COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Libraries, Cmake, C++ classes

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Libraries

Libraries are an indispensable tool in any programming language. They are pre-existing code that is compiled and ready for use. They usually provide generic functionality, like linked lists or binary trees that can hold any data, or specific functionality like an interface to a database server.

Most larger software projects will contain several libraries, some of which you may use later on in some other project, or that you just want to separate out for organizational purposes.

When you have a reusable or logically distinct set of functions, it is helpful to build a library from it so that you do not have to copy the source code into your current project and recompile it all the time. Once it is been written and tested, you can safely reuse it over and over again, saving the time and hassle of building it into your project every time.

Libraries

It might help to get a quick rundown of everything that happens from source code to running program:

1. **C Preprocessor**. This stage processes all the preprocessor directives. Basically, any line that starts with a #, such as #define and #include.

2. **Compilation**: Once the source file has been preprocessed, the result is then compiled. Since many people refer to the entire build process as compilation, this stage is often referred to as compilation proper. This stage turns a .c file into an .o (object) file.

Libraries

3. **Linking**: In this phase all of the object files and any libraries are linked together to make your final program. When using static libraries, the actual library is placed in your final program, while for shared libraries, only a reference to the library is placed inside. Now you have a complete program that is ready to run. You can launch it from the shell, and the program is handed off to the loader.

4. **Loading**: This stage happens when your program starts up. Your program is scanned for references to shared libraries. Any references found are resolved and the libraries are mapped into your program.

Creating a Static library

A static library is a collection of object files linked directly into an application at compile time. The code becomes part of the executable, and no external dependencies are needed during runtime.

Steps:

1. **Create the source files**: write C files containing the functions to be included in the library
2. **Compile the object files**: use gcc to compile the source files into object files (.o)
3. **Create the Static library**: use ar (archiver) to bundle the object files into a static library (.a)
4. **Use the Static library in a program**: create a program that uses the library
5. **Run the program**

Creating a Static library

Let's see an example

Create the source files: write C files containing the functions to be included in the library

```
// lib_utils.c
#include "lib_utils.h"

int addition(int a, int b) {
    return a + b;
}

int subtraction(int a, int b) {
    return a - b;
}
```

```
// lib_utils.h
#ifndef LIB_UTILS_H
#define LIB_UTILS_H

int addition(int a, int b);
int subtraction(int a, int b);

#endif
```

Creating a Static library

Compile the object files: use gcc to compile the source files into object files (.o)

gcc -c lib_utils.c

Create the Static library: use ar (archiver) to bundle the object files into a static library (.a)

ar rcs libutils.a lib_utils.o

Creating a Static library

Use the Static library in a program: Create a program that uses the library

```
// main.c
#include <stdio.h>
#include “lib_utils.h"

int main() {
    int a = 5, b = 3;
    printf("Add: %d\n", addition(a, b));
    printf("Subtract: %d\n", subtraction(a, b));
    return 0;
}
```

Creating a Static library

Compile and link the program with the static library:

gcc main.c -L. -lutils -o program

Run the program:

./program

Creating a Dynamic library

A dynamic library (shared library) is linked at runtime rather than compile time. This results in smaller executables and allows multiple programs to share the same library.

Steps:

1. **Create the source files**: write C files containing the functions to be included in the library (we can use the same source file defined in the previous slides)

2. **Compile the object files**: use gcc to compile the source files as position independent code

3. **Create the Dynamic library**: use gcc to bundle the object files into a shared library (.so)

Creating a Dynamic library

4. **Use the Dynamic library in a program**: create a program that uses the library (as shown in the static library section)

5. **Compile and link the program with the shared library**

6. **Set the library path**: i.e. add the current directory to the LD_LIBRARY_PATH to allow the program to find the shared library

7. **Run the Program**

Creating a Dynamic library

Create the source files: use the same source files as above

Compile the object files: compile the source files as position-independent code (PIC) with the -fPIC flag

gcc -c -fPIC lib_utils.c

Create the Dynamic library: use gcc to bundle the object files into a shared library (.so)

gcc -shared -o libutils.so lib_utils.o

Creating a Dynamic library

Use the Dynamic library in a program: create a program as shown in the static library section.

