2023年春季学期

软件设计实践

C++多文件工程

谢 涛　　马 郓

回顾：程序设计语言

- **机器语言**：由二进制的机器指令构成，可直接执行。
- **汇编语言**：为便于记忆，将机器指令用助记符和符号写成的程序
- **高级语言**：用易于理解的形式，如英文单词、数学公式，按严格的语法规则和一定的逻辑关系写出的程序集合

地址		内容		汇编程序
0000	0000	1011	0000	MOV AL , n
0000	0001	0000	0111	n = 7
0000	0010	0000	0100	ADD AL , n
0000	0011	0000	1010	n = 10
0000	0100	1111	0100	HLT
0000	0101			

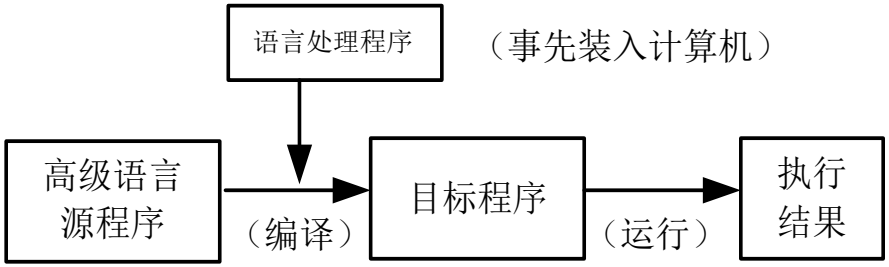
```cpp
#include <iostream>
using namespace std;
void printSum(int x, int y) {
    cout << x + y << endl;
}
int answer{42};
int main() {
    int a, b;
    cin >> a >> b;
    printSum(a, b);
    cout << answer << endl;
}
```

高级语言

回顾：语言处理程序

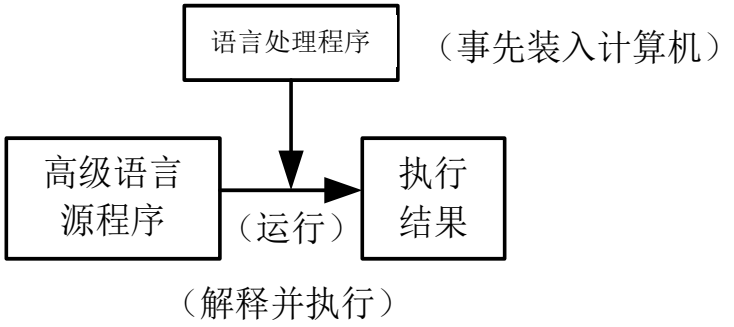
❑ 语言处理程序

- ➢ 源程序：用汇编语言或高级语言各自规定的符号和语法规则编写的程序
- ➢ 目标程序：将计算机源程序翻译成机器语言后的程序
- ➢ 源程序翻译成目标程序有两种：**编译**方式和**解释**方式
  - **优缺点？**



