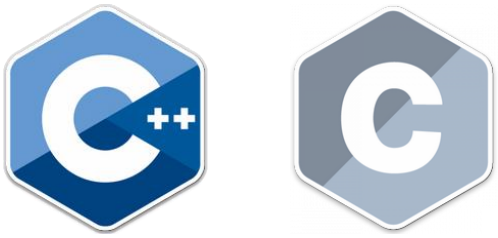
CON TRỎ

Phần 3 – Con trỏ và mảng



[2017 – 04 – 15]

Biên soạn bởi: Nguyễn Trung Thành
https://www.facebook.com/abcxyztcit

“ Học từ cái đáy của thế giới đi lên là cách duy nhất bạn trở thành master. ”

Nguyễn Trung Thành

Mục lục

A. Giới thiệu

1. Điều kiện để học được tài liệu này

- Có kiến thức nền tảng vững chắc với ngôn ngữ C (hoặc C++).
- Đã học xong 2 bài giảng trước đó (con trỏ phần 1 và phần 2).

2. Dẫn nhập

Khi bạn học lập trình cơ bản với C/C++, bạn được học về mảng, chuỗi.

Cụ thể hơn, bạn đã học về ***mảng bình thường***.

Mảng bình thường là mảng có kích thước cố định, khai báo một phát là cố định luôn không có thay đổi kích thước mảng được nữa.

Ví dụ:

```
int a[3];
int n = 0; // số lượng phần tử của mảng a
```

Khi khai báo như vầy, thì ***mảng a sẽ có tối đa 3 phần tử***.

> Ta có thể chỉ sử dụng 3 phần tử, 2 phần tử, 1 phần tử, hoặc không sử dụng phần tử nào của mảng.
>
> Nhưng ta không được phép sử dụng 4 phần tử, 5 phần tử.... vì mảng a chỉ có tối đa 3 phần tử.

Biến n là đại diện số lượng phần tử của mảng a, nó chỉ là một "cách hiểu ngầm". Dù n có giá trị nào đi nữa thì n cũng không được vượt quá 3, không được phép xài nhiều hơn.

Biến n chỉ cho ta biết một ý niệm là "ta đang sử dụng bao nhiêu phần tử của mảng".

Hiểu rõ như vậy thì bạn sẽ học con trỏ dễ dàng hơn.

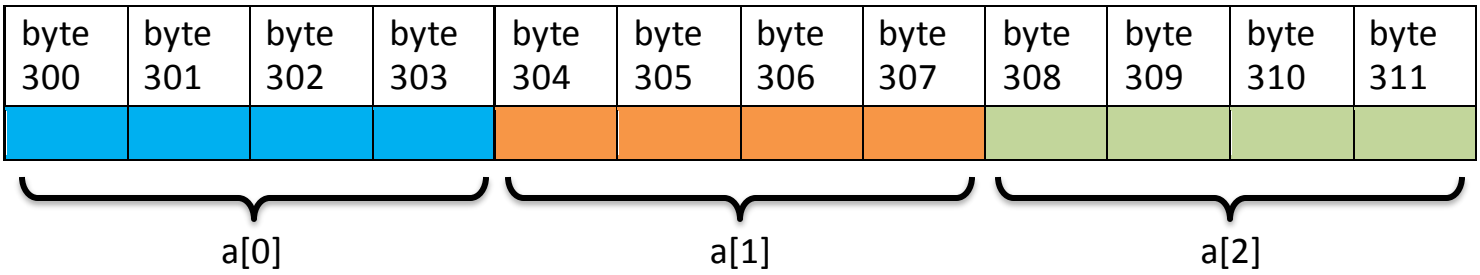
B. Mảng và con trỏ

1. Hình ảnh của mảng trong bộ nhớ

Nối tiếp những gì đã nói ở phần dẫn nhập với mảng a có tối đa 3 phần tử.

```
int a[3];
```

