CS100 Introduction to Programming
Fall 2023
Midterm Exam

Instructors: Ying Cao

Time: Dec 19th 13:00-14:40

INSTRUCTIONS

Please read and follow the following instructions:

- You have 100 minutes to answer the questions.
- You are not allowed to bring any electronic devices including regular calculators.
- You are not allowed to discuss or share anything with others during the exam.
- You should write the answer to every problem in the dedicated box **clearly**.
- You should write **your name and your student ID** as indicated on the top of **each page** of the exam sheet.

Name	
Student ID	

Please write your answers to the multiple choices questions in the following table.

(1)	(2)	(3)	(4)	(5)
(6)	(7)	(8)	(9)	(10)
(11)	(12)	(13)	(14)	(15)

1. (75 points) Multiple Choices

Each of the following questions has **one or more** correct choices.

You will get some proportion of a question's points if you choose a non-empty proper subset of its correct choices.

The questions marked "`[C]`" are based on the C17 standard (ISO/IEC 9899:2018). The questions marked "`[C++]`" are based on the C++17 standard (ISO/IEC 14882:2017).

(1) (5') `[C]` Read the following code. Select the correct statement(s).

```c
#define N 100

int a[N], b[N]; // (1)

void plus(int *a, int *b, int n) { // (2)
  for (int i = 0; i < n; ++i) // (3)
    a[i] += b[i];
}

void minus(int *a, int *b, int n) {
  for (int i = 0; i < n; ++i) // (4)
    a[i] -= b[i];
}
```

A. The line (1) will be rewritten as `int a[100], b[100];` by the preprocessor.

B. The identifier `a` at (1) and the one at (2) are the same variable.

C. The identifier `i` at (3) and the one at (4) are the same variable.

D. The elements in `a` at (1) are initialized to zero.

(2) (5') `[C]` The following function accepts a string and tests whether it is a palindrome.

```c
int is_palindrome(char *str) {
  size_t n = strlen(str);
  for (size_t i = 0, j = n - 1; i < j; ++i, --j)
    if (str[i] != str[j])
      return 0;
  return 1;
}
```

Select the correct statement(s).

A. Since `is_palindrome` does not modify the given string, the type of the parameter `str` should be `const char *`.

B. Since `sizeof(char) == 1`, `strlen(str)` can be replaced with `sizeof(str)`.

C. `is_palindrome("(())")` returns `1`.

D. The code involves undefined behavior if `str` is an empty string.

(3) (5') `[C]` Suppose we have defined the following `struct` represents a vector in linear algebra.

```
struct Vector {
  size_t dim;
  double *data;
};
```

Select the correct statement(s).

A. The following function sets a given `Vector` to empty.

```
void vector_init(struct Vector vec) {
  vec = (struct Vector){.dim = 0, .data = NULL};
}
```

To use this function to set `v` to empty, we can write `vector_init(v);`.

B. The following function sets a given `Vector` to $(0, \cdots, 0) \in \mathbb{R}^n$.

```
void vector_init(struct Vector *vec, size_t n) {
  vec = (struct Vector){.dim = n, .data = malloc(sizeof(double) * n)};
}
```

To use this function to set `v` to $(0, \cdots, 0) \in \mathbb{R}^n$, we can write `vector_init(&v, n);`.

C. The following function returns the sum of two `Vector`s.

```
struct Vector *vector_add(const struct Vector *lhs, const struct Vector *rhs) {
  size_t dim = lhs->dim < rhs->dim ? rhs->dim : lhs->dim;
  struct Vector result = {.dim = dim, .data = calloc(dim, sizeof(double))};
  for (size_t i = 0; i < lhs->dim; ++i)
    result.data[i] += lhs->data[i];
  for (size_t i = 0; i < rhs->dim; ++i)
    result.data[i] += rhs->data[i];
  return &result;
}
```

D. `free(v)`, where `v` is of type `struct Vector`, will call `free(v.data)` automatically to release the memory it has allocated.

E. None of the above.

(4) (5') `[C]` Which of the following statements is/are true?

A. Suppose `sizeof(int) == 4` and `sizeof(long long) == 8`.

```
int ival = 10000000;
long long llval = 1ll * ival * ival;
```

This code has undefined behavior because `ival * ival` overflows.

B. Suppose `i` is of type `int`. `printf("%d%d\n", i, i++)` has undefined behavior.

C. Suppose `f` is a `float`. `printf("%d", f)` has the same effect as `printf("%d", (int)f)`. For example, `printf("%d", f)` prints "3" if the value of `f` is `3.14`.

