

第4章 组织程序和数据

刘 卉

huiliu@fudan.edu.cn



前言

□ 组织大型程序

1) 函数和数据结构

- 把两者结合在一个概念中——类

2) 把程序分成几个文件, 分别编译.

□ 任务/示例

1) 读取一位学生的考试和家庭作业成绩, 计算最终成绩.

2) 读取并计算全班同学的成绩.



4.1 使用函数组织计算

□ 计算最终成绩的函数

```
// compute a student's overall grade
double grade(double midterm, double final, double homework)
{
    return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}
```

□ 调用grade函数

```
cout << "Your final grade is " << setprecision(3) <<
grade(midterm, final, sum/count) << setprecision(prec) << endl;
```

4.1.1 查找中值的函数

```
// compute the median of a vector<double>
// note that calling this function copies the entire argument vector
double median(vector<double> vec)
{
    typedef vector<double>::size_type vec_sz;
    vec_sz size = vec.size();
    if (size == 0)
        throw domain_error("median of an empty vector");
    sort(vec.begin(), vec.end());
    vec_sz mid = size/2;
    return size%2 == 0 ? (vec[mid]+vec[mid-1])/2 : vec[mid];
}
```

如果vec为空,抛出异常



程序异常

```
if (size == 0)
    throw domain_error("median of an empty vector");
```

- ❑ 程序抛出异常时, 执行暂停, 并把异常对象传递给诊断程序catch.
- ❑ 抛出异常对象的类型+异常的内容→让诊断程序知道该如何处理.
- ❑ domain_error: 定义在<stdexcept>中的类型.
- ❑ domain_error对象所带字符串说明median函数不能接受空vector实参, 该内容可用在诊断信息中.



值传递

函数定义 `double median(vector<double> vec){...}`

调用时 `median(hw);` 形参 `vec` 从实参 `hw` 拷贝值

- ❑ 缺点: 调用时, 实参 `hw` 被复制到形参 `vec` 中, 耗费时间&空间.
- ❑ 优点: `median()` 函数体中调用了 `sort()`, 改变了形参 `vec` 的值, 但不会影响实参 `hw`.
- ❑ 原则: 得到一个 `vector` 的中值时, 不应改变 `vector` 本身.

4.1.2 计算最终成绩的第二种方法

```
// this function does not copy its argument, because median does so  
// for us.
```

```
double grade(double midterm, double final, const vector<double>& hw)  
{  
    if (hw.size() == 0)  
        throw domain_error("student has done no homework");  
    return grade(midterm, final, median(hw));  
}
```

□ 函数的重载(overload)

- 若干函数有相同的名字, 只要参数不同, 系统就能分辨.



抛出自己的异常

```
if (hw.size() == 0)
    throw domain_error("student has done no homework");
```

- ❑ 检测hw是否为空，如果为空，抛出异常"student has done no homework".
- ❑ median函数也会检测其参数vec是否为空，为空时会输出一般性提示信息"median of an empty vector".
- ❑ 前者能给用户更多信息!
- ❑ 后者可作为一般性功能函数, 提供给其它程序调用.



对象的引用

□ 对象的别名

- 所有对引用的操作就是对该对象的操作.

```
e.g. vector<double> homework;  
      vector<double>& hw = homework;  
      // hw is a synonym for homework
```

从定义hw开始, 对hw的任何操作 \leftrightarrow 对homework的操作.

```
e.g.  const vector<double>& chw = homework;  
      // chw is a read-only synonym for homework
```

- const: 不允许对chw作任何改变值的操作.

□ 不存在引用的引用

- 定义一个引用的引用 \Rightarrow 定义一个原先对象的引用

```
vector<double>& hw1 = hw;
```

```
const vector<double>& chw1 = chw;
```

```
// hw1 and chw1 are synonyms for homework; chw1 is read-only
```

□ 非const引用不能指向常量(对象/引用)

e.g. `vector<double>& hw1 = chw;` ✗

□ const引用可以指向常量或变量

e.g. `const vector<double>& chw1 = hw;`



引用作函数形参

- 不复制实参, 而是直接访问实参
 - 避免复制实参的开销.
 - 函数中对形参的改变就是对实参的改变.
- const引用作形参: 不允许修改所引用的实参

```
double grade(double midterm, double final, const vector<double>& hw)
```

- 实参可以是const vector对象, 也可以是vector对象.

```
double final_grade = grade(midterm, final, homework);
```