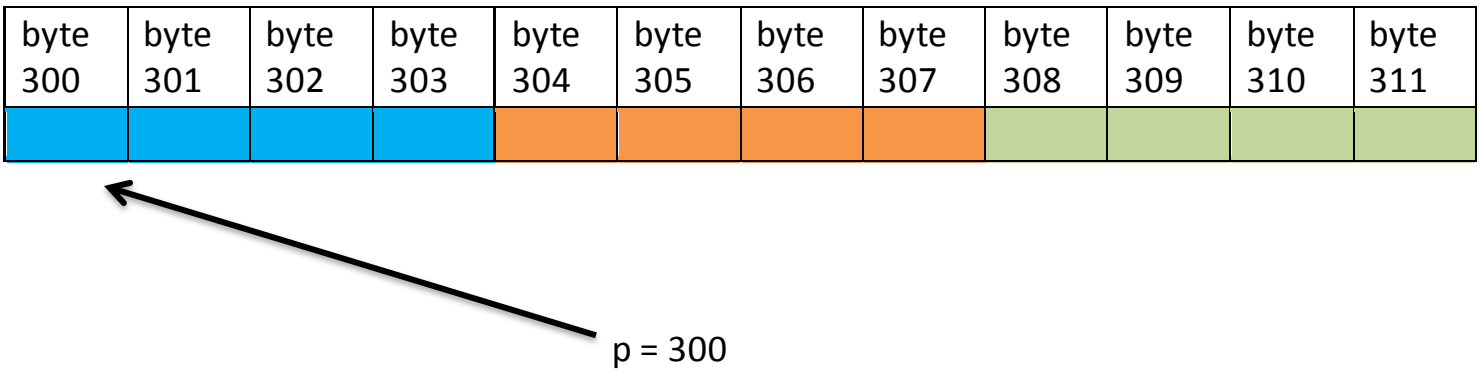
Kiểu int có độ lớn 4 bytes, do đó mảng a sẽ có độ lớn 12 bytes. Hình ảnh full HD không che sắc nét nhất về mảng a của chúng ta sẽ như sau:



Bạn hãy chú ý rằng: a[0] có địa chỉ là 300, a[1] có địa chỉ là 304 và a[2] có địa chỉ là 308.

Điều gì sẽ xảy ra nếu như ta khai báo con trỏ `int *p`, sau đó ta cho p lưu địa chỉ của a[0] ?

```
p = &a[0]; // p = 300 và p trỏ đến a[0]
```



Và điều gì sẽ xảy ra nếu ta có lệnh `printf("%d", *p)` (hoặc `cout << *p`)

Chắc chắn là ta sẽ in giá trị a[0] ra màn hình.

Điều gì sẽ xảy ra nếu cho p = p + 1 ??? Lúc này p = 304.

byte 300	byte 301	byte 302	byte 303	byte 304	byte 305	byte 306	byte 307	byte 308	byte 309	byte 310	byte 311

p = 304

Vậy nếu ta cho `printf("%d", *p)` (hoặc `cout << *p`) thì sao ?

Ta in giá trị a[1] ra màn hình.

Lại tiếp tục p = p + 1.

byte 300	byte 301	byte 302	byte 303	byte 304	byte 305	byte 306	byte 307	byte 308	byte 309	byte 310	byte 311

p = 308

Và thêm 1 lần nữa, ta thử printf (hoặc cout) → ta sẽ in giá trị a[2] ra màn hình.

NHẬN XÉT QUAN TRỌNG:

Khi cho p = p + 1 dần dần, ta nhận thấy p trỏ đến từng phần tử trong mảng a.

Wowwwwww. 

Từ những quan sát trên, ta thử viết một chút code nhé:

C

```c
#include <stdio.h>

int main()
{
	int a[3] = { 9, 8, 7 };

	int *p = &a[0]; // p đang trỏ đến a[0]

	int i;
	for (i = 0; i < 3; i++)
	{
		printf("%5d", *p);

		// p trỏ đến phần tử tiếp theo
		p++;
	}

	return 0;
}
```

C++

```cpp
#include <iostream>

int main()
{
	int a[3] = { 9, 8, 7 };

	int *p = &a[0]; // p đang trỏ đến a[0]


	for (int i = 0; i < 3; i++)
	{
		std::cout << (*p) << "   ";

		// p trỏ đến phần tử tiếp theo
		p++;
	}

	return 0;
}
```

Chạy thử chương trình

```
    9    8    7
```

Vậy, mảng là gì ?

