程序设计实习Qt大作业报告：HachiDraw

“直接上手过柱子”组 王妙杰 蔡雨阳 董洋

一、程序功能概述

1.1 基本功能

本项目作为基于Qt的业内优秀的化学绘图软件——**ChemDraw**的仿制品，支持如下功能：

1. 原子与键、文本的绘制与写入：点选单键图标即可拉动生成确定长度的化学键；点选文本图标即可单击输入文字。
2. 原子和键、文本的移动和删除：点选鼠标图标即可拖动画出的化学结构，同时还可以拖出虚线框，同时拖动框内所有结构；点选橡皮图标即可删除画出的化学结构，包括原子、键和文本。
3. 环状结构的自动绘制：点选多元环图标后可自动计算环上原子坐标，支持拖拽成环和在键上单点两种成环模式，且会自动合并原子。
4. 文件读写：`Ctrl+S`和`Ctrl+O`快捷键分别对应于保存文件和打开文件，对于已经保存的文件也支持实时保存到原地址。

1.2 主要特点

本项目的主要特点是**可持久化设计**、**数据关联**与**动态交互**。

1. **可持久化设计**：每进行一个操作就时刻将整个画布上的内容保存到负责记录历史状态的`QStack`中，若按下`Ctrl+Z`即可弹出`QStack`内容实现撤销。
2. **数据关联**：任何增删原子/键操作都会影响所有关联原子/键的内容，程序使用指针操作确保了这一点。
3. **动态交互**：程序中存在鼠标吸附功能，在一定像素距离内会标**红**对应的原子或化学键，以便于操作。

二、项目各模块与类设计细节

2.1 主要模块组成

本项目由4个主要模块组成：

`main.cpp`：项目的主程序，负责调整画布和工具栏的整体布局，同时从`images.qrc`中导入对应的图像资源。

`mainwindow.h`：负责定义使用到的主要类，如`Atom`类，`Bond`类，`Widget`类等等。

`mainwindow.cpp`：项目的主程序，包含`mainwindow.h`中定义的类及其成员函数的实现。

`images.qrc`：项目的资源文件，包含工具栏初始与被点下后的图片资源，是项目可跨机器运行的重要保证。

2.2 主要类构成

`Atom`类，从Qt自带的`QPointF`类中继承而来，可以使用父类的`rx()`,`ry()`函数以访问原子的位置。同时加入了element代表元素符号，初始默认是C（碳元素）。使用`QVector`记录相连的键，以保证最重要的数据关联。

```
class Atom : public QPointF { //原子
public:
    QString element; //原子的代码
    QVector<Bond> bonds; //相连的键
    Atom(double x=0, double y=0, QString _element = "C");
    Atom(QPointF a);
    Atom(Atom& a);
    Atom(const Atom& a);
    Atom& operator=(Atom a);
    Atom& operator=(QPointF a);
    bool operator==(Atom& a);
    bool operator!=(Atom& a);
};
```

`Bond`类，从Qt自带的`QLineF`类中继承而来，可以使用父类的`p1()`,`p2()`函数以访问两个关联的点，然而此处为确保数据的关联性加入了两个`Atom`类的指针，以代表一根化学键两端的两个原子。成员变量`level`指示的是化学键的键级。

同时，`Bond`类自带`paint`成员函数，该函数负责使原子间的连接更加美观，如双键和三键的绘制及与杂原子相连的键都需要该函数额外进行绘制。

```
class Bond : public QLineF { //化学键
public:
    Atom* atom1;
    Atom* atom2;
    int level = 0;
    Bond();
    Bond(Atom* a, Atom* b, int _level);
    Bond(Bond& a);
    Bond(const Bond& a);
    Bond& operator=(const Bond& a);
    Bond& operator+=(Atom a);
    bool operator==(Bond& a);
    bool operator!=(Bond& a);
    void paint(QPainter& painter);
};
```

`ToolWidget`类和`MyButton`类分别代表工具栏和工具栏中的按钮，前者继承自`QWidget`类，后者继承自`QPushButton`类。

前者主要包含两个成员函数`buildconnect`和`Changemode`。其中`buildconnect`函数利用槽机制将工具栏中模式的切换联系到鼠标光标的切换，以及调用`ChangeMode`函数，进行工具栏模式图标的切换（从`image.qrc`中寻找新的图片）。

