CS100 Introduction to Programming
Spring 2024
Midterm Exam

Instructors: Lan Xu, Yuexin Ma

Time: May 27th 8:15-9:55

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	

1. (75 points) Fill in the blanks

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) (10') The type of the string literal `"hello"` is `char [6]` in C, but `const char [6]` in C++.

(2) (5') [C] Read two integers from input, separated by an unknown sequence of whitespaces: `scanf("%d%d", &a, &b)`.

> **Solution:** Any other equivalent way is also correct.

(3) (5') [C] Given the following definition of a type representing a ghost in some game,

```
struct Ghost {
  int posX;
  int posY;
  char icon;
  bool isSmart;
};
```

allocate a block of memory to store `n` ghosts: `malloc( n * sizeof(struct Ghost) )`.

> **Solution:** Any other equivalent way is also correct.

(4) (10') [C] Suppose we have the following declarations.

```
int array[10] = {0};
int *p1 = &array[2], *p2 = array + 6;
```

Is the type of the expression `p1 - p2` an integer type or a pointer type? - Integer (write "Integer" or "Pointer").

The value of the expression `p1 - p2` is -4 (write "UB" if it is undefined behavior).

(5) (15') [C++] Suppose `s` is an object of type `std::string`. Use `new` to create a `std::string` that is move-constructed from `s`:

```
auto t = new std::string(std::move(s));
```

The type of `t` is `std::string *`.

To destroy this string and deallocate the memory: `delete t;`

(6) (15') The following code reads `n` strings from input, and stores them into a `std::vector<std::string>`. Fill in the blank with **the best way** of appending the string `s` to the end of `strings`.

```
std::vector<std::string> strings;
for (auto i = 0; i != n; ++i) {
  std::string s;
  std::cin >> s;
  strings.push_back(std::move(s));
}
```

Suppose `k` is a variable of type `std::size_t`. Use a *range-based `for` loop* to traverse `strings` to count how many of the strings have length **less than `k`**. Fill in the blank with **the best answer**.

```
int cnt = 0;
for (const auto &s : strings) {
  if (s.size() < k)
```

```
    ++cnt;
}
```

`std::count_if` is a standard library function that accepts **a sequence** and **a unary predicate**, and returns the number of elements in the sequence for which the predicate returns `true`. Use `std::count_if` to rewrite the code above.

```
int cnt = std::count_if(
  ____________________________strings.begin()____________________________,
  ____________________________strings.end()____________________________,
  _______[k](const std::string &s) { return s.size() < k; }_______
);
```

Solution: Any other equivalent way is also correct.

(7) Read the following code.

```
class Matrix {
  std::vector<std::vector<double>> mData;

public:
  explicit Matrix(std::vector<std::vector<double>> data) : mData{std::move(data)} {}

  double &operator()(std::size_t row, std::size_t col) {
    return mData[row][col];
  }

  double operator()(std::size_t row, std::size_t col) const {
    return mData[row][col];
  }

  /* (!!) */ friend Matrix operator-(const Matrix &) _____________;

  Matrix operator+(const Matrix &rhs) ____const____;

  Matrix &operator+=(const Matrix &rhs) _____________;
};
```

i. (5') For each blank above, write a "`const`" where you think is necessary.

ii. (5') Let `a` and `b` be two objects of type `Matrix`. Write an expression that uses the function marked "(!!)": _____-a_____.

iii. (5') Let `a` be an object of type `Matrix`. How will you access the element on the `i`-th row and `j`-th column of `a` (the private members are not accessible)? - ___a(i, j)___.

