Câu 1: Hãy trình bày về kiểu con trỏ (pointer) và kiểu tham khảo (reference)? Cho ví dụ cho mỗi kiểu? Nêu các điểm khác biệt của hai loại kiểu này? Hãy giải thích vì sao các loại kiểu này gây ra hiện tượng alias?

Kiểu dữ liệu con trỏ:

- Kiểu con trỏ là một kiểu dữ liệu có miền giá trị đủ để biểu diễn tất cả địa chỉ ô nhớ của máy tính. Trong đó có một giá trị đặc biệt là NULL (hoặc None, Nil) để chỉ ra đây là con trỏ không trỏ vào ô nhớ nào cả.
- Kiểu con trỏ cho phép truy suất ô nhớ gián tiếp, từ đó cho phép ta quản lý bộ nhớ động (nghĩa là quản lý các ô nhớ được cấp phát khi chương trình thực thi).
- Kiểu con trỏ hỗ trợ 2 toán tử là *"Truy suất giá trị ô nhớ" (dereference)* và *"Phép gán" (assignment)***.**
- Ví dụ trong ngôn ngữ C:

```
// Initialize the void pointer
void* ptr = NULL;
// Pointer assignment operator
ptr = &var;
// Pointer dereference operator
*((int*) ptr) += 10;
```

Kiểu dữ liệu tham chiếu:

- Kiểu tham chiếu cho phép ta tham chiếu đến một đối tượng (object) hay giá trị (value) trong chương trình.
- Kiểu tham chiếu không hỗ trợ thêm các toán tử, sau khi khởi tạo, biến tham chiếu được sử dụng giống hoàn toàn như biến nó tham chiếu tới.
- Ví dụ trong ngôn ngữ C:

```
// Intitialize the reference variable
int& a = b;
// a and b is the same object
a = 5; ++b; printf("%i", a); // output: 6
```

So sánh kiểu con trỏ và kiểu tham khảo:

- Kiểu con trỏ:
    - Chứa địa chỉ của một biến.
    - Cho phép không khởi tạo khi khởi tạo, cho phép giá trị NULL.
    - Cho phép thay đổi giá trị sau khi khởi tạo (phép gán).
    - Để truy suất giá trị ô nhớ thì sử dụng phép dereference (* in c).

- Kiểu tham chiếu:
    - Là bí danh (alias) của một biến.
    - Bắt buộc phải khởi tạo giá trị khi khai báo.
    - Không được phép tham chiếu đến biến khác.
    - Để truy suất giá trị ô nhớ sử dụng tên biến tham chiếu.

Câu 2: Hãy nêu điểm khác biệt chính của các cơ chế gọi chương trình con: đệ qui (recursive), biến cố (exception), trình cộng hành (coroutine), trình định thời (scheduled subroutine) và công tác (task) so với cơ chế Gọi-Trở về đơn giản (simple call-return)?

- Đệ qui (recursive): cho phép gọi lại đệ quy (gián tiếp hoặc trực tiếp).
- Biến cố (exception): không thể gọi chương trình con một cách rõ ràng (explicit). Chương trình sẽ được gọi ngầm định (implicit) khi có biến cố (exception) hoặc sự kiện (event) xảy ra.
- Trình cộng hành (coroutine): chương trình có nhiều điểm nhập (entry point), từ đó cho phép chương trình tạm hoãn (postpone) và chuyển quyền điều khiển lại lại cho chương trình đã gọi nó (caller). Khi được gọi lại, chương trình sẽ bắt đầu tại điểm đã toạn hoãn.
- Công tác (task): cho phép thực thi đồng thời nhiều chương trình con bằng cách cách trên các máy đa bộ xử lý hoặc có cơ chế time-sharing.
- Trình định thời (scheduled subroutine): Các chương trình con được gọi không thực thi ngay mà bị trì hoãn (delay):
    - They thời gian (by time): Sau 1 khoảng thời gian nhất định.
    - Theo độ ưu tiên (by priority): Sau khi các chương trình con có độ ưu tiên cao hơn thực thi xong

Câu 3: Hãy giải thích các phương pháp (lock-and-key, tombstone) tránh tham chiếu treo (dangling reference)? Nêu rõ cách truy xuất trong trường hợp bình thường, khi huỷ bỏ một đối tượng và cách phát hiện khi có tham chiếu treo.

