CSCI-1200 Data Structures — Spring 2023
Test 1 — Solutions

1 Short Answer Round [/ 24]

For each of the following statements, write if it is true or false, and then write 1-2 *complete* sentences explaining why. Most of these statements are false.

Reminder: Write complete sentences!

1.1 Big Parameters [/3]

True or False Variables larger than 8 bytes should always be passed in by constant reference.

Solution: False. If we want to change the value of an argument and have the caller see the change, we will need to pass by reference instead of by constant reference.

1.2 Default private [/3]

True or False If we do not use the `public` or `private` keyword in a class definition, then all members of the class will be treated as `private`.

Solution: True. For classes, the default is `private`. It's structs that have everything default to `public`.

1.3 Member Output Operator [/3]

True or False We should write the output stream `operator<<` for our own class `Foo` as a member function of `Foo`, so we can have access to private member variables of `Foo` when printing.

Solution: False. When we make a call like `cout << Foo`, either this must be a non-member function with first argument being `cout`, or this must be a member of the `cout` class. Neither of those situations will let us access `Foo`'s private member variables.

1.4 Returning Constant Reference [/3]

True or False When returning a `string`, we should never use a `const& string` return type.

Solution: False. We can only (and should) return a `const string&` if we are returning a member variable. If we are returned a locally declared variable, a reference doesn't make sense because the local copy will be invalidated when automatic memory management cleans it up at the end of the function.

Note: The intent was to ask about `const string&` but the `&` was in the wrong place on the test. So "`const& string`" is not a valid type is also an acceptable response.

1.5 Sorting Tie Breakers [/3]

True or False When writing a custom sort function to pass into `std::sort`, if there is a tie the function must return `return false;`.

Solution: True. Since what we pass must provide return true only if we want the first value to come before the second value that is passed in, we will have to return false if they are "equivalent" in placement. Consider that the default `sort()` behavior uses `operator<`, which is strictly a less-than operator and not a less-than-or-equal-to operator.

1.6 Default Constructor? [/3]

True or False If a class `Foo` only has one constructor in the class declaration, `Foo(int x=5)`, we cannot declare a instance like `Foo f;` because there is no default constructor.

Solution: False. As shown in Lecture 4, as long as all arguments have default values in the constructor's

argument list, we can still declare an instance of the object without providing explicit values.

1.7 Cleaning Up Memory [/3]

True or False There are no memory leaks in the following code:

```
int* p = new int;
int* q = p;
delete p;
```

Solution: True. There is only one int allocated on the heap in this code fragement, and it is properly deleted. If we added `delete q` we would introduce a memory bug by deleting memory that was already being deleted by `delete p`.

1.8 Array Direction [/3]

True or False Arrays have their index 0 at the highest (largest) memory address and for an `int a[n]` where `n` is a positive number, the valid range of addresses is `(a-n, a]`.

Solution: False. Arrays have their base address (index 0) at the lowest memory address so that using `operator++` on an array pointer advances it to the next index. This means that the bounds of the array are `[a,a+n)`.

2 Clown Class [/ 32]

In this problem you will write a `Clown` class to represent students studying at a clown school to learn skills. Each clown has a name and keeps track of the skills they learn. You can assume no two clowns will have the same name.

Skills are taught through the `learn_skills` function. A clown cannot learn a skill they already know. Sorting by `order_by_skill` function should sort by the number of skills, resolving any ties by name. The following example illustrates how the class should work:

```
  Clown a("Applejack"), b("Fluttershy"), c("Rarity");
+++  std::cout << a.get_name() << " knows " << a.skill_count() << " skills." << std::endl;
---  assert(a.get_name() == "Applejack");
---  assert(a.skill_count() == 0);

bool success = b.learn_skill("Juggling");
---  assert(success && b.skill_count() == 1);

  std::vector<std::string> skills({"Juggling", "Tightrope"});
  int newly_learned = b.learn_skills(skills);
+++  std::cout << b.get_name() << " learned " << newly_learned << " skills." << std::endl;
---  assert(newly_learned == 1 && b.skill_count() == 2);

  newly_learned = c.learn_skills(skills);
+++  std::cout << c.get_name() << " learned " << newly_learned << " skills." << std::endl;
---  assert(newly_learned == 2);

  std::vector<Clown> clowns;
  clowns.push_back(c);
  clowns.push_back(b);
  clowns.push_back(a);

+++  std::cout << std::endl << "Summary:" << std::endl;
  std::sort(clowns.begin(), clowns.end(), order_by_skill);

  for(unsigned int i = 0 ; i < clowns.size() ; i++){
    clowns[i].print(std::cout);
  }
```

