CS 32 Week 6 Discussion 1B

this week's topics:
templates and STL, *reviewing recursion*

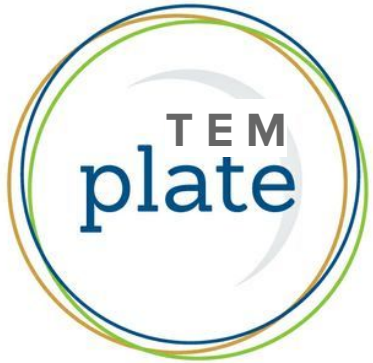
TA: **Yiyou Chen** / LA: **Ian Galvez**

a couple things before we start...

- ***First, please find a partner / form a group!*** (Yes, I mean right now.)
- We'll be doing practice problems throughout the class.
- Just a reminder: ***I highly recommend going to TA/LA office hours*** if you're ever having any questions about projects/homeworks!
- Discussion section is more for reviewing and practicing concepts that were gone over in class (i.e. doing worksheet practice problems)

Topics

- C++ Template.
- C++ Standard Template Library (STL):
  1. Containers: queue, stack, vector, list
  2. iterator
- C++ standard library algorithm:
  1. find and find_if
  2. sort



These make C++ programmers look “lazier” than C programmers!

Template

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

class Pair {
  public:
    Pair() {
      m_first = 0;
      m_second = "";
    }
    Pair(int first, string second)
      : m_first(first), m_second(second){}
    void Set_Second(const string& second);
    int Get_First() const;
    string Get_Second() const {
      return m_second;
    }
  private:
    int m_first;
    string m_second;
};

void Pair::Set_Second(const string& second) {
  m_second = second;
}

int Pair::Get_First() const {
  return m_first;
}
```

This code compiles and the objects are pairs of the form (int, string).

What if we want to modify it to pairs of general types (FirstType, SecondType)?

Template

```
31 template<typename Type1, typename Type2>
32 class Pair {
33   public:
34     Pair() {
35       m_first = 0;
36       m_second = "";
37     }
38     Pair(Type1 first, Type2 second)
39       : m_first(first), m_second(second){}
40     void Set_Second(const Type2& second);
41     Type1 Get_First() const;
42     Type2 Get_Second() const {
43       return m_second;
44     }
45   private:
46     Type1 m_first;
47     Type2 m_second;
48 };
49
50 template<typename Type1, typename Type2>
51 void Pair<Type1, Type2>::Set_Second(const Type2& second) {
52   m_second = second;
53 }
54
55 template<typename Type1, typename Type2>
56 Type1 Pair<Type1, Type2>::Get_First() const {
57   return m_first;
58 }
```

Pass by constant reference since we don't know if the copying will be expensive.

Template

```
31 template<typename Type1, typename Type2>
32 class Pair {
33   public:
34     Pair() {
35       m_first = 0;
36       m_second = "";
37     }
38     Pair(Type1 first, Type2 second)
39       : m_first(first), m_second(second){}
40     void Set_Second(const Type2& second);
41     Type1 Get_First() const;
42     Type2 Get_Second() const {
43       return m_second;
44     }
45   private:
46     Type1 m_first;
47     Type2 m_second;
48 };
49
50 template<typename Type1, typename Type2>
51 void Pair<Type1, Type2>::Set_Second(const Type2& second) {
52   m_second = second;
53 }
54
55 template<typename Type1, typename Type2>
56 Type1 Pair<Type1, Type2>::Get_First() const {
57   return m_first;
58 }
```

Create (int, string) Pair.

Pair<int, string> p;
Pair<int, string> p(1, “hi”);

Is there any possible runtime issue with this code?

Template