Compile and link the program with the shared library:

gcc main.c -L. -lutils -o program

Set the library path: add the current directory to the LD_LIBRARY_PATH to allow the program to find the shared library

export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH

Run the program

./program

Static vs. Dynamic libraries

Key Differences Between Static and Dynamic Libraries

Feature	Static library (.a)	Dynamic library (.so)
Linking	At compile time	At runtime
Executable Size	Larger (library code included)	Smaller (library code not included)
Portability	Self-contained executable	Requires library to be available at runtime
Updates	Requires recompilation	Library can be updated independently

Additional Notes

- For static libraries, the lib prefix and .a extension are conventional.
- For dynamic libraries, the lib prefix and .so extension are standard.
- To see the libraries linked to an executable, use:

  ldd ./program

Creating libraries in MS Visual Studio

Creating static or dynamic libraries in C using Microsoft Visual Studio involves several steps. Let's see a step-by-step guide for both types, with some examples.

Creating a Static library in Visual Studio

A static library in Windows has a .lib extension. It is linked at compile time, and its code is included in the final executable.

Steps:

1. **Create a Static library project**
2. **Add source files**
3. **Build the Static library**
4. **Use the Static library in another project**

Creating a Static library in Visual Studio

Create a Static library project

1. Open Microsoft Visual Studio.
2. Go to File > New > Project.
3. Select C++ > Library > Static Library (.lib) from the project templates.
4. Name your project (e.g., StaticMathLibrary) and click Create.

Creating a Static library in Visual Studio

Add source files

1. Add a new header file (e.g., MathUtils.h):

```
#pragma once

#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int addition(int a, int b);
int subtraction(int a, int b);

#endif
```

Creating a Static library in Visual Studio

Add source files

2. Add a new source file (e.g., MathUtils.c):

```
#include "pch.h"
#include "MathUtils.h"

int addition(int a, int b) {
    return a + b;
}

int subtraction(int a, int b) {
    return a - b;
}
```

Creating a Static library in Visual Studio

Build the Static library

1. Build the project (Ctrl + Shift + B or Build > Build Solution).

2. A .lib file (e.g., StaticMathLibrary.lib) will be generated in the Debug or Release folder.

Creating a Static library in Visual Studio

Use the Static library in another project

1. Create a new Console Application project (e.g., StaticLibraryClient).
2. Add the MathUtils.h file to the project.
3. Configure the project to use the .lib file.
   a. Right-click the project, select Properties.
   b. Under Configuration Properties > Linker > Input, add the .lib file name under Additional Dependencies (e.g., StaticMathLibrary.lib).
   c. Under Configuration Properties > VC++ Directories, add the path to the .lib file under Library Directories.
4. Write a program to use the library:

Creating a Static library in Visual Studio

Use the Static library in another project

```
#include <stdio.h>
#include "MathUtils.h"

int main() {
    int a = 5, b = 3;
    printf("Add: %d\n", addition(a, b));
    printf("Subtract: %d\n", subtraction(a, b));
    return 0;
}
```

5. Build and run the program.

Creating a Dynamic library in Visual Studio

Creating a Dynamic Library in Visual Studio

A dynamic library in Windows has a .dll extension. It is linked at runtime.

Steps:

1. **Create a Dynamic library project**
2. **Add source files**
3. **Build the Dynamic library**
4. **Use the Dynamic library in another project**

Creating a Dynamic library in Visual Studio

Create a Dynamic library project

1. Open Microsoft Visual Studio.
2. Go to File > New > Project.
3. Select C++ > Library > Dynamic-link Library (.dll) from the project templates.
4. Name your project (e.g., DynamicMathLibrary) and click Create.

