CS100 Computer Programming
Spring 2023
Midterm Exam

Instructors: Lan Xu, Yuexin Ma

Time: May 15th 8:15-10:15

INSTRUCTIONS

Please read and follow the following instructions:

- You have 120 minutes to answer the questions.
- You are not allowed to bring any books or electronic devices including regular calculators.
- You are not allowed to discuss or share anything with others during the exam.
- You may bring two pieces of A4-sized double-sided cheat sheets.
- 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)
CD	ABCE	BC	BC	C
(6)	(7)	(8)	(9)	(10)
AE	B	AB	AD	ACDE
(11)	(12)	(13)	(14)	(15)
BDF	AD	AB	AB(C)	ACD
(16)	(17)	(18)	(19)	(20)
C	AD	B	BC	B

1. (60 points) Multiple Choices

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

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

The questions marked "`[C]`" are based on C17. The questions marked "`[C++]`" are based on C++17.

(1) (3') `[C]` Read the following code.

```c
#include <stdio.h>
int x = 42;                // (1)
int incre(int x) {         // (2)
  return x + 1;
}
int main(void) {
  int x;                   // (3)
  x = 15;
  if (x == 15) {
    int x = incre(10);
    printf("%d\n", x++);   // (4)
  }
  printf("%d\n", incre(x)); // (5)
  return 0;
}
```

A. The `x` in (1) and (2) refer to the same variable.

B. In (3), the value of `x` is `42` because it was initialized to `42` in (1).

C. The value printed by (4) is `11`.

D. The value printed by (5) is `16`.

E. The value printed by (5) is `13`.

(2) (3') `[C]` Which of the following statements is/are true?

A. A string is an array (using either static memory or dynamic memory) of `char`s with a null character `'\0'` at the end.

B. To obtain the length of a string, use `strlen(s)`.

C.
```c
char *cp1 = "hello";
char cp2[] = "hello";
```
`cp1` is a pointer that points to the string literal `"hello"`. `cp2` is an array whose elements are copied from the string literal `"hello"`.

D. With `cp1` and `cp2` defined as above, both `cp1[0] = 'b';` and `cp2[0] = 'b';` correctly modify the first character of that string.

E. With `cp2` defined as above, `sizeof(cp2)` is equal to `6`.

(3) (3') `[C]` Suppose we have the following `struct` that stores the basic information of a student.

```c
struct student {
  bool gender; // use true/false to represent male/female
  char *name;
  int id;
};
```

We want to define a function `student_new` that creates a `student` in dynamic memory and returns a pointer to it, and a function `student_delete` that destroys a `student` and deallocates its memory. It is unspecified whether `student_new` should copy the student's name, but the behaviors of `student_new` and `student_delete` should match with each other. Select the correct definitions that do not lead to compile-errors, memory leaks or undefined behaviors.

A.
```c
struct student *student_new(bool is_male, char *name, int i) {
  struct student *student
      = malloc(sizeof(bool) + sizeof(char *) + sizeof(int));
  student->gender = is_male;
  student->name = name; // The name is not copied.
  student->id = i;
  return student;
}
void student_delete(struct student *student) {
  free(student);
}
```

B.
```c
struct student *student_new(bool is_male, char *name, int i) {
  struct student *student = malloc(sizeof(*student));
  *student = (struct student) {
    .gender = is_male,
    .name = name, // The name is not copied.
    .id = i
  };
  return student;
}
void student_delete(struct student *student) {
  free(student);
}
```

C.
```c
struct student *student_new(bool is_male, char *name, int i) {
  struct student *student = malloc(sizeof(struct student));
  student->gender = is_male;
  student->name = malloc(strlen(name) + 1);
  strcpy(student->name, name); // The name is copied.
  student->id = i;
  return student;
}
void student_delete(struct student *student) {
  free(student->name);
  free(student);
}
```

D.
```c
struct student *student_new(bool is_male, char *name, int i) {
  struct student *student
      = malloc(sizeof(bool) + sizeof(int) + strlen(name) + 1);
  student->gender = is_male;
  strcpy(student->name, name);
  student->id = i;
  return student;
}
void student_delete(struct student *student) {
  free(student->name);
  free(student);
}
```

(4) (3') `[C]` Suppose the following code appears very frequently in our program.

```
// ptr is a const char *, pointing to some position in a string.
int depth = 0;
while (*ptr != '\0' && !some_condition(*ptr, depth))
  ++ptr;
```

Now we want to write a function to do this job, in order to make our code more readable and less repetitive. Select the correct designs. (Both the function definition and the usage should be correct.)