Mảng là dãy các phần tử có cùng kiểu dữ liệu và **nằm *liên tục trên bộ nhớ***.

Các phần tử trong mảng được lưu bởi các ô nhớ liên tục của RAM, vì vậy mà con trỏ có thể duyệt và in ra giá trị.

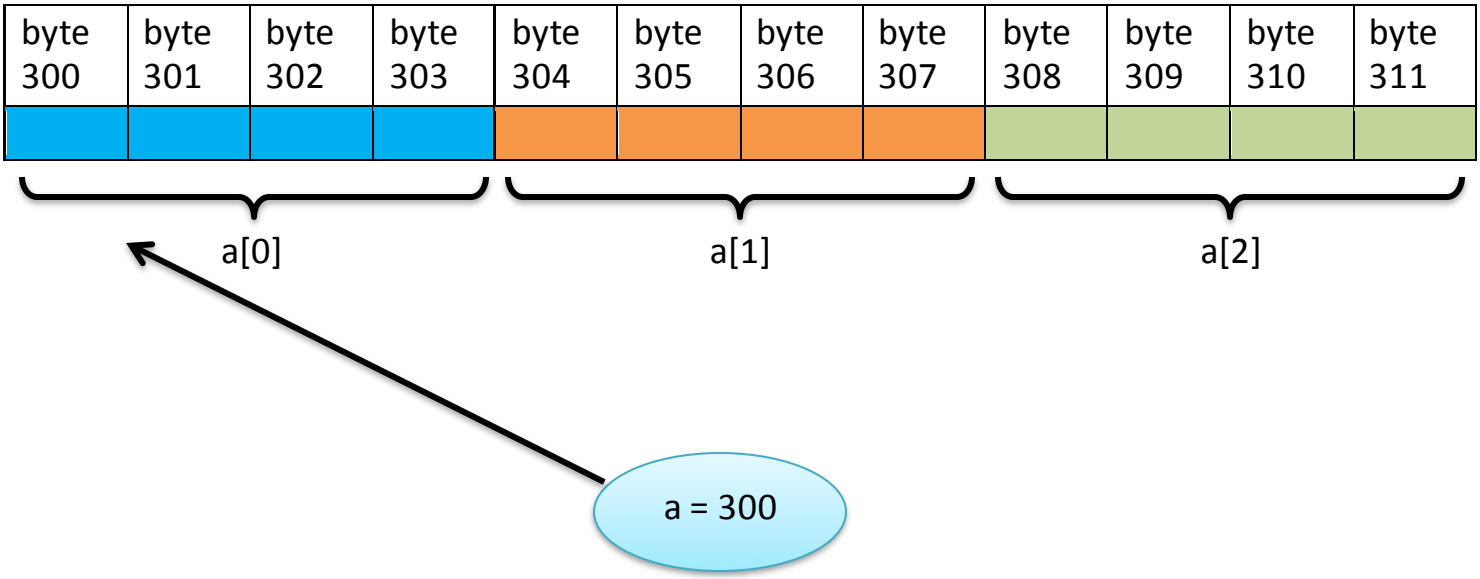
Bạn đã bắt đầu thấy có sự liên quan giữa mảng và con trỏ chưa ? Sang phần tiếp theo ở dưới bạn sẽ thấy nó còn ghê và ảo diệu hơn nữa.

2. Bất ngờ nho nhỏ…

Xét mảng số nguyên a có tối đa 3 phần tử

```
int a[3];
```

Hình ảnh thực tế là như sau:



a được gọi là "mảng", và đồng thời, a cũng là "con trỏ".

Cụ thể hơn: a là con trỏ và a luôn trỏ đến phần tử đầu tiên trong mảng (a[0]).

