Plugins - Allow others to extend your application

Sławomir Grabowski — 27.11.2024

Wro.cpp × QuickTurn STUDIO

Who I am?

Sławomir Grabowski

10+years of experience mostly in:
— modern C++
— Qt
— 2D &3D graphics engines like Godot, Unity & Unreal Engine
— last 5 years focused on developing graphical frameworks for 2D &3D apps
Other:
— co-founder & developer in Quick Turn Studio
— modern CMake and modern C++ trainer
— currently Game Developer and Producer of "Whispers of Elenrod" - strategy deckbuilder with RPG elements

Presentation scope

- what is a plugin?
- why do we consider plugin as useful tool?
- example implementations in C++
- tips & tricks for plugins for CMake build system
- fill free to ask questions in any moment

Compile artifacts types

Applications

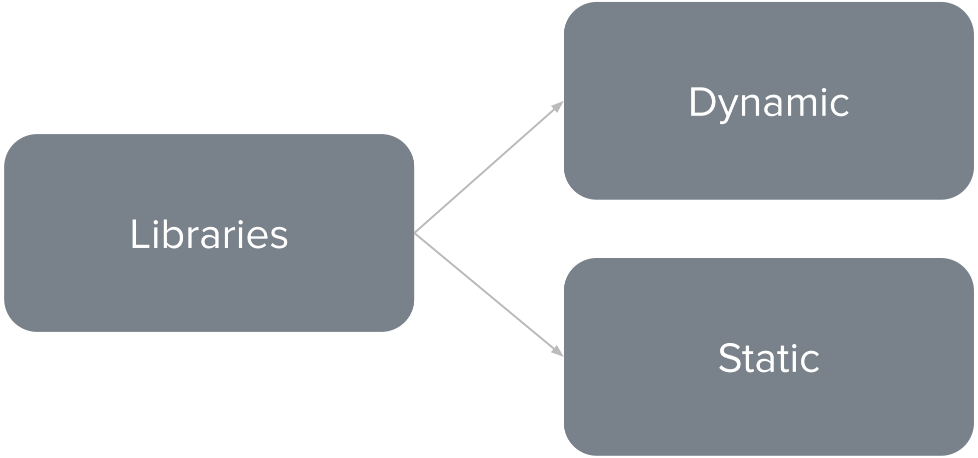
Libraries

Objects

Compile artifacts types

- Applications
- Libraries
- Objects

Compile artifacts types

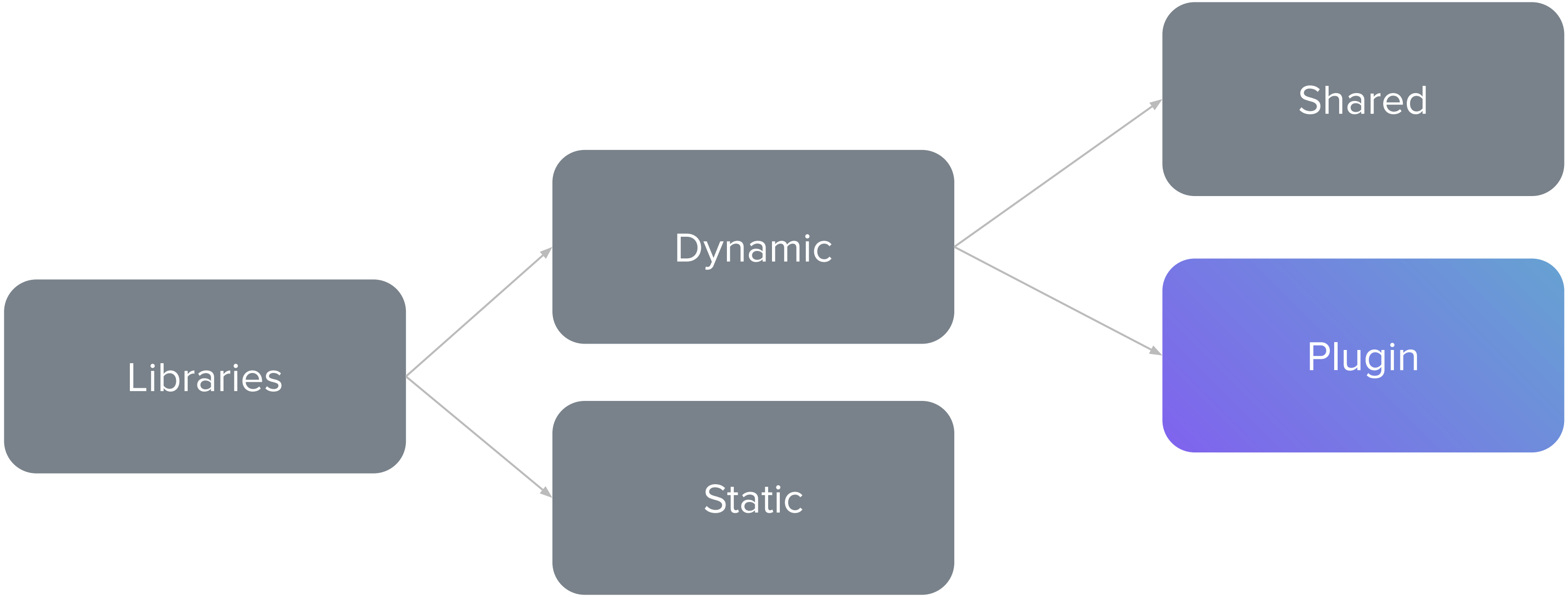


Libraries

Dynamic

Static

Compile artifacts types



Shared vs Plugin

Shared	Plugin
Opened by operating system with application start	Opened by application in runtime
Closing by operating system	Needs to be closed in runtime
Needs to be linked to target (application/ another library) during build	Target potentially uses plugin can be build without building plugin
Needs to be delivered with application	Can be delivered separately from target by another party

Why plugin instead of shared library?

- Adding/extending application functionality without application rebuild
- We can split software functionality on many parts
- Plugin can be delivered separately
- Plugin developer (eg. third party) doesn't need access to application source code (just need types definitions)

Why plugin instead of shared library?

- Plugin can be build with different compiler, different language standard (but keeping compatibility with API)
- Updating functionality provided by plugin usually does not require delivering entire product again

Example scenarios

- supporting custom file formats
- extending functionality
- adding new components
- feature polymorphism

```
// CMakeLists.txt o plugin project

add_library(MyPlugin MODULE meshLoader.cpp)

target_link_libraries(MyPlugin PRIVATE PluginsAPI)

if (MSVC)
    target_compile_definitions(
        MyPlugin PRIVATE "PLUGIN_API=__declspec(dllexport)")
else()
    target_compile_definitions(
        MyPlugin PRIVATE "PLUGIN_API=")
endif()
```

```cpp
// meshLoader.h

#include <Mesh.h>  // struct/class Mesh from Plugin's API

PLUGIN_API Mesh loadMeshFile(const char* filePath);

// meshLoader.cpp

#include "MeshLoader.h"

