CS100 Homework 7 Problem 1

Unit Test: Inheritance and Operator Overloading

Deadline: May 20 23:59

Answer the following questions according to the C++17 standard. For the compiler-generated special member functions, ignore whether they are `constexpr` or `noexcept`.

1. Read the following code.

```cpp
auto x = 42;
const auto y = x;
auto z = y;
auto &r = y;
auto m = r;
```

(a) (1 point) Write down the type of `x`.

(a) ______________

(b) (1 point) Write down the type of `y`.

(b) ______________

(c) (1 point) Write down the type of `z`.

(c) ______________

(d) (1 point) Write down the type of `r`.

(d) ______________

(e) (1 point) Write down the type of `m`.

(e) ______________

2. (5 points) Read the following code.

```cpp
class Dynarray {
  friend std::string foo();

 public:
  using size_type = std::size_t;
  static const size_type npos = -1;

  friend int fun(int);
};
```

Let `a` and `b` be two objects of type `Dynarray`. Which of the following is/are true?

A. `foo` is a function that takes no arguments and returns a `std::string`.

B. `foo` is a private friend of `Dynarray`, while `fun` is a public friend of `Dynarray`.

C. To access the type alias member `size_type`, use `a.size_type`.

D. The in-class initialization for `npos` is allowed, because `npos` is a `const static` member.

E. `a.npos` and `b.npos` refer to different variables.

F. `fun` is not a member function of `Dynarray`.

G. The `public` access modifier does not apply to the type alias member `size_type`. Type alias members are always public.

3. (5 points) Suppose `Derived` is a derived class of `Base`, with **some** members defined as follows. Which of the following statements is/are true?

```
class Derived : public Base {
 public:
  Derived() = default;
  Derived(const Derived &) = default;
  // other members are omitted.
};
```

A. An object of type `Derived` always contains a subobject of type `Base`, unless `Base` is an empty class that may be optimized out by **Empty Base Optimization**.

B. The default constructor of `Derived`, if not implicitly deleted, calls the default constructor of `Base` to initialize the base class subobject before initializing all its members.

C. A constructor of `Derived` may or may not call a constructor of `Base`. If no constructor of `Base` is called, the base class subobject is not contained in that object and the data members are not inherited.

D. The compiler-generated copy constructor for `Derived`, if not implicitly deleted, copies the base class subobject first, and then performs member-wise copy of the data members.

4. (5 points) Read the following code.

```
class Base {
 private:
  std::string someMemberA;
 public:
  ~Base() = default;
};
struct Derived1 : public Base {
  std::string someMemberB;
  ~Derived1() = default;
};
struct Derived2 : public Base {
  std::string someMemberC;
};
int main() {
  Base *bp = new Derived1{};
  delete bp;
}
```

Which of the following statements is/are true?

A. The compiler-generated destructor of `Base` is virtual.

B. `Derived2` does not have a destructor.

C. The destructor of `Derived1` is virtual, while the destructor of `Derived2` is not.

D. If the destructor of `Base` is virtual, the compiler-generated destructors of `Derived1` and `Derived2` are both virtual.

E. To make `delete bp` call the correct destructor corresponding to the dynamic type of `*bp`, the destructor of `Base` must be virtual.

F. `Base::someMemberA` is not inherited by `Derived1`, because it is a private member of `Base`.

5. (5 points) Read the following code.

```
class Base {
  std::string someMemberA;
};
class Derived : public Base {
  std::string someMemberB;
};
int main() {
  Base *bp = new Derived{};
  delete bp;
}
```

Which of the following statements is/are true?

A. The compiler-generated destructor of `Base` is non-virtual.

B. The compiler-generated destructor of `Derived` is non-virtual.

C. `delete bp;` will call the destructor of `Base`, because the static type of `*bp` is `Base`.

D. If we use smart pointers instead, i.e.

```
int main() {
  std::unique_ptr<Base> up = std::make_unique<Derived>();
}
```

The destructor of `Derived` will be called.

6. (5 points) Let `Derived` be a derived class of `Base`. Suppose `Base` has a member function declared as follows.

```
class Base {
 public:
  int foo(int a, const std::string &b) const;
};
```

Select the definitions of `Derived` in which the member function `foo` overrides `Base::foo`.