2. (20 points) Dynamic array

Consider the `Dynarray` class, which represents a dynamic array. It is the example given in the lectures, recitations and homework assignments.

```
class Dynarray {
public:
  Dynarray() = default;

  /* (a) */ explicit Dynarray(std::size_t n) : m_length{n}, m_storage{new int[n]{}} {}

  /* (b) */ Dynarray(std::size_t n, int x) : Dynarray(n) {
    std::fill_n(m_storage, n, x);
  }

  /* (c) */ Dynarray(const int *begin, const int *end) : Dynarray(end - begin) {
    std::copy(begin, end, m_storage);
  }

  /* (d) */ Dynarray(const Dynarray &other);

  Dynarray(Dynarray &&other) noexcept
      : m_length{other.m_length}, m_storage{other.m_storage} {
    other.m_length = 0;
    other.m_storage = nullptr;
  }

  ~Dynarray() { delete[] m_storage; }

  void swap(Dynarray &other) noexcept {
    std::swap(m_length, other.m_length);
    std::swap(m_storage, other.m_storage);
  }

  /* (e) */ Dynarray &operator=(Dynarray other) noexcept {
    swap(other);
    return *this;
  }

  // other members ...

private:
  std::size_t m_length{0};
  int *m_storage{nullptr};
};
```

(1) (5') The way that the constructors (b) and (c) are defined is interesting: By writing `Dynarray(n)` and `Dynarray(end - begin)`, they *delegate* part of the initialization work to the constructor (a). Such constructors are called *delegating constructors*.

Can you define the constructor (d) as a *delegating constructor* too? Fill in the blank below.

```
Dynarray::Dynarray(const Dynarray &other)
    : Dynarray(__other.m_storage, other.m_storage + other.m_length__) {}
```

(2) (5') What kind of assignment operator is (e)? Why?

> **Solution:** It is both a copy assignment operator and a move assignment operator.
>
> When assigning a `Dynarray` object with an expression *`expr`*, the parameter `other` will be initialized from *`expr`*, which performs copy-construction if *`expr`* is an lvalue and move-construction if it is an rvalue.

(3) (10') Define a group of member functions `begin` and `end` for `Dynarray`, so that `Dynarray` can be traversed using a *range-based `for` loop*.

Write your answers by filling in the blanks below. If any of the functions is not needed, leave the blanks blank.

```
class Dynarray {
public:
    int *       begin() ________{ return m_storage; }________
  const int *   begin() ____const { return m_storage; }_____
                begin() ____________________________________
    int *       end()   ___{ return m_storage + m_length; }___
  const int *   end()   _const { return m_storage + m_length; }_
                end()   ____________________________________

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

3. (20 points) Understanding compiler's diagnostics

In each part of this question, you will be given **a code snippet** and **the compiler's diagnostics on the code**. The code is compiled using GCC 13 with `-Wall -Wpedantic -Wextra -std=c++17`. Explain **what the diagnostics mean** and **what is wrong with the code**.

Note: Some of you may be MSVC or Clang users and are unfamiliar with GCC's diagnostics, but don't worry. We are sure that the following diagnostics are easy to understand as long as you know the mistakes in the code.

(1) (10') Code:

```
3  class Action {
4    // ...
5  };
6
7  void runAction(std::unique_ptr<Action> action) {
8    // ...
9  }
10
11 void runTool() {
12   auto action = std::make_unique<Action>(/* ... */);
13   runAction(action);
14 }
```

Compiler's diagnostics:

```
tool.cpp: In function ‘void runTool()’:
tool.cpp:13:12: error: use of deleted function ‘std::unique_ptr<_Tp, _Dp>::unique_ptr(cons
t std::unique_ptr<_Tp, _Dp>&) [with _Tp = Action; _Dp = std::default_delete<Action>]’
   13 |   runAction(action);
      |   ~~~~~~~~~^~~~~~~~
In file included from /usr/include/c++/13/memory:78,
                 from tool.cpp:1:
/usr/include/c++/13/bits/unique_ptr.h:522:7: note: declared here
  522 |       unique_ptr(const unique_ptr&) = delete;
      |       ^~~~~~~~~~
tool.cpp:7:40: note:   initializing argument 1 of ‘void runAction(std::unique_ptr<Action>)’
    7 | void runAction(std::unique_ptr<Action> action) {
      |                ~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~
```

The compiler says that the code uses a deleted function. What function is it? Why is it used by the code? How will you fix the error **if you can only modify the code in `runTool`**?

Solution: The deleted function this code uses is the copy constructor of `std::unique_ptr`. It is used because on line 13 `action` is an lvalue and is passed by value.

To fix the error, we may pass `std::move(action)` as the argument, or we may just write `runAction(std::make_unique<Action>(/* ... */))` directly.

(2) (10') Code: Suppose `MatchFinder` is a default-constructible class type.