A.
```
void move_ptr(const char *ptr) {
  int depth = 0;
  while (*ptr != '\0' && !some_condition(*ptr, depth))
    ++ptr;
}
```
To use this function: `move_ptr(ptr);`

B.
```
void move_ptr(const char **ptr) {
  int depth = 0;
  while (**ptr != '\0' && !some_condition(**ptr, depth))
    ++*ptr;
}
```
To use this function: `move_ptr(&ptr);`

C.
```
const char *move_ptr(const char *ptr) {
  int depth = 0;
  while (*ptr != '\0' && !some_condition(*ptr, depth))
    ++ptr;
  return ptr;
}
```
To use this function: `ptr = move_ptr(ptr);`

D.
```
void move_ptr(const char **ptr) {
  int depth = 0;
  while (**ptr != '\0' && !some_condition(**ptr, depth))
    ++*ptr;
}
```
To use this function: `move_ptr(*ptr);`

(5) (3') `[C]` Which of the following statements is/are true?

A. `printf(NULL)` has the same effect as `printf("%p", NULL)`, which prints something indicating a null pointer value (possibly `"null"`).

B. 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`.

C.
```
int *ptr = NULL;
free(ptr);
```
This code does not lead to undefined behaviors.

D.
```
int ival = 10000000;
long long llval = ival * ival;
```

Suppose `sizeof(int) == 4` and `sizeof(long long) == 8`. This code does not have undefined behaviors.

Solution:

A `printf` expects the first argument to be a string. Passing `NULL` to it is undefined behavior.

B If the conversion specifier does not match the argument type, the behavior is undefined.

C `free(NULL)` does not do anything.

D Overflow happens when calculating `ival * ival`, no matter what type of variable is the result stored into.

(6) (3') `[C]` Read the following code.

```
void mystery(char *str, int n) {
  if (n <= 1) {
    return;
  } else if (n == 2) {
    char tmp = str[0];
    str[0] = str[1];
    str[1] = tmp;
    return;
  } else {
    mystery(str, n / 2);
    mystery(str + n / 2, n - n / 2);
  }
}

char str[] = /* SOME VALUE */;
mystery(str, strlen(str));
printf("%s\n", str);
```

Which of the following statements is/are true?

A. If `/* SOME VALUE */` is `"01234567"`, the code will print `10325476`.

B. If `/* SOME VALUE */` is `"01234567"`, the code will print `76543210`.

C. If `/* SOME VALUE */` is `"012345678"`, the code will print `103254768`.

D. If `/* SOME VALUE */` is `"012345678"`, the code will print `876543210`.

E. If `/* SOME VALUE */` is `"012345678"`, the code will print `103254687`.

F. If `/* SOME VALUE */` is `"012345678"`, the code will print less than 9 characters, because the terminating character `'\0'` is moved to somewhere in the middle of that string.

Solution: Be careful with the case `n == 3`.

(7) (3') `[C]` A student writes these functions to remove all digits from a C string:

```
void remove_one_char(char *cur) {
  while (*cur != '\0') {
    *cur = *(cur + 1);
    ++cur;
  }
}
void my_awesome_remove_digits(char *str) {
  while (*str != '\0') {
    if (isdigit(*str))
      remove_one_char(str);
    str++;
  }
}

char str[] = "0a12b3c4d";
my_awesome_remove_digits(str);
```

After the call to `my_awesome_remove_digits`, what is the value of `strlen(str)`?

