



|  |  |  |  |
| --- | --- | --- | --- |
| **学号** | 17130130312 | **姓名** | 曹晨瑶 |
| **班级** | **1713013** | **任课教师** | **王献青** |
| **实验名称** | 继承与重置 | | |
| **实验学期** | **2018 – 2019 学年第2学期** | | |
| **实验日期** | 2019年6月15日 | **实验地点** | **G334** |
| **报告成绩** |  |  |  |

西安电子科技大学软件学院

# 实验目的

类使得自定义类型表现和基本类型相似，继承则表示若干类型/概念之间存在相似性的事实，重置则表示若干类型的对象具有接口相同、但语义不一定相同的操作。本次实验通过设计并实现几个程序，了解并熟悉类型继承、虚函数、方法重置机制解决问题的基本方法，提高学生对面向对象方法的理解和应用能力。

# 实验环境

操作系统：Windows10

开发工具：Code::Blocks

# 实验内容

## 继承与多态

Given a class:

#include <string>

class **Animal** {

**protected:**

std::string name;

**public**:

Animal(const char\* nm ="A") : name("A") { if(nm != NULL) name=nm; }

Animal(const std::string& nm) : name(nm) { }

**virtual** **void** **iam** () { cout << "Animal" << name << \n"; }

**void hello** () { cout << "Animal::hello from " << name; }

void common() { cout << "Animal::common"; }

}

Derive（派生） two classes from ***Animal***, and for each define ***iam***() to write out the name of the class and the object, and for each define ***hello***() to write out any different text with the name of the object. Create objects of these classes and call ***iam***(),***hello***() and common() for them. Assign pointers to objects of the derived classes to ***Animal \**** pointers and call ***iam***() , ***hello***() and common() through those pointers.

## 定义表达式类型

修改第四次上机实验题“定义算术表达式类型 Expr”，使得其实现满足以下要求：

（1）保持 Expr 给用户提供的接口不变；

（2）在 Expr 内部，算术表达式的存储结构必须采用“表达式语法树”；

（3）定义一个类层次结构，这些类既体现你对表达式语法树的结点（按其性质）的种类划分，也提供每种结点的特征。从而使得 Expr 类能以一致的方式访问表达式语法树的任一结点。

# 数据结构与算法设计

## 继承与多态

**模块结构及文件组织设计：**

模块1：主控模块，仅包括文件main.cpp ，定义了 main()函数。

模块2：仅包括文件 animal.hpp,定义了类animal及其派生类Cat和Dog

**关键数据结构设计：**

class Animal

{

protected:

std::string name;

public:

Animal(const char\* nm ="A") : name("A") { if(nm != NULL) name=nm; }

Animal(const std::string& nm) : name(nm) { }

virtual void iam () { cout << "Animal " << name <<" \n"; }

void hello () { cout << "Animal::hello from " << name << std::endl; }

void common() { cout << "Animal::common\n"; }

};

class Cat :public Animal{

public:

Cat(const char \* nm = "A") :Animal("A") { if (nm != NULL) name = nm; }

Cat(const std::string& nm) :Animal(nm) { }

void iam() { std::cout << "Cat " << name << " \n"; }

void hello() { std::cout << "Cat::hello from " << name << std::endl;}

};

class Dog :public Animal {

public:

Dog(const char \* nm = "A") :Animal("A") { if (nm != NULL) name = nm; }

Dog(const std::string& nm) :Animal(nm) { }

void iam() { std::cout << "Dog " << name << " \n"; }

void hello() { std::cout << "Dog::hello from " << name << std::endl; }

};

**算法1.1 int main()**

作 用：主控函数，对其类的相关测试

参 数：无参数。

返回值：总是返回0。

计算过程：

1. 定义一个Animal的对象A
2. 定义一个Cat的对象C
3. 定义一个Dog的对象D
4. 三个对象分别调用iam()、hello()、commen()
5. 定义类Animal及其派生类的指针，分别指向上面的三个对象
6. 三个指针分别调用iam()、hello()、common()
7. 定义类Animal及其派生类的引用，分别引用上面的三个对象
8. 三个引用分别调用iam()、hello()、common()
9. 返回0

**算法1.2 Animal(Cat / Dog)(const char \* nm = "A") :name("A") { if (nm != NULL) name = nm; }**

作 用：构造函数。

参 数：const char \* nm= ”A”