Which produces the output:

```
Applejack knows 0 skills.
Fluttershy learned 1 skills.
Rarity learned 2 skills.

Summary:
Fluttershy has learned Juggling, Tightrope
Rarity has learned Juggling, Tightrope
Applejack has learned no skills
```

Note: On the exam the sample code was not up to date, and was missing the lines starting with `+++` and had lines which we had intended to remove, marked with `---`. We also intended to remove `learn_skill()` to save some writing. Having a `learn_skill()` function does simplify `learn_skills()`. Graders based their grading on whether you were using the first or second half of output for your `print()` function, we had to treat either approach as correct due to the old sample code issue.

The `assert()` function was not covered in lecture prior to the exam, there is an example in the Lecture 4 materials on the calendar that has equivalent code. We will address it again in our "C++ Exceptions" lecture near the end of the semester.

2.1 `Clown` class declaration (Clown.h file) [/12]

Write the declaration for the `Clown` class and any non-member functions. We recommend that for functions that will only have a single short line of implementation, you place the implementation in this file. Remember that longer function implementations should be put in the `.cpp` file in the next question.

Solution:

```
class Clown{
public:
  Clown(const std::string& name) { m_name = name; }
  const std::string& get_name() const { return m_name; }
  int skill_count() const { return m_skills.size(); }
  bool learn_skill(const std::string& skill);
  int learn_skills(const std::vector<std::string>& skills);
  void print(std::ostream& ostr) const;

private:
  std::string m_name;
  std::vector<std::string> m_skills;
};

bool order_by_skill(const Clown& c1, const Clown& c2);
```

2.2 Non-member functions (Clown.cpp) [/6]

Write the implementation of any **non-member** functions for the `Clown` class. You do not need to put any `#include` statements in this part.

Solution:

```
bool order_by_skill(const Clown& c1, const Clown& c2){
  if(c1.skill_count() == c2.skill_count()){
    return c1.get_name() < c2.get_name();
  }
  else{
    return c1.skill_count() > c2.skill_count();
  }
}
```

2.3 `Clown` implementation (Clown.cpp) [/14]

Write the implementation of the `Clown` class's member functions.

Solution:

```
#include "clown.h"

bool Clown::learn_skill(const std::string& skill){
//Option 1
if(std::find(m_skills.begin(), m_skills.end(), skill) != m_skills.end()){
return false;
}

//Option 2
for(unsigned int i=0; i<m_skills.size(); i++){
if(m_skills[i] == skill){
return false;
}
}

m_skills.push_back(skill);
return true;
}
```

```
int Clown::learn_skills(const std::vector<std::string>& skills){
int ret = 0;
for(unsigned int i=0; i<skills.size(); i++){
if(learn_skill(skills[i])){
ret++;
}
}
return ret;
}

void Clown::print(std::ostream& ostr) const{
  ostr << m_name << " has learned ";
  if(skill_count() == 0){
    ostr << "no skills" << std::endl;
  }
  else{
    ostr << m_skills[0];
    for(unsigned int i=1; i<m_skills.size(); i++){
      ostr << ", " << m_skills[i];
    }
    ostr << std::endl;
  }
}
```

3 Acronym Finder [/ 24]

In this question, we define the *acronym* of a phrase to be the string that is formed when taking the uppercase first letters of each word (as separated by spaces) in that phrase. For example, "Data Structures", "Denial of Service", and "D S" are phrases that have the acronym "DS". The phrase "DataStructures" has the acronym "D" instead of "DS", because the lack of spaces means it is treated as one word. "Data Structures and Algorithms" has the acronym "DSA" because we skip "and" since it does not start with a capital letter.

