目录

1. 实验环境、题目、目的 1

1.1. 实验环境 1
1.2. 实验题目 1
1.3. 个人的实现程度以及原因 2
1.4. 实验目的 2

2. 设计过程 3

2.1. 前言 3
2.2. 问题的整体构架 3
2.3. 算法核心：最长公共子序列(LCS) 4
2.4. 扩展 5
2.5. 命令行 7
2.6. 类的关系 9
2.7. 合并功能 11
2.8. 代码的其他细节 11

3. 设计的详细信息 12

3.1. 程序的调用过程 12
3.2. 类 Compare 的功能与实现 13
3.3. 类 Folder 的功能与实现 14
3.4. 类 LCS 的功能与实现 15
3.5. 主函数 main 的功能与实现 17

4. 测试数据及其结果 18

4.1. 测试数据 18

5. 试验结束之后的思考 21

5.1. 程序暴露出的问题 21
5.2. 实验体会 21

1. 实验环境、题目、目的

1.1. 实验环境

表 1.1：系统配置、编译环境、其他条件

CPU	Core i5-7300HQ @2.5GHz
RAM	16G
硬盘	西部数据蓝盘 3D 500G
系统	Windows 10 18362.30
电源性能	连接电源、均衡
其他同时运行的程序	Word，VSCode，TIM，Chrome
编译命令	clang++ **.cpp -o **.exe -std=c++17 -static-libgcc --target=x86_64-w64-mingw -O2

其中编译命令中，**表示某些文件名。由于使用大量的 STL 内容，因此打开 O2 提升性能（实际上 Visual Studio 也是默认打开 O2 的），否则速度太慢。不过为了防止过慢，基本所有的都是简单数据结构和简单算法，比如 vector 和 sort，pair 之类。

本次实验的所有代码都移交给 Git 托管，不包括内容详见.gitignore。

1.2. 实验题目

注意：下面题目内容是从实验题目 Word 文件复制，可能最终实现的题目有点差别。

有两个内容相似的文本文件，一个是未修改过的老版本，一个是修改过的新版本。要求实现以下功能：

1) 比较两个文件内容的差异，新老文件每行前均显示行号，比较结果要求能区分并标记出插入、修改、删除和移动的数据行;

2) 能够从一个文件中选择存在差异的数据行，合并到另一文件中，然后保存合并结果。

3) 对两个文件夹内的多个文件或子文件夹进行比较，标记出存在差异的文件，可从一个文件中选择一个文件复制到另一个文件夹中

1.3. 个人的实现程度以及原因

1) 考虑到大型软件的核心比较程序都没有界面（比如 windows 自带的 diff，git 的 diff），也有好用的软件（但是我没听说过）基于 python 实现了界面，VSCode 是基于 chrome 的渲染引擎实现的，因此对于任何的输出结果都可以方便的显示，因此我这里选择命令行实现。
2) 考虑到所有的 diff 算法都不识别移动，因此移动被忽略掉。举个例子：两个 10k 行的 bin 文件，每一行是 1-10 之一，在这种情况下，diff 无法识别出修改和移动。
3) 考虑到这个程序应当有一定的用户体验，我针对体验进行了一些优化。比如只有两个文件的时候是直接比较，但是两个文件夹则是取两者相对路径的交集进行比较。
4) 考虑到命令行的要求，我加入了几个可选项，方便对比。实际合并的时候会忽略选项，因为这影响到了文件的真实性。
5) 设计主要参考了代码需求，甚至没有考虑过 docx 之类的需求。

1.4. 实验目的

本实验的主要目的再于以下几点：

1) 小规模软件的构建。
2) 字符串的±1调整。
3) Windows 系统或者 C 底层的应用。
4) 设计模式的简单应用。

2. 设计过程

2.1. 前言

麻雀虽小但是五脏俱全，这个小软件花费的时间其实一点都不少，因为需要了解到很多东西才能比较严谨的实现每一个部分。

由于细节比较多，因此打算按照整体思路→算法→扩展→命令行→类的关系→合并功能一步一步地说明，这也是我大致的开发过程，只是少许不同而已。区别大概就是在算法的同时有简单的命令行加入以及一些细节问题。

注意，这只是一个设计过程，是在最小化知识量的情况下帮助你完成这份代码的内容，而不是详细的设计，甚至有可能你实现出来的内容和这份报告的完全不一样。详细设计请看第三部分。