Mesh loadMeshFile(const char* filePath)
{
    auto mesh = Mesh();

    // loading data from custom format
    // and filling 'mesh' variable with vertices, faces etc.
    // ...

    return mesh;
}


```

```cpp
#include <dlfcn.h> // Linux specific

typedef Mesh (*funcType)(const char*);

int main() {
    const auto pluginPath = "example/plugin/path/MyPlugin.so"
    auto dllHandle = dlopen(pluginPath, RTLD_LAZY);

    funcType functionPtr = dlsym(dllHandle, "loadMeshFile");

    const auto mesh = functionPtr("file.mesh");

    // do something with the mesh...

    dlclose(dllHandle);

    return 0;
}
```

```
int main() {
    const auto pluginPath = "example/plugin/path/MyPlugin.so"
    auto dllHandle = dlopen(pluginPath, RTLD_LAZY);
    if (!dllHandle) {
        std::cerr << "Cannot open plugin file: " << pluginPath << '\n';
        return 1;
    }
    dlerror(); // let's clear previous errors
    funcType functionPtr = dlsym(dllHandle, "loadMeshFile");

    const auto error = dlerror();
    if (error) {
        std::cerr << "Cannot find function loadMeshFile: ";
        std::cerr << error << '\n';
        dlclose(dllHandle);
        return 2;
}
    const auto mesh = functionPtr("file.mesh");

    dlclose(dllHandle);
    return 0;
}
```

```cpp
#include <windows.h> // Windows specific

typedef Mesh (*funcType)(const char*);

int main() {
    const auto dllHandle = LoadLibrary(pluginPath);

    funcType functionPtr = GetProcAddress(dllHandle, "loadMeshFile");

    const auto mesh = functionPtr("file.mesh");

    // do something with the mesh...

    FreeLibrary(dllHandle);

    return 0;
}
```

```cpp
// ============ MeshLoader.h ============

#include <Mesh.h>

PLUGIN_API Mesh loadMeshFile(const char* filePath);

// =========== MeshLoader.cpp ===========

#include "MeshLoader.h"

Mesh loadMeshFile(const char* filePath)
{
    auto mesh = Mesh();

    // ...

    return mesh;
}

// there is small mistake here


// ============ MeshLoader.h ============

#include <Mesh.h>

extern "C"
{
    PLUGIN_API Mesh loadMeshFile(const char* filePath);
}

// =========== MeshLoader.cpp ===========

#include "MeshLoader.h"

Mesh loadMeshFile(const char* filePath)
{
    auto mesh = Mesh();

    // ...

    return mesh;
}

```

```cpp
// ============ MeshLoader.h ============

#include <Mesh.h>

extern "C" PLUGIN_API Mesh loadMeshFile(const char* filePath);

// =========== MeshLoader.cpp ===========

#include "MeshLoader.h"

Mesh loadMeshFile(const char* filePath)
{
    auto mesh = Mesh();

    // ...

    return mesh;
}
```

Returning own type

- Sometimes we would like to add new type in plugin and make application using it, solution?