```
31 template<typename Type1, typename Type2>
32 class Pair {
33   public:
34     Pair() {
35       m_first = 0;
36       m_second = "";
37     }
38     Pair(Type1 first, Type2 second)
39       : m_first(first), m_second(second){}
40     void Set_Second(const Type2& second);
41     Type1 Get_First() const;
42     Type2 Get_Second() const {
43       return m_second;
44     }
45   private:
46     Type1 m_first;
47     Type2 m_second;
48 };
49
50 template<typename Type1, typename Type2>
51 void Pair<Type1, Type2>::Set_Second(const Type2& second) {
52   m_second = second;
53 }
54
55 template<typename Type1, typename Type2>
56 Type1 Pair<Type1, Type2>::Get_First() const {
57   return m_first;
58 }
```

Is there any possible runtime issue with this code?

Pair <int, double> p;

Incorrect!
double m_second=“” .

How to fix it?

Template

```
31 template<typename Type1, typename Type2>
32 class Pair {
33   public:
34     Pair() {
35       m_first = Type1();
36       m_second = Type2();
37     }
38     Pair(Type1 first, Type2 second)
39       : m_first(first), m_second(second){}
40     void Set_Second(const Type2& second);
41     Type1 Get_First() const;
42     Type2 Get_Second() const {
43       return m_second;
44     }
45   private:
46     Type1 m_first;
47     Type2 m_second;
48 };
49
50 template<typename Type1, typename Type2>
51 void Pair<Type1, Type2>::Set_Second(const Type2& second) {
52   m_second = second;
53 }
54
55 template<typename Type1, typename Type2>
56 Type1 Pair<Type1, Type2>::Get_First() const {
57   return m_first;
58 }
```

Pair <int, double> p;

Correct!
Type() creates a Type object using the default constructor.
int(), double() are 0 by default.
string() is “” by default.

Template

```
60 template<typename T>
61 //T can be int, bool, string, Pair, ...
62 bool Greater(const T& a, const T& b) {
63   if (a > b) {
64     cout << "Yes\n";
65     return true;
66   }
67   cout << "No\n";
68   return false;
69 }
```

Is there any possible runtime issue with this code?

Template

```
60 template<typename T>
61 //T can be int, bool, string, Pair, ...
62 bool Greater(const T& a, const T& b) {
63   if (a > b) {
64     cout << "Yes\n";
65     return true;
66   }
67   cout << "No\n";
68   return false;
69 }
```

Is there any possible runtime issue with this code?

Incorrect!
Pair<int, int> p1, p2;
cout<<Greater(p1, p2)<<endl;
‘>’ is not defined for Pairs.

Template

```
60 template<typename Type1, typename Type2>
61 bool operator>(const Pair<Type1, Type2>& a, const Pair<Type1, Type2>& b) {
62   return a.Get_First() > b.Get_First(); // return the comparison of first
63 }
64
65 template<typename T>
66 //T can be int, bool, string, Pair, ...
67 bool Greater(const T& a, const T& b) {
68   if (a > b) {
69     cout << "Yes\n";
70     return true;
71   }
72   cout << "No\n";
73   return false;
74 }
```

Correct!

Those self-implemented functions have higher priority.
For Pairs, it will use the overloaded >.

Standard Template Library (STL)

Pros: greatly reduces the length of the code and amount of work since we no longer have to implement some data structures by ourselves.

Cons: some “programmers” don’t really know what data structures are used for STL and simply use them, which sometimes makes their programs slow.

In reality, almost all C++ programmers use STL for most data structures.

STL: containers

In my opinion, the best way to learn and verify operations of a STL container is looking up online. A good site is cplusplus.com: https://www.cplusplus.com/reference/stl

Define a container of type CType with elements of type EType.

Ctype<EType> c;

E.g. queue<int> q; vector<double> v; list<Pair<int, string>> l; …..

Iterator: a container “pointer” that “points” to the elements of the container.

For a container c, usually
c.begin() returns an iterator to the location of its first element
c.end() returns an iterator to the location just passing the last element

Standard Template Library (STL)

queue<type> q: queue. #include <queue>
stack<type> s: stack. #include <stack>
vector<type>v : dynamic array (size not fixed). #include <vector>
list<type> l: linked list. #include <list>

Reminder: **generally, check the size of the container before popping, erasing and accessing elements.**

STL: vector

Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

Allocator:

get_allocator	Get allocator (public member function)

Popular member functions:
vector <EType> v;
v[int n]
EType& v.at(int n)
EType v.front()
EType v.back()
void v.push_back(EType e)
bool v.empty()
int v.size()
Iterator v.erase(Iterator it)
Iterator v.insert(Iterator it, EType e)

Please read the documentations to learn more ways to use them.

STL: list

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return size (public member function)
max_size	Return maximum size (public member function)

Element access:

front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign new content to container (public member function)
emplace_front C++11	Construct and insert element at beginning (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
emplace C++11	Construct and insert element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
resize	Change size (public member function)
clear	Clear content (public member function)

Operations:

splice	Transfer elements from list to list (public member function)
remove	Remove elements with specific value (public member function)
remove_if	Remove elements fulfilling condition (public member function template)
unique	Remove duplicate values (public member function)
merge	Merge sorted lists (public member function)
sort	Sort elements in container (public member function)
reverse	Reverse the order of elements (public member function)

Popular member functions:
list <EType> l;
EType l.front()
EType l.back()
void l.push_back(EType e)
void l.pop_back()
void l.push_front(EType e)
void l.pop_front()
bool l.empty()
int l.size()
Iterator l.erase(Iterator it)
Iterator l.insert(Iterator it, EType e)

Please read the documentations to learn more ways to use them.

STL: iterator

Create an iterator of container CType with elements EType:
Ctype<EType>::iterator it;
Move to next element: it++
Get the element value: *it

E.g. Traverse elements of an integer vector v.

