**2024年《面向对象程序设计》上机安排与要求**

班级：22级3班

辅导：毕文静

# 1 上机安排与要求

## 1.1上机时间&地点



## 1.2上机要求：

* 每次上机均进行验收。若当次没有验收则可以推迟到下一次验收（仅适用于前5次上机）。
* 每次上机结束后，提交当次的**源码及设计说明**（对所设计的类、类中重要的方法和数据进行说明。从第3次上机开始，还需说明对前一次上机完成的代码的改动情况）。
* 六次上机结束之后，提交针对**实验2**的**上机报告**（请依照模板进行撰写）。

## 1.3成果提交要求：

1. 每次实验的成果包括以下两部分：
   1. 第1~5次上机后：提交源码和设计说明。
   2. 第6次上机后：提交实验2的最终源码and实验2的实验报告；
2. 最终提交的实验报告必须按照模板文件《面向对象程序设计实验报告-学号-姓名.docx》撰写（需提交word版）。电子版报告的文件名统一命名为：

OOP实验报告-学号-姓名.docx 。

1. 各次提交的源代码中，仅提供你编写的源文件（\*.hpp, \*.cpp）即可。
2. 每一次提交的实验成果文件请按下图示意的目录结构整理：

/学号-姓名-OOP第N次上机

|\_\_ OOP第N次上机设计说明.docx

|\_\_ 源代码目录和源代码文件

|\_\_ OOP实验报告-学号-姓名.docx

注1：你的源程序可按需定义多个目录（文件夹）。

注2：实验报告仅在第6次上机后提交。

注意：不要提交开发环境自用的其他文件、编译生成的其他文件、可执行文件。它们很大、很占空间、会被扣分！

1. 将上述目录内全部文件、子目录压缩打包为一个 ZIP文件，命名为

“学号-姓名-OOP第N次实验成果.ZIP”。

1. 每次上机结束后，则将上述第5条的ZIP文件**提交到“学在西电”课程作业板块**（到时会有对应的作业通知）。

# 2 实验1：

实验目的：

本次实验要求使用“类”相关的基本机制来设计并实现计算器程序，以熟悉 C++ 提供的面向对象基本概念和机制，掌握数据抽象的基本手段，用类型上的操作来封装数据结构，为后期学习并掌握类型继承、方法重置等机制奠定基础。

下面两道题目中，你应至少完成其中一道题。

题目1：定义并实现class Date。

Complete and test ***Date*** of §10.2.

Reimplement it with ‘‘number of days after 1/1/1970’’ representation.

[提示]

1. 上机实验、设计说明中只需要给出“Reimplement”版的内容，并只需提交该版本的源代码。
2. 本文附录1中给出了该实验的实现要点。

题目2：定义并实现一个表示“算术表达式”的类型。

Define a class for analyzing, storing, evaluating, and printing simple arithmetic expressions consisting of integer or floating number constants and the operators +, -, \*, and /. The public interface should look like this:

class Expr {

// …

public:

Expr(const char\* exprText); //对象初始化

***double*** eval(); // 计算表达式的值并返回该值

void printPostfix(); // 在cout输出表达式的后缀式

void printPrefix(); // 在cout 输出表达式的前缀式

… 其他你想添加的方法 …

};

The string argument, **exprText**, for the constructor ***Expr::Expr()*** is the text of an expression. A user program might look like this:

*Expr x*("*( 1*/(*4*+*1) )\*2.0*");

*cout* << "*x* = " << *x*.*eval*() << "*\ n* ";

*x*.*printPostfix*();

Experiment with different ways of printing the expression: fully parenthesized, postfix notation, prefix notation,etc.