形参hw是实参
homework的常引
用——只读

4.1.3 读取家庭作业成绩的函数

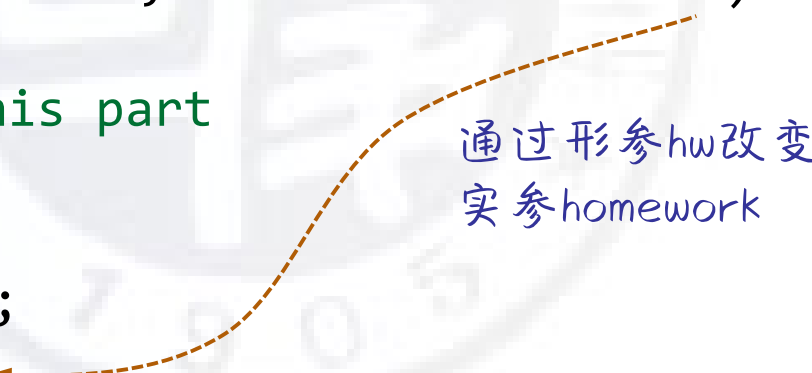
□ 函数如何返回两个值?

1) 保存家庭作业的vector对象, 2) 读取是否成功.

■ 使用非常量引用作形参, 可修改所引用的实参对象.

```
// read homework grades from an input stream into a vector<double>
istream& read_hw(istream& in, vector<double>& hw)
{
    // we must fill in this part
    return in;
}

vector<double> homework;
read_hw(cin, homework);
```



通过形参hw改变
实参homework



非常量引用作函数形参

函数定义 `istream& read_hw(istream& in, vector<double>& hw)`

函数调用 `read_hw(cin, homework);`

□ `read_hw`函数的第一个形参也是非常量引用

- 实参`cin`: 标准库定义的数据结构, 包含标准输入文件的所有信息.
- 从标准输入文件中读取→改变文件状态→`cin`发生改变→函数返回改变后的状态.

□ 优点

- 1) 函数调用时, 不需要复制流对象; 函数返回同一个对象, 也不需要复制.
- 2) 函数调用可作为条件表达式

```
if (read_hw(cin, homework)) {...}
```



读取成绩

```
// first try - not quite right
istream& read_hw(istream& in, vector<double>& hw)
{
    double x;
    while (in >> x)
        hw.push_back(x);
    return in;
}
```

- 问题一: hw是调用程序定义的, 可能已包含数据→读取前应清空
→hw.clear();
read_hw(cin, homework);

```
// second try - still not right
istream& read_hw(istream& in, vector<double>& hw)
{
    // get rid of previous contents
    hw.clear();
    double x;
    while (in >> x)
        hw.push_back(x);
    return in;
}
```

□ 问题二: 停止输入时如何处理?

- 不能再读取的两种情况: 遇到文件结束标志/无效输入
- 调用in.clear()把in的错误状态清除, 以便继续读取.


```
// read homework grades from an input stream into a vector<double>
istream& read_hw(istream& in, vector<double>& hw)
{
    if (in) {
        // get rid of previous contents
        hw.clear();
        // read homework grades
        double x;
        while (in >> x)
            hw.push_back(x);
        // clear the stream so that input will work for the next student
        in.clear();
    }
    return in;
}
```

4.1.3 三种函数形参

1. median函数的vector<double>类型形参

- 尽管效率较低, 但确保获得vector对象的中值时, 不会改变vector对象本身.

4.1.1 查找中值的函数

```
// compute the median of a vector<double>
// note that calling this function copies the entire argument vector
double median(vector<double> vec)
{
    typedef vector<double>::size_type vec_sz;
    vec_sz size = vec.size();
    if (size == 0) { // 如果vec为空, 则返回0
        throw domain_error("median of an empty vector");
    }
    sort(vec.begin(), vec.end());
    vec_sz mid = size/2;
    return size%2 == 0 ? (vec[mid]+vec[mid-1])/2 : vec[mid];
}
```

2. grade函数的const vector<double>&类型形参

- &告诉系统无需复制实参, const保证程序不会改变实参.
- 使用这样的形参效率更高.
- 当函数不会修改形参的值, 且从形参复制实参耗费时间/空间的情况下, 应使用const&类型的形参.

4.1.2 计算最终成绩的第二种方法

```
// this function does not copy its argument, because median does so
// for us.
double grade(double midterm, double final, const vector<double>& hw)
{
    if (hw.size() == 0)
        throw domain_error("student has done no homework");
    return grade(midterm, final, median(hw));
}
```

函数的重载(overload)

- 若干函数有相同的名字, 只要参数不同, 系统就能分辨.

