Exam #3 Practice #1

VOCAB KEY

1 Abstract Class

2 Iterator

3 Override

4 Abstraction

5 Method

6 Destructor

7 Container

8 Constructor

9 Encapsulation

10 Standard Template Library

11 Subclass

12 Polymorphism

13 Definition

14 Abstract Method

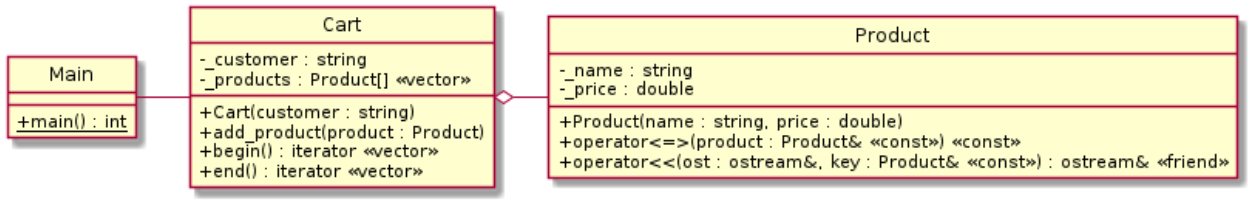
15 Declaration

MULTIPLE CHOICE KEY

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

Free Response

1. (class, operators, iomanip, find, iterators) Consider the class diagram below. Class Cart contains the name of the _customer (set by the constructor) and a vector of _products selected by the customer via method add_product.



a. {5 points} **In file product.h**, begin writing class Product. Write just the guard, class declaration, and fields.

```
#ifndef __PRODUCT_H
#define __PRODUCT_H

#include <iostream> // May be omitted

class Product {
  private:
   std::string _name;
   double _price;
```

b. {3 points} In file product.h, continue writing class Product. Write just the spaceship operator <=>. Specify that the compiler should generate default implementations for operators ==, !=, <, <=, >, >=. (Alternately, you may write the `inline` definitions for these 6 operators along with a `compare` method *declaration*. You do NOT need to implement `compare` for this question.)

```
auto operator<=>(const Product& rhs) const = default;
```

c. {2 points} In file product.h, continue writing class Product. Write just the `operator<<` declaration.

```
friend std::ostream& operator<<(std::ostream& ost, const Product& product);
```

d. {4 points} **In file product.cpp**, write just the implementation of `operator<<`. **Using I/O manipulators**, set the output stream to fixed floating point with 2-digit precision. Then stream out the product name and price, for example, if the name is "Dr. Pepper" and the price is 1.5, stream out "Dr. Pepper ($1.50)".

```
std::ostream& operator<<(std::ostream& ost, const Product& product) {
    ost << std::setprecision(2) << std::fixed;
    ost << product._name << " ($" << product._price << ")";
    return ost;
}
```

e. {3 points} **In file cart.h**, write just the field declarations for `_customer` and `_products`, including declaring them explicitly as private.

```
private:
  std::string _customer;
  std::vector<Product> _products;
```

f. {3 points} **In file cart.cpp**, write just the constructor. If the parameter is empty, throw a runtime error with the message "No customer name". Construct `_customer` from the parameter. If `_products` requires any constructor code, include it as well.

```
Cart::Cart(std::string customer) : _customer{customer} {
    if(customer.empty() throw std::runtime_error{"No customer name"};
}
```

g. {2 points} In file cart.cpp, write just the `add_product` method. Add the parameter to field `_products`.

```
void Cart::add_product(Product product) {
    _products.push_back(product);
}
```

h. {5 points} **In file main.cpp**, write the main function. Include classes `Cart` and `Product`, but assume all Standard Template Library files are already included. In the body of main,

- Instance a `Product` with the name "Dr. Pepper" at a price of $1.50. You will need this variable later, so name it `product`.
- Instance a `Cart` with the name "Exam".
- Add the Dr. Pepper to the cart.
- Then add 2 more products to the cart, "Texas flag" at $11.87 and "Fajitas" at $14.95.
- Sort the cart.
- Find `product` in the sorted cart using `std::find`.
- Using iterators, print the cart from the result of `std::find` to the end.

The output of this program would be

```
Dr. Pepper ($1.50)
Rattlesnake fajitas ($14.95)
Texas flag ($11.87)
```

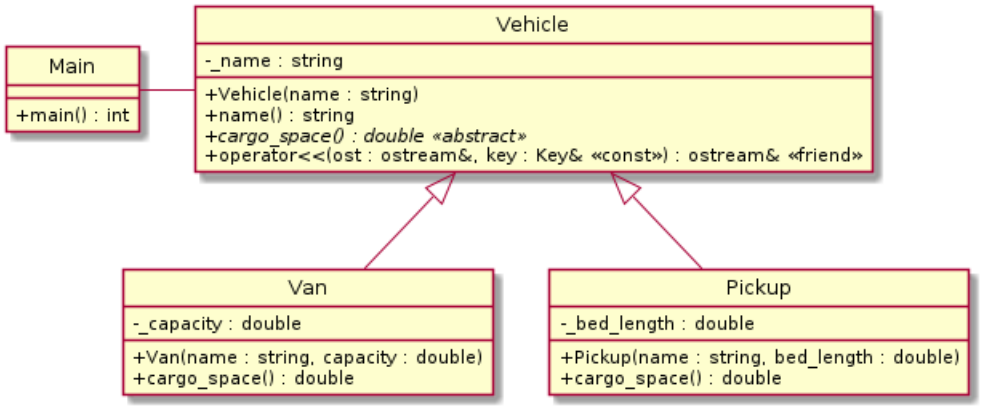
```
int main() {
   Product product{"Dr. Pepper",  1.50};

   Cart cart{"Exam"};
   cart.add_product(product);
   cart.add_product(Product{"Texas flag", 11.87});
   cart.add_product(Product{"Fajitas", 14.95});

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

   auto it = std::find(cart.begin(), cart.end(), product);
   while(it != cart.end()) std::cout << *(it++) << std::endl;
}
```