Creating a Dynamic library in Visual Studio

Add source files

1. Add a new header file (e.g., MathUtils.h):

```
#pragma once
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

#ifdef MATHUTILS_EXPORTS
#define MATHUTILS_API __declspec(dllexport)
#else
#define MATHUTILS_API __declspec(dllimport)
#endif

MATHUTILS_API int addition(int a, int b);
MATHUTILS_API int subtraction(int a, int b);
#endif
```

Creating a Dynamic library in Visual Studio

Add source files

2. Add a new source file (e.g., MathUtils.c):

```
#include "pch.h"
#include "MathUtils.h"

int addition(int a, int b) {
    return a + b;
}

int subtraction(int a, int b) {
    return a - b;
}
```

Creating a Dynamic library in Visual Studio

Build the Dynamic library

1. Build the project (Ctrl + Shift + B or Build > Build Solution).

2. A .dll file (e.g., DynamicMathLibrary.dll) and an import library (e.g., DynamicMathLibrary.lib) will be generated in the Debug or Release folder.

Creating a Dynamic library in Visual Studio

Use the Dynamic library in another project

1. Create a new Console Application project (e.g., DynamicLibraryClient).
2. Add the MathUtils.h file to the project.
3. Configure the project to use the .lib file.
   a. Right-click the project, select Properties.
   b. Under Configuration Properties > Linker > Input, add the .lib file name under Additional Dependencies (e.g., DynamicMathLibrary.lib).
   c. Under Configuration Properties > VC++ Directories, add the path to the .lib file under Library Directories.
4. Write a program to use the library.

Creating a Dynamic library in Visual Studio

Use the Dynamic library in another project

```
#include <stdio.h>
#include "MathUtils.h"
int main() {
    int a = 7, b = 2;
    printf("Add: %d\n", addition(a, b));
    printf("Subtract: %d\n", subtraction(a, b));
    return 0;
}
```

5. Ensure the .dll file is in the same directory as the executable or in a system path.
6. Build and run the program.

Creating libraries in Visual Studio

Key Differences in Visual Studio

Feature	Static library (.lib)	Dynamic library (.dll)
Linking	At compile time	At runtime
Library extension	.lib	.dll (plus an import .lib)
Executable Size	Larger (library code included)	Smaller (library code not included)
Usage	Self-contained executable	Requires .dll library to be available at runtime

Using Visual Studio should make the process of creating and using libraries straightforward, thanks to its templates and configuration options.

CMake is a powerful cross-platform build system that simplifies the process of building executables on different operating systems using the same source files.

Basic Workflow of CMake

1. **Create a CMakeLists.txt file**: this file contains instructions for building the project, such as source files, target executables, and dependencies.
2. **Generate build files**: use the cmake command to generate platform-specific build files (e.g., Makefiles on Linux, Visual Studio project files on Windows).
3. **Build the project**: Use the generated build files to compile the source code into an executable.

Example: Building an Executable

Directory structure:

```
project/
├── src/
│   ├── main.c
│   └── helper.c
├── include/
│   └── helper.h
└── CMakeLists.txt
```

Source code:

- main.c

```
#include <stdio.h>
#include "utils.h"

int main() {
    printf("Result: %d\n", addition(5, 3));
    return 0;
}
```

CMake

- utils.c

```
#include "utils.h"
int addition(int a, int b) {
    return(a + b);
}
```

- utils.h

```
#ifndef UTILS_H
#define UTILS _H

int addition(int a, int b);

#endif
```

CMake

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.10)