A. 4.

B. 5.

C. 9.

D. This call to `my_awesome_remove_digits` will trigger a runtime error. Memory past the end of string `'\0'` is accessed.

Solution: To make that function correct, add an `else` before `str++;`.

(8) (3') `[C++]` Select the pieces of code in which the call to `fun` compiles.

A.
```
void fun(int a[100]);
int ival = 42;
fun(&ival);
```

B.
```
void fun(int a[100]);
int a[20];
fun(a);
```

C.
```
void fun(int (&a)[100]);
int a[100];
fun(&a);
```

D.
```
void fun(int (*a)[100]);
int a[100];
fun(*a);
```

(9) (3') `[C++]` Read the following code.

```
std::string foo(const std::string &str) {
  std::string result(str.size(), '*');
  for (int i = 0, j = str.size() - 1; i <= j; i++, j--) {
    result[i] = str[j];
    result[j] = str[i];
  }
  return result;
}
std::string func(const std::string &str, const std::string &part) {
  std::size_t pos = str.find(foo(part));
```

```
    if (pos == std::string::npos) {
      return {};
    } else {
      return str.substr(0, pos)
             + foo(str.substr(pos, part.size()))
             + str.substr(pos + part.size(), str.size() - pos - part.size());
    }
  }

  std::cout << func("aabbcbcaab", "bc") << std::endl;
```

A. `foo(str)` always returns a reversed `str`.

B. The string returned by `foo(str)` may contain a `'*'` if the length of `str` is odd.

C. The code prints `aabcbbcaab`.

D. The code prints `aabbbccaab`.

Solution: `foo(str)` always returns a reversed `str` even if the length of `str` is odd, because the loop condition is `i <= j`.
`func(str, part)` returns a copy of `str` with the first occurrence of reversed `part` reversed.

(10) (3') `[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";`

E. `void foo(int (&a)[10]) { /* ... */ }`

F. `void foo(int a[10]) { /* ... */ }`

(11) (3') `[C++]` Read the following code.

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

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 `operator[]` function does not compile, because it is a `const` member function but returns a non-`const` reference to one of its members.

B. `ca[0]` compiles.

C. `a[0]` does not compile, because a `const` member function can only be called on a `const` object.

D. `++ca[0]` compiles, and increments the element indexed `0` by `1`.

E. `++ca[0]` compiles, but the behavior is undefined.

F. If we change the type of the member `m_storage` to `int[10]`, this code does not compile.

(12) (3') `[C++]` Read the following code.

```
int iarr[10]; // global
int main() {
  int *p1;
  int *p2 = new int[10];
  int *p3 = new int[10]{};
  int a[10] = {1};
  delete[] p2;
  delete[] p3;
}
```

A. The elements in `iarr` are initialized to zero.

B. `p1` is *default initialized*, whose value is the null pointer value.

C. The values of `p2[0], p2[1], ..., p2[9]` are zero.

D. The values of `p3[0], p3[1], ..., p3[9]` are zero.

E. Every element in `a` is initialized to `1`.

(13) (3') `[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_); // (*)
 private:
  const std::string title;
  const 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 member `price` does not appear in the initializer list of some constructor, it is initialized to `0.0` due to the in-class default member initializer.

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

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

D. The constructor (*) can be defined as follows.

```
// In class Book
Book(const std::string &title_, const std::string &isbn_, double price_) {
  title = title_;
  isbn = isbn_;
  price = price_;
}
```

Solution: D. Since `title` and `isbn` are `const`, they must be initialized in the initializer list.

(14) (3') `[C++]` Consider the `Book` class again, with more member functions added to it.

```
class Book {
 public:
  Book(const std::string &title_, const std::string &isbn_, double price_);
  auto&  get_title()        { return title;     }
  auto&  get_isbn()         { return isbn;      }
  double total_price(int n) { return n * price; }
 private:
  const std::string title;
  const std::string isbn;
  double price = 0.0;
};
```

Which of the following statements is/are true?