```
84   vector<int>::iterator it;
85   for (it = v.begin(); it != v.end(); ++it) {
86     cout << *it << endl;
87   }
```

E.g. Remove elements with value val in an integer list l.

```
94   list<int>::iterator it = l.begin();
95   while(it != l.end()) {
96     if ((*it) == val)
97       it = l.erase(it);
98     else
99       it++;
00   }
```

STL: iterator

Q: given a vector iterator it, can we do *(it+2) and it = it + 2?
Given a list iterator it, can we do *(it+2) and it = it + 2?
Why?

STL: iterator

Q: given a vector iterator it, can we do *(it+2) and it = it + 2?
Given a list iterator it, can we do *(it+2) and it = it + 2?
Why and how to resolve the issue?

Yes for vector iterator since it's a dynamic array with contiguous space allocation. It + 2 points to 2 elements after it.
No for list iterator since for linked list the space is not contiguous. It + 2 doesn't move by 2 elements.
For list iterator, to get to 2 elements after the current it, one can do it++; it++;

Std library: algorithm

#include <algorithm>
This allows us to use implemented algorithms including find() and sort(), which can apply to STL containers and arrays.

Std library: find by value

#include <algorithm>
Find by value:
iterator find(iterator begin, iterator end, EType value)
It returns end if value not found, an iterator to the element if found

```
1 template<class InputIterator, class T>
2   InputIterator find (InputIterator first, InputIterator last, const T& val)
3 {
4   while (first!=last) {
5     if (*first==val) return first;
6     ++first;
7   }
8   return last;
9 }
```

E.g. check if val is in an integer list l.

```
103    list<int>::iterator it = find(l.begin(), l.end(), val);
104    if (it != l.end())
105      cout << 1 << endl;
106    else cout << 0 << endl;
```

Std library: find by value

#include <algorithm>
Find by value:
iterator find(iterator begin, iterator end, EType value)
It returns end if value not found, an iterator to the element if found

```
1 template<class InputIterator, class T>
2   InputIterator find (InputIterator first, InputIterator last, const T& val)
3 {
4   while (first!=last) {
5     if (*first==val) return first;
6     ++first;
7   }
8   return last;
9 }
```

E.g. check if val is in an integer list l.

```
103   list<int>::iterator it = find(l.begin(), l.end(), val);
104   if (it != l.end())
105     cout << 1 << endl;
106   else cout << 0 << endl;
```

Sometimes, value is not precisely defined.

For instance, for a self-defined class object, seach by value may not be well defined.

Std library: find by predicate

#include <algorithm>
Find by a predicate function:
iterator find_if(iterator begin, iterator end, bool predicate function f)
It returns an iterator to the first element satisfying a predicate f, and end if not found.