返回值：无返回值。

计算过程：

1. 若指针nm不为空，则将其对name赋值

**算法1.3 Animal(Cat / Dog)(const std::string& nm):name(nm){ }**

作 用：构造函数。

参 数：const std::string& nm //存储名称字符串的引用

返回值：无返回值。

计算过程：对name赋值

**算法1.4 virtual void iam()** (在派生类中进行重置)

作 用：输出类名及对象名称

参 数：无参数。

返回值：无返回值。

计算过程：输出类名“Animal（Cat / Dog）”和对象名称

**算法1.5 void hello()**（在派生类里重载）

作 用：输出解释及对象名称

参 数：无参数。

返回值：无返回值。

计算过程：输出字符串“Animal(Cat / Dog)::hello from ”和对象名称

**算法1.6 void common()**

作 用：输出字符串

参 数：无参数。

返回值：无返回值。

计算过程：输出字符串“Animal::common”

## 定义表达式类型

**模块结构及文件组织设计：**

模块1：主控模块，仅包括文件main.cpp，定义了main()函数和相关测试函数

模块2：功能模块，包括以下文件：

Expr.hpp 定义了表达式类型 Expr。

Expr.cpp 实现了表达式类型 Expr 的成员函数。

ExprNode.hpp 定义了语法树类型 SyntaxTree 和结点的基本类型 basic\_node

ExprNode.cpp 定义了表达式语法树及其所有结点类型的实现。

ExprNode\_derived.hpp 定义了树结点类型 basic\_node 的全部派生类。

Parser.hpp 定义了表达式的解析器类型 class CexprParser。

Parser.cpp 实现了解析器类型 CExprParser 和词法分析器 CExprLexer 的成员函数。

token.hpp 定义了表达式中的记号Token的相关类型。

error.hpp 定义了相关的“异常”类型。

**关键数据结构设计：**

1. class basic\_node表达式语法树的基本节点类型。

class basic\_node//抽象类

{

protected://派生类可访问

t\_token mToken;//结点记号

basic\_node(const t\_token& token)//构造函数

:mToken(token){}

public:

virtual ~basic\_node(){}//析构函数

virtual double eval() const=0;//纯虚函数

virtual void generatePostfix(string& retValue) const;// 产生表达式的后缀式

virtual void generatePrefix(string& retValue) const;// 产生表达式的前缀式

virtual void generateInfix(string& retValue) const;// 产生表达式的中缀式，并添加括号

virtual void printTree(ostream& outDevice,int deepth) const ;//打印以当前结点为根的子树

const string& getText() const;//结点中存放的记号的文本

Token\_value getKind() const;//结点中存放的记号的种类

long getPosition() const;//结点中存放的记号在表达式字符串中的位置

virtual const char\* typeName() const=0;//纯虚函数。获取表示当前结点的类名

};

2. class SyntaxTree表达式的语法树类型

class SyntaxTree

{

private:

basic\_node\* mRootNode;//语法树根结点

public:

SyntaxTree(basic\_node\* theRoot)//构造函数

:mRootNode(theRoot) {}

virtual ~SyntaxTree();//析构函数

void setRoot(basic\_node\* theRoot);//设置根结点

basic\_node\* getRoot() const;//获取根结点

double eval() const;// 计算（子）表达式的值

void generatePostfix(string& retValue) const;// 产生表达式的后缀式

void generatePrefix(string & retValue) const; // 产生表达式的前缀式

void generateInfix(string & retValue) const; // 产生表达式的中缀式，并添加括弧

void printTree(ostream& outDevice, int deepth) const;//先序遍历打印语法树

};

3. class COpNode: public basic\_node语法树非叶子结点的类型

class COpNode : public basic\_node//语法树非叶子结点的类型

{

public:

virtual ~COpNode(){}//析构函数

protected:

COpNode(const t\_token& token)//构造函数

:basic\_node(token) {}//初始化基类

};

4. class CBinaryOpNode : public COpNode所有二元运算对应的语法树结点之基类

class CBinaryOpNode : public COpNode//所有二元运算对应的语法树结点之基类

{

protected:

basic\_node \*mLeftChild;//左孩子，第一个操作数的子树

basic\_node \*mRightChild;//右孩子，第二个操作数的子树

CBinaryOpNode(const t\_token& token,basic\_node \*pLeft,basic\_node \*pRight)

