

Ngôn ngữ lập trình C


Bài 7. Con trỏ, Mảng và Hàm

Soạn bởi: TS. Nguyễn Bá Ngọc

Nội dung

- Kiểu con trỏ và con trỏ
- Kiểu mảng và mảng
- Các phép toán với con trỏ
- Hàm, mảng và con trỏ

Nội dung

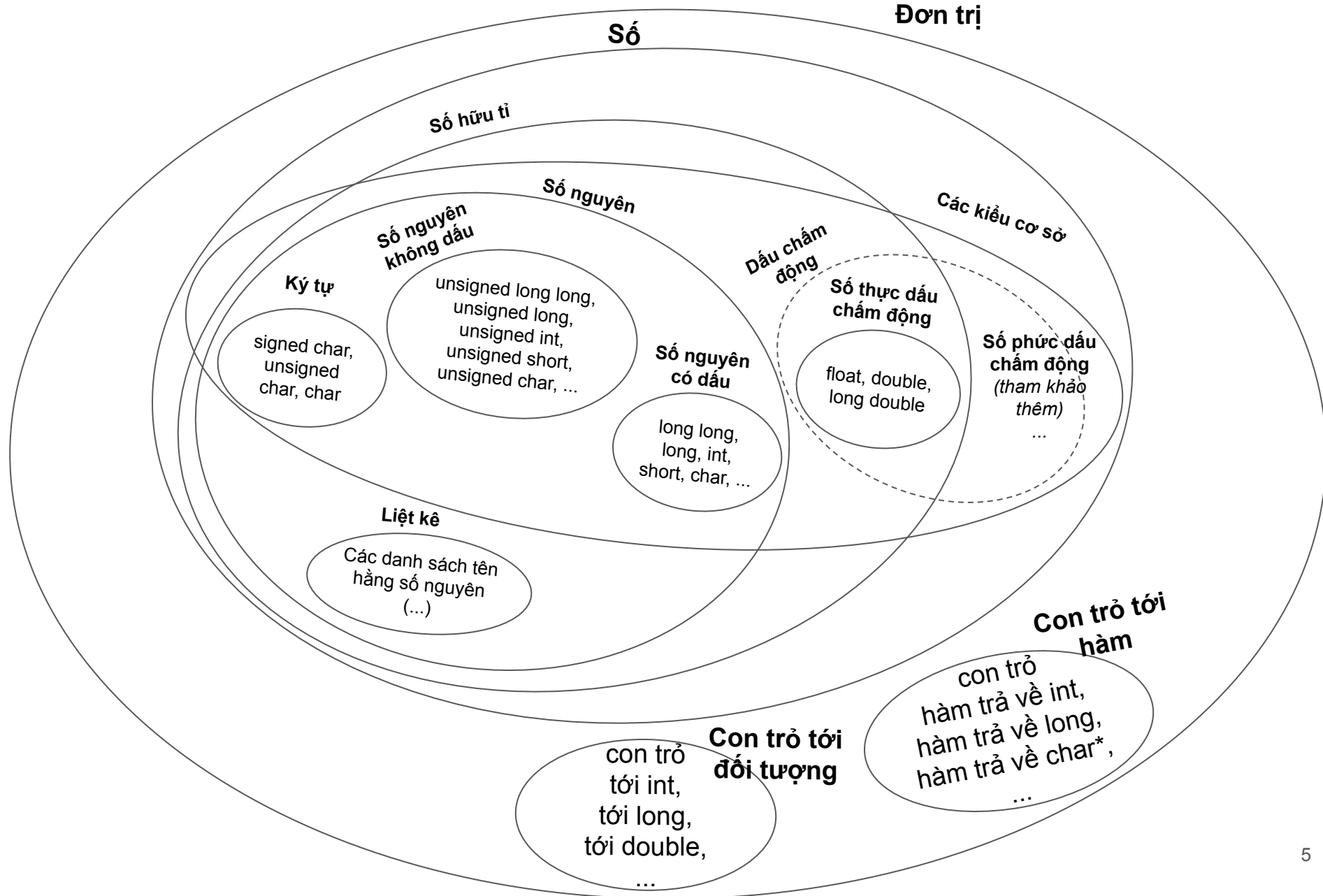
- 
- Kiểu con trỏ và con trỏ
 - Kiểu mảng và mảng
 - Các phép toán với con trỏ
 - Hàm, mảng và con trỏ

Kiểu con trỏ và đối tượng con trỏ

- Kiểu con trỏ mô tả các đối tượng có giá trị là **tham chiếu** của đối tượng khác hoặc tham chiếu của hàm.
- Kiểu con trỏ được định nghĩa theo cơ chế suy diễn từ kiểu đã có.
 - Nếu T là một kiểu thì kiểu con trỏ được định nghĩa từ T được gọi là kiểu con trỏ tới T.
- Cơ chế suy diễn kiểu con trỏ có thể được áp dụng đệ quy, trong trường hợp đó chúng ta có kiểu con trỏ tới kiểu con trỏ ...

*Chúng ta gọi đối tượng thuộc kiểu con trỏ tới T là con trỏ tới T, và đối tượng thuộc kiểu con trỏ được gọi đơn giản là **con trỏ**.*

Một số kiểu dữ liệu trong C99



Sơ lược cú pháp con trỏ tới đối tượng

- Nếu `T D;` khai báo định danh `D` là 1 đối tượng thì *trong điều kiện cú pháp hợp lệ*:
 - `T *P` khai báo `P` là 1 con trỏ tới kiểu của `D`, `P` có thể trỏ tới `D`;
 - `typedef T *Pt;` định nghĩa `Pt` là kiểu con trỏ tới kiểu của `D` (kiểu được xác định cho `P`)
- Ví dụ:
 - `int x;` // `x` là 1 đối tượng kiểu `int`
 - `int *p;` // `p` là 1 con trỏ tới `int`
 - `typedef int *pt;` // `pt` là kiểu con trỏ tới `int`
 - `pt p1, p2;` // `p1` và `p2` là các con trỏ tới `int`
 - `p = &x;` // `p` trỏ tới `x`; `p` lưu tham chiếu tới `x`
 - `int *;` // Kiểu con trỏ tới `int`
 - `int *x, y, z, *t;` // `y, z` là các đối tượng; `x, t` là các con trỏ