```
template<class InputIterator, class UnaryPredicate>
  InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
{
  while (first!=last) {
    if (pred(*first)) return first;
    ++first;
  }
  return last;
}
```

E.g. check if there is an element with value > 5 in an integer list l.

```
79 bool f(const int& a) {
80   return a > 5;
81 }
```

```
115   list<int>::iterator it = find_if(l.begin(), l.end(), f);
116   if (it != l.end())
117     cout << 1 << endl;
118   else cout << 0 << endl;
```

Std library: find by predicate

Check if there's a Pair in a list such that its first element is greater than 5.

```
79 bool f(const Pair<int, int>& s1) {
80   return s1.Get_First() > 5;
81 }
```

```
110   list<Pair<int, int>>::iterator it = find_if(l.begin(), l.end(), f);
111   if (it != l.end()) {
112     cout << 1 << endl;
113   }
114   else cout << 0 << endl;
```

Std library: sort by value

#include <algorithm>
Sort by value:
void sort(iterator begin, iterator end)

E.g. sort an array of strings (alphabetical order) s[] from s[1] to s[10].

```
sort(s + 1, s + 11);
```

Sometimes, value is not precisely defined.

For self-defined class objects, ‘<’ is not well defined.

Std library: sort by predicate

#include <algorithm>
Sort by a predicate function:
void sort(iterator begin, iterator end, bool predicate function f)

E.g. sort an array of strings by length (larger length comes first, same length preserves original order).

```
bool f(const string& s1, const string& s2) {
  return s1.size() > s2.size();
}
sort(s + 1, s + 11, f);
```

Interpretation the predicate:
changes the order of s1 and s2 if and only if f returns true.

Std library: sort by predicate

Sort an array of pairs<int, char> in decreasing order by comparing the values of their second elements.

```
bool f(const Pair<int, char>& s1, const Pair<int, char>& s2) {
  return s1.Get_Second() > s2.Get_Second();
}

sort(p, p+10, f);
```

templates/STL
Practice Time!

Foo Wars

(taken from Wk 6 LA Worksheet, Section 2, Problem 1)

What's the output of this program?

```
template <class T>
void foo(T input) {
     cout << "Inside the main template foo(): " << input << endl;
}

template <>
void foo<int>(int input) {
     cout << "Specialized template for int: " << input << endl;
}

int main() {
     foo<char>('A');
     foo<int>(19);
     foo<double>(19.97);
}
```

Solution

(to Foo Wars)

The second function `template<> void foo<int>(int input)` overrides the base template if you call ***foo*** using an ***int***!

Therefore, the main template is called for the first and third lines, and the special template is called for the second line.

```
Inside the main template foo(): A
Specialized template for int: 19
Inside the main template foo(): 19.97
```

Tater Iterator

(taken from Wk 6 LA Worksheet, Section 2, Problem 2)

The following code has 3 errors that cause either runtime or compile time errors.
Find and fix all of the errors.

```
class Potato {
public:
  Potato(int in_size) : size(in_size) { }
  int getSize() const {
    return size;
  }
private:
  int size;
};

int main() {
  vector<Potato> potatoes;
  Potato p1(3);
  potatoes.push_back(p1);
  potatoes.push_back(Potato(4));
  potatoes.push_back(Potato(5));

  vector<int>::iterator it = potatoes.begin();
  while (it != potatoes.end()) {
    potatoes.erase(it);
    it++;
  }

  for (it = potatoes.begin(); it != potatoes.end(); it++) {
    cout << it.getSize() << endl;
  }
}
```

Solution

(to Tater Iterator)

1: ***potatoes.begin()*** gives iterator of ***potatoes***, which is a ***vector<Potato>***, so the iterator given will be of the type ***vector<Potato>::iterator***

2: After calling erase with the iterator ***it***, ***it*** is invalidated. Instead of incrementing ***it***, the return value of ***potatoes.erase(it)*** should be assigned to it. The ***erase*** method returns the iterator of the element that is after the erased element.

3: Iterators use pointer syntax, so the last for loop should use ***it->getSize()*** instead of ***it.getSize()***.