(a) 编译过程示意图

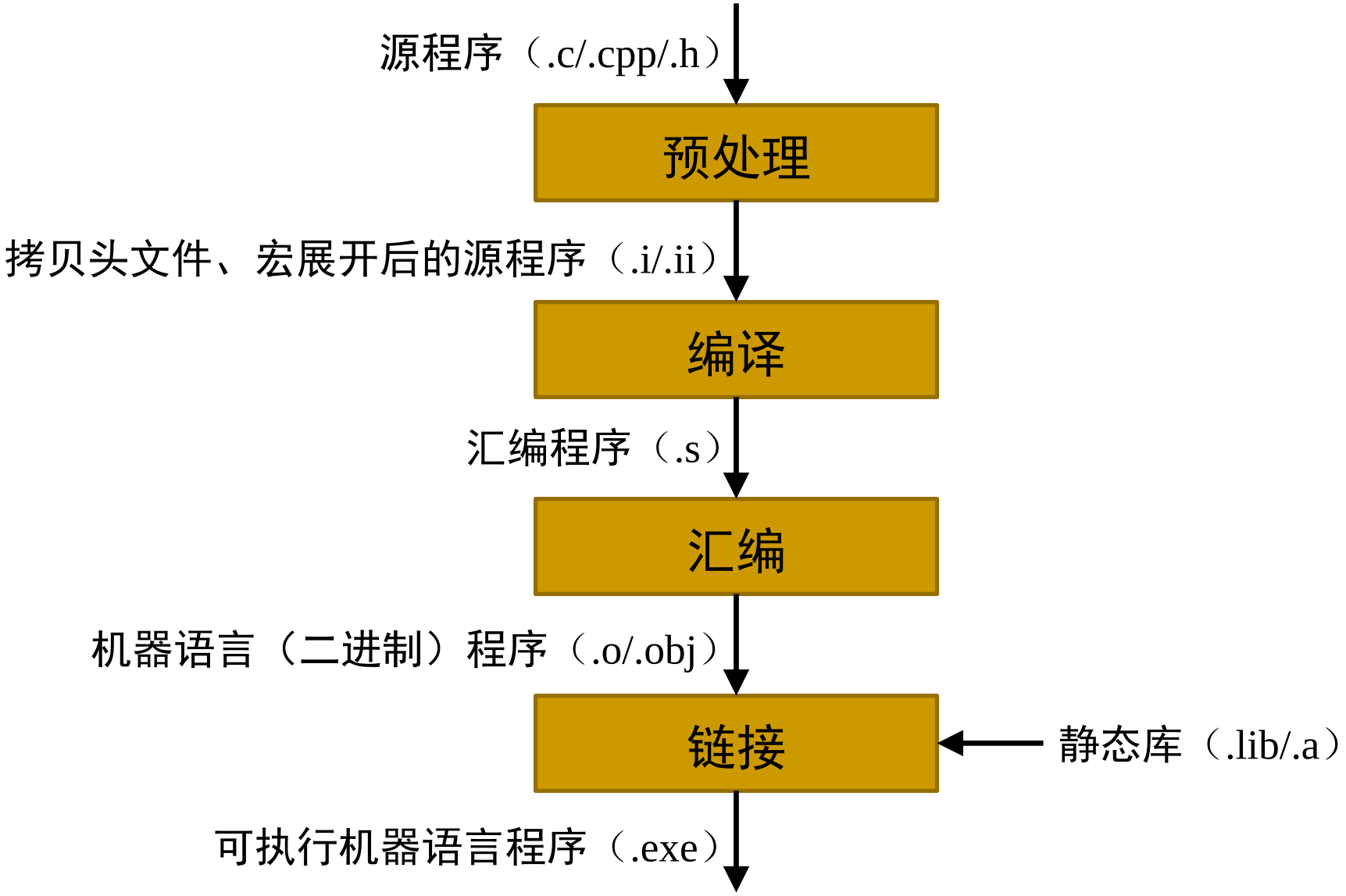
C/C++



(b) 解释过程示意图

Python

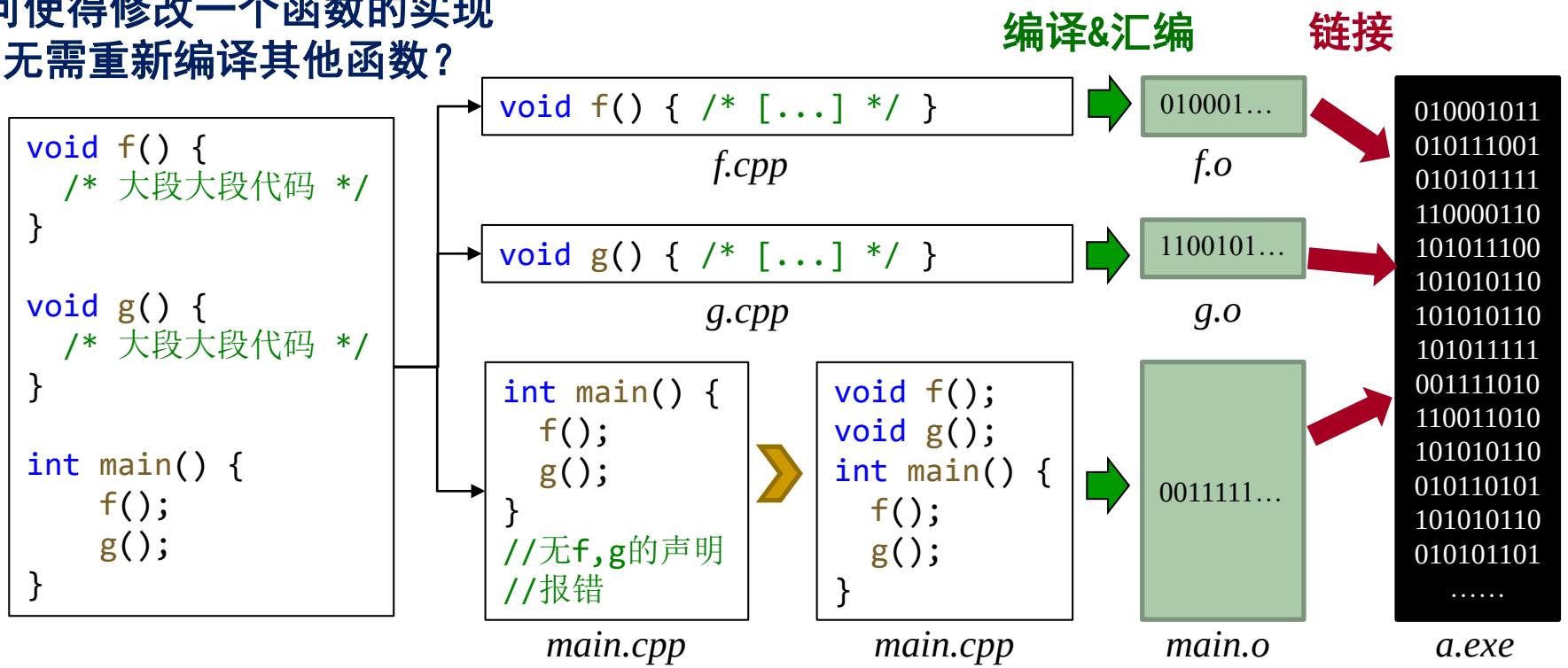
C++的“翻译”过程



为什么要分成“编译”和“链接”两个步骤？

拆分单个代码文件

如何使得修改一个函数的实现
而无需重新编译其他函数？



- 拆分为多个代码文件后，当修改某个代码文件时，只需重新编译改过的文件并重新链接即可，整体的翻译效率得到提升

本讲概览

- C++多代码文件的组织
  - 条件编译
  - 命名空间
  - 单一定义原则
- 命令行
  - 命令行参数
  - 命令行编译
  - GNU Make与Cmake
- 输入输出
  - 文件读写

C++项目的代码文件分类

- 一个完整的c++项目常常是由多个代码文件组成，根据后缀名不同，大致分为两类
  - .h 头文件（Header file）一般只存放类型的定义和符号的声明（在不涉及模板的情形下）
    - 由于头文件只含声明，所以单独编译头文件不会产生任何二进制指令（只有定义才可以让编译产生结果）
    - 头文件的作用仅在于被 .cpp 文件所包含
  - .cpp 实现文件（Implementation file）存放对应头文件声明的定义
    - 为了引入相关类型的定义，一般需要将同名头文件也包含进来
    - 每一个实现文件一般都对应着一个翻译单元，这些翻译单元会在最后链接为整个执行程序
    - C++项目一般会包含一个带有main()函数的特殊实现文件，作为程序的执行入口，该实现文件一般只会包含必需的头文件