*Dấu * thường được viết liền với định danh*

Sơ lược cú pháp con trỏ tới hàm

- Nếu `T F(/*...*/);` khai báo định danh `F` là 1 hàm thì *trong điều kiện cú pháp hợp lệ*:
 - `T (*PF) (/* ... */);` // Khai báo `PF` là con trỏ tới kiểu hàm của `F`.
 - `typedef T(*PFt) (/*...*/);` // `PFt` là kiểu con trỏ tới kiểu hàm của `F`.
 - (Chúng ta cũng có thể khai báo con trỏ hàm dựa trên kiểu hàm:)
 - `typedef T Ft(/*..*/);` // Khai báo `Ft` là kiểu hàm (như kiểu của `F`).
 - `Ft *PF1;` // `PF1` là con trỏ hàm, tương đương với `PF`
- Ví dụ:
 - `int f(float x);` // `f` là 1 hàm nhận 1 tham số float và trả về int
 - `int *f1(float x);` // `f1` nhận 1 tham số float và trả về con trỏ int *
 - `int (*pf)(float x);` // `pf` là con trỏ tới hàm nhận 1 float và trả về int.
 - `typedef (*ft)(float x);` // `ft` là kiểu con trỏ hàm, *kiểu của pf*.
 - `pf = &f;` // `pf` trỏ tới `f`
 - `pf = f;` // `pf` trỏ tới `f`, C quy ước biểu thức `f` tương đương với `&f`

*Dấu * và định danh của con trỏ hàm cần được đặt trong ()*

Toán tử địa chỉ và toán tử chỉ định

- Toán tử địa chỉ: &

- Toán hạng phải là đối tượng hoặc hàm *hoặc thực thể tương đương*, cho kết quả là con trỏ tới toán hạng.
 - Ngoài đối tượng và hàm, toán hạng có thể là phần tử mảng, kết quả của biểu thức chỉ số, v.v..
- Nếu toán hạng có kiểu T thì kết quả có kiểu con trỏ tới T.
- Ví dụ: `int x; int f() { /*...*/ }`
- `&x` => con trỏ tới biến x. `&f` => con trỏ tới hàm f

- Toán tử chỉ định: *

- Toán hạng phải là con trỏ, cho kết quả là đối tượng hoặc hàm *hoặc thực thể tương đương* đang được trỏ tới bởi toán hạng.
- => *Nếu toán hạng có giá trị không hợp lệ thì kết quả là bất định.*
- Nếu toán hạng có kiểu con trỏ tới T thì kết quả có kiểu T.
- Ví dụ: `int *p = x; int (*pf)(void) = f; int *p1 = 0; { int *p2; }`
- `*p` => chỉ định biến x. `*pf` => Chỉ định hàm f.
- `*p1 = 100; // Hành vi bất định (chỉ định con trỏ NULL).`
- `*p2 = 200; // Thường dẫn tới lỗi (do p2 chưa được khởi tạo).`

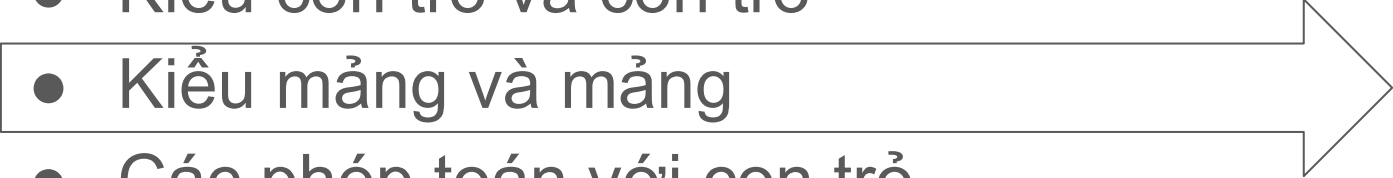
Sơ lược quy tắc ép kiểu với kiểu con trỏ

- Có thể ép kiểu con trỏ tới **đối tượng** với kiểu bất kỳ thành con trỏ tới **void**, rồi sau đó nếu ép kiểu con trỏ **void *** thu được thành kiểu ban, thì con trỏ thu được sau chuỗi ép kiểu bằng con trỏ ban đầu.
 - Ví dụ: `void print_i(void *x) {printf("%d", *((int*)x));}`
`int x = 100; print_i((void*)&x);`
 - *Quy chuẩn C không cung cấp khả năng ép kiểu con trỏ hàm thành con trỏ void **
- Ép kiểu con trỏ thành số nguyên và ngược lại là hành vi phụ thuộc triển khai
 - Giá trị địa chỉ trong các triển khai thường là số nguyên.
 - `int *p = &x; int y = (int)p; p = (int*)y; // phụ thuộc triển khai.`
 - Các triển khai có thể định nghĩa **intptr_t** và **uintptr_t** là các kiểu số nguyên đủ rộng để lưu giá trị địa chỉ, *có thể ép kiểu con trỏ void * thành intptr_t hoặc uintptr_t và ngược lại.*

Sơ lược quy tắc ép kiểu với kiểu con trỏ₍₂₎

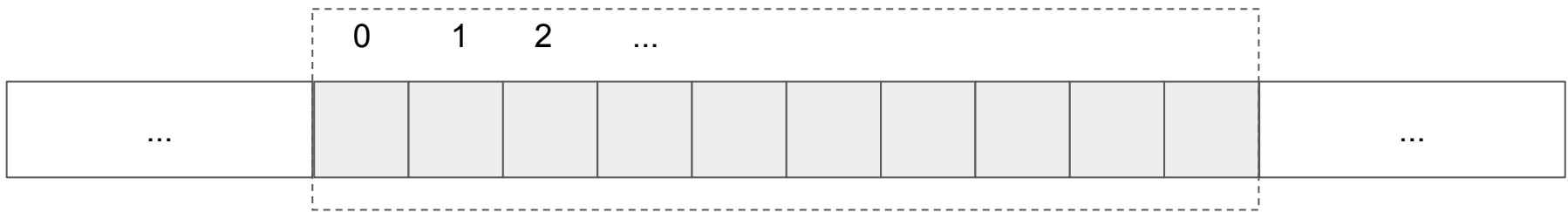
- Khi ép kiểu con trỏ tới *đối tượng* thành con trỏ khác kiểu, nếu *cấu trúc bộ nhớ là hợp lệ* thì thông tin được bảo toàn, nếu ngược lại thì hành vi là bất định.
 - Ví dụ 1: `int a[100];` // Mảng 100 phần tử int, sẽ học sau
`char *pc = a; int *pi = pc;` // OK: `pi == a`
 - Ví dụ 2: `char s[100];`
`int *pi = s; char *pc = pi;` // NOK: pc có thể `!= s`
- Có thể ép kiểu con trỏ tới hàm thành con trỏ tới kiểu hàm bất kỳ khác, nếu sau đó ép kiểu con trỏ thu được thành kiểu ban đầu thì kết quả = con trỏ ban đầu.
 - Thường được sử dụng để gọi hàm bằng con trỏ.
 - Ví dụ: `void (*pf)(void)=sum; x=((int (*)(int, int))pf)(1, 2);` // OK
- Giá trị 0 có thể được ép kiểu thành con trỏ thuộc kiểu bất kỳ (con trỏ null của kiểu đó). Con trỏ null trỏ tới void được đặt tên là NULL (`stddef.h`, `#define NULL ((void*)0)`).