D.
```
char buffer[100];
scanf("%s", buffer);
```

If the input is longer than `100` characters, `scanf` will allocate a larger block of memory for `buffer` automatically.

(5) (5') `[C++]` Let `a` be of type `int [9][10]`. Select the pieces of code that makes `foo(a)` compile.

A. `void foo(int **a);`

B. `void foo(int (*a)[9]);`

C. `void foo(int (&a)[9][10]);`

D. `void foo(int (&a)[10][9]);`

(6) (5') `[C++]` We want to use a nested `vector` to represent a matrix. Which of the following statements is/are true?

A. `std::vector<std::vector<double>>` is a valid type.

B. `std::vector<std::vector<double>> matrix(n, m);`
`matrix` is initialized to be with `n` rows and `m` columns. That is, `matrix` is a vector that contains `n` elements, each of which is a vector containing `m` `double`s.

C. `std::vector matrix(n, std::vector(m, 0.0));`
The type of `matrix` is deduced to be `std::vector<std::vector<double>>`.

D. `std::vector matrix(n, std::vector(m, 0.0));`
`matrix` is initialized to be with `n` rows and `m` columns, with each element initialized to `0.0`.

(7) (5') `[C++]` Select the pieces of code in which the following range-based for loop can be used to traverse `a`.

```
for (const auto &x : a)
  do_something_with(x);
```

A. `int a[100]{};`

B. `int *a = new int[100]{};`

C. `std::vector<std::string> a(10, "hello");`

D. `std::string a = "world";`

(8) (5') [C++] Consider the class example Dynarray (which is also in Homework 5). Suppose its data members are defined as follows.

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

Select the correct statement(s). For choices C and D, suppose we want to add the member functions begin and end so that a Dynarray can be traversed using a range-based for loop.

A. m_length is private.

B. In the following member function, the expression m_storage has type int *const, because this has type const Dynarray *.

```
class Dynarray {
public:
  int operator[](std::size_t n) const { return m_storage[n]; }
};
```

C. The following is a good design for begin and end.

```
class Dynarray {
public:
  int *begin() { return m_storage;            }
  int *end()   { return m_storage + m_length; }
};
```

D. The following is a good design for begin and end.

```
class Dynarray {
public:
  int*       begin()       { return m_storage;            }
  int*       end()         { return m_storage + m_length; }
  const int* begin() const { return m_storage;            }
  const int* end()   const { return m_storage + m_length; }
};
```

(9) (5') [C++] Now we want to design a Book class representing a book. This may be used in a bookstore management program, where each book has its title, the ISBN number and a price.

```
class Book {
public:
  Book(const std::string &title_, const std::string &isbn_, double price_); // (*)
  Book() = default;

private:
  std::string title;
  std::string isbn;
  double price = 0.0;
};
```

Which of the following statements is/are true?

A. Book does not have a default constructor, since it has a user-declared constructor.

B. If the constructor (*) is defined as follows, title will be initialized after price.

```
Book::Book(const std::string &title_, const std::string &isbn_, double price_)
    : price{price_}, isbn{isbn_}, title{title_} {}
```

C. The compiler generates a default constructor for `Book`, in which the member `price` is initialized to `0.0`.

D. Since the move operations of `std::string` are cheap, we can rewrite the constructor (*) as

```
Book::Book(std::string title_, std::string isbn_, double price_)
    : title{std::move(title_)}, isbn{std::move(isbn_)}, price{price_} {}
```

so that rvalue string arguments are moved, not copied.

(10) (5') `[C++]` Consider the `Book` class again, with more member functions added to it.

```
class Book {
public:
  const std::string &get_title() { return title; }
  double total_price(int n) { return n * price; }
  bool operator==(const Book &rhs) const { return isbn == rhs.isbn; }
  bool operator!=(const Book &) const;
  // other members ...

private:
  std::string title;
  std::string isbn;
  double price = 0.0;
};
```

Which of the following statements is/are true?

A. The member functions `get_title` and `total_price` should be `const` member functions, because they do not modify the state of the object.

B. Let `b1` and `b2` be two `Book`s. The expression `b1 == b2` is effectively `b1.operator==(b2)`, where the parameter `rhs` is bound to `b2`.

C. The member function `operator==` does not compile, because it attempts to access `rhs.isbn` which is a private member of `rhs`.

