CSTL技术文档

简介：

为了让C语言写算法更加容易，本文件封装了几个在C++中常用的STL，即：stack , queue , vector , set , map，同时也为了与C++语言特性相容和编程方便，使用了一个standard文件来模拟C++（如bool类型和TRUE FALSE）。下面将逐个说明可用的函数。

GitHub@BJTU-ChrisLee

注：文中使用注释来表示代码和注解。

通用标准：

所有文件都使用C….h/C….c来命名，下会称.c和.h文件

Cstandard没有.c文件，这个文件里定义了常用行为以简化代码

由于C语言不支持template（模版），所以每个文件都是用typedef定义了相应数据结构使用的类型

//例如：typedef int TYPE\_STACK; 即stack这个数据结构里存储的都将是int

如果需要改动，应在.c、.h文件中改变，如果需要同时使用同一个数据结构的不同类型（如stack<double>和stack<int>混用），请新建相同的文件，使用编译器的refactor功能逐个改变函数名以支持混用

如果需要将TYPE改为结构，则需要在对应文件里定义结构体

非常不建议将TYPE改为指针（很容易导致内存泄漏）

//注意，char和double是可以混用的

引用时只需要include .h文件即可。有的函数实际上是被封装在.c里的，比如查询树的节点的查找，是没有在.h文件里声明的，因为STL使用的时候不需要这个功能

这个虽然叫做C-STL但是实际上实现的STL功能并不多（如很重要的迭代器就没有实现），既由于C语言的限制性，也是作者比较菜的缘故，也欢迎大家交流拍砖

若分配空间失败，代码会输出”ERROR IN …”并以-1 ( 有的编译器是255 ) 退出

如果发生违规行为（如试图从空栈中获得数据），代码会printf错误信息，并且会exit(ERROR)，这与C++STL一致。其他数据结构也有类似性质，将不再赘述

所有的数据结构在使用前应该初始化，数据结构函数在使用时应该传入数据结构的指针，在使用结束的时候应该删除以防止内存泄漏，再次使用必须再初始化。

以stack为例如下：

/\*

stack s;

Stack\_Create(&s);

Stack\_Push(&s,ele);//ele必须是TYPE\_STACK类型的

while(!Stack\_Empty(&s)){  
 Stack\_Pop(&s);

}

Stack\_Del(&s);

Stack\_Create(&s);//删除之后必须重新再初始化

\*/

编码风格方面，所有的函数都采用Stl\_Cammand的方式定义，后面参数第一个一定是所需要的数据类型的指针，再后面是什么建议查阅本技术文档。在设计的时候理论上和C++的STL相近，但是由于C语言局限性和作者能力有限无法完全达到，不得已使用了一些奇怪的做法

所有的CSTL都是一个结构体，里面包含了相应的信息，理论上不推荐使用 . 访问成员元素，因为对成员元素的操作可能导致后面的函数失效，推荐使用函数获得函数的信息

文件说明

注：将从实现方法，函数效率，具体函数，实现细节、使用注意等方面说明

Cstandard.h

由于使用了typedef int bool;可能会和C++（的环境）不兼容

定义了TRUE 为 1 ，定义了FALSE 为 0，和C语言一样，这样可以强化可读性

定义了ERROR 为 -1，在分配内存失败等发生时会exit(ERROR);

定义了unsigned long 和 unsigned int 为 ulong uint

Cstack.h/Cstack.c

stack使用的是线性表实现~~（虽然作者后来发现可以用单链表实现，并且很后悔没用）~~

实现的过程中使用了指向栈顶的指针。所有函数的效率都是O(1)

定义了如下函数：

void Stack\_Create(stack\* s); //初始化stack

TYPE\_STACK Stack\_\_Top(stack\* s); //获得栈顶元素

/\*

尽管可以，但是非常不建议使用\*(s.top)的方式访问栈顶元素。更不推荐使用 s.top++这类的写法！类似警告在后文将不再给出。

\*/

void Stack\_Pop(stack\* s); //推出栈顶元素，不会返回栈顶元素的值

bool Stack\_Empty(stack\* s); //判断栈是否空，返回值为TRUE/FALSE

void Stack\_Push(stack\* s,TYPE\_STACK x); //元素x压栈

void Stack\_Del(stack\* s); //删除栈

单链表的初始长度、增长长度为定义在Cstack.c里的INI\_SIZE、PLUS\_SIZE

建议如果数据不再使用的时候就Stack\_Del以节省空间（使用线性表实际开出的空间很可能会远大于需要的空间，本代码不会自动缩小线性表占有的空间）

必须要先Stack\_Create，用过后必须Stack\_Del防止内存泄漏

Cqueue.h/Cqueue.c

queue是使用双链表实现的，也使用了指针指向头(front)和尾(push时用到)，所以函数操作时间复杂度都不会太高

定义了如下函数：

void Queue\_Create(queue\* q); //创建队列

bool Queue\_Empty(queue\* q); //判断队列是否为空

TYPE\_QUEUE Queue\_Front(queue\* q)//队首元素

void Queue\_Push(queue\* q,TYPE\_QUEUE x); //在队尾插入

void Queue\_Pop(queue\* q); //弹出队首元素，不会返回弹出元素的值

void Queue\_Del(queue\* q); //删除队列

ulong Queue\_Size(queue\* q); //返回队列长度，不推荐使用q.size!