Từ đó ta có thể suy ra…

- a = 300
- a – 1 = 296
- a + 1 = 304
- a + 2 = 308

Hoặc thậm chí:

```
int *p = NULL;

p = a;      // p = a = 300
p = a + 1;  // p = 304
p = a + 2;  // p = 308
```

Bạn chỉ cần lưu ý 1 điều: a là hằng con trỏ nên ta không thể thay đổi giá trị của a.

```
a = a + 1;  // sai
a++;        // sai
p = a;      // ok
p = a + 1;  // ok
```

Vì a là con trỏ nên bạn có thể thay đổi code cho linh động, ví dụ hàm XuatMang:

```
void XuatMang(int a[], int n) // cách 1

void XuatMang(int *a, int n)  // cách 2
```

Bạn hãy thử động não xem nhé…

C

```
#include <stdio.h>


int main()
{
	int a[3] = { 9, 8, 7 };

	printf("%d \n", *a);		// 9
	printf("%d \n", *(a + 1));	// 8
	printf("%d \n", *(a + 2));	// 7

	*(a + 1) = 4;
	printf("%d \n", a[1]);		// 4

	return 0;
}
```

C++

```
#include <iostream>
using namespace std;

int main()
{
	int a[3] = { 9, 8, 7 };

	cout << *a << endl;		// 9
	cout << *(a + 1) << endl;	// 8
	cout << *(a + 2) << endl;	// 7

	*(a + 1) = 4;
	cout << a[1] << endl;		// 4

	return 0;
}
```

Áp dụng tất cả những gì đã học, bạn có thể sử dụng hoàn toàn con trỏ để nhập mảng và xuất mảng như sau:

C

```c
#include <stdio.h>


int main()
{
	int a[20];
	int n = 0;

	int *p = NULL;

	printf("Nhap vao so luong phan tu: ");
	scanf("%d", &n);

	for (p = a; p < a + n; p++)
	{
		printf("a[%d] = ", p - a);
		scanf("%d", p);
	}

	printf("\n");
	// IN RA MÀN HÌNH

	for (p = a; p < a + n; p++)
		printf("%5d", *p);

	return 0;
}
```

C++

```cpp
#include <iostream>
using namespace std;

int main()
{
	int a[20];
	int n = 0;



	cout << "Nhap vao so luong phan tu: ";
	cin >> n;

	for (int *p = a; p < a + n; p++)
	{
		cout << "a[" << (p - a) << "] = ";
		cin >> *p;
	}

	cout << endl;
	// IN RA MÀN HÌNH

	for (int *p = a; p < a + n; p++)
		cout << *p << "  ";

	return 0;
}
```

Chạy thử chương trình:

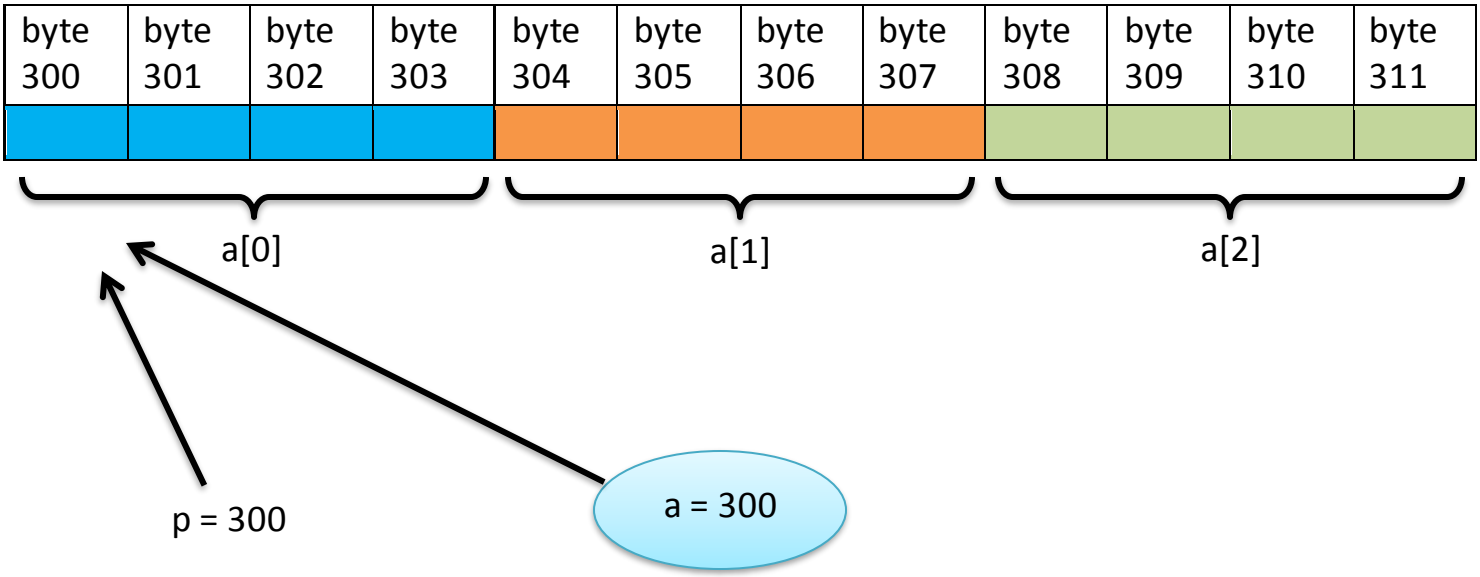
```
Nhap vao so luong phan tu: 4
a[0] = 9
a[1] = 8
a[2] = 7
a[3] = 6

    9    8    7    6
```

