Question 1

Not answered

Mark 0.00 out of 10.00

Con trỏ - Pointers

Con trỏ (Pointers), tham chiếu (References) và cấp phát bộ nhớ động (Dynamic Memory Allocation) là những tính năng cực kì mạnh mẽ trong ngôn ngữ lập trình C/C++. Những tính năng này cho chép các lập trình viên quản lý và tối ưu tài nguyên của máy tính một cách hiểu quả nhất. Tuy nhiên, đây cũng là nội dung khá phức tạp và gây khó khăn cho người mới bắt đầu.

Con trỏ (Pointers) cho phép chúng ta truy cập và kiểm soát nội dung của các vùng nhớ trong bộ nhớ máy. Nếu được sử dụng đúng cách, con trỏ có thể giúp tăng hiệu năng của chương trình rất nhiều. Tuy nhiên, sử dụng con trỏ không hợp lý có thể dẫn tới rất nhiều vấn đề khác nhau, từ việc làm cho mã nguồn (code) trở nên khó đọc và khó bảo trì cho đến gây ra các lỗi nguy hiểm như thất thoát bộ nhớ (memory leaks) hay tràn bộ nhớ đệm (buffer overflow). Các hacker có thể dựa vào những lỗi này để gây hại cho chương trình của bạn. Nhiều ngôn ngữ lập trình gần đây (như Java hay C#) đã loại bỏ các cú pháp sử dụng con trỏ để tránh những rắc rối mà nó có thể gây ra.

1. Biến con trỏ

Trong máy tính của chúng ta, mỗi ô nhớ (computer memory location) có một địa chỉ (address) và dữ liệu (content) tại ô nhớ đó. Địa chỉ (address) của ô nhớ là một số thường được biểu diễn ở hệ 16 (hexa), ví dụ `0x7ffc7570aa94`. Giá trị của ô nhớ thường được lưu ở dạng nhị phân (binary), người lập trình có thể dịch nó thành các kiểu dữ liệu khác nhau như số nguyên `int`, số thực `double`, kí tự `char` hay xâu `string`.

Tuy nhiên, thông thường các lập trình viên thường gặp khó khăn trong việc sử dụng con trỏ một cách trực tiếp, vì thế các ngôn ngữ lập trình đã đưa ra khai niệm biến (variable). Thông qua tên của biến, lập trình viên có thể truy cập đến dữ liệu của vùng nhớ có địa chỉ mà hệ điều hành đã cấp phát cho biến đó. Cùng với tên biến, lập trình viên cũng phải xác định kiểu của biến (ví dụ: `int`,`double`,`char`) để giúp chúng ta có thể dễ dàng dịch nội dung của biến từ dạng nhị phân (kiểu dữ liệu dùng để lưu trong bộ nhớ máy tính) sang dạng dữ liệu phù hợp mà con người có thể đọc và hiểu được.

Mỗi ô nhớ trong bộ nhớ máy tính thường có kích cỡ 8 bit (1 byte). Để lưu một biến kiểu nguyên `int` 4-bytes trong bộ nhớ, chúng ta cần sử dụng 4 ô nhớ.

Hình vẽ dưới đây biểu diễn mỗi quan hệ giữa địa chỉ (address) và nội dung (content) của bộ nhớ máy tính (computer memory) với tên (name), kiểu (type) và giá trị (value) của biến được sử dụng khi lập trình.

1.1 BIẾN CON TRỎ (POINTER VARIABLE)

Một biến con trỏ (pointer), giống như một biến thông thường khác như `int` hay `double`, được dùng để lưu một giá trị nào đó. Nhưng thay vì được dùng để lưu các giá trị số nguyên (`int`), số thực (`double`) ..., thì biến con trỏ lưu địa chỉ (address) của một vùng nhớ trong bộ nhớ máy tính.

1.2 KHAI BÁO BIẾN CON TRỎ

Tương các biến thông thường khác, biến con trỏ cần phải được khai báo trước khi sử dụng. Để khai báo một biến con trỏ, chung ta cần phải thêm dấu `*` vào trước tên biến. Biến con trỏ cũng cần được khai báo kiểu, ví dụ `int`, `double`, `boolean`.

```
// Cách khai báo con trỏ
type *ptr; // Cách 1
// hoặc
type* ptr; // Cách 2
// hoặc
type *ptr; // Cách 3
```

Ví dụ, chúng ta có thể khai báo các biến con trỏ như sau:

```
        int *iPtr; // Khai báo một biến con trỏ kiểu int tên là iPtr trỏ đến một vùng nhớ chứa số nguyên.
double * dPtr; // Khai báo một biến con trỏ kiểu double
```

Chú ý rằng, chúng ta cần phải đặt một dấu `*` trước **từng** tên biến con trỏ mà chúng ta muốn khai báo. Điều này có nghĩa là dấu `*` chỉ có tác dụng với tên biến đi theo ngay sau đó. Khi viết theo cú pháp khai báo con trò, dấu `*` không phải là một phép toán nhân mà chúng ta hay dùng.

```
    int *p1, *p2, i; // p1 và p2 là biến con trỏ, i là một biến thông thường kiểu int.
int* p1, p2, i; // p1 là biến con trỏ, p2 và i là biến thông thường kiểu int.
int * p1, * p2, i; // p1 và p2 là biến con trỏ, i là biến thông thường kiểu int.
```

Quy uóc đặt tên biến: Chúng ta nên đặt tên cho biến con trỏ bắt đầu bằng chữ `p` hoặc kết thúc bằng chữ `Ptr` để giúp mã nguồn của chúng ta dễ đọc và bảo trì hơn. Ví dụ: `iPtr`, `numberPtr`, `pNumber`, `pStudent`.