Nội dung

- Kiểu con trỏ và con trỏ
 - Kiểu mảng và mảng
 - Các phép toán với con trỏ
 - Hàm, mảng và con trỏ
- 

Khái niệm mảng

- Kiểu mảng mô tả tập hợp các đối tượng cùng kiểu được cấp phát liên nhau trong bộ nhớ. Kiểu của các đối tượng thuộc mảng được gọi là kiểu phần tử.
- Kiểu mảng được xác định theo kiểu phần tử và số lượng phần tử, vì vậy kiểu mảng còn được coi là kiểu suy diễn từ kiểu phần tử của nó:
 - Nếu các phần tử có kiểu T thì chúng ta gọi kiểu mảng tương ứng là mảng của T.
 - Đối tượng có kiểu mảng được gọi đơn giản là mảng.



*Kích thước mảng = kích thước phần tử * số lượng phần tử*

Sơ lược cú pháp mảng

- Chúng ta có thể phân biệt 3 định dạng khai báo mảng theo cách xác định số lượng phần tử của mảng:
 - Mảng với kích thước cố định: Kích thước các chiều là hằng số nguyên dương và được cung cấp trong khai báo mảng.
 - Mảng với số lượng phần tử ngầm định: Kích thước chiều trái nhất được xác định tự động dựa trên biểu thức khởi tạo.
 - Mảng với độ dài thay đổi: Kích thước các chiều trong khai báo là các biến hoặc biểu thức có giá trị được xác định trong thời gian thực hiện chương trình (VLA, C99).
- Mảng với độ dài thay đổi/Variable Length Array (VLA, C99)
 - Có 1 số quy tắc riêng cho các trường hợp, ví dụ không được khởi tạo trong khai báo, v.v..
 - => *Tham khảo thêm, không phân tích chi tiết trong bài giảng này, (có thể được thay thế bởi bộ nhớ cấp phát động.)*

Sơ lược cú pháp mảng 1 chiều

Mảng 1 chiều là định dạng mảng được sử dụng phổ biến nhất

- Khai báo với kích thước cố định:
 - Nếu T D; khai báo D là 1 đối tượng kiểu T thì T A[N]; khai báo A là 1 mảng gồm N phần tử (tương tự như D) có kiểu T
 - [N] được thêm vào sau định danh.
 - N phải là 1 hằng số nguyên dương, ví dụ 10, 100, 1000, ...
 - `int x, a[10];` // x là biến kiểu int, a là mảng 10 phần tử kiểu int.
 - `double *p;` // p là con trỏ tới double
 - `double *ap[100];` // ap là mảng 100 con trỏ tới double
- Khởi tạo giá trị của phần tử mảng:
 - Khởi tạo tuần tự: `int a[3] = {1, 2, 3};` // `a[0]=1; a[1]=2; a[2]=3;`
 - Lựa chọn phần tử theo chỉ số: `int a[100]={10, [10]=100, 101};`
// `a[0]=10; a[10]=100; a[11]=101`, còn lại = 0.
 - Nếu mảng được khởi tạo 1 phần thì phần còn lại được khởi tạo = 0. Nếu không có biểu thức khởi tạo thì áp dụng quy tắc *mặc định* theo phân lớp lưu trữ và phạm vi khai báo.

Mảng 1 chiều với kích thước ngầm định

- Khai báo:

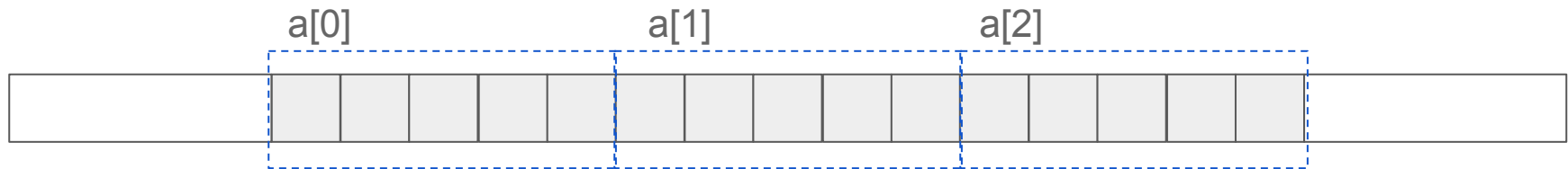
- Tương tự mảng với kích thước cố định có biểu thức khởi tạo nhưng để trống số lượng phần tử.
- `int a[] = {1, 2, 3};` // Tương đương với `int a[3] = {1, 2, 3};`
 - Nếu không sử dụng chỉ số thì số lượng phần tử = số lượng giá trị khởi tạo.
- `int b[] = {1, 2, [9] = 90};` // `int b[10] = {1, 2, [9] = 90};`
 - Nếu có sử dụng chỉ số thì số lượng phần tử = chỉ số lớn nhất + 1.

- Toán tử sizeof:

- Với `a` là 1 mảng thì `sizeof(a)` trả về kích thước mảng (= số lượng phần tử * kích thước phần tử).
- $\Rightarrow \text{sizeof}(a) / \text{sizeof}(a[0])$ cho kết quả là số lượng phần tử mảng.

