```
child Alice Williams
child Ellen Davis
child Frank Jones
pet Garfield Davis
child Henry Williams
pet Mittens Brown
child Ryan Jones
pet Spot Jones
pet Tweety Davis
```

We will use the Family class to organize, sort, and print this output:

```
Jones Family
  children:  Frank Ryan
  pets:  Spot
Williams Family
  children:  Alice Henry
Davis Family
  children:  Ellen
  pets:  Garfield Tweety
Brown Family
  pets:  Mittens
```

Note that the children and pets are grouped by last name. The families
Families with the same number of children are ordered by last name.

6.1 Using the Family Class [/15]

Complete this fragment of code to read the input file and produce the outp

```
std::string filename = "family_input.txt";
std::ifstream istr(filename);
if (!istr.good()) {
  std::cerr << "ERROR: could not open " << filename << std::endl;
  exit(1);
}
```

Solution:

```
std::vector<Family> families;
std::string type, first, last;
while (istr >> type >> first >> last) {
  int found;
  for (found = 0; found < families.size(); found++)
    if (families[found].lastName() == last) {
      break;
    }
  }
  if (found == families.size()) {
    families.push_back(Family(last));
  }
  if (type == "child") {
    families[found].addChild(first);
  } else {
    assert (type == "pet");
    families[found].addPet(first);
  }
}


std::sort(families.begin(), families.end());
for (int i = 0; i < families.size(); i++) {
  families[i].print();
}
```

```
class Family {
public:
  // CONSTRUCTORS
  Family(const std::string& n);
  // ACCESSSORS
  const std::string& lastName() const;
  int numChildren() const;
  bool isPet(const std::string &n) const;
  // MODIFIERS
  void addChild(const std::string& n);
  void addPet(const std::string& n);
  // PRINT
  void print() const;
private:
  // REPRESENTATION
  std::string name;
  std::vector<std::string> children;
  std::vector<std::string> pets;
};

// SORTING HELPER FUNCTION
bool operator< (const Family &a, const Family &b);
```

Solution:

```
// CONSTRUCTOR
Family::Family(const std::string& n) {
  name = n;
}

// ACCESSORS
const std::string& Family::lastName() const {
  return name;
}
int Family::numChildren() const {
  return children.size();
}
bool Family::isPet(const std::string &n) const {
  for (int i = 0; i < pets.size(); i++) {
    if (pets[i] == n) return true;
  }
  return false;
}

// MODIFIERS
void Family::addChild(const std::string& n) {
  children.push_back(n);
}
void Family::addPet(const std::string& n) {
  pets.push_back(n);
}

// SORTING HELPER FUNCTION
bool operator< (const Family &a, const Family &b) {
  return (a.numChildren() > b.numChildren() ||
          (a.numChildren() == b.numChildren() && a.lastName() < b.lastName()));
}
```

Text Justification

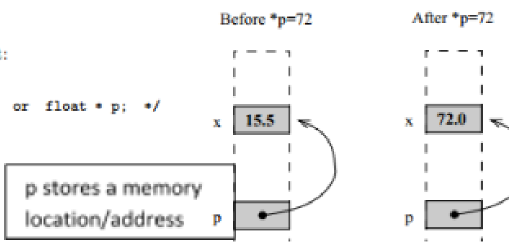
```
void print_square(const std::string& sentence) {
  // calculate dimensions of smallest square
  int dim = ceil(sqrt(sentence.size()));
  std::cout << std::string(dim+2,'*') << std::endl;
  // helper variable to select next character of the sentence
  int k = 0;
  for (int i = 0; i < dim; i++) {
    std::cout << "*";
    for (int j = 0; j < dim; j++) {
      // make sure we don't attempt to access characters beyond the end of the string
      if (k < sentence.size()) {
        std::cout << sentence[k];
        k++;
      } else {
        std::cout << " ";
      }
    }
    std::cout << "*" << std::endl;
  }
  std::cout << std::string(dim+2,'*') << std::endl;
}
```

```
*******
*Here *
*is an*
* exam*
*ple. *
*     *
*******
```

