CON TRỎ

Người thực hiện: Trần Thanh Tịnh – K16 CNTT

Lời dẫn: Con trỏ được xem là 1 trong những phần khó nhất trong lập trình C/C++. Nhưng thông qua tài liệu này hi vọng các bạn sẽ phần nào hình dung ra và sử dụng một cách thành thạo con trỏ.

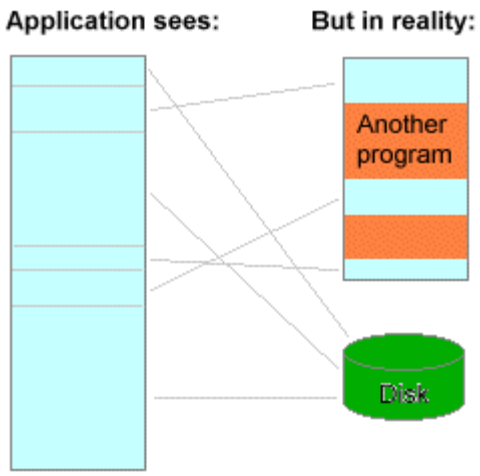
Chúc các bạn học tốt!

PHẦN I: BỘ NHỚ

1. Bộ nhớ ảo là gì?

- Quản lý bộ nhớ vật lý (cấp phát, thu hồi) là 1 vấn đề cực kì phức tạp trong hệ thống máy tính, để bảo đảm sự hiệu quả, đúng đắn, an toàn cho việc quản lý đó, hệ điều hành xây dựng lên các vùng nhớ ảo.

- Trong hệ thống máy tính, bộ nhớ ảo (virtual memory) là một kĩ thuật cho phép một chương trình ứng dụng tưởng rằng mình đang có một dải bộ nhớ liên tục (một không gian địa chỉ), trong khi thực ra phần bộ nhớ này có thể bị phân mảnh trong bộ nhớ vật lý và thậm chí có thể được lưu trữ cả trong đĩa cứng. So với các hệ thống không dùng kĩ thuật bộ nhớ ảo, các hệ thống dùng kĩ thuật này cho phép việc lập trình các ứng dụng lớn được dễ dàng hơn và sử dụng bộ nhớ vật lý thực (ví dụ RAM) hiệu quả hơn.



- Định nghĩa của "bộ nhớ ảo" có nền tảng là việc định nghĩa lại không gian địa chỉ bằng một dải liên tục các địa chỉ bộ nhớ ảo để "đánh lừa" các chương trình rằng chúng đang dùng các khối lớn các địa chỉ liên tục.

2. Địa chỉ ảo là gì ?

- Trong vùng bộ nhớ ảo kia, để cho tiến trình dễ sử dụng, hệ thống quy định rằng chia nhỏ ra theo từng byte, và đánh số từ 1 đến hết các ô nhớ nào đó, đã được đánh số là i thì ta nói địa chỉ của ô nhớ đó là i.
Giả sử có biến a khai báo như sau:
int a;
và a nằm trong ô thứ 452321 tại vùng nhớ trên, vậy a có địa chỉ là 452321.

Tiến trình hiểu là thế, còn hệ điều hành thì hiểu hơn 1 tí (địa chỉ này tương ứng với ô nhớ nào trong thanh ram mà ta đang quản lý).

Lưu ý:

+ Mỗi tiến trình có 1 vùng nhớ ảo riêng.
+ Vùng nhớ ảo là 1 không gian địa chỉ ảo trải dài từ thấp đến cao (từ 0x0000 cao hơn)
+ Ở trong windows 32bit thì không gian địa chỉ ảo có địa chỉ từ 00000000h trải dài đến 7fffffffh
+ Bạn cần hiểu nó chỉ là ảo, không phải vùng nào cũng có bộ nhớ vật lý thật.

3. Đặc điểm của vùng nhớ:

Tổng quan:

- Có 4 vùng nhớ cần được quan tâm:

- Code segment
- Data segment
- Heap segment
- Stack segment

3.1. Code segment:

- Khi một chương trình được biên dịch, nó sẽ được chuyển thành các lệnh mã máy chỉ bao gồm 0 và 1. Đây là vùng nhớ chứa mã máy để thực thi chương trình của bạn. Tuy nhiên, đây là vùng nhớ mang tính chất read only nhằm ngăn chặn việc vô tình thay đổi những chỉ thị trong vùng nhớ này khi chương trình thực thi.

