#A. Unit Test: Class Basics and Copy Control ✓ ✓

客观题

Unit Test: Class Basics, Copy Control and Smart Pointers

Answer the following questions according to the C++17 standard.

For all the compiler-generated member functions, we will ignore whether they are `constexpr` since we have not learned about it. Apart from move operations, we also ignore whether the generated functions are `noexcept`.

`Dynarray` is the class we wrote in Homework 5 Problem 3.

Part 0: Warm Up

1. Read the following code.

```
std::vector<std::string> wordList;
int n; std::cin >> n;
for (auto i = 0; i != n; ++i) {
  std::string word;
  std::cin >> word;
  wordList ____(1.a)_____;
}
for (____(1.b)_____)
  std::cout << word << std::endl;
```

Copy

(a) Fill in the blank (1.a) to append the string `word` to the end of `wordList`. Write it in a common way.

Note: `word` is not used anymore. Can you *move* it into the vector instead of copying it?

.push_back(word)

(b) Fill in the blank (1.b) to use **range-based for loops** to traverse `wordList`. You shoul

avoid copying strings and use `const` if possible. Write in a common way.

2. Let `Type` be some complete non-`const`, non-reference type. Let `i` and `j` be two integers in the interval $[0, 10]$. Let `p1`, `p2` and `a` be defined as follows.

```
Type a[10];
Type *p1 = a + i, *p2 = a + j;
```

Which of the following statements is/are true?

- ☐ A. In the declaration `Type *p1 = &a[i];`, `*` has the same meaning as that in the expression `*p1 = 42`.
- ☐ B. If either `i` or `j` is equal to `10`, the behavior of `p1 - p2` is undefined.
- ☐ C. If `i < j`, the behavior of `p1 - p2` is undefined.
- ☑ D. The result type of `p1 - p2` is `std::ptrdiff_t`.
- ☐ E. The result type of `p1 - p2` is an unsigned integer type.
- ☐ F. `p1 - p2` is equal to `(i - j) * sizeof(Type)`.

3. Try this code on your computer and understand the compiler's output. You may need to add `#include`s or a `main` function on your own.

```
void fun(int) = delete;
void fun(double) {
  std::cout << "fun(double) called.\n";
}
int ival = 42;
fun(ival);
```

Select the correct statement(s).

- ☐ A. `=delete` is not allowed here. It can only be used for default constructors or copy-control members.
- ☐ B. Functions marked as `=delete` will use a `delete` expression to destroy its arguments and deallocate the memory where they are stored.
- ☐ C. Functions marked as `=delete` do not participate in overload resolution. Therefore, `fun(ival)` calls `void fun(double)`.
- ☑ D. Functions marked as `=delete` still participate in overload resolution. `fun(ival)` still tries to call `void fun(int)`, which results in a compile-error.

4. Which of the following statements regarding type alias declarations is/are true?

- ☐ A. `typedef int[100] arr_t;` declares `arr_t` as an alias of `int [100]`.
- ☑ B. `using arr_t = int[100];` declares `arr_t` as an alias of `int [100]`.
- ☐ C. `#define arr_t int[100]` declares `arr_t` as an alias of `int [100]`.

Part 1: Class Basics

5. Which of the following statements is/are true?

- [ ] A. A default constructor is a constructor declared as `=default`.
- [ ] B. `Dynarray a = b;` is a copy-assignment to `a`. `=` here is the assignment operator.
- [x] C. The name of a constructor is the same as that of the class.
- [x] D. A constructor has no return type.

6. Read the following code.

```
struct Book {
  std::string m_title;
  std::string m_isbn;
  double m_price = 0.0;
};
```

Copy

Select the correct statement(s).

- [ ] A. `Book` has no constructors because it is a `struct`, not a `class`.
- [x] B. All the members of `Book` are `public`.
- [ ] C. `Book` has an implicitly-declared default constructor which, if used, default-initializes all its members, thereby default-initializing `m_price` with an indeterminant value.
- [x] D. `Book` has an implicitly-declared default constructor which, if used, does the same thing as `Book() {}`. The member `m_price` will be initialized with `0.0` as specified by the in-class default initializer `= 0.0`.