Pointer Example

Consider the following code segment:

```
float x = 15.5;
float *p; /* equiv:  float* p;  or  float * p;  */
p = &x;
*p = 72;
if ( x > 20 )
  cout << "Bigger\n";
else
  cout << "Smaller\n";
```

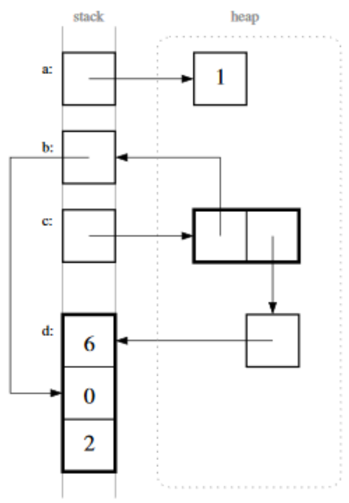


8 Memory Diagramming [/ 22]

Write code to produce the memory structure shown in the diagram to the right.

Solution:

```
int* a = new int;
*a = 1;
int* b;
int*** c = new int**[2];
c[0] = &b;
c[1] = new int*;
int d[3];
d[0] = 6;
d[1] = 0;
d[2] = 2;
*c[1] = d;
b = &d[1];
```



TA: Mauricio, Alec Eric P, Nate, Osama

#include <vector>

```
// Initialize
std::vector<type> vec;
//Makes vec. of 10 doubles set to 3.1
std::vector<double> scores(10, 3.1);
//Makes vec. b exact copy of scores.
std::vector<double> b(scores);

vec.empty() //Returns if vec is empty
vec.size() //Returns size of vec
vec.clear() //NR clears values of vec
vec.insert(pos (itr),val)// inserts value
vec.push_back(val) //NR adds val to
                     end of vector
```

#include <algorithm>

```
std::sort(vec.begin(), vec.end(), opt);
//NR default is alphabetically
```

```
int do_something(int& a, int& b)
/* Passing by reference (&) gives
address of original instead of copyin
entire value into function.
    If you make a change to "a" in
function, the original value will also
be changed.
    Passing item by "const &" means
the item is not copied, but any
changes to it will not affect original
*/
```

#include <string>

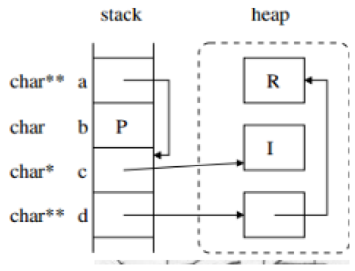
```
// Initialize
std::string str;
std::string str = "hello";
//Makes string of five 'a's "aaaaa"
std::string str(5,'a')
str = "Susan";  // str[1] Equals 'u'

//Makes temp. string that's not
//assoc. with variable name.
std::string(num, 'char')

str.length() //Returns length of str
str.substr(index, length) //To go from
index to end, use string::npos
str.find(str1,pos) //Returns first pos
   str1 was found in str.
str.rfind(str1) //Find, just last
   occurrence

if (str.find(str1) != std::string::npos)
      // It was found
```

POINTERS AND DYNAMIC MEMORY:

new //word to create space in heap
delete //word to clean up heap var.
1 delete for every new

```
c->do()
   ==
 (*c).do()
```

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



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

int *a; (*a).x = 5; a->x = 5

st

//

```
class Student {

// ACCESSORS
const std::string& name() const { return name_; }
const std::string& id_number() const { return id_number_; }
double hw_avg() const { return hw_avg_; }
// MUTATORS
bool read(std::ifstream& in_strm, unsigned int num_homeworks, unsigned int num_tests);
void compute_averages(double hw_weight);
std::ofstream& output_name(std::ofstream& out_str) const;

private:            // Because no constructors were made, default name is
std::string name_;  // equal to the private variables initial values.
std::string id_number_;
std::vector<int> hw_scores_;
double test_avg_;
double final_avg_;
}; //←NEED SEMICOLON AT END OF CLASS DECLARATION
bool less_names(const Student& stu1, const Student& stu2);
```