:COpNode(token),mLeftChild(pLeft),mRightChild(pRight)

{}//构造函数

public:

virtual ~CBinaryOpNode();//析构函数

virtual void generatePostfix(string& retValue) const;// 产生子表达式的后缀式

virtual void generatePrefix(string& retValue) const;// 产生子表达式的前缀式

virtual void generateInfix(string& retValue) const;// 产生子表达式的中缀式，并添加括号

virtual void printTree(ostream& outDevice,int deepth) const ;

};

5. class CPlusNode二元 加法 运算的语法树结点类型

class CPlusNode : public CBinaryOpNode

{

public:

CPlusNode(const t\_token& token,basic\_node \*pLeft,basic\_node \*pRight)

:CBinaryOpNode(token,pLeft,pRight)

{}//构造函数

virtual double eval() const//计算子表达式的值

{

return mLeftChild->eval()+mRightChild->eval();

}

virtual const char\* typeName() const//输出结点所属类名

{

return "CPlusNode";

}

};

6. class CMinusNode二元 减法 运算的语法树结点类型

class CMinusNode : public CBinaryOpNode

{

public:

CMinusNode(const t\_token& token,basic\_node \*pLeft,basic\_node \*pRight)

:CBinaryOpNode(token,pLeft,pRight)

{}//构造函数

virtual double eval() const//计算子表达式的值

{

return mLeftChild->eval()-mRightChild->eval();

}

virtual const char\* typeName() const//输出结点所属类名

{

return "CMinusNode";

}

};

7. class CMulNode二元 乘法 运算的语法树结点类型

class CMulNode : public CBinaryOpNode

{

public:

CMulNode(const t\_token& token,basic\_node \*pLeft,basic\_node \*pRight)

:CBinaryOpNode(token,pLeft,pRight)

{}//构造函数

virtual double eval() const//计算子表达式的值

{

return mLeftChild->eval()\*mRightChild->eval();

}

virtual const char\* typeName() const//输出结点所属类名

{

return "CMulNode";

}

};

8. class CDivNode二元 除法 运算的语法树结点类型

class CDivNode : public CBinaryOpNode

{

public:

CDivNode(const t\_token& token,basic\_node \*pLeft,basic\_node \*pRight)

:CBinaryOpNode(token,pLeft,pRight)

{} //构造函数

virtual double eval() const//计算子表达式的值

{

double r=mRightChild->eval();

if(r>=-0.000001&&r<=0.000001)

throw CDiv0(mRightChild->getPosition());

return mLeftChild->eval()/r;

}

virtual const char\* typeName() const//输出结点所属类名

{

return "CDivNode";

}

};

9. class CUnaryOpNode一元 运算所对应的语法树结点类型

class CUnaryOpNode : public COpNode//所有二元运算对应的语法树结点之基类

{

protected:

basic\_node \*mChild;//一元运算操作数的子树

CUnaryOpNode(const t\_token& token,basic\_node \*pChild)

:COpNode(token),mChild(pChild)

{}//构造函数

public:

virtual ~CUnaryOpNode();//析构函数

virtual void generatePostfix(string& retValue) const;// 产生子表达式的后缀式

virtual void generatePrefix(string& retValue) const;// 产生子表达式的前缀式

virtual void generateInfix(string& retValue) const;// 产生子表达式的中缀式，并添加括号

virtual void printTree(ostream& outDevice,int deepth) const ;

};

10. class CUnaryPlusNode一元 加法 运算的结点类型

class CUnaryPlusNode : public CUnaryOpNode

{

public:

CUnaryPlusNode(const t\_token& token,basic\_node \*pChild)

:CUnaryOpNode(token,pChild)

{}//构造函数

virtual double eval() const//计算子表达式的值

{

return mChild->eval();

}

virtual const char\* typeName() const//输出结点所属类名

{

return "CUnaryPlusNode";

}

};

11. class CUnaryMinusNode一元 减法 运算的结点类型

class CUnaryMinusNode : public CUnaryOpNode

{

public:

CUnaryMinusNode(const t\_token& token,basic\_node \*pChild)

:CUnaryOpNode(token,pChild)

{}//构造函数

virtual double eval() const//计算子表达式的值

{

return (-1)\*mChild->eval();