A. The return type of `get_title` and `get_isbn` is `const std::string &`.

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

C. `Book` has a copy constructor, a copy assignment operator, a move constructor, a move assignment operator and a destructor.
Whether this choice is selected or not does not affect the score.

D. The compiler will generate a destructor for `Book` as if it were defined as

```
// In class Book
~Book() {
  delete &price;
  delete &isbn;
  delete &title;
}
```

Solution: The copy constructor, the move constructor and the destructor of `Book` are implicitly declared, and implicitly defined if they are used. The copy assignment operator and the move assignment operator are implicitly declared and implicitly deleted since `title` and `isbn` are `const`.

The word "has" in choice C is not clear. Whether C is selected does not affect the score.

(15) (3') `[C++]` Let the class `Book` be defined as above. Suppose we use a `std::vector<Book>` to store the books.

```
std::vector<Book> books;
```

Which of the following is/are true?

A.
```
Book b("C++ Primer", "9780321714114", 47.99);
books.push_back(b);
```
This code appends a copy of `b` to the end of `books`.

B.
```
void add_book(const std::string &title, const std::string &isbn, double price) {
  Book b(title, isbn, price);
  books.push_back(b);
}
```

b is appended to the end of books through a move, not a copy.

C.
```
void add_book(const std::string &title, const std::string &isbn, double price) {
  books.push_back(Book(title, isbn, price));
}
```
An object of type Book is constructed by Book(title, isbn, price), and then moved into the vector.

D.
```
void add_book(const std::string &title, const std::string &isbn, double price) {
  books.emplace_back(title, isbn, price);
}
```
The arguments (title, isbn, price) will be forwarded to the constructor of Book, so that an object of type Book is constructed directly in that vector without move or copy.

(16) (3') [C++] Suppose we have two classes Item and DiscountedItem defined as follows:

```
class Item {
 public:
  Item(const std::string &name, double price)
      : m_name(name), m_price(price) {}
 protected:
  std::string m_name;
  double m_price = 0.0;
};

class DiscountedItem : public Item {
 public:
  DiscountedItem(const std::string &name, double price, double disc);
 private:
  double m_discount = 1.0;
};
```

Which of the following is/are true?

A. The constructor of DiscountedItem can be defined as follows:
```
DiscountedItem(const std::string &name, double price, double disc)
    : m_name{name}, m_price{price}, m_discount{disc} {}
```
Because m_name and m_price are protected members that are accessible in DiscountedItem.

B. The constructor of DiscountedItem can be defined as follows:
```
DiscountedItem(const std::string &name, double price, double disc) {
  m_name = name;
  m_price = price;
  m_discount = disc;
}
```

C. If the member m_name of Item is private, it is still inherited by DiscountedItem.

D.
```
int main() {
  std::shared_ptr<Item> dip = std::make_shared<DiscountedItem>("Coke", 3.0, 0.8);
}
```
The destructor of dip will call the destructor of DiscountedItem because dip is a smart pointer.

Solution: B. Since `Item` has a user-provided constructor, it does not have a default constructor. If the constructor of `Item` is not called explicitly in the constructor initializer list of `DiscountedItem`, the default constructor of `Item` is called, which results in an error.

(17) (3') [C++] Let `Item` and `DiscountedItem` be defined as follows.

```
class Item {
 public:
  Item(const std::string &name, double price)
      : m_name(name), m_price(price) {}
  std::string to_string() const {
    return "Name: " + m_name + ", price: " + std::to_string(m_price);
  }
  void set_price(double price) { m_price = price; }
  virtual ~Item() = default;
 protected:
  std::string m_name;
  double m_price = 0.0;
};

class DiscountedItem : public Item {
 public:
  DiscountedItem(const std::string &name, double price, double disc)
      : Item(name, price), m_discount(disc) {}
  void set_discount(double disc) { m_discount = disc; }
 private:
  double m_discount = 1.0;
};
```

Consider the following code.

```
std::vector<std::unique_ptr<Item>> snacks;
snacks.push_back(std::make_unique<Item>("cookies", 7.5));
snacks.push_back(std::make_unique<DiscountedItem>("chips", 6.0, 0.6));
snacks[0]->set_discount(0.5);                       // (1)
snacks[1]->set_discount(0.8);                       // (2)
snacks[1]->set_price(8.0);                          // (3)
std::cout << *snacks[0] << std::endl;               // (4)
std::cout << std::string(*snacks[0]) << std::endl;  // (5)
```

Which of the following explanations is/are correct?

