Spike: Spike No.21
Title: Task 21 – Messaging

Author: Khang Trinh - 102118468

Goals / Deliverables:

The goal of this spike is to teach the developer how to implement a basic one-to-one messaging system.

Technologies, Tools, and Resources used:

- Visual Studio 2017

Useful Links:

1. Explanation and example of implementation: https://seanballais.github.io/blog/implementing-a-simple-message-bus-in-cpp/

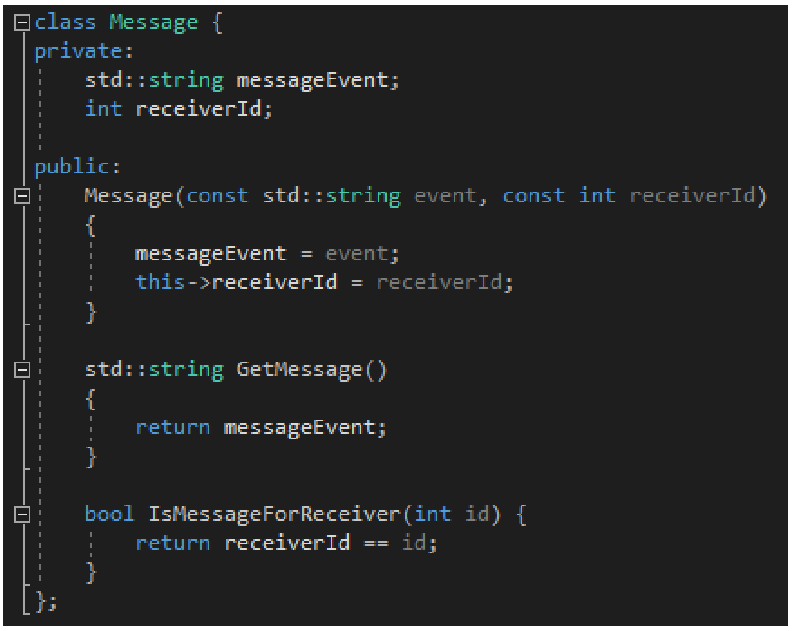
Tasks undertaken:

A messaging system requires 3 main parts:

- The message being sent
- A container of the messages that can store/send the message
- Senders/Receivers of the message

Step 1: The message class

There are 2 things that a message needs: the content, and the detail of the receiver. For this example, we'll determine a receiver by assigning them a unique id (in this case, a unique int).

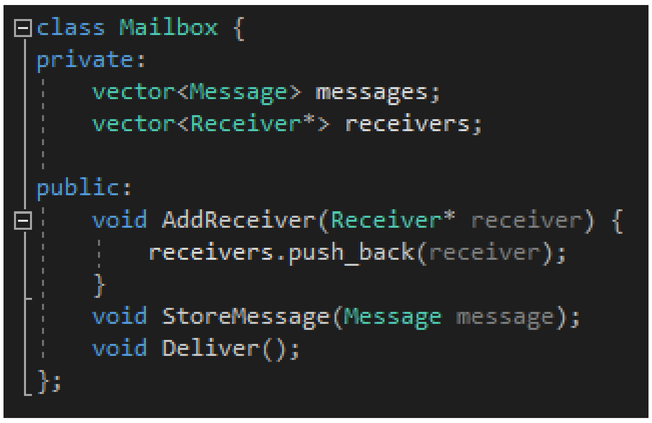


```
class Message {
private:
    std::string messageEvent;
    int receiverId;

public:
    Message(const std::string event, const int receiverId)
    {
        messageEvent = event;
        this->receiverId = receiverId;
    }

    std::string GetMessage()
    {
        return messageEvent;
    }

    bool IsMessageForReceiver(int id) {
        return receiverId == id;
    }
};
```

It's also useful to implement a method that checks whether a message belongs to a receiver by comparing the receiver id with the id in the message as well (this allows the ids to be kept private).

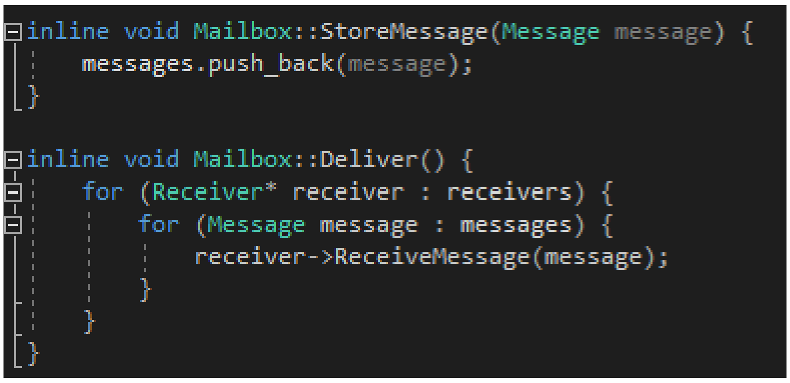
Step 2: The container

Think of this like a mailbox. It contains a list of messages to deliver and a list of addresses to the people who will be receiving those messages.



```
class Mailbox {
private:
    vector<Message> messages;
    vector<Receiver*> receivers;

public:
    void AddReceiver(Receiver* receiver) {
        receivers.push_back(receiver);
    }
    void StoreMessage(Message message);
    void Deliver();
};
```

The container should also be responsible for storing and delivering messages to the subscribed components as well.