Returning own type

- Sometimes we would like to add new type in plugin and make application, using it, solution?
- Of course - polymorphism!

Returning own type

- Create abstract type
- Deliver header files with classes declaration to allow inherit from them

```cpp
// defining interface

class IMeshLoader
{
public:
    virtual ~IMeshLoader() = default;

    virtual std::vector<std::string> getSupportedFormats() const = 0;
    virtual Mesh importMesh(const std::string& path) = 0;
    virtual bool exportMesh(const Mesh& mesh, const std::string& path) = 0;
};

class IMeshLoadersModule
{
public:

    virtual ~IMeshLoadersModule() = default;
    virtual IMeshLoader* getMeshLoader() const = 0;
};
```

```
// plugin implementation
extern "C" MeshLoaders_API IMeshLoadersModulePtr getModuleInstance();

class MeshLoadersModule : public IMeshLoadersModule
{
public:
    ~MeshLoadersModule() override = default;
    IMeshLoader* getMeshLoader() const override;
};

// application usage
int main() {
    // ..
    auto pluginModule = pluginHandle->getFunction("getModuleInstance");

    // creating OBJ Parser from plugin
    auto loader = pluginModule->getMeshLoader();
    auto mesh = loader->load("game/models/character.obj");
    // using mesh data...

    delete load;

    return 0;
}
```

```cpp
// plugin implementation
extern "C" MeshLoaders_API IMeshLoadersModulePtr getModuleInstance();

class MeshLoadersModule : public IMeshLoadersModule
{
public:
    ~MeshLoadersModule() override = default;
    std::unique_ptr<IMeshLoader> getMeshLoader() const override;
};

// application usage
int main() {
    // ..
    auto pluginModule = pluginHandle->getFunction("getModuleInstance");

    // creating OBJ Parser from plugin
    auto loader = pluginModule->getMeshLoader();
    auto mesh = loader->load("game/models/character.obj");
    // using mesh data...

    // now memory is released by RAII

    return 0;
}
```

```
// plugin implementation
extern "C" MeshLoaders_API IMeshLoadersModulePtr getModuleInstance();

class MeshLoadersModule : public IMeshLoadersModule
{
public:
    ~MeshLoadersModule() override = default;
    std::unique_ptr<IMeshLoader> getMeshLoader() const override;
};
```

```cpp
// plugin implementation
extern "C" MeshLoaders_API IMeshLoadersModulePtr getModuleInstance();

class MeshLoadersModule : public IMeshLoadersModule
{
public:
    ~MeshLoadersModule() override = default;
    std::unique_ptr<IMeshLoader, std::default_delete>
        getMeshLoader() const override;
};
```

Memory management

- Memory allocated by plugin should be released by plugin
- Using `std::unique_ptr` forces way of releasing memory, because `std::unique_ptr` uses `std::default_delete`

```cpp
template<class T, class Deleter = std::default_delete<T>>
class unique_ptr;


template <typename T>
struct CustomDeleter
{
    void operator ()(T* ptr)
    {
        // call your release, eg.
        delete ptr;
    }
}

// example usage
auto myPointer = std::unique_ptr<int, CustomDeleter<int>>(new int(0));
```

