

Ngôn ngữ lập trình C

Bài 7. Con trỏ, Mảng và Hàm

Soạn bởi: TS. Nguyễn Bá Ngọc

Nội dung

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

Nội dung

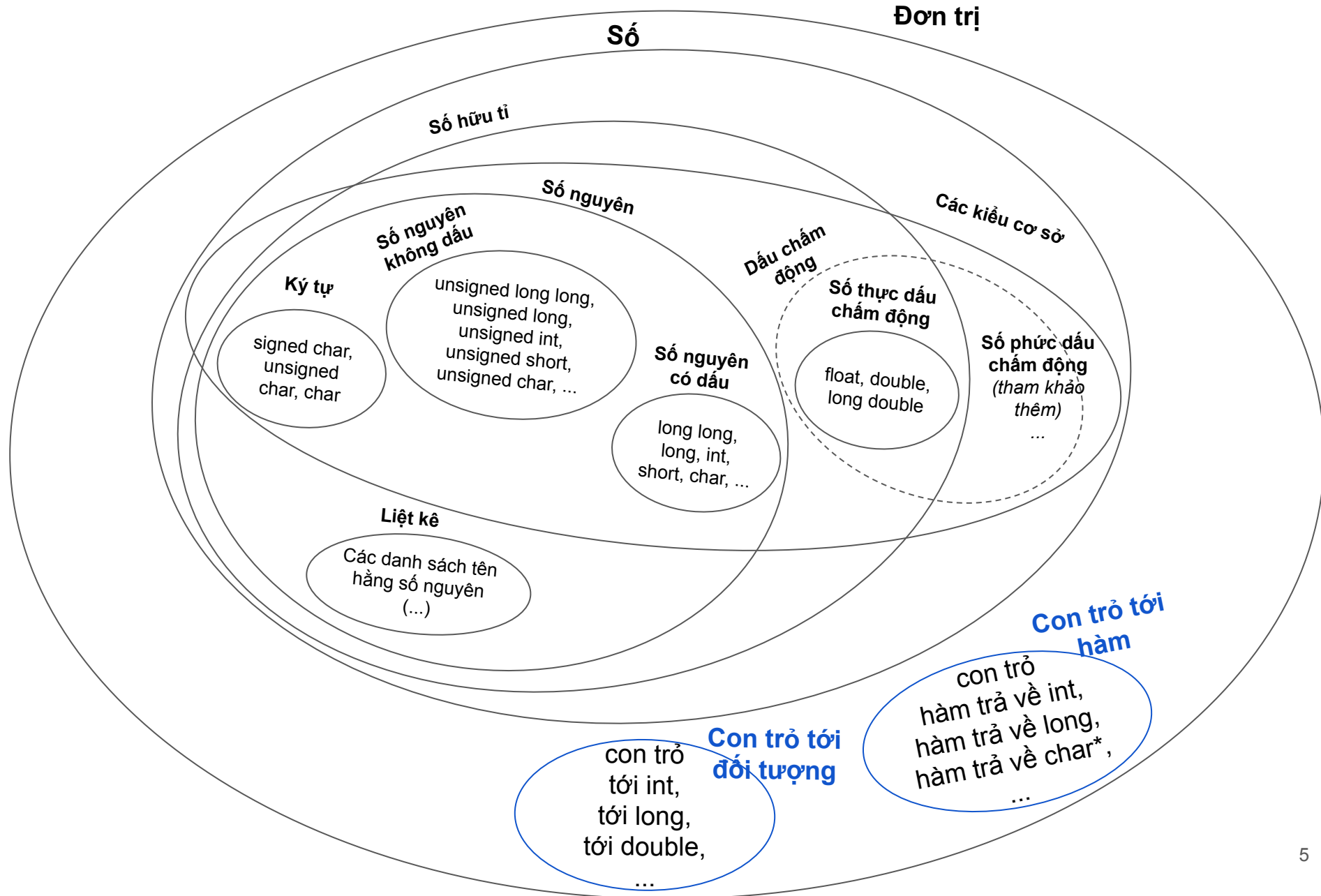
- Kiểu con trỏ và biến con trỏ
- Kiểu mảng và biến mảng
- Các phép toán với con trỏ
- Hàm, mảng và con trỏ

