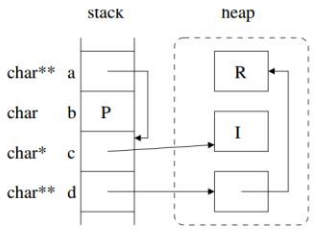
4 School Spirit [/15]

In this problem you will work with pointers and dynamically allocated memory. Write a fragment of code to create the memory diagram on the right.



Solution:

```
char** a;
char b = 'P';
char* c = new char;
*c = 'I';
char** d = new char*;
*d = new char;
**d = 'R';
a = &c;
```

Now, write a fragment a C++ code that first accesses the data in the structure to print the abbreviation of our university to `std::cout` and then cleans up all dynamically allocated memory within the above example so that the program will not have a memory leak.

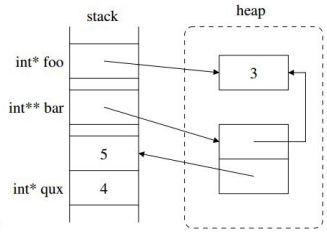
Solution:

```
std::cout << **d << b << *c << std::endl;
delete *d;
delete c;
delete d;
```

10 Diagramming Pointers & Memory [/16]

In this problem you will work with pointers and dynamically allocated memory. The fragment of code below allocates and writes to memory on both *the stack* and *the heap*. Following the conventions from lecture, draw a picture of the memory after the execution of the statements below.

```
int* foo;
int** bar;
int qux[2];
bar = new int*[2];
bar[0] = new int;
qux[0] = 4;
qux[1] = 5;
bar[1] = &qux[1];
foo = bar[0];
*foo = 3;
```



Solution:

Now, write a fragment a C++ code that cleans up all dynamically allocated memory within the above example so that the program will not have a memory leak.

Solution:

```
delete [] bar;
delete foo;
```

```
//Example of snowfall class h file
class Snowfall {
public:
// CONSTRUCTOR
   Snowfall(const std::string &c, int hours, float inches_per_hour);
// ACCESSORS
   const std::string& getCityName() const;
   float getTotal() const;
// MODIFIERS
   void setSnowInHour(int which_hour, float amount);
private:
// REPRESENTATION
   std::string city;
   std::vector<float> snow_per_hour;
};
// helper function for sorting
bool by_total_snowfall(const Snowfall &a, const Snowfall &b);
```

```
Snowfall::Snowfall(const std::string &c, int hours, float inches_per_hour) {
   city = c;
   assert (hours >= 1 && inches_per_hour >= 0);
   snow_per_hour = std::vector<float>(hours, inches_per_hour);
}
const std::string& Snowfall::getCityName() const {
   return city;
}
float Snowfall::getTotal() const {
   float total = 0;
   for (int i = 0; i < snow_per_hour.size(); i++) { total += snow_per_hour[i]; }
   return total;
}
void Snowfall::setSnowInHour(int which_hour, float amount) {
   snow_per_hour[which_hour] = amount;
}
bool by_total_snowfall(const Snowfall &a, const Snowfall &b) {
   if (a.getTotal() >= b.getTotal()) return true;
   else return false;
}
```

```
#include <string>, std::string
Using comparison (>, <) operators with strings compares ascii value.
      -ascii goes symbols, numbers, capital letters, lower case letters.
      -"A" has the lowest ascii value of uppercase letters, "a" of lowercase
std::string str = "The string.";
str.insert(3,x); //inserts x at index 3 of string str.
```

```
#include <iostream>, std::cout, std::cin
cout << std::endl; // ends line in output stream, clears buffer
//stream manipulators mentioned below are important.
//setprecision requires std::fixed
```

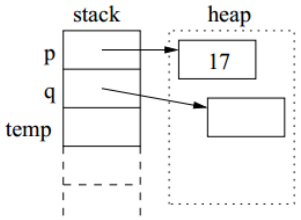
```
#include <vector>, std::vector<TYPE> //examples of vectors
vec.push_back(object); //adds object to the end of vec
vector.size() ; //returns size_type (similar to unsigned int) of the length of vector
vector<int> c (1000) //creates a vector of 1000 ints, all unassigned
vector<int> b( c ) //creates a vector that is an exact copy of c, must be similar type

int replace_in_matrix(std::vector<std::vector<int> >& matrix, int old_val, int new_val) {
int rows = matrix.size();
int count = 0;
for (int i = 0; i < rows; i++) {
int cols = matrix[0].size();
for (int j = 0; j < cols; j++) {
if (matrix[i][j] == old_val) {
matrix[i][j] = new_val;
count++;
}
}
}
return count;
}
```