A.
```
class Derived {
 public:
  int foo(int a, const std::string &b) const;
};
```

B.
```
class Derived : public Base {
 public:
  int foo(int x, std::string const &y) const;
};
```

C.
```
class Derived : public Base {
 private:
  int foo(int, const std::string &) const;
};
```

D.
```
class Derived : public Base {
 public:
  virtual int foo(int a, const std::string &b) const;
};
```

E. None of the above.

7. (5 points) Let `Derived` be a derived class of `Base`. Suppose `Base` has a member function declared as follows.

```
class Base {
 public:
  virtual double calculate(const std::vector<double> &c, double x) const;
};
```

Select the definitions of `Derived` in which the member function `calculate` overrides `Base::calculate`.

A.
```
class Derived : public Base {
 public:
  using dvec = std::vector<double>;
  double calculate(dvec const &c, double xx) const;
};
```

B.
```
class Derived : public Base {
 public:
  virtual double calculate(const std::vector<double> &, double) const;
};
```

C.
```
class Derived : public Base {
 public:
  virtual double calculate(const std::vector<double> &, double);
};
```

D.
```
class Derived : public Base {
 private:
  virtual double calculate(const std::vector<double> &, double) const;
};
```

E. None of the above.

Note: The best way to know whether a function overrides a virtual function in the base class is to consult the compiler by adding the keyword `override`.

8. (5 points) Read the following code.

```
class Shape {
 public:
  virtual double area()  const = 0;
  virtual Shape *clone() const = 0;
  virtual ~Shape() = default;
};
class Rectangle : public Shape {
  double m_length, m_breadth;
 public:
  Rectangle(double l, double b) : m_length{l}, m_breadth{b} {}
  double area()      const override { return m_length * m_breadth; }
  Rectangle *clone() const override { return new Rectangle(*this); }
};
class Circle : public Shape {
  double m_radius;
 public:
  Circle(double r) : m_radius{r} {}
  double area()   const override { return 3.14159 * m_radius * m_radius; }
  Circle *clone() const override { return new Circle(*this); }
};
```

Which of the following is/are true?

A. This code does not compile, because the return types of `Rectangle::clone` and `Circle::clone` are not identical to that of `Shape::clone`.

B. Creating an object of type `Shape` is not allowed, because `Shape` is an abstract class.

C.
```cpp
Shape *sp = some_value();
auto p = sp->clone();
```
The type of `p` is determined at runtime according to the dynamic type of `*sp`. For example, `p` will be of type `Rectangle *` if `sp` points to a `Rectangle`.

D.
```cpp
Shape *sp = some_value();
Circle *cp = new Circle(2);
auto p1 = sp->clone();
auto p2 = cp->clone();
```
The type of `p1` is `Shape *` while the type of `p2` is `Circle *`.

To override a virtual function, the return type must be identical to **or covariant with** that of the corresponding function in the base class. This example shows a typical situation where the covariant-with" rule is made use of. A similar example can be found in *More Effective C++* Item 25.

9. (5 points) Read *Effective C++* Item 32. Select the best design that correctly represents the idea Penguins cannot fly".

A.
```cpp
class Bird {
 public:
  virtual void fly() = 0;
  // ...
};
class Penguin : public Bird {
 public:
  void fly() override {
    error("Penguins cannot fly!"); // reports a runtime error.
  }
  // ...
};
```

B.
```cpp
class Bird { /* fly() is not declared */ };
class FlyingBird : public Bird {
 public:
  virtual void fly() = 0;
};
class Penguin : public Bird { /* fly() is not declared */ };
```

C.
```cpp
class Bird {
 public:
  virtual void fly() = 0;
  // ...
};
class Penguin : public Bird {
 public:
  void fly() override = delete;
  // ...
};
```

D.
```cpp
class Bird {
 public:
```

```cpp
    virtual void fly() = 0;
  };
  class Penguin : public Bird {
    // fly() is not overridden
  };
```

Read *Effective C++* Item 34, and answer the following three questions.

