make
  make install
  ```

  or
- based on *Cmake*, which you recognize by the presence of `CMakeLists.txt` file:

  ```
  configure ## lots of options
  make
  make install
  ```

64.1.2.1 Cmake installation

The easiest way to install a package using `cmake` is to create a build directory, next to the source directory. The `cmake` command is issued from this directory, and it references the source directory:

```
mkdir build
cd build
cmake ../fancylib-1.0.0
make
make install
```

Some people put the build directory inside the source directory, but that is bad practice.

Apart from specifying the source location, you can give more options to `cmake`. The most common are

- specifying an install location, for instance because you don't have *superuser* privileges on that machine; or

- specifying the compiler, because cmake will be default use the *gcc* compilers, but you may want the Intel compiler.

```
CC=icc CXX=icpc \
cmake \
    -D CMAKE_INSTALL_PREFIX:PATH=${HOME}/mylibs/fancy \
    ../fancylib-1.0.0
```

64.2 Options processing: cxxopts

Supppose you have a program that does something with a large array, and you want to be able to change your mind about the array size. You could

- You could recompile your program every time.
- You could let your program parse *argv*, and hope you remember precisely how your commandline options are to be interpreted.
- You could use the *cxxopts* library. This is what we will be exploring now.

The *cxxopts* 'commandline argument parser' can be found at https://github.com/jarro2783/cxxopts. After a *Cmake* installation, it is a 'header-only' library.

- Include the header

  ```
  #include "cxxopts.hpp"
  ```

  which requires a compile option:

  ```
  -I/path/to//cxxopts/installdir/include
  ```

- Declare an options object:

  ```
  cxxopts::Options options("programname", "Program description");
  ```

- Add options:

  ```
  options.add_options()
    ("h,help","usage information")
    ("n,nsize","size of the thing",
     cxxopts::value<int>()->default_value("4096"))
     // et cetera
    ;
  ```

- Parse the options:

  ```
  auto result = options.parse(argc, argv);
  ```

- Get result values:

  ```
  if (result.count("help")>0) {
    std::cout << options.help() << std::endl;
    return 0;
  }
  int array_size = result["nsizes"].as<int>();
  ```

Options can be specified the usual ways:

```
myprogram -n 10
myprogram --nsize 100
myprogram --nsize=1000
```

Exercise 64.1. Incorporate this package into primality testing: exercise 52.20.

64.3 Catch2 unit testing

Test a simple function

```
int five() { return 5; }
```

Sucessful test:

Code:

```
TEST_CASE( "needs to be 5","[1]" ) {
    REQUIRE( five()==5 );
}
```

Output
[catch] require:

```
Filters: [1]
===============================================

All tests passed (1 assertion
     in 1 test case)
```

Unsucessful test:

Code:

```
TEST_CASE( "not six","[2]" ) {
    REQUIRE( five()==6 );
}
```

Output
[catch] requirerr:

```
require.cxx:31: FAILED:
  REQUIRE( five()==6 )
with expansion:
  5 == 6

===============================================

test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Function that throws:

```
void even( int e ) {
    if (e%2==1) throw(1);
    cout << "Even number: "
         << e << "\n";
}
```

Test that it throws or not:

Code:

```
TEST_CASE( "even fun","[3]" ) {
    REQUIRE_NOTHROW( even(2) );
    REQUIRE_THROWS( even(3) );
}
```

Output
[catch] requireven:

```
Filters: [3]
Even number: 2
=============================================

All tests passed (2
    assertions in 1 test case)
```

Run the same test for a set of numbers:

Code:

```
TEST_CASE( "even set","[4]" ) {
    int e = GENERATE( 2,4,6,8 );
    REQUIRE_NOTHROW( even(e) );
}
```

Output
[catch] requirgen:

```
Filters: [4]
Even number: 2
Even number: 4
Even number: 6
Even number: 8
=============================================

All tests passed (4
    assertions in 1 test case)
```

How is this different from using a loop? Using `GENERATE` runs each value as a separate program.

Variants:

```
int i = GENERATE( range(1,100) );
int i = GENERATE_COPY( range(1,n) );
```