`MyButton`类则主要涉及到几个全局静态变量如`mode`，`bondlevel`等，负责管理目前工具栏的模式与新增键的键级。

```
class MyButton: public QPushButton { //各种模式
    Q_OBJECT
public:
    static int mode;
    int buttonmode;
    static int bondlevel;
    MyButton(QWidget *parent = nullptr);
    ~MyButton();
};
class ToolWidget : public QWidget {
    Q_OBJECT
public:
    static MyButton* LastButton;
    void buildconnect(MyButton* button, Widget* window);
public slots:
    void ChangeMode(int mode, MyButton* button){
        //更改mode
        QString s = ":/new/prefix1/images/";
        //更改图片资源
        button->setIcon(QIcon(s));
    }
};
void ToolWidget::buildconnect(MyButton* button, Widget* window){
    connect(button, &QPushButton::clicked, this, [=](){
        ChangeMode(button->buttonmode, button);
        if (button->buttonmode == 0) window->setCursor(Qt::OpenHandCursor);
        else if (button->buttonmode == 5) window->setCursor(Qt::IBeamCursor);
        else window->setCursor(Qt::ArrowCursor);
    });
}
```

`Widget`类，继承自`QWidget`，是主要的画布，其中包含的大部分函数将在下一部分具体介绍。其包含的主要数据结构如下所示。

```
class Widget : public QWidget { //画布
    QVector<Bond> bonds; // 所有化学键
    QVector<Atom*> atoms; // 所有原子
    QVector<QPair<QPointF, QString>> texts; // 所有文本框
    // 历史记录栈
    QStack<QVector<Bond>> BondRecords;
    QStack<QVector<Atom>> AtomRecords;
    QStack<QVector<QPair<QPointF, QString> > > TextRecords;
    //其他变量与函数略去
};
```

2.3 重要函数解析

2.3.1 重要的事件函数

如下六个事件函数是本项目主要的前端函数，负责管理键鼠事件及关闭窗口事件，下面来一一介绍。

```
void mousePressEvent(QMouseEvent* e);
void mouseMoveEvent(QMouseEvent* e);
void mouseReleaseEvent(QMouseEvent* e);
void mouseDoubleClickEvent(QMouseEvent* e);
void keyPressEvent(QKeyEvent* e);
void closeEvent(QCloseEvent* e);
```

限于篇幅，在四个鼠标事件函数中仅以`mousePressEvent`的部分代码为例介绍，其他函数的逻辑大致相同。

开头设置了200ms的延迟，这是因为在Qt中`mouseDoubleClickEvent`和`mousePressEvent`并不冲突，导致前者执行必然会导致后者执行两次，故这里采用延迟方法判定到底是连续双击还是单击。

接下来根据`MyButton:mode`的不同分别执行不同的代码。

若模式为成键模式，则检测是否按下的是鼠标左键，然后开始绘制化学键。在绘制之前先通过调用`CalcAtomDistance`和`CalcBondDistance`分别计算是否应该吸附到画布上的原有的原子和化学键上去，否则自然成键。