}

virtual const char\* typeName() const//输出结点所属类名

{

return "CUnaryMinusNode";

}

};

12. class CLeafNode叶子结点 对应的语法树结点类型

class CLeafNode : public basic\_node//叶子结点的基类

{

protected:

CLeafNode(const t\_token& token) :basic\_node(token){}//构造函数

};

13. class CNumberNode数值型操作数对应的语法树叶子结点类型

class CNumberNode : public CLeafNode

{

public:

CNumberNode(const t\_token& token)

:CLeafNode(token)

{}//构造函数

virtual double eval() const//记号的数值

{

return atof(mToken.tok\_txt.c\_str());

}

virtual const char\* typeName() const//输出结点所属类名

{

return "CNumberNode";

}

};

14. class CExprParser 语法分析器

class CExprParser//语法分析器

{

private:

t\_token curr\_token; // 当前的记号对象

class CExprLexer;//声明该类

CExprLexer\* mLexer;//词法分析器对象

Token\_value get\_token();//获取下一个记号

basic\_node\* expr(bool get,bool afterLP);//识别二元加、减法，并构造对应的语法树

basic\_node\* term(bool get);//识别乘、除法，并构造对应的语法树

basic\_node\* prim(bool get);//识别一元加、减法，括号，数值，并构造对应的语法树

public:

CExprParser();//构造函数

virtual ~CExprParser();//析构函数

SyntaxTree\* parse\_expr(const char\* exprText);//解析表达式并构造语法树

};

15. class Expr 表示表达式的类

class Expr

{

private:

string mText;//算术表达式的原始文本

SyntaxTree\* mTree;//表达式的语法树的树根

exprException \*mError;//表达式中存在的错误描述

public:

Expr() ;//构造函数

Expr(const char\* strExpr) ;

~Expr();//析构函数

void setText(const char\* strExpr);//重新填写表达式的完整文本并将之解析为语法树

const string& getText() const;//获取表达式的完整文本

double eval();//计算表达式的值

string prin\_pre\_fix() const;//产生表达式的前缀式

string print\_in\_fix() const;//产生表达式的中缀式

string print\_post\_fix() const;//产生表达式的后缀式

void printTree(ostream& outDevice) const;//先序遍历，打印语法树

const exprException\* getError() const;//获取表达式中存在的错误信息描述

};

16. class exprException 表示异常概念的基本类型

class exprException

{

protected :

long m\_where;//在表达式中的哪个地方出错

string m\_strMsg;//错误描述

exprException() : m\_where(-1){}//缺省构造函数

public :

exprException(const char\* msg, long pos) ;//构造函数

virtual ~exprException(){}//析构函数

const char\* what() const//获取错误描述文本

{

return m\_strMsg.c\_str();

}

virtual exprException\* clone() const//复制对象本身

{

exprException\* newObject = new exprException(\*this);

return newObject;

}

};

17. class CDiv0表示除数为0的错误类型

class CDiv0 : public exprException//除数为0的错误

{

public:

CDiv0(long pos) : exprException("Divided by zero", pos)

{}//除数为0

virtual CDiv0\* clone() const

{

CDiv0\* newObject = new CDiv0( \*this );

return newObject;

}

protected: // 允许派生类访问

CDiv0() {}

};

18. class char\_error表达式存在无效字符的错误类型

class char\_error : public exprException

{

public:

char\_error(char c, long pos);//构造函数

virtual char\_error\* clone() const//对象赋值自身

{

char\_error\* newObject = new char\_error(\*this);

return newObject;

}

protected: // 允许派生类直接访问

char\_error() {}

};

19．class Syntax\_error 表达式中存在结构错误的类型

class Syntax\_error : public exprException

{

public:

Syntax\_error(const char\* msg, long pos) : exprException(msg, pos)

{}

virtual Syntax\_error\* clone() const//对象赋值自身

{

Syntax\_error\* newObject = new Syntax\_error( \*this );

return newObject;

}

protected: // 允许派生类直接访问

Syntax\_error() {}

};

20. class CExprParser词法分析器类型

class CExprParser::CExprLexer

{

private:

const char\* mpText;// 被扫描的字符串起始地址

const char\* mpPos;// 当前扫描到的位置

char lastChar;// 最近一次读到的字符

bool ignoreChar(char c) const;//是否忽略某个字符

bool getChar(char& outChar);//获取字符串中的下一个字符

void backChar();//回退一个字符

long tellg() const

{

return long(mpPos-mpText);

