程序开发实践报告

分数计算器设计

学　院	悉尼智能科技学院
专　业	通信工程
班级序号	230218
学　号	**202319193**
姓　名	万荣涛
指导教师	李国瑞
验收日期	2024年1月5日

核心知识点清单，由学生确认

数组	链表	指针	文件读写	默认参数	函数模板	多文件	类	派生	虚函数	友元函数	重载	多继承
√	√	√	√	√	√	√	√	√	√	√	√	

以下为教师评分表，学生不可填写

程序质量（60%）	课程设计报告（20%）	答辩效果（20%）	总分	等级

评分标准与说明：

程序质量（60%）包含程序正确性与所用知识点数量（40%），代码可读性（10%）与界面友好性（10%）。

课程设计报告（20%）要求排版规范，模块设计有文字说明，图、表、代码清单要有序号和名字。

现场答辩（20%）要求根据学生制作的PPT、讲述清晰、回答问题等情况综合评分。

收齐所有纸质报告的同时，要求学委收集所有学生的代码工程、报告电子版和答辩PPT以备存档。

设计要求：

设计一基于Windows的应用程序，使其能完成简单的分数计算功能。

计算功能包括：加、减、乘、除、记录变量

程序界面基于命令行界面。

目录

分数计算器

1 基本功能描述

分数计算器使用分数和整数进行输入和输出，可以执行加、减、乘、除的四则运算并自动检测、处理异常输入。

分数计算器每次计算表达式得到值，可以为变量进行赋值操作，并在后续计算中引用已赋值的变量。

提示符后可以输入表达式，表达式可以由+ - * / ( )六种符号、数字、已赋值的变量组成。

可以使用“xxx = 123”的格式创造变量并赋值，输入单纯的表达式则默认赋值给变量 _ ，所以变量 _ 也发挥了输入历史记录的作用。变量的命名规则同C++等编程语言中标识符的命名规则，可以包括大小写英文字符与数字，以及下划线‘_’与美元‘$’，但是不可以以数字开头。

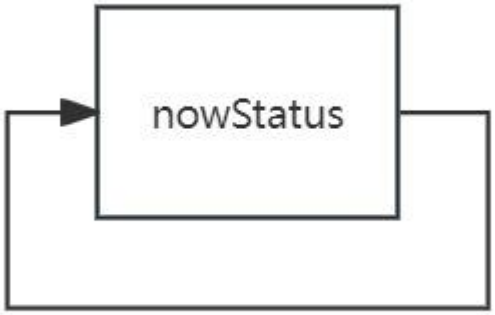
分数计算器的顶部菜单栏可以随时通过快捷键触发相应的功能。

表达式的计算结果用分数存储并表达。分数存储内部采用了高精度整形，没有整数溢出的问题。

如果输入有误，会给出相应的提示并允许修改原表达式。

2 设计思路

如图是整个计算器的流程图。有一函数指针nowStatus指向当前要运行的函数，每次借由这个函数指针调用目标函数，并由目标函数返回下一次将要运行的函数的指针，保存为新的nowStatus。



整个计算器的主要逻辑写在各个status函数中，通过每次返回不同的函数指针，可以改变计算器的状态并做到界面的切换。

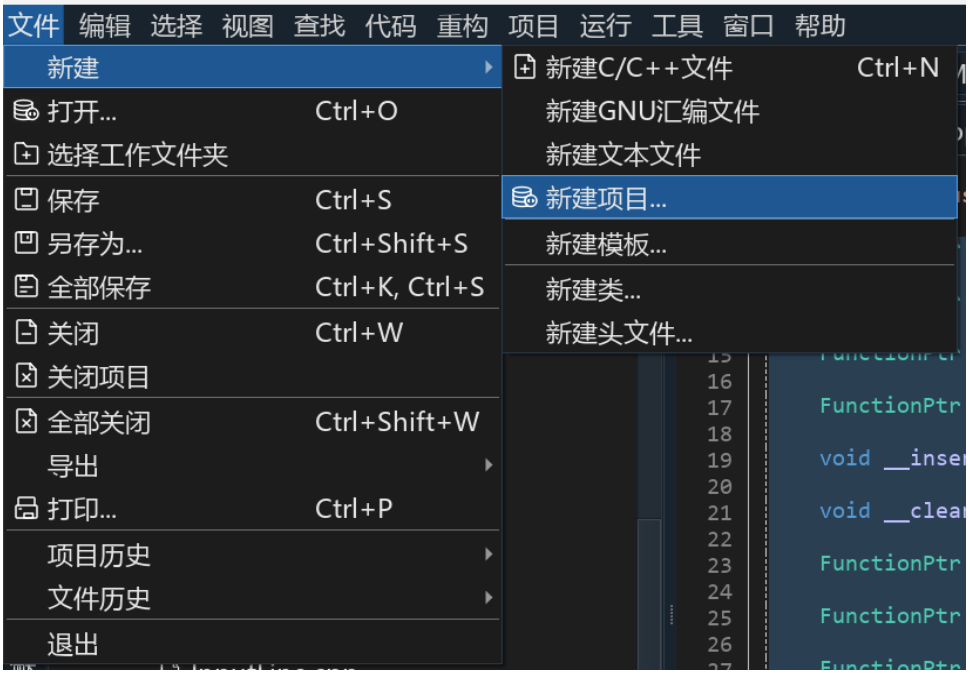
以下是几个使用到的status：

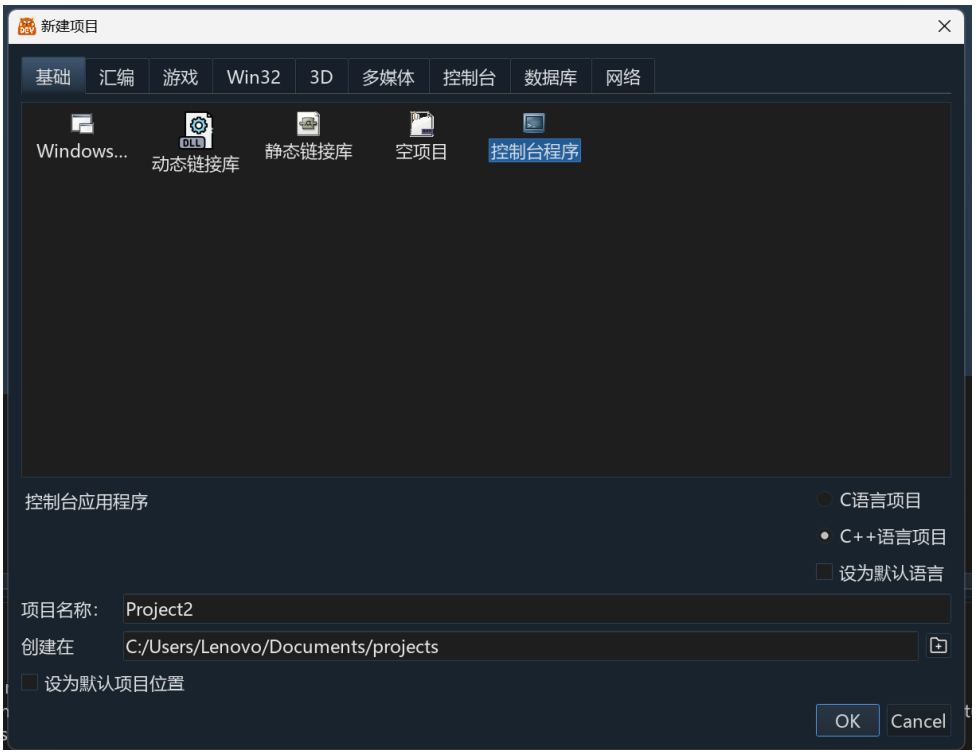
```
namespace status
FunctionPtr initStatus();
FunctionPtr getCodeStatus();
FunctionPtr inputStatus();
FunctionPtr helpStatus();
FunctionPtr __leftStatus();
FunctionPtr __rightStatus();
FunctionPtr __deleteStatus();
FunctionPtr __backspaceStatus();
FunctionPtr exitStatus();
FunctionPtr __exitStatus();
```

