DAY3 1번 과제

2023741024
로봇학부 박건후

목차

(1) mainwindow.h 파일 설명

```
private slots:
    void handleKey(int id); //버튼 눌렀을 때 발동됨
    void commitTimeout();   //타임아웃에 대한 슬롯 메서드
    void toggleLang();      // 한/영 키 바꾸기
    void toggleShift();     //시프트 키 눌렸을 때
    void saveToTxt();       //엔터키 눌렸을 때 저장
```

slots 선언 부분입니다. 주석에 작성한 내용대로, 버튼이 눌렸을 때 작동하는 것을 많이 사용하게 되었습니다. 각 버튼에 따라 기능이 다른 특수한 버튼이 있기 때문입니다. 시프트 버튼이나 엔터키 등이 그 예입니다.

```
private:
    // ---- 고정 키 ID ----
    int BACK_ID  = 9;   //  뒤로가기 버튼 번호
    int SPACE_ID = 20;  // 스페이스 버튼 번호
    int BLANK_ID = 21;  // 동작 없는 공백 버튼 번호

    Ui::MainWindow *ui;
    QButtonGroup   *keys = nullptr; //버튼을 그룹화하여 한 곳에서 관리하기 위해
    QTimer          multiTimer;

    enum class Layout { EnLower, EnUpper, Ko }; //현재 한글인지 영어의 소문자
    Layout layout = Layout::EnLower;

    int  lastKeyId     = -1;
    int  cycleIndex    = 0;
    int  lastInsertPos = -1;

    QString savePath; //경로 설정을 위해

    // 키배열(멀티탭 후보)
    QHash<int, QStringList> mapEnLower, mapEnUpper, mapKo;  //버튼 번호와

    // 버튼 포인터 테이블 (라벨 갱신용)
    QHash<int, QPushButton*> btns;  //버튼 번호와 버튼의 포인터와의 매치
```

private 에 선언한 변수, 선언된 private 멤버함수입니다. 제가 이번에 특정 키를 몇 번 눌렀을 때 입력창에 어떤 문자를 출력하는지를 계산하기 위해 눌렀던 키 버튼의 번호와 키 버튼을 몇 번 눌렀는지를 멤버변수로 설정하여 이 값에 따른 문자를 출력하도록 설계했습니다. 그리고 lastInsertPos 변수는 입력창에 출력할 때 커서의 위치를 저장하는 변수입니다.

```
private:
    // ---- 고정 키 ID ----
    int BACK_ID  = 9;   //  뒤로가기 버튼 번호
    int SPACE_ID = 20;  // 스페이스 버튼 번호
    int BLANK_ID = 21;  // 동작 없는 공백 버튼 번호

    Ui::MainWindow *ui;
    QButtonGroup   *keys = nullptr; //버튼을 그룹화하여 한 곳에서 관리하기 위해
    QTimer          multiTimer;

    enum class Layout { EnLower, EnUpper, Ko }; //현재 한글인지 영어의 소문자
    Layout layout = Layout::EnLower;

    int  lastKeyId     = -1;
    int  cycleIndex    = 0;
    int  lastInsertPos = -1;

    QString savePath; //경로 설정을 위해

    // 키배열(멀티탭 후보)
    QHash<int, QStringList> mapEnLower, mapEnUpper, mapKo;  //버튼 번호와

    // 버튼 포인터 테이블 (라벨 갱신용)
    QHash<int, QPushButton*> btns;  //버튼 번호와 버튼의 포인터와의 매치
```

이번 1 번 과제를 풀면서 제가 느끼기로는 특정 값에 대응되는 값을 연결짓는 것이 많이 필요하다고 느꼈습니다. 그래서 연결짓는 것으로 제일 좋은 자료구조인 해시테이블을 사용하고자 했습니다. 처음에는 해시테이블을 버튼의 번호와 영어 문자를 연결짓는 것에 이어서 영어 대문자, 한글까지 이를 추가적으로 선언하여 사용하게 되었고 버튼의 번호와 실제 그 버튼을 제어하기 위해 버튼의 번호와 ui->PushButton 을 넣어 btns 라는 변수까지 선언하게 되었습니다.