```
class ClassHandler {
  MatchFinder &mFinder;
  // ...

public:
  explicit ClassHandler(MatchFinder &finder) : mFinder{finder} {
    // ...
  }
};
class Tool {
  ClassHandler mClassHandler;
  MatchFinder mFinder;
  // ...

public:
  Tool() : mFinder{}, mClassHandler{mFinder} {
    // ...
  }
};
```

Compiler's diagnostics:

```
tool.cpp: In constructor 'Tool::Tool()':
tool.cpp:17:15: warning: 'Tool::mFinder' will be initialized after [-Wreorder]
   17 |   MatchFinder mFinder;
      |               ^~~~~~~
tool.cpp:16:16: warning:   'ClassHandler Tool::mClassHandler' [-Wreorder]
   16 |   ClassHandler mClassHandler;
      |                ^~~~~~~~~~~~~
tool.cpp:21:3: warning:   when initialized here [-Wreorder]
   21 |   Tool() : mFinder{}, mClassHandler{mFinder} {
      |   ^~~~
```

What is the compiler warning against? Is there a real mistake in the code? How will you eliminate the warning and fix the mistake?

Solution: The warning says that the `mFinder` will be initialized after `mClassHandler`, but `mFinder` appears before `mClassHandler` in the constructor initializer list.

When initializing `mClassHandler` using `mFinder`, `mFinder` is uninitialized, so the behavior is undefined.

To eliminate the warning and fix the mistake, declare the member `mFinder` before `mClassHandler`.

4. (15 points) Inheritance

Suppose we have three different kinds of dialogue boxes.

```
class DialogueBox {
  std::string caption;
  // ...

public:
  virtual void display();
  virtual ~DialogueBox() = default;
  // ...
};

class Alert : public DialogueBox { /* ... */ } // An alert box

class Confirm : public DialogueBox { /* ... */ } // The "yes-or-no" box

class Progress : public DialogueBox { /* ... */ } // "loading ..."
```

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

(1) (5') Suppose there is **no** well-defined "*default*" behavior for displaying a dialogue box. Select **the (unique) best design** for `DialogueBox::display`.

A.
```
class DialogueBox {
public:
  virtual void display(); // without a definition
};
```

B.
```
class DialogueBox {
public:
  virtual void display() = 0; // declared as pure virtual
};
```

C.
```
class DialogueBox {
public:
  virtual void display() {} // defined and does nothing
};
```

D.
```
class DialogueBox {
public:
  virtual void display() { // reports a runtime error
    error("Calling DialogueBox::display is not allowed.");
  }
};
```

(2) (10') Suppose now there **is** a "*default*" behavior for displaying a dialogue box: Draw a box of default size, and then display the caption on it. It may be implemented as follows:

```
auto handle = Screen::drawBox(DefaultSize);
handle.setCaption(caption);
```

This default behavior should be provided by the base class `DialogueBox`, but

- we still want `DialogueBox` to be an abstract base class, and

- we don't want this default behavior to be inherited *automatically* and *silently*. That is, if a derived class wants to adopt this default behavior, it must write *something*.

How will you design `DialogueBox` to achieve this goal? Write your answer in the code block below.

Note: For your convenience, you only need to write "`// DB`" to represent the code of the default behavior.

```
class DialogueBox {
  std::string caption;
  // ...

public:
  virtual void display()



};
```

Suppose `Progress::display` wants to adopt the default behavior. Implement it in the code block below.

```
class Progress : public DialogueBox {
public:
  void display() override {



  }
};
```

Solution: One possible way: a `protected` non-`virtual` function.

```
class DialogueBox {
  std::string caption;
  // ...

public:
  virtual void display() = 0;

protected:
  void defaultDisplay() {
    // DB
  }
};
```

```
class Progress : public DialogueBox {
public:
  void display() override {
    defaultDisplay();
  }
};
```

Another way: a pure `virtual` function with a definition.

```
class DialogueBox {
  std::string caption;
  // ...

public:
  virtual void display() = 0;
};

void DialogueBox::display() {
  // DB
}
```

```
class Progress : public DialogueBox {
public:
  void display() override {
    DialogueBox::display();
  }
};
```