3.2. Data segment:

- Đây là nơi lưu trữ biến tĩnh và biến có phạm vi global. Được chia làm 2 loại nhỏ:

- Initialized Data segment (đã khởi tạo).
- Unittialized Data segment (chưa khởi tạo).

3.3. Stack segment:

- Stack có thể được biết đến như một cấu trúc First In – Last Out (FILO). Trong bộ nhớ của máy tính, Stack là một phân khúc lưu trữ những biến tạm thời được tạo ở trong phương thức (kể cả hàm main ()). Phân khúc này được CPU quản lý và tối ưu khá chặt chẽ. Khi một biến được khai báo trong hàm, nó sẽ được push vào trong Stack, và khi hàm đó kết thúc, toàn bộ những biến đã được đẩy vào trong stack sẽ được pop và giải phóng. Vì đặc tính này cho nên biến lưu trên Stack mang tính chất Local.
- Khi bạn sử dụng biến lưu trên Stack, bạn sẽ không phải quan tâm đến việc cấp phát và thu hồi biến đó. Ngoài ra, truy cập ở phân khúc Stack sẽ nhanh hơn một chút so với Heap nhưng bù lại phân khúc này lại có kích thước nhỏ hơn Heap khá nhiều lần.

Lưu ý: Trong trường hợp bạn chưa học cấu truc Stack thì bạn có thể hiểu chung rằng Stack chứa các biến mang tính local, như các biến được khai báo trong hàm, trong if, while…

3.4. Heap segment:

- Heap là phân khúc bộ nhớ cho lập trình viên có thể sử dụng. Lập trình viên phải tự quản lý toàn bộ những vùng nhớ mà bạn cấp phát trên Heap, nếu không kiểm soát chặt chẽ sẽ gây ra rò rỉ bộ nhớ (memory leak). Và phải tự thu hồi bằng các hàm free hoặc delete mà ta sẽ học trong những chương sau.

PHẦN II: TỔNG QUAN

1. Khái niệm về con trỏ:

1.1. Con trỏ chỉ là 1 biến bình thường:

- Con trỏ cũng chỉ là một biến bình thường có chứa thêm địa chỉ ảo đã nói ở trên.
Ví dụ như là: 0x6E6E1 hoặc 0x4B6088 hoặc 454321
Khi ta khai báo:
void *p;
char *p;
hay là
double *p;
long long *p;
thì p vẫn là một biến nguyên.

1.2.Trong hệ điều hành 32bit thì nó có độ dài là 32 bit:

Trong windows 32bit (xp, vista, 7) thì địa chỉ ảo có độ dài là 32 bit, tương ứng với số hexa có 8 chữ số. Vì vậy con trỏ bất cứ dữ liệu nào cũng mang độ lớn 32bit = 4byte.

Vì sao lại chỉ có 32bit?
Vì nó cần 32bit là vừa đủ để chỉ trỏ hết vùng nhớ ảo đó. (đối với win 32)

1.3. Con trỏ dùng để làm gì?

- Đơn giản, đúng như cái bản chất của nó thì nó để chỉ trỏ lung tung trong vùng nhớ ảo của tiến trình hiện tại.
- Đặc biệt: con trỏ chỉ là 1 công cụ, là 1 kiểu dữ liệu, để ta cài đặt các giải thuật, chứ không phải là 1 giải thuật hay thuật toán, nên câu nói như là "dùng con trỏ để giải bài A", "giải bài tập B bằng con trỏ" là hoàn toàn sai.
Nói đúng phải là "giải bài tập C sử dụng con trỏ".

PHẦN III:KHAI BÁO

Cấu trúc khai báo:

<Kiểu dữ liệu>*******<tên con trỏ>;

Kiểu dữ liệu ở đây có thể là:
+ Kiểu dữ liệu có sẵn (built-in data type) : int , char , void , double , long , ...
+ Kiểu dữ liệu cấu trúc do người dùng định nghĩa (user-defined data type) : struct , union
+ Kiểu dữ liệu là lớp do người dùng định nghĩa (C++)
+ Kiểu dữ liệu dẫn xuất
+ Kiểu con trỏ hàm

Nhắc lại lần nữa, kiểu dữ liệu này là kiểu dữ liệu của vùng nhớ mà nó trỏ đến.