2.2. 问题的整体构架

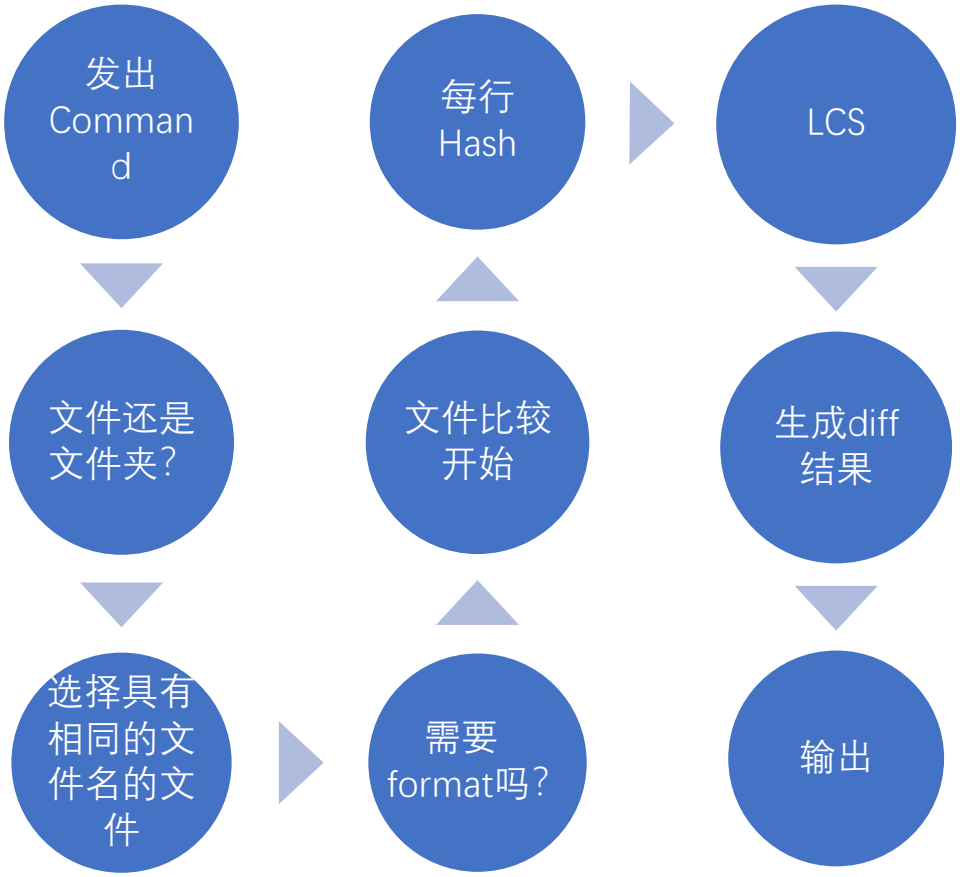


图 2.1 程序的整体思路[出自开题报告的 ppt]

从上图可以看出问题的解决方案的整体构架，具体的代码可以在 main 的

Compare 构造函数一步一步追踪出来。

由于一次不可能实现出整个程序，因此我打算从简单的出发：先不考虑命令行，以单个文件的比较和算法为核心建立出整个程序，之后再进行扩展。

2.3. 算法核心：最长公共子序列(LCS)

在一开始挑选算法的时候，有很多种大型软件和论文可供挑选进行参考，结果发现要么是 LCS，要么是 LCS 的加强或者魔改版、加入限制条件等等。于是最终选定 LCS 作为实现算法。

最长公共子序列是一个这样的问题：有两个序列，问其中的哪一部分公共部分最多，这一部分可以由很多个序列组成，不必要连接在一起。例如：

abcdefg

cedabeg

的 LCS 是 abeg，这个序列在下面是一个整体，但是在上面是三个部分。

那么核心就很明显了：意图求出两个文件的差异，就是反过来求出两个文件的公共部分。如果我们能保证公共部分最多，即差异最小，那么对于人来说也是最友好的。就算是两个文件夹进行比较，也只是多次调用这个算法而已。

求出 LCS 的算法基于动态规划。我们考虑一个二维数组f[i][j]，用于记录序列A[1~i]和B[1~j]中的最长公共子序列（为方便起见，这里和数组不太一样，是从 1 开始的）。那么就有这样的状态转移方程：

$$\mathrm{f[i][j]} = \begin{cases} f[i-1][j-1]+1 & A[i] == B[j] \\ \max(f[i-1][j], f[i][j-1]) & else \end{cases}$$

其中边界条件为$\mathrm{f[i][j]} = 0, \mathrm{iff\ i} == 0 \mid \mathrm{j} = 0$。

算法的伪代码是这样的：

```
function LCS(X[1..m], Y[1..n])
    初始化边界条件
    for i := 1..m
        for j := 1..n
            按照状态转移方程修改 f
```

函数不需要返回值，因为可以包装在类中。

虽然有了算法核心，但是这里还有两个问题需要解决：

1) 输入。我们如何保证把文件转化成一系列数字？对于大部分的文件，可以

考虑把文件按照行分解，并按照内容进行hash，这样得到的序列能保证离散度（前提是hash靠谱）和内容压缩。

2) 输出。我们根据这个算法能求出对应位置的 LCS 长度，但是无法求出两行是否不相同。相对而言这个问题更复杂一些。考虑将不同转化为相同，只要相同的两行之间肯定不相同。与此同时，最长公共子序列的求解过程其实暗含了哪些相同，哪些不同。

2.4. 扩展

有了之前的基础，我们可以轻而易举地写出整个算法的代码，其中大致的思路如以下所示。注意，这里的限制非常多，主要有单文件比较，没有命令行参数。

```
function main()
    获取输入
    打开文件，获得文件的每一行
    用 LCS 比较每一行的内容
    求出相同的行号
    根据行号求出修改过的痕迹，并输出
```

在这里我们需要加入文件夹的比较。根据程序的要求，需要考虑三种情况。

1) 单文件对单文件。在这里程序的处理是无视所处的文件夹，直接进行比对。

2) 单文件对文件夹。在这里程序的处理和文件夹对文件夹相同。

3) 文件夹对文件夹。考虑到文件夹是按照d:\tmp类型输入的，程序会抽取出两个文件夹下相对路径相同的所有文件一一比对。这是考虑到类似于git的代码仓库管理的功能，也是设计时首先想到的要求。

