Exam 3 Practice 2 Key

VOCAB KEY

1 Declaration

2 Polymorphism

3 Shadowing

4 Operator

5 Namespace

6 Container

7 Template

8 Standard Template Library

9 Method

10 Abstract Method

MULTIPLE CHOICE KEY

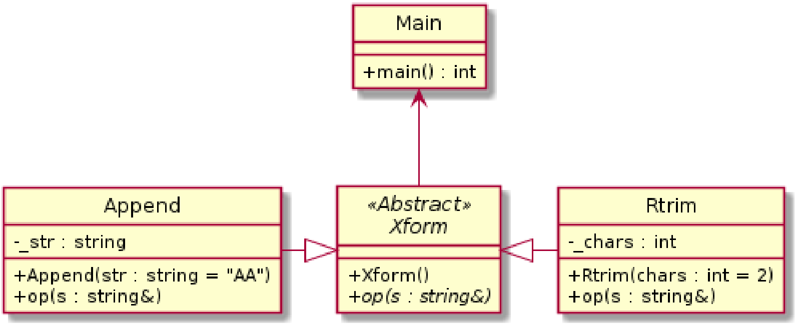
1 D	6 B	11 D
2 C	7 A	12 D
3 D	8 D	13 D
4 C	9 D	14 A
5 C	10 B	15 A

Free Response

Provide clear, concise answers to each question. Write only the code that is requested. You will NOT write an entire application! You need NOT copy any code provided to you - just write the additional code specified. You need NOT write `#include` statements - assume you have what you need.

While multiple questions may relate to a given application or class diagram, **each question is fully independent and may be solved as a stand-alone problem.** Thus, if you aren't able to solve a question, skip it until the end and move on to the next.

1. **(Polymorphism)** Consider the class diagram below. Note that class `Xform` is abstract.



Superclass **Xform** has 2 members:

- A **default constructor** that has an empty body
- A **pure virtual (abstract) void method op** that accepts a string reference parameter

a. {6 points} Write superclass `Xform` in file xform.h.

```
class Xform {
  public:
    Xform();
    virtual void op(std::string& s) = 0;
};
```

Two classes are derived from superclass Xform, each of which overrides method op to modify the string parameter *in place* (that is, op is a void method, and the parameter itself is modified). One of these is specified below.

- **Subclass Rtrim** has a non-default constructor with parameter chars (default value 2) that chains to Xform's constructor and sets its field _chars to chars, and an op method that overrides Xform::op and erases (trims) characters from the end of its parameter by _chars characters (that is, if the parameter s is "Hello" and _chars is 2, op changes s to "Hel"). Note that the string class has a substring method string substr (size_t pos = 0, size_t len = npos) const;

b. {6 points} Write subclass Rtrim as a unified file or as a .h and .cpp file pair, as you please.

```
class Rtrim : public Xform {
  public:
    Rtrim(int chars = 2);
    void op(std::string& s) override;
  private:
    int _chars;
};

Rtrim::Rtrim(int chars) : _chars{chars} { }

void Rtrim::op(std::string& s) {
    s = s.substr(0, s.size() - _chars);
}
```

The **main function** should:

i. Create a vector named `xforms` to manage instances of Xform's subclasses and add one instance of each.

- Add `Rtrim` first with a constructor parameter of $int .
- Add `Append` second with a constructor parameter of $str .

ii. Request a string from `std::cin` (a complete line or a word, as you please) and store it in the string variable `s` which you must first declare.

iii. Iterate through the vector, applying each op method **polymorphically** to string `s` in turn. That is, if the user entered "Fussbudget", your code would first pass it as the parameter to `op` on the $op1 object and that result as the parameter to `op` on the $op2 object, thus applying both transforms to string `s`.

iv. Finally, stream the transformed s to `std::cout`.

c. {8 points} Write the main function.

```
int main() {
    std::vector<Xform*> xforms{
        new Rtrim{3},
        new Append{"><(((('>"},
    };
    std::string s;
    std::cout << "Enter a string: ";
    std::getline(std::cin, s);
    for(Xform* x : xforms) x->op(s);
    std::cout << s << std::endl;
}
```

2. **(Parameters)** {4 points} Method `apply` accepts a C++ string named `change` as its single parameter. Write 4 declarations for this method using the parameter style specified.