```
class Potato {
public:
  Potato(int in_size) : size(in_size) { }
  int getSize() const {
    return size;
  }
private:
  int size;
};

int main() {
  vector<Potato> potatoes;
  Potato p1(3);
  potatoes.push_back(p1);
  potatoes.push_back(Potato(4));
  potatoes.push_back(Potato(5));

  vector<Potato>::iterator it = potatoes.begin(); // instead of vector<int>
  while (it != potatoes.end()) {
    it = potatoes.erase(it); // reassign it instead of calling it++
  }

  for (it = potatoes.begin(); it != potatoes.end(); it++) {
    cout << it->getSize() << endl; // instead of it.getSize()
  }
}
```

recursion

recursion

recursion

recursion

Practice Time!

Recursion to the Max

(taken from Wk 6 LA Worksheet, Section 1, Problem 1)

Implement the function ***getMax*** recursively.

The function returns the maximum value in ***a***, an integer array of size ***n***.

You may assume that ***n*** will be at least 1.

Interface:

```
int getMax(int a[], int n);
```

Solution (Intuition)

(to Recursion to the Max)

What's our base case?

If our array has one element, the max is trivially that element.

What's our inductive step?

Split the array into two parts: the array containing the first n-1 elements, and the last element.

If we assume we already know the max of the first n-1 elements, we can just compare whether or not the last element is greater than that max value.

If it is, the last element will be the new max, and if not, we just use the max that we already have.

In actual implementation, we just call our getMax function recursively on the first n-1 elements.

Solution (Code)

(to Recursion to the Max)

```
int getMax(int a[], int n) {
    if (n == 1)
        return a[0]; // base case: array is 1 element
    int x = getMax(a, n-1); // reduce: getMax of n-1 elems
    if (x > a[n-1]) // compare max to last element
        return x;
    else
        return a[n-1];
}
```

Discount WolframAlpha

(taken from Wk 6 LA Worksheet, Section 1, Problem 4)

Implement the following function in a recursive fashion:

```
bool isSolvable(int x, int y, int c);
```

This function should return true if there exists nonnegative integers ***a*** and ***b*** such that the equation ***ax + by = c*** holds true. It should return false otherwise.

Examples:

```
isSolvable(7, 5, 45) == true    // a == 5 and b == 2
isSolvable(1, 3, 40) == true    // a == 40 and b == 0
isSolvable(9, 23, 112) == false
```

Solution (Intuition)

(to Discount WolframAlpha)

What's our base case?

Think about when $c = 0$. Then we're solving the equation $ax + by = 0$.

No matter what x and y are, if $a = b = 0$, then this equation is always true.

What's our inductive step?

If $ax + by = c$, then $(a-1)x + by = c - x$ and $ax + (b-1)y = c - y$.

These are going to be our two "reduction cases".

If we know that either of these equations are solvable,

then $ax + by = c$ is also solvable.

Since c is always decreasing in the inductive step, if we ever get some equation $ax + by = c$ for some $c < 0$ (i.e. c is negative), it means that we *overshot* $c = 0$ and we've reached a dead end!

Solution (Code)

(to Discount WolframAlpha)

```
bool isSolvable(int x, int y, int c) {
    if (c == 0)
        return true;
    if (c < 0)
        return false;
    return isSolvable(x, y, c - x) || isSolvable(x, y, c - y);
}
// reduce: slowly take out x to check (a-1)x + by = c - x
//      or slowly take out y to check ax + (b-1)y = c - y
```

Discount WolframAlpha

(taken from Wk 6 LA Worksheet, Section 1, Problem 4)

Given a string str, recursively compute a new string such that all the ‘x’ chars have been moved to the end.

Interface: `string endX(string str);`

Example: `endX("xrxe") => "rexx"`

So No Head?

(taken from Wk 5 LA Worksheet, Section 2, Problem 1)

Given a singly-linked list class ***LL*** with a member variable ***head*** that points to the first ***Node*** struct in the list, write a function ***void LL::deleteList()*** that recursively deletes the whole list.

Assume each ***Node*** object has a ***next*** pointer.

(Hint: ***deleteList()*** takes no parameters, but to use recursion, you need to pass parameters - perhaps a helper function can do the job?)