3. Con trỏ truy xuất như mảng

Quay về với hình vẽ cũ:

```
int a[3];    // a = 300
int *p = a;  // hoặc int *p = &a[0], ta có p = 300
```



Bạn nhận thấy a cũng có thể xem là con trỏ (a trỏ đến phần tử đầu tiên trong mảng).

Vậy bây giờ ta thử động não xem...

```c
#include <stdio.h>

int main()
{
	int a[] = { 9, 8, 7 };
	int *p = a;

	// in ra *p

	return 0;
}
```

```
9
```

Chắc chắn là in ra số 9, NHƯNG BÂY GIỜ TA THỬ SỬA CODE LẠI xem…

```
#include <stdio.h>

int main()
{
	int a[] = { 9, 8, 7 };
	int *p = a;

	// in ra p[0]

	return 0;
}
```

```
9
```

Vẫn in ra số 9 !!!

Chuyện gì đã xảy ra ? Con trỏ truy xuất như mảng.

Vậy điều gì xảy ra nếu…. ta in ra `p[1]` ??? Ta sẽ in ra số 8.

Và tương tự, nếu ta in ra `p[2]`, ta sẽ in ra số 7.

→ Đây là 1 tính chất cực kì mạnh mẽ của con trỏ, nó cho phép ta truy xuất như mảng và hoạt động như mảng vậy.

Nó mạnh đến mức nào ? Bạn hãy qua phần sau để hiểu thêm nhé.

C. Mở rộng kiến thức

1. Sự ảo diệu của con trỏ trong ngôn ngữ C/C++

Trong hầu hết các ngôn ngữ lập trình thì mảng bắt đầu từ vị trí 0.

Có bao giờ bạn truy xuất a[-1] hay chưa ?

```
int a[3];
a[-1] = 7;
```

Rõ ràng truy xuất chỉ số âm của mảng là 1 điều bất hợp lý, chắc chắn ta sẽ vi phạm bộ nhớ (truy xuất trái phép). Trong nhiều trường hợp ta cần truy xuất chỉ số âm của mảng, điều này quả thật gây khó khăn cho ta.

NHƯNG với sự ảo diệu của con trỏ trong C/C++, bạn sẽ làm được.

Giả sử khai báo mảng a có tối đa 7 phần tử, khai báo thêm con trỏ p.
Ta cho `p = a + 3` hoặc `p = &a[3]`

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
12	2	0	8	9	7	4

Khi ta in ra *p HOẶC p[0] thì bạn đã in số 8 ra màn hình.

➔ `*p = p[0] = a[3] = 8`.

Vậy, nếu ta in ra `*(p - 1)` HOẶC `p[-1]` thì sao ?

➔ Ta sẽ in số 0 ra màn hình.
➔ `*(p - 1) = p[-1] = a[2]`.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
12	2	0	8	9	7	4

Vậy, nếu ta in ra `*(p - 2)` HOẶC `p[-2]` thì tương đương bạn in ra `a[1]`.

Và cuối cùng, nếu in ra `p[-3]` thì tương đương bạn in ra `a[0]`.

Bạn thấy gì chưa ?

Như vậy, áp dụng con trỏ ta có thể truy xuất vị trí âm. Ta có thể xem con trỏ là 1 mảng.