[提示]

1. 本文附录2中给出了该实验的实现要点。
2. 可参考代码：

学在西电/资料/课堂中的示例程序/classExpr\*\*\*.zip，有两个版本可供参考：

1. classExpr简易的面向对象版，程序结构简单
2. classExpr-OOP-Version（复杂的编码对象版，类更多），程序结构优点复杂。

# 3 实验2：实现一个购物车模拟系统

（在第2-6次上机时，分别完成下文的5项任务）

模拟系统的用户分为两类，管理员和顾客：

* 管理员的主要功能是维护系统中的后台数据，
* 顾客可以通过系统实现商品的购买。管理员只有一名，顾客有多名。

本题的实验目的：

本实验促进学生将面向对象范型相关机制灵活运用于实际程序设计活动，以熟悉、掌握各机制的作用、语法结构。

## 第1次任务（功能需求）：

1. 实现登录及注册功能

1. 管理员和顾客通过账户名和密码登录系统。
2. 管理员的账户和密码通过提前预设的方式指定。
3. 顾客在首次登录系统时，需要先进行注册，注册成功后设置账户和密码（注册时需要提供的信息由你自行设计）。

说明：管理员必须登录系统之后才能操作后台数据；

顾客未登录系统时，可以查询商品信息。若要购买商品，必须登录系统。

2. 管理顾客的信息

（1）保存顾客注册的信息和账户信息；

（2）顾客可以修改自己的密码。

## 第2次任务（功能需求）：

1. 实现管理员的商品管理功能

（1）能够根据管理员的输入，添加新的商品信息，包括商品的名称、单价、描述和库存等信息（可以自行添加你认为必要的信息）；

（2）能够删除指定商品的信息；

（3）能够修改指定商品的信息；

2. 实现顾客查询商品信息的功能

（1）能够根据用户给出的商品名称查询商品信息，并显示查询结果；

（2）能够根据用户给出的关键字（与商品名称不一致），给用户推荐最接近的商品信息。

3. 实现购物车功能

（1）为每个顾客创建购物车；

（2）能够根据顾客的输入，向购物车中添加商品；

（3）顾客能够查询购物车中的商品信息；

（4）顾客能够删除购物车中的商品；

（5）顾客能够修改购物车中某个商品的购买数量。

## 第3次任务（功能需求）：

1. 顾客的购买功能

（1）能够计算购物车中所有商品的总价，并将总价显示给用户；

（2）顾客能够选择购物车中的商品进行结算，计算并显示结算商品的总价；

（3）从购物车中删除已经结算的商品。

2. 优惠与折扣

（1）给顾客随机派送优惠券（优惠券面值也可以随机），顾客结算商品时可以输入优惠券码，对结算总价进行折扣计算。

（2）活动折扣。特定优惠活动的折扣，结算时按折扣规则进行折扣计算。

## 第4次任务（功能需求）：

实现购物车持久化与恢复：

- 保存购物车：能够将购物车中的商品信息保存到文件中，以便下次运行程序时能够恢复之前的购物车数据。

- 恢复购物车：能够从文件中读取之前保存的购物车数据，以便在程序启动时能够加载之前的购物车信息。此后，允许顾客可通过已有功能对购物车信息进行编辑（增加、删除、修改）。

## 第5次任务（功能需求）：

扩充购物车功能：【可选】：分数占比为实验成绩的10%

（1）购物历史：能够保存顾客的购物历史记录，包括购买的商品和购买时间

（2）购物数据分析：根据顾客选定的时间，按照所购买商品的种类展示购买的金额。

# 附录1 关于class Date 的实验要点

要求分析：

类型 Date 是教材第10.2--10.3节的讲解用例，在第2次实验中曾有类似的题目。

教材中对日期三要素（日、月、年）先使用三个int整数分别表示，10.3节又用enum Month作为月份的类型。

第1句要求：首先按照教材第十章介绍的知识定义/实现该类型的所有特征（数据结构和操作），并进行一些测试工作。教材中已经指出了日期相关的操作集，你也可以对操作集扩充。

第2句要求：当上步所得程序没有问题之后，**保持 Date 类型公布的对外接口（包括类名、公开的操作及其接口（即函数原型））不变**的前提下，令每个日期对象的数据结构仅仅有一个整数（删除原先的日、月、年三个数据成员），该整数表示“从1970年1月1日开始的**天数**”的整数。如

1969年12月30日用-2表示，

1969年12月31日用-1表示，

1970年1月1日用0表示，

1970年1月2日用1表示，…，

1971年1月1日用365表示，…。

反之，你也应当将-1解释为1969年12月31日，将0解释为1970年1月1日，…..