1.3 KHỞI TẠO BIẾN CON TRỎ SỬ DỤNG PHÉP & (ADDRESS-OF-OPERATOR)

Ban đầu, khi khai báo một biến con trỏ, nội dung của biến con trỏ sẽ được gán bằng một địa chỉ ngẫu nhiên nào đó mà chúng ta không quan tâm tới (địa chỉ không hợp lệ). Điều này **cực kì nguy hiểm** vì nó có thể gây hại đến các các chương trình khác đang chạy trong máy tính. Vì vậy, chúng ta cần phải khởi tạo biến con trỏ bằng cách gán nó với một địa chỉ mà chúng ta muốn làm việc với nó. Chúng ta có thể gán con trỏ với địa chỉ của một biến thông thường thông qua phép `&`.

Phép `&` (address-of-operator) trả về địa chỉ của biến thông thường. Ví dụ, nếu biến `number` là một biến thông thường kiểu `int`, lệnh `&number` sẽ trả về địa chỉ trong bộ nhớ máy tính mà biến `number` được lưu.

Ví dụ:

```
int number = 88; // Khai báo biến thông thường và gán gía trị
int *pNumber; // Khai bảo biến con trỏ
pNumber = &number; // Khởi tạo biến con trỏ, gán giá trị con trỏ bằng địa chỉ của biến number


int *numberPtr = &number; // Chúng ta cũng có thể khởi tạo con trỏ ngay sau lệnh khai báo
```

Như đã mô tả ở hình trên, biến `number` chứa số nguyên `int` có giá trị bằng 88, và được lưu trong 4 ô nhớ bắt đầu từ ô có địa chỉ `0x22ccec`. Câu lệnh `&number` trả về địa chỉ của biến `number` (là địa chỉ của ô nhớ đầu tiên, `0x22ccec`). Địa chỉ này sau đó được gán vào (khởi tạo) biến `pNumber`.

1.4 PHÉP LẤY GIÁ TRỊ CON TRỎ *

Khi sử dụng dấu `*` trước một biến con trỏ, chúng ta sẽ lấy được giá trị của ô nhớ mà con trỏ đó chỉ tới. Ví dụ, nếu `pNumber` là một biến con trỏ kiểu `int`, lệnh `*pNumber` sẽ trả về giá trị của ô nhớ mà `pNumber` chỉ tới.

Ví dụ:

```
    int number = 98; // Khai báo biến kiểu int
int* pNumber = &number; // Khai báo biến con trỏ kiểu int và khởi tạo nó bằng địa chỉ của biến number
cout << pNumber << endl; // In ra giá trị của con trỏ pNumber, là một địa chỉ
cout << *pNumber << endl; // In ra giá trị của ô nhớ mà biến pNumber chỉ đến
*pNumber = 99; // Sửa giá trị của vùng nhớ mà biến pNumber chỉ tới.
cout << *pNumber << endl; // In ra gía trị của vùng nhớ vừa bị thay đổi
cout << number << endl; // Giá trị của biến number cũng thay đổi theo.
```

Lưu ý: Phép `*` có ý nghĩa khác nhau khi sử dụng trong câu lệnh khai báo và trong biểu thức. Khi dấu `*` được sử dụng trong khai báo biến (ví dụ: `int * pNumber;`), nó chỉ ra rằng biến được khai báo ngay phía sau đấu `*` là một biến con trỏ. Khi sử dụng trong một biểu thức (ví dụ: `*pNumber = 99; std::cout << *pNumber;`), nó trả về giá trị của vùng nhớ mà con trỏ chỉ tới.

1.5 KIỂU CỦA BIẾN CON TRỎ

Trong quá trình khai báo biến con trỏ, lập trình viên phải xác định kiểu dữ liệu của con trỏ đó. Khi đã khai báo, một con trỏ chỉ có thể chứa địa chỉ của vùng nhớ lưu kiểu dữ liệu đã khai báo cho con trỏ đó và không thể chứa kiểu dữ liệu khác.