Suppose we have three different kinds of dialogue boxes:

```cpp
class DialogueBox { // abstract base
  std::string caption;

 public:
  virtual void display();
  virtual ~DialogueBox() = default;
  // Other members ...
};
class Alert : public DialogueBox { // An alert box
  // ...
};
class Confirm : public DialogueBox { // The "yes-or-no" box
  // ...
};
class Progress : public DialogueBox { // "loading ..."
  // ...
};
```

All different kinds of dialogue boxes must support the `display()` interface.

10. (5 points) Suppose there is no well-defined default behavior for displaying a dialogue box if we don't know what kind of dialogue box it is (`Alert`, `Confirm` or `Progress`). Select the best design for `DialogueBox::display`.

   A.
   ```cpp
   class DialogueBox {
    public:
     virtual void display(); // without a definition
   };
   ```
   B.
   ```cpp
   class DialogueBox {
    public:
     virtual void display() = 0; // declared as pure virtual
   };
   ```
   C.
   ```cpp
   class DialogueBox {
    public:
     virtual void display() {} // defined and does nothing
   };
   ```
   D.
   ```cpp
   class DialogueBox {
    public:
     virtual void display() { // reports a runtime error
       error("Calling DialogueBox::display is not allowed.");
     }
   };
   ```

11. (5 points) Suppose we provide a default behavior for displaying a dialogue box, which draws a box of default size and displays its caption. This default behavior may be used when it is needed, but we

don't want it to be inherited automatically. `DialogueBox` should still be an abstract class. Select the acceptable designs to achieve this.

A.
```
class DialogueBox {
  std::string caption;
 public:
  virtual void display() = 0;
 protected:
  void defaultDisplay() {
    auto handle = Screen::drawBox(DEFAULT_SIZE);
    handle.setCaption(caption);
  }
};
```

B.
```
class DialogueBox {
  std::string caption;
 public:
  virtual void display() {
    if (typeid(*this) == typeid(DialogueBox))
      error("display() should not be called on an abstract class.");
    auto handle = Screen::drawBox(DEFAULT_SIZE);
    handle.setCaption(caption);
  }
};
```

C.
```
class DialogueBox {
  std::string caption;
 public:
  virtual void display() = 0;
};
void DialogueBox::display() {
  auto handle = Screen::drawBox(DEFAULT_SIZE);
  handle.setCaption(caption);
}
```

D. Declare `DialogueBox::display` as pure virtual without a definition. If the default behavior is needed, just copy-and-paste that piece of code.

12. (5 points) Which of the following statements regarding designs is/are true?

A. A class should be made abstract **only when** at least one of its member functions needs to be declared pure virtual.

B. If a class should be made abstract but none of its member functions should be pure virtual, we may declare its destructor as pure virtual with a definition, like this:

```
class ShouldBeAbstract {
 public:
  virtual ~ShouldBeAbstract() = 0;
};
ShouldBeAbstract::~ShouldBeAbstract() {
  // ...
}
```

C. Inheritance of a pure virtual function is **inheritance of interface**.

D. Inheritance of an impure virtual function is **inheritance of interface with a default implementation**.

E. To force an implementation of a member function to be inherited, the base class should define it as non-virtual.

13. (5 points) Read the following code. Suppose `DialogueBox` is an abstract class.

```
class DialogueBox { // abstract base
 public:
  virtual ~DialogueBox() = default;
  // ...
};
class Alert : public DialogueBox { // An alert box
  // ...
};
class Confirm : public DialogueBox { // The "yes-or-no" box
  // ...
};
class Progress : public DialogueBox { // "loading ..."
  // ...
};
int getDialogueType(const DialogueBox &dia) {
  auto ptr = &dia;
  if (dynamic_cast<const Alert *>(ptr))
    return 1;
  else if (dynamic_cast<const Confirm *>(ptr))
    return 2;
  else if (dynamic_cast<const Progress *>(ptr))
    return 3;
  return 0;
}
```

Which of the following statements is/are true?