3 软件设计

3.1 设计步骤

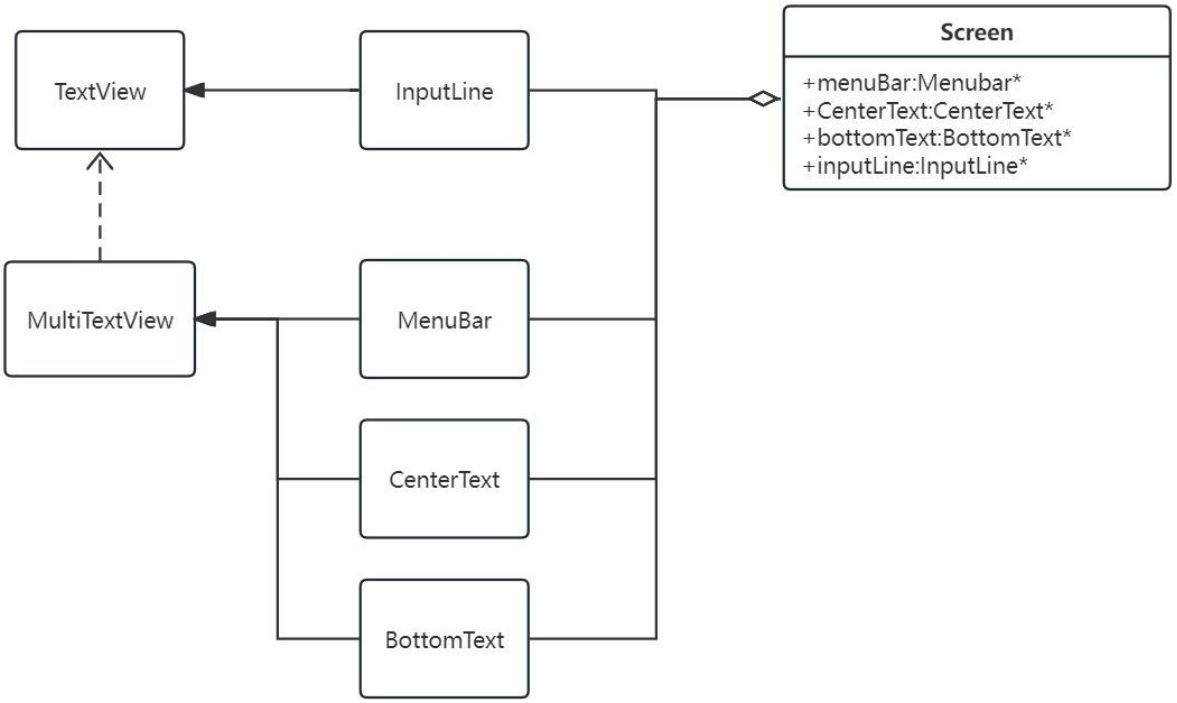
打开Red Panda C++，选择新建/新建项目/基础/控制台程序，点选OK，成功创建新C++控制台项目。





3.2 界面设计

1、创建控件



1.创建TextView类

TextView类的灵感来自于安卓的TextView视图。它实现了“在屏幕的指定位置以指定风格显示指定文字”的基础功能，是后续所有类的基础。

类设置在view命名控件中，可以避免污染命名空间。

主要的业务代码存在于print()与clear()两个虚方法中，可以完成显示和清除的功能。

print()方法接收一个偏移量，可以在自身坐标的基础上偏移一个坐标显示，这在后续的使用中简化了代码。

```
namespace view
{
    //this text view should be in one line, otherwise printing is is undefined behaviour.
    class TextView {
    public:
        TextView() = default;
        TextView(std::string _content,console::Position _position = {0,0},WORD _feature = FOREGROUND_INTENSITY);

        virtual void print(const console::Position& pos = {0,0});
        virtual void clear();

        virtual void setContent(std::string ws);
        virtual std::string getContent() const;
        virtual void setPosition(console::Position pos);
        virtual console::Position getPosition() const;
    protected:
        std::string content;
        console::Position position {};
        console::Position endPosition {};
        WORD feature = FOREGROUND_INTENSITY;
    };
}
```

2.创建MultiTextView类

TextView本身的实现很简单，只能支持最多一行文字的显示。为了改进这个不足，MultiTextView内部容纳了多个TextView，可以指定它们按照一定的布局显示（水平或竖直等），这就用到了前面print()

方法传入的偏移量参数。

```
namespace view
{
    class MultiTextView
    {
    public:
        std::vector<TextView> textViews;

        MultiTextView() = default;

        virtual void push_back(TextView textView);
        virtual void print(const console::Position& pos = {0,0});
        virtual void clear();

        virtual void setPosition(console::Position pos);
        virtual console::Position getPosition() const;

        console::Position position;
        console::Position endPosition;
    };
}
```

3.创建以上两个控件的子类

TextView的子类包括：

InputLine，可以实时显示输入内容并更新光标位置的输入行，从文件中读取输入提示符。

MultiTextView的子类包括：

MenuBar，从文件中读取顶部菜单栏的各个选项，并指定为将它们显示在页面顶部的同一行内。

CenterText，从文件中读取页面中的提示文字，并指定为将它们分行显示在页面前几行的中央位置。

BottomText，从std::map<std::string,math::RationalNumber>类型的变量存储记录中读取变量和值，将它们分行显示在页面的侧边。

2、创建Screen视图适配器

Screen类作为页面的视图适配器，容纳以上各个控件，并有Screen::print()方法将它们显示在页面上。

作为视图适配器，Screen存储当前显示区域的大小并据此动态调整各个子控件的位置，避免位置冲突。

```cpp
namespace status
{
    class Screen {
    public:
        Screen() = default;
        view::MenuBar* menuBar;
        view::CenterText* centerText;
        view::BottomText* bottomText;
        view::InputLine* inputLine;
        void print();
        console::Position size {100,20};
    private:
        console::Position endPosition {};
    };
}
```

3.3 关键功能的实现

1、Status的跳转

每个Status方法返回一个函数指针指向下一次要运行的Status方法，这当然是显然成立的逻辑。但是写的时候才发现这个做法有一个数学上的不可行性。

假设当前的Status方法的返回值为T，则有以下推理：

*T()的类型等于T，即：这个函数指针的类型为T，它解引用再调用之后的返回值类型也为T。

第一眼可能看不出什么问题，但是我们把这个式子展开：

T = *T(void) = *(T(void))(void) = *((*T(void))(void))(void) = …

我们如果想要得到T到底是什么类型，就不得不不断地展开等号右边的T，但是这个展开过程永远也没有尽头。

换言之：没有办法不用T来表达T，T没办法用普通的表达式表达。

难道一个这么好的点子，就要丢弃了吗？当初之所以会想这个方法实现Status之间的切换，是因为，如果为了方便，直接在前一个方法里调用后一个方法，则会将调用栈一步一步延伸下去。随着操作次数的增加，结果必将是爆栈。

在经过长时间的思考之后，我给出了我的解决方案：

```cpp
namespace status
{
    using FunctionPtr = void*;

    FunctionPtr invoke(const FunctionPtr _func);
}
```

我们既然无法表达T是什么类型，那就使用万能的void*指针。

我们为它取一个类型别名：FunctionPtr，日后需要使用的时候只需要用这个别名就好了，减少了易错性，增强了可读性。

同样地，我们把对指针解引用再调用函数获取返回值的操作也封装成一个函数，减少了易错性，增强了可读性和复用性。

2、用户操作的分析

让用户每次完成输入后都敲一下回车，这种做法有损于使用体验，还难以做到实时检测用户的快捷键操作。所以，我使用的API并非传统的std::cin或者::scanf()，而是Windows下专有的<conio.h>中

