Sonova CPP Course Day 3 Homework

Ray Boucher & Nathaniel Tye

Task 1 – Individual Task:

```cpp
#include <iostream>

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

int main()
{
    int stack_var = 42; // allocate variable in .stack memory
    int* heap_pointer_var = new int(); // allocate pointer to .heap memory
    static int data_var = 42; // allocate variable in .data memory
    static int bss_var; // allocate variable in .bss memory
    void* text_pointer = (void*)add; ; // allocate pointer variable
                                       // to .text memory

    return 0;
}
```

Task 2 – Group Task

ButtonEvent header (ButtonEvent.hpp):

```cpp
#pragma once

enum class PressType
{
    cShortPress,
    cLongPress,
    cVeryLongPress
};

enum class ButtonNumber
{
    cButton0,
    cButton1
};

class ButtonEvent
{
```

```
public:
    ButtonEvent(PressType pressType, ButtonNumber buttonNumber) :
        _pressType{pressType},
        _buttonNumber{buttonNumber}
    {

    }

    const ButtonNumber& getButtonNumber() const
    {
        return _buttonNumber;
    }

    const PressType& getPressType() const
    {
        return _pressType;
    }

    const void print() const;

private:
    PressType _pressType;
    ButtonNumber _buttonNumber;
};
```

ButtonEvent source (ButtonEvent.cpp):

Here we define a print function for testing.

```
#include <iostream>

#include "ButtonEvent.hpp"

const void ButtonEvent::print() const
{
    switch (_buttonNumber)
    {
    case ButtonNumber::cButton0:
        std::cout << "Button #0, ";
        break;
    case ButtonNumber::cButton1:
        std::cout << "Button #1, ";
        break;
    default:
        std::cerr << "Invalid buttonNumber, ";
        break;
    }
    switch (_pressType)
    {
    case PressType::cShortPress:
        std::cout << "Short press.\n";
        break;
    case PressType::cLongPress:
        std::cout << "Long press.\n";
        break;
    case PressType::cVeryLongPress:
        std::cout << "Very long press.\n";
        break;
    default:
        std::cerr << "invalid pressType.\n";
        break;
    }
}
```

ButtonEventDispatcher header (ButtonEventDispatcher.hpp):

We modify the ButtonEventDispatcher class to use a Singleton design pattern here. In a hearing device, we only require a single dispatcher and so this ensures only a single instance can be created.

```
#pragma once

#include <list>
#include <algorithm>
#include <iostream>
#include <mutex>

#include "ButtonEvent.hpp"

class ButtonEventDispatcher
{
public:
    // Only allow one instance of ButtonEventDispatcher
    static ButtonEventDispatcher& getInstance()
    {
        if (pInstance == nullptr)
        {
            pInstance = new ButtonEventDispatcher();
        }
        return *pInstance;
    }

    ButtonEventDispatcher(const ButtonEventDispatcher&) = delete;
    ButtonEventDispatcher& operator=(const ButtonEventDispatcher&) = delete;


    bool RegisterButtonHandler(void (*handler)(const ButtonEvent& e));
```

```cpp
    void UnregisterButtonHandler(void (*handler)(const ButtonEvent& e));
    void Dispatch(const ButtonEvent& e) const;

private:
    ButtonEventDispatcher() {}
    ~ButtonEventDispatcher() {}
    static ButtonEventDispatcher *pInstance;

    std::mutex mutex;
    std::list<void (*)(const ButtonEvent&)> _callbacks;
};
```

ButtonEventDispatcher source (ButtonEventDispatcher.cpp):

Here we modify the return type of the RegisterButtonHandler method to a Boolean as, in an embedded context, we may not have a terminal to know whether the code executed successfully. We also introduce a mutex to ensure that multiple threads cannot access the same list of callback functions, avoiding conflicts.

```cpp
#include "ButtonEventDispatcher.hpp"

constexpr unsigned int cHandlersSizeMax = 4;

bool ButtonEventDispatcher::RegisterButtonHandler(void (*handler)(const
ButtonEvent& e))
{
    bool result = false;
    mutex.lock();

    if(_callbacks.size() < cHandlersSizeMax)
    {
        _callbacks.push_back(handler);
        result = true;
    }
    else
    {
        std::cout << "\nError, exceeded number of allowed handlers.\n";
        result = false;
    }

    mutex.unlock();
    return result;
};

void ButtonEventDispatcher::UnregisterButtonHandler(void (*handler)(const
ButtonEvent& e))
{
    auto it = std::find(_callbacks.begin(), _callbacks.end(), handler);
    if (it != _callbacks.end())
    {
        _callbacks.erase(it);
    }
};

void ButtonEventDispatcher::Dispatch(const ButtonEvent& e) const
{
    for (auto callback : _callbacks)
    {
        if (callback != nullptr)
        {
            callback(e);
        }
    }
};
```

We then built some test classes to verify the functionality. Firstly we define handlers:

ButtonPressHandlers header (ButtonPressHandlers.hpp):

```
#pragma once

#include "ButtonEvent.hpp"

void ButtonPressHandlerA(const ButtonEvent& event);
void ButtonPressHandlerB(const ButtonEvent& event);
void ButtonPressHandlerC(const ButtonEvent& event);
void ButtonPressHandlerD(const ButtonEvent& event);
void ButtonPressHandlerE(const ButtonEvent& event);
void ButtonPressHandlerF(const ButtonEvent& event);
```