Giả sử chúng ta cần truy xuất từ a[-10] đến a[10], như vậy ta có thể áp dụng con trỏ như sau:

```
int A[21];
int *a = A + 10; // hoặc a = &A[10]

a[-10] = 4; // tương đương A[0] = 4
a[10] = 5;  // tương đương A[20] = 5
```

Đây chính là sự ảo diệu của con trỏ trong C/C++.

2. Mẹo: truy xuất phần tử

Khi ta ghi a[2] thì C/C++ hiểu là *(a + 2).

Vậy nếu ta ghi 2[a] thì sao ? Cũng hiểu là *(2 + a) → tương đương a[2].

```
int main()
{
	int a[] = { 9, 8, 7, 6 };

	printf("%d \n", 2[a]); // cout << 2[a] << endl;

	// in ra số 7

	return 0;
}
```

3. Vì sao mảng bắt đầu từ vị trí 0

Trong hầu hết các ngôn ngữ lập trình thì mảng bắt đầu từ vị trí 0.

Vì sao lại như vậy ?

Vì vị trí 0 là vị trí bắt đầu giá trị trong bộ nhớ RAM.

Nếu mảng bắt đầu từ 0	**Nếu mảng bắt đầu từ 1**
Khi truy xuất phần tử đầu tiên thì ta có thể ghi là a[0] hoặc *a Không bỏ sót phần tử a[0]	Khi truy xuất phần tử đầu tiên thì ta có thể ghi là a[1] hoặc *(a + 1) Bỏ sót phần tử a[0] trong bộ nhớ Nếu là mảng số nguyên int, bạn đã phung phí 1 số nguyên a[0] có độ lớn 4 bytes. Nếu là mảng các HocSinh, các TruongHoc, bạn nghĩ bạn đã phung phí bao nhiêu tài nguyên ? Có khi là hàng trăm bytes.

Nhìn chung, khi mảng bắt đầu từ 1 ta cảm thấy tự nhiên hơn nhưng ta đã lãng phí bộ nhớ và có thể làm chậm tốc độ tính toán 1 chút xíu.

D. Bài tập

Bài 1. Nhập vào mảng 1 chiều, tính tổng các phần tử của mảng. Thao tác xuất ra mảng phải thực hiện bằng con trỏ. Không cần viết thêm hàm phụ.

```
for (p = a; p < a+n; p++)
     in ra màn hình *p;
```

Bài 2. Nhập vào mảng 1 chiều, tính tổng các phần tử của mảng. Thao tác tính tổng các phần tử thực hiện bằng phép tính với con trỏ như ở dưới. Không cần viết thêm hàm phụ.

```
for (i = 0; i < n; i++)
     tong = tong + *(a + i);
```

Bài 3. Nhập vào mảng 1 chiều, in ra mảng ngược. Thao tác in ra mảng ngược được code bằng 3 cách khác nhau, mỗi cách viết trong 1 hàm.

Cách 1:

```
     for (i = n-1; i >= 0; i--)
          In ra màn hình a[i];
```

Cách 2:

```
     for (i = n-1; i >= 0; i--)
          In ra màn hình *(a+i);
```

Cách 3:

```
     for (p = a + n-1; p >= a; p--)
          In ra màn hình *p;
```

Bài 4. Nhập vào mảng 1 chiều, tính tổng các số chẵn của mảng. Yêu cầu: mọi thao tác nhập, xuất, tính tổng đều phải thực hiện bằng con trỏ. Không cần viết thêm hàm phụ.

Kể từ bài 5 trở đi, bạn cần viết các hàm phụ hỗ trợ (hàm nhập mảng, xuất mảng, xử lý,…).

Bài 5. Đảo ngược mảng, sử dụng con trỏ. Làm bằng 2 cách.

Cách 1: làm bình thường, không con trỏ.

```
for (i = 0; i < n/2; i++)
      HoánĐổi a[i] và a[n-1-i]
```

Cách 2: sử dụng 2 con trỏ, một con trỏ chạy bên trái và một con trỏ chạy bên phải.

```
int *pLeft = a, *pRight = a + n-1;
while (pLeft < pRight)
{
      HoánĐổi *pLeft và *pRight;
      pLeft++;
      pRight--;
}


Ghi chú: vòng lặp while ở trên có thể rút ngắn lại như sau
for (; pLeft < pRight; pLeft++, pRight--)
      HoánĐổi *pLeft và *pRight;
```

Bài 6. Viết hàm tìm kiếm một giá trị nào đó trong mảng. Chương trình phải có code minh họa.