a. Pass by value: `void apply(std::string change);`

b. Pass by reference: `void apply(std::string& change);`

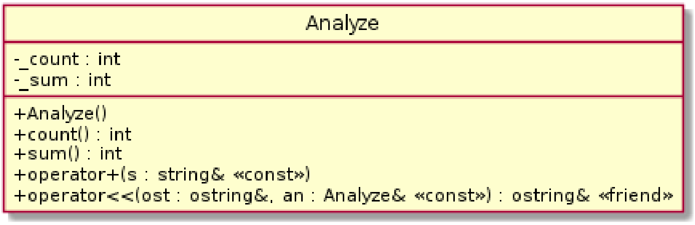
c. Pass by const reference: `void apply(const std::string& change);`

d. Pass by pointer: `void apply(std::string* change);`

3. **(Streams / Map)** {8 points} Write a C++ main function. Instance a standard map as variable `words` using a `std::string` as the key and an `int` as the value. Using a 3-term for loop, iterate over the program arguments (`argc` and `argv`), treating each as a filename. Open each filename (printing "Bad filename:" and the filename to the error channel if opening fails), then count the number of whitespace-separated words in each file, storing each filename (as the key) and its number of words (as the value) in map `words`. After counting the words in every file, iterate over map `words` (for example, with a for-each loop), printing the filenames in sorted order along with the number of words of text in each.

```
int main(int argc, char* argv[]) {
    std::map<std::string, int> words;
    std::string s;
    for(int i=1; i<argc; ++i) {
        std::string filename{argv[i]};
        std::ifstream ifs{filename};
        if(ifs) {
            int count = 0;
            while(ifs >> s) ++count;
            words[filename] = count;
        } else {
            std::cerr << "Bad filename: " << filename << std::endl;
        }
    }
    for(auto& [filename, count] : words) {
        std::cout << filename << " has " << count << " words" << std::endl;
    }
}
```

4. **(Operator Overloading)** {2 points} Class analyze collects the number (`_count`) and sum of the chars (`_sum`) in every string added to the object using the += operator. (This isn't a good class design, but is intended to check your ability to overload operators.)


Analyze
-_count : int
-_sum : int
+Analyze()
+count() : int
+sum() : int
+operator+(s : string& «const»)
+operator<<(ost : ostring&, an : Analyze& «const») : ostring& «friend»

a. {5 points} Overload operator += to increment `_count` and add the number of characters in the parameter to `_sum`. Thus, `Analyze an; an += "hello"; std::cout << an;` would output "1 words of 5 total characters".

```
Analyze& operator+=(std::string next) {
```

```
Analyze& operator+=(std::string next) {
    ++_count;
    _sum += next.size();
    return *this;
}
```

b. {5 points} Overload operator << to stream the number of elements (field `_count`) and their total size (field `_sum`). For example, if `_count` was 12 and `_size` was 128, your stream should be `12 elements of 128 characters`.

```
std::ostream& operator<<(std::ostream& ost, const Analyze& an) {
```

```
std::ostream& operator<<(std::ostream& ost, const Analyze& an) {
    return ost << an._count << " words of " << an._sum << " total characters";
}
```

5. **(Iterators, Algorithms)** {8 points} Given a vector named `states` containing strings, write code to print the index of the first "TX" and percentage of "TX" among states in the vector. You MUST correctly use `std::count`, `std::find`, and `std::distance` at least once each these calculations.

For example, if `states` contains "MS", "TX", "NY", "TX", "FL", "TX", then your code should print "First TX at index 1 and 50% are TX".

```
int index = std::distance(states.begin(),
                std::find(states.begin(), states.end(), "TX"));
std::cout << "First TX at index " << index << " and "
          << std::count(states.begin(), states.end(), "TX") * 100 / states.size()
          << "% are TX" << std::endl;
```

Bonus

Bonus: {+4 points} `std::map m` may be accessed using `operator[ ]` OR the `at` method. In no more than 2 *brief* sentences, explain how EACH approach works, emphasizing the main difference.

```
m[x] returns the value if present,
or adds the default value at x if not present and returns that.

m.at(x) returns the value if present
or throws std::out_of_range if not.
```