```
struct Node {
  int data;
  Node* next;
};

class LL {
  public: // other functions such as insert not shown
    void deleteList(); // implement this function
  private: // additional helper allowed
    Node* m_head;
};
```

Solution

(to So No Head?)

What's our base case?

The list is empty, i.e. ***head == nullptr***

So do nothing!

What about the recursive step?

Assume the list is not empty, i.e. head is not nullptr. We essentially only have access to the ***head*** pointer and the ***next*** pointer in head. What can we do?

```
// We must pass the pointer by reference
// so that head gets set to nullptr!
// If we don't do this, head won't actually be modified.
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
    return;
```

Solution, cont.

(to So No Head?)

What do we do?

We can break the linked list into two sublists: the first element (which ***head*** gives us access to) and the rest of the list (which ***next*** gives us access to).

The ***next*** pointer points to what is essentially another linked list, just an element shorter. So assuming our helper function already works for size n-1, we can just recursively call it!

```
// We must pass the pointer by reference
// so that head gets set to nullptr!
// If we don't do this, head won't actually be modified.
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
    return;

  deleteListHelper(head->next);
```

Solution, cont.

(to So No Head?)

What about the first element?

Now that we know we've handled the rest of the list, we can think about what to do with the first element.

We just have to deallocate the Node object by using ***delete head;*** and then set ***head = nullptr*** for pointer safety.

Note that we have to pass the ***head*** pointer by reference so we can set it to nullptr (what would happen if we instead passed the pointer by value?)

```
// We must pass the pointer by reference
// so that head gets set to nullptr!
// If we don't do this, head won't actually be modified.
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
    return;

  deleteListHelper(head->next);
  delete head;
  head = nullptr;
}
```

Solution, cont.

(to So No Head?)

One more final touch...

This is not the actual function we're trying to implement - this is the ***helper function!*** The actual interface takes no parameters (and even if we changed it, we can't even pass ***m_head*** in because it's private). So we do a quick little workaround to get our final code.

```
// We must pass the pointer by reference
// so that head gets set to nullptr!
// If we don't do this, head won't actually be modified.
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
    return;

  deleteListHelper(head->next);
  delete head;
  head = nullptr;
}
void LL::deleteList() {
  deleteListHelper(m_head);
}
```

The Possibilities Are Endless

(not from the worksheet, credit to Yiyou for this one)

Given a character array without possible repetitions, print all permutations.

Hint: Think about induction! If we are given all the permutations of n-1 letters from the array, what's an easy way to construct permutations of n letters?

interface and expected output:

