University of Toronto

Duration: 1 hour 15 minutes
Examiner: Kuei (Jack) Sun

Please fill your student number, last and first name below and then read the instructions carefully.

Student Number: ____ ____ ____ ____ ____ ____ ____ ____ ____ ____

Last Name: _____

First Name: _____

Instructions

Examination Aids: Ruler and examiner approved aid sheet are allowed.

Do not turn this page until you have received the signal to start.

You may remove the aid sheet from the back of this test book. Do not remove any other sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Use the blank space in last page as scratch space. Its content will not be marked.

This exam consists of 5 questions on 10 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 95 marks.

For the written answers, explain your reasoning clearly. Be as brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer. Please write legibly!

Work independently.

MARKING GUIDE

Q1: _____ (20)

Q2: _____ (22)

Q3: _____ (19)

Q4: _____ (16)

Q5: _____ (18)

Total: _____ (95)

Question 1. **True or False** [20 marks]

Circle **T** if the statement is true, otherwise circle **F** if the statement is false. 2 marks each.

1. Python does not allow overloading the assignment operator. **(T)** **F**
2. Languages that allow arbitrary pointer arithmetic are always memory unsafe. **(T)** **F**
3. Python list, tuple, and dictionary are all sequence types. **T** **(F)**
4. Importing a new module (e.g. `import foo`) in Python can only be done in global scope. **T** **(F)**
5. All attributes in Python can be overridden by a subclass. **(T)** **F**
6. `container_of()` is a C macro function **(T)** **F**
7. An abstract base class cannot be instantiated. **(T)** **F**
8. Unlike a C enum, an enum class in C++ is type-safe by disallowing casting to and from integers. **T** **(F)**
9. Assume `i` and `j` are integers, then `(i>66) ? i : j = 666;` is a valid C++ statement. **(T)** **F**
10. A sufficiently powerful static type checking system displaces the need for dynamic type checking. **T** **(F)**

Question 2. **Multiple Choices** [22 marks]

Pick all answer(s) that are correct. You will lose 1 mark per wrong choice, down to 0 marks.

a) For each programming language concept, circle whether it is applicable to generic programming, reflective programming, and/or metaprogramming (more than one choice is possible). [6 marks]

i. Void pointer and pointer arithmetic

Generic programming Reflective programming Metaprogramming

ii. Customized attribute management (e.g. __setattr__, __getattr__, __get__, …etc)

Generic programming **Reflective programming** Metaprogramming

iii. Macro systems

Generic programming Reflective programming **Metaprogramming**

iv. Template (e.g. C++ template)

Generic programming *Reflective programming* **Metaprogramming**

Note: You neither lose nor gain marks for the two choices in *italics*.

b) Choose all language design features or choices that apply to C++. [4 marks]

i. Off-side rule

ii. Function scope

iii. Weakly-typed

iv. Structural Typing

v. Contravariant parameters

c) Which of the following about Python metaclass are true? [4 marks]

i. The __call__ method of a metaclass initiates the process of creating a new class.

ii. The __new__ method of a metaclass instantiates new objects for its classes.

iii. To avoid infinite recursion, a metaclass cannot have its own metaclass.

iv. Like regular classes, multiple inheritance is supported for metaclasses.

v. During name resolution, a class's metaclass is looked up before its super classes are searched.

d) Which of the following functions or operators are examples of introspection in C++ (static or runtime)? [4 marks]

i. `decltype`

ii. `getattr`

iii. `sizeof`

iv. `typeid`

v. `reinterpret_cast`

e) Which of the following statements are true about dynamic programming? [4 marks]

i. It is another name for memoisation.

ii. It requires the function to be referentially transparent to guarantee the same program behaviour.

iii. It is done as part of recursion.

iv. For a particular set of input, *args*, it may solve subproblems that are not necessary to produce the solution for the set of input *args*.

v. It can be used to improve performance of sorting algorithms. Hint: think about whether sorting algorithms in general have overlapping subproblems.

Question 3. **Short Questions** [18 marks]

a) List two advantages inheritance have over composition. Your answer must relate to programming language features, e.g. "it is more flexible" is not an acceptable answer. [4 marks]

Immediate code reuse, does not require forwarding.

Dynamic dispatch or interface inheritance

b) Given a dictionary *d*, and a list *r*¸create a new dictionary *dp* where all keys in *r* are removed and all remaining values in *d* are squared. You must use exactly one line of dictionary comprehension to solve this problem. [5 marks]

Example:

```
d = { 'A':6, 'B':4, 'C':8, 'D':1, 'E':7, 'F':9, 'G':5 }
r = ['C', 'E', 'G']
# dp would be {'A': 36, 'B': 16, 'D': 1, 'F': 81}
```

```
dp = { k : v**2 for k, v in d.items() if k not in r }
```

```
dp = { k : d[k]**2 for k in d if k not in r }
```