为了方便抽取出文件夹，这里需要一个描述文件夹下所有文件相对路径的类。

```
class Folder
{
private:
    string _M_base_dir;
    vector<string> _M_ext_dirs;
    //判定是否为单个文件，这样可以防止单文件对比单文件的不匹配
    bool _M_only_file;
    //私有化更新，防止外部随意调用
    void _M_update_base(const string &filedir);
    void _M_update_ext(const string &dir);
    Folder() {} //私有化构造函数，防止默认构造
```

```
public:
    Folder(const string &filedir);
    void print_everything(); //验证用
};
```

这个文件夹的最主要目的是描述文件夹下所有的内容，其中_M_update_base进行更新根路径，而_M_ext_dirs存储所有的相对路径。为了方便进行特判，加入单文件标志。初始化时会调用两个私有函数，并且默认初始化函数使用private进行屏蔽防止随意调用。

整个类中最核心的是更新路径的实现，为了方便阅读，进行了一定程度的删减，具体的可以在hpp\Folder.impl.hpp中查看。

```
void Folder::_M_update_base(const string &filedir)
{
    //首先清空文件属性
    clear();
    _finddata_t fileinfo; //C 的结构体，用于访问文件系统
    long hfile = 0;        //文件句柄
    if ((hfile = _findfirst(filedir.c_str(), &fileinfo)) != -1)
    {
        //如果是文件
        if (fileinfo.attrib & _A_ARCH)
        {
            _M_base_dir = filedir.substr(0, pos);       //basedir 更新
            _M_ext_dirs.push_back(filedir.substr(pos)); //更新 extdir
            _M_only_file = 1;
        }
        else if (fileinfo.attrib & _A_SUBDIR)
        {
            _M_base_dir = filedir; //更新文件夹
            if (_M_base_dir.back() != '/' && _M_base_dir.back() != '\\')
                _M_base_dir.push_back('\\'); //如果没有则自动补全
        }
    }
}
```

这是获取根路径的过程，其中最重要的是结构体_finddata_t，包含在<io.h>中，上下文是这样的：

```
  struct _finddata32i64_t {
    unsigned attrib;
    __time32_t time_create;
    __time32_t time_access;
    __time32_t time_write;
    __MINGW_EXTENSION __int64 size;
```

```
    char name[260];
  };
#define _finddata_t _finddata64i32_t
```

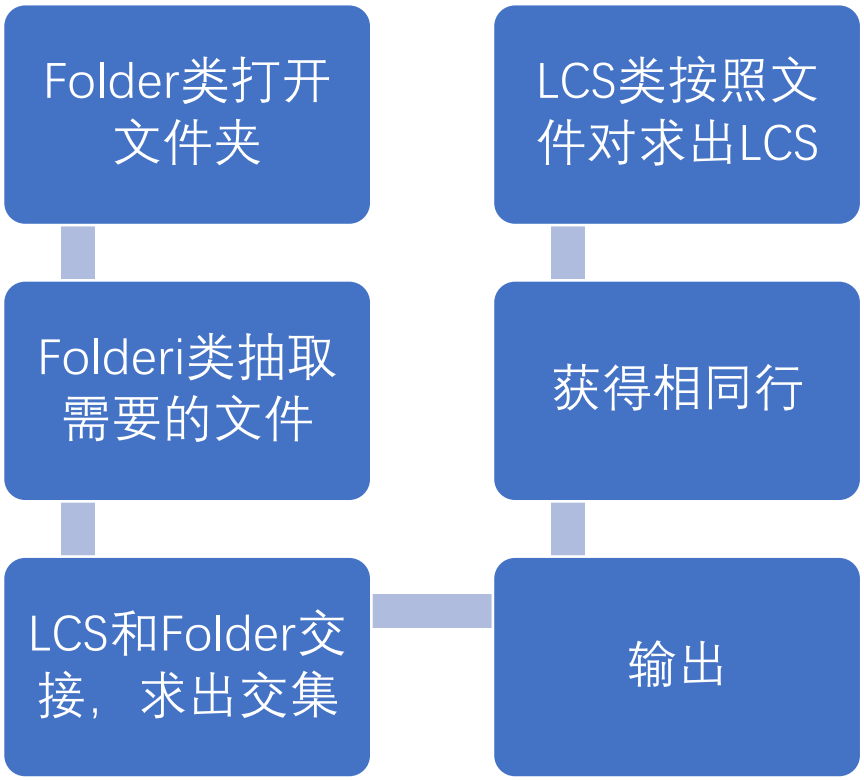
其中每一项的意义是：文件属性，文件创建时间，文件最后一次访问的时间，文件最后一次写入的时间，文件长度，文件名，如果无法定义（比如无法写入，对应项为−1。其中最重要的是attrib和名字name，一个用以定性文件，一个用于路径扩展。其中_A_ARCH和_A_SUBDIR都有文件中定义：

```
#define _A_NORMAL 0x00
#define _A_RDONLY 0x01
#define _A_HIDDEN 0x02
#define _A_SYSTEM 0x04
#define _A_SUBDIR 0x10
#define _A_ARCH 0x20
```

属性attrib只可能是以上六项中若干项的逻辑或。

根据这点可以轻而易举地写出更新子文件路径的方法，只需要注意这是一个递归过程即可。如果根路径是一个文件，这个方法不应该运行，这应当在初始化中限定，而不是更新子文件的入口处判断。

流程走到这里，可以发现代码结构是这样的：



2.5. 命令行

作为一个命令行程序，没有命令行参数是不可能的。命令行参数这么迟才会实现的主要原因有两点：首先是命令行参数不是算法的核心，第二是没有参数程序也可以运行。

设定参数有很多需要考虑，我查看了 linux 下的 diff 参数，选择了一些比较实用的参数，可能看上去比较相似，但是是最实用而且实现难度不大的比较参数。

```
const string _C_format = "--format";                          //代码格式化
const string _C_ignore_space = "--ignore-space";              //行前与行后空格
const string _C_ignore_all_space = "--ignore-all-space";      //所有空格(包括代码内的)
const string _C_ignore_blank_lines = "--ignore-blank-lines"; //空行
const string _C_ignore_tabs = "--ignore-tabs";                //所有 tab
```