A. Line (1) does not compile, because `Item` has no member `set_discount()`.

B. Line (2) compiles, because `snacks[1]` points to a `DiscountedItem`.

C. Line (3) does not compile, because `DiscountedItem` does not have an implementation of `set_price()`.

D. Line (4) does not compile, because there is no `operator<<` defined with operands `std::ostream&` and `Item`.

E. Line (5) compiles, because `Item` has a member function `to_string()` that will be automatically used as a conversion to `std::string`.

(18) (3’) `[C++]` Let `Item` and `DiscountedItem` be defined as in question (17). 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`.

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

Which of the following definitions is/are correct?

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

(19) (3’) `[C++]` Consider the following class representing a complex number $a + bi$, where $a, b \in \mathbb{R}$.

```cpp
class Complex {
  double real;
  double imaginary;
 public:
  Complex(double a, double b = 0) : real(a), imaginary(b) {} // (1)
  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`.

(20) (3') `[C++]` Consider the class `Complex` defined as above. Now we want to define `operator+=` for it. Select the **unique best** implementation of `operator+=` which compiles, involves no undefined behaviors and adheres to the conventions.

A.
```
// In class Complex
Complex &operator+=(Complex &rhs) {
  real += rhs.real;
  imaginary += rhs.imaginary;
  return *this;
}
```

B.
```
// In class Complex
auto &operator+=(const Complex &rhs) {
  real += rhs.real;
  imaginary += rhs.imaginary;
  return *this;
}
```

C.
```
// In class Complex
void operator+=(const Complex &rhs) {
  real += rhs.real;
  imaginary += rhs.imaginary;
}
```

D.
```
// In class Complex
Complex &operator+=(const Complex &rhs) {
  real += rhs.real;
  imaginary += rhs.imaginary;
}
```

E.
```
// In class Complex
Complex &operator+=(const Complex &rhs) {
  real += rhs.real;
  imaginary += rhs.imaginary;
  return this;
}
```

2. (15 points) Single Peak

We call a series of numbers **Single-Peaked** if there is one and only one “peak” such that:

- “Peak” is the **only** greatest number (high peak) / smallest number (low peak), and cannot be the first or last in the series.
- Numbers before the “peak” are non-descending (high peak) / non-ascending (low peak).
- Numbers after the “peak” are non-ascending (high peak) / non-descending (low peak).

Write a function named `isSinglePeaked`, that takes a parameter of `const std::vector<int>&`, and returns `true` if its contents are single-peaked, and `false` otherwise. The behavior is undefined if the vector passed in contains less than three elements.

Here are some examples of how your function should behave. Variables with ”true” in names should evaluate to `true`, and similarly for `false`.

```
bool true1  = isSinglePeaked({1, 3, 5, 4, 2});
bool true2  = isSinglePeaked({8, 2, -2, -5, 6});
bool true3  = isSinglePeaked({1, 3, 3, 4, 6, 4, 4});
bool false1 = isSinglePeaked({5, 4, 3, 2, 1});
bool false2 = isSinglePeaked({6, 0, 0, 2});
```

Write your function here:

Solution:

```
auto checkPeak(const std::vector<int> &v) {
  std::size_t i = 0;
  while (i + 1 < v.size() && v[i] <= v[i + 1])
    ++i;
  auto j = v.size() - 1;
  while (j > 0 && v[j] <= v[j - 1])
    --j;
  return i > 0 && i + 1 < v.size() && i == j;
}
auto negate(std::vector<int> v) {
  for (auto &x : v)
    x = -x;
  return v;
}
bool isSinglePeaked(const std::vector<int> &v) {
  return checkPeak(v) || checkPeak(negate(v));
}
```

3. (15 points) Time and Clocks

Suppose you are coding three classes, `Time`, `Clock`, and `AlarmClock`.

- `Time` stores the hour and minute of a certain time moment.
- `Clock` has a `Time`, and can `display` the time or `tick` to the next minute.

- `AlarmClock` inherits from `Clock`, and saves another `Time` for alarming.

Here is some incomplete code of these classes. Complete it according to the instructions.

