(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)

#include: <vector>, <utility>, <algorithm>, <iostream>, <fstream>, <algorithm>, <sstream>

#include “file.h”

String are array of char with (‘/0’) at end

.c_str()-return const char

std::vector <type>

std::pair<type, type>

int main(int argc, char* argv[])

std::string name(len, thing)

std::ifstream in_str(); (.good() return true if good)

std::ofstream out_str();

out_str.close();

stoi(string to int);

.substr(index,len);

.push_back(stuff)

&something - reference

-you want to const the get functions.

-pointers hold memory addresses, *means follow the pointer, no alias unless pass by reference

if (p == q) or **if (p != q)** are legal ways to compare pointers

-arrays point to a block of memory ++p moves to next location in the array.

for (p=a; p<a+n; ++p) loop through array with pointers.

std::sort (.begin(), .end(), function to return true or false)

std::sort(rooms.begin(), rooms.end());

std::vector<std::string>::iterator unique_rooms = std::unique(rooms.begin(), rooms.end());

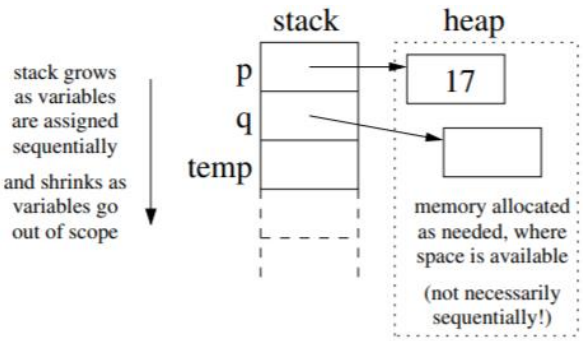
rooms.resize(std::distance(rooms.begin(), unique_rooms));

-use **new** to put it on the heap, use **delete** to remove from heap.

```
while (!in_str.eof()) {
    in_str >> my_variable;
    // do something with my_variable
  }
```

(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)(ᵔᴥᵔ)

```
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;
```



Static memory: variables allocated statically (with the keyword static). They are are not eliminated when they go out of scope. They retain their values, but are only accessible within the scope where they are defined. NOTE: Static variables are not very common.

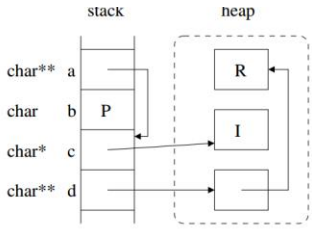
Automatic memory: memory allocation inside a function when you create a variable. This allocates space for local variables in functions (on the stack) and deallocates it when variables go out of scope. For example: int x; double y;

Dynamic memory: explicitly allocated (on the heap) as needed. This is our focus for today.

operator<(for sorting)

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.

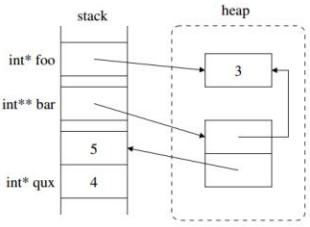
Solution:

```
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;
```