Ví dụ:

int *a;

ra khỏi câu khai báo này ta sẽ nói : a là con trỏ

Chú ý 1 :

int *a,b; // thì a là con trỏ, b là biến nguyên

Chú ý 2:

int* a, b; //thì a là con trỏ, b là biến nguyên

//và cách viết như này cực kì đáng ghét vì gây ra toàn hiểu lần đáng ghét

Chú ý 3::

void *a; //đúng, hoàn toàn có con trỏ void

PHẦN IV : KHỞI TẠO

1. Khởi tạo là gì?

Có 1 số bạn sẽ lạ lẫm vì cái tiêu đề khai báo với khởi tạo nghe có vẻ giống nhau…
Nhưng khai báo (declared, register) và khởi tạo(initialize) hoàn toàn khác nhau.
int a; // khai báo biến a
int b=2; //khai báo biến b và kết hợp với khởi tạo giá trị cho biến b bằng 2

Khi ta khai báo 1 biến thì câu lệnh đầu tiên thiết lập giá trị cho biến đó thì đó là khởi tạo. Trong C03, C++03 trở lên khi ta khai báo 1 biến local, chưa khởi tạo giá trị mà đã đem sử dụng thì sẽ phát sinh lỗi runtime .

2. Khởi tạo giá trị cho biến con trỏ:

Cấu trúc khởi tạo:
TênConTrỏ = ĐịaChỉ;
+ Trong đó tên con trỏ là tên của biến con trỏ
+ Địa chỉ là vùng địa chỉ mà ta muốn trỏ đến

Chú ý 1: Bản thân p cũng là 1 biến (nguyên), p cũng nằm trong bộ nhớ, cũng có địa chỉ

riêng.

Chú ý 2: Toán tử & ở đây chính xác phải gọi là unary operator &, toán tử & 1 ngôi, nó hoàn toàn khác với toán tử & 2 ngôi (bitwise). Toán tử & 1 ngôi này dùng để lấy địa chỉ của 1 biến. Trước khi động đến lý thuyết về con trỏ, chúng ta đã từng sử dụng toán tử này rồi đó.

Ví dụ: scanf ("%d", &a);
Chú ý 3: Có thể viết ví dụ trên ngắn gọn lại thành
int a=1987, p=&a;

3. Có được điều gì sau khi khởi tạo như ví dụ trên?

+ Khi giá trị nằm trong p là địa chỉ của a thì ta nói p trỏ vào a
+ Lúc này thì *p hoàn toàn tương đương với a , người ta coi *p là bí danh của a , thao tác với *p cũng như thao tác với a, thao tác với a cũng như thao tác với *p.

Ví dụ :

a. câu lệnh a=2; hoàn toàn tương đương với câu lệnh *p=2;
b. câu lệnh a++; hoàn toàn tương đương với (*p)++
// chú ý khác với *p++ nhé, phải cho *p vào trong đóng mở ngoặc vì toán tử * có độ

ưu tiên thấp hơn ++
c. câu lệnh b=a+c-9; hoàn toàn tương đương với câu lệnh b=(*p)+c-9;
d. câu lệnh (*p)=(*p) -1227; hoàn toàn tương đương với a=a-1227;
+ Lúc này câu lệnh scanf("%d",&a); ta có thể thay bằng scanf("%d",p);

Chú ý : Toán tử *
Toán tử * ở đây là toán tử 1 ngôi , tác dụng là truy xuất đến ô nhớ mà con trỏ đang trỏ đến
Để tránh những hiểu lầm không đáng có, khi có sự nhập nhằng mà bạn không thể đoán được, bạn hãy thêm cặp () nha
(*p)++
a+(*p)*c // thêm vào cho nó sáng sủa code ra

4. Kiểu dữ liệu con trỏ và các phép toán trên con trỏ

4.1. Kiểu dữ liệu con trỏ

- Khi ta viết int *p, b; chúng ta luôn viết * gần p, vì sao? vì * này là của p, p là con trỏ, b ko phải con trỏ.
- Kiểu dữ liệu của b là int
- Kiểu dữ liệu của p là gì ??????????????

