第一周作业说明

第一周作业说明

(1) 整理nth_element函数的用法和技巧
(2) 整理string的几种基本操作的使用方法
(3) 讨论HZOJ287合并果子题目和Huffman编码的关系
(4) 在C++中实现复数类
(5) 补充内容

(1) 整理nth_element函数的用法和技巧

在大家的作业中，较多同学对nth_element函数的说明细节有偏差。

nth_element对[first, last)这个**左闭右开区间**中的元素进行**部分排序**（对于选取左闭右开区间的理由，感兴趣的同学可以参考《C traps and pitfalls》，中译名《C陷阱与缺陷》，其中第3.6节关于不对称区间的内容）。

部分排序后，这个区间范围内的元素满足以下性质：

- 下标为**n**的元素左侧都是**不大于**该元素值的元素
- 下标为**n**的元素右侧都是**不小于**该元素值的元素

(2) 整理string的几种基本操作的使用方法

在大家的作业中，部分同学对string::find_first_of函数的操作说明理解有误。

string::find_first_of函数并不是搜索某个特定的字符串，而是搜寻参数字符串中**任意一个字符**的出现。

例如，下面的代码会将字符串中的所有元音字母都替换成星号。

```
#include <iostream>       // std::cout
#include <string>         // std::string
#include <cstddef>        // std::size_t

int main ()
{
  std::string str ("Please, replace the vowels in this sentence by asterisks.");
  std::size_t found = str.find_first_of("aeiou");
  while (found!=std::string::npos)
  {
```

```
    str[found]='*';
    found=str.find_first_of("aeiou",found+1);
  }

  std::cout << str << '\n';

  return 0;
}
```

(3) 讨论HZOJ287合并果子题目和Huffman编码的关系

在大家的作业中，这道题的描述大多写的不太清楚，有的同学存在证明错误的情况（例如使用了均值不等式）。

Huffman编码的目的是使用**最短的变长二进制编码**来编码数据，编码后数据的长度为$\sum_{i=1}^{n}(L[i]\times C[i])$，其中$L[i]$是第$i$种字符对应的二进制编码长度，$C[i]$是字符对应的数量。可以通过微扰法证明Huffman树的结构满足总和最小的形式。

HZOJ287合并果子题目的目的是**使用最少的体力来移动果子**，消耗的总体力为$\sum_{i=1}^{n}(S[i]\times C[i])$，其中$S[i]$为每次搬运第$i$堆果子的体力消耗，而$C[i]$是第$i$堆果子被搬运的次数。

不难发现两个和式的结构完全一致，因此两个问题可以使用相同的办法求解，构造一颗Huffman树即可。

(4) 在C++中实现复数类

在大家的作业中主要有以下几个问题。

- 没有给出默认构造函数

```
class Complex {
public:
    Complex(double r, double i) : real(r), imag(i) {};
    ...
}
```

由于在类中声明了构造函数，编译器不会自动为这个类生成一个默认构造函数。

且`Complex(double r, double i)`是没有提供默认参数的，因此进行形为`Complex c = Complex()`的调用时会产生错误。

如果代码是**刻意**进行如此设计的，通常建议声明一个标记了**`=delete`**的默认构造函数（参考Effective C++ 条款05和06），来**显式地表明自己的意图**。

- 没有重载+=操作符

一部分同学只实现了基本的四则运算操作符，这个行为本身没有什么问题，但是编译器并不会为此而生成对应的+=/-=等操作符。

对于重载的设计应该是符合操作者的直觉的。

由于C++表达式的语法中在提供了加法的时候通常也提供+=操作，实现者也应当尽量模拟出此类行为。

- 没有考虑到除以0的问题

C++标准并没有规定double类型除以0操作的具体行为，这是一个**未定义行为**。

由于大多数编译器都支持了IEC559对于浮点数运算的规定，除以0这个运算操作通常会得到NaN值和Inf值其中的一种，可以通过`assert(std::numeric_limits<double>::is_iec559)`来检验编译器是否支持IEC559标准。

需要注意的是这个值的运算是具有传播性的，如`1 + NaN = NaN`，即使出于调试的需要我们可能也需要记录一下这个值是从什么时候开始传播的，因此有必要额外处理一下除以0的操作。

- 没有使用double 而使用了float

除非拥有**非常明确**的理由，否则在代码中应当尽可能使用double。

double在大多数情况下可以提供更高的精度、更大的表示范围（更难触发Inf值）以及和float几乎一致的运算速度。

另外，浮点数字面值（如`1.0`）默认情况下是double类型的。初始化float型变量可以使用`f`后缀来在旧版的编译器上避免一次类型转换，如`float f = 1.0f;`。

在此文章中，作者给出了关于两种类型更详细的讨论。

(5) 补充内容

`deque<T>`在提供了O(1)的首尾操作的同时，还提供了**O(1)的随机访问**。因此`deque<T>`也非常适合实现大数运算的数据部分。

这是因为`deque<T>`使用了分段的数组而非链表实现。具体的实现方式可以参考侯捷撰写的《STL源码剖析》。

关于`list<T>`，`vector<T>`,`deque<T>`的性能评测可以参考此文章，其中有一些很有趣的non-trivial的结论。