的_getwch()方法，可以对用户的每次键盘输入作出响应。

每一次输入都会需要进行判断，并且需要将判断的结果分发给不同的方法进行消息处理，我觉得在这里最合适的选择是复用一下之前的Status设计，使用std::map将输入内容与需要调用的Status之间的对应关系保存下来，这样可以少写大量的if-else或switch-case（实际上是交给了std::map内部的二分搜索），还可以让代码的可复用性大大提升。

```
namespace keys
{
    inline std::map<int,status::FunctionPtr> funcChar
    {
        {esc,(status::FunctionPtr)&status::exitStatus},
        {_left,(status::FunctionPtr)&status::__leftStatus},
        {_right,(status::FunctionPtr)&status::__rightStatus},
        {_delete,(status::FunctionPtr)&status::__deleteStatus},
        {_backspace,(status::FunctionPtr)&status::__backspaceStatus},
        {ctrlT,(status::FunctionPtr)&status::helpStatus}
    };
}
```

在以上代码中可以看到，我不仅对菜单栏的快捷键触发使用了这个设计，也对删除与光标移动的相关键位采用了相关处理，因为我阻止了用户进行任意操作和输入任意字符，所以我需要自己写一套文本编辑框的业务逻辑，从输入接收到相关的控制按键后调用相应的Status方法进行处理。

如此，我便成功实现了让用户按下Esc键就能立刻跳转到退出的确认界面的逻辑。

3、输入内容的分析

只要用户输入的字符没有触发任何快捷键，并且是合法的字符，我就按照光标位置插入输入序列，这当然是简单的判断逻辑。但是当用户按下回车的那一刻，我就不得不面对一个问题：用户输入的这一串表达式序列，我该怎么分析呢？

其实这是一个典型的“剥洋葱”的问题，要递归地把大问题拆解成小问题去解决。

洋葱的第一层：判断这是什么类型的输入。如果是赋值类型的输入，就原封不动转发给里层方法；如果是一个单纯的表达式，就改动一下，把它变成赋值给 _ 的赋值式。（使用一个下划线表示上次的计算结果，这是Python给我的灵感）

洋葱的第二层：既然接收的一定是一个赋值式，那就直接拆分就好了。拆出来要赋值的变量和值表达式，把值表达式交给里层方法去求值。

到了这里，就可以开始递归了：

如果是一个整体包裹着括号的表达式，那就先去除外层括号；

如果是一个由算术符号连接着多个项的“多项式”，那就先分割求出各项的值，再按照算术运算符的优先级顺序求值；

如果是一个纯粹的不可再分的“单项式”，就可以直接交给里层方法求值了。

到这里，我们终于剥出了洋葱的芯：对于“单项式”，如果不是数字，我们就直接去查找变量的储存记录。

```
{
    std::pair<std::string,math::RationalNumber> analyze(std::string_view text,const status::ValueHistory& history
    {...}

    std::pair<std::string,math::RationalNumber> __analyze(std::string_view text,const status::ValueHistory& histo
    {...}

    math::RationalNumber __getValue(std::string_view text,const status::ValueHistory& history)
    {
        util::log(nowInfo,fmt(text));

        if(text.size() == 0)
        {...}

        std::vector<std::size_t> bracketRecord = __countBracket(text);

        util::log(fmt(test::toString(bracketRecord)));

        //bracketed expression
        if(std::count(bracketRecord.begin(),bracketRecord.end(),0)==1 && text.size()!=1)
        {...}

        //single expression
        auto isSingleExpression = [&]()
        {...}
        if(isSingleExpression())
        {...}

        //multiple expression
        util::log("multiple expression");
        std::list<std::size_t> operatorSplit = __splitByOperator(text,bracketRecord);
        operatorSplit.push_back(text.size());
        std::list<math::RationalNumber> valueSplit;

        //getting subvalue
        {...}
        operatorSplit.pop_back();

        auto processWith = [&](auto _operator,char _operator_char)
        {...}
        processWith([](auto a,auto b){return a / b;},'/');
        processWith([](auto a,auto b){return a * b;},'*');
        processWith([](auto a,auto b){return a + b;},'+');
        processWith([](auto a,auto b){return a - b;},'-');

        return valueSplit.front();
    }
```

此外，我的函数接口都采用std::string_view，以避免不必要的复制，提高效率。

4、数据的存储与计算

我作为ACM预备队的队员，手写一个漂漂亮亮的高精度整数算法自然不在话下。

但是作为课设的重要部分，我还是有必要说明一下这里的类层次结构的：

类BigInteger存储一个正负无限制的长整数，重载了算术运算，并配备了math::gcd()方法（供约分使用）。

```cpp
namespace math
{
    class BigInteger
    {
    private:
        std::vector<long long> data;
        constexpr static long long BASE = 1e9;
        constexpr static int BASE_LENGTH = 9;
        void cutFrontZero();
        void unify();
        void sortOut();
        BigInteger& leftMove(int distance);
    public:
        BigInteger() = default;
        BigInteger(std::string number);
        BigInteger(long long number);
        bool positivity() const;
        friend bool operator <(const BigInteger& a,const BigInteger& b);
        friend bool operator >(const BigInteger& a,const BigInteger& b);
        BigInteger operator -() const;
        BigInteger& operator +=(const BigInteger& bi);
        friend BigInteger operator +(const BigInteger& a,const BigInteger& b);
        friend BigInteger operator -(const BigInteger& a,const BigInteger& b);
        std::string toString() const;
        friend std::ostream& operator <<(std::ostream& os,const BigInteger& bi);
        friend bool operator ==(const BigInteger& a,const BigInteger& b);
        BigInteger& operator *=(const int time);
        friend BigInteger operator *(const BigInteger& bi,const int time);
        friend BigInteger operator *(const int time,const BigInteger& bi);
        friend BigInteger operator *(const BigInteger& a,const BigInteger& b);
        friend BigInteger operator /(const BigInteger& a,const long long b);
        friend BigInteger operator /(const BigInteger& a,const BigInteger& b);
        friend BigInteger operator %(const BigInteger& a,const BigInteger& b);
    };

    BigInteger gcd(const BigInteger& a,const BigInteger& b);
    auto operator ""_bi(unsigned long long number);
    auto operator ""_bi(char const * const text,std::size_t size);
}
```

类RationalNumber具有以BigInteger存储的分子与分母，用以表达一个分数。
构造方法确保了RationalNumber对象无论何时均既约。

```cpp
namespace math
{
    class RationalNumber
    {
    public:
        bool positivity = true;
        BigInteger numerator = 1;
        BigInteger denominator = 1;
        RationalNumber() = default;
        RationalNumber(const BigInteger& value);
        RationalNumber(const BigInteger& _numerator,const BigInteger& _denominator);
        void reduce();
        friend bool operator <(const RationalNumber& a,const RationalNumber& b);
        friend bool operator >(const RationalNumber& a,const RationalNumber& b);
        RationalNumber operator -() const;
        RationalNumber reciprocal() const;
        RationalNumber& operator +=(const RationalNumber& bi);
        friend RationalNumber operator +(const RationalNumber& a,const RationalNumber& b);
        friend RationalNumber operator -(const RationalNumber& a,const RationalNumber& b);
        std::string toString() const;
        friend std::ostream& operator <<(std::ostream& os,const RationalNumber& bi);
        friend bool operator ==(const RationalNumber& a,const RationalNumber& b);
        RationalNumber& operator *=(const RationalNumber& num);
        friend RationalNumber operator *(const RationalNumber& a,const RationalNumber& b);
        friend RationalNumber operator /(const RationalNumber& a,const RationalNumber& b);
    };
}
```

5、异常处理

std::exception不算是特别好用(尤其远远不如Java等语言的异常处理，不方便循迹)，所以我也自己造了一套方便的异常系统。

```cpp
namespace exception
{
    struct Exception
    {
        std::string content {};
    };
    struct InvalidNameException : public Exception
    {};
    struct NullInputException : public Exception
    {};
    struct BracketMismatchException : public Exception
    {};
    struct UnknownVarException : public Exception
    {};
    struct DividedByZeroException : public Exception
    {};
}
```