Ví dụ **đúng**:

```
    int i = 88;
double d = 55.66;
int * iPtr = &i;    // Con trỏ kiểu int trỏ đến vùng nhớ chứa dữ liệu kiểu int
double * dPtr = &d; // Con trỏ kiểu double trỏ đến vùng nhớ chứa dữ liệu kiểu double
```

Ví dụ **sai**:

```
       int i = 88;
double d = 55.66;
int * iPtr = &d;    // LỖI: Con trỏ kiểu int không được chứa địa chỉ vùng nhớ chứa dữ liệu kiểu double
double * dPtr = &i; // LỖI: Con trỏ kiểu double không được chứa địa chỉ vùng nhớ chứa dữ liệu kiểu int
iPtr = i; // LỖI: Con trỏ chỉ chứa địa chỉ, không chứa giá trị.
```

1.6 LỖI CON TRỎ CHƯA KHỞI TẠO

Đoạn code sau đây có một lỗi nguy hiểm mà các lập trình viên hay mắc phải:

```
   int * iPtr;
*iPtr = 55;
cout << *iPtr << endl;
```

Biến con trỏ `iPtr` đã được khai báo nhưng chưa khởi tạo. Vì vậy, hệ điều hành sẽ gán cho nó ngẫu nhiên một giá trị địa chỉ trỏ đến một vùng nhớ ngẫu nhiên nào đó trong máy tính, vùng nhớ này có thể nằm ngoài quyền kiểm soát của chương trình đang chạy. Khi thực thi lệnh `*iPtr = 55;`, chương trình vô hình dung thay đổi giá trị ở một ô nhớ nào đó một cách không hợp pháp. Điều này có thể gây ra lỗi và làm cho chương trình ngừng hoạt động. Hơn nữa, hầu hết các trình biên dịch (compiler) không thông báo lỗi (error) hay cảnh báo (warning) cho lập trình viên về lỗi này trong quá trình biên dịch, nên chúng ta phải tự kiểm tra lại các lỗi kiểu như trên trước khi biên dịch chương trình.

1.7 CON TRỎ RỖNG - NULL

Chúng ta có thể khởi tạo một biến con trỏ bằng giá trị `0` hoặc `NULL`, khi đó con trỏ sẽ không trỏ đến bất cứ một vùng nhớ nào cả. Truy cập vào con trỏ có giá trị `NULL` sẽ gây ra lỗi `STATUS_ACCESS_VIOLATION`.

```
   int * iPtr = 0; // Khai báo biến con trỏ iPtr và gán nó với giá trị NULL
cout << *iPtr << endl; // LỖI: STATUS_ACCESS_VIOLATION
int *p = NULL; // Cách khác để khai con trỏ Null.
```

Bài tập

Hàm `double* getPointerToPi()` có nhiệm vụ như sau:

- Khai báo một biến kiểu `double` đặt tên là `pi` ở ngoài hàm (biến toàn cục) và gán giá trị cho của số $\pi = 3.14159$ cho biến đó.
- Khai báo một biến con trỏ kiểu `double*` và khởi tạo con trỏ bằng địa chỉ của biến `pi` đã khai báo ở trên.
- Trả về con trỏ đã khai báo.

Hãy viết mã mã C++ để hoàn thành hàm `double* getPointerToPi()` có yêu cầu như trên.

Answer: (penalty regime: 0 %)

```
1
```

Question 2

Not answered

Mark 0.00 out of 10.00

[Normalization]

3. Cấp phát bộ nhớ động - (Dynamic Memory Allocation)

3.1 TOÁN TỬ NEW VÀ DELETE

Thay vì khai báo một biến `int` (`int number`) và dùng địa chỉ của biến đó để khởi tạo cho một biến con trỏ khác (`int *pNumber = &number`). Lập trình viên có thể yêu cầu hệ điều hành cấp động một vùng nhớ dụng toán tử `new` và dùng con trỏ để lưu lại địa chỉ của vùng nhớ đó. Trong C++, bất kể khi nào bạn sử dụng toán tử `new` để cấp phát động, bạn cần phải sử dụng toán tử `delete` để giải phòng vùng nhớ đó khi không sử dụng đến nó nữa. Trong một số ngôn ngữ khác, như Java, thì việc dọn rác (gabbage collection) được thực hiện một cách tự động.

Toán tử `new` trả về địa chỉ của vùng nhớ được cấp phát. Toán tử `delete` nhận con trỏ làm đối số và giải phóng vùng nhớ mà con trỏ đó trỏ tới.

Ví dụ:

```
    // Cấp phát tĩnh
int number = 88;
int * p1 = &number;  // Gán địa chỉ của một biến vào con trỏ

// Cấp phát động
int * p2;            // Chưa được khởi tạo, con trỏ được trỏ đến một vùng nhớ ngẫu nhiên trong bộ nhớ
cout << p2 << endl; // In ra địa chỉ trước khi cấp phát bộ nhớ
p2 = new int;       // Cấp phát bộ nhớ động và gán địa chỉ đã cấp phát cho con trỏ

*p2 = 99;
cout << p2 << endl;  // In địa chỉ sau khi đã cấp phát động
cout << *p2 << endl; // In ra giá trị ô nhớ mà con trỏ chỉ tới
delete p2;           // Giải phóng vùng nhớ đã cấp phát động
```

Để khởi tạo vùng nhớ đã được cấp phát, cúng ta có thể sư dụng bộ khởi tạo (initializer) đối với các biến kiểu cơ bản (nguyên thủy) hoặc gọi hàm khởi tạo đối với các đối tượng (object) với từ khóa `new` theo phía trước.

