CMPT 135-D100 Midterm Exam 1
Spring 2024

This is a **50 minute closed-book exam**: notes, books, computers, calculators, electronic devices, etc. are **not** permitted. Do not speak to any other students during the exam or look at their work. Please remain seated and **raise your hand** if you have a question.

Pointers and Memory Management

Suppose `pv` is a variable of type `vector<string*>*` that points to a vector that was allocated on the free store using `new`. The pointers in it point to *different* strings that were also allocated on the free store using `new`. There are no null pointers.

a) (1 mark) What's the type of the expression `&pv` ?

Sample Solution

```
vector<string*>**
```

b) (3 marks) Write a fragment of C++ code that properly de-allocates the vector `pv` points to, and also all the strings pointed to by it. There should be no memory leaks or other errors.

Sample Solution

```
for (int i = 0; i < pv->size(); i++) {
    delete (*pv)[i];
}
delete pv;
```

c) (6 marks) Write a function called `sum_positive(vector<int*> a)` that returns the sum of all the `int`s greater than 0 that `a` points to. Treat any null pointers in `a` as if they equal 0.

For example:

```
vector<int*> a;
a.push_back(new int(5));
a.push_back(new int(-2));
a.push_back(nullptr);
a.push_back(new int(3));
a.push_back(new int(-1));

cout << sum_positive(a);  // prints 8
```

Note that the type of `a` is `vector<int*>`, which is **not** the same type as `pv` from the previous question.

Write your answer here. **To get full marks, use a for-each style loop with a ":"**.

```
int sum_positive(vector<int*> a)
{
```

Sample Solution

```
int sum_positive(vector<int*> v) {
    int result = 0;
    for (int* p : v) {
        if (p != nullptr && *p > 0)
        {
            result += *p;
        }
    }
    return result;
}
```

Object-oriented Programming and Inheritance

a) (5 marks) Add the following to the `Movie` class below:

1. A **copy constructor** that uses an **initialization list** to make a new `Movie` object that is a copy of another `Movie` object.
2. A **destructor** that prints "done!".
3. A **setter** that lets the user change the name of a `Movie`.
4. A **getter** that returns the year of a `Movie`.

```
class Movie {
    string title;
    int year;

public:
    Movie(const string& t, int y) {
       title = t;
       year = y;
    }

    // ... your code goes here ...
```

Sample Solution

```
   class Movie {

   // ...
   Movie(const Movie& m)
   : title(m.title), year(m.year) { }

   // Alternative copy constructor using
   // delegation:
   // Movie(const Movie& m)
   // : Movie(m.title, m.year) { }


   ~Movie() { cout << "done!"; }

   void set_title(string t) { title = t; }

   int get_year() const { return year; }

}; // class Movie
```

b) (5 marks) Implement a class called `int_list` that has **all** the methods and features of a `vector<int>`, and also has a method called `zero_count()` that returns the number of 0s in the vector.

For example:

```
int_list lst;
lst.push_back(5);
lst.push_back(10);
lst.push_back(0);
cout << lst[0];           // prints 5
cout << lst[1];           // prints 10
cout << lst.zero_count(); // prints 1
```

Sample Solution

```
class int_list
: public vector<int>

{

public:
    // sample implementation 1
    int zero_count() const {
        int result = 0;
        for(int i = 0; i < size(); i++) {
            if ((*this)[i] == 0) {
               result++;
            }
        }
        return result;
    }


    // sample implementation 2
    int zero_count() const {
        int result = 0;
        for(int n : *this) {
            if (n == 0) {
               result++;
            }
        }
        return result;
    }

};
```

Multiple Choice

For each question, fill-in **the one best answer** on the answer sheet.

Every correct answer is worth 1 mark. Incorrect answers, unanswered questions, questions with more than one answer, or questions with illegible answers, are worth 0.

1) What function is called first when you run a C++ program?

A. the function that appears **first** in the source code
B. the function that appears **last** in the source code
C. whichever function the programmer designates should be called first
D. `main` is always called first

2) Where are **local** variables stored?

A. only the call stack
B. only the free store
C. sometimes the call stack, sometimes the free store
D. only static memory

3) Consider these statements:

i) An advantage of assert-style testing over plain if-statement testing is that asserts can tell you the line they failed on.
ii) Table-based testing can be used for unit testing.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

4) Consider these statements:

i) System testing is a kind of whitebox testing.
ii) Whitebox testing is a kind of blackbox testing.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

5) Consider these statements:

i) Property-based testing is when you test if a function satisfies certain properties.
ii) ChatGPT is *terrible* at creating test cases for C++ code.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

6) Consider these statements:

i) It's always an error to de-reference a null pointer.
ii) It's *not* an error if a pointer `m` points to an invalid memory location and never evaluates `*m`

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

7) This code fragment calls the `safe` function in the box:

```
int* a = new int(3);
safe(a);
```

```
void safe(int* p) {
   if (p != nullptr) {
      delete p;
      p = nullptr;
  }
}
```

Consider these statements:

i) **After** the code fragment runs, the `int` that `a` points to has been properly de-allocated.
ii) **After** the code fragment runs, `a == nullptr`

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

8) Consider these two functions:

```
void g() {
   int n = 6;
   int* p = &n;
   p = nullptr;
   delete p;
}
```

```
void h() {
   int n = 6;
   int* p = &n;
   delete p;
   p = nullptr;
}
```

i) Calling `g()` causes a memory error.
ii) Calling `h()` causes a memory error.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

9) Consider this (correctly working) code fragment:

```
string* p = new string("SFU");
```

i) `*p.size()` returns 3
ii) `p->size()` returns 3

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

10) Consider these statements:

i) A class must have exactly one constructor.
ii) A class can have multiple destructors.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

11) Consider these statements:

i) Immutable objects can have public member variables.
ii) A non-`const` getter will always cause a compile-time error.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

12) Consider this code:

```
class Nameable {
public:
   string get_name() const = 0;
};
```

i) `get_name()` is **abstract**
ii) `get_name()` is **virtual**

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

13) Consider these statements:

i) a destructor in a base class should always be declared **virtual**
ii) a destructor in a base class should always be declared **abstract**

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

14) Suppose class `A` and class `B` both inherit from class `Base`.

i) A pointer of type `A*` can point to an object of type `B`.
ii) A pointer of type `Base*` can point to an object of either type `A` or type `B`.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is false and ii) is true
D. i) is true and ii) is false

15) Consider this code:

```
class Menu {
public:
   string get_title() const {
       return "Menu";
   }
};

class Fancy_menu : public Menu {
public:
    string get_title() const {
        return "Fancy Menu";
    }
};
```

And also:

```
Menu* a = new Menu();       // line 1
Menu* b = new Fancy_menu(); // line 2
```

A. `a->get_title()` returns `"Menu"`, and `b->get_title()` returns `"Fancy Menu"`
B. `a->get_title()` returns `"Menu"`, and `b->get_title()` returns `"Menu"`
C. line 2 would cause a compile-time error