}

public:

CExprLexer(const char\* exprText)//构造函数

:mpText(exprText),mpPos(exprText),lastChar('\0')

{}

~CExprLexer(){}

void setSource(const char\* exprText)//给词法分析器设定值

{

mpText = mpPos = exprText;

lastChar = '\0';

}

t\_token nextToken();//从给定字符串中获取下一个字符

};

21．struct t\_token表示“记号”这一概念的类型

struct t\_token

{

Token\_value tok\_kind;//记号的种类

string tok\_txt;//记号的文本

long tok\_pos;//记号再原表达式串中的位置

t\_token() : tok\_kind(END), tok\_pos(-1) {}//记号的缺省构造函数

t\_token(Token\_value kind, const string& txt, long pos)//记号的非缺省构造函数

: tok\_kind(kind), tok\_txt(txt), tok\_pos(pos)

{}

void clear()

{

tok\_kind =END; tok\_txt.clear(); tok\_pos =-1;

}

};

22. class expr\_testcase表示对单个表达式的测试类

class expr\_testcase//对单个表达式的测试类

{

public:

enum test\_err\_kind//表达式错误类型的种类

{

ERR\_NONE, //表达式没有错误

ERR\_UNKNOWN, //表达式出现未知错误

ERR\_VALUE, //表达式的值计算错误

EX\_UNKNOWN, //未知类型的异常

EX\_SYNTAX, //语法错误 类异常

EX\_CHAR, //无效字符类异常

EX\_DIV0 //除数为0类异常

};

private:

string mExprText;//被测试的表达式文本

test\_err\_kind mExpectedErrKind;//表达式错误种类

double mExpectedValue;//期待的表达式的值

test\_err\_kind mErrKind;//测试中出现的异常种类

double mValue;//测试所得表达式的值

string mErrorMsg;//测试时发生的错误描述

bool passed;//测试结果是否正确

void setError(test\_err\_kind ek, const char\* msg)//设置错误信息

{

mErrKind = ek;

mErrorMsg.assign( msg );

}

bool test\_parse(Expr& testee);//测试表达式的解析操作

bool test\_print(Expr& testee);//测试表达式的打印操作

bool test\_eval(Expr& testee);//测试表达式求值时的错误

public:

expr\_testcase(const char\* exprText, double expectedValue, test\_err\_kind ek =ERR\_NONE)

: mExprText(exprText), mExpectedErrKind(ek),

mExpectedValue(expectedValue), mErrKind(ERR\_UNKNOWN),

mValue(0), passed(false)

{}//构造函数

int run()//执行一个测试用例

};

23. class expr\_testcase\_runner 表示测试用例的执行器

class expr\_testcase\_runner//测试用例的执行器

{

public:

void run();//测试函数

private:

void action(expr\_testcase \*pAllCases, size\_t sz);//执行全部测试用例

};

**算法 2.1 int main()**

作 用：主控函数，也实现对题目所需其他内容的测试。

参 数：无

返回值：总是返回0。

计算过程：

1. 定义了两个类expr\_testcase（对单个表达式的测试）和expr\_testcase\_runner（输出测试结果），实现对表达式的测试功能。
2. 在main函数中调用测试对测试集的测试函数，输出测试结果。