Ví dụ:

```
    // Sử dụng bộ khởi tạo với các biến kiêu nguyên thủy.
int * p1 = new int(88);
double * p2 = new double(1.23);

// Gọi hàm khởi tạo đối với các đối tượng (ví dụ Date, Time)
Date * date1 = new Date(1999, 1, 1);
Time * time1 = new Time(12, 34, 56);
```

3.2 MẢNG ĐỘNG VỚI TOÁN TỬ NEW [] VÀ DELETE []

Thay vì sử dụng mảng có kích thước cố định (mảng tĩnh), chúng ta có thể cấp phát mảng động bằng cách sử dụng toán tử `new[]`. Để giải phóng vùng nhớ mà đã cấp phát cho mảng động, chúng ta sử dụng toán tử `delete []`.

`/* Cấp phát mảng động */ #include #include using namespace std; int main() { const int SIZE = 5; int * pArray; pArray = new int[SIZE]; // Cấp phát mảng động thông qua toán tử new[] // Gán mỗi phần tử của mảng với một số ngẫu nhiên nằm trong khoảng 1 và 100 for (int i = 0; i < SIZE; ++i) { *(pArray + i) = rand() % 100; } // In ra mảng for (int i = 0; i < SIZE; ++i) { cout << *(pArray + i) << " "; } cout << endl; delete[] pArray; // Giải phóng vùng nhớ thông qua toán tử new[] return 0; }`

4. Con trỏ, mảng và hàm

4.1 MẢNG LÀ CON TRỎ

Trong C/C++, tên của mảng là con trỏ, chỉ đến địa chỉ của phần từ đầu tiên của mảng. Ví dụ, biến `numbers` là một mảng kiểu `int`, thì `numbers` đồng thời cũng là một con trỏ kiểu `int`, chỉ đến ô nhớ chứa phần tử `numbers[0]`. Vì vậy, `numbers` là `&numbers[0]`, `*number` là `numbers[0]`, `*(numbers+i)` là `numbers[i]`.

Ví dụ:

```
/* Con trỏ và mảng */
#include
using namespace std;

int main() {
    const int SIZE = 5;
    int numbers[SIZE] = {11, 22, 44, 21, 41};  // Mảng số nguyên int

    // Tên mảng numbers là một con trỏ, trỏ đến phần tử đầu tiền của mảng
    cout << &numbers[0] << endl; // In ra địa chỉ của phần tử đầu tiền của mảng
    cout << numbers << endl;     // Tương tự như trên
    cout << *numbers << endl;         // Giống như numbers[0] (11)
    cout << *(numbers + 1) << endl;   // Giống như numbers[1] (22)
    cout << *(numbers + 4) << endl;   // Giống như numbers[4] (41)
}
```

4.2 CÁC PHÉP TOÁN TRÊN CON TRỎ

Nếu `numbers` là một mảng `int`, nó được xem như là con trỏ chỉ đến phần tử đầu tiên của mảng `numbers[0]`. Khi đó, `numbers+1` trỏ đến phần tử tiếp theo, chứ không phải địa chỉ của ô nhớ tiếp theo trong bộ nhớ. Nhớ rằng, kiểu `int` có kích thước `4` bytes. Vì vậy, `numbers +1` sẽ chỉ đến ô nhớ sau ô nhớ hiện tại mà `numbers` trỏ tới 4 ô.

```
int numbers[] = {11, 22, 33};
int * iPtr = numbers;
cout << iPtr << endl;        // In ra địa chỉ hiện tại (ở dạng hexa): 0x22cd30 (2280752)
cout << iPtr + 1 << endl;    // In ra địa chỉ sau khi cộng 1: 0x22cd34 (2280756 = 2280752 + 4)
cout << *iPtr << endl;       // 11
cout << *(iPtr + 1) << endl; // 22
cout << *iPtr + 1 << endl;   // 12
```

4.3 PHÉP LẤY KÍCH THƯỚC`SIZEOF`

Phép `sizeof(arrayName)` trả về tổng số `bytes` (kích thước) của mảng. Chúng ta có thể tính số phần tử của mảng bằng cách chia số `bytes` của cả mảng cho kích thước của một phần tử trong mảng. Ví dụ:

```
int numbers[100];
cout << sizeof(numbers) << endl;     // Kích cỡ của cả mảng (400)
cout << sizeof(numbers[0]) << endl;  // Kích cỡ của một phần tử trong mảng
cout << "Array size is " << sizeof(numbers) / sizeof(numbers[0]) << endl;  // Số lượng phần tử trong mảng (100)
```

4.4 TRUYỀN MẢNG VÀO HÀM

Khi mảng được truyền vào hàm, trình biên dịch sẽ xem nó như một con trỏ. Khi khai báo đối số cho hàm, chúng ta có thể sử dụng cú pháp mảng `int[]` hoặc cú pháp con trỏ `int *`. Ví dụ, những cách khai báo hàm dưới đây là giống nhau.

```
int max(int numbers[], int size);
int max(int *numbers, int size);
int max(int number[50], int size);
```