Mảng nhiều chiều

- Trong C mảng n chiều ($n \geq 2$) với kích thước $i * j * \dots * k$ được coi như mảng 1 chiều của i mảng $n - 1$ chiều với kích thước $j * \dots * k$.
 - Định dạng này còn được gọi là định dạng hướng dòng.
 - Khái niệm mảng 1 chiều được áp dụng đệ quy.
- Xét 1 mảng 2 chiều:
 - `int a[3][5];` // Kích thước các chiều là 3×5
 - Có thể coi a như mảng 1 chiều của 3 mảng 1 chiều 5 phần tử `int`.

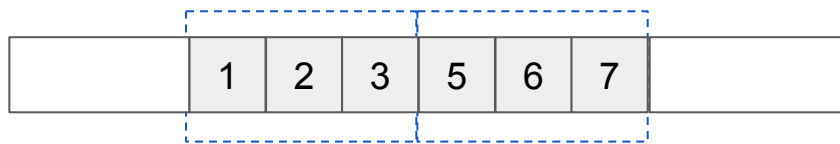


- (Mảng với số chiều > 2 có thể được mở rộng tương tự).

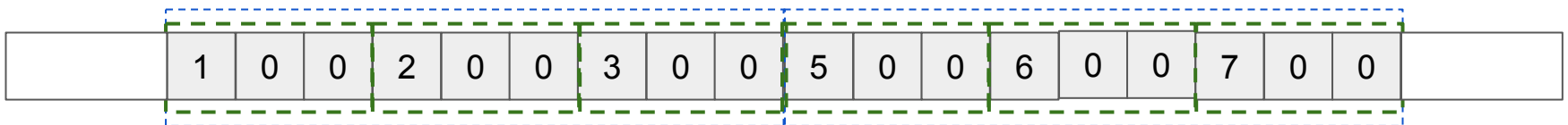
Mảng nhiều chiều ít phổ biến hơn mảng 1 chiều, và có thể được thay thế bằng mảng con trỏ với bộ nhớ cấp phát động.

Mảng nhiều chiều với kích thước cố định

- Khai báo lần lượt kích thước các chiều, ví dụ:
 - `a[10][20];` // Mảng 2 chiều của $10 * 20$ đối tượng `int`
 - `a[10][20][30]` // Mảng 3 chiều của $10 * 20 * 30$ đối tượng `int`
- Khởi tạo mảng nhiều chiều:
 - Có thể khởi tạo với 1 danh sách các giá trị tuần tự:
 - `int a[2][3] = {1, 2, 3, 5, 6, 7};` // `a[0][0] = 1, a[0][1] = 2, ...` Các phần tử được khởi tạo tuần tự theo thứ tự .



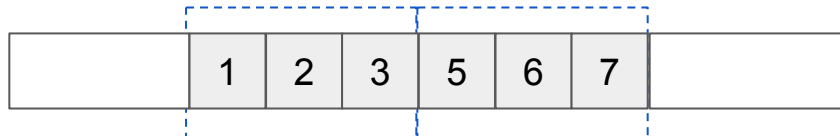
- `int a[2][3] = {[1][1] = 100};` // Lựa chọn theo chỉ số
- Hoặc phân cấp với các cặp `{}` lồng nhau:
- `int a[2][3] = {{1, 2, 3}, {5, 6, 7}};`
- `int a[2][3][3] = {{{1}, {2}, {3}}, {{5}, {6}, {7}}}`



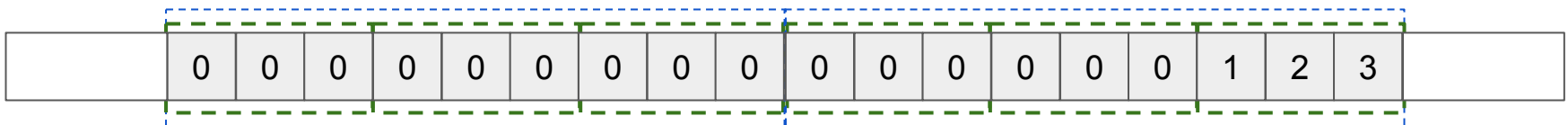
Mảng nhiều chiều với kích thước ngầm định

Chúng ta có thể ngầm định chiều đầu tiên (chiều trái nhất) của mảng nhiều chiều và khai báo mảng n chiều ($n > 1$) như mảng 1 chiều của các mảng $n - 1$ chiều.

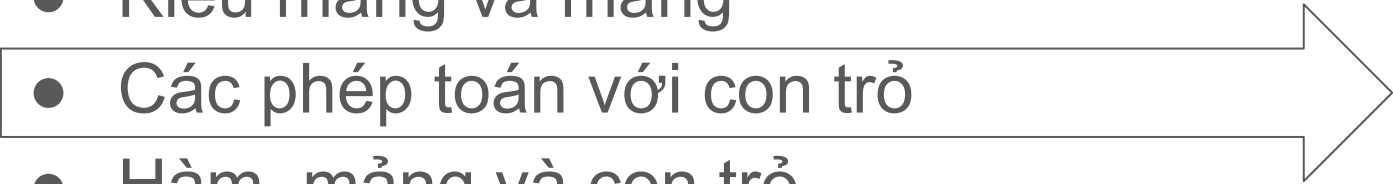
- Sử dụng danh sách khởi tạo 1 chiều:
 - `int a[][3] = {1, 2, 3, 5, 6, 7}; // a[2][3];` chiều đầu tiên là giá trị nhỏ nhất đủ lớn để chứa tất cả các phần tử khởi tạo.
- Sử dụng các danh sách khởi tạo lồng nhau:
 - `int a[][3] = {{1, 2, 3}, {5, 6, 7}}; // => a[2][3]`



- Kích thước mảng thành phần được xác định bởi các chiều còn lại. Chúng ta cũng có thể sử dụng chỉ số:
- `int a[][3] = {[5] = {1, 2, 3}}; // => a[6][3]`



Nội dung

- Kiểu con trỏ và con trỏ
 - Kiểu mảng và mảng
 - Các phép toán với con trỏ
 - Hàm, mảng và con trỏ
- 

Cộng, trừ với toán hạng là con trỏ

- Khi cộng 1 số nguyên n với 1 con trỏ, nếu con trỏ đang trỏ vào phần tử thứ i của mảng và kích thước mảng đủ lớn thì kết quả là con trỏ tới phần tử thứ $i + n$ trong mảng.
- Khi trừ 1 số nguyên n từ con trỏ P ($P - n$), nếu con trỏ đang trỏ vào phần tử thứ i của mảng và kích thước mảng đủ lớn thì kết quả là con trỏ tới phần tử thứ $i - n$ trong mảng.
- Trường hợp đặc biệt:
 - Nếu P đang trỏ vào phần tử cuối cùng của mảng thì $P + 1$ cho kết quả là con trỏ vào phần tử liền sau phần tử cuối cùng (1 vị trí lô-gic, không phải là 1 phần tử của mảng).
 - Nếu Q đang trỏ vào phần tử liền sau phần tử cuối cùng của mảng thì $Q - 1$ là con trỏ tới phần tử cuối cùng của mảng.
- Cộng hoặc trừ con trỏ tới phần tử mảng với số nguyên trong các trường hợp còn lại là hành vi bất định.