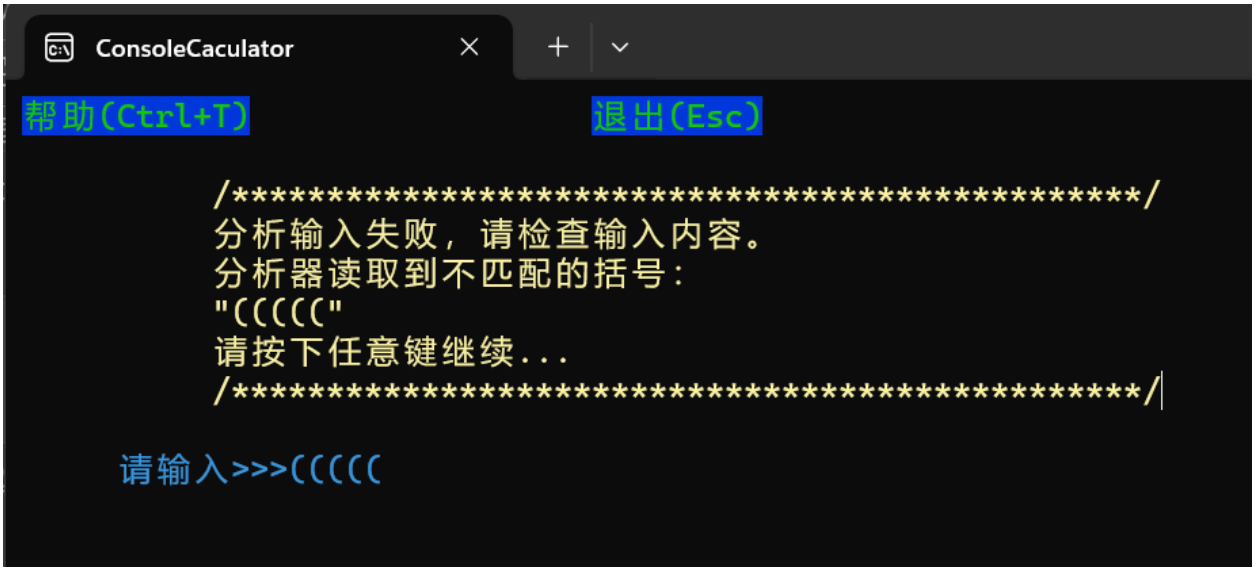
有一公共父类exception::Exception，其下的各个异常均服务于业务代码的需求。

看类名就可以知道，这些异常分别服务于非法标识符、空输入、括号不匹配、未知标识符、除以0这些实际的需求。但是检测到这些问题以后，可不是throw了以后就不管了的。

我的策略是：在Status中一起处理。

```
69          try
70          {
71              auto result = util::analyze(inputString,history);
72
73              history[result.first] = result.second;
74              nowScreen.bottomText->update();
75
76              __clear();
77          }
78          catch(exception::InvalidNameException e)
79          {
80              hint("begin","fail","invalid name",e.content,"continue","end");
81              util::log("InvalidNameException : " + e.content);
82          }
83          catch(exception::NullInputException e)
84          {
85              hint("begin","fail","null input","continue","end");
86              util::log("NullInputException : " + e.content);
87          }
88          catch(exception::BracketMismatchException e)
89          {
90              hint("begin","fail","bracket mismatch",e.content,"continue","end");
91              util::log("BracketMismatchException : " + e.content);
92          }
93          catch(exception::UnknownVarException e)
94          {
95              hint("begin","fail","unknown name",e.content,"continue","end");
96              util::log("UnknownVarException : " + e.content);
97          }
98          catch(exception::DividedByZeroException e)
99          {
100             hint("begin","fail","divided by zero","continue","end");
101             util::log("DividedByZeroException : ");
102         }
```

可以看到，这些异常其实都是解析用户输入的时候才会产生的，那就一起并行地catch就好了。对于每次catch，都会在程序的日志文件中留下记录并调用status::hint()函数模板反馈给用户：



```
2024-01-04 06:24:05 init
Status.cpp : void* status::initStatus() at line 9
Status.cpp : void* status::inputStatus() at line 37
get valid Ascii charactor ( 40
inputString = "("
get valid Ascii charactor ( 40
inputString = "(("
get valid Ascii charactor ( 40
inputString = "((("
get valid Ascii charactor ( 40
inputString = "((((("
get valid Ascii charactor ( 40
inputString = "(((((("
endl
Analyze.cpp : std::pair<std::__cxx11::basic_string<char>, math::RationalNumber> util::analyze(std::string_view, const status::ValueHistory&) at line 7 text
Analyze.cpp : std::pair<std::__cxx11::basic_string<char>, math::RationalNumber> util::__analyze(std::string_view, const status::ValueHistory&) at line 34 te
Analyze.cpp : math::RationalNumber util::__getValue(std::string_view, const status::ValueHistory&) at line 63 text = "((((("
BracketMismatchException : "(((((("
Status.cpp : void* status::inputStatus() at line 37
Status.cpp : void* status::exitStatus() at line 170 input = 27
Status.cpp : void* status::__exitStatus() at line 183
```

日志记录是个大话题，放到后面单说。在“异常处理”里我只提hint函数模板。

其实它的逻辑很简单：将错误提示反馈给用户，等待用户下一次敲击键盘再复原界面。

但是，在原则上应该做到“资源与代码分离”，也就是说，应该在代码之外另设一个文件保存这些错误信息，需要的时候应该使用文件里面的信息。怎样才能让这个功能用起来丝滑爽快呢？

一则为资源编号。这其实是大部分需要加载资源的程序的必修，不过要让我自己手写确实麻烦，所以我就很简单地分行写成文件，需要的时候可以通过资源名从一个std::map里面读取：

```
begin
/************************************************/
end
/************************************************/
fail
分析输入失败，请检查输入内容。
invalid name
分析器读取到非法标识符：
unknown name
分析器读取到未知标识符：
```

```
hint("begin","fail","invalid name",e.content,"continue","end");
```

如此就直接获取到了名为begin等等的资源了。

二则优化函数接口。为什么我的hint用起来方便？因为它拥有可变参数，接收了若干个字符串并尝试解析：如果是资源名则解析对应的资源，否则直接把字符串内容送到用户面前。

```
template <typename ...T>
int hint(const T& ...args)
{
    return __hint(
        {
            hintText.find(args)==hintText.end() ? args : hintText[args]
            ...
        }
    );
}
```

通过使用可变形参包、形参包展开等语法，我们调用了里层方法接口，使得其他代码与用户互动的成本大大降低。

6、日志记录

代码未动，调试先行。调试我这种准图形界面的代码非常依赖于优秀、详细的日志记录，我也花费了很多时间完善我的相关代码。

首先登场的是相关信息的格式化：

在Test.h中定义了一些十分实用的宏，我可以用非常简短的代码获取当前位于代码的什么位置、当前的系统时间，以及使用fmt(n)把n格式化为“n = 15”输出。

```
#define varName(x) std::string(#x)
#define fmt(x) (varName(x)+std::string(" = ")+test::toString(x))
#define nowFunc __PRETTY_FUNCTION__
#define nowTime test::currentTime()
#define nowInfo (std::string(__FILE__)+" : "+nowFunc+" at line "+std::to_string(__LINE__))
```

在Logger.h中定义了向文件写日志的函数模板log，同样是使用了形参包展开的手法，可以轻松写入任意的内容到日志中。

```
namespace util
{
    static std::ofstream logger {"log.txt",std::ios::app | std::ios::out};

    template <typename...T>
    void log(const T&... args)
    {
        ((logger << args << ' '),...) << std::endl;
    }
};
```