由于使用了队列实现，所以不会有多余空间，不会像栈一样出现空间浪费的情况。也因此队列的删除的时间复杂度时O(n)。其他操作时间复杂度为O(1)

在只include .h文件时也会有Node\_Queue这个结构体，此为队列的节点，不推荐随意使用。后续set和map也会有类似问题，将不再赘述

Cvector.h/Cvector.c

使用的是线性表实现的，同时和C++STL比不能用一个长度来初始化数组，所以只能通过一个个push的方式来初始化。。。就挺鸡肋的

（主要是作者认为malloc和calloc已经足够优秀了，vector就划水吧）

定义了如下函数：

void Vector\_Create(vector\* v); //创建动态数组

bool Vector\_Empty(vector\* v); //判断是否为空。注意这里判断的是有没有 push过，而不是动态数组是否占有空间

void Vector\_Del(vector\* v); //删除动态数组

ulong Vector\_Size(vector\* v); //push进的元素数量（长度）

void Vector\_Push(vector\* v,TYPE\_VECTOR x); //在最后位推入元素

TYPE\_VECTOR Vector\_Ele(vector\* v,ulong index); //取index下标的元素

总体没什么好讲的，就挺鸡肋的没啥用233

初始长度和增长长度与stack的一致

Cset.h/Cset.c

使用的是查找树实现。C++STL中使用的是红黑树实现，本人本来准备用AVL树的，奈何太菜实在码不出代码（其实主要是AVL树的删除太麻烦了给劝退的），只好使用一般的查找树实现，因此时间复杂度都要弱于C++STL

同样由于没有实现迭代器，所以set.find()返回的是bool值而不是迭代器

另外由于初始代码的问题，所有的函数原形都是set\* tree作为形参，注意一下即可

定义了如下函数：

bool Set\_Compare\_Ele(TYPE\_SET a,TYPE\_SET b); //即a<b?

void Set\_Create(set\* tree); //创建set

bool Set\_Insert(set\* tree,TYPE\_SET ele); //插入元素，返回是否已经存在此元素

bool Set\_Find(set\* tree,TYPE\_SET ele); //查找元素，返回是否存在此元素

void Set\_DelValue(set\* tree,TYPE\_SET ele);//删除某个元素

void Set\_Print(set\* tree);//按照Set\_Node\_Print里定义的方式从小到大打印集合

void Set\_TYPE\_Print(TYPE\_SET x); //自定义打印模式，可以在.c文件中调整

void Set\_Del(set\* tree); //删除set

另外，在.c文件中包含了一些对节点操作的函数，这里不详细介绍了

数据结构使用的是查找树实现，由于不是AVL树，所以搜索、插入时间复杂度都是O(nlogn)-O(n)，而删除元素则是从O(1)-O(n)

如果需要按照自己的方式遍历打印，可以在.c文件中修改Set\_TYPE\_Print函数来达到相应目的，默认是printf(“%d ”,x);注意如果改变了TYPE\_SET，此处也要改变

由于使用了查找树，所以必须要定义元素之间的大小关系，即改变Set\_Compare\_Ele的内容。注意，默认Set\_Compare\_Ele(TYPE\_SET a,TYPE\_SET b)返回的应该是a<b的真假

Cmap.h/Cmap.c

同set，使用的是一般的查找树实现，同样由于没有实现迭代器，所以和C++STL会有不少不同，也会有很多地方实现地很奇怪

（实际上实现迭代器是可以的，但是由于C没有重载运算符的功能，所以迭代器做起来很麻烦，必须用函数来++/--，故直接放弃）

map typedef了两种数据类型，分别是TYPE\_KEY和TYPE\_VALUE，分别对应map中的key值和value值。key值有排序函数

同样由于初始代码的问题，所有的形参都是map\* tree

定义了如下函数：

bool Compare\_Ele(TYPE\_KEY a,TYPE\_KEY b); //a<b?

void Map\_Create(map\* tree); //创建map

bool Map\_Insert(map\* tree,TYPE\_KEY key,TYPE\_VALUE value);

//插入key和对应的value，返回是否已经存在，注意这里会改变原来的值

bool Map\_Find(map\* tree,TYPE\_KEY key,TYPE\_VALUE\* value);

//找到key，并通过指针value返回key对应的值。返回值为是否存在，不存 在时\*value不会发生变化

void Map\_DelKey(map\* tree,TYPE\_KEY key); //通过key值删除对应的节点

void Map\_Print(map\* tree);

//按照key值从小到大打印map中的内容方式为下一个函数定义的

void Map\_TYPE\_Print(Node\_Map\* n); //自定义打印map节点的方式

void Map\_Del(map\* tree); //删除map

和set一样，使用的是一般的查找树，所以查找、创建、删除的效率与set一样。同样，也必须定义Compare\_Ele(TYPE\_KEY a,TYPE\_KEY b)来定义a<b?以完成树的构建

如果需要得到某个key对应的value，应该用的是Map\_Find函数，此时也可以知道是否存在这个key。如果想要插入/改变某个key对应的value则应该使用Map\_Insert。。。所以如果想要完成“查找元素是否存在？不存在则插入元素”这个行为要遍历两次，这也算没用迭代器的问题吧

后记

自大一上做OJ题发现C没有C++一系列功能之后就很想把C++的STL功能移植到C上，没想到有一天居然是我自己（闲着没事做）实现了这些。其实真说这个有用吗？至少对于自己来说，我算是更近一步地了解了C++的STL，也加深了C++和C的区别（即那个++）的理解，也算是有利的一点吧

感谢你看到这。