# Project name and version
project(CrossPlatformExecutable VERSION 1.0)

# Specify the C standard
set(CMAKE_C_STANDARD 99)
set(CMAKE_C_STANDARD_REQUIRED True)

# Include directories
include_directories(include)

# Add the executable
add_executable(my_program src/main.c src/utils.c)
```

CMake

cmake_minimum_required specifies a minimum CMake version. This ensures that the following CMake functions are run with a Cmake compatible version.

The *project()* command sets the project name. This call is required with every project. This command can also be used to specify other project level information such as the language or version number.

The *add_executable()* command tells CMake to create an executable using the specified source code files.

CMake

The *set()* command can set a normal, cache, or environment variable to a given value.
CMAKE_C_STANDARD property specifies the C standard whose features are requested to build this target.
CMAKE_C_STANDARD_REQUIRED property is a boolean describing whether the value of C_STANDARD is a requirement.
The *include_directories()* command adds include directories to the build. The given directories are added to those the compiler uses to search for include files. Relative paths are interpreted as relative to the current source directory.

Latest guide at the link below
https://cmake.org/cmake/help/latest/guide/tutorial/index.html

Generating Build Files on Different Platforms

Linux / macOS

1. Open a terminal in the project directory.

2. Run:
   mkdir build
   cd build
   cmake ..
   make

3. The executable my_program will be generated in the build/ directory.

CMake

If CMake is not available, you have to install it

Download the latest version (or an older one) from https://cmake.org/files/
Extract cmake source code from downloaded file:

tar xzf cmake-3.31.3.tar.gz
cd cmake-3.31.3

If you want to see the available configuration options, run command below.

./configure --help

In order to configure cmake before installation, run command below.

./configure --prefix=/opt/cmake

instructing the install script to install CMake in /opt/cmake.

CMake

Compilation:

make

Installation:

make install

After installation without any errors you can verify the installation by running the command:

/opt/cmake/bin/cmake -version

The output should look something like below (depending upon cmake version you are installing).

cmake version 3.31.3

Windows (using Visual Studio)

1. Open a Command Prompt or Developer Command Prompt.

2. Run:
   mkdir build
   cd build
   cmake .. -G "Visual Studio 17 2022"

3. Open the generated .sln file in Visual Studio.

4. Build the project in Visual Studio to generate the my_program.exe.

Cross-Compiling for Other Operating Systems

Windows from Linux (using MinGW)

1. Install MinGW (*mingw-w64*) on Linux.
2. Run:

   mkdir build
   cd build
   cmake .. -DCMAKE_SYSTEM_NAME=Windows
   -DCMAKE_C_COMPILER=x86_64-w64-mingw32-gcc
   make

3. This will generate a Windows executable (my_program.exe).

macOS from Linux (Using Cross-Compilation)

1. Install a cross-compilation toolchain, such as *osxcross*.

2. Configure CMake to use the macOS SDK:

   mkdir build
   cd build
   cmake .. -DCMAKE_SYSTEM_NAME=Darwin
   -DCMAKE_OSX_SYSROOT=/path/to/macOS/SDK
   make

3. The executable for macOS will be created.

Advanced Cross-Platform Features

Specifying OS-specific configurations

You can use conditional checks in CMakeLists.txt to handle OS-specific settings:

```
if(WIN32)
    message(STATUS "Configuring for Windows")
    add_definitions(-DPLATFORM_WINDOWS)
elseif(APPLE)
    message(STATUS "Configuring for macOS")
    add_definitions(-DPLATFORM_MACOS)
elseif(UNIX)
    message(STATUS "Configuring for Linux/Unix")
    add_definitions(-DPLATFORM_LINUX)
endif()
```

Installing the executable

Add an installation step to make the executable portable:

install(TARGETS my_program DESTINATION bin)

The *install()* command generates installation rules for a project. DESTINATION <dir> specifies the directory on disk to which a file will be installed. <dir> should be a relative path. An absolute path is allowed, but not recommended.

Run the following command to install the program:

cmake --install build --prefix /path/to/install

Testing Cross-Platform Code

You can use tools like Docker or Virtual Machines to test builds for different operating systems. Additionally, CI/CD platforms like GitHub Actions, GitLab CI, or Azure Pipelines support building on multiple platforms automatically.

To sum up we can state that CMake abstracts away the complexities of building for multiple platforms by generating the appropriate build files for each OS. By carefully configuring CMakeLists.txt, you can create executables that work seamlessly across Linux, Windows, and macOS.

Using third party libraries

OpenSSL is a powerful toolkit for implementing cryptographic protocols, such as SSL/TLS, and includes various cryptographic algorithms. To create a C program that uses the OpenSSL library on linux, you can follow the steps illustrated below.

Install OpenSSL development libraries

1. Retrieve source code
2. Configuration
3. Compilation

Using OpenSSL library

The OpenSSL source code can be downloaded from https://openssl-library.org/source/

After having extracted the source files, you can usually run *./config* and it will do the correct thing.

After configuring the library, you should run make.
make

What above should create the libraries and the header files to be used in your programs.

Using OpenSSL library

Write your C program

Here's an example C program that uses OpenSSL to calculate the SHA256 hash of a string.