Tất cả đối số `numbers` trong các khai báo trên đều được xem như là một biến con trỏ. Hơn nữa, khi truyền vào hàm, thông tin về kích cỡ của mảng sẽ bị mất, vì vậy thông thường chúng ta thường phải truyền thêm một đối số `size` kèm theo tên của mảng để xác định kích cỡ của mảng. Khi truy cập các phần tử của mảng trong hàm, trình biên dịch cũng không kiểm tra liệu chúng ta có truy cập

ngoài mảng hay không.Vì thế, chúng ta phải cẩn thận kiểm tra khi sử dụng mảng trong hàm.

Bài tập

Một véc-tơ n chiều: $\vec{x} = (x_1, x_2, \ldots, x_n)$ có thể biểu diễn bằng một mảng gồm n số. Trong nhiều bài toán người ta muốn giá trị các chiều của véc-tơ nằm trong đoạn $[0, 1]$ (hoặc $[-1, 1]$) tức là $x_i \in [0, 1] \forall i$. Một cách để làm việc đó là chia các phần tử của mảng cho số lớn nhất có thể có của các phần tử đó.

Hàm `void normalize(double *out, int *in, int n)` nhận các tham số là:

- Con trỏ trỏ đến mảng đầu vào `in`. Mảng đầu vào chứa các số nguyên trong đoạn $[0, 255]$.
- Con trỏ trỏ đến mảng đầu ra `out`. Mảng đầu ra là mảng chuẩn hóa của mảng đầu vào, chứa các số thực sau khi chia số nguyên tương ứng của mảng đầu vào cho 255.
- Số nguyên `n` là số phần tử của hai mảng.

Nhiệm vụ của hàm `void normalize(double *out, int *in, int n)` là chuẩn hóa các giá trị trong mảng đầu vào `in` về khoảng $[0, 1]$ và lưu vào mảng đầu ra `out`.

Hãy viết mã C++ để hoàn thành hàm `void normalize(double *out, int *in, int n)` thực hiện các yêu cầu trên.

For example:

Input	Result
5 78 167 90 119 99	0.306 0.655 0.353 0.467 0.388

Answer: (penalty regime: 0 %)

```
1
```

Question 3

Not answered

Mark 0.00 out of 10.00

[Cube]

Lũy thừa mũ 3 được sử dụng để tính thể tích của khối lập phương.

Viết hàm `void cube(double *p)` nhận tham số là con trỏ đến một biến thực. Hàm thực hiện phép chỉnh sửa giá trị của biến mà con trỏ trỏ đến thành lập phương giá trị đó.

For example:

Input	Result
`124.45319`	`1927605.24319`

Answer: (penalty regime: 0 %)

```
1
```

Question 4

Not answered

Mark 0.00 out of 10.00

[HigherOrLower]

Cho một dãy gồm n số nguyên và một ngưỡng nguyên (threshold). Viết hàm kiểm tra xem các số trong dãy cao hơn hay thấp hơn ngưỡng cho trước.

Hàm `bool* isHigher (int* arr, int num, int thres)` nhận đầu vào là mảng arr có n số nguyên và một ngưỡng $thres$.

Hàm kiểm tra và trả về một mảng số kiểu $bool$ với phần tử thứ i là $true$ nếu số nguyên thứ i trong mảng arr lớn hơn hoặc bằng ngưỡng $thres$, và bằng $false$ trong trường hợp ngược lại.

For example:

Input	Result
6 67 13 99 14 41 20 15	1 0 1 0 1 1

Answer: (penalty regime: 0 %)

```
1
```

Question 5

Not answered

Mark 0.00 out of 10.00

[ImageFilter]

Giả sử bạn được thuê xây dựng một hệ thống xử lý ảnh.

Trong đó có một bước lọc ảnh. Tại đây, bạn phải đọc ảnh vào và kiểm tra giá trị tại từng điểm ảnh (**pixel**) có lớn hơn hoặc bằng một ngưỡng **(threshold)** cố định không.

Nếu có, giá trị tại pixel đó giữ nguyên, nếu không, giá trị tại đó được gán giá trị bằng 0.

Viết hàm `int** getImage (int nRows, int nCols)` đọc vào một ảnh có kích thước $nRows \times nCols$. Hàm trả về một con trỏ trỏ tới con trỏ lưu trữ ảnh có kiểu **int****.

Viết hàm `void fillImage (int** image, int nRows, int nCols, int threshold)` thực hiện việc lọc ảnh với ngưỡng cho trước $threshold$. Hàm lọc và thay đổi giá trị của ảnh lưu bằng con trỏ $image$.

Cuối cùng, viết hàm `void print (int** image, int nRows, int nCols)` để in ra ảnh sau khi đã lọc.

Lưu ý: Các điểm ảnh là số nguyên có giá trị nằm trong khoảng từ 0 đến 255.