D. The following is a reasonable design for the inequality operator.

```
bool Book::operator!=(const Book &rhs) const { return title != rhs.title; }
```

(11) (5') `[C++]` The function `create` in the following code is a *factory function* that creates an object dynamically and returns a smart pointer to it.

```
class Book {
private:
  Book(std::string t, std::string i, double p)
      : title{std::move(t)}, isbn{std::move(i)}, price{p} {}
  Book(const Book &) = default;
  Book(Book &&) = default;

public:
  static auto create(std::string title, std::string isbn, double price) {
    return std::unique_ptr(new Book(std::move(title), std::move(isbn), price));
  }
  // other members ... but with no public constructors
private:
  // data members as before
};
```

Select the correct statement(s).

A. In the function `create`, the `this` pointer has type `Book *`.

B. The return type of the function `create` is `std::unique_ptr`.

C. The only way for the user to create a `Book` is to call the factory function `create`, e.g. `Book::create("C++ Primer", "9780321714114", 75)`.

D. The function `create` can be rewritten as

```
// In class Book
static auto create(std::string title, std::string isbn, double price) {
  return std::make_unique<Book>(std::move(title), std::move(isbn), price);
}
```

(12) (5') [C++] Consider the following class representing a complex number $a + bi$, where $a, b \in \mathbb{R}$.

```
class Complex {
  double real;
  double imaginary;
public:
  Complex(double x) : real(x), imaginary(0) {} // (1)
  Complex(double a, double b) : real(a), imaginary(b) {}
  Complex operator-(const Complex &x) const;   // (2)
  Complex operator*(const Complex &x) const {
    return {
      real * x.real - imaginary * x.imaginary,
      real * x.imaginary + x.real * imaginary
    };
  }
  friend Complex operator+(const Complex &, const Complex &); // (3)
};
Complex operator+(const Complex &lhs, const Complex &rhs) {
  return {lhs.real + rhs.real, lhs.imaginary + rhs.imaginary};
}
```

Let `z` be an object of type `Complex`. Which of the following is/are true?

A. The function (2) is the unary minus operator (`-x`), because it only accepts one argument.

B. `0 + z` compiles, while `0 * z` does not compile.

C. If the function (1) is `explicit`, the expression `0 + z` does not compile.

D. The function (3) is a member of `Complex`.

(13) (5') The standard library has a function `std::copy_if`, which is similar to `std::copy` but accepts one more parameter `pred` that is a unary predicate. Only the elements for which `pred` returns true will be copied. Now we want to copy the integers less than a threshold `k` from a `std::vector<int>` into a new vector. Select the correct implementations.

A.
```
std::vector<int> work(const std::vector<int> &v, int k) {
  std::vector<int> result;
  std::copy_if(v.begin(), v.end(), result.begin(), [k](int x) { return x < k; });
  return result;
}
```

B.
```
struct LessThanK {
  int k;
  LessThanK(int k_) : k{k_} {}
```

```
    bool operator()(int x) const { return x < k; }
  };
  std::vector<int> work(const std::vector<int> &v, int k) {
    std::vector<int> result(v.size());
    std::copy_if(v.begin(), v.end(), result.begin(), LessThanK{k});
    return result;
  }
```

C.
```
bool less_than_k(int x, int k) {
  return x < k;
}
std::vector<int> work(const std::vector<int> &v, int k) {
  std::vector<int> result(v.size());
  std::copy_if(v.begin(), v.end(), result.begin(), less_than_k);
  return result;
}
```

D.
```
std::vector<int> work(const std::vector<int> &v, int k) {
  std::vector<int> result(v.size());
  std::copy_if(v.begin(), v.end(), result.begin(), [k](int x) { return x < k; });
  return result;
}
```

(14) (5') `[C++]` Suppose we have two classes `Item` and `DiscountedItem` defined as follows:

```
class Item {
public:
  Item(std::string name, double price) : m_name(std::move(name)), m_price(price) {}
protected:
  std::string m_name;
  double m_price = 0.0;
};
class DiscountedItem : public Item {
public:
  DiscountedItem(std::string name, double price, double disc); // (*)
private:
  double m_discount = 1.0;
};
```

Which of the following is/are true?