그리고 영어 소문자일 때, 영어 대문자일 때, 한글일 때마다 규칙이 조금씩 다른 것이 있습니다. 그러다보니 private 함수를 작성하면서 조건식을 매번 써줘야 했었는데 이를 편리하게 하기 위해 상태를 나타내는 변수를 추가하여 만든 것입니다. 그래서 열거형 클래스를 만들어서 쉽게 조건식을 작성할 수 있도록 만든 것입니다.

```
    // 내부 유틸
    void setLayout(Layout l);   //버튼에 출력되는 문자열 초기화할 때 사용
    void startOrCycle(int keyId, const QStringList &candidates); //입력창에 
    void replaceLast(const QString &ch);    //2번 같은 키를 연속으로 누를 때 그 때
    void insertNew(const QString &ch);      //새 문자 삽입
    void doBackspace();                     //뒤로 가기 버튼 눌렀을 때 처리

    // 라벨 갱신
    void refreshKeyLabels();                //이걸로 버튼에 출력되는 문자열 초기화
    const QHash<int, QStringList>& currentMap() const;
    QString labelFor(int id, const QHash<int, QStringList>& m) const;
};

#endif // MAINWINDOW_H
```

이어서 몇 개의 추가적인 private 멤버함수들입니다. 이 멤버함수들 또한 mainwindow.cpp 파일을 보여드리며 설명드리겠습니다.

(2) mainwindow.cpp 파일

(2-1) 기본적인 초기화

```
//키 버튼 등록
keys->addButton(ui->pushButton_10, 10);     //그룹화를 시킴
keys->addButton(ui->pushButton_11, 11);
keys->addButton(ui->pushButton_12, 12);
keys->addButton(ui->pushButton_30, 30);
keys->addButton(ui->pushButton_29, 29);
keys->addButton(ui->pushButton_28, 28);
keys->addButton(ui->pushButton_26, 26);
keys->addButton(ui->pushButton_25, 25);
keys->addButton(ui->pushButton_24, 24);
keys->addButton(ui->pushButton_23, 23);
keys->addButton(ui->pushButton_21, BLANK_ID); // 동작 없는 공백 버튼
keys->addButton(ui->pushButton_19, 19);
keys->addButton(ui->pushButton_20, SPACE_ID); //  스페이스
keys->addButton(ui->pushButton_9,  BACK_ID);  //  백스페이스

connect(keys, SIGNAL(idClicked(int)), this, SLOT(handleKey(int)));


// 영어(소문자)
mapEnLower[10] = { ".",",","?","!" };
mapEnLower[11] = { "a","b","c" };
mapEnLower[12] = { "d","e","f" };
mapEnLower[30] = { "g","h","i" };
mapEnLower[29] = { "j","k","l" };
mapEnLower[28] = { "m","n","o" };
mapEnLower[26] = { "p","q","r","s" };
mapEnLower[25] = { "t","u","v" };
mapEnLower[24] = { "w","x","y","z" };
mapEnLower[19] = { "," };
mapEnLower[23] = { "<SHIFT>" };
mapEnLower[BACK_ID]  = { "<BACK>" };
mapEnLower[SPACE_ID] = { "<SPACE>" };
```

기본적인 초기화는 항상 main.cpp 에서 객체가 생성될 때 같이 초기화되는 것이 좋으므로 MainWindow 객체가 생성되는 생성자 본문에서 모두 초기화 작업을 해줬습니다. keys 는 QButtonGroup 클래스로써 keys 가 가리키는 동적할당 객체에 해당 버튼을 모두 추가해주는 작업입니다. 이렇게 설계한 이유는 여러 개의 버튼을 하나의 변수로 동시에 처리하기 위함입니다. 그러면 굳이 매번 ui->pushButton_n 을 넣어줄 필요 없이 각각의 n 값으로 매핑이 되기 때문에 버튼별 관리가 편해집니다. 이를 보여주는 부분이 keys 초기화 후 바로 아래 connect 하는 부분입니다. 이렇게 되면 모든 버튼이 눌릴 때마다 handleKey 로 넘어가면서 각 버튼마다 하드코딩할 필요를 없애줍니다.