Các khái niệm

- Kiểu con trỏ được định nghĩa theo cơ chế suy diễn. Có thể suy diễn kiểu con trỏ từ kiểu đối tượng, kiểu hàm, và cả kiểu chưa hoàn thiện.
 - Kiểu con trỏ được suy ra từ kiểu **T** đôi khi được gọi ngắn gọn là con trỏ tới **T** - mô tả các đối tượng có thể hàm chứa dữ liệu cần thiết để truy cập tới/tương tác với các thực thể (đối tượng hoặc hàm) kiểu **T**.
 - Có thể áp dụng cơ chế suy diễn kiểu con trỏ theo đệ quy - chúng ta có thể có con trỏ tới con trỏ ...tới **T**
- Đối tượng/biến có kiểu con trỏ tới **T** cũng có thể được gọi ngắn gọn là con trỏ tới **T** tuy có thể nhập nhằng về nghĩa (giữa biến và kiểu).

Con trỏ có vai trò như đường dẫn tới các thực thể trong chương trình.

Các kiểu đơn trị trong C99



Sơ lược cú pháp khai báo con trỏ đối tượng

- Nếu T D; khai báo định danh D là 1 đối tượng kiểu T thì *trong điều kiện cú pháp hợp lệ*:
 - $T *P$ khai báo định danh P là 1 con trỏ tới T , P có thể trỏ tới các đối tượng kiểu T - ví dụ đối tượng D ;
- Ví dụ:
 - `int x;` // x là 1 đối tượng kiểu `int`
 - `int *p = &x;` // p là 1 con trỏ tới `int`, p trỏ tới x - có thể truy cập x thông qua p
 - `int*` - Con trỏ tới `int` - là kiểu của p
 - `int *x, y, z, *t;` // y, z có kiểu `int`; x, t có kiểu `int *`

Trong đoạn mã nguồn khai báo con trỏ dấu `` có ý nghĩa gắn với định danh vì vậy thường được viết liền với định danh*

Sơ lược cú pháp khai báo con trỏ hàm

Nếu T F (tham số_{opt}); khai báo định danh F là 1 hàm thì *trong điều kiện cú pháp hợp lệ*:

- $T (*PF)$ (tham số_{opt}); - Khai báo PF là 1 con trỏ hàm có kiểu con trỏ tới kiểu của hàm F .
 - *Dấu * và định danh của con trỏ hàm cần được đặt trong ()*
- **typedef** $T (*PFt)$ (tham số_{op}) - PFt là kiểu con trỏ tới kiểu của hàm F .
- **typedef** $T Ft$ (tham số_{opt}); - Ft là kiểu của hàm F .
- $PFt PF1$; - $PF1$ là 1 con trỏ hàm tương tự PF
- $Ft *PF2$; - $PF2$ cũng là 1 con trỏ hàm tương tự $PF1$ và PF

Ví dụ 7.1. Con trỏ hàm

- `int f(float x);` // `f` là 1 hàm nhận 1 tham số float và trả về int
- `int *f1(float x);` // `f1` là 1 hàm nhận 1 tham số float và trả về int *
- `int (*pf)(float x);` // `pf` là 1 con trỏ tới hàm nhận 1 tham số float và trả về int.
- `typedef int (*pft)(float x);` // `pft` là kiểu con trỏ hàm nhận 1 tham số float và trả về int - kiểu của `pf`.
- `pf = &f;` // `pf` trỏ tới `f`
- `pft pf1 = f;` // `pf1` cũng là con trỏ hàm và trỏ tới `f`, C quy ước nếu `f` là hàm thì biểu thức `f` tương đương với `&f`

Toán tử địa chỉ

- Định dạng:
 - `&x`
- Điều kiện:
 - Toán hạng phải là đối tượng hoặc hàm *hoặc thực thể tương đương*
 - Ngoài đối tượng và hàm, toán hạng có thể là phân tử mảng, kết quả của biểu thức chỉ số, v.v..
- Ý nghĩa:
 - Nếu toán hạng có kiểu **T** thì kết quả có kiểu con trỏ tới **T** và cũng là con trỏ tới toán hạng.
- Ví dụ:
 - `int x;`
 - `int f() { ... }`
 - `&x` Cho kết quả là con trỏ tới biến **x**.
 - `&f` giống như `f`, cùng cho kết quả là con trỏ tới hàm **f**.

