| Câu | Điểm | Nội dung | Điểm | Ghi chú |
| --- | --- | --- | --- | --- |
| 1.a | **1** | **Mục tiêu của việc sử dụng template là gì?**  1. **Reduced code duplication** (Loại bỏ mã nguồn trùng lặp): viết mã một lần cho nhiều kiểu dữ liệu khác nhau  2. **Improved code maintainability**: Dễ bảo trì mã nguồn vì chỉ viết mã một lần cho nhiều loại kiểu dữ liệu khác nhau  3. **Type safety**: đảm bảo tương thích kiểu khi biên dịch | 0.25 | Chỉ cần 1 ý đúng là được trọn điểm. |
| **Hãy cho một ví dụ minh họa cài đặt và sử dụng function template.**  1. **Cài đặt** function template  2. **Ví dụ** minh họa gồm ít nhất 2 kiểu dữ liệu khác nhau | 0.25  0.5 |  |
| 1.b | **1** | **Hãy phân biệt hàm ảo (virtual function) và hàm thuần ảo (pure virtual function).**  **a. Cú pháp khai báo:**  + Virtual function sử dụng từ khóa virtual trước tên hàm  + Pure virtual function sử dụng từ khóa virtual trước tên hàm cùng khai báo = 0 ở cuối.  **b. Thân hàm**  + Virtual function: có cài đặt thân hàm  + Pure virtual function: không có cài đặt thân hàm  **c. Ý nghĩa ở lớp con**  + Virtual function: đóng vai trò cách hành xử mặc định ở lớp cha. Lớp con **có thể** ghi đè (override) cách hành xử theo ý thích nếu muốn  + Pure virtual function: đóng vai trò khuôn mẫu, lớp con **bắt buộc** phải cài đặt - cung cấp cách hành xử cụ thể.  **d. Lớp chứa hàm tương ứng (lớp cha)**  + Virtual function: Có thể tạo ra thể hiện như bình thường  + Pure virtual function: Trở thành lớp trừu tượng (abstract class) và không thể tạo ra thể hiện | 0.5 | Mỗi ý đúng 0.25 điểm. Chỉ cần 2/4 ý là được trọn 0.5 điểm. |
|  | **Cho ví dụ minh họa.** | 0.25 |  |
| **Tại sao hàm ảo quan trọng cho việc kế thừa trong lập trình hướng đối tượng?**  1. **Runtime Polymorphism** (Đa hình khi chạy): Lựa chọn đúng cài đặt theo kiểu dữ liệu tương ứng khi chạy, tạo ra cấu trúc mã nguồn đồng nhất và linh hoạt  2. **Thiết kế dễ mở rộng**: Lớp cơ sở cung cấp cài đặt mặc định để cho các lớp con dễ dàng sử dụng hoặc tự cung cấp cài đặt của riêng mình.  3. **Loại bỏ** **mã nguồn trùng lắp** và **tăng tính dễ bảo trì**.  4. Hàm thuần ảo và lớp trừu tượng giúp **tạo ra khuôn mẫu**, dễ dàng tổng quát hóa chương trình | 0.25 | Chỉ cần đúng 1/4 ý là được |
| 2.a | **0.5** | **Khai báo giao diện IStack với 4 hàm: push, pop, size, clear**  class IStack {  public:  virtual void push(int x) = 0;  virtual int pop() = 0;  virtual int size() = 0;  virtual void clear() = 0;  }; | 0.5 |  |
| 2.b | **0.5** | **Khai báo lớp StackByLinkedList sử dụng danh sách liên kết đơn**  class **StackByLinkedList**: public **IStack** {  private:  struct **Node** {  int data;  Node\* next;  Node(int value);  };  Node\* \_top = nullptr;  public:  void push(int x) override;  int pop() override;  int size() override;  void clear() override;  }; | 0.25  0.25 | Không có từ khóa override **không bị trừ điểm**. |
| 2.c | **1** | **Cài đặt các hàm push(x), pop(), size(), clear() và các hàm liên quan cho StackByLinkedList**  **// Node đầu tiên chính là phần tử trên cùng.**  **// Không dùng node cuối cùng làm phần từ trên cùng vì sẽ chậm**  void StackByLinkedList::**push**(int x) {  Node\* newNode = new Node(x);  newNode->next = \_top;  \_top = newNode;  }  int StackByLinkedList::**pop**() {  Node\* temp = \_top;  int data = temp->data; // Unboxing the data    \_top = \_top->next; // Phần tử kế sẽ là phần tử trên cùng  delete temp;    return data;  }  int StackByLinkedList::**size**() {  int count = 0;  Node\* p = \_top;    while (p != nullptr) {  count++;  p = p->next;  }    return count;  }  void StackByLinkedList::**clear**() {  Node\* p = \_top;  Node\* temp = nullptr;    while (p != nullptr) {  temp = p;  p = p->next;  delete temp;  }  } | 0.25  0.25  0.25  0.25 |  |
| 2.d | **1** | **Chỉnh sửa hàm pop() có bổ sung cơ chế xử lí ngoại lệ**  int StackByLinkedList::**pop**() {  if (\_top == nullptr) {  throw std::runtime\_error("Stack underflow");  }  // Unboxing the data  Node\* temp = \_top;  int data = temp->data;    \_top = \_top->next;  delete temp;    return data;  } | 0.5 | Nếu viết dạng ý tưởng throw exception(“Stack dang rong”) chứ không phải dùng chính xác **runtime\_error** cũng được chấp nhận  Lí do là lớp exception không chấp nhận chuỗi làm đối số hàm tạo |