➢ Chúng ta có thể nhận thấy điều này, kiểu dữ liệu của p là (int *)

Gợi ý:

+ Trong câu lệnh khai báo con trỏ nên viết * gần tên con trỏ

+ Khi viết kiểu dữ liệu nên viết * đứng gần kiểu dữ liệu cơ bản : cụ thể ở kiểu dữ liệu trả về của hàm, ở tiêu đề và nguyên mẫu hàm

+ Ở câu lệnh ép kiểu thì tùy ý theo bạn muốn, có thể viết cách ra cho thoáng code

4.2. Các phép toán trên con trỏ

a. Phép gán:

Phép gán đối với con trỏ thì tham khảo phần khởi tạo nhưng có 1 vài yếu tố sau đây :

+ Tất cả các loại con trỏ đều có phép gán

+ Phép gán với con trỏ yêu cầu vế trái là 1 con trỏ và vế phải là 1 địa chỉ

+ Phép gán yêu cầu sự tương xứng về kiểu dữ liệu, nếu ko tương xứng chúng ta phải ép kiểu

ví dụ p=(int*)8232;

p có kiểu dữ liệu là int*

còn 8232 là 1 hằng số nguyên, nên phải ép kiểu về int* rồi thực hiện phép gán

+ Phép gán với 1 con trỏ kiểu void ko cần thiết phải tương xứng hoàn toàn về kiểu dữ liệu, void* có thể tương ứng với tất cả (như ở ví dụ chap trước), thậm chí là vượt cấp (vượt hẳn 2 cấp) như ví dụ sau

```
void *p,**q;
p=&q;
```

b. Phép so sánh

Phép so sánh ngang bằng dùng để kiểm tra 2 con trỏ có trỏ vào cùng 1 vùng nhớ hay không, hoặc kiểm tra 1 con trỏ có phải là đang trỏ vào NULL hay không (trong trường hợp cấp phát động, mở file, mở resource,…)

Phép so sánh lớn hơn nhỏ hơn : > , < , >= , <= sử dụng để kiểm tra về độ thấp cao giữa 2 địa chỉ. Con trỏ nào nhỏ hơn thì trỏ vào địa chỉ thấp hơn.

+ Được quyền so sánh mọi con trỏ với 0, vì 0 chính là NULL

+ Ngoài ra thì khi so sánh 2 con trỏ hoặc con trỏ với 1 địa chỉ xác định (số nguyên) cần có sự tương xứng về kiểu dữ liệu

Ví dụ:

```
int main()
```

```
{
int a=197, *p = &a;
double b = 0, *x = &b;

// so sánh 2 con trỏ
(int)p == (int)x;
p==(int *)x;
(double*)p==x;
(void*)p==(void*)x;
p==(void*)x;
(float*)p==(float*)x;

//so sánh con trỏ với số nguyên
p==(int*)9999;
int(p)==9999;
```

Chú ý: Con trỏ void có thể đem ra so sánh với tất cả các con trỏ khác

c. Phép cộng trừ và phép tăng giảm:

Bản chất của việc tăng/ giảm con trỏ p đi 1 đơn vị là cho p trỏ đến ô nhớ bên cạnh phía

dưới/trên.

Chú ý:

+ Khi tăng giảm con trỏ p đo 1 đơn vị không có nghĩa là trỏ sang byte bên cạnh
+ Việc tăng giảm con trỏ đi 1 đơn vị phụ thuộc vào kiểu dữ liệu và nó trỏ đến, quy tắc là

p + 1 nghĩa là giá trị chứa trong p + sizeof(kiểu dữ liệu của biến mà p trỏ đến)

+ Không có phép tăng giảm trên con trỏ void
+ Không có phép tăng giảm trên con trỏ hàm
+ Không có phép cộng 2 con trỏ với nhau
+ Phép trừ 2 con trỏ trả về độ lệch pha giữa 2 con trỏ

Vậy ta có kết luận như sau : kiểu dữ liệu trỏ đến có tác dụng xác thực sự rõ ràng tất cả các phép toán trên con trỏ (bao gồm cả phép = * &)

5. Con trỏ với mảng, xâu, cấp phát bộ nhớ động

5.1. Hằng con trỏ và con trỏ hằng

a. Hằng là gì?

Ta đã biết hằng số (toán học) là những đại lượng có giá trị không đổi, trong lập trình là

những đại lương có giá trị không đổi trong suốt chương trình.
Hằng trong C/C++ có định kiểu rõ ràng
Hằng trong C/C++ được định nghĩa bằng từ khóa const

Chú ý : Có 1 số người hiểu lầm rằng dùng từ khóa define định nghĩa hằng số, đây thật sự là 1 cách hiểu sai lầm hoàn toàn. Define định nghĩa nên macro và có rất nhiều sự khác nhau khi ta dùng define và const.

b. Hằng con trỏ?