C++项目的代码文件分类

❑ 示例

```cpp
#include <iostream>
using namespace std;

class Stuent{
public:
    char *name;
    int age;
    float score;

    void say(){
        cout << name << " " << age << endl;
    }
};
int main()
{
    Stuent *pStu = new Stuent;
    pStu -> name = "xiaoming";
    pStu -> age = 25;
    pStu -> score = 92.5f;
    pStu -> say();

    delete pStu;
    return 0;
}
```

main.cpp

```cpp
class Student{
public:
    const char *name;
    int age;
    float score;
    void say();
};
```

student.h

```cpp
#include <iostream>
#include "student.h"
void Student::say(){
    std::cout << name << " " << age;
}
```

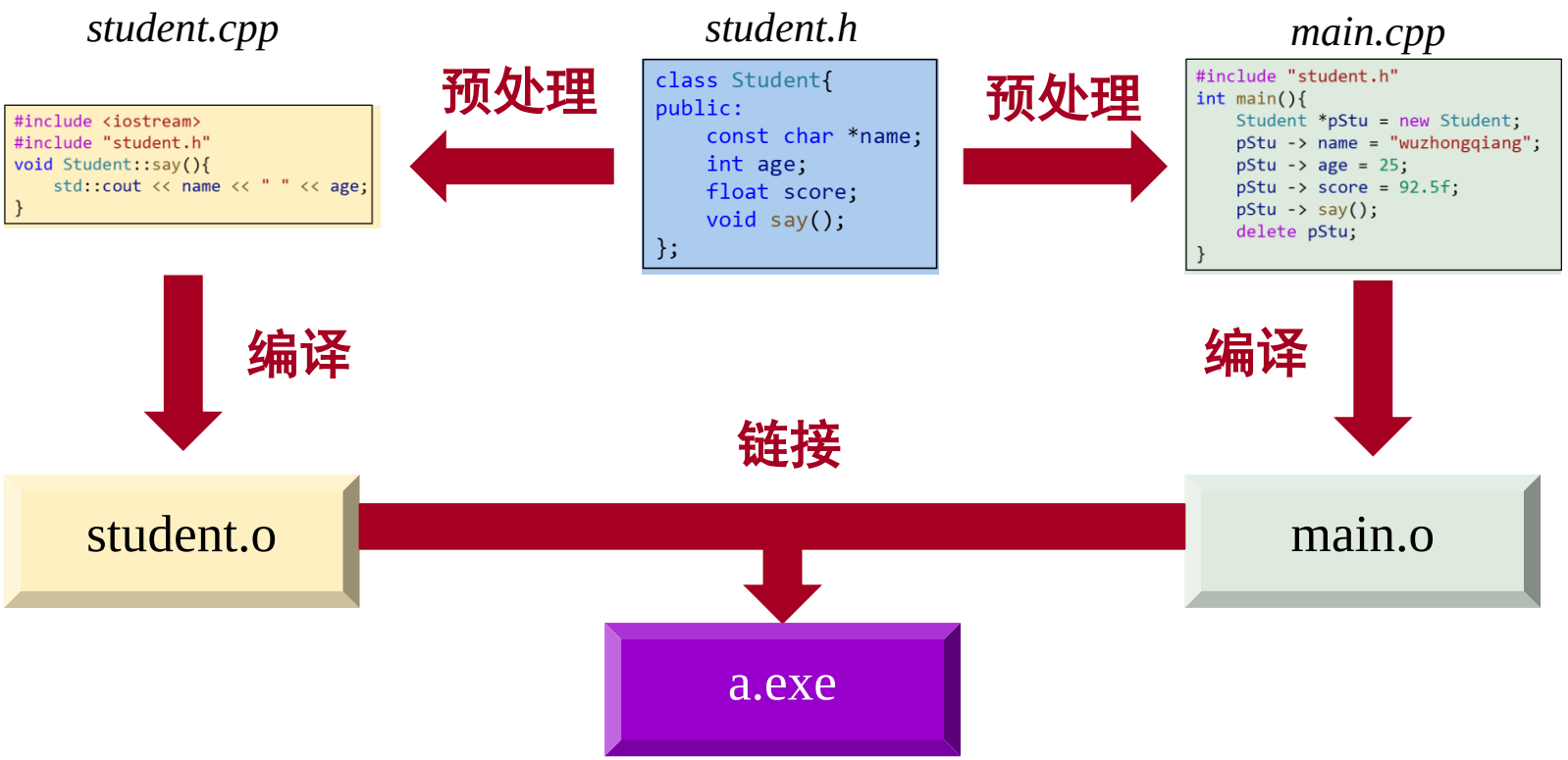
student.cpp

```cpp
#include "student.h"
int main(){
    Student *pStu = new Student;
    pStu -> name = "wuzhongqiang";
    pStu -> age = 25;
    pStu -> score = 92.5f;
    pStu -> say();
    delete pStu;
}
```

main.cpp

预处理指令

- cpp文件中以 # 开头且以换行符为结尾的语句是预处理指令，在编译之前完成特定的替换任务



预处理指令

- #include预处理指令
  - 把某个源文件直接插入进指令所在的位置
    - 效果类似于文件内容拷贝
  - 预处理时不断递归地展开 #include 指令，直至不存在 #include 指令
- #include <头文件名>
  - 优先在系统库文件中查找
  - <iostream> <cmath> 等库就是位于系统库路径的名为iostream、cmath等的文件
- #include "文件名"
  - 优先在当前源文件所在的路径中查找