For example:

Input	Result
5 2 126 82 26 132 81 153 106 185 207 85 95	126 0 0 132 0 153 106 185 207 0

Answer: (penalty regime: 0 %)

```
1
```

Question 6

Not answered

Mark 0.00 out of 10.00

[Multiply]

Khi truyền một con trỏ vào hàm, mọi thay đổi với biến con trỏ sẽ làm thay đổi giá trị của biến tương ứng bên ngoài hàm.

Viết hàm `void multiply (int* n, int k)` thực hiện phép nhân giá trị biến n lên k lần.

Biết hàm nhận đối số lần lượt là kiểu con trỏ và tham trị.

For example:

Input	Result
2 3	6

Answer: (penalty regime: 0 %)

```
1
```

Question 7

Not answered

Mark 0.00 out of 10.00

[PointerToArray]

Viết hàm `int* getPointerToArray(int n)`. Hàm này khai báo một con trỏ kiểu nguyên, cấp phát bộ nhớ động cho con trỏ đó n phần tử kiểu nguyên và sau đó gán giá trị cho n phần tử đó các số được nhập từ bàn phím. Hàm trả về con trỏ được khai báo.

For example:

Input	Result
3 -5021 4497 -9846	-5021 4497 -9846

Answer: (penalty regime: 0 %)

```
1
```

Question 8

Not answered

Mark 0.00 out of 10.00

[Reference Variable - Swap]

2. Biến tham chiếu (Reference Variable)

Một biến tham chiếu là một **alias**, hay là một tên khác của một biến đã tồn tại. Khi một biến được khai báo là tham chiếu của một biến khác, chúng ta có thể truy cập một biến sử dụng tên thật hoặc tên tham chiếu của nó.

Tham chiếu chủ yếu được dùng để truyền vào các hàm (pass-by-reference). Khi tham chiếu được truyền vào hàm, hàm sẽ làm việc trực tiếp với biến gốc thay vì tạo ra một bản copy (clone) và làm việc trên bản copy đó. Như vậy, mọi thay đổi trên biến sẽ được lưu lại trên biến gốc sau khi kết thúc hàm.

Biến tham chiếu có nhiều điểm tương tự với biến con trỏ. Trong nhiều trường hợp, biến tham chiếu có thể được sử dụng thay thế cho con trỏ, đặc biệt khi được sử dụng làm đối số của các hàm.

2.1 THAM CHIẾU &

Ở bài học trước, chúng ta sử dụng phép & để lấy địa chỉ của một biến trong các biểu thức. Ở phần này, chúng ta sẽ học thêm một cách sử dụng nữa của phép & để khai báo biến tham chiếu của một biến khác.

Phép & có ý nghĩa khác nhau khi sử dụng trong biểu thức và khi khai báo biến. Khi được sử dụng trong một biểu thức, phép & đóng vai trò là phép lấy địa chỉ (address-of operator), trả về địa chỉ của một biến (Ví dụ: nếu `number` là một biến kiểu `int`, `&number` sẽ trả về địa chỉ của vùng nhớ chứa biến `number`).

Tuy nhiên, khi & được sử dụng trong khai báo biến, thì biến được khai báo ngay sau đó được xem là biến tham chiếu. Biến tham chiếu thường được dùng như tên thay thế của một biến khác đã được khai báo trước đó.

Để khai báo một biến tham chiếu, chúng ta sử dụng cú pháp sau:

```
type &newName = existingName;
// hoặc
type& newName = existingName;
// hoặc
type & newName = existingName;
```

Ở các cách khai báo trên, biến `newName` tham chiếu đến biến `existingName`. Chúng ta có thể truy cập đến biến `existingName` sử dụng chính tên gốc của biến đó hoặc qua tham chiếu `newName`.

Ví dụ:

```
/* Khai báo và khởi tạo tham chiếu */
#include
using namespace std;

int main() {
    int number = 88;          // Khai báo biến number
    int & refNumber = number; // Khai báo biến refNumber tham chiếu đến biến number.

    cout << number << endl;    // In ra giá trị của biến number (88)
    cout << refNumber << endl; // In ra giá trị của tham chiếu (88)

    refNumber = 99;            // Gán giá trị mới cho tham chiếu refNumber
    cout << refNumber << endl;
    cout << number << endl;    // Giá trị của biến number cũng thay đổi theo (99)

    number = 55;               // Gán gía trị mới cho biến number
    cout << number << endl;
    cout << refNumber << endl; // Giá trị của tham chiếu cũng thay đổi (55)
}
```

2.3 THAM CHIẾU HOẠT ĐỘNG NHƯ THẾ NÀO?

Tham chiếu hoạt động như một con trỏ. Một tham chiếu được được sử dụng như tên thứ 2 của một biến. Nó chứa địa chỉ của biến, như mô tả dưới đấy:

 2.4 Tham chiếu và con trỏ

Con trỏ và tham chiếu có chức năng khá giống nhau trừ một số đặc điểm sau:

- 1. Một tham chiếu (reference) là một biến hằng số lưu địa chỉ. Tham chiếu phải được khởi tạo ngay sau khi khai báo.

```
int & iRef;   // Lỗi: 'iRef' được khai báo là tham chiếu nhưng chưa được khai báo.
```

  Khi đã khai báo và khởi tạo, biến tham chiếu không thể thay đổi giá trị.
- 2. Để truy vấn giá trị mà con trỏ chỉ đến, chúng ta sử dụng phép *. Tương tự, để gán giá trị của con trỏ với địa chỉ của một biến bình thường, chúng ta sử dụng phép &. Tuy nhiên, đối với biến tham chiếu, hai quy trình này được thực hiện một cách ngầm định, chúng ta không cần phải sử dụng các phép toán như trong con trỏ.