```
#include <stdio.h>
#include <string.h>
#include <openssl/sha.h>

void print_hash(unsigned char *hash, int length) {
    for (int i = 0; i < length; i++) {
        printf("%02x", hash[i]);
    }
    printf("\n");
}
```

Using OpenSSL library


```
int main() {
    const char *input = "Hello, OpenSSL!";
    unsigned char hash[SHA256_DIGEST_LENGTH];

    // Compute SHA256 hash
    SHA256((unsigned char *)input, strlen(input), hash);

    printf("Input: %s\n", input);
    printf("SHA256 Hash: ");
    print_hash(hash, SHA256_DIGEST_LENGTH);

    return 0;
}
```

Using OpenSSL library

Compile the program

To compile a program using OpenSSL, you must link it with the OpenSSL libraries (libssl and libcrypto). Using the gcc compiler:

*gcc hash_example.c -I**openssl_include_folder** -L**openssl_libraries_folder** -o hash_example -lssl -lcrypto*

In the case of my own installation the compilation command is:

gcc hash_example.c -o hash_example –I/home/luigi/openssl-3.0.14/include -L/home/luigi/openssl-3.0.14/ -lssl –lcrypto

-lssl: Links the SSL library.
-lcrypto: Links the cryptographic library.

Using OpenSSL library

Run the program

Execute the compiled program:

./hash_example

Output:

Input: Hello, OpenSSL!

SHA256 Hash:

2c1f3c14f9ef92c263f8a3cf45327d6e0e4e8b57e9a3dcaf34468b127c18e2ba

Remember that the folder where the OpenSSL libraries are available shall be included in LD_LIBRARY_PATH environment variable

LD_LIBRARY_PATH=/home/luigi/openssl/openssl-3.0.14:$ LD_LIBRARY_PATH

Using OpenSSL library

OpenSSL offers a wide range of features, such as encryption, decryption, key generation, and TLS communication. Below an additional example about **AES Encryption and Decryption**:

```
#include <openssl/aes.h>
#include <stdio.h>
#include <string.h>

void print_data(const char *label, const unsigned char *data, int len) {
    printf("%s: ", label);
    for (int i = 0; i < len; i++) {
        printf("%02x", data[i]);
    }
    printf("\n");
}
```

Using OpenSSL library


```
int main() {
    unsigned char key[AES_BLOCK_SIZE] = "mysecretkey12345"; // 16-byte key
    unsigned char input[AES_BLOCK_SIZE] = "plaintextblock"; // 16-byte plaintext
    unsigned char encrypted[AES_BLOCK_SIZE];
    unsigned char decrypted[AES_BLOCK_SIZE];
    AES_KEY encryptKey, decryptKey;

    // Set encryption and decryption keys
    AES_set_encrypt_key(key, 128, &encryptKey);
    AES_set_decrypt_key(key, 128, &decryptKey);

    // Encrypt
    AES_encrypt(input, encrypted, &encryptKey);
    print_data("Encrypted", encrypted, AES_BLOCK_SIZE);

    // Decrypt
    AES_decrypt(encrypted, decrypted, &decryptKey);
    print_data("Decrypted", decrypted, AES_BLOCK_SIZE);