看着这些代码也许不会有什么感触，但是如果看到实际效果就会知道这有多重要了：

```
2024-01-04 06:47:58 init
Status.cpp : void* status::initStatus() at line 9
Status.cpp : void* status::inputStatus() at line 37
get valid Ascii charactor a 97
inputString = "a"
get valid Ascii charactor   32
inputString = "a "
get valid Ascii charactor = 61
inputString = "a ="
get valid Ascii charactor   32
inputString = "a = "
get valid Ascii charactor 1 49
inputString = "a = 1"
get valid Ascii charactor 2 50
inputString = "a = 12"
get valid Ascii charactor 3 51
inputString = "a = 123"
endl
Analyze.cpp : std::pair<std::__cxx11::basic_string<char>, math::RationalNumber> util::analyze(std::string_view, const status::ValueHistory&) at line 7 text = "a = 123"
Analyze.cpp : std::pair<std::__cxx11::basic_string<char>, math::RationalNumber> util::__analyze(std::string_view, const status::ValueHistory&) at line 34 text = "a=123"
Analyze.cpp : math::RationalNumber util::__getValue(std::string_view, const status::ValueHistory&) at line 63 text = "123"
test::toString(bracketRecord) = "{0,0,0,}"
single expression
Analyze.cpp : math::RationalNumber util::__getSingleValue(std::string_view, const status::ValueHistory&) at line 157 text = "123"
Status.cpp : void* status::inputStatus() at line 37
Status.cpp : void* status::exitStatus() at line 170 input = 27
Status.cpp : void* status::__exitStatus() at line 183
```

要知道，这么丰富的日志只是写了一些很简单的代码就生成的：

```
util::log("get valid Ascii charactor",char(input),std::to_string(input));
```

```
util::log(fmt(inputString));
```

```
util::log(nowInfo,fmt(text));
```

```
    util::log("\n");
    util::log(nowTime,"init");
    util::log(nowInfo);
```

4 结论与心得体会

作为有一定基础的学生，我觉得我对自己的要求应该更高一点。

所以我自己从头造了一套控制台界面的控件，并自己做了很多附件，做了很多架构的设计。

越是自己写，就越能感受到，那些看起来理所当然的东西，其实都是前人智慧的结晶。自己写一个控件体系就能发现系统的UI其实非常难造，绝非草草可为；自己遇到了头文件循环include的问题，才发现包管理器是多么的重要和伟大。

虽说如此，在写的过程中，我并未畏难，而是更加地喜爱编程了。因为编程是人类造给人类的语言，学习曲线相较于数学等等自然学科要平缓得多，但其中蕴含的人类智慧相较而言却一点不少。自己查资料，学习新的语法，学习API，虽然也有看着模板模板形参和模板推导指引一窍不通的时候，但现在回想起来，总有几分甘美的滋味。

编程不难，不复杂。如果把代码写成了搬砖，那就没有意义了。我对此有自信，我只要一直热爱着，就能做好并且快乐，让编程成为我的职业，我的追求。

虽然说我自认为在编程上算比较上进的，但是偶然看见一些同僚的背影还是会自愧不如。他们比我水平高，也有很多别的优秀的地方，对我这样的愿意学的人总是不吝赐教。我对他们致以敬意。

5 参考文献

[1]Marc Gregoire. C++20高级编程. 清华大学出版社. 2022.3

[2]https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

[3]https://learn.microsoft.com/zh-cn/cpp/cpp/welcome-back-to-cpp-modern-cpp?view=msvc-170

6 附录

6.1 调试报告

Windows下，由于功能控制键没有GBK对应码位，功能控制键是以2次输入的形式获取的。由此，经过多次调试与日志输出对比，我得出结论：

endl = 13;
left = 224 + 075;
right = 224 + 077;
delete = 224 + 083;
backspace = 8;
esc = 27;
Ctrl + T = 20;
以下为调试日志：

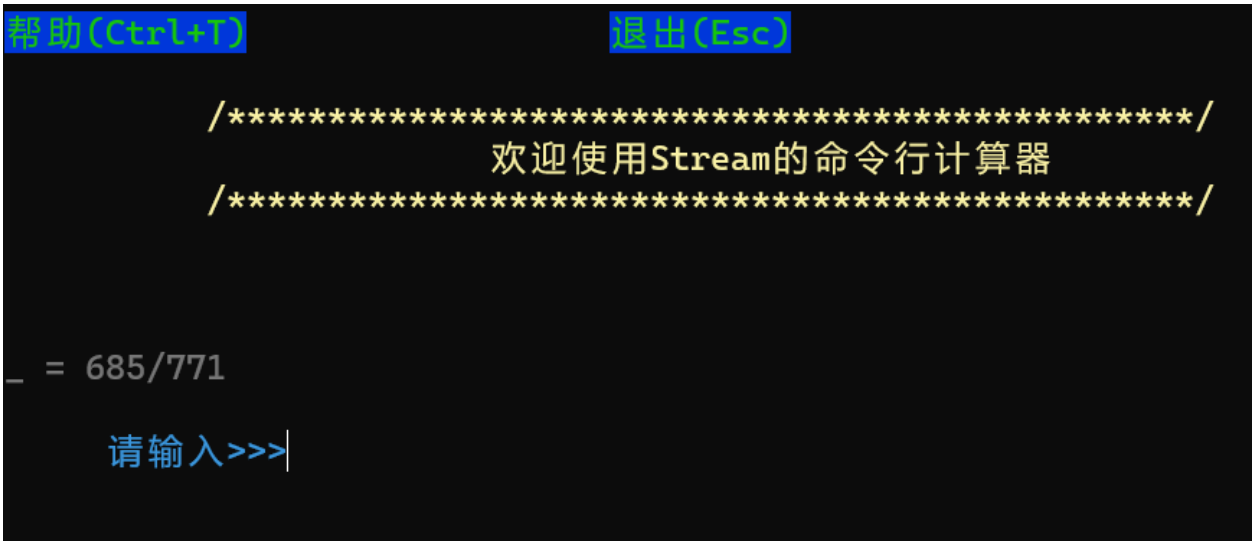
```
2024-01-04 07:25:10 init
Status.cpp : void* status::initStatus() at line 9
Status.cpp : void* status::getCodeStatus() at line 20
2024-01-04 07:25:13 input = -224072
2024-01-04 07:25:13 input = -224080
2024-01-04 07:25:17 input = -224075
2024-01-04 07:25:17 input = -224077
2024-01-04 07:25:19 input = 13
2024-01-04 07:25:19 input = 13
2024-01-04 07:25:21 input = -224083
2024-01-04 07:25:21 input = -224083
2024-01-04 07:25:21 input = 8
2024-01-04 07:25:22 input = 8
2024-01-04 07:25:22 input = 27
2024-01-04 07:25:22 input = 27
```

以下为调试代码：

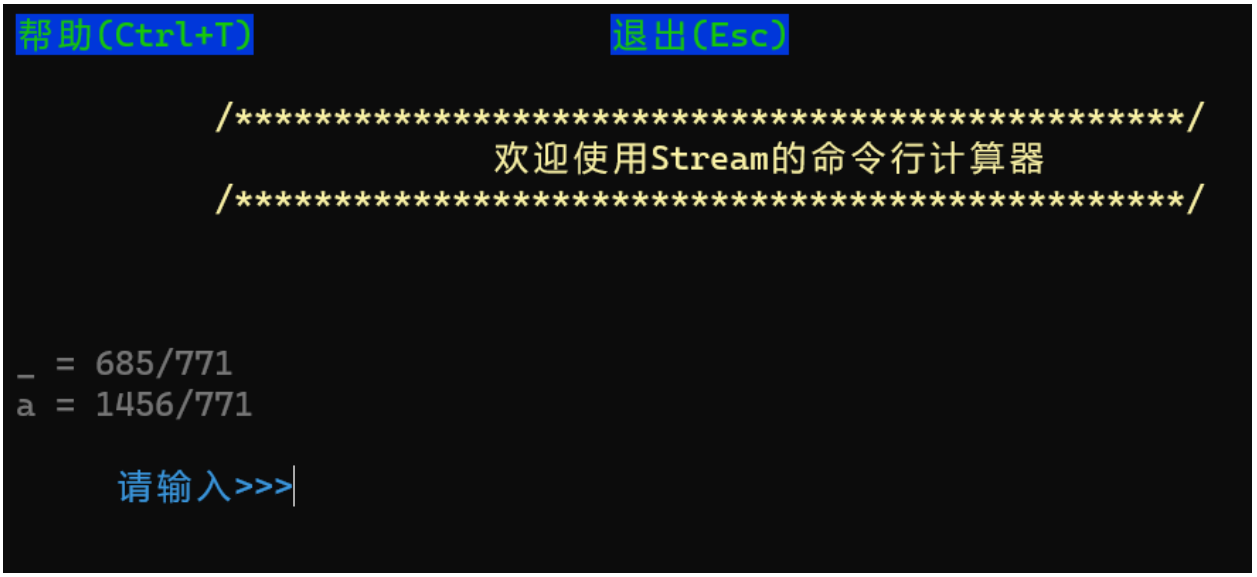
```
FunctionPtr initStatus()
{
    util::log("\n");
    util::log(nowTime,"init");
    util::log(nowInfo);

    nowScreen.menuBar = &menuBar;
    nowScreen.centerText = &defaultHelper;
    nowScreen.bottomText = &currentBottomText;
    nowScreen.inputLine = &defaultInputLine;
    return (FunctionPtr)&getCodeStatus;
}

FunctionPtr getCodeStatus()
{
    util::log(nowInfo);

    console::clear();
    nowScreen.inputLine = &defaultInputLine;
    nowScreen.print();

    while(true)
    {
        int input = console::next();
//          hint("begin","invalid character",std::to_string(input),"continue","end");
        util::log(nowTime,fmt(input));
    }
    return (FunctionPtr)&getCodeStatus;
}
```