其中—format是自行添加的，主要用于代码的格式化，如果不了解的可以去了解一下 clang-format 这个软件。

除此以外，命令行还需要能够识别出输入的参数，并进行分类。通过

```
int main(int argc, char **argv)
```

进行参数的识别，其中argc代表输入的数量（包括程序本身），argv代表输入的每一个参数。除此以外，由于程序还需要进行分类，并且我更希望无论参数出现在哪里都能识别出来，因此我选择了这样处理：

1) 把所有的参数放进一个 vector 中。
2) 遍历所有参数，如果在能接受的参数表中则将对应位置为 1。如果不是而且不以短划线开头，那么认为这是一个文件，不论是否输入正确。

具体的实现则是一个循环：

```
    for (const string &_argv : _argvs)
    {
        if (_argv == _C_format)
            _cmds[0] = 1;
        else if (_argv == _C_ignore_space)
            _cmds[1] = 1;
        else if (_argv == _C_ignore_all_space)
            _cmds[2] = 1;
        else if (_argv == _C_ignore_blank_lines)
            _cmds[3] = 1;
        else if (_argv == _C_ignore_tabs)
            _cmds[4] = 1;
        else if (_file_cnt < 2 && _argv[0] != '-')
            _file_dir[_file_cnt++] = _argv;
```

```
    }
```

对于命令行的处理则主要由 LCS 处理，每一对文件的 LCS 都获得 main 输入的参数，虽然看上去很多，但是因为只是一群 bool 变量，实际占用空间并不大。如果参数包含了—format，则直接通过 dos 调用 clang-format，例如：

```
string ext =……;
//如果是 clang-format 可识别后缀
if (ext==".cpp"||ext==".hpp"||ext==".c"||ext==".h"||ext==".cc")
{
    string p("clang-format -i -style=llvm ");
    p.append(_M_filedir);
    system(p.c_str()); //命令行格式化
}
```

代码忽略了 ext 的求解过程，只需要知道这是后缀即可。如果是可以识别的后缀那么就加入对应的 format 程序，这是一个可扩展的内容，甚至 clang-format 也不仅仅可以格式化 C/C++的代码，还可以格式化 C#，java，Obj-C，Obj-C++，在 github 上可以看到很多配置文件。如果要求加入 python，那么只需要一个 else if 就可以加入对应的程序，比如 yapf。

2.6. 类的关系

2.2~2.5 是在检查之前完成的，其实功能并不完整，结构也不是很满意，因为需要一个暴露的函数管理两个类的内容，这会导致加入新功能时代码复杂度变得不可控制，因此需要重新整理类与类之间的关系。这部分的代码可以参考 1.0 版本的实现，5.27 完成，使用 git 倒带回去即可。

问题的关键在于一个裸露的函数 link，他需要过多的 friend 函数声明，如果将其包装成一个类就能更加好，在这个类中调用一些函数就能完成绝大多数的功能，同时保持对外界的封锁是最好不过。

因此我选择了设置一个顶层类，类声明如下：

```
#define _M_cnt_cmds 6
class Compare
{
private:
    string _M_dir_merge;                //合并位置
    bool _M_need_merge;                 //是否需要合并
    bool _M_cmds[_M_cnt_cmds];          //cmd 命令
    Folder *_M_folder[2];               //文件夹
    vector<LCS *> _M_results;           //结果的存放位置
```

```
    vector<string> _M_file_intersection; //同时在A 和B 中的，方便合并，注意和差集的区别

private:
    void _M_link();  //寻找匹配的文件，放入 LCS*的数组
    void _M_merge(); //合并（如果需要的话）

public:
    using size_type = LCS::size_type;

    Compare() {}
    Compare(const string &_A, const string &_B, const bool *_cmds); //不合并的时候接受的参数
    Compare(const string &_A, const string &_B, const string &_C);  //合并时接受的参数

    void print() //输出
    {
        for (LCS *const result : _M_results)
            result->print_diff();
    }
};
```

很显然，最小化操作数量是比较好的解决方法，同时每当打开一个窗口的时候则等价于新建一个这个类的实例化对象。

与此同时，在类的实现中发现在_M_link()方法中涉及到非常多的 folder 的内容，因此选择了友元类来方便访问，即以下内容：

```
public:
    Folder(const string &filedir);
    void print_everything(); //验证用
    friend class Compare;    //友元类，因为发现 Compare 需要大量使用类中内容。
```

到此为止，代码框架应当是这样的：

2.7. 合并功能

实际上，有了之前的框架，合并只需要考虑一些比较简单的内容，而更多的内容则意图复刻代码的你来定夺。

1) 命令行的处理。如果需要文件合并，则必然是一个一个合并，只需要给每一个 LCS 对象传送一个合并地址即可，不论是初始化传送还是合并时传送都是可以的。
2) 合并的实现。在涉及到深层次的文件目录时，可能需要重新生成大量的目录，这是因为单纯的文件流不能在访问不了文件夹的情况下访问目录。这一部分需要使用系统函数 mkdir 系列（因为 mkdir 在 windows 下为_mkdir，在 linux 下位 mkdir(filedir,)）和 access 系列实现一个递归创建文件夹的 smkdir 函数。
3) 合并的文件输出。LCS 算法计算出的差异行号需要减 1 才能作为最终的结果，否则必然会报错。