ButtonPressHandlers source (ButtonPressHandlers.cpp):

```
#include <iostream>

#include "ButtonPressHandlers.hpp"

void ButtonPressHandlerA(const ButtonEvent& event)
{
    std::cout << "In ButtonPressHandler 'A' " << ButtonPressHandlerA <<
std::endl;
    event.print();
}

void ButtonPressHandlerB(const ButtonEvent& event)
{
    std::cout << "In ButtonPressHandler 'B' " << ButtonPressHandlerB <<
std::endl;
    event.print();
}

void ButtonPressHandlerC(const ButtonEvent& event)
{
    std::cout << "In ButtonPressHandler 'C' " << ButtonPressHandlerC <<
std::endl;
    event.print();
}

void ButtonPressHandlerD(const ButtonEvent& event)
{
    std::cout << "In ButtonPressHandler 'D' " << ButtonPressHandlerD <<
std::endl;
    event.print();
}

void ButtonPressHandlerE(const ButtonEvent& event)
{
    std::cout << "In ButtonPressHandler 'E' " << ButtonPressHandlerE <<
std::endl;
    event.print();
}

void ButtonPressHandlerF(const ButtonEvent& event)
{
    std::cout << "In ButtonPressHandler 'F' " << ButtonPressHandlerF <<
std::endl;
    event.print();
}
```

Lastly, we define a main function to run the code:

```
#include <iostream>
#include <string>

#include "ButtonEvent.hpp"
#include "ButtonEventDispatcher.hpp"
#include "ButtonPressHandlers.hpp"

// Singleton design pattern - define an instance
ButtonEventDispatcher* ButtonEventDispatcher::pInstance = nullptr;

int main()
{
    bool result;

    // Create events for button 0
    ButtonEvent button0ShortPress(PressType::cShortPress,
ButtonNumber::cButton0);
    ButtonEvent button0LongPress(PressType::cLongPress,
ButtonNumber::cButton0);
    ButtonEvent button0VeryLongPress(PressType::cVeryLongPress,
ButtonNumber::cButton0);

    // Create events for button 1
    ButtonEvent button1ShortPress(PressType::cShortPress,
ButtonNumber::cButton1);
    ButtonEvent button1LongPress(PressType::cLongPress,
ButtonNumber::cButton1);
    ButtonEvent button1VeryLongPress(PressType::cVeryLongPress,
ButtonNumber::cButton1);

    // Get a reference to the single button event dispatcher instance.
    ButtonEventDispatcher& buttonEventDispatcher =
        ButtonEventDispatcher::getInstance();

    // Attempt to register six handlers.
    // Should only register the first four (A, B, C & D)
    result = buttonEventDispatcher.RegisterButtonHandler(ButtonPressHandlerA);
    result = buttonEventDispatcher.RegisterButtonHandler(ButtonPressHandlerB);
    result = buttonEventDispatcher.RegisterButtonHandler(ButtonPressHandlerC);
    result = buttonEventDispatcher.RegisterButtonHandler(ButtonPressHandlerD);
    result = buttonEventDispatcher.RegisterButtonHandler(ButtonPressHandlerE);
    result = buttonEventDispatcher.RegisterButtonHandler(ButtonPressHandlerF);

    // Verify that handlers A, B, C & D have been called.
    buttonEventDispatcher.Dispatch(button0ShortPress);

    // Unregister the first two handlers (A & B).
    buttonEventDispatcher.UnregisterButtonHandler(ButtonPressHandlerA);
    buttonEventDispatcher.UnregisterButtonHandler(ButtonPressHandlerB);

    // Verify that handlers C & D have been called.
    buttonEventDispatcher.Dispatch(button1ShortPress);

    // Attempt to register handlers E & F again, now they will get registered.
    result = buttonEventDispatcher.RegisterButtonHandler(ButtonPressHandlerE);
    result = buttonEventDispatcher.RegisterButtonHandler(ButtonPressHandlerF);

    // Verify that handlers C, D, E & F have been called.
    buttonEventDispatcher.Dispatch(button0LongPress);

    // Unregister handlers C, D, E & F.
    buttonEventDispatcher.UnregisterButtonHandler(ButtonPressHandlerC);
    buttonEventDispatcher.UnregisterButtonHandler(ButtonPressHandlerD);
    buttonEventDispatcher.UnregisterButtonHandler(ButtonPressHandlerE);
    buttonEventDispatcher.UnregisterButtonHandler(ButtonPressHandlerF);

    // Verify that all handlers have been unregistered.
    buttonEventDispatcher.Dispatch(button1LongPress);

    // Attempt to unregister handlers A & B again, nothing should happen.
    buttonEventDispatcher.UnregisterButtonHandler(ButtonPressHandlerA);
    buttonEventDispatcher.UnregisterButtonHandler(ButtonPressHandlerB);

    // Verify that nothing happened.
    buttonEventDispatcher.Dispatch(button1LongPress);
}
```

Compiling and running gives the following output:

```
[11ntye/Documents]$./TestButtonDispatchHandler

Error, exceeded number of allowed handlers.

Error, exceeded number of allowed handlers.

In ButtonPressHandler 'A' 1
Button #0, Short press.
In ButtonPressHandler 'B' 1
Button #0, Short press.
In ButtonPressHandler 'C' 1
Button #0, Short press.
In ButtonPressHandler 'D' 1
Button #0, Short press.
In ButtonPressHandler 'C' 1
Button #1, Short press.
In ButtonPressHandler 'D' 1
Button #1, Short press.
In ButtonPressHandler 'C' 1
Button #0, Long press.
In ButtonPressHandler 'D' 1
Button #0, Long press.
In ButtonPressHandler 'E' 1
Button #0, Long press.
In ButtonPressHandler 'F' 1
Button #0, Long press.
```

Thus we have verified the behaviour is as expected.