**提示1：需要考虑 闰年 和 二月份 的特殊处理细节。**

**提示2：C/C++标准库仅支持1970/1/1及之后的日期，且最大日期取决于你采用的开发环境——编译器+操作系统。所以你在实现时，可以不支持1970/1/1之前的日期。**

**\*\*重要提示\*\*：你所提交的实验成果中，无论是上机报告、还是源代码，都【只需】出现满足第2句要求的Date的定义和实现。**

实现要点：

1. 首先根据教材第10章内容整理并定义好Date的接口，然后再考虑其实现细节（包括数据结构、操作的实现、必要的访问控制说明等）。

--- 先定接口再考虑实现，这是所有编程方法中的基本做法。定义函数、模块、类型都是这样。

1. 确定‘日期’对象的**缺省初始值**的方法不唯一。

教材中使用两个成员 Date::set\_default() 和 Date::default\_date、配合构造函数实现了日期的缺省初始值。大家在实现时，还可以借助“系统当前日期”来设置缺省初始值，其要点如下：

*#include <ctime> // 此文件是 C 语言的 time.h 文件的简单包装；*

* 1. main()开始时，调用std::time()获得当前系统时间 T0（该时间就是从1970年1月1日0时0分0秒到目前为止的秒数）；
  2. 调用std::localtime(T0)，将T0分解为年、月、日、时、分、秒等要素；
  3. 依据上步所得的年、月、日，调用 Date::set\_default() 变更缺省日期。

1. 当用“天数”作为数据结构时，你需要 C/C++ 的库函数 mktime()，它实际上是 std::localtime() 的逆运算。
2. 【强烈建议养成好习惯】采用多个源代码文件。这样是实践中常采用的方式，便于代码维护、隔离复杂性、减少对用户的干扰。
   1. 头文件 date.hpp。该文件中出现class Date{…}; 的定义。
   2. 实现文件 date.cpp。该文件中出现 Date 的成员函数的实现（定义）、静态数据成员的定义（带初始化）。

\*\*\* 为验证（体验）成员函数的两种定义策略，你在编程时，可将函数体较短的（不超过3行）成员函数定义直接出现在类定义之内(Date.hpp)，而将较长的成员函数定义在类定义之外(Date.cpp)。

OR

全部采用后一种。

* 1. 测试程序文件，你可将该文件命名为 testDate.cpp。该文件包括main函数定义、其他测试所需的程序定义。

其他实验都这样做。

* 1. 在你的开发环境（如Visual Studio, Code::Blocks等）中，需要将上述\*.cpp文件全部添加到同一个项目（project）中，然后再编译、生成可执行程序。

# 附录2 关于class Expr 的实验要点

要求分析：

本题实际上是对第六章给出的桌面计算器的进化——将原有的全局变量、全局函数等通过 class Expr{ … } 封装起来，即将计算器抽象为一个类型。这样，程序中就可以同时存在多个计算器对象了。只要你读懂了第6章的计算器程序、自行完成了第3次上机，这个题目就容易了。

题干先说明并演示了 Expr 类型的接口，然后要求给出：

1. 构造函数的参数是一个常规中缀形式的算术表达式的原始文本。该表达式既可以是数值（整数和/或实数），也可以是数值和加、减、乘、除等算术运算、括弧构成的复杂表达式。
2. 接口操作 print()，要求可将表达式按多种形式输出（详见实现要点3）：
   1. full parenthesizd——每个子表达式均用一对括弧包围，即中缀式；
   2. postfix notation——后缀式（操作数在前，操作符在后）；
   3. prefix notation,前缀式（操作符在前，操作数在后）。
   4. 为同时提供多种输出，你可将要求的 print() 分裂为3个print\_\*\*\*()函数，或者为 print()增加一个参数来区分输出形式。
3. 题目并未限定类型内部的数据结构，所以要自己设计，只需让接口及其实现满足要求即可。