条件编译

❑ 多次#include头文件导致重复的问题

```
class Student {
    //......
};
```

student.h

```
#include "student.h"
class School {
    //......
private:
    Student stu[50];
};
```

school.h

```
#include "student.h"
#include "school.h"
int main() {
    //......
    return 0;
}
```

main.cpp

运行此项目会发现，编译器报“Student 类型重定义”错误

解决方案：

利用预处理指令

- #define
- #if
- #elif
- #else
- #endif
- #ifdef
- #ifndef

使编译器选择性地编译某一部分而不编译其它部分

注：该过程实际在预处理阶段完成，因此相关条件需要在编译时能确定

条件编译

- 若常量表达式为非0值，则编译代码1 但不编译代码2，#else可以省略
  - 如果连续多个 #else #if ， 可以使用 #elif 来简化

```
#if 常量表达式
代码1
#else
代码2
#endif
```

```
#if __cplusplus >= 201103L
int* ptr = nullptr;
#else
int* ptr = NULL;
#endif
```

- defined 运算符可以返回当前环境是否定义了某个宏
  - 若宏存在，则值为 1 ；若宏不存在，则值为 0

```
#if defined ONLINE_JUDGE
cout <<"welcome";
#endif
```

  - #ifdef 等价于 #if defined ， #ifndef 等价于 #if !defined

```
#ifdef 宏名
```

```
#ifndef 宏名
```

条件编译

示例

```
class Student {
    //......
};
```

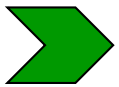
student.h

```
#include "student.h"
class School {
    //......
private:
    Student stu[50];
};
```

school.h

```
#include "student.h"
#include "school.h"
int main() {
    //......
    return 0;
}
```

main.cpp



```
#ifndef STUDENT_H
#define STUDENT_H
class Student{
    //....
};
#endif
```

student.h

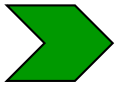
虽然该项目main.cpp文件中仍#include了2次 "student.h"，但鉴于 STUDENT_H 宏只能定义一次，所以 Student 类也仅会定义一次

命名空间

❑ 名字冲突问题

```
……
constexpr float PI{3.14159f};
……
```

a.h

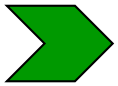


```
namespace libA {
  constexpr float PI{3.14159f};
}
```

a.h

```
……
constexpr double PI{3.14159265359};
……
```

b.h

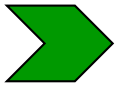


```
namespace libB {
   constexpr double PI{3.14159265359};
}
```

b.h

```
#include <iostream>
#include "a.h"
#include "b.h"
using namespace std;
int main() {
    cout << PI; //编译错误
}
```

main.cpp



```
#include <iostream>
#include "a.h"
#include "b.h"
using namespace std;
int main() {
    cout << libA::PI; //a.h中声明的PI
    cout << libB::PI; //b.h中声明的PI
    PI; //编译错误，未指明命名空间
}
```

main.cpp

命名空间

```
namespace 命名空间名{
    程序片段
}
```

- 自定义命名空间
  - 命名空间一般在.h头文件中
  - 当类的声明位于指定的命名空间中时，如果要在类的外部实现其成员函数，需同时注明所在命名空间名和类名
  - 不同的头文件中可以使用不同的命名空间， 也可以使用相同的命名空间， 但后者的前提是位于该命名空间中的成员必须保证互不相同
- 作用域解析运算符::
  - 基本用法 命名空间名::名字
  - using namespace 命名空间名；using 命名空间名::名字;

单一定义原则

- 声明与定义
  - 声明：仅仅告诉编译器该符号存在，至于该符号具体含义，等链接的时候才能知道
  - 定义：某个符号完整的描述清楚， 是变量还是函数， 变量类型以及变量值是多少，函数的返回值是什么等
- 单一定义原则（One Definition Rule, ODR）是一系列规则，规定了 C++ 如何处理声明和定义，比较重要的包括：
  - 一个翻译单元中，允许出现变量、函数、类型和模板的多次声明，但最多只允许出现一次定义
  - 一个翻译单元中，如果 ODR-使用了（即需要取其地址的行为）一个符号，那么它最少要出现一次定义
  - 整个程序中，非内联的符号最多只允许出现一次定义

单一定义原则

- 多个文件涉及同一变量名的处理
  - 多个文件共享全局非只读变量
    - 在一个文件中定义，其他文件中用extern 声明
  - 多个文件共享全局只读变量
    - 在一个文件中用extern const 定义，其他文件中用extern const 声明
  - 多个文件各自使用全局只读变量
    - const默认表示变量被各文件分别管理

本讲概览

- C++多代码文件的组织
  - 条件编译
  - 命名空间
  - 单一定义原则
- 命令行
  - 命令行参数
  - 命令行编译
  - GNU Make与Cmake
- 输入输出
  - 文件读写

命令行

- 当前操作系统大多默认以图形界面（Graphical User Interface，GUI）作为人机交互的方式
- 早期的计算机，使用者只能在一个由纯文本组成的界面中，使用键盘向其中键入“命令”，然后得到计算机显示的结果，称为命令行界面（Command-Line Interface，CLI）
- 使用终端模拟器（Terminal Emulator）在GUI上使用CLI
  - 终端模拟器启动时会打开一个称为Shell的程序，该程序会不断地接受键盘的输入（被称作“命令”），然后分析这个输入是在做什么，然后做一些事情
    - Windows：cmd
    - GNU/Linux：bash
    - macOS：zsh

❑ 命令的格式

- ➢ 程序名[空格]参数1 [空格]参数2[空格]...
- ➢ 每一个命令按空格分成若干段
  - 第一段称作“程序名”，其指定了一个程序，Shell会在 CLI 中启动这个程序，然后将输入输出的控制权交给它
  - 剩下的部分称为命令的“参数”，这些参数将在程序启动时通过一种特别的方式传入程序，而程序可以通过一些方法获得这些参数