**算法2.2 bool expr\_testcase::test\_parse(****Expr&** **testee)**

作 用： 表达式的解析操作。

参 数： Expr& testee // **testee**为被测试表达式的文本

返回值： true 或者 false

计算过程： 调用算法2.12，对表达式testee进行测试，并进行相关错误处理

**算法2.3 bool expr\_testcase::test\_print(Expr& testee)**

作 用： 表达式的打印操作。

参 数： **Expr& testee**

返回值：true 或者 false

计算过程： 调用算法2.17，对表达式testee进行测试，并进行相关错误处理

**算法2.4 bool expr\_testcase::test\_eval(Expr& testee)**

作 用： 测试表达式求值时的错误。

参 数： **Expr& testee**

返回值：true 或者 false

计算过程：调用算法2.13，对表达式testee进行测试，并进行相关错误处理

**算法2.5 int expr\_testcase::run()**

作 用： 执行测试

参 数： 无

返回值： 整型//表示不同的测试结果。

计算过程：对一个表达式调用上述测试函数来进行测试，根据测试结果返回不同的值来表示不同形式的测试结果。

**算法2.6 t\_token CExprParser::CExprLexer::nextToken()**

作 用： 从给定字符串中获取下一个字符

参 数： 无

返回值： 返回记号的类型。

计算过程：

1. 首先跳过当前表达式中的空白字符，如空格、制表符、换行符等。
2. 然后调用switch…case…语句，来判断当前字符的种类。
3. 若为换行、分号等表达式结束符，或为\*、+、-、/、；、（），则返回其相应记号的种类，然后访问下一个字符。若为数字0~9或’.’， 判断小数点出是否出错。其他情况下，则报错：出现无效字符

**算法2.7 SyntaxTree\* CExprParser::parse\_expr(const char\* exprText)**

作 用：解析表达式并构造语法树。

参 数： **const char\* exprText**

返回值： 返回生成语法树的根节点

计算过程：

1. 先清空之前所留下的信息，准备词法分析器。
2. 然后获取该表达式的第一个记号；
3. 若该记号为END，则返回空指针，表示表达式为空，不再分析。
4. 否则，调用算法2.7，解析得到表达式的根结点，并返回其根结点。

**算法2.8 basic\_node\* CExprParser::expr(bool get,bool afterLP)**

作 用： 识别二元加、减法，并构造对应的语法树。

参 数： 参数为**get**，**bool**类型，来决定是否要获取下一个记号；参数**afterLP**来判断表达式中的右括号是否多余。

返回值： 返回生成的（子）语法树的根结点。

计算过程：

1. 先调用算法2.9，得到加减运算的左操作数子树。（该过程若有错则抛出异常）。
2. 当前记号若为加或减，则获取其第二个操作数子树。（该过程若有错则抛出异常）创建该记号标记的父亲节点，即构造子树。
3. 若为END标志，则表示表达式结束，返回生成的表达式的语法树的根结点。
4. 若为右括弧，则根据afterlp来判断该右括号是否多余，则多余则抛出错误。
5. 否则，为无效字符，抛出Syntax\_error错误类型。

**算法2.9 basic\_node\* CExprParser::term(bool get)**

作 用： 识别乘、除法，并构造对应的语法树。

参 数： **bool get**

返回值： 返回生成的（子）语法树的根结点。

计算过程：

1. 首先调用算法2.10，得到乘除运算的左操作数子树。（该过程若有错则抛出异常）。
2. 当前记号若为乘或除，则获取其第二个操作数子树。（该过程若有错则抛出异常）创建该记号标记的父亲节点，即构造子树。
3. 否则，不做任何操作，直接返回根结点。

**算法2.10 basic\_node\* CExprParser::prim(bool get)**

作 用： 识别一元加、减法，括号，数值，并构造对应的语法。

参 数： **bool get**

返回值： 返回生成的（子）语法树的根结点。

计算过程：

1. 通过传入的参数决定是否获取下一个字符。然后进行对应的分析。
2. 若为NUMBER，则构造语法树的叶子结点，即数值类结点，然后获取下一个记号并返回该结点。
3. 若为一元加或一元减，则调用算法本身，构造其右操作数的子树。（该过程若有错则抛出异常），构造该子树并返回其根结点。
4. 若为左括号，则调用算法2.8，生成该表达式的语法树。若最后记号不为右括号，则抛出异常；否则获取下一个记号并返回该语法树的根结点。
5. 其他情况下，即为无效单词，抛出异常，返回NULL。

**算法2.11 char\_error ::char\_error(char c, long pos) : exprException()**

作 用： 无效字符类型的错误。

参 数： **char c,//**无效字符，**long pos//**为错误出现在表达式中的位置。

返回值： 无。

计算过程：

1. 首先将出错地方以及错误内容记录在错误对象中。
2. 然后对于可打印的字符直接拼接在错误描述信息后面；对于不可打印的字符则将其转换为16进制的表示形式，再进行拼接。
3. 最后，再在错误描述串后拼接表示出现错误的位置。