Toán tử truy cập

- Định dạng:
 - `*p`
- Điều kiện:
 - Toán hạng phải là con trỏ
- Ý nghĩa:
 - Nếu toán hạng có kiểu con trỏ tới T thì kết quả có kiểu T và cũng là *thực thể* đang được trỏ tới bởi toán hạng.
 - Nếu toán hạng không trỏ vào thực thể thì kết quả là bất định.
- Ví dụ:
 - `int x;`
 - `int *p = &x;`
 - `*p = 100;` // thao tác với `*p` cũng như đang thao tác với `x`
 - `int *p1 = 0, *p2;`
 - `*p1 = 100;` // Hành vi bất định (truy cập con trỏ NULL).
 - `*p2 = 200;` // Hành vi bất định (do `p2` chưa được khởi tạo).

Sơ lược quy tắc ép kiểu con trỏ

- Có thể ép kiểu con trỏ tới kiểu dữ liệu bất kỳ thành con trỏ tới **void**, rồi sau đó nếu ép kiểu con trỏ tới **void** thu được thành kiểu ban đầu, 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 main() {  
    int x = 100;  
    print_i(&x);    // mặc định (void*)&x  
}
```

*Quy chuẩn C không cho phép ép kiểu con trỏ hàm thành con trỏ void **

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

- 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, tuy nhiên không có quy định chung về độ rộng biểu diễn.
 - Ví dụ:

```
int *p = &x;  
int y = (int)p;  
p = (int*)y; // phụ thuộc triển khai, có thể  
mất dữ liệu - p có thể != &x.
```
- Các triển khai tự định nghĩa **intptr_t** và **uintptr_t** là các kiểu số nguyên đủ rộng để lưu con trỏ tới đối tượng
 - Có thể ép kiểu con trỏ **void *** thành **intptr_t** hoặc **uintptr_t** và ngược lại, giá trị được bảo toàn sau chuỗi ép kiểu.

Sơ lược quy tắc ép kiểu với kiểu con trỏ⁽³⁾

- Khi ép kiểu con trỏ *đối tượng* thành con trỏ đối tượng khác kiểu, nếu *cấu trúc bộ nhớ/căn chỉnh hợp lệ* thì thông tin được bảo toàn, nếu ngược lại thì hành vi là bất định.
 - Ví dụ 1: Trong ví dụ này `pi == a`
`int a[100]; // Mảng 100 phần tử int, sẽ học sau`
`char *pc = a;`
`int *pi = pc;`
 - Ví dụ 2: Trong ví dụ này `pc` có thể `!= s` bởi vì địa chỉ của `c` có thể không đáp ứng được yêu cầu căn chỉnh của `int *`
`char c = 'C';`
`int *pi = &c;`
`char *pc = pi;`
- Khi ép kiểu con trỏ đối tượng thành con trỏ `char *` thì kết quả trở tới byte có địa chỉ thấp nhất của đối tượng. Các bước tăng 1 sau đó di chuyển con trỏ qua các bytes còn lại của đối tượng.

Sơ lược quy tắc ép kiểu với kiểu con trỏ₍₄₎

- 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:


```
int sum(int a, int b) { return a + b; }  
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** và được định nghĩa trong **stddef.h**

```
#define NULL ((void*)0) )
```

Từ khóa **void**

- **void** là 1 kiểu dữ liệu đặc biệt - là 1 tập rỗng, không bao gồm bất kỳ giá trị nào - nó là 1 kiểu chưa hoàn thiện và không thể hoàn thiện.
 - Không thể tạo biến kiểu **void** nhưng có thể tạo con trỏ tới **void**. Con trỏ **void*** thường được sử dụng để trung chuyển các con trỏ đối tượng (lược bỏ thông tin định kiểu).
 - Ép kiểu biểu thức thành **void** có nghĩa là bỏ qua giá trị biểu thức, tuy nhiên biểu thức vẫn được tính để thực hiện các hiệu ứng.
- **void** như danh sách tham số của hàm có ý nghĩa khẳng định hàm không có tham số.
- **void** như kiểu trả về của hàm có ý nghĩa khẳng định hàm không trả về giá trị.