若模式为文本模式，则打开一个文本框等待输入，若无输入则回退，否则将新的文本压入`QVector`中。

若模式为橡皮擦除模式，则同样根据`CalcDistance`系列函数分别判定应当擦除什么对象，若无对象可供擦除则直接返回。

若模式为成环模式，则和成键模式一样判断吸附并开始绘制。

最后，若为默认的鼠标模式，则代表可能发生拖拽或框选。根据`CalcDistance`系列函数判断属于哪种，并作出对应的操作。

在会发生更改的操作完毕后，需要统一调用`update()`，这个函数会将后面介绍的`paint`函数压入栈中并发生重绘，这样就能达成画布更新的结果。

```
void Widget::mousePressEvent(QMouseEvent *e) {
    if (e->button() == Qt::LeftButton) {
        if (!timer->isActive()) timer->start(200);  // 启动200ms延迟
        else{
            timer->stop();
            return ;
        }
    }
    if (DoubleClickFlag) return ;
    if (MyButton::mode == 2 || MyButton::mode == 3 || MyButton::mode == 4){ //化学
键
        if (e->button() == Qt::LeftButton) {
            setMouseTracking(false); //在拖拽过程中停止检测自由鼠标
            //先检测吸附过程
            //然后开始拖拽
        }
    }
    else if (MyButton::mode == 5){ //文本输入
        if (e->button() == Qt::LeftButton){
            int f = CalcAtomDistance(Atom(e->pos()));
            if (f != -1){
                //说明输入在原子上
            }
            //输入在别的位置
```

```
            Save();
            texts.push_back({OutputPos, OutputText});
            update();
        }
    }
    else if (MyButton::mode == 1){ //橡皮
        if (e->button() == Qt::LeftButton){
            int f = CalcAtomDistance(e->pos());
            if (f != -1) //擦除原子
            else{
                f = CalcBondDistance(e->pos());
                if (f != -1) //擦除化学键
                else{
                    f = CalcTextDistance(e->pos());
                    if (f != -1) //擦除文本
                    else return ;
                }
            }
            update();
        }
    }
    else if ...//略去
}
```

`keyPressEvent` 函数则主要负责键盘上的快捷键，包括Ctrl+Z(撤销)，Ctrl+S(保存)，Ctrl+O(打开)，Ctrl+C(复制)，Ctrl+V(粘贴)，Delete(删除)等。由于这些指令都是模块化的，因此在本函数中只需要调用并更改一些对象变量就行了。

```
void Widget::keyPressEvent(QKeyEvent* e){
    if (e->modifiers() == Qt::ControlModifier && e->key() == Qt::Key_Z){
        undo();
        m_isDraging = 0;
        Draged_bonds.clear();
        Draged_atoms.clear();
        Draged_texts.clear();
        update();
    }
    else if (e->modifiers() == Qt::ControlModifier && e->key() == Qt::Key_S)
savefile();
    else if (e->modifiers() == Qt::ControlModifier && e->key() == Qt::Key_O)
openfile();
    else if (e->modifiers() == Qt::ControlModifier && e->key() == Qt::Key_C &&
MyButton::mode == 0) Copy();
    else if (e->modifiers() == Qt::ControlModifier && e->key() == Qt::Key_V)
Paste();
    else if (e->key() == Qt::Key_Delete){
        DeleteItems();
        m_isDraging = 0;
        update();
    }
}
```

`closeEvent` 则只关注关闭画布的时候有没有保存文件，若为保存则跳出弹窗提示。

```
void Widget::closeEvent(QCloseEvent* e){
    if (!saved){
        QMessageBox::StandardButton reply = QMessageBox::question(
            this,
            "保存对此文件所做的更改？",
            "要把改动保存到" + File + "中吗？",
            QMessageBox::Yes | QMessageBox::No | QMessageBox::Cancel
            );
        if (reply == QMessageBox::Yes) {
            savefile();
            e->accept();
        }
        else if (reply == QMessageBox::Cancel) e->ignore();
        else e->accept();
    }
}
```

2.3.2 重要的后端函数

如下六个函数是和上述键盘快捷键相关的后端函数，下面简单地一一介绍。

```
void Save(); //保存目前历史状态
void undo(); //撤销
void savefile(); //保存文件
void openfile(); //打开文件
void Copy(); //复制
void Paste(); //粘贴
```

`Save`函数是在每一次画布更新前被调用的函数，它能存储当前画布上的所有数据到`QStack`中。而在`undo`函数被调用的时候则弹出栈更新回原先的数据。

```
void Widget::Save(){
    BondRecords.push(bonds);
    TextRecords.push(texts);
    QVector<Atom> tmp;
    for (int i=0;i<atoms.size();i++) tmp.push_back(*atoms[i]);
    AtomRecords.push_back(tmp);
    saved = 0;
    return ;
}
void Widget::undo(){
    if (AtomRecords.empty()) return ; //没有可回溯空间
    QVector<Atom> tmpAtom = AtomRecords.top(); AtomRecords.pop();
    QVector<Bond> tmpBond = BondRecords.top(); BondRecords.pop();
    texts = TextRecords.top(); TextRecords.pop(); //写入texts
    atoms.clear();
    bonds.clear();
    for (int i=0;i<tmpAtom.size();i++) //写入atoms
    for (int i=0;i<tmpBond.size();i++) //写入bonds
    update();
}
```