```
char s[3] = {'a', 'b', 'c'};
Perm(s, 0, 3);
```

Output:
abc
acb
bac
bca
cba
cab

Solution, cont.

(to The Possibilities Are Endless)

Time to rely on our friend combinatorics!
If we have ***n*** objects, we have ***n!*** permutations. Why??

Let's say we have 3 objects, A, B, and C.
There are 3 ways to pick the first object: either A, B, or C can go first.

Once we pick the first object, we have 2 objects left.
So now there are 2 ways to pick the next object.

Finally, after we've picked the first two objects, we only have 1 object left.
So there's only 1 possibility for the last object.

That means the number of orderings, or ***permutations***, of 3 objects are 3 * 2 * 1 = ***3!***
In general, the number of permutations of ***n*** objects is n * (n-1) * … * 2 * 1 = ***n!***
Also, note that n! = n * (n-1)! This will be very useful for doing this recursively.

Solution, cont.

(to The Possibilities Are Endless)

Okay, so what does combinatorics have to do with recursion?

Similar to the linked list problem, we can split the array into two parts: the first element, and the rest of the list.

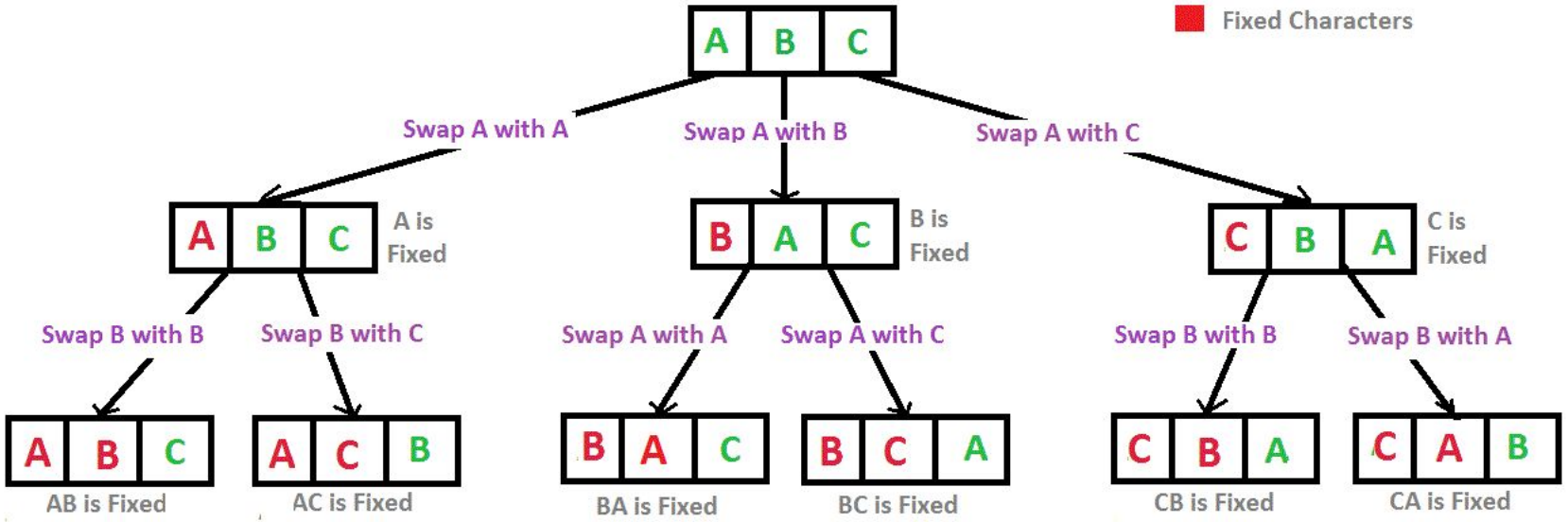
We basically need to choose which character is gonna be the first one in our specific ordering. For a list of size n, there are n ways to do this.
(In practice, we can do this by swapping each character to the front of the array.)

Then, we can realize that once we've chosen the first character, we can let our recursive function operate on the rest of the remaining characters that are left!
Essentially, we "fix" characters from the array one-by-one until we run out!

Solution, cont.

(to The Possibilities Are Endless)

One visualization of how the recursion might work:



Recursion Tree for Permutations of String "ABC"

Solution, cont.

(to The Possibilities Are Endless)

What's our base case?

Here, ***curlen*** represents the number of characters we've fixed, and ***n*** represents the total length of the string. So if ***curlen == n***, we're done creating this permutation and we can output it!

What about the recursive step?

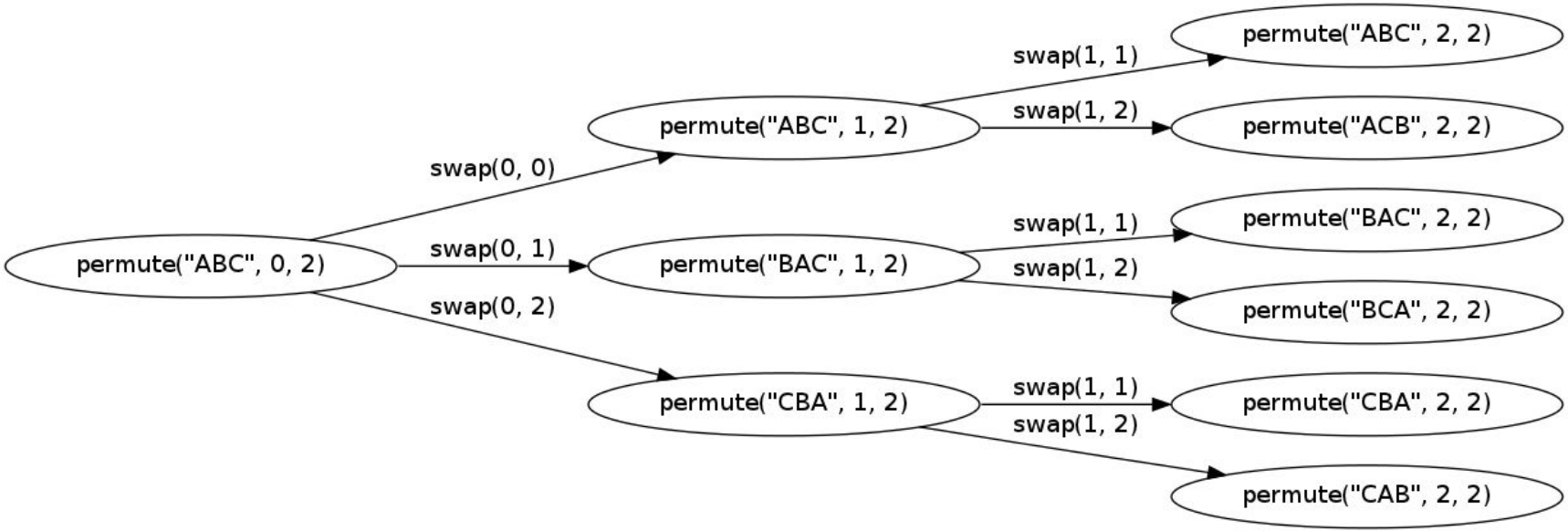
We use a for loop to swap all the remaining characters to the front of the list, and call ***Perm*** on each iteration. Doing it this way means we can operate on the string in place without creating new variables.