3. read_hw函数的vector<double>&类型形参

- 函数需要改变实参的值
- 对应实参必须是左值, 而前两种类型的形参可与任意实参对应.

e.g. `vector<double> emptyvec()`

```
{  
    vector<double> v; // no elements  
    return v;  
}
```

```
// read homework grades from an input stream into a vector<double>  
istream& read_hw(istream& in, vector<double>& hw)  
{  
    if (in) {  
        // get rid of previous contents  
        hw.clear();  
        // read homework grades  
        double x;  
        while (in >> x)  
            hw.push_back(x);  
        // clear the stream so that input will work for the next student  
        in.clear();  
    }  
    return in;  
}
```

- 函数调用`grade(midterm, final, emptyvec())`会抛出一个异常, 但语法上正确.

- 函数调用`read_hw(cin, emptyvec())`会产生错误

在Dev-C++中,

```
//[Error] invalid initialization of non-const reference of  
type 'std::vector<double>&' from an rvalue of type  
'std::vector<double>'
```

在Visual Studio中,

```
read_hw(cin, emptyvec());
```

`std::vector<double> emptyvec()`

非常量引用的初始值必须为左值

使用函数计算学生的成绩

```
int main() {  
    ... // ask for and read the student's name, midterm and final grades  
    ... // ask for the homework grades  
    vector<double> homework;  
    read_hw(cin, homework); // read the homework grades  
    // compute and generate the final grade, if possible  
    try {  
        double final_grade = grade(midterm, final, homework);  
        ... // output the result  
    } catch (domain_error) {  
        cout << endl << "You must enter your grades.  "  
        "Please try again." << endl;  
        return 1;  
    }  
    return 0;  
}
```

□ try...catch...

- 执行try后面{}里的语句; 如果某处发生domain_error, 程序跳转到catch, 执行其后{}中的语句.

□ 为什么将成绩的计算和输出分成两条语句?

- 避免一条语句产生多个副作用: 1) 抛出异常, 2) 输入输出

```
// this example doesn't work
try {
    streamsize prec = cout.precision();
    cout << "Your final grade is " << setprecision(3) <<
    grade(midterm, final, homework) << setprecision(prec);
} ...
```



4.2 组织数据

□ 计算全班同学的成绩

- 输入: 逐条输入.
- 输出: 按字母顺序、对齐输出最终结果.
- 需求: 1) 保存多条学生记录, 2) 获取最长名字的长度, 以便对齐输出.

```
Harris 97 90 92 95 100 87 93 91
Smith 87 92 93 60 0 98 75 90
Carpenter 47 90 92 73 100 87 93 91
^Z
^Z
Carpenter 82
Harris 92.4
Smith 87.2
```



把学生的所有数据集合起来

□ 把学生的名字和成绩放在一起

```
struct Student_info {  
    string name;  
    double midterm, final;  
    vector<double> homework;  
}; // note the semicolon--it's required
```

- 每个Student_info对象都含有一个学生的信息.
- vector<Student_info>对象可保存多条学生信息.



处理学生记录

□ 把问题分解为易于处理的子问题

- 1) 把数据读到一个Student_info对象中;
- 2) 计算Student_info对象的成绩;
- 3) 对vector<Student_info>对象按名字排序.



读取数据

```
istream& read(istream& is, Student_info& s)
{
    // read and store the student's name, midterm and final exam grades
    is >> s.name >> s.midterm >> s.final;
    // read and store all the student's homework grades
    read_hw(is, s.homework);
    return is;
}
```



计算成绩

```
double grade(const Student_info& s)
{
    return grade(s.midterm, s.final, s.homework);
}
```

- 无需捕捉异常→函数体中调用的grade函数会处理没有完成任何作业的学生.

排序

```
sort(students.begin(), students.end(), compare); ✓
```

```
sort(students.begin(), students.end()); // not quite right
```

- 原因: 无法比较两个或多个Student_info对象
- sort函数提供一个可选参数——谓词
 - 谓词: 产生真假值(布尔值)的函数.
 - 如果提供谓词参数, sort函数将使用它来进行比较排序.

```
bool compare(const Student_info& x, const Student_info& y)  
{ return x.name < y.name; }
```

- 把比较Student_info对象的工作交给string类处理→string类提供了'<'操作符来比较两个字符串.