Hàm này có dạng như sau

```
int* TimKiem(int *a, int n, int GiaTri)
```

Hàm truyền vào một mảng số nguyên (a), số lượng phần tử (n) và giá trị cần tìm kiếm.

Hàm trả về địa chỉ của phần tử = GiaTri xuất hiện đầu tiên trong mảng. Nếu không tìm được thì trả về NULL.

Ví dụ: cho mảng `a = { 5, 9, 1, 6, 7, 3, 7 }`

Gọi hàm TimKiem (a, 6, 7) sẽ trả về địa chỉ của a[4] → xuất ra vị trí là 4.
Gọi hàm TimKiem (a, 6, 9) sẽ trả về địa chỉ của a[1] → xuất ra vị trí là 1.
Gọi hàm TimKiem (a, 6, 2) sẽ trả về NULL → xuất ra kết quả "không tìm thấy".

Bài 7. Viết hàm tìm kiếm số nguyên tố trong mảng. Chương trình phải có code minh họa.

Hàm này có dạng như sau

```
int* TimKiemSNT(int *a, int n)
```

Hàm truyền vào một mảng số nguyên (a) và số lượng phần tử của mảng (n).

Hàm trả về địa chỉ của số nguyên tố xuất hiện đầu tiên trong mảng. Nếu không tìm thấy thì trả về NULL.

Ví dụ 1: cho mảng `a = { 0, -2, 5, 8, 7 }`

Mảng a này có 2 số nguyên tố là 5 và 7.
Gọi hàm TimKiemSNT (a, 5) sẽ trả về địa chỉ của a[2] vì a[2] = 5.

Ví dụ 2: cho mảng `a = { -2, 10, 8, 0 }`

Gọi hàm TimKiemSNT (a, 4) sẽ trả NULL vì không tìm thấy số nguyên tố nào cả.

Bài 8. Viết chương trình tìm tất cả số nguyên tố trong mảng và gán giá trị = 0. Chương trình này sử dụng lại hàm TimKiemSNT ở bài tập trước đó.

Ví dụ: cho mảng `a = { 5, -2, 3, 11, 4, 6, 7 }`

Sau khi xử lý: `a = { 0, -2, 0, 0, 4, 6, 0 }`

Bài này phải chia mã nguồn hợp lý thành 3 file: main.c, BaiLam.h, BaiLam.c (hoặc main.cpp, BaiLam.h và BaiLam.cpp).

Bài 9. Nhập vào mảng các số nguyên.

Mỗi phần tử nằm trong đoạn [-500, 500], tức là $-500 \le a[i] \le 500$.

Cho biết các phần tử nào xuất hiện nhiều nhất mảng và xuất hiện ít nhất mảng.

Ví dụ: `a = {8, -2, 8, -2, 5, 0, 8}`

Số 8 xuất hiện nhiều nhất (xuất hiện 3 lần).

Số 5 và số 0 xuất hiện ít nhất (xuất hiện 1 lần).

Bài 10. Nhập vào mảng các số nguyên, mảng có tối đa 1000 phần tử.

Mỗi phần tử nằm trong đoạn [-100, 100], tức là -100 ≤ a[i] ≤ 100.

Xuất ra mảng theo chiều tăng dần giá trị.

Ví dụ: `a = {9, 3, 7, 0, 7, -4, 0}`

→ `a = {-4, 0, 0, 3, 7, 7, 9}`

Gợi ý cách giải: tìm hiểu về thuật toán Counting Sort. Không được sử dụng các thuật toán sắp xếp khác.

Không được phép thay đổi mảng gốc, chỉ cần xuất ra màn hình kết quả đúng là ok.

E. Tổng kết

Hi vọng tài liệu này sẽ giúp ích cho bạn. Cảm ơn bạn đã xem.

Tác giả: Nguyễn Trung Thành
Facebook: https://www.facebook.com/abcxyztcit