❑ 工作路径和PATH

- ➢ Shell需要在文件系统的一个特定的路径中工作，称为工作路径，一般作为输入提示符来呈现

```
Microsoft Windows [版本 10.0.19044.2728]
(c) Microsoft Corporation。保留所有权利。

C:\Users\mayun>
```

  - 可以通过“cd 路径名”来更换工作路径
  - . 代表当前路径；.. 代表上层路径
- ➢ 操作系统提供了一些常用的命令行程序，为了调用它们的时候不写路径名，因此操作系统为这些路径提供了快捷方式——PATH 环境变量
  - PATH 的程序在执行时只需给出程序名，而不用给出其路径

命令行

C/C++程序利用main函数的参数获取命令行参数

- argc：代表启动程序时，命令行参数的个数。C/C++ 语言规定，可执行程序本身的文件名，也算一个命令行参数，因此argc 的值至少是1
- argv：指针数组，其中的每个元素都是一个char* 类型的指针，该指针指向一个字符串，这个字符串里就存放着命令行参数（argv[0]始终是程序名本身）

```
#include <iostream>
using namespace std;
int main(int argc, char** argv) {
    if (argc == 1) {
        cout << endl;
    } else {
        cout << argv[1] << endl;
    }
}
```

❑ 命令行重定向

- ➢ 重定向输出(>)
  - 将标准输出重定向到out.txt 文件中

```
test.exe > out.txt
```

- ➢ 重定向输入(<)
  - 将in.txt 文件中的内容重定向到标准输入中

```
test.exe < in.txt
```

❑ Windows的PowerShell不支持这种形式的重定向

命令行编译

以g++为例

- 当指定多个翻译单元时，编译器会分别编译这些翻译单元，并在编译完成后自动调用链接器来链接它们，最终将链接的结果转换为可执行文件保存到 ./main.exe 中
- -o 选项指定了编译的结果存放在哪里

```
#ifndef HELLO_H
#define HELLO_H
void sayHello(const char* who);
#endif // HELLO_H
```

hello.h

```
#include "hello.h"
int main() {
  sayHello("world");
}
```

hello.cpp

```
#include "hello.h"
#include <iostream>
using namespace std;
void sayHello(const char* who) {
  cout <<"Hello, "<<who<<endl;
```

main.cpp

g++ ./main.cpp ./hello.cpp -o ./main.exe

命令行编译

g++ 通过文件后缀名来判断其流程的起点

后缀名	g++ 的行为
.cpp .c++ .cc .cxx 等	视为源文件，将预处理作为第一步
.ii	视为预处理后的源文件，将编译作为第一步
.s .S 等	视为汇编文件，将汇编作为第一步
其它（一般为 .o .obj）	视为对象文件，将链接作为第一步

g++通过若干个选项来控制器流程的终点

选项	g++ 的行为
-E	将预处理作为最后一步，得到预处理后的源文件
-S	将编译作为最后一步，得到汇编文件
-c	将汇编作为最后一步，得到对象文件
不含以上选项	将链接作为最后一步，得到可执行文件

命令行编译

示例

- -E 并不通过 -o 选项来控制输出位置，它直接将预处理结果输出到屏幕（stdout）上。所以可使用重定向运算符 > 将其输出保存到想要的位置 ./hello.ii 中

```
g++ ./hello.cpp -E >  ./hello.ii    # 预处理
g++ ./hello.ii  -S -o ./hello.s     # 编译
g++ ./hello.s   -c -o ./hello.o     # 汇编
g++ ./hello.o      -o ./hello.exe   # 链接
/hello.exe                          # 运行
```

GNU Make（简称 Make）是一个经典的构建工具

- “构建”（Build）是指在一个项目中安排编译的顺序、时机，最终得到想要的结果（称为“目标”（Target））
- Make 使用一种称为 Makefile 的文件来指明这些构建方法
  - Makefile 就是文件名为 makefile 的文件，不带后缀名
- Makefile的格式
  - 生成命令前面的空白是一个 Tab 字符，而不是若干个空格

```
目标文件: 依赖文件1  依赖文件2 ...
        生成命令
```

```
main.exe: f.o g.o main.o
        g++ f.o g.o main.o -o main.exe
f.o: f.cpp
        g++ f.cpp -c -o f.o
g.o: g.cpp
        g++ g.cpp -c -o g.o
main.o: main.cpp
        g++ main.cpp -c -o main.o
```

GNU Make

- 在CLI中执行 make 目标文件 完成构建
- Make可以通过每个文件的最后改动时间来确定一个目标是否需要重新生成，而不用再手动判断

- 通过 Make 来管理构建方法是一种非常原始的策略
  - 翻译单元越多，编写 Makefile 的难度就越大

CMake

- CMake可以通过一个名为CMakeLists.txt的脚本来生成Makefile
  - 第一行规定了最低 CMake 版本
  - 第二行指定了项目名称
  - 第三行 add_executable 告诉 CMake，最终要生成一个可执行文件，可执行文件的名字叫 main，它需要从 main.cpp f.cpp g.cpp 三个文件编译链接得到

```
cmake_minimum_required(VERSION 3.18.0)
project(HelloWorld)
add_executable(main main.cpp f.cpp g.cpp)
```