实现要点：[本题是“接口简单，但实现复杂”的典型代表]

要点1：考虑到表达式的字符串来自类型的使用者，所以要考虑字符串无效的可能，如“1+”，“1+(2$”等。建议大家的程序应能检查出可能的错误，并进行适当处理。

处理措施可参见第8章的exception机制、第10章Date构造函数中抛出异常的策略、第3次上机实验。

要点2： class Expr内对表达式的存储策略

教材的原始题目提出了两种数据结构（存储方式）：链表、字符串。

* 链表方式。每个结点存储表达式中的一个单词，如 “1+2\*3”应表示为这样子：

List\_head🡪 1 🡪 + 🡪 2 🡪 \* 🡪 3 🡪 NULL

但是，这个结构不便于求值和输出，因为体现不了计算顺序。

* 一个字符串。相对于链表，这种方式更加不方便，因为每次对表达式求值、打印时都需要先分解表达式。

这里给大家介绍一种经典的、更便利的存储方式——**语法树**。这种树具有以下特点：

* 非叶子结点均表示一个运算（可用运算符标记该结点）；
* 叶子结点表示（存储）表达式中最基本的操作数（即数值），如1, 2.3等。
* 任何一颗只有父子关系的子树中，父结点对应某个特定运算，其孩子结点对应该运算的操作数。因此，孩子结点的数量 = 父结点运算所需的操作数个数；
* 树结构本身体现了运算次序，因此中缀表达式中用来改变运算次序的圆括弧不出现在树中。

提示：语法树所有结点内部，除存储运算标识、数值之外，还可以存储其他信息，如该运算的结果值、（子）表达式的原始文本等。

如 1+2、1\*(2+3)的语法树分别如下： （怎么遍历才能求出其值、打印表达式？）

+

1

2

\*

1

+

2

3

当你认真观察语法树之后，就会发现：

1. 对于每颗子树，其树根总是对应（子）表达式中的最后一个运算。

对于整颗树，其树根总是对应完整表达式中的最后一个运算。

1. 基于语法树可以非常方便地实现表达式求值、按照后缀形式、前缀式、中缀式等方式输出等操作。

提示：这就需要你定义能表示语法树的相关类型，如最简单的方式就是定义一个 class ExprNode{…}，它是语法树所有结点的共同类型，提供结点上的维护、遍历等操作。

提示：为简单起见，此处将语法树等同于树结点。注意和上一题的区别！

类型 ExprNode 可以是类型Expr 的（私有的）成员类型，也可以全局类型，无论如何，用户侧均不需要知道该类型的存在！

要点3：关于表达式的若干种输出方式

要求的几种输出形式都是文本，具体形式应至少有三种。

a. fully parenthesized：可定义成员函数 print()提供该结果。

这种形式实际上就是常用的“中缀式”，要求为每个子表达式都必须（或按需）用一对圆括弧包围，必须与原表达式等价。

如： 1+(2\*3)可输出为：(1+(2\*3))，或((1)+((2)\*(3)).