A. Using the magic numbers `0`, `1`, `2` and `3` is a bad idea. It should be replaced with `enum`:

```
enum class DialogueType {
  base, alert, confirm, progress
};
DialogueType getDialogueType(const DialogueBox &dia) {
  auto ptr = &dia;
  if (dynamic_cast<const Alert *>(ptr))
    return DialogueType::alert;
  else if (dynamic_cast<const Confirm *>(ptr))
    return DialogueType::confirm;
  else if (dynamic_cast<const Progress *>(ptr))
    return DialogueType::progress;
  return DialogueType::base;
}
```

B. Here `dynamic_cast<const Alert *>(ptr)` can be replaced with the C-style cast `(const Alert *)ptr`.

C. `dynamic_cast<const Alert *>(ptr)` returns a null pointer if the cast fails, while `dynamic_cast<const Alert &>(dia)` throws `std::bad_cast` on failure.

D. Compared to the following implementation that uses a group of virtual functions, the original one that uses three `dynamic_cast`s is preferable, because it does not add any members to the classes.

```
enum class DialogueType {
  alert, confirm, progress
};
class DialogueBox {
```

```cpp
 public:
  virtual ~DialogueBox() = default;
  virtual DialogueType getType() const = 0;
  // ...
};
class Alert : public DialogueBox {
 public:
  DialogueType getType() const override { return DialogueType::alert; }
  // ...
};
class Confirm : public DialogueBox {
 public:
  DialogueType getType() const override { return DialogueType::confirm; }
  // ...
};
class Progress : public DialogueBox {
 public:
  DialogueType getType() const override { return DialogueType::progress; }
  // ...
};
auto getDialogueType(const DialogueBox &dia) {
  return dia.getType();
}
```

14. (5 points) Read the following code.

```cpp
class Uncopyable {
  Uncopyable(const Uncopyable &);
  Uncopyable &operator=(const Uncopyable &);
 public:
  Uncopyable() = default;
};
class A : public Uncopyable {
  int x;
};
class B {
  Uncopyable _;
  int x;
};
class C : Uncopyable {
  int x;
};
```

Which of the following is/are true?

A. The copy constructor and copy assignment operator for `A` and `C` are implicitly deleted.

B. The copy constructor and copy assignment operator for `B` are implicitly deleted.

C. `sizeof(A)` is probably equal to `sizeof(int)` due to Empty Base Optimization.

D. `sizeof(B)` is never equal to `sizeof(int)`.

E. A reference to `Uncopyable` can be bound to an object of type `C` without any explicit casts.

This example shows an interesting way of using inheritance. The `=delete` syntax for declaring deleted functions was not introduced until C++11, and this was a good way of disabling compiler-generated copy operations before C++11. Polymorphism is not used here, so `Uncopyable` does not need a virtual destructor. Inheritance of `Uncopyable` can even be private (as it is for `C`).

15. (5 points) Which of the following is/are true?

    A. If a binary operator `@` is defined as a non-member function, the expression `a @ b` is equivalent to `operator@(a, b)`.

    B. If a binary operator `@` is defined as a member function, the expression `a @ b` is equivalent to either `a.operator@(b)` or `b.operator@(a)`.

    C. Operator overloading may redefine the associativity of the operator, but cannot redefine the precedence of it.

    D. The expression `a += b` will be treated as `a = a + b` if there is no matching `operator+=` available.

16. (5 points) When writing overloaded operators, we should adhere to conventions and make them have similar behaviors as the built-in operators, unless there is an arguable reason not to do so. Select the behaviors that adhere to the conventions.

    A. The postfix increment operator (`x++`) returns a copy of the object before incrementation.

    B. The prefix increment operator (`++x`) returns a copy of the object after incrementation.

    C. The assignment operator (`=`), as well as compount assignment operators (`+=`, `-=`, etc.), should return a reference to the left-hand side object.

    D. The relational operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) should be consistent with each other. For example, `a < b` is true if and only if `a >= b` is false.

17. (5 points) Unlike the named functions, the name of an overloaded operator provides very little information about the exact behavior of it. Just consider `concat(a, b)` vs `a + b`, `compareByID(a, b)` vs `a < b`, `p1.getX()` vs `*p1`, `v1 * v2` vs `dot_product(v1, v2)`, etc. Therefore, it is not recommended to use operator overloading unless there is a unique, clear and natural definition for it.

    Among the following overloaded operators, select the abused ones that should be replaced with a named function.