**算法2.12 void Expr::setText(const char\* strExpr)**

作 用： 重新填写表达式的完整文本并将之解析为语法树。

参 数： **const char\* strExpr**

返回值： 无

计算过程：

1. 清空原有信息，为构造新的表达式的语法树做准备。
2. 将传入的表达式复制给mText，记录表达式的原始文本。
3. 然后，设置一个语法分析器的对象theParser，调用算法2.7，生成表达式的语法树，并将其树根节点返回给mTree。
4. 若解析为语法树的过程中出现错误，则catch，捕获异常，并将错误描述信息记录在mError中，再抛出异常。

**算法2.13 double Expr::eval()**

作 用： 计算表达式的值。

参 数： 无

返回值： 返回为表达式的值。

计算过程：若表达式的语法树的根结点为空，则返回0；否则按层次调用各类中的函数eval（），计算出该表达式的值。

**算法2.14 string Expr::prin\_pre\_fix() const**

作 用：求前缀表达式。

参 数： 无

返回值： 返回该表达式的前缀表达式。

计算过程：

1. 先判断表达式的语法树根结点mTree是否为空。若为空，则判断其表达式的错误描述是否为空，若空，则表明该表达式为空，否则，返回其错误描述信息，该表达式无效。
2. 若根结点不为空，则由语法树对象中的函数获得该表达式生成其前缀表达式，并去掉表达式末尾的多余项，返回retValue表示的表达式语法树的前缀表达式。

**算法2.15 string Expr::print\_in\_fix() const**

作 用： 求中缀表达式。

参 数： 无

返回值： 返回该表达式的中缀表达式。

计算过程：

1. 先判断表达式的语法树根结点mTree是否为空。若为空，则判断其表达式的错误描述是否为空，若空，则表明该表达式为空，否则，返回其错误描述信息，该表达式无效。
2. 若根结点不为空，则由语法树对象中的函数获得该表达式生成其中缀表达式，并去掉表达式末尾的多余项，返回retValue表示的表达式语法树的中缀表达式。。

**算法2.16 string Expr::print\_post\_fix() const**

作 用： 求后缀表达式。

参 数： 无

返回值： 返回该表达式的后缀表达式式。

计算过程：

1. 先判断表达式的语法树根结点mTree是否为空。若为空，则判断其表达式的错误描述是否为空，若空，则表明该表达式为空，否则，返回其错误描述信息，该表达式无效。
2. 若根结点不为空，则由语法树对象中的函数获得该表达式生成其后缀表达式，并去掉表达式末尾的多余字符串，返回retValue表示的表达式语法树的后缀表达式。

**算法2.17 void Expr::printTree(std::ostream& outDevice) const**

作 用： 先序遍历打印语法树。

参 数： **std::ostream& outDevice**

返回值： 无返回值

计算过程：

1. 判断表达式的语法树根结点mTree是否为空。若为空，则判断其表达式的错误描述是否为空，若空，则表明该表达式为空，否则，返回其错误描述信息，该表达式无效；
2. 然后按层次调用printTree函数，先序遍历表达式的语法树，按层次打印遍历结果。