此时需对语法树进行“中序遍历”。

b.postfix notation（后缀式）：可定义成员函数 print\_postfix() 提供该结果。

这种形式的基本特点是“操作数在前，操作符在后”，括弧是不需要的。

如： 1+2的这种输出为：1 2 +，注意要添加适当的空格。

1\*(2+3)的这种输出为：1 2 3 + \* ，注意要添加适当的空格。

这就要求对语法树采用“后序遍历”。

c. prefix notation（前缀式）：可定义成员函数 print\_prefix() 提供该结果。

这种形式的基本特点是“操作数在后，操作符在前”，括弧是不需要的。

如： 1+2的这种输出为：+ 1 2 ，注意要添加适当的空格。

1\*(2+3)的这种输出为：\* 1 + 2 3 ，注意要添加适当的空格。

这就要求对语法树采用“先序遍历”。

要点4：将字符串形式的表达式转换为内部的语法树

因为Expr的构造函数的参数是一个中缀表达式的完整文本（字符串），所以要实现表达式的其他操作，就必须对该字符串进行解析（拆分），并将解析结果存储到合适的成员变量中。

你必须考虑在何处解析字符串：在构造函数中？在其他函数中？是否需要增加一个/一组辅助的成员函数（及数据成员）、一个/一组辅助的类型，来专门进行字符串的解析、构造+遍历树？

你必须考虑如何解析字符串？当然可以采用/模仿第6章给出的桌面计算器程序那样的思路了，包括函数 expr、term、prim、get\_token，以及枚举类型 Token\_value等。

这里给大家提供一个参考的[半面向对象的]程序框架（每个类应该有各自的头文件和实现文件，头文件中最好不要给出成员函数的实现）。

辅助类型1：词法分析器类

class Lexer{ //将第6、8章的**词法分析器**模块提升为这个类

public: //该成员类型也可定义为全局类型

enum Token\_type{ NUMBER, PLUS=’+’, MINUS=’-’ ……};

public:

Lexer(const char\* str) { exprText = str; }

Token\_type get\_token(); // 读取串中的下一个记号(TOKEN)

~Lexer(){ }

Token\_type currentToken() const { return curr\_tok; }

Double value() const { return number\_value; }

private:

const char\* exprText; // 仅需要记录原始表达式的地址

Token\_type curr\_tok; // 到目前为止，识别到的最后一个记号的种类

double number\_value; // 若 curr\_tok=NUMBER，该成员记住它所表示的数值

… … … …

};

词法分析器类型对外提供的关键操作是 get\_token()，在第6章定义词法分析器的全局变量，改为这个类的数据成员，根据需要设定合理的访问控制。原本属于该模块的辅助类型 enum Token\_type 可定义为全局类型，亦可定义为 Lexer类的成员类型（但需public）。

另注意：get\_token()实现中，应将第6章代码中的读输入的操作，如 cin >>… , cin.getch() 等均修改为扫描字符串 exprText方式来实现。

辅助类型2：语法树结点类

#include “Lexer.hpp”

class exprNode{

public:

exprNode(Lexer::Token\_type tokType /\*见辅助类型1 \*/ ,

exprNode\* leftChild, exprNode\* rightChild, // 结点的左、右孩子（子树）

double value1 =0); // 结点中存放的数值，如前图中的1、2等

~exprNode();

Double value() const ; // 计算并返回表达式的值

exprNode\* getLeftChild() const { return left;}

exprNode\* getRightChild() const { return right;}

std::string getText\_mid() const; // 返回中缀式字符串，需对 语法树 中序遍历

std::string getText\_pre() const; // 返回前缀式字符串，需对 语法树 先序遍历

std::string getText\_post() const; // 返回后缀式字符串，需对 语法树 后序遍历

……

Private: //下面是存放在树结点中的数据

Lexer::Token\_type tok; // 记住该结点的性质：哪个运算？数值？

double value; // 若 tok 是Lexer::NUMBER，则该成员存储对应的数值

exprNode \*left, \*right; // 指向两个孩子结点的指针

};

该类型表示语法树的一个结点，该类型也简介表示语法树。

其构造函数是在生成新结点对象时，将参数指定的数据存储到结点自己的数据成员中。

其析构函数 ~exprNode() 释放本结点的左子树、右子树： delete left; delete right。

成员 getText\*\*\*() 则通过遍历以当前结点为根的子树以获得对应的字符串。如

string exprNode:: getText\_mid () //中序遍历，构造表达式之中缀式文本

{ string retValue = "(";

if(NULL != left) retvalue += left-> getText\_mid ();

switch( this->tok ) // 拼接上当前结点的记号文本，这段代码可定义为另一辅助的成员函数

{ case Lexer::PLUS: retValue += “+”; break;