7. (Hard) Try this code on your computer and understand the output. You may add `#include`s on your own.

```
struct A {
  int arr[100];
};
void print(const A &a) {
  for (auto i = 0; i != 100; ++i)
    std::cout << a.arr[i] << ' ';
  std::cout << std::endl;
}
int main() {
  A a;
  A b{};
  print(a);
  print(b);
}
```

Copy

Select the correct statement(s).

[✓] C. A has an implicitly-defined default constructor which default-initializes every element of its array member arr. As a result, all the elements in arr have indeterminant values.

[✓] D. According to the output, all the elements in a.arr are default-initialized and have indeterminant values, while all the elements in b.arr are initialized with **zero**.

We said in lectures and recitations that for a class type, value-initialization is default-initialization, both of which call the default constructor of that class. From the code above we see that this is not always true. In fact, this is true only if the default constructor of that class type is non-trivial. In the example above, A has a trivial default constructor that does nothing, and in this case the value-initialization for A performs **zero-initialization**.

The explanations can be found here (value-initialization) and here (default-initialization). You don't have to read the entire pages. Just pay attention to the sections **Syntax** and **Explanation**, especially the paragraphs starting with "The effects of default/value initialization are: ......". You may ignore everything marked with "until C++11". You may ignore everything about enum or union.

8. Let the class Dynarray be defined with the following data members.

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
};
```

Which of the following is/are true?

[ ] A.

The default constructor

```
Dynarray() = default;
```

correctly initializes this object to be an empty dynamic array.

[ ] B.

The default constructor

```
Dynarray() {
  m_length = 0;
}
```

correctly initializes this object to be an empty dynamic array. The destructor can simply do delete[] m_storage; to release the memory it manages.

[✓] C.

```
Dynarray(std::size_t n) : m_length(n), m_storage(new int[m_length]{}) {}
```

Copy

[✓] D.

If the constructors are defined as follows:

```
Dynarray() : m_storage(nullptr), m_length(0) {}
Dynarray(std::size_t n) : m_storage(new int[n]{}), m_length(n) {}
```

Copy

The following destructor definition does not lead to memory leaks or undefined behaviors.

```
~Dynarray() { delete[] m_storage; }
```

Copy

☐ E.

If one of the constructors is defined as follows:

```
Dynarray(std::size_t n)
    : m_storage(n == 1 ? new int{} : new int[n]{}),
      m_length(n) {}
```

Copy

The following destructor definition does not lead to memory leaks or undefined behaviors.

```
~Dynarray() { delete[] m_storage; }
```

Copy

9. Read the following code.

```
class Book {
  std::string m_title;
  std::string m_isbn;
  double m_price = 0.0;
 public:
  void setPrice(double p) {
    m_price = p;
  }
  auto totalPrice(int n) {
    return n * m_price;
  }
};
```

Copy

Which of the following is/are true?

[✓] A. The return type of the member function `totalPrice` is `double`.

[✓] B. Logically, `totalPrice` should be callable on a `const` object because it is a read-only operation.

☐ C. Logically, `setPrice` should be callable on a `const` object.

☐ D. To make `totalPrice` callable on a `const` object, we should add the `const` keyword before the return type.

10. Let the class `Dynarray` be defined with the following members: yangzhy22022

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
 public:
  int &at(std::size_t n) const {
    return m_storage[n];
  }
};
```

Copy

Let `a` be a `Dynarray` and `ca` be a `const Dynarray`, and suppose both of them are non-empty. Which of the following is/are true?

- ☐ A. This `at` member function does not compile, because it is a `const` member function but returns a non-`const` reference to one of its members.
- ☑ B. Both `a.at(0)` and `ca.at(0)` compile.
- ☑ C. `++ca.at(0)` compiles, and increments the element indexed `0` by `1`.
- ☐ D. `++ca.at(0)` compiles but the behavior is undefined.
- ☐ E. `++ca.at(0)` does not compile.
- ☑ F. If we change the definition of the member `m_storage` to `int m_storage[10];`, this code does not compile.

Part 2: Copy Control

11. Let `X` be a class. Let `a` and `b` be two objects of type `X`.

(a) Select the situations where a **copy constructor** of `X` is required.