Fill in the function prototype for `isAcronym()`, which takes in two strings, one being the phrase and other being the acronym, and returns a boolean indicating whether the phrase's acronym is the one that's supplied. Also fill in the function implementation. You cannot declare any additional variables or loops, you must make use of the arguments, the two `int`s we declare, and the single `while` loop.

You can assume the phrase only has letters and spaces, and that the acronym only has capital letters. Here are a few usage cases:

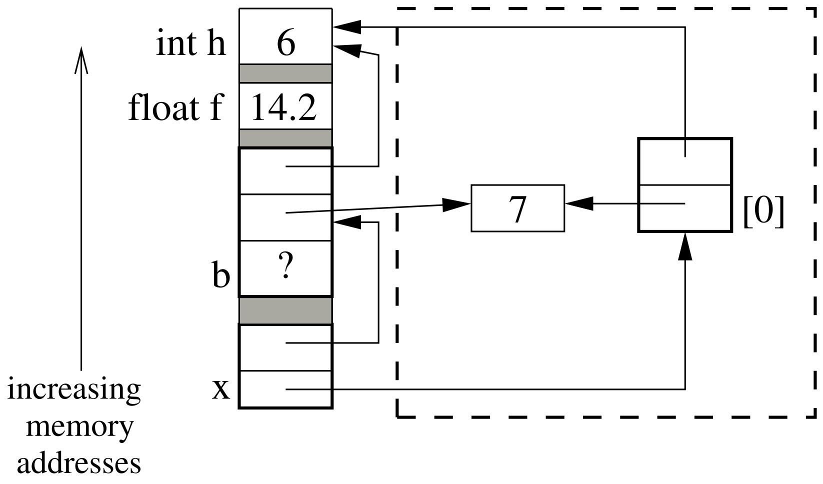
```
isAcronym("Data Structures", "DS")); //true
isAcronym("Data Structures and Algorithms", "DSA")); //true
isAcronym("DataStructures", "DS"); //false
```

Note: We intended to mention that `isupper()` is a function – it's defined in `cctype`, but this got lost in the editing process. Since any valid acronym is all capital letters, the second argument was already all caps and thus checking for capitals is unnecessary. If we writing a `getAcronym()` function, then it would matter!

```
bool isAcronym(const std::string& phrase, const std::string& acronym){
  unsigned int phrase_pos = 0;
  unsigned int acronym_pos = 0;
  while(phrase_pos !=phrase.size()){
    if((phrase_pos == 0 || phrase[phrase_pos - 1] == ' ')){
      //Could use phrase[phrase_pos] <= 'Z' && phrase[phrase_pos] >= 'A' instead of isupper()
      //This if statement is not needed for isAcronym
      if(isupper(phrase[phrase_pos])){
        if(acronym_pos < acronym.size() && phrase[phrase_pos] == acronym[acronym_pos]){
          acronym_pos++;
        }
        else{
          return false;
        }
      }
    }
    phrase_pos++;
  }
  return acronym_pos == acronym.size();
}
```

4 Memory Drawing [/ 17]

Write code to produce the memory diagram shown on this page. Some types have been left out. Thicker bold boxes indicate arrays.



```
int** x[2];
int* b[3];
float f = 14.2; //double works too
int h = 6;
x[0] = new int*[2];
x[1] = b + 1; // or &(b[1]), parentheses unneeded but help readability
b[1] = new int; //Depending on order, may have b[1] = x[0][0]
                    //Only if x[0][0] = new int done first though.
*(b[1]) = 7; //Parentheses unneeded, but helps readability
b[2] = &h;
x[0][0] = b[1];
x[0][1] = &h;


//All memory can be freed:
delete b[1];
delete [] x[0];
```