CSCI-1200 Data Structures — Fall 2016
Test 1 — Solutions

1 Keeping up with not-the-Kardashians [/45]

In this problem you will implement a simple class named `Family` to keep track of the children and pets that are members of a family. All members of a family have a common last name. *IMPORTANT: Read through all 4 pages of this problem before beginning your portion of the implementation.*

Here's a simple example using the `Family` class:

```
Family king("King");
king.addChild("Chris");
king.addPet("Buddy");
king.addChild("Sally");
std::cout << "The " << king.lastName() << " family has "
          << king.numChildren() << " children." << std::endl;
if (!king.isPet("Socks")) {
  std::cout << "The family does not have a pet named Socks." << std::endl;
}
king.print();
```

And here's the output from this code:

```
The King family has 2 children.
The family does not have a pet named Socks.
King Family
  children:  Chris Sally
  pets:  Buddy
```

We'll also parse data on the children and pets of multiple families from a file. For example if the input file `family_input.txt` contains:

```
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 with the most children are printed first. Families with the same number of children are ordered by last name.

1.1 Using the `Family` Class [/15]

Complete this fragment of code to read the input file and produce the output on the previous page.

```
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();
}
```

1.2 `Family` Class Declaration [/18]

Using the sample code on the previous pages as your guide, write the class declaration for the `Family` object. That is, write the *header file* (`family.h`) for this class. You don't need to worry about the `#include` lines or other pre-processor directives. Focus on getting the member variable types and member and non-member function prototypes correct. Use `const` and call by reference where appropriate. Make sure you label what parts of the class are `public` and `private`. Save the implementation of *all functions* for the `family.cpp` file, which is the next part.

Solution:

```
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);
```

1.3 `Family` Class Implementation [/12]

Now implement all of the functions prototyped in the `family.h` file, as they would appear in the corresponding `family.cpp` file. *NOTE: You may omit the implementation of the print() function.*

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()));
}
```

2 Text Justification Redux [/18]

Write a function named `print_square` that takes in a single argument, an STL `string`, and reformats that text to fit in the *smallest square box*, surrounded by a border of stars. Unlike Homework 1, we won't worry about fitting complete words or hyphenation. Just break the words when you get to the end of the row. A few sample calls to the function are shown below, and the output to `std::cout` of each call is shown on the right.

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

```
*********
*the qui*
*ck brow*
*n fox j*
*umped o*
*ver the*
* lazy d*
*ogs    *
*********
```

```
*************
*Twinkle, tw*
*inkle, litt*
*le star, ho*
*w I wonder *
*what you ar*
*e. Up above*
* the world *
*so high, li*
*ke a diamon*
*d in the sk*
*y.         *
*************
```

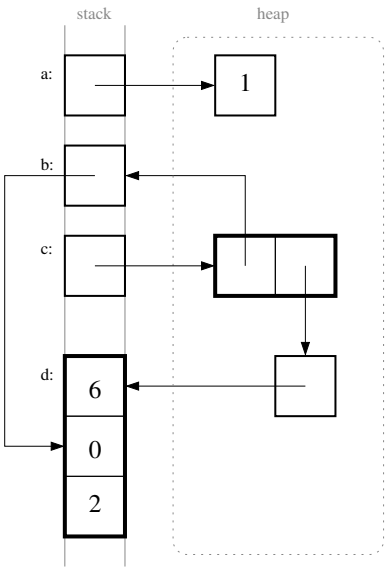
```
print_square("Here is an example.");
print_square("the quick brown fox jumped over the lazy dogs");
print_square("Twinkle, twinkle, little star, how I wonder what you are. Up above the " +
             "world so high, like a diamond in the sky.");
```

Solution:

```
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;
}
```

3 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];
```

Write code to print the current year to `std::cout` using ***ALL*** of the variables (`a`, `b`, `c`, and `d`).

Solution:

```
  std::cout << d[2] << *b << *a << **c[1] << std::endl;
```

Finally, write code to clean up the dynamically-allocated memory so we don't have any leaks.

Solution:

```
  delete a;
  delete c[1];
  delete [] c;
```

4 HasLetter [/12]

For this problem you will write a function named `HasLetter` that accepts 3 arguments named `words`, `letter`, and `selected`. The function should examine the strings in the `words` vector, collecting all strings that contain the character `letter` in the `selected` vector.

If `words` contains these 8 words: `dog bird cat fish turtle horse goat hedgehog`
Then after executing this command:

```
HasLetter(words,'r',selected);
```

The `selected` vector will contain: `bird turtle horse`
If we then execute:

```
HasLetter(words,'o',selected);
```

Now the `selected` vector will contain: `dog horse goat hedgehog`

Solution:

```
void HasLetter(const std::vector<std::string> &words, char letter, std::vector<std::string> &selected) {
  // clear out the vector of any previous answer
  selected.clear();
  for (int i = 0; i < words.size(); i++) {
    // use a boolean to check for the letter
    // (in case there are repeated letters)
    bool flag = false;
    for (int j = 0; j < words[i].size(); j++) {
      if (words[i][j] == letter) {
        flag = true;
        break;
      }
    }
    if (flag)
      selected.push_back(words[i]);
  }
}
```