- Phương pháp lock-and-key: phương pháp này sử dụng cặp khóa địa chỉ (key - address) làm con trỏ trỏ vào vùng nhớ được cấp phát, và ngoài ra, một giá trị được gọi là khóa (lock) sẽ được lưu kèm theo trên vùng nhớ được cấp phát. Khi cấp phát giá trị lock và key là giống nhau.
    - Chỉ có thể truy suất giá trị vùng nhớ, khi key và lock là giống nhau.
    - Khi hủy cấp phát, giá trị lock được thay đổi để khác key.
    - Phát hiện tham chiếu treo khi ô nhớ được truy suất mà key và lock khác nhau.

- Phương pháp tombstone: phương pháp tạo thêm một ô nhớ để lưu con địa chỉ của vùng nhớ được cấp phát, gọi là tombstone. Con trỏ sẽ trỏ vào tombstone.
    - Khi truy suất vùng nhớ, từ pointer có truy suất tombstone, từ tombstone ta truy suất giá trị vùng nhớ.
    - Khi hủy cấp pháp, giá trị tombstone sẽ được gán thành Null.
    - Phát hiện tham chiếu treo khi tombstone là Null.

Câu 4: Hãy thực hiện (viết các phương trình thể hiện các ràng buộc kiểu) suy diễn kiểu để suy ra kiểu của hàm sau. Biểu thức điều kiện của phát biểu if phải có kiểu boolean. Kiểu của biểu thức sau return phải cùng kiểu trả về của hàm.

```
H(x, f, h) { if (f(x)) return h(h(x)); else return f(x);}
```

Biểu thức điều kiện của phát biểu if là boolean

```
=> f(x):boolean => h(x):boolean => x:boolean
f(x) : boolean -> boolean; h(x) : boolean -> boolean;
H(x, f, h) : (boolean, boolean -> boolean, boolean -> boolean) -> boolean
```

Câu 5: Cho đoạn mã sau được viết trên ngôn ngữ Scala dùng qui tắc tầm vực tĩnh (static-scope rule). Nhắc lại, trên Scala, từ khoá var để khai báo biến, def để khai báo hàm, Int=>Int là kiểu hàm nhận vào 1 giá trị nguyên trả về 1 giá trị nguyên, giá trị của biểu thức cuối cùng trong một hàm là giá trị trả về của hàm.

```
def main={
    var a=0;var b=1;var c=4 //1
    def sub1(a:Int) = { //2
        def sub2(a:Int,c:Int,f:Int=>Int)=(f(a)−f(c))∗2 //3
        def sub3(b:Int) = b∗c−a; //4
        b = sub2(1,2,sub3)
    }
    sub1(3)
    print(b) //5
}
```

α. Cho biết môi trường tham khảo tĩnh (static referencing environment) của các hàm main, sub1, sub2, sub3 (với các tên a, b và c phải ghi kèm // dòng khai báo của tên, ví dụ a//1).

Frame	**Referencing Environment**
main	main, sub1, a//1, b//1, c//1
sub1	main, sub1, sub2, sub3, a//2, b//1, c//1
sub2	main, sub1, sub2, sub3, a//3, b//1, c//3, f//3
sub3	main, sub1, sub2, sub3, a//2, b//4, c//1

b. Hãy vẽ các bản ghi hoạt động của các chương trình con còn đang tồn tại khi hàm main được gọi và chương trình thực thi đến dòng lệnh trong thân hàm sub3 // 4 lần thứ nhất? Trình bày tiếp sự thực thi cho đến khi in giá trị b của bản hoạt động main ở dòng // 5?