`savefile`和`openfile`两个函数分别负责保存文件和打开文件。文件固定为先atoms，后bonds最后texts的格式，便于阅读和理解。

```
void Widget::savefile(){
    if (FilePath == ""){ //未保存的
        FilePath = QFileDialog::getSaveFileName(this, "保存文件",
"C:\\Users\\PC\\Desktop\\untitled", "项目文件(*.hachi)");
    }
    QFile file(FilePath);
    if (file.open(QIODevice::WriteOnly | QIODevice::Text)){
        QTextStream out(&file);
        out << atoms.size() << Qt::endl;
        for(int i=0;i<atoms.size();i++) //输出atoms
        out << bonds.size() << Qt::endl;
        for(int i=0;i<bonds.size();i++) //输出bonds
        out << texts.size() << Qt::endl;
        for(int i=0;i<texts.size();i++) //输出texts
        saved = 1;
        update();
        file.close();
    }
    else if (FilePath != ""){
        QMessageBox::critical(this, tr("错误"), tr("文件保存错误!"));
    }
}
void Widget::openfile(){
    if (!saved) //若当前文件未保存则弹出提示弹窗
    QString filepath = QFileDialog::getOpenFileName(this, "打开文件",
"C:\\Users\\PC\\Desktop", "项目文件(*.hachi)");
    QFile file(filepath);
    if (file.open(QIODevice::ReadOnly | QIODevice::Text)){
        QTextStream in(&file);
        int atomnum, bondnum, textnum;
        in >> atomnum; atoms.clear();
        for (int i=0;i<atomnum;i++) //读取atoms
        in >> bondnum; bonds.clear();
        for (int i=0;i<bondnum;i++) //读取bonds
        in >> textnum; texts.clear();
        for (int i=0;i<textnum;i++) //读取texts
        //更新File名，在画布标题栏上显示，略
        saved = 1;
        update();
        file.close();
    }
    else if (filepath != "") QMessageBox::critical(this, tr("错误"), tr("文件打开错
误!"));
}
```

`Copy`函数将框选选中的结构全部存储到`copied`系列中去，而`Paste`函数则将`copied`系列中存储的结构插入到全局`QVector`中去，完成了复制-粘贴的过程。

```
void Widget::Copy(){
    if (!Draged_atoms.empty() || !Draged_bonds.empty() || !Draged_texts.empty())
{
        // 清空之前的复制内容
        qDeleteAll(copiedAtoms); copiedAtoms.clear(); copiedBonds.clear();
copiedTexts.clear();
```

```
        // 保存复制位置（框选的左上角）
        copyPosition = rect.topLeft();
        for (Atom* atom : Draged_atoms) // 复制选中的原子
        for (Bond* bond : Draged_bonds) // 复制选中的键，需要找到对应的复制后的原子,过程略
        for (int textId : Draged_texts) // 复制选中的文本
        // 清空已框选数组
        Draged_atoms.clear(); Draged_bonds.clear(); Draged_texts.clear();
        update();
    }
}
void Widget::Paste(){
    if (!copiedAtoms.empty() || !copiedTexts.empty()) {
        const QPointF pasteOffset(30, 30);  // 固定偏移量
        QPointF pastePosition = copyPosition + pasteOffset;
        // 检查是否会超出画板
        bool outOfBound = false;
        QRectF widgetRect(0, 0, width(), height());
        // 检查所有原子和文本是否会超出边界
        for (Atom* atom : copiedAtoms) // 检查所有原子是否会超出边界
        for (const auto& text : copiedTexts) // 检查所有文本是否会超出边界
        if (outOfBound) {
            QMessageBox::warning(this, "警告", "粘贴内容将超出画板边界");
            return;
        }
        Save();
        QPointF offset = pastePosition - copyPosition;
        QVector<Atom*> newAtoms;
        for (Atom* atom : copiedAtoms) // 粘贴原子
        for (const Bond& bond : copiedBonds) // 粘贴键，需要找到对应的新原子，过程略
        for (const auto& text : copiedTexts) // 粘贴文本
        // 将框选部分替换为刚粘贴的部分
        Draged_atoms = newAtoms;
        Draged_bonds.clear();
        for (int i = bonds.size() - copiedBonds.size(); i < bonds.size(); i++) {
            Draged_bonds.append(&bonds[i]);
        }
        Draged_texts.clear();
        for (int i = texts.size() - copiedTexts.size(); i < texts.size(); i++) {
            Draged_texts.append(i);
        }
        if (!newAtoms.empty() || !copiedTexts.empty()) {
            ModifyRect(newAtoms, Draged_texts); //框选框自适应化学结构
            copyPosition = pastePosition; // 更新粘贴位置，以便下次粘贴时使用
        }
        m_isDraging = 2;
        update();
    }
}
```

