Boost C++ Libraries Cheat Sheet

A quick reference for commonly used Boost C++ libraries, providing concise information on their purpose, usage, and key features.

Smart Pointers

Overview

Boost Smart Pointers provide automatic memory management, preventing memory leaks and simplifying resource handling.

They act like regular pointers but automatically deallocate the memory they point to when no longer in use.

Types of Smart Pointers

`scoped_ptr`	Unique ownership. The object is automatically deleted when the `scoped_ptr` goes out of scope. Not copyable.
`shared_ptr`	Shared ownership. The object is deleted when the last `shared_ptr` pointing to it goes out of scope. Thread-safe reference counting.
`weak_ptr`	A non-owning observer of a `shared_ptr`. It can be used to detect if the object managed by the `shared_ptr` is still alive.
`unique_ptr`	C++11 and later. Replaces `scoped_ptr` with more features and move semantics.

Example Usage

```
#include <boost/smart_ptr.hpp>
#include <iostream>

int main() {
  boost::shared_ptr<int> ptr(new int(10));
  std::cout << *ptr << std::endl; // Output: 10
  return 0;
}
```

```
#include <boost/scoped_ptr.hpp>

void foo() {
  boost::scoped_ptr<int> ptr(new int(20));
  // Memory is automatically released when ptr goes out of scope.
}
```

Boost.Asio

Overview

Boost.Asio is a cross-platform C++ library for network and low-level I/O programming.

It provides an asynchronous model, allowing for efficient handling of multiple concurrent connections.

Key Components

`io_context`	The core of Asio, providing the event loop for asynchronous operations.
`sockets`	Classes for creating and managing network sockets (e.g., TCP, UDP).
`buffers`	Classes for representing data buffers used in I/O operations.
`timers`	Classes for creating and managing asynchronous timers.

Example: Simple TCP Server

```
#include <boost/asio.hpp>
#include <iostream>

using boost::asio::ip::tcp;

int main() {
  try {
    boost::asio::io_context io_context;
    tcp::acceptor acceptor(io_context, tcp::endpoint(tcp::v4(), 1234));

    tcp::socket socket(io_context);
    acceptor.accept(socket);

    std::cout << "Client connected." << std::endl;
  } catch (std::exception& e) {
    std::cerr << "Exception: " << e.what() << std::endl;
  }
  return 0;
}
```

Boost.Filesystem

Overview

Boost.Filesystem provides portable facilities to manipulate files and directories.

It abstracts away platform-specific details, allowing for consistent file system operations across different operating systems.

Key Classes and Functions

`path`	Represents a file or directory path.
`exists(path)`	Checks if a file or directory exists at the given path.
`create_directory(path)`	Creates a new directory at the given path.
`remove(path)`	Removes a file or directory.

Example: Checking File Existence

```
#include <boost/filesystem.hpp>
#include <iostream>

namespace fs = boost::filesystem;

int main() {
  fs::path p("example.txt");
  if (fs::exists(p)) {
    std::cout << "File exists." << std::endl;
  } else {
    std::cout << "File does not exist." <<
std::endl;
  }
  return 0;
}
```

Boost.Serialization

Overview

Boost.Serialization enables serializing C++ data structures to various formats (e.g., binary, XML) and deserializing them back.

It simplifies the process of saving and loading complex objects.

Key Concepts

`serialize` function	A member function (or a free function) that defines how an object is serialized and deserialized.
Archive	A class that handles the actual serialization/deserialization process (e.g., `binary_oarchive`, `xml_oarchive`).

Example: Serializing a Class

```
#include
<boost/serialization/serialization.hpp>
#include
<boost/serialization/binary_archive.hpp>
#include <fstream>

class MyData {
public:
  int x;
  double y;

  template <class Archive>
  void serialize(Archive & ar, const unsigned
int version)
  {
    ar & x;
    ar & y;
  }
};

int main() {
  MyData data = {5, 3.14};
  std::ofstream ofs("data.bin");
  boost::archive::binary_oarchive ar(ofs);
  ar << data;
  return 0;
}
```