그 이후에는 mapEnLower 라는 해시테이블을 초기화하는 부분입니다.

```
mapEnUpper = mapEnLower;
for (auto &v : mapEnUpper)
    for (QString &s : v)
        if (s.size()==1 && s[0].isLetter()) s = s.toUpper();

// 한글

mapKo[10] = { "ㅣ" };
mapKo[11] = { "." };
mapKo[12] = { "ㅡ" };
mapKo[30] = { "ㄱ","ㅋ" };
mapKo[29] = { "ㄴ","ㄹ" };
mapKo[28] = { "ㄷ","ㅌ" };
mapKo[26] = { "ㅂ","ㅍ" };
mapKo[25] = { "ㅅ","ㅎ" };
mapKo[24] = { "ㅈ", "ㅊ" };
mapKo[19] = { "한자" };
mapKo[23] = { ".",",","?","!"  };
mapKo[BACK_ID]  = { "<BACK>" };
mapKo[SPACE_ID] = { "<SPACE>" };
mapKo[BLANK_ID]= {"ㅇ", "ㅁ"};

// 타이머
multiTimer.setInterval(2000);
multiTimer.setSingleShot(true);
connect(&multiTimer, &QTimer::timeout, this, &MainWindow::commitTimeout);

// 초기 레이아웃 즉 버튼 문자열 초기화
setLayout(Layout::Ko); // 시작을 한글로 보고 싶다면 Ko, 아니면 EnLower
connect(ui->pushButton_31, &QPushButton::clicked, this, &MainWindow::toggleLang);
```

그리고 영문자의 경우에는 toUpper 라는 메서드를 통해 대문자로 만드는 작업을 범위기반 for 문으로 진행하였습니다.

한글에 대한 해시테이블 또한 초기화해줬습니다.

그리고 2 초 동안 연속으로 같은 키를 누르지 못했을 때 다음 위치로 커서가 넘어가도록 하는 것을 만들어서 천지인의 원리와 같은 특성을 갖도록 만들었습니다. setLayout 으로 한글, 영어 소문자, 대문자 등으로 버튼의 문자열을 초기화하는 것을 하는 것이 아랫줄에 나와 있습니다.

```
btns = {
    {10, ui->pushButton_10}, {11, ui->pushButton_11}, {12, ui->pushButton_12},
    {30, ui->pushButton_30}, {29, ui->pushButton_29}, {28, ui->pushButton_28},
    {26, ui->pushButton_26}, {25, ui->pushButton_25}, {24, ui->pushButton_24},
    {23, ui->pushButton_23}, {BLANK_ID, ui->pushButton_21}, {19, ui->pushButton_19},
    {SPACE_ID, ui->pushButton_20}, {BACK_ID, ui->pushButton_9}
};
```

그 이후에 btns 라는 버튼 해시테이블을 초기화하는 부분입니다. 버튼의 번호와 버튼을 실제 다루기 위한 포인터를 묶음으로써 한 번에 쉽게 버튼을 다루려고자 선언하였습니다.

(2-2) 버튼 표시 관련 메서드

```
void MainWindow::setLayout(Layout l) {
    layout = l;
    lastKeyId  = -1;
    cycleIndex = 0;
    lastInsertPos = -1;
    refreshKeyLabels();
}
```