…

case Lexer::NUMBER:

{ char str[20];

sprintf(str, “%lf”, this-> value);

retvalue += str;

}

Break;

…

} // end of switch

if(NULL != right) retValue += right-> getText\_mid ();

return retValue += ")";

}

成员函数 value() 通过遍历以当前结点为根的子树，以计算表达式的值，这个函数的计算请大家自己完成。

辅助类型3：语法分析器类

#include “exprNode.hpp”

class Parser{

public:

……

exprNode\* parse\_expr(const char\* strExpr);

private:

std::string exprText; 存储表达式文本

Lexer \*theLexer; 词法分析器对象，用于下面三个函数

epxrNode\* expr(bool get);

epxrNode\* term(bool get);

epxrNode\* prim(bool get);

……

};

将第6、8章的**语法分析器**模块提升为这个类，其主要职责是对字符串形式的表达式进行解析，并在解析过程中逐步构造表达式的语法树。解析过程中用到Lexer::get\_token() 以得到表达式中的一个个记号对象。

（按常见做法）解析整个表达式并构造语法树应包装为一个总控函数，且为public:

epxrNode\* Parser::parse\_expr(const char\* str)

{ exprText = str;

theLexer = new Lexer( exprText.c\_str()); //这个对象需要在Parser的析构函数中delete掉

Lexer::Token\_type tok = theLexer->get\_token();

If( tok == Lexer::PRINT || tok == Lexer::END) return NULL; // 空表达式

exprNode\* root = expr( false );

delete theLexer; theLexer = NULL;

return root; //语法分析器不保存语法树!!

}

该函数返回所得的语法树树根。若表达式有错误，或字符串为空串，均返回 NULL，或者抛出异常来报错。

Class Parser中的expr(),term()，prim()函数，均是对第6章实现的expr(),term()，prim()等的修改。关键修改是：将原先函数体中的算术运算，替换为构造语法树（子树/结点）的操作。这里给出 expr()的修改结果仅供参考，其他类似。

epxrNode\* Parser::expr(bool get)

{ epxrNode \* left ; // 左操作数的树根

epxrNode \* right; // 右操作数的树根

left = term(get); // 得到+、-的左操作数子树

if(NULL == left) return NULL; // 第1个操作数不正确的话，就不再分析了.

for(;;) // ``forever'' for ([+|-] term)\*

{

switch (theLexer-> currentToken () ) // 当前记号种类

{

case Lexer::PLUS: // 二元加法

right = term(true); // 得到 + 的右操作数子树

if(NULL == right) // 第2操作数不正确, 必须销毁第1操作数的子树

{ delete left; return NULL;

}

// 创建 加号 标记的父亲结点，暨 构造子树

left = new exprNode(Lexer::PLUS, left, right);

break;

case Lexer::MINUS: // 二元减法

right = term(true); // 得到 - 的右操作数子树

if(NULL == right) // 第2操作数不正确, 必须销毁第1操作数的子树

{ delete left; return NULL;

}

// 创建 减号 标记的父亲结点，暨 构造子树

left = new epxrNode(Lexer::MINUS, left, right);

break;

default:

return left;

}

}

return NULL; // 这里永不可到达

}

基于以上代码，终极类型的部分实现就可以简化如下：

class Expr {

public:

Expr(const char\* str){

Parser theParser; // 该对象不应该定义为数据成员

root = theParser. parse\_expr ( str );

}

double val() { if(NULL != root ) return root->value(); }

void print——mid() {

std::string txt = root-> getText\_mid();

cout << txt << endl;

}

…………………….

private:

exprNode\* root; // 指向表达式的树根结点

}

=== 本文完 ===