Bản ghi hoạt động hàm (Local data -> Static link -> Dynamic link -> Return Address)
Local data: dữ liệu được đưa vào Stack (theo quy tắc FIFO)
Static link: trỏ vào dưới (bottom) của ARI (Activation Record Instance)
Dynamic link: trỏ vào trên (top) của ARI
Return Address: Instruction Address + 1 của caller
=> Vẽ theo quy luật trên

Main -> Sub1 -> Sub2 -> Sub3
Chương trình chạy là in ra : -8

Câu 6: Cho một đoạn chương trình được viết trên một ngôn ngữ tựa C:

```
int A[3] = {4,6,14}; //index of A starts from 0
int j = 0;
int n = 3;
int sumAndDecrease(int a,int i){
    int s = 0;
    for(; i < n; i = i + 1){
        s = s + a;
        A[j] = A[j] – 1;
    }
    return s;
}

void main(){
    int s = sumAndDecrease(A[j],j);
    cout << s << A[0] << A[1] << A[2]; //1
}
```

Hãy cho biết và giải thích kết quả in ra của chương trình trong các trường hợp sau:
// Kết quả ở câu a,b,c là không có dấu cách khoảng

a. Nếu a và i được truyền bằng giá trị **12 1 6 14**
b. Nếu a và i được truyền bằng giá trị-kết quả. ***12 4 6 14***
c. Nếu a và i được truyền bằng tham khảo. ***10 3 5 13***
d. Bếu a và i được truyền bằng tên. ***24 3 5 13***

Câu 7: Giải thích đặc điểm của kiểu union? Cho một ví dụ sử dụng kiểu union và giải thích vì sao phải dùng kiểu union trong ví dụ của em?

Union là một cấu trúc dữ liệu đặc biệt cho phép ta lưu trữ nhiều loại dữ liệu cho cùng một vùng nhớ. Ta có thể định nghĩa kiểu Union với nhiều kiểu dữ liệu, nhưng tại một thời điểm chỉ một kiểu dữ liệu được chọn.

Union được sử dụng để cung cấp một cách hiệu quả việc xử dụng 1 ô nhớ cho nhiều mục đích. Ví dụ như ta muốn tạo một Linked List chứa 2 loại dữ liệu (int, float). Bằng cách này ta có thể lưu một list các các phần tử (kiểu int hoặc float) mà không tốn gấp đôi chi phí (1 biến thay vì 2 biến).

```
struct Node {
      struct Node* next;
      union Num {int i; float f} data;
};
```

Union còn được sử dụng để tạo ra giả đa hình (pseudo polymorphism) trong các ngôn ngữ không hộ trợ lập trình hướng đối tượng.

```
union Animal {
      Dog d;
      Cat c;
};
```

Câu 8: Hãy viết lại biểu thức ở dạng trung tố (infix) sau sang dạng biểu thức tiền tố (prefix) Cambridge Polish:

Biết rằng độ ưu tiên và tính kết hợp của các phép toán trong biểu thức như thông thường (đều kết hợp trái và * có ưu tiên cao hơn +, -). Trong biểu thức tiền tố nhiều toán hạng, thứ tự tính toán cũng từ trái sang phải.

Yêu cầu: Biểu thức dạng tiền tố Cambridge Polish phải thỏa các yêu cầu sau:

- Thứ tự xuất hiện các toán hạng trong biểu thức dạng tiền tố phải có cùng thứ tự xuất hiện như trong biểu thức trung tố.
- Số (và) là ít nhất.
- Có cùng thứ tự tính toán các phép toán với thứ tự đó trong biểu thức dạng trung tố.

a - b * c * d - e + f => (+ (- a (* b c d) e) f)

Câu 9: Cho một cấu trúc dữ liệu được định nghĩa trên Scala như sau:

```
trait Element
case class Many(value:List[Element]) extends Element
case class Inte(value:Int) extends Element
case class Flt(value:Float) extends Element
```

và hàm `sum(lst:List[T],f:T=>Int):Int` trả về tổng các giá trị nguyên được ánh xạ từ các phần tử của danh sách lst qua hàm f. Ví dụ sử dụng hàm sum để tính tổng của một danh sách: `sum(List(1,3,4),(x:Int)=>x)` sẽ trả về 8.