Cộng, trừ với toán hạng là con trỏ₍₂₎

- Khi trừ 2 con trỏ P và Q, nếu P và Q theo thứ tự đang trỏ vào phần tử thứ i và j của cùng 1 mảng thì kết quả P - Q là i - j và có thể được biểu diễn bằng kiểu ptrdiff_t (stddef.h).
- Trường hợp đặc biệt: Nếu mảng có n phần tử thì chỉ số của phần tử liền sau phần tử của cuối cùng trong phép trừ 2 con trỏ = n.
- Chúng ta cũng có thể giản lược các biểu thức cộng và trừ với toán hạng là con trỏ, ví dụ, với p là con trỏ:
 - `p += n; // p = p + n`
 - `p -= n; // p = p - n`
 - `++p; --p; // p += 1; p -= 1;`
 - `p++; p--; // tmp = p; p+= 1 (hoặc p-=1); kết quả là tmp;`

Các phép toán với con trỏ void là không hợp lệ trong quy chuẩn ISO.*

So sánh thứ tự với con trỏ

Có thể so sánh thứ tự các con trỏ tới các đối tượng (các toán tử $<$, $>$, $<=$, $>=$):

- Kết quả so sánh con trỏ phụ thuộc vào vị trí tương đối của các đối tượng được trỏ tới trong bộ nhớ.
- Nếu các đối tượng được trỏ đến đều là các trường của 1 đối tượng cấu trúc thì con trỏ tới trường được khai báo sau được coi là $>$ con trỏ tới trường được khai báo trước.
 - *(Chi tiết về cấu trúc sẽ được học sau)*
- Nếu cả 2 con trỏ cùng trỏ tới các phần tử của cùng 1 mảng thì kết quả so sánh tương đương với so sánh các chỉ số mảng tương ứng.
- Nếu con trỏ P trỏ tới 1 phần tử thuộc mảng và con trỏ Q trỏ tới phần tử liền sau phần tử cuối cùng của mảng thì $P < Q$.

So sánh bằng với con trỏ

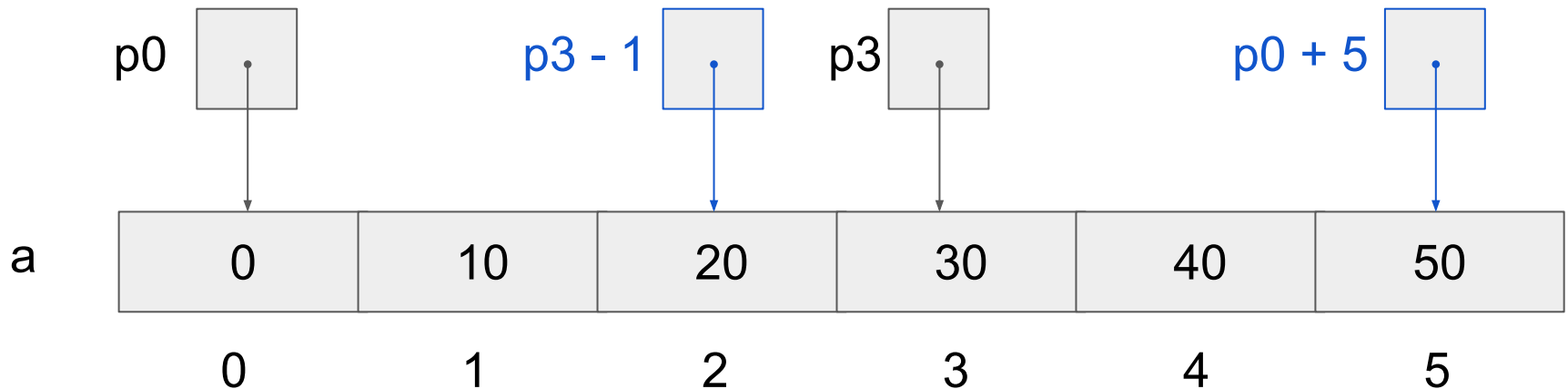
Các toán tử (==, !=)

- So sánh bằng với 2 con trỏ cho kết quả đúng trong các trường hợp cả 2 con trỏ:
 - Đều là các con trỏ null.
 - Đều trỏ đến cùng 1 đối tượng hoặc cùng 1 hàm.
 - Cùng trỏ đến 1 phần tử của mảng.
 - Cùng trỏ đến phần tử liền sau phần tử cuối cùng của mảng.
- Trường hợp đặc biệt: Nếu 1 con trỏ đang trỏ đến phần tử liền sau phần tử cuối cùng của mảng và 1 con trỏ đang trỏ tới phần tử đầu tiên của 1 mảng được cấp phát ngay sau nó thì kết quả so sánh 2 con trỏ bằng nhau là đúng.
 - 2 đối tượng khác nhau có thể được cấp phát liền kề nhau trong bộ nhớ.
 - Các mảng thành phần của 1 mảng nhiều chiều. v.v..

Minh họa phép toán với con trỏ

```
int a[6] = {0, 10, 20, 30, 40, 50};  
int *p0 = &a[0], *p3 = &a[3];
```

```
p0 - p3 == -3  
*(p3 - 1) == 20  
*(p0 + 5) == 50
```



```
int *p = &a[1], *q = &a[3];  
p < q; // Đúng  
p = &a[6]; // Liên sau a[5]  
p > q; // Đúng  
q += 3; // q == &a[6]  
p == q; // Đúng
```


Đối tượng chỉ đọc và con trỏ chỉ đọc

Chúng ta cũng có thể giới hạn khả năng thay đổi giá trị của con trỏ như với các đối tượng khác.