Nội dung

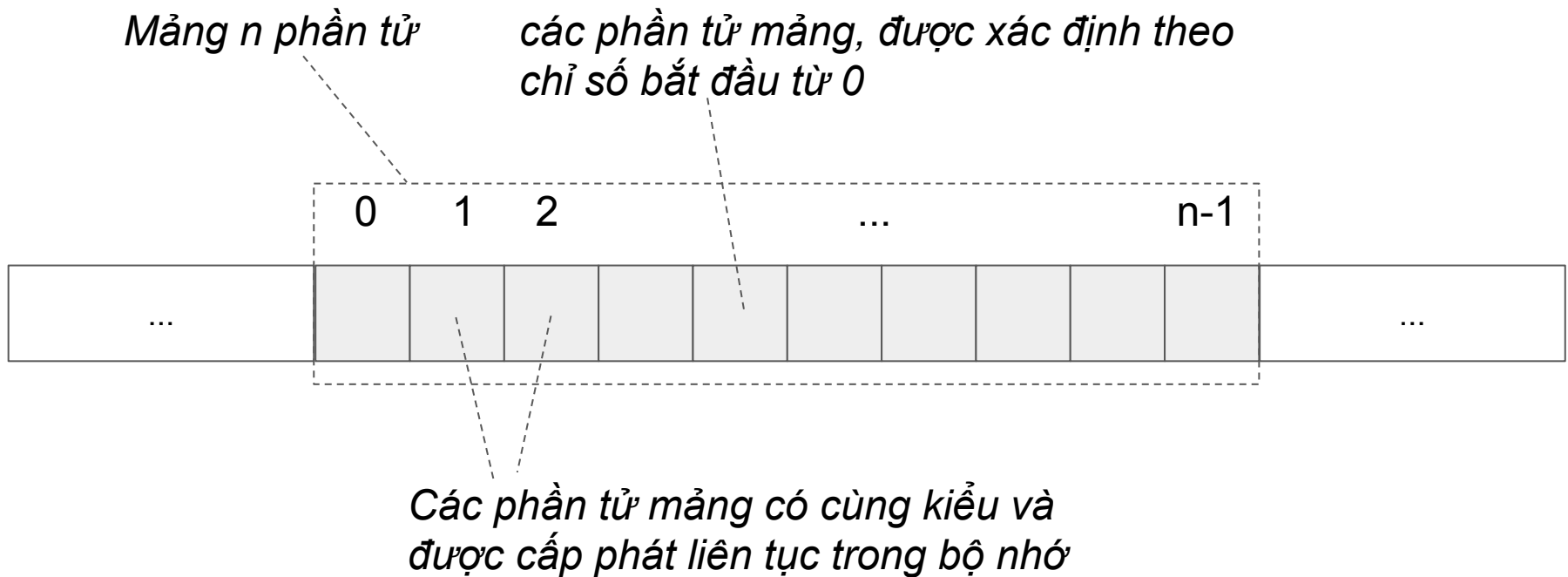
- Kiểu con trỏ và biến con trỏ
 - Kiểu mảng và biến mảng
 - Các phép toán với con trỏ
 - Hàm, mảng và con trỏ
- 

Các khái niệm

- Kiểu mảng mô tả 1 dãy đối tượng cùng kiểu được cấp phát liên tục trong bộ nhớ, trong đó kiểu chung của các đối tượng được gọi là kiểu phần tử của mảng. Kiểu mảng được coi là kiểu suy diễn từ kiểu phần tử của nó:
 - Nếu kiểu phần tử là **T** thì kiểu mảng tương ứng thường được gọi là mảng (của) **T**.
 - **T** phải là kiểu đối tượng hoàn thiện - Không suy diễn được kiểu mảng từ kiểu hàm, hoặc kiểu chưa hoàn thiện.
- Đối tượng có kiểu mảng của T cũng thường được gọi đơn giản là mảng của T tuy có thể gây nhập nhằng về nghĩa.
 - Các đối tượng trong mảng được gọi là phần tử mảng
 - Có thể truy cập từng phần tử mảng bằng chỉ số tương ứng - là số nguyên, bắt đầu từ 0.