CMakeLists.txt

- 运行cmake命令可根据CMakeLists.txt生成一系列文件，其中包含Makefile
- 再运行make命令来生成可执行文件

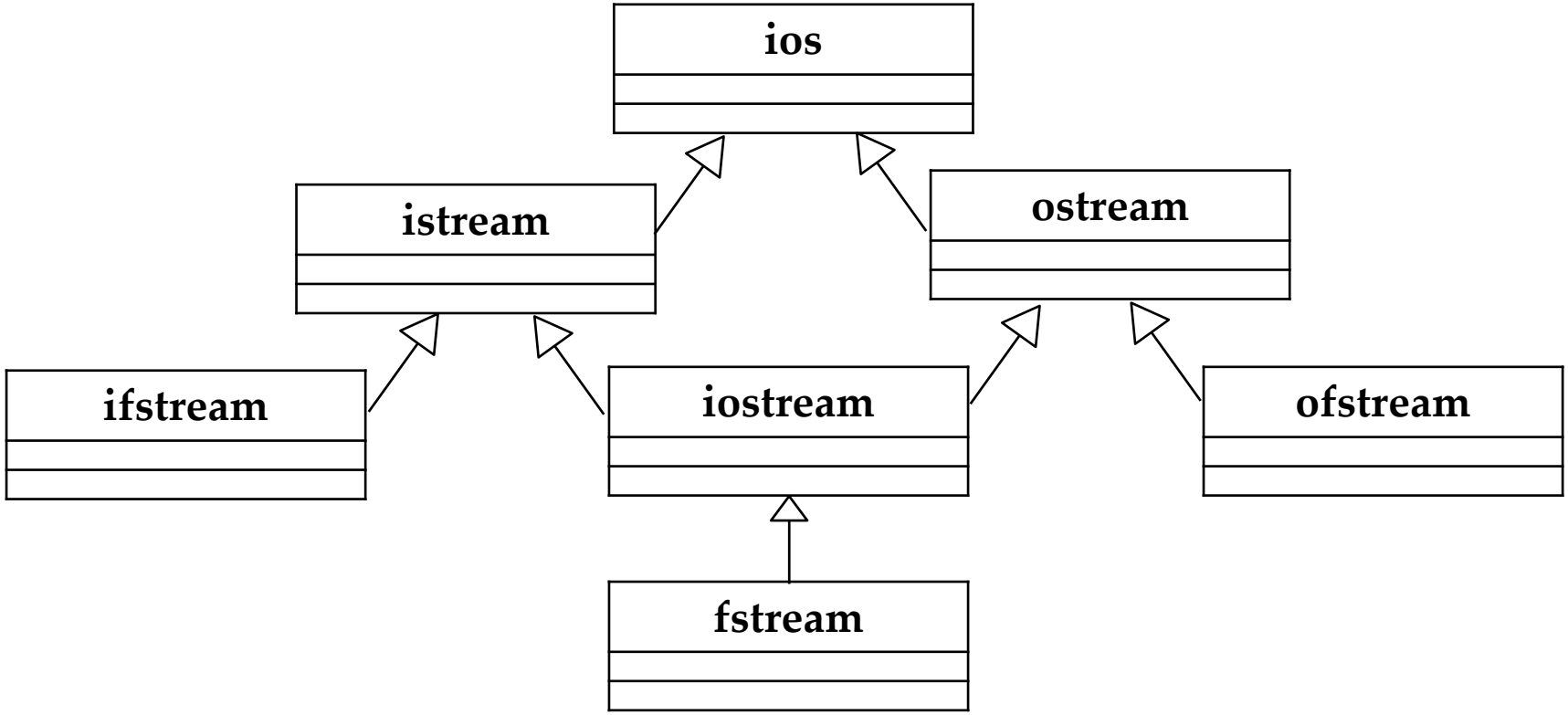
CMakeLists.txt中的常用命令

- add_executable(tgt srcs...)
  - 从指定的源文件编译出一个可执行文件
- target_include_directories(tgt PRIVATE dirs...)
  - 添加包含目录
- target_link_directories(tgt PRIVATE dirs...)
  - 添加库查找目录
- target_link_libraries(tgt PRIVATE libs...)
  - 添加库链接（-l...）
- target_compile_options(tgt PRIVATE opts...)
  - 设置编译选项
- set_target_properties(tgt PROPERTIES prop1 val1 prop2 val2...)
  - 设置其它的编译属性（标准版本、输出路径等等）

本讲概览

- C++多代码文件的组织
  - 条件编译
  - 命名空间
  - 单一定义原则
- 命令行
  - 命令行参数
  - 命令行编译
  - GNU Make与Cmake
- **输入输出**
  - **文件读写**

C++与输入输出相关的类



- istream是用于输入的流类，cin 就是该类的对象
- ostream是用于输出的流类，cout 就是该类的对象
- ifstream是用于从文件读取数据的类
- ofstream是用于向文件写入数据的类
- iostream是既能用于输入，又能用于输出的类
- fstream是既能从文件读取数据，又能向文件写入数据的类

标准流对象

输入流对象

- cin 与标准输入设备相连，对应于标准输入流，用于从键盘读取数据，也可以被重定向为从文件中读取数据

输出流对象

- cout 与标准输出设备相连，对应于标准输出流，用于向屏幕输出数据，也可以被重定向为向文件写入数据
- cerr 与标准错误输出设备相连
- clog 与标准错误输出设备相连
- cerr和clog的区别在于cerr不使用缓冲区，直接向显示器输出信息；而输出到clog中的信息先会被存放在缓冲区，缓冲区满或者刷新时才输出到屏幕

判断输入流结束

- 可以用如下方法判输入流结束
  - 如果从键盘输入，则输入Ctrl+Z代表输入流结束

```
int x;
while (cin >> x) {
  //…
}
```

- 原理
  - istream重载>>运算符的定义

```
istream& operator>>(int &a) {
  //…
  return *this;
}
```

  - 在istream或其基类里重载了explicit operator bool

istream类的成员函数

- istream& getline(char * buf, int bufSize);
  - 从输入流中读取bufSize-1个字符到缓冲区buf, 或读到'\n'为止
- istream& getline(char * buf, intbufSize, char delim);
  - 从输入流中读取bufSize-1个字符到缓冲区buf，或读到delim为止
    - 以上两个函数都会自动在buf中读入数据的结尾添加'\0'
    - '\n'或delim都不会被读入buf, 但会被从输入流中取走
- bool eof();判断输入流是否结束
- int peek();返回下一个字符, 但不从流中去掉
- istream& putback(char c); 将字符ch放回输入流