Ví dụ:

```
    /* Tham chiếu vs. Con trỏ */
#include
using namespace std;

int main() {
    int number1 = 88, number2 = 22;

    // Tạo một con trỏ trỏ đến biến number1
    int * pNumber1 = &number1;  // Khởi tạo con trỏ
    *pNumber1 = 99;             // Thay đổi giá trị mà con trỏ chỉ đến
    cout << *pNumber1 << endl;  // 99
    cout << &number1 << endl;   // 0x22ff18
    cout << pNumber1 << endl;   // 0x22ff18 (Giá trị của <a class="autolink" title="Biến con trỏ"
href="https://dev.uet.vnu.edu.vn/mod/quiz/view.php?id=4995">biến con trỏ</a> pValue)
    cout << &pNumber1 << endl;  // 0x22ff10 (Địa chỉ của <a class="autolink" title="Biến con trỏ"
href="https://dev.uet.vnu.edu.vn/mod/quiz/view.php?id=4995">biến con trỏ</a>)
    pNumber1 = &number2;        // Con trỏ có thể đổi giá trị để trỏ đến một địa chỉ khác

    // Tạo một tham chiếu cho biến number1
    int & refNumber1 = number1;  // Tham chiếu ngầm định ( không phải &number1)
    refNumber1 = 11;             // Thay đổi giá trị mà tham chiếu chỉ đến  - không cần sử dụng phép *
    cout << refNumber1 << endl;  // 11
    cout << &number1 << endl;    // 0x22ff18
    cout << &refNumber1 << endl; // 0x22ff18
    //refNumber1 = &number2;     // Lỗi: Tham chiếu không thể thay đổi giá trị, nó là một hằng số.
    refNumber1 = number2;        // refNumber1 vẫn là tham chiếu của biến number1
                                 // Lệnh trên chỉ copy giá trị của biến number2 vào tham chiếu refNumber1 (hay number1)
    number2++;
    cout << refNumber1 << endl;  // 22
    cout << number1 << endl;     // 22
    cout << number2 << endl;     // 23
}
```

2.4 TRUYỀN THAM CHIẾU VÀO HÀM SỬ DỤNG BIẾN THAM CHIẾU HOẶC BIẾN CON TRỎ

Truyền bằng giá trị

Trong ngôn ngữ lập trình C/C++, các đối số được truyền vào hàm bằng gía trị (ngoại từ mảng `array`, bởi vì mảng thực chất là con trỏ). Có nghĩa là, các bản sao (clone) của các đối số sẽ được tạo ra và truyền vào hàm. Những thay đổi tác động lên bản sao của các đối số trong hàm sẽ không làm thay đổi bản gốc của đối số - được khai báo bên ngoài hàm.

Ví dụ:

```
    /* Truyền bằng giá trị */
#include
using namespace std;

int square(int);

int main() {
    int number = 8;
    cout <<  "In main(): " << &number << endl;  // 0x22ff1c
    cout << number << endl;         // 8
    cout << square(number) << endl; // 64
    cout << number << endl;         // 8 - bản gốc của đối số : gía trị không thay đổi
}

int square(int number) {  // non-const
    cout <<  "In square(): " << &number << endl;  // 0x22ff00
    number *= number;           // Chỉnh sửa trên bản sao
    return number;
}
```

Ở đoạn code trên, biến `number` ở hàm `main()` và biến `number` ở hàm `square()` là hoàn toàn khác nhau. Việc thay đổi biến `number` ở hàm `square()` không làm thay đổi giá trị của biến `number` ở hàm `main()`.

Truyền tham chiếu sử dụng đối số con trỏ.

Trong nhiều trường hợp, chúng ta muốn chỉnh sửa trực tiếp lên bản gốc của một biến chứ không phải bản copy. Để làm được điều này, chúng ta có thể truyền biến con trỏ làm đối số của hàm.

Ví dụ:

```
    /* Truyền tham chiếu sử dụng biến con trỏ */
#include
using namespace std;

void square(int *);

int main() {
    int number = 8;
    cout <<  "In main(): " << &number << endl;  // 0x22ff1c
    cout << number << endl;   // 8
    square(&number);          // Truyền địa chỉ của biến thay vì truyền tên biến
    cout << number << endl;   // 64
}

void square(int * pNumber) {  // Hàm nhận đối số con trỏ
    cout <<  "In square(): " << pNumber << endl;  // 0x22ff1c
    *pNumber *= *pNumber;      // Thay đổi vào ô nhớ mà biến pNumber chỉ đến.
}
```

Trong ví dụ trên, biến `pNumber` ở hàm `square()` và biến `number` ở hàm `main()` cùng trỏ đến một vùng nhớ trên máy tính, nên thay đổi ở hàm `square()` sẽ tác động lên hàm `main()` và được lưu lại sau khi hàm kết thúc.