1 mark for k
1 mark for v**2
1 mark for "for k in d"
1 mark for if k not in r
1 mark for { … }
-1 mark for wrong order (if before for)

c) Complete the move constructor and the destructor for the Blackjack class [10 marks]

```
class Blackjack {
  Hand * dealer;           // dealer's hand
  double result;           // win/loss amount
  std::vector<Hand> hands; // player's hands

public:
  void start(Shoe * shoe) {
       dealer = new Hand(shoe);  // create dealer hand
       hands.emplace_back(shoe); // create player's first hand
  }

  Blackjack() : dealer(nullptr), result(0.) {}
  ~Blackjack() {  // implement me


       delete dealer;



  }

  Blackjack(Blackjack && rhs) {   // implement me


       dealer = rhs.dealer;
       rhs.dealer = nullptr;
       result = rhs.result;
       hands = std::move(rhs.hands);



       // 2 marks for each line


  }

  /* … other member functions … */
}
```

Question 4. **Method Resolution Order** [16 marks]

Given:

```
class A : foo = 1
class B : qux = 2
class W(A) : baz = 3
class X(A) : bar = 4
class Y(B) : foo = 5
class Z(B) : baz = 6
```

```
class N(type) : foo = 11
class M(N) : qux = 7
class P(W, X, Y) : pass
class Q(X, Y, Z): bar = 9
class R(W, Z) : bar = 0
class H(P, Q, R, metaclass=M) : pass
```

a) What is the C3 Linearization of H? [12 marks]

```
L[A] = (A, o)
L[B] = (B, o)
L[W] = (W, A, o)
L[X] = (X, A, o)
L[Y] = (Y, B, o)
L[Z] = (Z, B, o)
L[P] = (P, merge((W, A, o), (X, A, o), (Y, B, o), (W, X, Y)))
L[P] = (P, W, X, merge((A, o), (A, o), (Y, B, o), (Y)))
L[P] = (P, W, X, A, merge((o), (o), (Y, B, o), (Y)))
L[P] = (P, W, X, A, Y, B, o)
L[Q] = (Q, merge((X, A, o), (Y, B, o), (Z, B, o), (X, Y, Z)))
L[Q] = (Q, X, A, merge((o), (Y, B, o), (Z, B, o), (Y, Z)))
L[Q] = (Q, X, A, Y, Z, B, o)
L[R] = (R, merge((W, A, o), (Z, B, o), (W, Z)))
L[R] = (R, W, A, Z, B, o)
L[H] = (H, merge((P, W, X, A, Y, B, o), (Q, X, A, Y, Z, B, o),
       (R, W, A, Z, B, o), (P, Q, R)))
L[H] = (H, P, Q, R, merge((W, X, A, Y, B, o), (X, A, Y, Z, B, o),
       (W, A, Z, B, o)))
L[H] = (H, P, Q, R, W, X, merge((A, Y, B, o), (A, Y, Z, B, o),
       (A, Z, B, o)))
L[H] = (H, P, Q, R, W, X, A, Y, merge((B, o), (Z, B, o),
       (Z, B, o)))

L[H] = (H, P, Q, R, W, X, A, Y, Z, B, o)
```

1 mark for each class in the right order, including o (11 marks)

1 mark for not including metaclasses (or put M, N, o at the end)

b) What are the values of H.foo, H.bar, H.baz, and H.qux? [4 marks]

`foo: 1` `bar: 9`

`baz: 3` `qux: 2`

Question 5. C++ Template Programming [18 marks]

a) Given the following template functions and regular functions:

```
struct Point {
  unsigned x, y;
  Point() : x(0), y(0) {}
} p;

template<class T> void f(T t) { cout << "A"; }
template<class T> void f(T * t) { cout << "B"; }
template<> void f(int t) { cout << "C"; }
void f(Point && t) { cout << "D"; }
void f(unsigned t) { cout << "E"; }
```

Determine the output of each invocation, 1 mark each. [6 marks]

`f(5)` **C**	`f(3.4)` **A**
`f(p.x)` **E**	`f(p)` **A**
`f(&p)` **B**	`f(Point())` **D**

b) Judging by the fact that in a C++ template body, it accepts any parameterized type that has the methods and/or fields that the template uses, how is type equivalence decided? [2 marks]

Nominal Typing Structural Typing **Duck Typing**

c) Write a C++ template class, `Pair`, which accepts one template parameter, `T`, and creates a class which has two fields, `x` and `y`, both of type `T`. Implement a constructor that takes two parameters to initialize both `x` and `y`, and implement a member function named `get` which returns a pointer to `x` if the single integer argument is 0, pointer to `y` if the argument is 1, and `nullptr` for all other argument values. You are required to implement the member function `get` outside of the body of the template class (i.e. you may not inline the member function inside the class). [10 marks]

-1 if missing
-1 if missing
1 mark
3 marks
1 mark

```
template<typename T>
class Pair {
     T x, y;
public:
     Pair(T x, T y) : x(x), y(y) {}
     T * get(int i);
};
```

1 mark
1 mark (for <T>)
1 mark
1 mark
1 mark

```
template<typename T>
T * Pair<T>::get(int i) {
     if (i == 0) return &x;
     else if (i == 1) return &y;
     return nullptr;
}
```

-1 if did not return pointer (&x)
All valid variations of the constructors are OK
1st mistake is free (not penalized)

END OF EXAMINATION