2.8. 代码的其他细节

在实现过程中，可能还有一些细节上的问题需要处理，比如

首先，这里涉及到了大量的字符串处理，需要进行一些比较“智能”的判断，比如文件路径的末尾没有斜杠，那么可能会导致文件夹产生错误的路径，进而导致一连串的错误。解决这个问题最重要的是产生一个统一的标准，比如没有斜杠则需要自动产生斜杠。

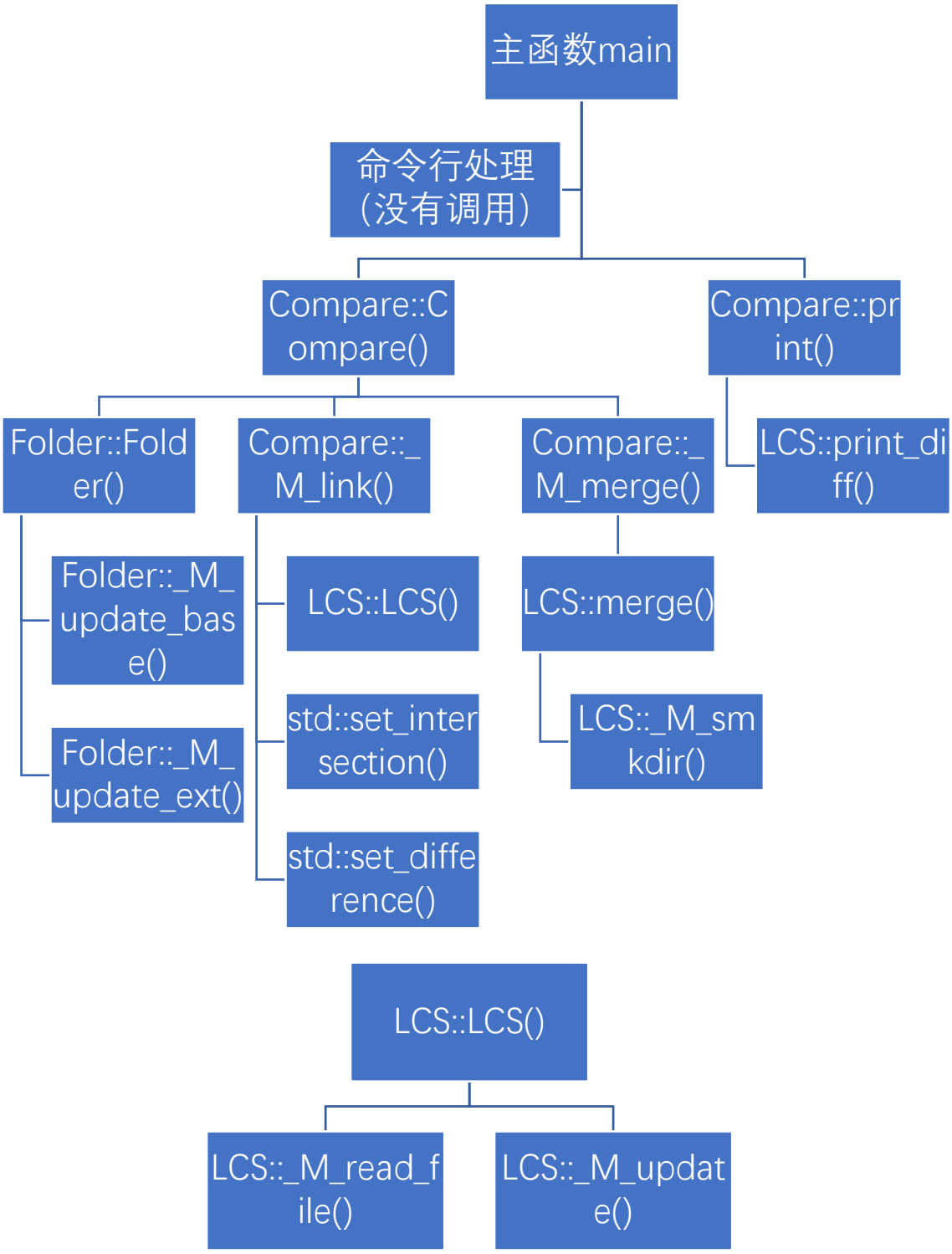
其次，对于文件路径这类需要注意是否需要 1 的问题，一般的解决方案是写一个 tmp 程序，保证正确之后再放进主程序。

最后再次注意 LCS 结果和源文件行号的不一致。

3. 设计的详细信息

3.1. 程序的调用过程

程序调用的过程如下图所示：



其中第二部分的图的 LCS::LCS()和第一张图的是同一个节点。所有的方法都没

有写参数。

3.2. 类 Compare 的功能与实现

类 Compare 的主要功能是位后续两个类提供包装和外部接口的调用。

类的声明如下：

```
#define _M_cnt_cmds 6
class Compare
{
private:
    string _M_dir_merge;                //合并位置
    bool _M_need_merge;                 //是否需要合并
    bool _M_cmds[_M_cnt_cmds];          //cmd 命令
    Folder *_M_folder[2];               //文件夹
    vector<LCS *> _M_results;           //结果的存放位置
    vector<string> _M_file_intersection; //同时在A 和B 中的，方便合并，注意和差集的区别

private:
    void _M_link();  //寻找匹配的文件，放入 LCS*的数组
    void _M_merge(); //合并（如果需要的话）

public:
    Compare() {}
    Compare(const string &_A, const string &_B, const bool *_cmds); //不合并的时候接受的参数
    Compare(const string &_A, const string &_B, const string &_C);  //合并时接受的参数
    void print(); //输出
};
#include "Compare.impl.hpp"
#endif
```