4.2.3 生成报表

□ 标准库函数max——<algorithm>

- max(arg1, arg2): arg1与arg2的类型必须相同.

```
maxlen = max (maxlen, record.name.size());
```

string::size_type 类型, 不能定义为int型

□ 构造string类型的无名对象

```
cout << students[i].name <<  
    string(maxlen+1-students[i].name.size(), ' ');
```

- 在student[i].name之后输出无名对象⇒总共输出maxlen+1个字符⇒对齐输出



main函数捕捉异常

- 如果学生一份作业都没做, grade函数抛出异常.
- 输出异常信息

```
try {  
    .....  
    if (hw.size() == 0)  
        throw domain_error("student has done no homework");  
} catch (domain_error e) {  
    cout << e.what();  
}
```

- 定义标准库异常类型对象→调用该对象的what成员函数→获得异常信息.



4.3 把各部分程序连接起来

□ 分别编译

- 把程序按功能分成若干文件⇒独立编译每个文件.

1) 把median函数打包⇒median.cpp

函数定义, using声明和所需的头文件.

2) 使median函数可被其他用户使用⇒median.h

创建自己的头文件, 允许他人访问自定义的程序对象⇒

在median.h中说明median函数的存在⇒与标准库函数一样

自定义头文件

```
// median.h
```

```
#include <vector>
```

```
double median(std::vector<double>);
```

Why std::vector?

Why not using std::vector?

```
// grade.cpp
```

```
#include "median.h" //a much better way to use median
```

```
#include <vector> //重复包含
```

```
double grade(double midterm, double final, const  
vector<double>& hw) {.....}
```


- 通过限定头文件中的名字, 为用户提供最大灵活性
 - 如果写了 `using std::vector` 声明, 那么包含 `median.h` 的所有程序都会有这样的声明, 不管它们是否需要.
- ⚠ 头文件应该使用完整的、经过限定的名字, 而非 `using` 声明.
- 每个头文件都可以被多次包含
 - 为了避免函数/数据结构被反复声明/定义, 在头文件中加上预处理命令 `"#ifndef...#define...#endif"`.

```
#ifndef GUARD_median_h
#define GUARD_median_h
// median.h--final version
#include <vector>
double median(std::vector<double>);
#endif
```

好习惯: #ifndef指令
位于头文件第一行!

- #ifndef指令: 检查预处理变量GUARD_median_h是否已定义
 - 否→处理#ifndef...#endif之间的所有内容.
 - 是→跳过#ifndef...#endif之间的所有内容
- 程序中第一次包含median.h时, GUARD_median_h未定义, 包含有效; 之后再包含median.h时, GUARD_median_h已定义, 包含无效.



文件包含

□ 将指定文件的内容嵌入当前源程序文件

1) #include <文件名>

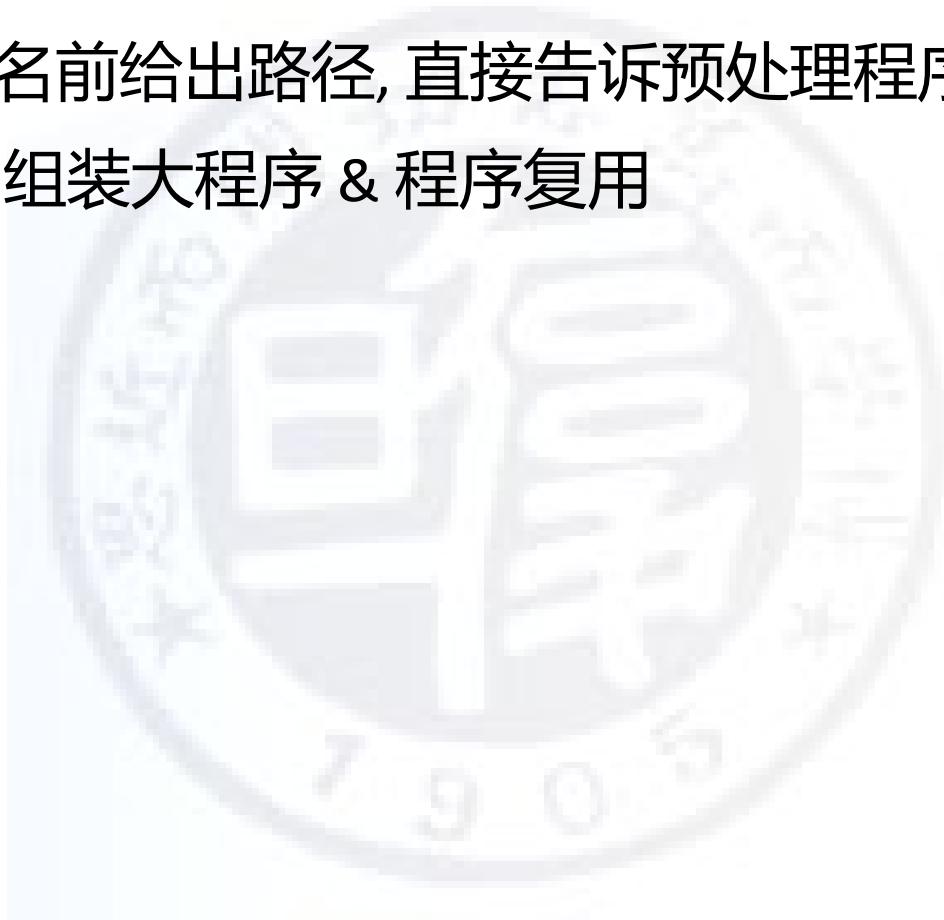
- 预处理程序到C/C++编译系统所在目录下搜索该文件. 适用于嵌入标准库的头文件.