Bây giờ ta sẽ tìm hiểu thêm về kiểu dữ liệu con trỏ ở hằng con trỏ. Vậy hằng con trỏ là gì ? Đối với hằng và con trỏ có 2 loại như sau
+ Những con trỏ mà chỉ trỏ cố định vào 1 vùng nhớ , những con trỏ này không có khả năng trỏ vào vùng nhớ khác, không thay đổi được (1)
+ Những con trỏ mà trỏ vào 1 vùng nhớ cố định, con trỏ này chỉ có tác dụng trỏ đến, chứ không có khả năng thay đổi giá trị của vùng nhớ này, con trỏ này được ứng dụng gần như là tác dụng của phương thức hằng trong OOP (2)

Để tiên phân biệt, ta gọi (1) là hằng con trỏ và (2) là con trỏ hằng, và chúng ta có thể gộp cả 2 kiểu này để thành 1 kiểu mới

Ví dụ về loại (1)

```
void main()
{
char buf[] = "6969";
char * const p = buf;

p++; // báo lỗi tại đây
p[4]++; //không vấn đề, hoàn toàn có thể thay đổi giá trị vùng nhớ mà p trỏ đến
}

void main()
{
char *p="con tro";
p++;

(*p)++; // báo lỗi tại đây (không báo lỗi khi biên dịch nhưng có lỗi trong run-time)
p[2]='b'; // báo lỗi tại đây (không báo lỗi khi biên dịch nhưng có lỗi trong run-time)
```

```
}
```

Ví dụ tiếp về loại (2)

```
char buf[] = "6969";
char const * p = buf; /* hay const char * p = buf; */

p++; /* được */
p[4]++; /* không thể thực hiện được */
```

Ví dụ về kết hợp

```
char buf[] = "6969";
char const * const p = buf;

p++; /* Sai */
p[4]++; /* Sai */
```

Ví dụ tiếp với hàm

```
void ConvertUnicodeTextToSomeThing(const unsigned short int *wstr)
{
unsigned short int const * p=wstr; //đúng

unsigned short int * q=wstr; //báo lỗi

}
```

c. Con trỏ hằng?

- Khái niệm: Con trỏ hằng là con trỏ trỏ đến vùng dữ liệu hằng. Ta không thể thay đổi giá trị mà nó đang trỏ đến. Nhưng có thể thực hiện tăng giảm địa chỉ con trỏ hay cho nó trỏ đến nơi khác.

```
int a=3;
const int *p;
p=&a; // bản thân p thì có thể thay đổi, cho p gán vào chỗ khác được nhưng

(*p)++;  // có lỗi tại đây
```

Ví dụ điển hình nhất ở đây là hàm strlen của chúng ta:

int strlen(const char *Str)
Khi bạn code trong 1 project C lớn 1 tí hoặc lớn nhiều, giả sử bạn có 1 hàm, thao tác với 1 mảng , hàm này chỉ đọc mảng thôi, không làm thay đổi các giá trị trong mảng . Và quan trọng là, khi share code cho các bạn khác trong cùng project, làm sao để họ biết điều này ?
Vậy ta sẽ cài đặt hàm của mình như sau
- Đối với trường hợp hằng con trỏ là tham số hình thức thì
void ham(const int *) và void ham(int const *) là như nhau, từ const khi đóng góp vào trong tham số hình thức là như nhau.

```
void ham(const int *a,int n)
{
//xử lý gì đó
}
void main()
{
int a[100]={1,2},n=2;
```

ham(a,n); // khi sử dụng hàm này ta hiểu nó ko thay đổi mảng a //yên tâm xài, nếu có lỗi gì đó thì ko phải sinh ra từ đây
}

5.2. Mảng liên quan gì đến con trỏ?

Khi ta khai báo mảng thì tương đương với: xin cấp phát 1 vùng nhớ có kích thước như bạn khai báo và khai báo ra 1 hằng con trỏ trỏ vào đầu vùng nhớ đó.

int a[100];
+ a là 1 hằng con trỏ trỏ vào phần tử thứ 0 của mảng
+ Các phép toán nhằm làm a trỏ tới vùng khác (thay đổi giá trị của a) là ko thể (++ -- =)
+ a tương đương với &a[0]
+ a+i tương đương với &a[i]
+ *a tương đương với a[0]
+ *(a+i) tương đương với a[i]

Chú ý : trình biên dịch luôn hiểu a[i] là *(a+i)

Biết điều này để làm gì ?