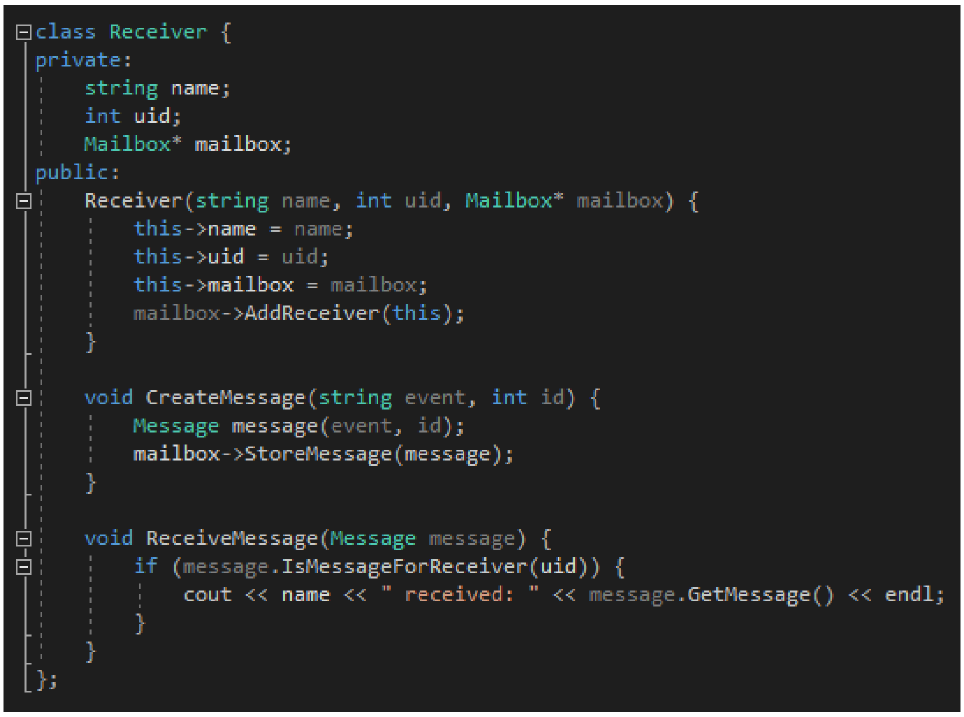


```
inline void Mailbox::StoreMessage(Message message) {
    messages.push_back(message);
}

inline void Mailbox::Deliver() {
    for (Receiver* receiver : receivers) {
        for (Message message : messages) {
            receiver->ReceiveMessage(message);
        }
    }
}
```

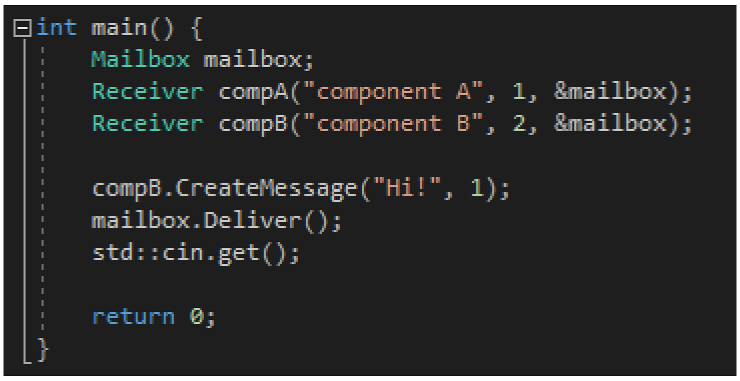
Step 3: The receiver

The receiver requires a unique id like we mentioned before, and a reference to the mailbox to subscribe to and send messages to other receivers.

When creating a message, it needs to provide the content, as well as the id to the other receiver.



```
class Receiver {
private:
    string name;
    int uid;
    Mailbox* mailbox;
public:
    Receiver(string name, int uid, Mailbox* mailbox) {
        this->name = name;
        this->uid = uid;
        this->mailbox = mailbox;
        mailbox->AddReceiver(this);
    }

    void CreateMessage(string event, int id) {
        Message message(event, id);
        mailbox->StoreMessage(message);
    }

    void ReceiveMessage(Message message) {
        if (message.IsMessageForReceiver(uid)) {
            cout << name << " received: " << message.GetMessage() << endl;
        }
    }
};
```



```
int main() {
    Mailbox mailbox;
    Receiver compA("component A", 1, &mailbox);
    Receiver compB("component B", 2, &mailbox);

    compB.CreateMessage("Hi!", 1);
    mailbox.Deliver();
    std::cin.get();

    return 0;
}
```

What we found

- Using a messaging system allows for decoupling in code. Sometimes there will be a need to call multiple methods executing from one message (ie. Sending “Take damage 10” to a visual component that will process “Take damage” to execute a blood splatter particle effect, while sending that same message to a health component will take that and process “10” into reducing hp by 10). The classes don’t need to know the existence of another class, only the message they receive.
- In our example, we manually assigned unique ids to the component. You may want to implement a more robust system that’s capable of automatically assigning unique ids to the receivers. This can also allow for a more complex id system rather than the simple integer id as well.
- We also combined both the act of “sending” and “receiving” into the same class. You may want to consider separating them if you feel the need to.
- Currently the components aren’t able to process the message it receives, only read the content. This can be expanded to work with the command pattern so that the receiver can have a message processor that will split the message into parts and use it as parameters.