2) #include "文件名"

- 预处理程序首先到当前工作目录寻找该文件; 找不到, 再到编译系统所在目录下查找.
- 适用于用户自己编写的头文件.

3) #include "E:\C Programming\test.h"

- 在文件名前给出路径, 直接告诉预处理程序该文件所在位置.
- 文件包含: 组装大程序 & 程序复用





自定义头文件

- 编程惯例: 将公共的常量定义/函数声明/类型定义/模板函数/内联函数/.....构成一个源文件.
- 特点: 没有执行代码, 以".h"作为扩展名.
- 其它.c/.cpp文件用到.h文件中定义/说明的程序对象
→ #include命令使它成为.c/.cpp文件的一部分.
e.g. 程序里要用到string, 就需要#include <string>

- 各.c/.cpp文件使用统一的数据结构和常量→保证程序的一致性, 便于修改程序.
- .h文件如同标准零件一样被其它.c/.cpp文件使用, 减少了重复定义的工作量.
- .h文件有改动→所有包含它的.c/.cpp文件都必须重新编译.

自定义头文件

```
// median.h
#ifndef GUARD_median_h
#define GUARD_median_h
#include <vector>
double median(std::vector<double>);
#endif
```

把median.h的全部内容复制到该处, 形成新的文件, 作为编译的源文件.

```
// grade.cpp
```

```
#include "grade.h"
#include "median.h"
#include "Student_info.h"
.....
double grade(double midterm,
              double final, const
              vector<double>& hw)
{
    .....
    return grade(midterm, final,
                  median(hw));
}
```



4.5 修改后的成绩计算程序

□ .h文件

- grade.h
- median.h
- Student_info.h

□ .cpp文件

- grade.cpp
- median.cpp
- Student_info.cpp
- main.cpp





小结

□ 头文件

- 使用`#ifndef GUARD_name_h`指令, 避免多次包含.
- 不应包含`using`声明→在标准库名字前显式使用`std::`前缀.

□ 引用

- `T&`: 类型`T`的引用, 常用于向函数传递一个它可以改变的参数, 对应的实参必须是左值.
- `const T&`: 不会改变所引用变量的值, 同时避免实参向形参传递值的复制开销.

□ 函数

- 函数必须在使用它的每个源文件中声明, 但只能定义一次.
- 函数可以重载.

□ 内联函数(inline)

- 为了避免函数调用的开销, 系统会在调用处使用函数体的一份副本.
- 内联函数定义在头文件中, 而不是源文件中.
- 功能简单的函数通常定义为内联函数.

□ 库程序

- `s1 < s2`: 按字典顺序比较两个string对象
- `s.width(n)`: 设置流s的宽度.
- `setw(n)`: 与`s.width(n)`效果相同.

□ 异常处理

- `try {...} catch {...}`: 一旦发生异常, 终止try块, 由catch块捕获异常并处理.
- `throw e`: 终止当前函数, 把e的值抛给诊断程序.
 - `e.what()`: 返回一个值, 报告错误信息.

■ 异常类

<code>logic_error</code>	<code>domain_error</code>	<code>invalid_argument</code>
<code>length_error</code>	<code>out_of_range</code>	<code>runtime_error</code>
<code>range_error</code>	<code>overflow_error</code>	<code>underflow_error</code>



using声明和using指令

□ using声明

```
using std::cin;
```

- 以后使用cin指的就是std::cin。

□ using指令

```
using namespace std;
```

- 导入std里面的所有名字。

□ 使用using声明更安全

- using声明只导入指定名称, 如果该名称与局部名称发生冲突, 编译器会报错.
- using指令导入整个命名空间中的所有名称, 包括那些根本用不到的名称;

如果有名称与局部名称发生冲突, 编译器不会报错, 而是用局部名称覆盖命名空间中的同名成员。

□ using声明仅导入所需成员, 占用资源少.