```
void Perm(char* s, int curlen, int n) {
  if (curlen == n) {
    cout << s << endl;
    return ;
  }
  for (int i = curlen; i < n; ++i) {
    swap(s[i], s[curlen]);
    Perm(s, curlen + 1, n);
    swap(s[i], s[curlen]);
  }
}
```

Solution, cont.

(to The Possibilities Are Endless)

One visualization of how the recursion function calls might work:



The Possibilities Are Not, In Fact, Endless

(not from the worksheet, credit to Yiyou for this one)

Given a character array without possible repetitions,
print all ***distinct*** permutations.

(the old code doesn't work anymore, otherwise we wouldn't be asking you this)

interface and expected output:

```
char s[3] = {'a', 'b', 'b'};
Perm(s, 0, 3);
```

Output (using old code):
abb
abb
bab
bba
bba
bab

Actual desired output:
abb
bab
bba

Same code doesn't work!

Solution, cont.

(to The Possibilities Are Not, In Fact, Endless)

How do we know if we have a duplicate?

Let's say we have the string "ABABC"
We know that A and B are duplicates, so we want to act ***as if*** the string was "ABC"

One way to do this without modifying the string: let's say we want to check if the second "B" is a duplicate. Walk through the string. When we see the first "B", we can conclude that "B" appears twice, so we don't perform a swap with the second "B".

This code is just one of many ways to implement the solution!

```
void Perm(char* s, int curlen, int n) {
  if (curlen == n) {
    cout << s << endl;
    return ;
  }
  for (int i = curlen; i < n; ++i) {
    bool flag = 0;
    //check if duplicate
    for (int j = curlen; j < i; ++j) {
      //if already swapped s[curlen] with s[j]=s[i], skip
      if (s[i] == s[j]) {
        flag = 1;
        break;
      }
    }
    //if no duplicates
    if (!flag) {
      swap(s[i], s[curlen]);
      Perm(s, curlen + 1, n);
      swap(s[i], s[curlen]);
    }
  }
}
```

thank you all for coming!

good luck on your midterms and projects :)

---

solutions to the rest of the worksheet problems will be posted in a couple days!