Cho biết trong đoạn mã sau, đối tượng nào có nhiều tên?

int *p = new int;

int *q = p;

- ○ p
- ○ q
- ◉ Đối tượng được tạo ra bởi new int
- ○ int

Đúng

Đối tượng dữ liệu có nhiều tên trong đoạn mã C trên có những tên gì?

- ○ p
- ○ p và q
- ○ *p và q
- ◉ *p và *q

Đúng, dựa vào 2 tên này, programmer có thể truy xuất (đọc, viết) trên đối tượng được tạo ra bởi new int

Cho đoạn mã C++ sau:

```
template <class myType>
myType GetMax (myType a, myType b) {
   return (a>b?a:b);
}
```

Cho biết đoạn mã trên có thể là ví dụ minh hoạ cho khái niệm nào?

- ○ Alias - bí danh
- ◉ Polymorphism - Đa hình
- ○ Garbage-Rác
- ○ Dangling reference-Tham chiếu treo

Đúng, đây là một ví dụ của đa hình (polymorphirsm): cũng với 1 tên là GetMax nhưng có thể tham chiếu đến những hàm khac nhau tuỳ theo kiểu của myType

Given the following code written in C:

x = a * b;

What is the binding time of variable x and its type?

- (●) Programming Time - Thời gian lập trình
- ( ) Language Design Time- Thời gian định nghĩa ngôn ngữ
- ( ) Language Implementation Time - Thời gian hiện thực ngôn ngữ
- ( ) Runtime - Thời gian chạy

Correct, in C, all user-defined variables must be declared before they are referred and their type must be provided in their declaration; therefore, the binding time between variable and its type is programming time

Cho x trong đoạn mã của câu trên là biến toàn cục. Cho biết thời gian xảy ra ràng buộc giữa biến x và địa chỉ tuyệt đối của x trong bộ nhớ của máy tính?

- ( ) Compiling time - Thời gian dịch
- ( ) Linking time- Thời gian kết nối
- (●) Loading time - Thời gian nạp
- ( ) Runtime - Thời gian chạy

Đúng, thời gian nạp là lúc loader nạp chương trình thực thi từ bộ nhớ ngoài vào bộ nhớ trong. Tuỳ theo thực tế của bộ nhớ trong lúc chương trình được nạp mà chương trình sẽ được cấp phát vào một vùng nhớ phù hợp. Sau khi cấp phát thì các biến toàn cục mới có địa chỉ tuyệt đối và khi đó, ràng buộc mới xảy ra,

Cho biết trên ngôn ngữ C, ràng buộc giữa dấu * và nghĩa của nó (phép nhân) xảy ra vào thời gian nào?

- (●) Language design time - Thời gian định nghĩa ngôn ngữ
- ( ) Runtime
- ( ) Language Implementation Time - Thời gian hiện thực ngôn ngữ
- ( ) Programming time

Đúng, trong các ngôn ngữ thủ tục như C, các hàm có tên dạng ký hiệu như dấu * được qui định ngay trong đặc tả ngôn ngữ (như AND trong ngôn ngữ Pascal và && trong ngôn ngữ C). Vì vậy ràng buộc giữa dấu * và nghĩa của nó xảy ra ngay trong thời gian định nghĩa ngôn ngữ.

Given the following C fragment:

```
int x;
void foo(int y) {
   static int z;
   int * t = malloc(sizeof(int));
   ...
}
```

Choose the WRONG statement?

- ○ x is allocated in static memory
- ○ y is allocated in stack memory
- ◉ z is allocated in static memory
- ○ t is allocated in heap memory

Correct

Given the following C fragment:

```
int x;
void foo(int y) {
   static int z;
   int * t = malloc(sizeof(int));
   ...
}
```

Choose the WRONG statement?

- ○ The lifetime of x is the same as the lifetime of the whole program
- ○ The lifetime of y is the same as the lifetime of function foo
- ○ The lifetime of z is the same as the lifetime of function foo
- ◉ The lifetime of t is the same as the lifetime of function foo

Correct, t is a local variable of pointer type so it lives just when the function runs. The lifetime of the object t points to however is not the same as the lifetime of function foo.

Given the following C code:

```
//position 1
int * foo() {
    // position 2
   x[0] = 1;
   return x;
}
```

which declaration of x and at which position can cause a runtime error (dangling reference or garbage)?

- int x[10]; //position 1
- **int x[10]; //position 2**
- static int x[10]; // position 2
- int* x = malloc(10*sizeof(int)); // position 2

Correct, an object declared at position 2(i.e. local) will be allocated in stack area. Its lifetime is the same as the lifetime of the enclosed function (i.e. foo); it is allocated when the function runs and destroyed when the function is terminated. The instruction "return x" gives the reference to the object to some variable outside the functions. The reference to the object still exists after the object is killed when the function is terminated. This scenario causes "dangling reference"