Truyền tham chiếu với đối số tham chiếu

Thay vì truyền con trỏ vào hàm, biến tham chiếu có thể được sử dụng thay thế để tránh những rắc rối mà con trỏ đem lại.

Ví dụ:

```
/* Truyền tham chiếu sử dụng đối số tham chiếu */
#include
using namespace std;

void square(int &);

int main() {
    int number = 8;
    cout <<  "In main(): " << &number << endl;  // 0x22ff1c
    cout << number << endl;  // 8
    square(number);          // Truyền tham chiếu
    cout << number << endl;  // 64
}

void square(int & rNumber) {  // Hàm nhận đối số là tham chiếu kiểu int
    cout <<  "In square(): " << &rNumber << endl;  // 0x22ff1c
    rNumber *= rNumber;
}
```

Bài tập

Phép hoán đổi giá trị của hai biến được sử dụng trong nhiều bài toán, chẳng hạn như bài toán sắp xếp. Thông thường, người lập trình sẽ xây dựng một hàm riêng thực hiện phép toán này nhằm mục đích tối ưu việc viết code.

Là một lập trình viên, bạn hãy viết hàm `void swap(int& a, int& b)` thực hiện việc hoán đổi giá trị của hai biến , trong đó hàm nhận đối số đầu vào là hai biến tham số `a` và `b`.

For example:

Input	Result
1 2	2 1

Answer: (penalty regime: 0 %)

```
1
```

Question 9

Not answered

Mark 0.00 out of 10.00

[Swap]

Việc hoán đổi giá trị của hai biến được sử dụng trong nhiều bài toán, chẳng hạn như bài toán sắp xếp. Viết hàm `void swap(int* a, int* b)` thực hiện việc hoán đổi giá trị của hai biến a và b.

For example:

Input	Result
1 2	2 1

Answer: (penalty regime: 0 %)

```
1
```

Question 10

Not answered

Mark 0.00 out of 10.00

[TheMatrix]

Viết hàm `int** inputMatrix(int nRows, int nCols)` đọc từ bàn phím một ma trận số nguyên có số hàng $nRows$ và số cột $nCols$, lưu vào một mảng động hai chiều và trả về con trỏ trỏ đến mảng động này.

Viết hàm `void printMatrix(int** matrix, int nRows, int nCols)` nhận tham số là con trỏ đến mảng động hai chiều $matrix$, số hàng $nRows$ và số cột $nCols$ của ma trận. Hàm này in ma trận đầu vào ra màn hình, các phần tử trên cùng một hàng cách nhau bởi một dấu cách.

For example:

Input	Result
2 3 1 2 3 3 4 5	1 2 3 3 4 5

Answer: (penalty regime: 0 %)

```
1
```

Question 11

Not answered

Mark 0.00 out of 10.00

[SymmetricMatrix]

Trong đại số tuyến tính, một **ma trận đối xứng** là một ma trận vuông, *A*, bằng chính ma trận chuyển vị của nó.

Mỗi phần tử của một ma trận đối xứng thì đối xứng qua đường chéo. Do vậy, nếu các phần tử được viết dưới dạng \(*A=a_{ij}*\), thì $a_{ij} = a_{ji}$.

Ta có thể biểu diễn một ma trận bằng một mảng hai chiều trong máy tính.

Viết hàm `void inputMatrix(int** matrix, int nRows, int nCols)` nhận tham số là con trỏ đến mảng hai chiều biểu diễn ma trận, số hàng và số cột của ma trận. Hàm này lưu các phần tử của ma trận nhập từ bàn phím vào mảng hai chiều biểu diễn ma trận.

Viết hàm `int isSymmetric(int** matrix, int nRows, int nCols)` nhận tham số là con trỏ đến mảng hai chiều biểu diễn ma trận, số hàng và số cột của ma trận. Hàm trả về 1 nếu ma trận đầu vào là ma trận đối xứng, ngược lại trả về 0.

For example:

Input	Result
3 3 1 2 3 2 3 4 3 4 5	1

Answer: (penalty regime: 0 %)

```
1
```

Question 12

Not answered

Mark 0.00 out of 10.00

[EvenNumberOnly]

Viết hàm `int** keepEven (int** matrix, int nRows, int nCols)` kiểm tra ma trận hai chiều.

Hàm nhận đầu vào là ma trận $matrix$ có kích thước $nRows \times nCols$. Hàm trả về một ma trận mới sao cho tất cả các giá trị là số lẻ trong ma trận ban đầu được gán giá trị 0 và giữ nguyên các số chẵn.

Answer: (penalty regime: 0 %)

```
1
```

Back to Course