A. The private members of `Item`, if any, are not inherited by `DiscountedItem`.

B. The compiler generates a copy constructor for `DiscountedItem` as if it were defined as

```
// In class DiscountedItem
DiscountedItem(const DiscountedItem &other)
    : m_name(other.m_name), m_price(other.m_price), m_discount(other.m_discount) {}
```

C. The destructor of `DiscountedItem` will invoke the destructor of `Item` to destroy the base class subobject.

D. The constructor (*) can be defined as

```
DiscountedItem(std::string name, double price, double disc)
    : Item(std::move(name), price), m_discount(disc) {}
```

(15) (5') `[C++]` Let `Item` and `DiscountedItem` be defined as in question (14). Now we want to add a group of functions `net_price(n)`, which returns the net price of `n` items. The following function should print the correct net price according to the dynamic type of `item`.

```
void print_net_price(const Item &item, int n) {
  std::cout << "net price: " << item.net_price(n) << std::endl;
}
```

Select the one best way of defining `net_price`.

A.
```
// In class Item
        double net_price(int n) const { return n * m_price; }
// In class DiscountedItem
        double net_price(int n) const { return n * m_price * m_discount; }
```

B.
```
// In class Item
virtual double net_price(int n) const { return n * m_price; }
// In class DiscountedItem
        double net_price(int n) const { return n * m_price * m_discount; }
```

C.
```
// In class Item
        double net_price(int n) const { return n * m_price; }
// In class DiscountedItem
virtual double net_price(int n) const { return n * m_price * m_discount; }
```

D.
```
// In class Item
virtual double net_price(int n) const { return n * m_price; }
// In class DiscountedItem
        double net_price(int n) const override { return n * m_price * m_discount; }
```

2. (15 points) The "is-a" relationship

Public inheritance models the "is-a" relationship. Explain your understanding on this. Give some good and bad examples.

3. (10 points) Ref-qualified member functions

Apart from the `const` qualification, a member function can also be *ref-qualified*. The syntax of ref-qualifed member functions is as follows.

(1) *return_type function_name*(*parameter_list*) `const`$_{\text{optional}}$ `&` `noexcept`$_{\text{optional}}$`;`

(2) *return_type function_name*(*parameter_list*) `const`$_{\text{optional}}$ `&&` `noexcept`$_{\text{optional}}$`;`

Explanation:

(1) *lvalue ref-qualified* member function of a class `X`: The implicit object parameter has type `X &` (or `const X &`, if it is a `const` member function).

(2) *rvalue ref-qualified* member function of a class `X`: The implicit object parameter has type `X &&` (or `const X &&`, if it is a `const` member function).

For example:

```cpp
#include <iostream>
struct X {
  void foo() const &  { std::cout << "lvalue reference-to-const" << std::endl; }
  void foo()       && { std::cout << "rvalue reference"          << std::endl; }
};
int main() {
  X x;
  x.foo();            // prints "lvalue reference-to-const"
  const X &cx = x;
  cx.foo();           // prints "lvalue reference-to-const"
  std::move(x).foo(); // prints "rvalue reference"
}
```

The member functions `foo` in the example above can be seen as

```cpp
void X_member_foo(const X &self) {
  std::cout << "lvalue reference-to-const" << std::endl;
}
void X_member_foo(X &&self) {
  std::cout << "rvalue reference" << std::endl;
}
```

and the calls to `foo` can be seen as

```cpp
int main() {
  X x;
  X_member_foo(x);            // matches "const X &self"
  const X &cx = x;
  X_member_foo(cx);           // matches "const X &self"
  X_member_foo(std::move(x)); // matches "X &&self"
}
```

Note: unlike `const` qualification, *ref-qualification* does not change the type and properties of the `this` pointer: the type of `this` is still `X *` (or `const X *` if it is a `const` member function), and `*this` is always an lvalue expression.

The ref-qualification allows us to define different versions of a member function for lvalues and rvalues. Now consider the `Dynarray` class representing a dynamic array:

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
public:
  Dynarray sorted() const {
    Dynarray ret = *this;
    std::sort(ret.m_storage, ret.m_storage + ret.m_length);
    return ret;
  }
  // some other members ...
};
```

The member function `sorted()` returns a copy of `*this` but with all elements sorted in ascending order. However, if `sorted()` is called on a non-`const` rvalue (e.g. `std::move(a).sorted()`, or `Dynarray(begin, end).sorted()`), there is no need to copy the original array - we can directly sort the elements in `*this`, and return an rvalue reference to `*this` (obtained by `std::move`).

Use the `const`- and ref-qualifications of member functions to achieve this. Fill in the blanks for the return types and qualifications, and complete the function bodies.

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
public:
  _________ sorted() _________ {
    // YOUR CODE HERE ...




  }
  _________ sorted() _________ {
    // YOUR CODE HERE ...




  }
  // some other members ...
};
```