输入输出重定向

输出重定向

```
#include <iostream>
using namespace std;
int main() {
    int x, y;
    cin >> x >> y;
    freopen("test.txt", "w", stdout); //将标准输出重定向到
test.txt文件
    if (y == 0) //除数为0在屏幕上输出错误信息
        cerr << "error." << endl;
    else
        cout << x / y; //输出结果到test.txt
    return 0;
}
```

输入重定向

```
#include <iostream>
using namespace std;
int main() {
    double f;
    int n;
    freopen("t.txt", "r", stdin); //cin被改为从t.txt中读取数据
    cin >> f >> n;
    cout << f << "," << n << endl;
    return 0;
}
```

流操纵算子

- 使用流操纵算子需要 #include <iomanip>

- 整数流的基数
  - 流操纵算子dec, oct, hex, setbase
- 浮点数的精度
  - 成员函数precision
  - 流操纵算子setprecision
  - 流格式操纵算子setiosflags
- 设置域宽
  - 成员函数setw
  - 流操纵算子width

流操纵算子

❑ 用户自定义流操纵算子

```
ostream &tab(ostream &output) {
    return output << '\t';
}
cout << "aa" << tab << "bb" << endl;
```

输出结果

```
aa      bb
```

❑ 原理

- ➢ iostream 里对 << 进行了重载（成员函数）

```
ostream &operator<<(ostream &(*p)(ostream &));
```

- ➢ hex, dec, oct 等流操纵算子都是函数

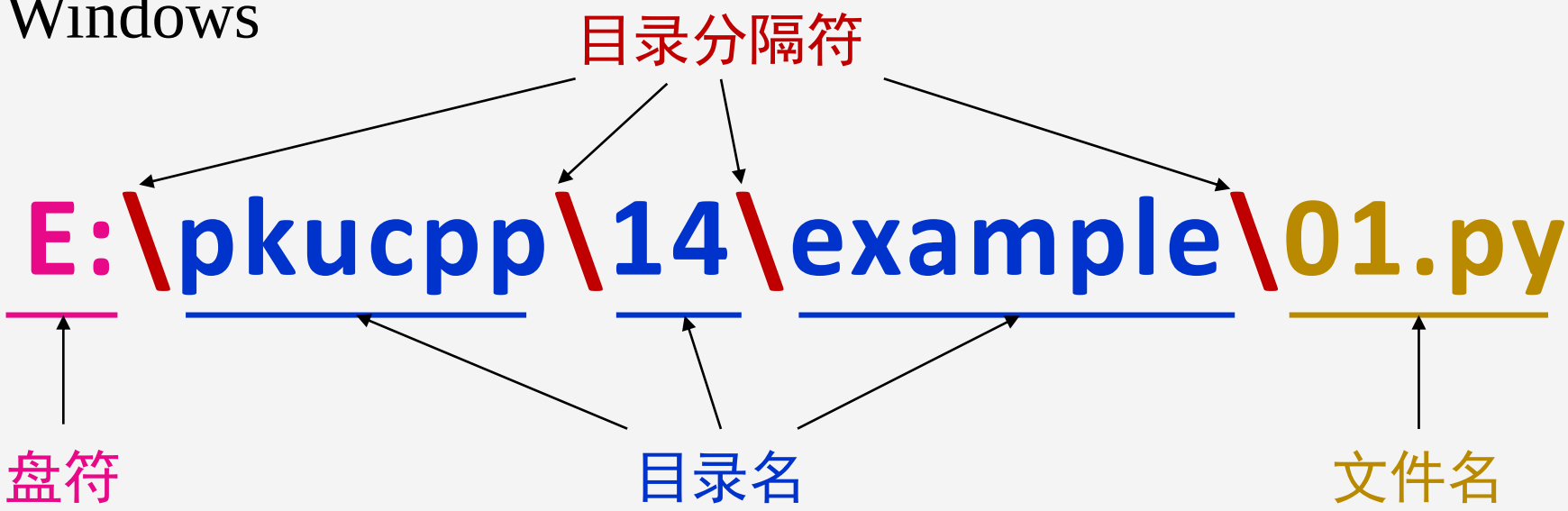
文件的基本概念

- **操作系统以文件为单位对外存进行管理**
  - 当访问磁盘等外存上的数据时，必须先按文件名找到指定的文件，然后再从该文件中读取数据
  - 如果要在外存上存储数据，也必须先建立一个文件，才能向其存入数据
- **文件通过目录来组织和管理**
  - 目录一般采用树状结构，在这种结构中，每个磁盘有一个根目录，它包含若干文件和子目录
  - 目录中可以包含文件或目录

文件的基本概念

- 路径：用于定位一个文件或者目录的字符串
  - 绝对路径：完整的描述文件位置的路径，包含绝对路径的文件名由磁盘驱动器、目录层次和文件名三部分组成

Windows



Mac

/usr/example/01.py

Mac系统没有盘符

文件的基本概念

- **路径：用于定位一个文件或者目录的字符串**
  - 绝对路径：完整的描述文件位置的路径，包含绝对路径的文件名由磁盘驱动器、目录层次和文件名三部分组成
  - 相对路径：目标文件相对于源文件所在目录的路径

文件的基本概念

- **路径：用于定位一个文件或者目录的字符串**
  - 绝对路径：完整的描述文件位置的路径，包含绝对路径的文件名由磁盘驱动器、目录层次和文件名三部分组成
  - 相对路径：目标文件相对于源文件所在目录的路径
    - 当前目录：省略　或　.\
    - 下一级目录：下一级目录名\　或　.\下一级目录名\
    - 上一级目录：..\