其中每一个方法的实现如下：

1) _M_print()：这个方法的实现方法是遍历 LCS*的结果数组，调用每一个的 print 函数，目的是输出每一个文件对的差异。
2) 构造函数 Compare()：有三个同名的构造函数。

第一个无参的什么都不做。

第二个输入 cmd 的则仅比较文件（夹），会进行获得文件夹参数输出，两个文件夹之间文件的匹配，如果能匹配到则讲这一个文件对压入 LCS 的 vector 开始比较。

第三个要求三个文件路径作为参数，不接受 cmd 输入，用于合并文件（夹）。不接受 cmd 输入的原因主要是尽量保证文件的真实，不会因为比较结果而丢失任何一行。显然 cmd 会在构造函数中的比较之前初始化为 0。合并文件会到第三个路径。

3) 私有方法_M_link()：这个方法在文件夹打开之后被构造函数调用，用途是匹配两个文件夹的共有文件，并且将不共有的文件检查并报告。其中共有文件的含义是指相对于输入的两个文件夹的相对路径完全一样，而不是文件名一样。所有的共有文件都会被压入 LCS 的 vector 中以开始比较。

4) 私有方法_M_merge()：这个方法在文件比较完成之后开始调用，而且仅在第三个构造函数才会被用到。这个方法主要的任务是为 LCS 的 vector 中每一个 LCS 调用 merge 方法，传入的参数有两类。

   如果确认为单文件比较，则认为是输出到特定文件，反之则认为输出到文件夹下的同名文件。

3.3. 类 Folder 的功能与实现

类 Folder 的主要功能是寻找一个文件夹的下属文件或者文件本身，但是不打开文件的内容。被 Compare 调用，而且设置了 Compare 的友元。

类 Folder 的声明如下：

```
class Folder
{
private:
    string _M_base_dir;
    vector<string> _M_ext_dirs;
    //判定是否为单个文件，这样可以防止单文件对比单文件的不匹配
    bool _M_only_file;
    //私有化更新，防止外部随意调用
    void _M_update_base(const string &filedir);
    void _M_update_ext(const string &dir);
    Folder() {} //私有化构造函数，防止默认构造

public:
    Folder(const string &filedir);
    void print_everything(); //验证用
    friend class Compare;    //友元类，因为发现 Compare 需要大量使用类中内容。
};
```

其中每一个方法的实现如下：

1) 构造函数 Folder()：主要用于在 Compare 创建文件（夹）路径的时候进行文件夹所含文件的获取，内部会进行_M_update_base，并根据情况进行_M_update_ext。私有化无参构造函数 Folder()用于防止被因为不期望的方式新建一个 Folder 对象。
2) _M_update_base()方法用于更新文件根路径，有两种情况。如果输入为文件夹路径时，将其调整为适当的形式，方便后续方法寻找新的文件。如果输入为文件路径时，会寻找它所处的文件夹作为根路径，文件名本身作为唯一的扩展路径，并且阻止寻找其他的文件（通过更新类成员_M_only_file 实现）。
3) _M_update_ext()方法用于更新文件夹下所有的文件，在只有一个文件的时候不会被构造函数 Folder()调用。这是一个递归的方法，通过_A_ARCH 来定位文件，通过_A_SUBDIR 来定位文件夹，并进行更深层次的递归。定位文件和文件夹的标准可以通过代码来修改。注意.和..两个文件夹分别代表本文件夹和上一层，如果不进行特别处理会进入死递归，无法停止。

以上两个方法都大量使用了文件句柄，用于寻找文件和能否访问。

3.4. 类 LCS 的功能与实现

类 LCS 的主要功能是寻找一个文件夹的下属文件或者文件本身，但是不打开文件的内容。被 Compare 调用，没有设置了 Compare 的友元。

类 LCS 的声明如下：

```
class LCS
{
private:
    //子类，方便存储结果
    struct LCS_base
    {
        size_type _M_res, _M_prev_x, _M_prev_y;
        LCS_base(value_type _res = 0, size_type _prev_x = 0, size_type
_prev_y = 0)
        {
            _M_res = _res;
            _M_prev_x = _prev_x;
            _M_prev_y = _prev_y;
        }
    };
```

```
    //static 哈希方法，仿函数方式调用 hash
    static const std::hash<string> _M_diff_hasher;

private:
    string _M_filedir[2];                          //两个文件的路径
    vector<value_type> _M_hashline[2];              //源文件经过格式化处理之
后的 hash 值
    vector<string> _M_line[2];                     //源文件的内容
    vector<pair<size_type, size_type>> _M_same_line; //比较结果
    bool *_M_cmds;                                 //比较之前的格式化方式

private:
    void _M_update();                          //LCS 更新
    void _M_read_file(const int &file_switch); //读取文件
    void _M_smkdir(string _newdir);            //创建文件夹
    LCS() {}

public:
    LCS(const string &_filedir1, const string &_filedir2, bool *_cmds);
    void print_same();                                                  //输出相
同的行，调试功能
    void print_diff();                                                  //输出不
同的行，有功能
    void swapfile();                                                    //交换文
件，这会交换 LCS 结果
    void modifyfile(const string &newfiledir, const int &whichfile); //更
改文件路径，这会重新计算 LCS
    void merge(const string &_merge2where);                             //合并
};
```