2.3.3 重要的辅助函数

计算距离的`CalcDistance`系列函数：

以`CalcAtomDistance`为例，遍历`atoms`数组中存储的全部的原子，计算距离。若距离小于给定的`threshold`则将其记录下来，并从中选择最短距离进行吸附。

```
int CalcAtomDistance(Atom pos);
int CalcBondDistance(Atom pos);
int CalcTextDistance(Atom pos);
```

```
int Widget::CalcAtomDistance(Atom pos){
    int ans = -1;
    double mn = AtomThreshold;
    for (int i=0;i<atoms.size();i++){
        if (*atoms[i] == *m_startPoint) continue;
        double dx = abs(atoms[i]->x() - pos.x());
        double dy = abs(atoms[i]->y() - pos.y());
        double len = sqrt(dx*dx + dy*dy);
        if (len < mn){
            mn = len;
            ans = i;
        }
    }
    return ans;
}
```

删除各个结构的`Erase`系列函数：

以`EraseAtom`为例，在移去原子的同时，还要移去其所连接的化学键。因此需要遍历`bonds`寻找其所连的其他原子，并将其删去。其他Erase函数也大致相同，不过有的不需要额外删去其他结构。

```
void EraseAtom(int id); //删去原子
void EraseBondForAtom(int id, Atom* other); //在删去原子的过程中删去键
void EraseBond(int id); //删去化学键
void EraseText(int id); //删去文本框
```

```
void Widget::EraseAtom(int id){
    Atom* tmp = atoms[id];
    auto iter = atoms.begin() + id;
    atoms.erase(iter);
    for (int i=0;i<bonds.size();){
        if (*bonds[i].atom1 == *tmp) EraseBondForAtom(i, bonds[i].atom2);
        else if (*bonds[i].atom2 == *tmp) EraseBondForAtom(i, bonds[i].atom1);
        else i++;
    }
    return ;
}
```

当然，还有其他的一些辅助函数，但是由于篇幅限制，以及它们复用性不高，故在此不再赘述。

2.3.4 负责重绘的`paintEvent`事件函数

负责对各种需要绘制的结构进行绘制，每次调用`update`更新函数之后都会发生重绘。

```
void Widget::paintEvent(QPaintEvent *e) {
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);
    painter.setPen(QPen(Qt::black, 2));
    painter.setFont(QFont("Arial", 12));
    for (int i=0;i<bonds.size();i++) //绘制化学键
    for (int i=0;i<atoms.size();i++) //绘制原子
    if (*m_startPoint != Atom(0, 0) && *m_endPoint != Atom(0, 0) && m_isDrawing){
        //说明正在生成一根新键，同样进行绘制
    }
    if (m_isDrawingRing) //正在生成一个环，进行绘制
    QFontMetrics fm(QFont("Arial", 12));
    for (int i=0;i<texts.size();i++) //绘制文本框
    OutputText = "";
    OutputPos = QPointF(0, 0);
    if (m_isDraging) //绘制框选的虚线
    //绘制标题
    QString title = "HachiDraw - [" + File;
    if (!saved) title += "*"; title += "]";
    this->setWindowTitle(title);
}
```

三、具体分工

1. 王妙杰：负责除Ctrl+C,Ctrl+V,成环外的全部后端工作，生成和擦除化学键/文本框的前端工作，撰写实践报告
2. 董洋：提出整体设计构思，负责成环的前端与后端工作，Ctrl+C,Ctrl+V的后端工作
3. 蔡雨阳：负责余下的全部前端工作，剪辑展示视频

四、项目总结与反思

4.1 项目总结

很好的完成了对`ChemDraw`的仿制，锻炼了组员们的代码水平，提高了用工程思维进行思考的能力。

4.2 未来展望

可以美化画布，引入格点和更多的化学预制结构，使得操作更加简单。

4.3 可修正的问题

使用智能指针如`shared_ptr`及`unique_ptr`来托管程序中出现的指针，防止内存泄漏。

可以将前后端进一步分离，同时重写一部分复用性不高的函数，使得代码可读性进一步提高。