**算法2.18 void basic\_node::generatePostfix(string& retValue) const**

作 用： 求后缀表达式式。

参 数： **string& retValue**

返回值： 无返回值

计算过程：祖先类输出结点自己的后缀式，子树的后缀式由其派生类来实现。

**算法2.19 void basic\_node::generatePrefix(string& retValue) const**

作 用： 求前缀表达式。

参 数： **string& retValue**

返回值： 无返回值

计算过程：祖先类输出结点自己的前缀表达式式，子树的前缀表达式式由其派生类来实现。

**算法2.20 void basic\_node::generateInfix(string& retValue) const**

作 用： 求中缀表达式。

参 数： **string& retValue**

返回值： 无。

计算过程：祖先类输出结点自己的中缀表达式，子树的中缀表达式由其派生类来实现。

**算法2.21 void basic\_node::printTree(ostream& outDevice,int deepth) const**

作 用： 先序遍历，打印语法树。

参 数：**ostream& outDevice,int depth**（缩进）

返回值： 无。

计算过程：祖先类仅输出结点自己的信息，子树的输出由派生类来实现。

**算法2.22 virtual double CDivNode ::eval() const**

作 用： 进行除法运算

参 数： 无参数。

返回值： 浮点数

计算过程：

1. 首先获得该除法结点的右操作数（除数）的值。
2. 判断除数是否为0。若为0，则抛出异常错误。
3. 否则，返回左操作数除以除数的结果。

# 测试用例与测试结果

## 继承与多态

|  |  |
| --- | --- |
| 主函数 | 测试 |
| cout<<"\*\*\*\*动物乐园\*\*\*\*"<<endl;  Animal A("Animal");  Cat C("cat\_tidy");  Dog D("dog\_tidy");  A.iam(); A.hello(); A.common();  C.iam(); C.hello(); C.common();  D.iam(); D.hello(); D.common();  Animal \*p = &A;  Animal \*q = &C;  Animal \*s = &D;  cout << "\*\*\*\*\*\*\*\*\*\*指针" << endl;  p->iam(); p->hello(); p->common();  q->iam(); q->hello(); q->common();  s->iam(); s->hello(); s->common();  Animal & z = A;  Animal & x = C;  Animal & c = D;  cout << "\*\*\*\*\*\*\*\*\*\*\*引用" << endl;  z.iam(); z.hello(); z.common();  x.iam(); x.hello(); x.common();  c.iam(); c.hello(); c.common();  return 0; | \*\*\*\*动物乐园\*\*\*\*  Animal Animal  Animal::hello from Animal  Animal::common  Cat cat\_tidy  Cat::hello from cat\_tidy  Animal::common  Dog dog\_tidy  Dog::hello from dog\_tidy  Animal::common  \*\*\*\*\*\*\*\*\*\*指针  Animal Animal  Animal::hello from Animal  Animal::common  Cat cat\_tidy  Animal::hello from cat\_tidy  Animal::common  Dog dog\_tidy  Animal::hello from dog\_tidy  Animal::common  \*\*\*\*\*\*\*\*\*\*\*引用  Animal Animal  Animal::hello from Animal  Animal::common  Cat cat\_tidy  Animal::hello from cat\_tidy  Animal::common  Dog dog\_tidy  Animal::hello from dog\_tidy  Animal::common |

* 1. 定义表达式类型

|  |  |  |
| --- | --- | --- |
| 序号 | 测试数据 | 计算结果 |
| 1 | 1+2 | TEST expression text [1+2]  Try to parse expression "1+2"  Try to print expression "1+2"  Syntax tree for expression [1+2]:  {  + 2 CPlusNode  1 1 CNumberNode  2 3 CNumberNode  }  Infix notation: [(1 + 2)]  Postfix notation: [1 2 +]  Prefix notation: [+ 1 2]  Try to evaluate expression "1+2"  表达式的值: [3]，期望值:3  TEST \*\*\* passed \*\*\* 没有错误. |
| 2 | 3.1415926+(1.0/.3)\*10 | TEST expression text [3.1415926+(1.0/.3)\*10]  Try to parse expression "3.1415926+(1.0/.3)\*10"  Try to print expression "3.1415926+(1.0/.3)\*10"  Syntax tree for expression [3.1415926+(1.0/.3)\*10]:  {  + 10 CPlusNode  3.1415926 1 CNumberNode  \* 19 CMulNode  / 15 CDivNode  1.0 12 CNumberNode  .3 16 CNumberNode  10 20 CNumberNode  }  Infix notation: [(3.1415926 + ((1.0 / .3) \* 10))]  Postfix notation: [3.1415926 1.0 .3 / 10 \* +]  Prefix notation: [+ 3.1415926 \* / 1.0 .3 10]  Try to evaluate expression "3.1415926+(1.0/.3)\*10"  表达式的值: [36.4749]，期望值:36.4749  TEST \*\*\* passed \*\*\* 没有错误. |
| 3 | （2+1）） | TEST expression text [(2+1))]  Try to parse expression "(2+1))"  Found 'Syntax\_error' exception: 多余的右括弧[)]at 6  TEST \*\*\* passed \*\*\* 错误一致:多余的右括弧[)]at 6. |

# 实验总结

本次是最后一次上机，通过此次上机加深了我对虚函数的运算，对继承和重置的理解。我们在编写程序前，一定要清楚派生类可以访问基类的哪些数据成员及成员函数。通过测试，可以很清楚地看出构造函数与析构函数的调用，虚函数在实际应用是十分有用的。但是在语法树的运用上，我还不是很熟练，以后会通过训练进行提高。在本课程的学习过程中，理解和掌握一些基本概念和思想是十分重要的，我们要通过实际编程不断自我能力，加深对其的理解。