Solution? - `std::shared_ptr`

- `std::shared_ptr` supports Type Erasure, so:
  - deleter object is type is not part of `std::shared_ptr` type
  - object destructor does not need to be **`virtual`**

```cpp
// plugin implementation
extern "C" MeshLoaders_API IMeshLoadersModulePtr getModuleInstance();

class MeshLoadersModule : public IMeshLoadersModule
{
public:
    ~MeshLoadersModule() override = default;
    std::unique_ptr<IMeshLoader> getMeshLoader() const override;
};
```

```cpp
// plugin implementation
extern "C" MeshLoaders_API IMeshLoadersModulePtr getModuleInstance();

class MeshLoadersModule : public IMeshLoadersModule
{
public:
    ~MeshLoadersModule() override = default;
    std::shared_ptr<IMeshLoader> getMeshLoader() const override;
};

// no we do not force allocation and release
std::shared_ptr<IMeshLoader> MeshLoadersModule::getMeshLoader() const
{
    return std::make_shared<MeshLoadersModule>();
}
```

```cpp
// plugin implementation
extern "C" MeshLoaders_API IMeshLoadersModulePtr getModuleInstance();

class MeshLoadersModule : public IMeshLoadersModule
{
public:
    ~MeshLoadersModule() override = default;
    std::shared_ptr<IMeshLoader> getMeshLoader() const override;
};

// no we do not force allocation and release
std::shared_ptr<IMeshLoader> MeshLoadersModule::getMeshLoader() const
{
    auto ptr = allocator->allocate<MeshLoadersModule>();

    return std::shared_ptr(ptr, [ptr, allocator]{
        allocator->deallocate(ptr);
    });
}
```

```cpp
// plugin implementation
extern "C" MeshLoaders_API IMeshLoadersModulePtr getModuleInstance();

class MeshLoadersModule : public IMeshLoadersModule
{
public:
    ~MeshLoadersModule() override = default;
    std::shared_ptr<IMeshLoader> getMeshLoader() const override;
};

// no we do not force allocation and release
std::shared_ptr<IMeshLoader> MeshLoadersModule::getMeshLoader() const
{
    auto ptr = allocator->allocate<MeshLoadersModule>();

    return std::shared_ptr(ptr, [ptr, allocator]{
        allocator->deallocate(ptr);
    });
}

// but there is a still one design mistake!

```

Plugin interface

- Plugin does not check function signatures, types etc.
- Types needs to be binary compatible
- Standard C++ Library specify things like types interface, computational complexity, but…

Plugin interface

- Plugin does not check function signatures, types etc.
- Types needs to be binary compatible
- Standard C++ Library specify things like types interface, computational complexity, but does not specify type declarations!

```
// compiler X implementation
template <typename T>
class vector {
public:
    T& at(size_t index) {
        return data[index];
    }
    size_t size() const {
        return size;
    }

private:
    size_t size;
    size_t capacity;
    T* data;
}
```

```
// compiler Y implementation
template <typename T>
class vector {
public:
    T& at(size_t index) {
        return *(data + index);
    }
    size_t size() const {
        return size;
    }

private:
    T* data;
    size_t capacity;
    size_t size;
}
```

```cpp
// defining interface

class IMeshLoader
{
public:
    virtual ~IMeshLoader() = default;

    virtual std::vector<std::string> getSupportedFormats() const = 0;
    virtual Mesh importMesh(const std::string& path) = 0;
    virtual bool exportMesh(const Mesh& mesh, const std::string& path) = 0;
};

class IMeshLoadersModule
{
public:

    virtual ~IMeshLoadersModule() = default;
    virtual std::shared_ptr<IMeshLoader> getMeshLoader() const = 0;
};
```

Solution?

- Require same compiler (not recommended!)
- Implement your interface classes and structures

```cpp
// defining interface

class IMeshLoader
{
public:
    virtual ~IMeshLoader() = default;

    virtual base::Vector<base:String> getSupportedFormats() const = 0;
    virtual Mesh importMesh(const base:String& path) = 0;
    virtual bool exportMesh(const Mesh& mesh, const base:String& path) = 0;
};

class IMeshLoadersModule
{
public:

    virtual ~IMeshLoadersModule() = default;
    virtual base::SharedPtr<IMeshLoader> getMeshLoader() const = 0;
};
```

Do I need to reimplement entire STL?

- You need to get rid of STL from plugin interface
- You can still use STL in your implementation files (in your core software and in plugins)

Do we have other limitations?

Plugin limitations

- forget about standard library in API
- header only API classes
- we cannot throw exceptions through plugin to application core

Am I safe when I have custom shared pointer?

Memory management

- memory allocated by plugin implementation should be released by plugin implementation
- we need to manually open and close plugin connections
- closing plugin library means releasing memory allocated by plugin to operating system

Dangling memory

- after closing connection to plugin library your SharedPtr objects can still point to plugins memory
- make sure that you do not have dangling pointers when you close connection to plugin

Compatibility management

- Divide your architecture for modules
- Define compatibility management for every module
- Implement versioning checking
- Create factory classes to force implementations of given modules
- Create `extern "C"` function for every separate module

Disadvantages of plugins

- No function signature verification
- Potential security risk
- More complex error handling
- Required designing compatibility policy
- More complex memory management
- Missing come compiler/linker errors

CMake
Tip & Tricks

Thank you!

Sławomir Grabowski

grabowski@quickturngames.com

https://quickturnstudio.com

https://quickturngames.com

https://www.linkedin.com/in/slawomirgrabowski/

https://github.com/Quick-Turn-Studio