文件1的绝对路径	文件2的绝对路径	文件2相当于文件1的相对路径
E:\pkucpp\13\01.cpp	E:\pkucpp\13\d.txt	d.txt 或 .\d.txt
	E:\pkucpp\13\data\d.txt	data\d.txt 或 .\data\d.txt
	E:\pkucpp\d.txt	..\d.txt
	E:\d.txt	..\..\d.txt
	E:\Data\d.txt	..\..\Data\d.txt

文件和流

- 将所有记录顺序地写入一个文件，称为顺序文件
- 可以将顺序文件看作一个有限字符构成的顺序字符流，然后像对cin, cout 一样的读写

创建文件

```
#include <fstream> // 包含头文件
ofstream outFile("clients.dat", ios::out|ios::binary); //打开文件
```

- 创建文件
  - ofstream是fstream中定义的类
  - outFile是程序定义的ofstream类的对象
  - “clients.dat” 是将要建立的文件的路径
  - ios::out 是打开并建立文件的选项
    - ios::out输出到文件，删除原有内容
    - ios::app 输出到文件，保留原有内容，总是在尾部添加
    - ios::ate 输出到文件, 保留原有内容, 在文件任意位置添加
  - ios::binary 以二进制文件格式打开文件
- 也可以先创建ofstream对象，再用open函数打开

```
ofstream fout;
fout.open("test.out", ios::out|ios::binary);
```

文件的读写指针

- 标识文件操作的当前位置，该指针在哪里，读写操作就在哪里进行

```
ofstream fout("a1.out", ios::ate);
long location = fout.tellp();    //取得写指针的位置
location = 10L;
fout.seekp(location);            //将写指针移动到第10个字节处
fout.seekp(location, ios::beg); //从头数location
fout.seekp(location, ios::cur); //从当前位置数location
fout.seekp(location, ios::end); //从尾部数location
//location 可以为负值
//istream类对象的读指针函数seekg()用法也类似
```

文件的关闭

- 流对象提供close成员函数用于关闭文件
- 几乎任何情形都不需要手动调用流对象的close成员函数
- 尽管部分教材可能会要求这样做，但这是完全没有必要的，因为文件的关闭操作会在 std::*fstream 的析构函数中自动完成，无需任何手动处理

字符文件的读写

- 因为文件流也是流，所以前面讲过的流的成员函数和流操作算子也同样适用于文件流
- 例：写一个程序，将文件in.txt里面的整数排序后，输出到out.txt

```
#include<iostream>
#include<fstream>
#include <vector>
#include<algorithm>
using namespace std;
int main() {
    vector<int> v;
    ifstream srcFile("in.txt", ios::in);
    ofstream destFile("out.txt", ios::out);
    int x;
    while (srcFile >> x)
        v.push_back(x);
    sort(v.begin(), v.end());
    for (int i = 0; i < v.size(); i++)
        destFile << v[i] << " ";
    return 0;
}
```

二进制文件读写

二进制读文件

```
istream &read(char *s, long n);
```

- 将文件读指针指向的地方的n个字节内容，读入到内存地址s，然后将文件读指针向后移动n字节
  - 以ios::in方式打开文件时，文件读指针开始指向文件开头

二进制写文件

```
ostream &write(const char *s, long n);
```

- 将内存地址s处的n个字节内容，写入到文件中写指针指向的位置，然后将文件写指针向后移动n字节
  - 以ios::out方式打开文件时，文件写指针开始指向文件开头
  - 以ios::app方式打开文件时，文件写指针开始指向文件尾部

二进制文件读写

- 例：从键盘输入几个学生的姓名的成绩，并以二进制文件形式保存

```
struct Student {
    char name[20];
    int score;
};
int main() {
    Student s;
    ofstream outFile("students.dat", ios::out | ios::binary);
    while (cin >> s.name >> s.score)
        outFile.write((char *)&s, sizeof(s));
    return 0;
}
```

输入：

```
Tom 60
Jack 80
Jane 40
```

则形成的students.dat 为72 字节，用windows记事本打开，呈现：

Tom 烫烫烫烫烫烫烫烫< Jack 烫烫烫烫烫烫烫藁
Jane 烫烫烫烫烫烫烫?

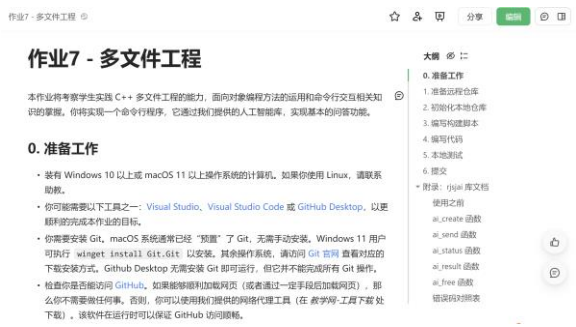
二进制文件读写

- 例：将students.dat 文件的Jane的名字改成Mike，并在屏幕上输出文件的内容

```
struct Student {
    char name[20];
    int score;
};
int main() {
    Student s;
    fstream iofile("students.dat", ios::in | ios::out | ios::binary);
    if (!iofile) {
        cout << "error";
        return 0;
    }
    iofile.seekp(2 * sizeof(s), ios::beg); //定位写指针到第3个记录
    iofile.write("Mike", strlen("Mike"));
    iofile.seekg(0, ios::beg); //定位读指针到开头
    while (iofile.read((char *)&s, sizeof(s)))
        cout << s.name << " " << s.score << endl;
    return 0;
}
```

上机安排/作业七

- 4月15日 13:00—15:00 理一1235机房
- 上机安排
  - 第五次作业讲解
  - 作业七说明
  - 模拟期中考试（13:20—15:00，共100分钟）
- 作业七：
  - 时间：4月12日 12:00—5月7日 23:59
  - 根据教学网“课程作业”栏目下的“作业7文档”的要求完成



谢谢

欢迎在课程群里填写问卷反馈