Nhập mảng

```
#include <stdio.h>
#include <conio.h>

void main()
{
float a[100];
int n;
//nhập n
printf("Nhap n :");
scanf("%d",&n);
// nhập mảng
for(int i=0;i<n;i++)
{
printf("Nhap vao phan tu thu %d",i+1);
scanf("%f",a+i);
}
```

Xuất mảng

```
printf("mang vua nhap : \n");
for(int i=0;i<n;i++)
printf("%f ",*(a+i));

getch();
}
```

VD 2:

```
#include <stdio.h>
#include <conio.h>

void main()
{
int a[100]={0,1,2,3,4,5,6};
printf("%d",2[a]); //in ra 2, tại sao vậy ?

getch();
}
```

Chắc chắn lúc nhìn thấy 2[a] không ít người sẽ thấy là lạ, nghĩ nó là lỗi.

Có người thì nghĩ là nó in ra 2, nhưng tại sao vậy?
Thật ra: 2[a] trình biên dịch sẽ hiểu là *(2+a)
*(2+a) hoàn toàn tương đương với *(a+2)
mà *(a+2) chính là a[2]
vậy 2[a] cũng đơn giản là a[2]

5.3. Xâu ký tự:

+ Xâu kí tự là trường hợp đặc biệt của mảng 1 chiều khi mà cách thành phần của mảng là 1byte
+ Xâu kí tự kết thúc bằng NULL. NULL là 1 kí tự đặc biệt có mã là 0,
Có 3 cách viết NULL trong C như sau: NULL , '\0' , 0

❖ Sai lầm thường gặp khi làm việc với xâu kí tự

Đối với xâu kí tự thì các bạn phải nhớ được những trường hợp sau

a. Chưa cấp phát bộ nhớ

```
char *xau;
gets(xau); // vẫn biên dịch được
//nhưng khi chạy sẽ sinh ra lỗi run-time
```

// ở 1 số trình biên dịch cũ thì có thể ko bị lỗi
// nhưng sai thì vẫn là sai, code này sai thuộc loại chưa cấp phát

b. Thay đổi giá trị của một hằng

char *xau = "hoc con tro";
xau[6]='A';// vẫn biên dịch được
//nhưng khi chạy sẽ sinh ra lỗi run-time
// lỗi này là lỗi cố tình thay đổi giá trị của 1 hằng
Nguyên nhân sâu xa của vấn đề như sau :
khi khai báo char *xau="hoc con tro"; thì bản chất là
+ Trong vùng nhớ data của chương trình sẽ có 1 hằng chuỗi “hoc con tro” . <<là hằng chuỗi, đã là hằng thì ko thể bị thay đổi.
+ Cho con trỏ xâu trỏ đến đầu của vùng nhớ đó.
Câu lệnh tiếp theo xau[6]='A'; cố tình thay đổi giá trị của hằng , chắc chắn sẽ sinh ra lỗi.

c. Cố tình thay đổi giá trị của hằng con trỏ

char xau[100];

xau="tinh cute"; // không biên dịch được
// vì phép toán trên có nghĩa là khai báo 1 chuỗi "tinh cute" trong vùng nhớ code
// rồi sau đó cho hằng con trỏ xâu trỏ vào đó
// rất tiếc xâu là hằng con trỏ nên ko thể trỏ đi đâu khác được
// ngoài vị trí đã được khởi tạo trong câu lệnh khai báo
chú ý char xau[100]="tinh cute"; hoặc char xau[100]={0}; thì hoàn toàn hợp lệ

d. Dùng phép toán so sánh để so sánh nội dung 2 xâu

```
void main()
{
char xau[100]="hello phong";
if (xau=="hello phong")
```

Code này ko sai về ngữ pháp, ko sinh ra lỗi runtime, nhưng mang lại kết quả ko như người dùng mong muốn vì theo mục b. ở trên ta có. Phép so sánh ngang bằng dùng để kiểm tra 2 con trỏ có trỏ vào cùng 1 vùng nhớ hay không hoặc kiểm tra 1 con trỏ có phải là đang trỏ vào NULL hay không (trong trường hợp cấp phát động, mở file, mở resource,.....) chứ ko phải là phép so sánh nội dung của xâu. Để so sáng nội dung của xâu ta phải dùng những hàm strcmp (string compare), stricmp hoặc những hàm bạn tự

định nghĩa
}

5.4. Cấp phát động

a. Bản chất của việc cấp phát động.