6.2 测试结果

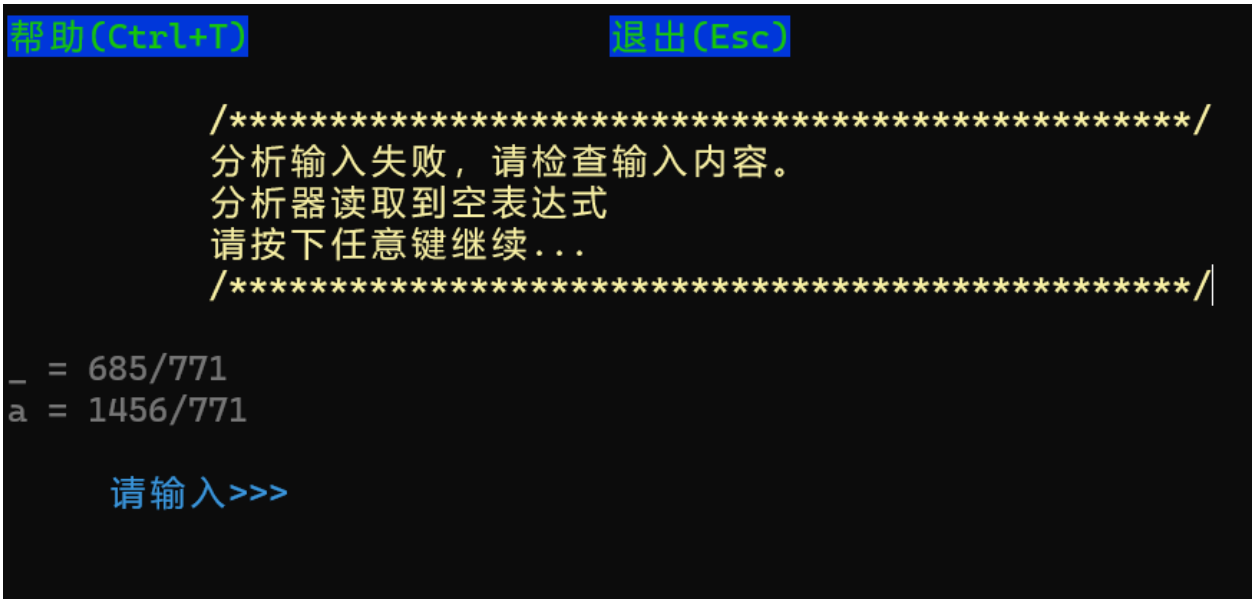
输入“1*2/3+(114/514)”



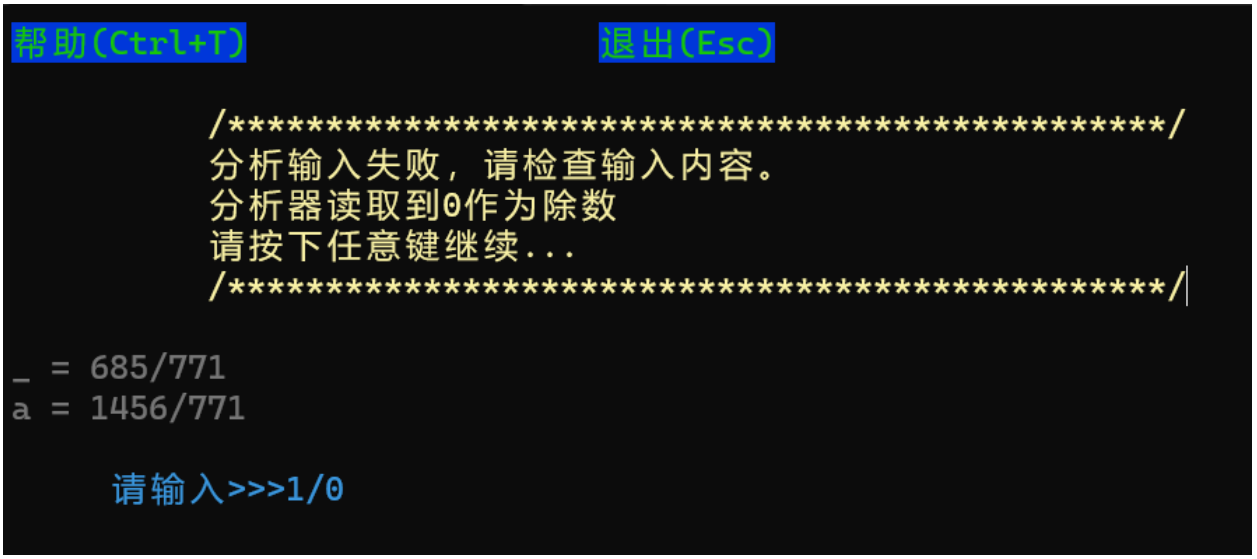
输入“a = _ + 1”



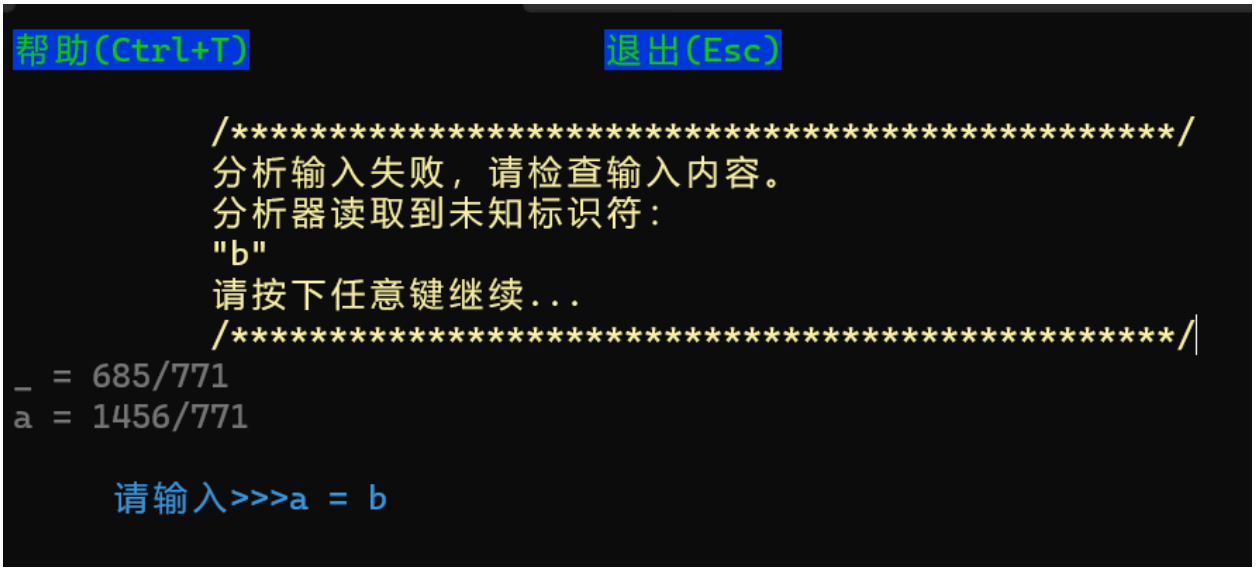
输入“”