이전에 설명드린 setLayout 부터 보겠습니다. 여기서는 Layout 을 받아서 그것으로 초기화를 합니다. 이 메서드는 초반부에만 초기화하는 용도로 만든 것이기에 lastKeyId 나 커서의 위치를 -1 로 초기화한 것입니다. 그리고 버튼에 출력을 해야 하니 layout 으로 받은 상태로 버튼에 출력하는 것을 해줄 refreshKdyLabels 를 호출합니다.

```
void MainWindow::refreshKeyLabels() {
    const auto &m = currentMap();
    for (auto it = btns.constBegin(); it != btns.constEnd(); ++it) {
        const int id = it.key();
        if (QPushButton *b = it.value()) {
            const QString lab = labelFor(id, m);
            if (!lab.isEmpty())
                b->setText(lab);
        }
    }
}
```

이 함수는 currentMap 이라는 것으로부터 layout 상태를 받아오도록 했습니다. 그리고 버튼의 처음부터 끝까지 모두 문자열을 출력해줘야 하기에 btns 를 사용하여 버튼 목록의 처음부터 끝까지 참조하여 이에 대해 labelFor 메서드를 따로 호출해서 그 메서드로부터 해당 버튼에 어떤 문자열을 출력해야 하는지 문자열 통째로 받아와서 이를 setText 로 버튼에 출력하도록 한 것입니다.

```cpp
const QHash<int, QStringList>& MainWindow::currentMap() const {
    switch (layout) {
    case Layout::EnLower: return mapEnLower;
    case Layout::EnUpper: return mapEnUpper;
    case Layout::Ko:      return mapKo;
    }
    return mapEnLower; // fallback
}

//뭘 표시해야 할지 여기서 정함
QString MainWindow::labelFor(int id, const QHash<int, QStringList>& m) const {
    // 특수키 라벨
    if (id == 23){
        if(layout == Layout::Ko) return QStringLiteral(".,?!");
        return QStringLiteral("⬆");   // shift
    }
    if (id == BACK_ID)  return QStringLiteral("⬅");   // backspace
    if (id == SPACE_ID) return QStringLiteral("␣");   // space 표시
    if (id == BLANK_ID) {
        if(layout == Layout::Ko) return QStringLiteral("ㅇㅁ");
        return QString();             // 공백 버튼: 라벨 유지/빈칸
    }
    if (!m.contains(id) || m.value(id).isEmpty()) return QString();
    const QStringList &c = m.value(id);

    if (c.size() == 1) return c.front();

    QString lab;
    for (const QString &s : c) lab += s;
    return lab;
}
//여기서 버튼 문자열 추가함
```

위의 currentMap 은 layout 상태를 받아오는 것이고, labelFor 는 QstringList 로 서로 떼어져 있던 문자열을 +라는 연산자 중복정의를 통해 하나로 붙이고 이를 리턴하여 setText 를 할 수 있도록 하는 것입니다. 이 때 띄어쓰기라든가 문자 삭제, 영어 문자에서의 아무 문자도 없는 빈칸에 대해 경우로 나눠 이에 대해 특별한 처리를 하였습니다. 또한 영어일 때와 한글일 때의 버튼에 출력되는 것이 다르고 그 버튼에 출력되는 것에 따라 실제 작동해야 하기에

```cpp
if (id == 23){
    if(layout == Layout::Ko) return QStringLiteral(".,?!");
    return QStringLiteral("⬆");   // shift
}
```

```cpp
if (id == BLANK_ID) {
    if(layout == Layout::Ko) return QStringLiteral("ㅇㅁ");
    return QString();             // 공백 버튼: 라벨 유지/빈칸
}
```

등의 한글의 경우에 한해 이렇게 버튼에 문자열이 출력되도록 하였습니다.