- Con trỏ chỉ đọc:
 - `int x, y; // int *p; // p là con trỏ tới int`
 - `int * const p = &x; // p là con trỏ chỉ đọc`
 - `*p = 100; // OK, x = 100`
 - `p = &y; // Lỗi thay đổi con trỏ chỉ đọc`
- Con trỏ tới đối tượng chỉ đọc:
 - `const int x = 3, y = 5; // x và y là các đối tượng chỉ đọc`
 - `const int *p = &x; // p là con trỏ tới đối tượng chỉ đọc`
 - `*p = 100; // Lỗi thay đổi đối tượng chỉ đọc`
 - `p = &y; // OK - có thể thay đổi con trỏ p`
- Có thể khai báo con trỏ chỉ đọc với kiểu con trỏ:
 - `typedef int * int_ptr;`
 - `const int_ptr p; // p là con trỏ chỉ đọc`

Mảng và con trỏ

Tên mảng có thể được sử dụng như 1 con trỏ chỉ đọc trỏ vào phần tử đầu tiên của mảng:

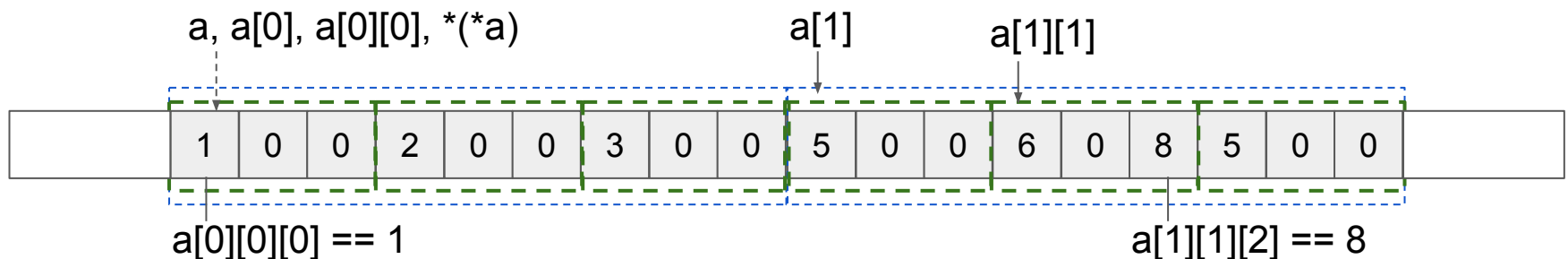
- Trong các biểu thức, trừ trường hợp là toán hạng của toán tử sizeof hoặc toán tử địa chỉ, mảng của phần tử kiểu T được ép kiểu thành con trỏ tới T.
 - Nếu a là 1 mảng và i là 1 chỉ số hợp lệ thì $a + i$ có kết quả là con trỏ tới phần tử thứ i của mảng.
 - Biểu thức a có kết quả là con trỏ tới phần tử đầu tiên của mảng (a[0]).
- Ví dụ: `int x, a[100];` // a là mảng 100 phần tử kiểu int
 - `int *p = a;` // p trỏ tới phần tử đầu tiên của mảng
 - `// a = &x;` // lỗi biên dịch (không được thay đổi mảng).
 - `p + 10 == a + 10;`
 - `*(a + 10);` // Phần tử có chỉ số 10 của mảng a

Toán tử chỉ số []

- Trong biểu thức $E[i]$ thì E phải là con trỏ tới đối tượng và i phải có kiểu số nguyên.
- Biểu thức $E[i]$ tương đương với $((E) + (i))$. Nếu E trỏ tới kiểu T thì $E[i]$ cho kết quả là đối tượng kiểu T .
- Khi kết hợp với toán tử địa chỉ trong biểu thức dạng $\&E[i]$ thì biểu thức được chuyển về dạng $((E) + (i))$, cho kết quả là con trỏ tới phần tử thứ i (cả $\&$ và $*$ đều được lược bỏ).
- Đối với mảng 1 chiều, ví dụ `int a[100];`
 - Biểu thức `a` cho kết quả là con trỏ `int*` trỏ tới phần tử đầu tiên của mảng.
 - Biểu thức `a[10]` tương đương với $((a) + 10)$, cho kết quả là 1 đối tượng kiểu `int`.
 - `a` trỏ tới phần tử đầu tiên, chính vì vậy phần tử đầu tiên phải có chỉ số là 0: `a[0]` tương đương với `*a`.

Toán tử chỉ số $[]_{(2)}$

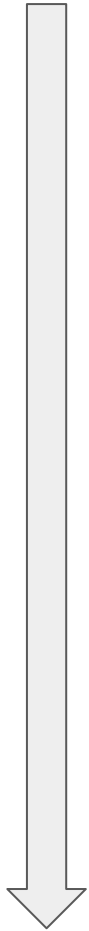
- Đối với mảng nhiều chiều, ví dụ `int a[2][3][3]`;
 - Mảng `a` bao gồm $2 \times 3 \times 3$ phần tử `int` liên tiếp.
 - Biểu thức `a` cho kết quả là con trỏ kiểu `int (*)[3][3]`;
 - $\Rightarrow a[i]$ là 1 mảng 3×3 và được chuyển thành con trỏ `int (*)[3]`;
 - $\Rightarrow a[i][j]$ là 1 mảng 3 phần tử và được chuyển thành `int *`.
 - Với chỉ số thứ 3, `a[i][j][k]` cho kết quả là 1 đối tượng kiểu `int`.
- Xét ví dụ truy cập phần tử `a[1][1][2]`:



- Do tính chất liên tục của các phần tử mảng:
 - Nếu `a` được cấp phát bộ nhớ từ địa chỉ `x` thì vùng nhớ của phần tử đầu tiên cũng bắt đầu từ `x` (`(int*) a` trỏ tới `a[0][0][0]`).
 - Có thể coi mảng nhiều chiều như mảng 1 chiều với cùng số lượng phần tử và ngược lại.