Hãy viết lệnh gọi hàm sum trên để đếm số phần tử Inte trực tiếp (phần tử của danh sách) và gián tiếp (đệ qui trong các phần tử của danh sách) trong một danh sách các Element?

Ví dụ:
`val lst = List(Many(List(Inte(3), Many(List(Flt(2.1f), Inte(5))), Flt(1.0f), Inte(2))), Inte(1))` và `sum(lst,...)` sẽ trả về 4, với ... là nội dung cần phải thực hiện cho câu này.

```
def count():
sum(
      lst, e:Element =>
            if(e.isInstanceOf[Inte]) 1
            else if(e.isInstanceOf[Flt]) 0
            else reduce(lambda x, y: x + sum(y),e, 0)
)
```

Câu 10: Hãy giải thích các phương pháp (reference counting, mark and sweep) để tự động dọn rác (garbage collector), nêu điểm mạnh và điểm yếu?

➔ Reference counting
- Khi một object được cấp phát trong heap, một biến đếm (counter) được cấp phát theo và được khởi tạo giá trị là 1.
- Khi có pointer tham chiếu tới object (referencing), counter tăng lên 1
- Khi có pointer ngừng trỏ tới object(dereferencing), counter giảm 1
- Khi counter=0, liên kết bị hủy, object bị deallocated và trả về cho hệ thống. Nhưng nếu object chứa pointer trỏ tới đối tượng khác nữa, thì việc giảm counter (đôi khi là deallocated) phải được thực hiện đệ quy.
- **Điểm mạnh**: Phù hợp với các real-time system có bộ nhớ hạn chế, biến sẽ được deallocated ngay khi không sử dụng nữa.
- **Điểm yếu**: Chi phí để duy trì trình dọn rác chạy liên tục là không nhỏ. Không thể deallocated các object chứa pointer mà chúng là tham chiếu vòng (A trỏ đến object B, B trỏ đến object C, C trỏ ngược lại B).

➔ Mark and Sweep
- Mark : Nhận biết các đối tượng không được sử dụng nữa.
  - Đầu tiên: Duyệt hết tất cả các đối tượng trong heap và gán nhãn "not in used"
  - Sau đó: Duyệt tuần tự tất cả, với mỗi pointer có trong stack, duyệt đệ quy tiếp các object trong heap nếu chúng là pointer theo depth-first, gán chúng với nhãn là "in used".
- Sweep: Deallocated tất cả các object có nhãn là "not in used"
- Mark and Sweep garbage collector sẽ thực thi khi bộ nhớ gần đầy hoặc theo lịch trình được cài đặt sẵn.
- **Điểm mạnh**: Deallocated được trong trường hợp tham chiếu vòng. Không chạy song song với chương trình nên không ảnh hưởng đến thời gian thực thi của thuật toán.
- **Điểm yếu**: Khi garbage collector được gọi, chương trình vào trạng thái treo (suspended) gây trì trệ hệ thống. Gây phân mảnh bộ nhớ, xuất hiện các vùng nhớ nhỏ, không đủ để cấp phát cho nhiều các object.

➔ Mark and Compact
- Mark :Giống Mark and Sweep
- Compact: Thay vì sweep thì giờ compact
- Cải tiến Mark and Sweep bằng cách xử lí tối ưu tình trạng phân mảnh sau khi deallocated
- Cần duyệt 3 lần:
  * Lần 1: Tính toán vị trí mới cho live object
  * Lần 2: Update internal pointer
  * Lần 3: Move object đến vị trí mới
- **Điểm mạnh**:
  * Các live object nằm liền kề nhau
  * Tối ưu tình trạng phân mảnh bộ nhớ
  * Các free block nằm liền kề nhau và chỉ cần 1 pointer quản lý
- **Điểm yếu**: Lâu (much more than M&S)