(2-3) 입력창에 문자 출력 관련 메서드

```
void MainWindow::handleKey(int id) {
    const auto &m = currentMap();

    // SHIFT
    if (id == 23 && m.value(23).contains("<SHIFT>")) {
        toggleShift();
        return;
    }
    else if(id == 23 && m.value(23).contains(".")){
        startOrCycle(id, {".",",", "?", "!"});
        return;
    }
    // BACKSPACE
    if (id == BACK_ID) {
        doBackspace();
        commitTimeout();
        return;
    }
    //띄어쓰기 문자 버튼일 경우
    if (id == SPACE_ID) {
        if(cycleIndex==0){
            insertNew(" ");
            commitTimeout();
        }
        else{
            ui->lineEdit->setCursorPosition(ui->lineEdit->cursorPosition()+1);
            commitTimeout();
        }
        return;
    }
```

특정 버튼을 입력받은 시그널이 들어오고 이에 대한 슬롯 메서드인 handleKey 가 호출되는 방식입니다. 시프트일 때 영어인 경우 대, 소문자를 바꾸고 한글일 때는 . , ? !를 startOrCycle 이라는 특정 버튼의 문자열을 출력해주는 함수를 호출하여 출력하도록 했습니다. 그리고 그 외의 예외 버튼에 대해서도 각각에 맞는 처리를 해주었습니다.

```
    if (id == BLANK_ID) {
        if(layout == Layout::Ko) startOrCycle(id, {"ㅇ","ㅁ"});
        return;
    }

    // 맵에 후보 없는 키는 무시
    if (!m.contains(id) || m.value(id).isEmpty()) return;

    // 기본 처리
    startOrCycle(id, m.value(id));
}
```

그리고 영문자의 경우에 완전히 아무 문자열이 없는 버튼에서 한국 버전에는 ㅇ, ㅁ이 나오기에 이데 대해 처리를 해줬습니다. 그리고 위의 예외 버튼의 경우가 아닐 때에는

startOrCycle 을 호출하여 2 번째 매개변수에 들어가는 QStringList 를 누른 횟수에 맞게 알맞은 문자열을 출력하도록 만들었습니다.

```
void MainWindow::startOrCycle(int keyId, const QStringList &candidates) {
    QStringList seq = candidates;
    if (layout == Layout::Ko) {
        switch (keyId) {
        case 30: seq << QStringLiteral("ㄲ"); break; // ㄱ,ㅋ,ㄲ // ㄲ을 candidates에 추가
        case 28: seq << QStringLiteral("ㄸ"); break; // ㄷ,ㅌ,ㄸ
        case 26: seq << QStringLiteral("ㅃ"); break; // ㅂ,ㅍ,ㅃ
        case 25: seq << QStringLiteral("ㅆ"); break; // ㅅ,ㅎ,ㅆ
        case 24: seq << QStringLiteral("ㅉ"); break; // ㅈ,ㅊ,ㅉ
        default: break;
        }
    }
    //같은 버튼을 연속으로 눌렀을 때
    if (keyId == lastKeyId && multiTimer.isActive()) {
        cycleIndex = (cycleIndex + 1) % seq.size();
        replaceLast(seq.at(cycleIndex));   // .at()을 쓰면 디버그에서 범위 체크도 됨
        multiTimer.start();
        return;
    }

    //해당 버튼을 처음 눌렀을 때
    lastKeyId  = keyId;
    cycleIndex = 0;
    insertNew(seq.first());
    multiTimer.start();
}
```

이 때 코딩 당시 문제가 생긴 부분은 된소리 출력이었습니다, 이 부분은 버튼 텍스트에 나와 있지는 않지만 3 번 눌렀을 때 저 된소리가 출력되어야 한다는 것이었습니다. 그래서 이 된소리를 해당 candidates 목록에 추가할 방법을 검색하여 <<연산자로 된소리 문자열을 추가할 수 있다는 것을 알게 되었고 이를 추가하여 같은 버튼을 연속으로 눌렀을 경우에 대해 문제 없이 출력할 수 있었습니다. 같은 버튼을 연속으로 눌렀다는 것을 확인하는 것에서는 lastKeyId 를 통해 가능했었습니다. 그리고 cycleIndex 로 몇 번 출력된 건지를 통해 QStringList 의 몇 번째 문자열을 출력해야 되는 것인지 알 수 있었습니다.

```
void MainWindow::replaceLast(const QString &ch) {
    QLineEdit *le = ui->lineEdit;
    if (!le) return;

    if (lastInsertPos < 0 || lastInsertPos > le->text().size()-1) {
        insertNew(ch);
        return;
    }
    //
    QString t = le->text();
    t.replace(lastInsertPos, 1, ch);
    le->setText(t);
    le->setCursorPosition(lastInsertPos + ch.size());
}

void MainWindow::insertNew(const QString &ch) {
    QLineEdit *le = ui->lineEdit;
    if (!le) return;
    int pos = le->cursorPosition();
    le->insert(ch);
    lastInsertPos = pos;
}
```