    return 0;
}
```

Using OpenSSL library

Compile and run the program

*gcc aes_example.c -I**openssl_include_folder** -L**openssl_libraries_folder***
-o aes_example -lssl –lcrypto

./aes_example

In the case of my installation the compilation command is:

gcc aes_example.c -o aes_example –I/home/luigi/openssl-3.0.14/include
-L/home/luigi/openssl-3.0.14/ -lssl –lcrypto

Using OpenSSL library

Output

Plaintext ascii: plaintextblock
Encrypted: 688f323f61bea6031efdd4060154ed69
Decrypted: 706c61696e74657874626c6f636b0000
Decrypted ascii: plaintextblock

Using OpenSSL library

If you want to use OpenSSL for secure network communication, you'd typically use its SSL/TLS APIs, below an overview about **SSL/TLS communication**:

1. Set Up the SSL context:

   *SSL_CTX *ctx = SSL_CTX_new(TLS_client_method());*

2. Create and connect SSL socket:

   *SSL *ssl = SSL_new(ctx);*
   SSL_set_fd(ssl, socket_fd);
   SSL_connect(ssl);

3. Perform read/write:

   SSL_write(ssl, buffer, length);
   SSL_read(ssl, buffer, length);

4. Clean up:

   SSL_free(ssl);
   SSL_CTX_free(ctx);

Classes in C++

In C++, a **class** is a user-defined data type that acts as a blueprint for creating objects. Classes encapsulate data members (variables) and member functions (methods) that operate on the data, enabling the concept of encapsulation and object-oriented programming (OOP).

- **Encapsulation**: bundling data and methods together.
- **Data Members**: variables that hold the data of the class.
- **Member Functions**: functions that operate on the data members.
- **Objects**: instances of a class created using the class blueprint.
- **Access Specifiers**: control the visibility of members.

Classes in C++


```
class ClassName {
public:
    // Public members (accessible from outside the class)
    int publicVar;

    void publicMethod() {
        // Code for the method
    }

private:
    // Private members (accessible only within the class)
    int privateVar;

    void privateMethod() {
        // Code for the method
    }

protected:
    // Protected members (accessible within the class and derived classes)
    int protectedVar;
};
```

Defining a class:

A class in C++ is defined using the ***class*** keyword

Classes in C++


```
#include <iostream>
using namespace std;

class Car {
public:
    // Data members
    string brand;
    int year;

    // Member function
    void displayInfo() {
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};

int main() {
    Car car1; // Create an object of the Car class
    car1.brand = "Toyota";
    car1.year = 2020;
    car1.displayInfo(); // Call the member function

    return 0;
}
```

Creating an object:
An object is an instance of a class

ClassName obj; // Creates an object of type ClassName

You can access the public members of the object using the dot operator (.)

Output:
Brand: Toyota, Year: 2020

Access Specifiers

- **Public**: members are accessible from outside the class.
- **Private**: members are accessible only within the class.
- **Protected**: members are accessible within the class and derived classes.

Constructor and Destructor

- **Constructor**: a special function automatically called when an object is created. Used to initialize objects.
- **Destructor**: A special function automatically called when an object is destroyed. Defined using ~ClassName().

Classes in C++


```
class Car {
public:
    string brand;
    int year;

    // Constructor
    Car(string b, int y) {
        brand = b;
        year = y;
    }

    void displayInfo() {
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};
```

Usage:

```
Car car1("Toyota", 2020);
```

Inheritance

A class can inherit members from another class using the **:** operator.

```
class Vehicle {
public:
    string type;
};

class Car : public Vehicle {
public:
    string brand;
};
```

Classes in C++


```
#include <iostream>
using namespace std;

class BankAccount {
private:
    string accountHolder;
    double balance;

public:
    // Constructor
    BankAccount(string name, double initialBalance) {
        accountHolder = name;
        balance = initialBalance;
    }

    // Member functions
    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: $" << amount << endl;
    }
    void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            cout << "Withdrawn: $" << amount << endl;
        } else {
            cout << "Insufficient funds!" << endl;
        }
    }

    void displayBalance() {
        cout << accountHolder << "'s Balance: $" << balance << endl;
    }
};

int main() {
    BankAccount account("John Doe", 500.0);
    account.displayBalance();
    account.deposit(150.0);
    account.withdraw(100.0);
    account.displayBalance();

    return 0;
}
```

Classes in C++

Output:
John Doe's Balance: $500
Deposited: $150
Withdrawn: $100
John Doe's Balance: $550

Key Features of Classes in C++

- Encapsulation: Data and functions are bundled together.
- Abstraction: Hides implementation details.
- Inheritance: Enables reusability of code.
- Polymorphism: Allows functions to be defined in multiple forms.