Thứ tự ưu tiên và chiều thực hiện chuỗi toán tử

Ưu tiên cao
(thực hiện trước)



Ưu tiên thấp
(thực hiện sau)

Tên toán tử	Ký hiệu	Thứ tự
Chỉ số Gọi hàm Tăng 1, giảm 1 (hậu tố)	<code>[]</code> <code>a[10]</code> <code>()</code> <code>f(3, 5)</code> <code>++, --</code> <code>(x++, x--)</code>	Trái -> Phải
Địa chỉ, chỉ định Tăng, giảm 1 (tiền tố) Dấu, phủ định lô-gic sizeof	<code>&, *</code> <code>(&x, *p)</code> <code>++, --</code> <code>(++x, --x)</code> <code>+, -, !</code> <code>(+x, -x, !e)</code> <code>sizeof</code> <code>(sizeof(int))</code>	Phải -> Trái
Ép kiểu	<code>()</code> <code>(double)5</code>	Phải -> Trái
Nhân, Chia, phần dư	<code>*, /, %</code> <code>(x * y, x / y, x%y)</code>	Trái -> Phải
Cộng Trừ	<code>+</code> <code>(x + y)</code> <code>-</code> <code>(x - y)</code>	Trái -> Phải
Dịch sang trái Dịch sang phải	<code><<</code> <code>(x << y)</code> <code>>></code> <code>(x >> y)</code>	Trái -> Phải
So sánh thứ tự	<code><, >, <=, >=</code>	Trái -> Phải
So sánh bằng	<code>==, !=</code>	Trái -> Phải
AND theo bit	<code>&</code> <code>(x & y)</code>	Trái -> Phải
XOR theo bit	<code>^</code> <code>(x ^ y)</code>	Trái -> Phải
OR theo bit	<code> </code> <code>(x y)</code>	Trái -> Phải
AND lô-gic	<code>&&</code>	Trái -> Phải
OR lô-gic	<code> </code>	Trái -> Phải
Lựa chọn	<code>?:</code>	Phải -> Trái (Rẽ nhánh)
Gán đơn giản Gán kết hợp	<code>=</code> <code>*=, /=, %=, +=, -=, <<=, >>=,</code> <code>&=, ^=, =</code>	Phải -> Trái

Mảng 1 chiều và mảng nhiều chiều

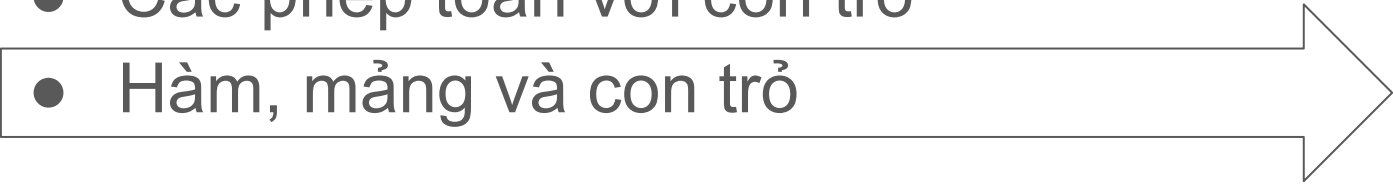
Chúng ta phân tích một số biểu diễn trên cùng 1 vùng nhớ:

```
int a[12] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

	0	1	2	3	4	5	6	7	8	9	10	11	
--	---	---	---	---	---	---	---	---	---	---	----	----	--

- Với `int *p = a;`
 - `*(p + 5)` tương đương với `a[5]`, tương đương với `p[5]` và `== 5`
- Với `int (*b)[2] = (int (*)[2])a; // con trỏ tới mảng 1 chiều`
 - Có thể sử dụng `b` như 1 mảng 2 chiều:
 - `b[3][0]` tương đương với `*(*(b + 3)) == 6.`
 - `b + 1` trỏ tới mảng 1 chiều 2 phần tử bắt đầu từ 2 trong `a`.
 - `*(b+1))[1]` tương đương với `a[3]` và `== 3.`
- Với `int (*c)[2][3] = (int(*)[2][3])a; // con trỏ tới mảng 2 chiều`
 - Có thể sử dụng `c` như 1 mảng 3 chiều
 - `c[1][1][1]` tương đương với `*(c + 1))[1][1]` và `== 10`

Nội dung

- Kiểu con trỏ và con trỏ
 - Kiểu mảng và mảng
 - Các phép toán với con trỏ
 - Hàm, mảng và con trỏ
- 

Hàm với tham số mảng

Chúng ta có thể sử dụng hàm để xử lý mảng:

- Do mảng có thể có kích thước lớn vì vậy thông thường chỉ có con trỏ tới mảng được truyền cho hàm
- Trong C tham số hàm có kiểu mảng của T được hiệu chỉnh (tự động) thành con trỏ tới T.
 - `int f(int a[100]);` // kiểu của a được hiệu chỉnh thành `int*`
 - `int f(int a[10][10]);` // - được hiệu chỉnh thành `int (*)[10]` - con trỏ tới mảng 10 phần tử `int`
- Trong trường hợp không sử dụng VLA:
 - Đối với mảng 1 chiều: Có thể sử dụng định dạng con trỏ và số lượng phần tử, ví dụ: `int f(int n, int *a);`
 - Đối với mảng nhiều chiều: Có 1 số lựa chọn tiêu biểu:
 - Cố định các chiều: ví dụ `int a[10][10];` - đơn giản nhưng ít hữu ích.
 - Xử lý như mảng 1 chiều: ví dụ `p[i * n + j]` là phần tử `[i][j]` trong mảng 2 chiều có kích thước `m * n` - khái quát nhưng phức tạp.
 - (Sử dụng cấp phát động (trong Heap) - sẽ học sau.)

Ví dụ 7-1a. Tính tổng các phần tử của mảng

Chương trình tính tổng các phần tử của mảng sử dụng chỉ số

```
vd7-1a.c x
6  #include <stdio.h>
7
8  long sum(const long n, long *a) {
9      long sum = 0;
10     for (long i = 0; i < n; ++i) {
11         sum += a[i];
12     }
13     return sum;
14 }
15 int main() {
16     long a[] = {1, 3, 5, 8, 9};
17     long n = sizeof(a) / sizeof(a[0]);
18     printf("sum = %ld\n", sum(n, a));
19     return 0;
20 }
21
```

bangoc:\$gcc -o prog vd7-1a.c
bangoc:\$./prog
sum = 26
bangoc:\$

Ví dụ 7-1b. Duyệt mảng 1 chiều bằng con trỏ

Chương trình tính tổng các phần tử của mảng sử dụng con trỏ