```
class Time {
 public:
  Time(int minutes = 0)
    : m_hour((minute / 60) % 24), m_minute(minute % 60) {}
  Time(int hour, int minute)
    : m_hour((hour + minute / 60) % 24), m_minute(minute % 60) {}
  // (1)
 private:
  int m_hour;
  int m_minute;
  static std::string fill2(int x) {
    return x < 10 ? "0" + std::to_string(x) : std::to_string(x);
  }
};
class Clock {
 public:
  Clock(Time time) : m_time(time) {}
  void tick() {
    ++m_time; // It should add 1 minute to m_time.
  }
  virtual void display() const {
    // It should print something like "Now: 09:03".
    std::cout << "Now: " <<  m_time << std::endl;
  }
  // (*)
 protected:
  Time m_time;
};
class AlarmClock : public Clock {
 public:
  AlarmClock(Time time, Time alarm)
    : Clock(time), m_alarm(alarm) {}
  // (2)
 private:
  Time m_alarm;
};
// (3)
```

(1) (3') `Clock` should have a destructor at line (*). Write the destructor so that no undefined behavior happens when `Clock` and `AlarmClock` are used in polymorphism.

Solution:

```
virtual ~Clock() = default;
```

Also correct:

```
virtual ~Clock() {}
```

(2) (6') Currently the code does not compile, because `Time` lacks some operators. Add code to (1), (3), or both places, so that the code compiles and behaves like stated in the comments. Specify which of your code goes to (1) or (3). You may find the `Time::fill2` function useful.

Solution: (1):

```
Time &operator++() {
  ++m_minute;
  if (m_minute == 60) {
    m_hour = (m_hour + 1) % 24;
    m_minute = 0;
  }
  return *this;
}
friend std::ostream &operator<<(std::ostream &, const Time &);
```

(3):

```
friend std::ostream &operator<<(std::ostream &os, const Time &t) {
  return os << Time::fill2(t.m_hour) << ":" << Time::fill2(t.m_minute);
}
```

This friend function `operator<<` can also be defined at (1). `Time::fill2` can be replaced with `t.fill2`.

(3) (6') Finally, `AlarmClock` should override `display()`, in the following manner:

- It should print 2 or 3 lines.
- The first line should look like `Now: 09:03`, identical to what `Clock::display()` prints.
- The second line should look like `Alarm: 09:15`, printing the value of `m_alarm`.
- If the time now is its alarm time, print `ALARM!` on the third line.

Add code to do so. Feel free to add any other functions. Any of your code can go to (1), (2), or (3), and you should specify where it goes.

Solution: (1):

```
bool operator==(const Time &rhs) const {
  return m_hour == rhs.m_hour && m_minute == rhs.m_minute;
}
```

(2):

```
void display() const override {
  Clock::display();
  std::cout << "Alarm: " << m_alarm << std::endl;
  if (m_alarm == m_time)
    std::cout << "ALARM!" << std::endl;
}
```

Other ways to enable `AlarmClock::display` to compare `m_alarm` with `m_time` are allowed. For example, declare `AlarmClock` as a friend class of `Time` in (1), so that `m_hour` and `m_minute` can be accessed directly.

The first line can also be printed directly using

```
std::cout << "Now: " << m_time << std::endl;
```

4. (10 points) Ref-qualified member functions

Note: This problem is actually easier than the previous one.

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 ...
};
```

Solution:

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

The return type of the second function was required to be `Dynarray &&` (which yields an **xvalue**), but it is also ok to use `Dynarray` (which yields a **prvalue**). But the returned expression must be `std::move(*this)`.