2. (polymorphism, abstract) Consider the class diagram below.



Vehicle is an abstact class with a name attribute and a cargo_space method (in *italics*) to calculate the cargo_space in ft³. Two classes derive from Vehicle and may be used polymorphically with it:

- Van specifies the cargo_space directly (attribute _capacity).
- Pickup specifies the _bed_length in ft, which may be multiplied by the 5.2 ft width and 1.9 ft depth to calculate its cargo_space.

a. {2 points} **In file vehicle.h**, write the declaration for method `cargo_space()` in class `Vehicle`. Note that `cargo_space()` will be called polymorphically in subclasses. If nothing is needed, write "N/A".

```
virtual double cargo_space() = 0;
```

b. {2 points} **In file vehicle.cpp**, write the definition for method `cargo_space()` in class `Vehicle`. If nothing is needed, write "N/A".

```
N/A
```

c. {2 points} **In file pickup.cpp**, write the constructor defintion for class `Pickup`. Ensure that construction of all fields in the superclass and subclass are properly specified.

```
Pickup::Pickup(std::string name, double bed_length)
  : Vehicle{name}, _bed_length{bed_length} { }
```

d. {2 points} In file pickup.h, write the declaration for method `cargo_space` in class `Pickup`. Ensure that the compiler will generate an error if the override doesn't happen.

```
double cargo_space() override;
```

e. **Write the main function.** You may assume any #include statements you need without writing them.

- Declare a vector named `vehicles`, with a "Short Bed" Pickup with bed_length 5.7 ft, a "Long Bed" Pickup with bed_length 14.5 ft, and a "Cargo" Van with capacity 164.5 ft³.
- Iterate over the motor vehicles, *polymorphically* printing the name and cargo_space of each vehicle (as supplied by the methods of the same names). Correctly written, the main function will produce the output shown. {5 points}

```
Short Bed (56.316 ft³)
Long Bed (143.26 ft³)
Cargo (164.5 ft³)
```

```
int main() {
   std::vector<Vehicle*> vehicles {
       new Pickup{"Short Bed", 5.7},
       new Pickup{"Long Bed", 14.5},
       new Van{"Cargo", 164.5},
   };

   for(Vehicle* v : vehicles)
       std::cout << *v << " ("
                 << v->cargo_space() << " ft³)\n";

}
```

3. {8 points} (file streams, string streams, arguments) **In file perimeters.cpp**, write a main method.
   - If a filename is not provided on the command line argument list, print "usage: " and the name of the executable and " <filename>" to standard error, then return a -1 error code to the operating system.
   - Open the filename for reading. If the open fails, print "Open failed" to standard error and return a -2 error code to the operating system.
   - Read newline-terminated line from the file until the end of the file is reached. With each line,
     - **Using a string stream**, parse each line into a word (the name of the shape) and then a sequence of side lengths (as an `int` each).
     - Print the name of the shape, a colon, and its perimeter. (The perimeter is just the sum of the side lengths.)
   - Verify that the file was read to the end. If not, print "Bad data file" to standard error then return a -3 error code to the operating system.

```
Data File              |  Output
=========              |  ======
```

```
Triangle 3 4 5         |  Triangle: 12
Rectangle 4 5 4 5      |  Rectangle: 18
Hexagon 2 2 2 2 2 2    |  Hexagon: 12
Irregular 4 6 8 10     |  Irregular: 28
```

```
int main(int argc, char* argv[]) {
   if(argc != 2) {
       std::cerr << "usage: " << argv[0] << " <filename>" << std::endl;
       return -1;
   }
   std::ifstream ifs{std::string{argv[1]}};
   if(!ifs) {
       std::cerr << "Open failed" << std::endl;
       return -2;
   }
   std::string line;
   std::string name;
   int side;
   while(std::getline(ifs, line)) {
       std::istringstream iss{line};
       iss >> name;
       int sum = 0;
       while(iss >> side) sum += side;
       std::cout << name << ": " << sum << std::endl;
   }
   if(!ifs.eof()) std::cerr << "Bad data file" << std::endl;
}
```

Bonus: Give one example each of a static cast and a dynamic cast, and *in one sentence each* explain what they do. {4 points}

Static cast verifies compatibility of the cast (as much as possible) at compile time:

```
A* a2 = static_cast<A*>(a);
```

Dynamic cast verifies polymorphic compatibility of the cast at runtime:

```
B* b3 = dynamic_cast<B*>(a);
```