- Vì bộ nhớ hệ thống là giới hạn nên để hạn chế việc sử dụng bộ nhớ phung phí ta có cơ chế cấp phát động. Chỉ khi nào cần tới vùng nhớ để sử dụng thì chương trình mới yêu cầu cấp phát nhằm tối ưu bộ nhớ.
- Chú ý: khi cấp phát động thì địa chỉ này nằm ở vùng nhớ heap.

b. Cấp phát động như thế nào?

- **Cấp phát**

+ malloc trả về 1 địa chỉ đến 1 vùng nhớ và coi vùng nhớ này là void *, nên trong câu lệnh malloc luôn đi kèm với việc ép kiểu.

Ví dụ: int *a = (int *) malloc(sizeof(int));

+ Một nguyên tắc mà mọi lập trình viên đều phải biết đó là việc cấp phát luôn phải đi đôi với giải phóng. Nếu không sẽ xảy ra hiện tượng rò rỉ bộ nhớ (memory leak).

- C++

Trong C++ chúng ta dùng new và delete để cấp phát động.

new và delete về cú pháp tham khảo trong sách hoặc tìm kiếm trên google.

Câu hỏi: sự khác nhau giữa malloc và new?

Trả lời:

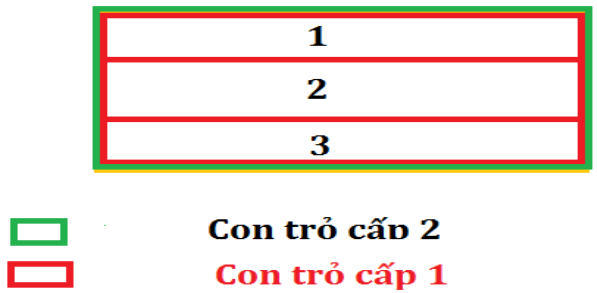
	malloc	new
Ngôn ngữ	C	C++
kích thước cần cấp phát	phải tính toán kích thước vùng nhớ cần cấp phát trước khi gọi	tự động tính toán kích thước vùng cần cấp phát dựa vào kiểu dữ liệ truyền vào
Cấp phát thất bại	Trả về con trỏ NULL	throw exception
Cấp phát thành công	Trả về con trỏ void *, muốn sử dụng phải ép kiểu về kiểu dữ liệu cần dùng	Gọi hàm khởi tạo của đối tượng cấp phát nếu đó là class Kiểu con trỏ là kiểu của đối tượng cấp phát
Loại đối tượng	Hàm	Toán tử (operator)
Khả năng override	Không thể	Có thể

Lưu ý:

+ malloc nhanh hơn so với calloc. Lý do là calloc ngoài việc có nhiệm vụ cấp phát vùng nhớ như malloc, nó còn phải gán giá trị cho tất cả các phần tử của vùng nhớ vừa cấp phát = 0.

+ Khi cấp phát bằng new thì vùng nhớ không thể thay đổi kích thước. malloc thì có thể bằng cách sử dụng hàm realloc.

c. Mảng 2 chiều, bản chất như thế nào, khác gì mảng một chiều ?



- Nhìn vào hình ta có thể dễ dàng hình dung ra được con trỏ cấp 2 như một vỏ bọc bao bọc 3 con trỏ cấp 1 bên trong.

- Lưu ý: con trỏ cấp 2 sẽ trỏ đến phần tử a [0][0].

Từ đây ta dễ dàng hình dung con trỏ cấp 3, cấp 4 sẽ thành hình dạng 3d, 4d…

CHAP V: CON TRỎ VỚI HÀM, CON TRỎ HÀM

1. Hàm cũng có địa chỉ

Khi 1 chương trình (1 pe file) chạy (tiến trình) thì các hàm nằm bên chương trình đó được được load lên không gian nhớ ảo, VA space, chúng nằm trong vùng nhớ code.