Don't forget ++I, I++, and i- -

Operator	Example	Equivalent expression
+=	`index += 2`	`index = index + 2`
-=	`*(pointer++) -= 1`	`*pointer = *(pointer++) - 1`
*=	`bonus *= increase`	`bonus = bonus * increase`
/=	`time /= hours`	`time = time / hours`
%=	`allowance %= 1000`	`allowance = allowance % 1000`

```
int * p = new int;
*p = 17;
cout << *p << endl;
int * q;
q = new int;
*q = *p;
*p = 27;
cout << *p << " " << *q << endl;
int * temp = q;
q = p;
p = temp;
cout << *p << " " << *q << endl;
delete p;
delete q;
```



- The expression `new int` asks the system for a new chunk of memory that is large enough to hold an integer and returns the address of that memory. Therefore, the statement `int * p = new int;` allocates memory from the heap and stores its address in the pointer variable p.
- The statement `delete p;` takes the integer memory pointed by p and returns it to the system for re-use.

- This memory is allocated from and returned to a special area of memory called the *heap*. By contrast, local variables and function parameters are placed on the *stack* as discussed last lecture.
- In between the `new` and `delete` statements, the memory is treated just like memory for an ordinary variable, except the only way to access it is through pointers. Hence, the manipulation of pointer variables and values is similar to the examples covered in Lecture 5 except that there is no explicitly named variable for that memory other than the pointer variable.
- Dynamic allocation of primitives like ints and doubles is not very interesting or significant. What's more important is dynamic allocation of arrays and objects.

```
int main() { //this code parses text, counts words/letters per sentence
   std::ifstream istr("little_engine_that_could.txt");
   std::string word;
   int num_chars = 0;
   int num_words = 0;
int num_sentences = 0;
while (istr >> word) {
Solution:
   num_words++;
   num_chars += word.size();
   // when we find the end of a sentence
   if (word[word.size()-1] == '.' || word[word.size()-1] == '?' || word[word.size()-1]
== '!') {
      num_sentences++;
      num_chars -= 1;
// print out stats about that sentence
std::cout << "sentence " << std::setw(3) << num_sentences
<< " # words: " << std::setw(3) << num_words
<< " avg chars/word:" << std::fixed << std::setprecision(1)
<< std::setw(5) << num_chars / double(num_words) << std::endl;
// reset the counters
num_chars = 0;
num_words = 0;
}
}
}
```

```
Triangle::Triangle(double a, double b, double c) {
a_ = a;
b_ = b;
c_ = c;
}
Triangle::Triangle(const std::vector<double>& vals) {
assert (vals.size() == 3);
a_ = vals[0];
b_ = vals[1];
c_ = vals[2];
}
double Triangle::getPerimeter() const {
return a_ + b_ + c_;
}
3double Triangle::getArea() const {
double s = getPerimeter()/2.0;
return sqrt(s*(s-a_)*(s-b_)*(s-c_));
}
void Triangle::doubleShortestEdge() {
if (a_ <= b_ && a_ <= c_) {
a_ *= 2;
} else if (b_ <= a_ && b_ <= c_) {
b_ *= 2;
} else {
assert (c_ <= a_ && c_ <= b_);
c_ *= 2;
}
}
```

```
class Triangle {
public:
// TWO CONSTRUCTORS
Triangle(double a, double b, double c);
Triangle(const std::vector<double>& edges);
// ACCESSOR
double getPerimeter() const;
double getArea() const;
// MODIFIER
void doubleShortestEdge();
private:
// REPRESENTATION
double a_;
double b_;
double c_;
};
```

TA: Mauricio
Mentors:

Matt S., John Allwein,
Alec, & Fred

SHINIGAMI LOVE APPLES

L-Value/R-Value, sides of equal sign, L-Value finds location in memory.

Add matrix cycling code.

Example of for loop, while loop, if not there

- O(1), a.k.a. CONSTANT: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
- O(log n), a.k.a. LOGARITHMIC. e.g., dictionary lookup, binary search.
- O(n), a.k.a. LINEAR. e.g., sum up a list.
- O(n log n), e.g., sorting.
- O(n2), O(n3), O(nk), a.k.a. POLYNOMIAL. e.g.,
- O(2n), O(kn), a.k.a. EXPONENTIAL. e.g., Fibonacci, playing chess.

Solution: O(n4) We have to search for each of the numbers from 1 ! n2. We look for that number in each of the n2 slots of the matrix. An alternate solution uses a helper vector of n2 bools, and checks o each element as it sweeps through the data ! O(n2). Or if sorting is used, the runtime is O(n2 log n).