- ☑ A. `X c = a;`
- ☐ B. `a = b;`
- ☑ C. `X c(a);`
- ☑ D. `auto p = new X(b);`
- ☑ E.

```
void fun(X x);
fun(a);
```

Copy

- ☐ F.

```
void fun(X &xr);
fun(a);
```

Copy

(b) Select the situations where a **copy assignment operator** of `X` is required.

☐ C. `X c(a);`

☐ D. `auto p = new X(b);`

☐ E.

```
void fun(X x);
fun(a);
```

☐ F.

```
void fun(X &xr);
fun(a);
```

12. Read the following code.

```
class Book {
  std::string m_title;
  std::string m_isbn;
  double m_price = 0.0;
 public:
  Book() = default;
  Book(const Book &) = default;
};
Book bookA, bookB;
```

Which of the following statements is/are true?

☐ A. The class `Book` has two default constructors.

☐ B. `bookA = bookB;` uses the copy constructor of `Book`, because `Book` has a copy constructor but has no copy assignment operator.

☑ C. The compiler generates a copy constructor of `Book` as if it were defined as:

```
Book(const Book &other)
    : m_title(other.m_title), m_isbn(other.m_isbn), m_price(other.m_price) {}
```

☑ D. The class `Book` has a destructor that destroys all its data members in reverse order of declaration.

☐ E. The compiler generates a destructor of `Book` as if it were defined as:

```
~Book() {
  delete &m_price;
  delete &m_isbn;
  delete &m_title;
}
```

☐ F. The compiler generates a destructor of `Book` as if it were defined as:

```
~Book() {
  delete this;
}
```

☑ A. Let `i`, `j` and `k` be three `int`s. The built-in assignment expression `j = k` returns the left-hand side object `j`, making it possible to write the "chained" assignment `i = j = k`.

☐ B. By convention, the return type of the copy assignment operator should be `void`.

☐ C. By convention, the return type of the copy assignment operator for class `X` should be `X`, in order to be consistent with the behavior of the built-in assignment operator.

☑ D. By convention, the return type of the copy assignment operator for class `X` should be `X &`, in order to be consistent with the behavior of the built-in assignment operator and to avoid unnecessary copy.

14. Suppose some members of `Dynarray` are defined as follows.

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
 public:
  Dynarray(const Dynarray &other)
      : m_storage(new int[other.m_length]), m_length(other.m_length) {
    for (std::size_t i = 0; i != m_length; ++i)
      m_storage[i] = other.m_storage[i];
  }
  ~Dynarray() {
    delete[] m_storage;
  }
  // other members ...
};
```

Copy

Select the correct copy assignment operators, which should be *self-assignment safe* and should not lead to memory leaks or undefined behaviors.

☐ A.

```
Dynarray &operator=(const Dynarray &) = default;
```

Copy

☐ B.

```
Dynarray &operator=(const Dynarray &other) {
  delete[] m_storage;
  m_length = other.m_length;
  m_storage = new int[m_length];
  for (std::size_t i = 0; i != m_length; ++i)
    m_storage[i] = other.m_storage[i];
  return *this;
}
```

Copy

☐ C.

```
Dynarray &operator=(const Dynarray &other) {
  if (this != &other) {
    auto new_storage = new int[other.m_length];
    for (std::size_t i = 0; i != other.m_length; ++i)
```

Copy

```
  }
  return *this;
}
```

[✓] D.

```
Dynarray &operator=(const Dynarray &other) {
  auto new_storage = new int[other.m_length];
  for (std::size_t i = 0; i != other.m_length; ++i)
    new_storage[i] = other.m_storage[i];
  delete[] m_storage;
  m_storage = new_storage;
  m_length = other.m_length;
  return *this;
}
```

[✓] E.

```
// In class Dynarray
void swap(Dynarray &other) {
  std::swap(m_storage, other.m_storage);
  std::swap(m_length, other.m_length);
}
Dynarray &operator=(const Dynarray &other) {
  Dynarray(other).swap(*this);
  return *this;
}
```

15. Read the following code.

```
int ival = 42;
int &&rref = ival * 42;
const int &cref = ival * 42;
int &ref = rref;
++ref;
```

☐ A. `rref` is a reference of reference.

[✓] B. `rref` is bound to an rvalue `ival * 42`, thereby extending the lifetime of it.

☐ C. `const int &cref = ival * 42;` does not compile, because `cref` can only be bound to lvalues.

[✓] D. The fourth and the fifth lines compile, and do not lead to any errors.

[✓] E. An rvalue reference is an lvalue.

16. Read the following code.