其中每一个方法的实现如下：

1) 构造函数 LCS()：传入两个文件路径和一系列命令行控制参数，先将 cmd 更新到_M_cmds 上，不占用额外空间，和 Compare 绑定在一起。之后会进行文件的更新，调用_M_read_file 打开文件，接着直接将两个文件的读取结果进行比较，是否需要合并需要额外从 Compare 调用的内容进行单独处理。私有化无参构造函数 LCS()用于防止被因为不期望的方式新建一个 LCS 对象。
2) 私有方法_M_read_file()用于在获得文件路径之后以行为单位读取对应的文件，并根据获取的 cmd 参数来调整放入比较的内容。原本文件的内容会被放入_M_line。只有参数—format 才会修改原本的文件，其他的不会修改文件，只会影响读取以后每一行的 hash 值。
3) 私有方法_M_update()用于比较两个文件，核心算法是最开始提出的 LCS。比较

结果会放入_M_same_line，内容是算法结果中相同的行号（从 1 开始）。

4) 私有方法_M_smkdir()用于生成文件夹，主要用于合并文件时防止文件夹不存在导致无法使用流文件打开文件。这个方法使用了迭代，每当遇到一个斜杠，就将其重设为'\0'，之后测试能否打开，如果不能打开就新建文件夹，直到最后一个字符的最后，如果依旧不能访问，还是要创建一个文件夹。
5) 公有方法 swapfile()会交换两个文件，这会导致 LCS 结果中每一对相同内容的行号发生交换，同时交换两个文件的信息。
6) 公有方法 modifyfile()会修改某一个文件，之后重新读取并且重新计算 LCS。
7) 公有方法 print_diff()会根据 LCS 的比较结果进行输出，当 LCS[i]中的每一项都与 LCS[i-1]的对应项多 1 时，说明这之间没有差异。如果不是则需要根据情况进行输出。

   如果某一边差值大于 1 则是添加和删除的某一种，如果差值大于 1 的出现在左边那么认为被删除了，反之则认为是新增行。如果两边的插值都是大于 1 的则认为是修改。此函数的实现还针对单数和复数进行了判断，实际可以不用写这么复杂(line(s)来同时代表两个)。
8) 公有方法 merge()需要一个文件路径来代表输出路径，由于 Compare 的调用，可以假定传入参数必然是一个文件名。输出主要思想依靠与 print_diff，唯一的区别是将删除和新增识别成附带对应下一行的修改。

3.5. 主函数 main 的功能与实现

主函数 main 主要负责的功能是获取命令行输入，并处理命令行，调用 Compare 的初始化开始运算以及输出差异。

处理命令行的方式就是把所有的命令行参数放在一起，然后将已知的命令行参数集合一个一个从中匹配。由于数量不多，因此不需要排序或者 set 来提升速度。程序接受任意个输入，其中-开头的命令行参数只会接受指定的，其他都会被忽略。不以-开头的命令行参数接受前三个，如果少于两个则会要求你输入文件（夹）路径补全到两个路径，超过三个的则会被忽略。

4. 测试数据及其结果

4.1. 测试数据

为了测试文件，我将某一个版本的这个项目复制了两份到这个文件夹中，分别取名叫 test1 和 test2。初始状态下，test1 全部格式化为四个空格作为缩进格式，test2 则设置为两个。与此同时，test2 还有一个 diff.1.cpp 是 test1 没有的。代码格式化依旧使用 clang-format，套用格式为 visual studio。

之后会在控制台依次进行如下命令以测试功能。

```
homework-part3\diff.exe test2\diff.1.cpp test2\diff.cpp
homework-part3\diff.exe test2\diff.1.cpp test2\diff.cpp --ignore-blank-
lines
homework-part3\diff.exe test1\diff.cpp test2\diff.cpp --ignore-space
homework-part3\diff.exe test1\diff.cpp test2\diff.cpp --ignore-space --
ignore-blank-lines
homework-part3\diff.exe test1 test2 --format
homework-part3\diff.exe test1\diff.cpp test2\diff.1.cpp
homework-part3\diff.exe test1\diff.cpp test2\diff.1.cpp --ignore-space
homework-part3\diff.exe test1\diff.cpp test2\diff.1.cpp --ignore-all-
space --ignore-blank-lines
```