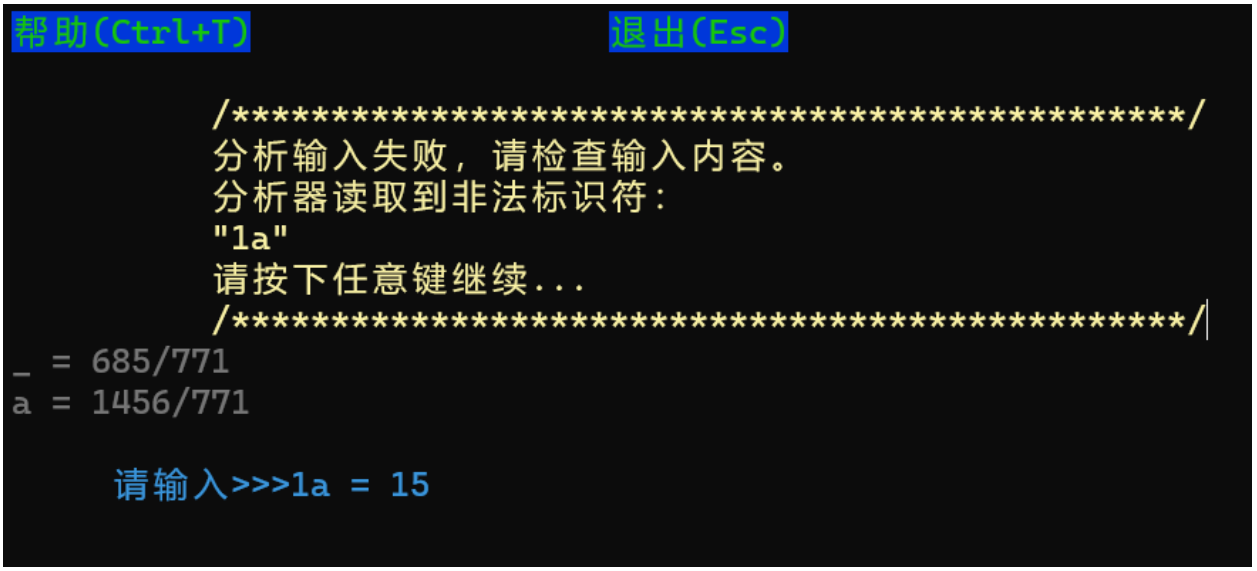
输入“1/0”



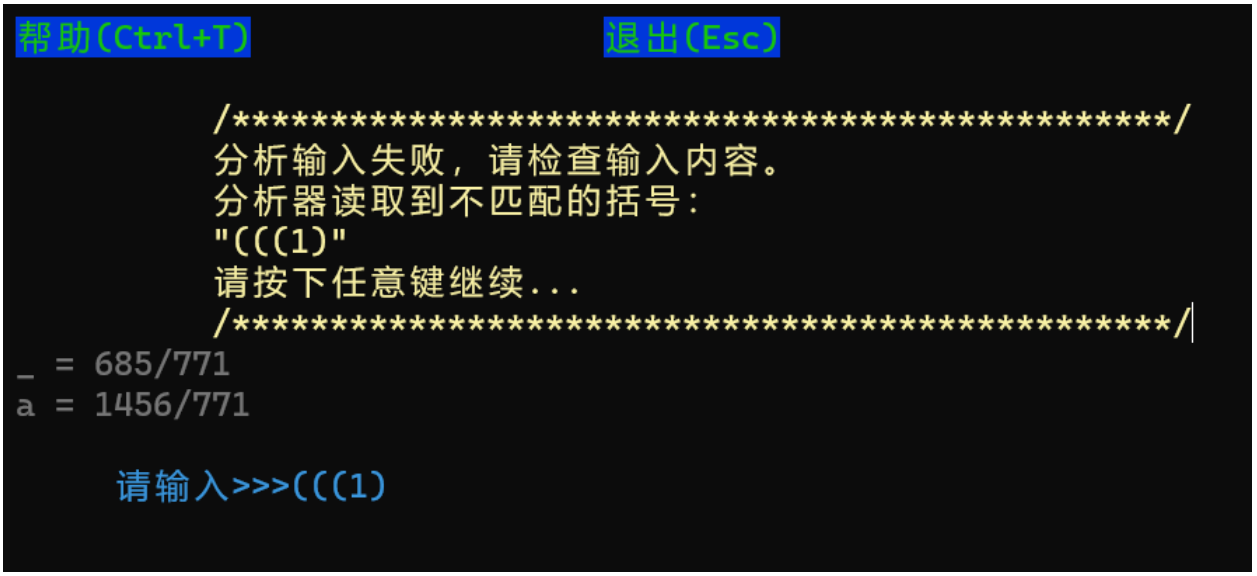
输入“a = b”



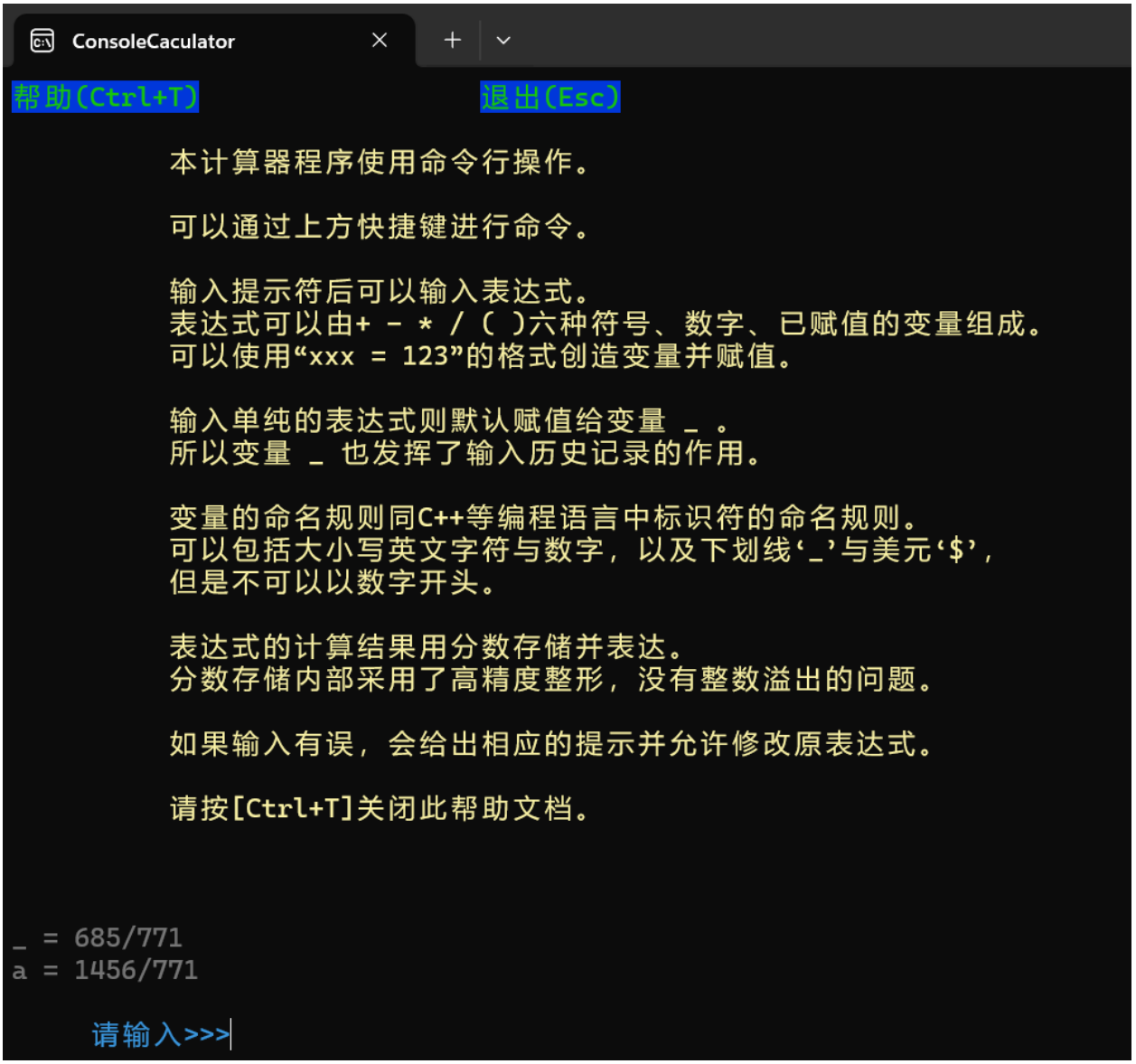
输入“1a = 15”