| **Code minh họa trong hàm main**  int main() {  StackByLinkedList stack;  try {  stack.pop();  } catch (const exception& e) {  std::cout << e.what();  }  return 0;  } | 0.5 |  |
| 3.a | **2** | **Vẽ sơ dồ lớp** |  | - Coding convention cho thành phần private khi vẽ sơ đồ UML được lựa chọn là camelCase, tuy nhiên nếu sinh viên làm theo thói quen \_camelCase vẫn được chấp nhận.  Lời giải gợi ý có cân nhắc yếu tố **tránh over-engineering**, tức tạo ra giải pháp quá sức phức tạp cho vấn đề đơn giản trong giới hạn tài nguyên (ở đây chỉ cân nhắc tài nguyên thời gian thi).  - Tiền tệ sẽ là việt nam đồng, vốn là số nguyên. Không cần phức tạp hóa thành số thực  - Tiền mã hóa mặc dù giá trị số lượng token có thể là số thực, tuy nhiên khi qui đổi giá trị thì cũng sẽ làm tròn về việt nam đồng, vốn là số nguyên.  - UML Class diagram **không có qui chuẩn** cho việc báo hiệu hàm là hàm ảo, hàm thuần ảo, override  - Có nhiều cấp độ UML class diagram, đáp án gợi ý sử dụng UML class diagram **cấp độ mã nguồn**, tức **vay mượn** nhiều từ khóa **ở cấp độ mã nguồn** như:   * Mảng: vector thay vì [] * Kiểu dữ liệu: float, double, int, string, \* cho con trỏ * Virtual và virtual = 0 cho hàm ảo / thuần ảo * Thêm từ khóa override   Như vậy có thể thấy   * Không có các từ khóa virtual, override **không bị trừ điểm** vì gốc UML ko có các từ khóa này. Để biểu diễn hàm abstract, chuẩn của UML là **in nghiêng**, vốn khá khó nhận biết trong ngữ cảnh chữ viết tay và chữ viết tay khi làm bài thi đa số là **rất xấu**. |
| 3.b | **2** | **Cài đặt chi tiết các hàm liên quan chức năng thống kê tổng giá trị của các tài khoản (in ra màn hình).**  int BankAccount::**value**() {  return \_balance;  }  int StockAccount::**value**() {  return \_amount \* \_price;  }  int CryptoAccount::**value**() {  return \_amount \* \_exchangeRate;  }  // Hướng suy nghĩ 1: thống kê giá trị của các tài khoản  // có n tài khoản thì in ra n tài khoản với giá trị tính được tương ứng  void NormalUser::reportTotalValue() {  cout << “Total accounts:” << \_accounts.size() << “\n”;  for(int i = 0; i < \_accounts.size(); i++) {  cout << \_accounts[i]->name() << “, “ << \_account->value() << “\n”;  }  }  // Hướng suy nghĩ 2: Thống kê gom nhóm theo loại  // Có 3 loại tài khoản, cần cộng dồn vào  // Giải pháp tránh over-engineer là dùng 3 số nguyên cho 3 loại  void NormalUser::reportTotalValue() {  vector<Account\*> prototypes = {  new BankAccount(),  new StockAccount(),  new CryptoAccount()  };  vector<int> sums = {0, 0, 0};    for(int i = 0; i < \_accounts.size(); i++) { //RTTI  for(int **j** = 0; **j** < \_prototypes.size(); **j**++) {  if (\_account[i]->name() == \_prototypes[**j**]->name()) {  sums[**j**] += \_account[i]->value();  break; // Có thì tốt không có không sao  }  }  }  **// In mảng sums ra => Dễ**  } | 0.25  0.25  0.25 | Coding convention lựa chọn khi lập trình cho thành phần private sẽ có tiền tố \_, tuy nhiên sinh viên không sử dụng \_camelCase mà sử dụng camelCase cũng được chấp nhận.  - Sinh viên **không cần cài** hàm **name**() trả ra tên của loại tài khoản cũng được.  - Không cần quan tâm đến đơn vị tiền tệ là đồng cũng không sao vì mặc định là vậy  - Không có std trước cout không bị trừ điểm.  **Suy nghĩ đơn giản vầy được 0.25 điểm**  **Giải theo cách này mới được trọn 1.25 điểm.**  Sử dụng std::**map** càng tốt  Chưa tính phải thu hồi vùng nhớ sau khi sử dụng  Sử dụng **typeid** thì ngắn hơn khỏi phải cài hàm name() hoặc toString() |
| 3.c | **1** | **Thêm hũ tài chính vào chương trình ở dạng cây** |  | **NormalUser** lúc này thay vì có \_accounts (các tài khoản) thì sẽ là \_jar (hũ tài chính gốc)  Các lớp còn lại là BankAccount, StockAccount, CryptoAccount **không cần thay đổi**  **Cài đặt 1 theo phương án dạng cây**  Cài đặt 2 theo dạng sử dụng mẫu **Composite** |