```
class Book {
  std::string m_title;
  std::string m_isbn;
  double m_price = 0.0;
 public:
  void setTitle(std::string newTitle) {
    m_title = std::move(newTitle);
```

Which of the following statements is/are true? yangzhy22022

[✓] A.

The compiler generates a move constructor of Book as if it were defined as

```
Book(Book &&other) noexcept
    : m_title(std::move(other.m_title)),
      m_isbn(std::move(other.m_isbn)),
      m_price(std::move(other.m_price)) {}
```

[ ] B.

The compiler generates a move constructor of Book as if it were defined as

```
Book(Book &&other) noexcept
    : m_title(other.m_title), m_isbn(other.m_isbn), m_price(other.m_price) {}
```

Since `other` is an rvalue reference, `other.m_title`, `other.m_isbn` and `other.m_price` are rvalues. So there is no need to apply `std::move` to them.

[✓] C.

The compiler generates a move assignment operator of Book as if it were defined as

```
Book &operator=(Book &&other) noexcept {
  if (this != &other) {
    m_title = std::move(other.m_title);
    m_isbn = std::move(other.m_isbn);
    m_price = std::move(other.m_price);
  }
  return *this;
}
```

[✓] D.

Let `book` be an object of type `Book` and let `s` and `t` be two `std::strings`. `book.setTitle(s)` copy-initializes `newTitle`, while `book.setTitle(s + t)` move-initializes `newTitle`.

We said that if the parameter is not modified in the function, it should be declared as a reference-to-`const`. From the example above, we see that passing-by-value also has benefits if move operations are available.

This function will make lvalues copied and rvalues moved. Another way to achieve this is through a group of overloaded functions:

```
class Book {
  std::string m_title;
  std::string m_isbn;
  double m_price = 0.0;
 public:
  void setTitle(std::string &&newTitle) {
    m_title = std::move(newTitle);
  }
};
```

Read *Effective Modern C++* Item 41 for more about this.

17. Which of the following statements regarding `std::move` is/are true?

- ☐ A. Let `a` and `b` be two objects of type `T` that has a copy assignment operator. `a = std::move(b)` invokes the move assignment operator of `T`. If `T` does not have a move assignment operator, this results in a compile-error.
- ☑ B. `std::move` does not move anything. It is used to indicate that we want to treat an lvalue as an rvalue.
- ☐ C. `std::move(x)` moves the resources of `x` out of it.
- ☑ D. Since `std::move` is a very special function, it is recommended not to omit `std::` to avoid any possible name collisions.
- ☑ E. We cannot make any assumptions about the value of an object after it is moved. All we know is that it can be safely assigned to or destroyed.

Part 3: Smart Pointers

18. Which of the following statements regarding `std::unique_ptr` is/are true?

- ☐ A. Copying a `std::unique_ptr` transfers ownership of the managed object.
- ☐ B. Copying a `std::unique_ptr` makes the managed object copied.
- ☑ C. Moving a `std::unique_ptr` transfers ownership of the managed object.
- ☑ D. Copying a `std::unique_ptr` is not allowed.
- ☐ E. Default-initialization of a `std::unique_ptr` makes it having indeterminant values.

19. Which of the following statements regarding smart pointers is/are true?

- ☑ A. Copying a `std::shared_ptr` increments the corresponding reference counter by `1`.
- ☐ B. Copying a `std::shared_ptr` makes the managed object copied.
- ☑ C. When the reference counter reaches zero, `std::shared_ptr` destroys the object it manages and deallocates the memory.
- ☑ D. To manage "dynamic arrays" (typically obtained from `new T[n]`) with `std::unique_ptr`, we should use `std::unique_ptr<T[]>` instead of `std::unique_ptr<T>`.

递交

yangzhy22022

✓ 已认领

查看作业

成绩表

我的最近递交记录

帮助

状态
正在进行...

题目
2

开始时间
2023-4-14 5:00

截止时间
2023-4-28 23:59

可延期
24 小时

状态

评测队列

服务状态

开发

开源

API