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

Biểu diễn trực quan của mảng



Sơ lược cú pháp khai báo 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 các biểu thức hằng và có giá trị là các số nguyên dương.
 - 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 biểu thức chứa các biến có giá trị được xác định trong thời gian thực hiện chương trình.
- Mảng với độ dài thay đổi/Variable Length Array (VLA)
 - Tính năng có từ C99, có 1 số quy tắc riêng cho các trường hợp, ví dụ không được khởi tạo trong khai báo, v.v..
 - *(có thể được thay thế bởi mảng cấp phát động.)*
 - => *Tham khảo thêm, không phân tích chi tiết trong bài này*

Sơ lược mảng 1 chiều kích thước cố định

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

- Nếu T D; khai báo định danh D là 1 đối tượng kiểu T thì T
 $A[N]$; khai báo định danh A là 1 mảng N phần tử kiểu T .
 - $[N]$ được thêm vào sau định danh.
 - N phải là 1 biểu thức hằng có giá trị là số nguyên dương
 - Ví dụ 100, sizeof long, v.v..
- `int a[10];` // a là mảng 10 phần tử kiểu int.
- `double b[10];` // b là mảng 10 phần tử kiểu double.
- `int *p1;` // p1 là con trỏ tới int
- `int *a1[10];` // a1 là mảng 10 con trỏ tới int
- `double *b1[10];` // b1 là mảng 10 con trỏ tới double

Sơ lược mảng 1 chiều kích thước cố định₍₂₎

- 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.

Sơ lược mảng 1 chiều kích thước ngầm định

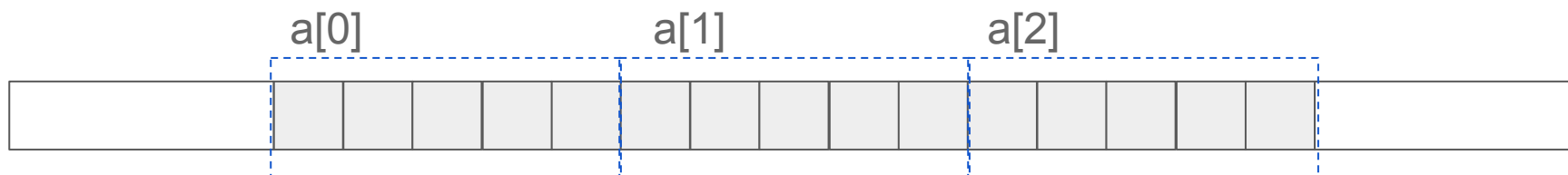
- Kích thước ngầm định được xác định dựa trên biểu thức khởi tạo = kích thước tối thiểu để có thể lưu hết các giá trị khởi tạo.
- `int a[] = {1, 2, 3};` // Như `int a[3] = {1, 2, 3};` - Nếu khởi tạo tuần tự thì kích thước = số lượng giá trị khởi tạo.
- `int b[] = {1, 2, [9] = 90};` // Như `int b[10] = {1, 2, [9] = 90};`
- `int b[] = {1, [9] = 90, 2};` // Như `int b[11] = {1, [9] = 90, 2};`
Nếu sử dụng chỉ số thì số lượng phần tử có thể khác số lượng giá trị khởi tạo và \geq chỉ số lớn nhất + 1.

Toán tử `sizeof` và mảng

- 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.
- Ví dụ:
 - `int a[10];`
 - `sizeof(a) == 10 * sizeof(int)`
 - `sizeof(a) / sizeof(a[0]) == 10`
 - `int b[] = {1, 3, 5};`
 - `sizeof(b) == 3 * sizeof(int)`
 - `sizeof(b) / sizeof(b[0]) == 3`

Sơ lược về 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.
- Ví dụ mảng 2 chiều:
 - `int a[3][5];` // Kích thước các chiều là 3 x 5
 - Có thể coi a như mảng của 3 mảng 5 phần tử `int`.



- (Tương tự với mảng có số chiều > 2).

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.