그리고 위 코드에서 나온 replaceLast 를 설명드리겠습니다. 이는 같은 키를 여러 번 눌렀을 때 그 때 커서가 움직이지 않고 그 위치에 고정되며 계속 출력되게 만들어야 했습니다. 이를 만들기 위해 lineEdit 의 setCursorPosition 메서드를 사용하여 커서 위치를 통제할 수 있었습니다. 그리고 insertNew 에서는 새로운 문자를 추가하는 것을 따로 메서드화 해서 만든 것으로 언제든 새로운 문자를 추가하게 될 때 이를 사용하도록 한 것입니다.

(2-4) 그 외의 메서드

```
//문자 삭제에 관한 커서위치와 출력 문자 처리
void MainWindow::doBackspace() {
    QLineEdit *le = ui->lineEdit;
    if (!le) return;

    int pos = le->cursorPosition();
    if (pos <= 0) return;

    QString t = le->text();
    t.remove(pos - 1, 1);
    le->setText(t);
    le->setCursorPosition(pos - 1);

    lastKeyId  = -1;
    cycleIndex = 0;
    lastInsertPos = -1;
}
```

이 함수는 문자 삭제하는 뒤로 가기 화살표를 동작시키는 것을 메서드로 만든 것입니다. 이 때도 lineEdit 를 사용하여 remove 를 통해 해당 글자를 지우고 커서의 위치를 1 줄이며 해당 버튼을 누르면 같은 키를 연속으로 누르는 것이 의미 없게 되므로 다시 관련 값을 초기화하는 것까지 진행했습니다.

```
void MainWindow::saveToTxt()
{
    const QString text = ui->lineEdit->text().trimmed();
    if (text.isEmpty()) return;

    QFile f(savePath);

    if (!f.open(QIODevice::WriteOnly | QIODevice::Append | QIODevice::Text)) {
        // 열기 실패 시 return
        return;
    }
    QTextStream out(&f);
    out << text << '\n';
    out.flush();
    f.close();

    ui->lineEdit->clear();
}
```

이는 엔터키를 누르면 저장하는 것을 수행하는 함수입니다. 스트림을 통해 문자를 파일에 출력을 하고 버퍼를 flush 로 비우는 것을 하고 그 후에는 lineEdit 에 있는 것을 모두 지우는 것이 맞으므로 clear 를 한 것입니다.

**2025.09.18에 수정한 부분: 기존에는 message.txt라는 파일을 직접 생성한 후에야 해당 파일에 저장하는 것이 가능했지만 이번에는 파일을 미리 만들지 않더라도 파일이 없다면 message.txt 파일을 생성하고 저장하도록 하는 기능을 추가하였습니다. 추가된 부분은

```
if (!f.open(QIODevice::WriteOnly | QIODevice::Append | QIODevice::Text))
```

이 사진의 부분으로써 https://blog.naver.com/browniz1004/220707555543 이 링크의

> ## 파일쓰기
> ```
> File.open(QFile::WriteOnly|QFile::Append|QFile::Text)) // 쓰기 전용, 텍스트, 이어쓰기
> QTextStream SaveFile(&File);
> SaveStream<<"저장될내용"<<"\n"<<;  // 맨뒤 \n은 줄바꿈
> SaveStream<<"저장될내용"<<"\n"<<; // 이런식으로 계속 추가 하면된다
> File.close(); // 파일닫기
> ```

이 부분을 참고하여 작성하였습니다. 감사합니다.