帮助(Ctrl+T)
退出(Esc)
/**/
分析输入失败，请检查输入内容。
分析器读取到非法标识符：
"1a"
请按下任意键继续...
/**/
_ = 685/771
a = 1456/771
请输入>>>1a = 15

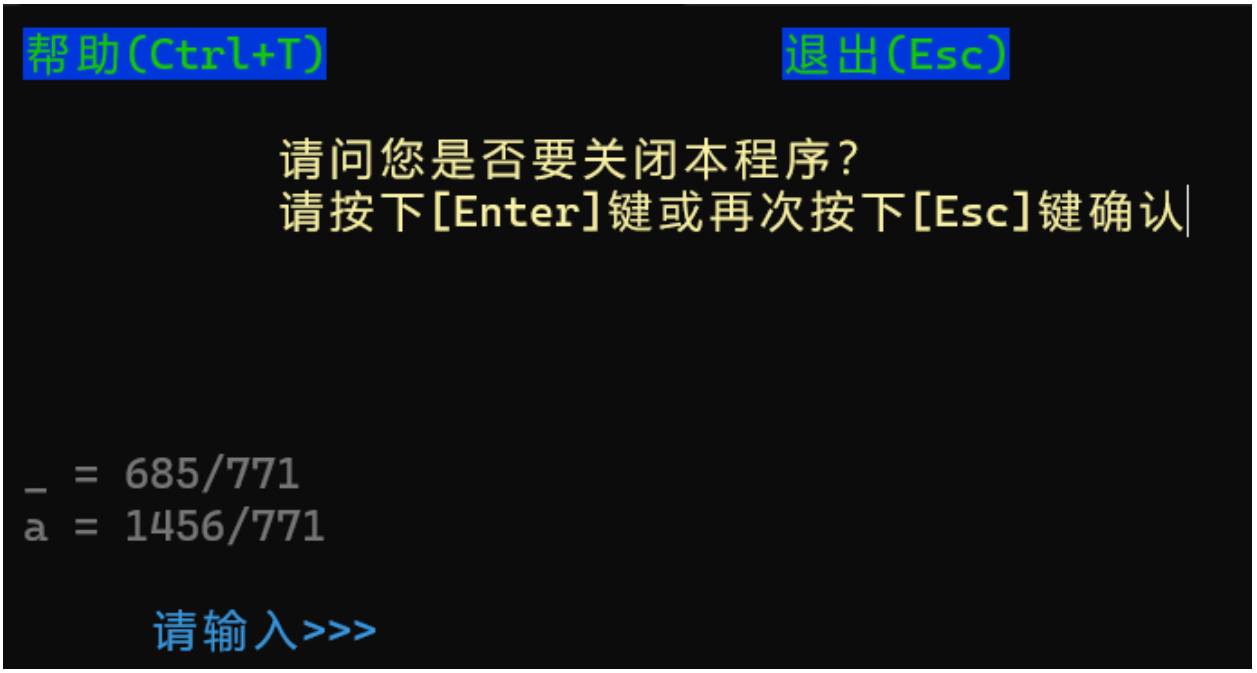
输入“(((1)”


帮助(Ctrl+T)
退出(Esc)
/**/
分析输入失败，请检查输入内容。
分析器读取到不匹配的括号：
"(((1)"
请按下任意键继续...
/**/
_ = 685/771
a = 1456/771
请输入>>>(((1)

按下Ctrl + T


ConsoleCaculator
帮助(Ctrl+T)
退出(Esc)
本计算器程序使用命令行操作。
可以通过上方快捷键进行命令。
输入提示符后可以输入表达式。
表达式可以由+ - * / ( )六种符号、数字、已赋值的变量组成。
可以使用“xxx = 123”的格式创造变量并赋值。
输入单纯的表达式则默认赋值给变量 _ 。
所以变量 _ 也发挥了输入历史记录的作用。
变量的命名规则同C++等编程语言中标识符的命名规则。
可以包括大小写英文字符与数字，以及下划线‘_’与美元‘$’，
但是不可以以数字开头。
表达式的计算结果用分数存储并表达。
分数存储内部采用了高精度整形，没有整数溢出的问题。
如果输入有误，会给出相应的提示并允许修改原表达式。
请按[Ctrl+T]关闭此帮助文档。
_ = 685/771
a = 1456/771
请输入>>>

按下Esc


帮助(Ctrl+T)
退出(Esc)
请问您是否要关闭本程序?
请按下[Enter]键或再次按下[Esc]键确认
_ = 685/771
a = 1456/771
请输入>>>

6.3 关键源代码

main.cpp

```cpp
status::FunctionPtr nowStatus = (status::FunctionPtr)&status::initStatus;

int main(int argc, char** argv)
{
    console::setTitle("ConsoleCaculator");
    while(true)
    {
        nowStatus = status::invoke(nowStatus);
    }
    return 0;
}
```

Invoke.h

```cpp
namespace status
{
    using FunctionPtr = void*;

    FunctionPtr invoke(const FunctionPtr _func);
}
```

CharSet.h

```cpp
namespace keys
{
    template <typename T = char>
    class CharSet : public std::set<T>
    {
    public:
        using std::set<T>::set;
        bool has(const T& c) const
        {
            return this->find(c) != this->end();
        }
    };
    template <typename... T>
    CharSet(T...) -> CharSet<std::common_type_t<T...>>;
}
```

Status.h

```cpp
    template <typename ...T>
    int hint(const T& ...args)
    {
        return __hint(
            {
                hintText.find(args)==hintText.end() ? args : hintText[args]
                ...
            }
        );
    }
```

Logger.h

```cpp
namespace util
{
    static std::ofstream logger {"log.txt",std::ios::app | std::ios::out};

    template <typename...T>
    void log(const T&... args)
    {
        ((logger << args << ' '),...) << std::endl;
    }
};
```

Test.h

```cpp
#define varName(x) std::string(#x)
#define fmt(x) (varName(x)+std::string(" = ")+test::toString(x))
#define nowFunc __PRETTY_FUNCTION__
#define nowTime test::currentTime()
#define nowInfo (std::string(__FILE__)+" : "+nowFunc+" at line "+std::to_string(__LINE__))

namespace test
{
    void show_wait(std::string_view content,int y);
    void show(std::string_view content,int y);

    std::string currentTime();

    template <typename T>
    std::string toString(const T& v)
    {...}
    std::string toString(const std::string& v);
    std::string toString(std::string_view v);
    std::string toString(const math::RationalNumber& v);
    template <typename T>
    std::string toString(const std::vector<T>& v)
    {...}
    template <typename T>
    std::string toString(const std::list<T>& v)
    {...}
}
```