结果如下：

```
PS D:\tmp> homework-part3\diff.exe test2\diff.1.cpp test2\diff.cpp
test2\diff.cpp
Compare test2\diff.1.cpp with test2\diff.cpp:
    add line in B:39
    add line in B:42
PS D:\tmp> homework-part3\diff.exe test2\diff.1.cpp test2\diff.cpp --
ignore-blank-lines
test2\diff.1.cpp
test2\diff.cpp
Compare test2\diff.1.cpp with test2\diff.cpp:
    Nothing different.
test1\diff.cpp
test2\diff.cpp
Compare test1\diff.cpp with test2\diff.cpp:
    add line in B:39
    add line in B:42

PS D:\tmp> homework-part3\diff.exe test1\diff.cpp test2\diff.cpp --
ignore-space --ignore-blank-linestest1\diff.cpptest2\diff.cppCompare
test1\diff.cpp with test2\diff.cpp:
```

```
    Nothing different.

PS D:\tmp> homework-part3\diff.exe test1 test2 --format
test1
test2
File not in A but in B: test2\diff.1.cpp
Compare test1\diff.cpp with test2\diff.cpp:
    add line in B:31
    add line in B:34
Compare test1\diff.exe with test2\diff.exe:
    Nothing different.
Compare test1\hpp\Folder.hpp with test2\hpp\Folder.hpp:
    Nothing different.
Compare test1\hpp\Folder.impl.hpp with test2\hpp\Folder.impl.hpp:
    Nothing different.
Compare test1\hpp\LCS.hpp with test2\hpp\LCS.hpp:
    Nothing different.
Compare test1\hpp\LCS.impl.hpp with test2\hpp\LCS.impl.hpp:
    Nothing different.

PS D:\tmp> homework-part3\diff.exe test1\diff.cpp test2\diff.1.cpp
test1\diff.cpp
test2\diff.1.cpp
Compare test1\diff.cpp with test2\diff.1.cpp:
    modify line in A:6 <======> lines in B:6 ~ 7
    modify line in A:9 <======> lines in B:10 ~ 11
    modify line in A:16 <======> lines in B:18 ~ 20
    modify line in A:19 <======> lines in B:23 ~ 25
    modify line in A:23 <======> lines in B:29 ~ 31

PS D:\tmp> homework-part3\diff.exe test1\diff.cpp test2\diff.1.cpp --
ignore-space
test1\diff.cpp
test2\diff.1.cpp
Compare test1\diff.cpp with test2\diff.1.cpp:
    modify line in A:6 <======> lines in B:6 ~ 7
    modify line in A:9 <======> lines in B:10 ~ 11
    modify line in A:16 <======> lines in B:18 ~ 20
    modify line in A:19 <======> lines in B:23 ~ 25
    modify line in A:23 <======> lines in B:29 ~ 31

PS D:\tmp> homework-part3\diff.exe test1\diff.cpp test2\diff.1.cpp --
ignore-space
test1\diff.cpp
test2\diff.1.cpp
```

```
Compare test1\diff.cpp with test2\diff.1.cpp:
    modify line in A:6 <======> lines in B:6 ~ 7
    modify line in A:9 <======> lines in B:10 ~ 11
    modify line in A:16 <======> lines in B:18 ~ 20
    modify line in A:19 <======> lines in B:23 ~ 25
    modify line in A:23 <======> lines in B:29 ~ 31

PS D:\tmp> homework-part3\diff.exe test1\diff.cpp test2\diff.1.cpp --
ignore-all-space --ignore-blank-linestest1\diff.cpp
test2\diff.1.cpp
Compare test1\diff.cpp with test2\diff.1.cpp:
    modify line in A:6 <======> lines in B:6 ~ 7
    modify line in A:9 <======> lines in B:10 ~ 11
    modify line in A:16 <======> lines in B:18 ~ 20
    modify line in A:19 <======> lines in B:23 ~ 25
    modify line in A:23 <======> lines in B:29 ~ 31
PS D:\tmp> homework-part3\diff.exe test1 test2 --ignore-all-space --
ignore-blank-lines test3test1
test2
File not in A but in B: test2\diff.1.cpp
Compare test1\diff.cpp with test2\diff.cpp:
    add line in B:31
    add line in B:34
Compare test1\diff.exe with test2\diff.exe:
    Nothing different.
Compare test1\hpp\Folder.hpp with test2\hpp\Folder.hpp:
    Nothing different.
Compare test1\hpp\Folder.impl.hpp with test2\hpp\Folder.impl.hpp:
    Nothing different.
Compare test1\hpp\LCS.hpp with test2\hpp\LCS.hpp:
    Nothing different.
Compare test1\hpp\LCS.impl.hpp with test2\hpp\LCS.impl.hpp:
    Nothing different.
```

对于这个测试结果，可能因为内容太多而看不懂，因此我把测试结果和原本代码一起打包发出。程序如预想的正确执行并输出了结果。

5. 试验结束之后的思考

5.1. 程序暴露出的问题

程序虽然能正常执行，但是这个程序依旧有一些需要优化的用户体验。

1) 合并没有提供选项，如果能提供更多选项，那么可以更好的定制个人体验的功能。比如针对两边不是同时拥有的文件的处理方式应当由用户来定夺，甚至每一个文件都应当用户定夺。
2) 格式化影响到了文件本身。这可能不是用户希望看到的，虽然可以在 format 所处的文件夹下自行提供自己的.clang-format 文件来修改格式化内容，不过加入更多的选项总是更好的。
3) 命令行操作依旧太少。实际上因为时间受限，程序的命令行选项不是很多，甚至正则表达式都没有提供。除此以外，还有一些输出选项也没有提供，较少的输出也可以优化用户体验。

总而言之，这算是花费了不算很少的时间完成了一个仅仅能称作一个 demo 的程序。

5.2. 实验体会

在这次实验中我收获很多。实验中我的代码分析能力有所提升，稍微了解了 C 的底层如何和 windows 交互，并且熟练使用命令行与编写命令行程序，基本达到了实验目的。

但是这次实验依旧有所不足，主要归因于时间安排过于紧张。虽然看似课程给了我们五周时间完成这一部分的实验，但是实际上扣去上课时间，考试复习（本学期考试从 5.12 开始，截止到 6.2 已经有七八场考试，时间间隔大约只有两三天左右）之间的时间已经很难找到很多时间，除非熬夜到两三点。

如果有充足的时间，下一步的计划主要是：重新设计合并功能，将其从比较程序分离，但是使用的内容基本一样；设计一个界面，并尝试将其与设计的类完全贴合在一起。

这次实验给我比较多的体会，涉足了不是很常用的开发领域，希望这样的实验能越来越多。