```
vd7-1b.c x
6  #include <stdio.h>
7
8  long sum(long *beg, long *end) {
9      long sum = 0;
10     for (long *p = beg; p < end; ++p) {
11         sum += *p;
12     }
13     return sum;
14 }
15 int main() {
16     long a[] = {1, 3, 5, 8, 9};
17     long n = sizeof(a) / sizeof(a[0]);
18     printf("sum = %ld\n", sum(a, a + n));
19     return 0;
20 }
```

```
bangoc:$gcc -o prog vd7-1b.c
bangoc:$./prog
sum = 26
bangoc:$
```

Ví dụ 7-1c. Xử lý mảng nhiều chiều

Chương trình duyệt các phần tử của mảng 2 chiều như mảng 1 chiều và tính tổng các phần tử

```
vd7-1c.c x
6  #include <stdio.h>
7
8  long sum(const long n, long *a) {
9      long sum = 0;
10     for (long i = 0; i < n; ++i) {
11         sum += a[i];
12     }
13     return sum;
14 }
15 int main() {
16     long a[2][3] = {1, 3, 5, 8, 9};
17     printf("sum = %ld\n", sum(2 * 3, a[0]));
18     return 0;
19 }
```

bangoc:\$gcc -o prog vd7-1c.c
bangoc:\$./prog
sum = 26
bangoc:\$

Hàm với tham số VLA

VLA được đưa vào quy chuẩn C từ C99, bên cạnh 1 số ý kiến trái chiều (ví dụ C++ không hỗ trợ VLA), VLA thực sự hữu ích cho việc truyền tham số mảng cho hàm:

- Xử lý mảng 1 chiều với VLA:
 - `int f(int n, int a[n]);` // a là VLA, có từ C99
 - `int f(const int n, int a[n]);` // chỉ đọc giá trị biến n
 - => Ưu điểm thể hiện được quan hệ giữa n và a.
 - *!Lưu ý: n phải được khai báo trước a*
 - `int f(int a[n], int[n]);` // không hợp lệ, chưa biết n khi khai báo a
- Xử lý mảng nhiều chiều:
 - `int f(int m, int n, int a[m][n]);`
 - `int f(const int m, const int n, int a[m][n]);`
 - *(Các mảng nhiều chiều hơn được truyền theo cách tương tự)*

Để truy cập đúng các phần tử mảng bằng chỉ số chúng ta cần biểu diễn mảng như khối nhớ liên tục: Tuy nhiên cần tránh cấp phát khối nhớ lớn trong Stack, chúng ta cũng có thể cấp phát khối nhớ liên tục trong Heap.

Ví dụ 7-2a. Nhân ma trận

*Chương trình nhân 2 ma trận
với biểu diễn mảng 1 chiều*

```
vd7-2a.c
6 #include <stdio.h>
7
8 // c(mxp) = a(mxn) %*% b(nxp)
9 void matrix_mult(int m, int n, int p,
10     double *a, double *b, double *c) {
11     for (int i = 0; i < m; ++i) {
12         for (int j = 0; j < p; ++j) {
13             double s = 0;
14             for (int t = 0; t < n; ++t) {
15                 s += a[i * n + t] * b[t * p + j];
16             }
17             c[i * p + j] = s;
18         }
19     }
20 }
21 int main() {
22     double a[3][2] = {{1, 1}, {1, 1}, {1, 1}},
23         b[2][3] = {{1, 2, 3}, {7, 6, 5}},
24         c[3][3];
25     matrix_mult(3, 2, 3, a[0], b[0], c[0]);
26     for (int i = 0; i < 3; ++i) {
27         for (int j = 0; j < 3; ++j) {
28             printf("\t%.0f", c[i][j]);
29         }
30         printf("\n");
31     }
32     return 0;
33 }
```

```
bangoc:$gcc -o prog vd7-2a.c
bangoc:$./prog
      8      8      8
      8      8      8
      8      8      8
bangoc:$
```


Ví dụ 7-2b. Nhân ma trận với VLA

```
vd7-2b.c x
6 #include <stdio.h>
7
8 // c(mxp) = a(mxn) %*% b(nxp)
9 void matrix_mult(int m, int n, int p, double a[m][n],
10                 double b[n][p], double c[m][p]) {
11     for (int i = 0; i < m; ++i) {
12         for (int j = 0; j < p; ++j) {
13             double s = 0;
14             for (int t = 0; t < n; ++t) {
15                 s += a[i][t] * b[t][j];
16             }
17             c[i][j] = s;
18         }
19     }
20 }
21 int main() {
22     double a[3][2] = {{1, 1}, {1, 1}, {1, 1}},
23             b[2][3] = {{1, 2, 3}, {7, 6, 5}},
24             c[3][3];
25     matrix_mult(3, 2, 3, a, b, c);
26     for (int i = 0; i < 3; ++i) {
27         for (int j = 0; j < 3; ++j) {
28             printf("\t%.0f", c[i][j]);
29         }
30         printf("\n");
31     }
32     return 0;
33 }
```

*Chương trình nhân 2 ma trận
sử dụng VLA*

```
bangoc:$gcc -o prog vd7-2b.c
bangoc:$./prog
      8      8      8
      8      8      8
      8      8      8
bangoc:$
```

Ví dụ 7-3. Mở rộng hàm với con trỏ hàm

```
vd7-3.c
6 #include <stdio.h>
7
8 void process(const int n, int *a, void (*op)(int*)) {
9     for (int i = 0; i < n; ++i) {
10         op(a + i);
11     }
12 }
13 void add1(int *p) { ++(*p); }
14 void mul3(int *p) { (*p) *= 3; }
15 void print_i(int *p) { printf(" %d", *p); };
16 void print_a(const int n, int *a) {
17     process(n, a, print_i);
18     printf("\n");
19 }
20 int main() {
21     int a[5] = {1, 2, 3, 4, 5};
22     print_a(5, a);
23     process(5, a, add1);
24     print_a(5, a);
25     process(5, a, mul3);
26     print_a(5, a);
27     int b[2][3] = {1, 2, 3, 5, 6, 7};
28     process(2 * 3, b[0], mul3);
29     for (int i = 0; i < 2; ++i) {
30         print_a(3, b[i]);
31     }
32     return 0;
33 }
```

```
bangoc:$gcc -o prog vd7-3.c
bangoc:$./prog
1 2 3 4 5
2 3 4 5 6
6 9 12 15 18
3 6 9
15 18 21
bangoc:$
```

Cộng 1 vào mỗi phần tử mảng

Nhân 3 với mỗi phần tử mảng