    A.
    ```cpp
    struct Student {
      std::string name;
      int chinese_score;
      int math_score;
      int english_score;
    };
    bool operator<(const Student &lhs, const Student &rhs) {
      return lhs.math_score < rhs.math_score;
    }
    ```

    B.
    ```cpp
    struct Node {
      int vertexID;
      int dist;
      bool operator<(const Node &rhs) const {
        return dist > rhs.dist;
      }
    };
    ```

    C.
    ```cpp
    struct ReverseIterator { // iterate in the reverse direction
      int *current;
      ReverseIterator &operator++() {
        --current;
        return *this;
      }
    };
    ```

D.
```cpp
class Vector { // "vector" in linear algebra
    std::vector<double> elements;
   public:
    auto dimension() const {
      return elements.size();
    }
    Vector operator*(const Vector &rhs) const {
      assert(dimension() == rhs.dimension());
      auto ret = *this;
      for (std::size_t i = 0; i != dimension(); ++i)
        ret.elements[i] *= rhs.elements[i];
      return ret;
    }
  };
```

18. (5 points) Consider a class representing a vector in the 3-d space. Suppose we also store the ℓ_2-norm of it because it is used very frequently.

```cpp
class Vec3 {
  double x_, y_, z_;
  double l2_norm_;
};
```

We will read the coordinates of vectors from input, so let's define an input operator for convenience.

```cpp
std::istream &operator>>(std::istream &is, Vec3 &v) {
  std::cin >> v.x_ >> v.y_ >> v.z_;
  v.l2_norm_ = std::sqrt(v.x_ * v.x_ + v.y_ * v.y_ + v.z_ * v.z_);
}
```

Which of the following is/are true?

A. Suppose `v` is an object of type `Vec3`. `std::cin >> v` is equivalent to `operator>>(&std::cin, v)`.

B. Since the return type is `std::istream &`, several inputs can be chained as follows:

```cpp
Vec3 v; int ival; std::string s;
std::cin >> v >> ival >> s;
```

This does not lead to errors or undefined behaviors.

C. The first line of the function body should be changed to

```cpp
is >> v.x_ >> v.y_ >> v.z_;
```

D. This `operator>>` should be declared as a `friend` of `Vec3`.

E. This `operator>>` handles input failures correctly. If some error happens when reading `v.x_`, `v.y_` and `v.z_`, `v.l2_norm_` will be zero.

19. (5 points) Consider a class `Rational` representing a rational number. Here are some members.

```cpp
class Rational {
  int numerator;
  int denominator; // This should always be positive.

  void simplify();

 public:
  Rational(int x = 0) : numerator{x}, denominator{1} {}
  Rational(int num, int denom) : numerator{num}, denominator{denom} { simplify(); }
```

```
  bool operator==(const Rational &rhs) {
    return numerator == rhs.numerator && denominator == rhs.denominator;
  }
  Rational &operator++();
  Rational operator++(int a);
};
```

Let `r` be an object of type `Rational`, and `cr` be an object of type `const Rational`. Which of the following is/are true?

A. The expression `cr == r` compiles.

B. The expression `r == cr` compiles.

C. The expression `r == 1` compiles.

D. The expression `1 == r` compiles.

E. The expression `++r` is equivalent to `r.operator++()`, while `r++` is equivalent to `r.operator++(0)`.

F. The parameter `int a` in the last member function indicates that the number is incremented by `a`.

G.
```
Rational Rational::operator++(int) {
  auto tmp = *this;
  ++*this;
  return tmp;
}
```

This is a correct implementation for the postfix increment operator, as long as the implementation of the prefix version is correct.

20. (5 points) Which of the following is/are true?

A. The boolean logic operators `&&` and `||` are rarely overloaded, because the overloads cannot implement short-circuit evaluation.

B. The address-of operator `&` can be overloaded.

C. In case the address-of operator is overloaded, we can use `std::addressof(x)` to take the address of an object `x`.

D. Since there is no built-in `operator+` for pointers, we can define an `operator+` for `const char *` operands to concatenate strings:

```
std::string operator+(const char *lhs, const char *rhs) {
  return std::string(lhs) + rhs;
}
```