2. Con trỏ hàm

Con trỏ hàm là 1 điều thú vị trong C/C++, bản chất của con trỏ hàm cũng là 1 con trỏ có định kiểu.
Ta có thể sử dụng con trỏ hàm để gọi hàm khi đã biết địa chỉ của hàm.

Ví dụ: Gọi nội ứng dụng
demo 1 ví dụ

```
#include <stdio.h>
#include <conio.h>
int main(int a,int b)
{
if (a>b) return a;
return b;
}
```

```
void main()
{
int (*p)(int,int);
p=min;
printf("min cua 4 va 5 la %d",p(4,5));
getch();
}
```

Chú ý : khi khai báo ta phải dùng toán tử () với ý nghĩa là * này thuộc về p, là 1 con trỏ hàm. int (*p)(int,int);

3. Ứng dụng của con trỏ hàm

Con trỏ hàm được ứng dụng trong nhiều trường hợp khác nhau khá rộng rãi.

+ Trường hợp đơn giản tất cả chúng ta đều sử dụng đó là cout<<endl;
+ Sử dụng trong các hàm mẫu, lớp mẫu , có tính tùy chọn cao.
+ Sử dụng để gọi hàm trong 1 ứng dụng khác khi đã biết địa chỉ của hàm đó.

4. Con trỏ với hàm (quan trọng)

4.1. Tổng quan về hàm:

- Hàm có tham số nhận giá trị: giá trị truyền vào hàm có thể là giá trị của biến, một hằng số hoặc một biểu thức.
- Hàm có tham số kiểu tham chiếu: giá trị truyền vào cho hàm là tên biến, và tham số của hàm sẽ tham chiếu trực tiếp đến vùng nhớ của biến đó.

4.2. Sai lầm thường gặp

- Có nhiều thật nhiều người nói rằng trong C, ta có thể sử dụng con trỏ trong tham số của hàm như là 1 tham biến, qua hàm ta có thể thay đổi được giá trị của tham số. Điều này thật là 1 hiểu lầm:

Nguyên nhân:

- Hàm trong C không hề có tham biến, hàm trong C đều hoạt động theo nguyên tắc sau:

- Khi gọi hàm, 1 bản sao của tham số được tạo ra (cấp phát vùng nhớ mới, sao chép giá trị sang, quá trình này gọi là shadow copy, là 1 yếu tố cần quan tâm,và hàm sẽ làm việc với bản sao này.
- Vậy khi làm việc với con trỏ thì hàm làm thế nào?

Hàm vẫn cứ làm theo nguyên tắc 1 và 1 bản sao của con trỏ được tạo ra, và hàm làm việc với bản sao hàm, và trước khi gọi hàm con trỏ trỏ vào đâu thì nó vẫn được trỏ vào

đấy. Đặc biệt, ta không thể thay đổi được địa chỉ của vùng nhớ đang trỏ mà chỉ thay đổi được giá trị.

4.3. Sai lầm trong hành động

Một trong những sai lầm cơ bản nhưng lại hay gặp đó là ví dụ sau.

Sai lầm vì trong hàm chúng ta cấp phát và gán địa chỉ cho bản sao của x (cơ chế hoạt động của hàm). Và thực tế con trỏ x chưa trỏ tới đâu cả, và vùng nhớ đã cấp phát bị mất địa chỉ - đây gọi là vùng nhớ mồ côi.

Ví dụ:

```
void nhap(int *a,int n)
{
a=(int*)malloc(n * sizeof(int));
for(int i=0;i<n;i++)
cin>>a[i];
}
```

```
void main()
{
int *x;
int n=6;
nhap(x,n);
//xuat
delete[] x; // sản sinh ra lỗi run-time , x chưa có vùng nhớ mà đã giải phóng.
}
```

4.4. Vậy làm thế nào thay đổi vùng nhớ của con trỏ trong hàm:

Cách 1 : dùng tham chiếu trong C++

```
void ham(int *&a)
{
a=new int[100];
}
void ham(int **&a)
{
a=new int*[100];
}
```

Chú ý là * đứng trước &

Cách 2 : up level của * dùng con trỏ cấp cao hơn con trỏ hiện tại

```
#include <stdio.h>
#include <conio.h>

void ham(int **a)
{
*a=(int*)malloc(100*sizeof(int));
}
```


DÙ BẠN MUỐN LÀM GÌ
THÌ HÃY LÀM NGAY
ĐÃ QUÁ NHIỀU NGÀY MAI RỒI
#awakepowerquotes