Sơ lược mảng nhiều chiều kích thước cố định

- Khai báo lần lượt kích thước các chiều, ví dụ:
 - `int a[10][20];` // Mảng 2 chiều của $10 * 20$ biến `int`
 - `int b[10][20][30];` // Mảng 3 chiều của $10 * 20 * 30$ biến `int`
- Khởi tạo:

- Có thể khởi tạo tuần tự các phần tử với 1 danh sách:

`int a[2][3] = {1, 2, 3, 5, 6, 7};` // `a[0][0] = 1, a[0][1] = 2, ...`



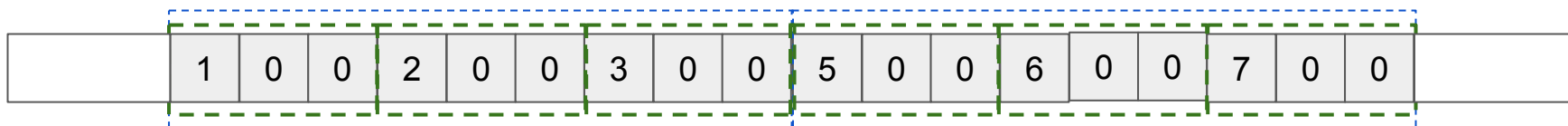
- Hoặc lựa chọn theo chỉ số:

`int a[2][3] = {[1][1] = 100};`

- Cũng có thể phân rã các chiều 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}}};` // Tương tự mảng 1 chiều - phần còn lại được mặc định = 0.



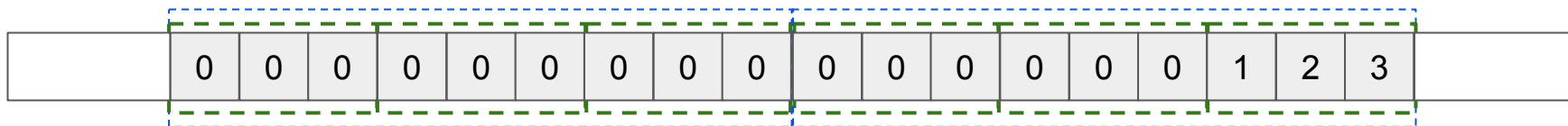
Sơ lược mảng nhiều chiều kích thước ngầm định

Có thể ngầm định chiều trái nhất của mảng nhiều chiều - mảng n chiều ($n > 1$) có thể được coi như mảng 1 chiều của các mảng $n - 1$ chiều.

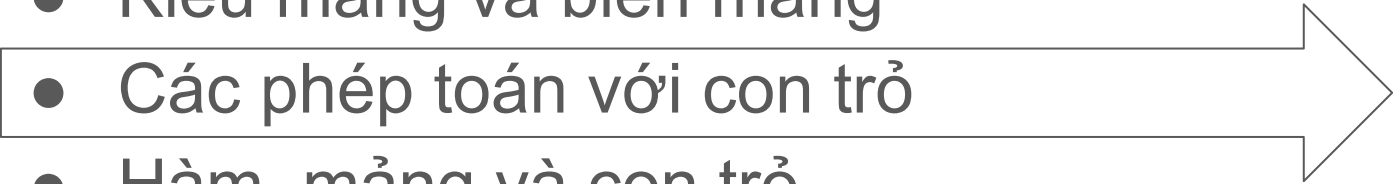
- Khởi tạo tuần tự với danh sách khởi tạo 1 chiều:
 - `int a[][3] = {1, 2, 3, 5, 6, 7}; // a[2][3]; - x * 3 >= 6`
- Phân rã các chiều với các danh sách khởi tạo lồng nhau:
 - `int a[][3] = {{1}, {5}}; // => a[2][3]`



- Lựa chọn theo chỉ số:
 - `int a[][3] = {[5] = {1, 2, 3}}; // => a[6][3]`



Nội dung

- Kiểu con trỏ và biến con trỏ
 - Kiểu mảng và biến 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 , 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 được coi như = n.
- Chúng ta cũng có thể sử dụng các toán tử +=, -=, và các toán tử tăng, giảm 1 với toán hạng con trỏ, ví dụ, với P là con trỏ:
 - P += n; // P = P + n
 - P -= n; // P = P - n
 - ++P; --P;
 - P++; P--;

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

So sánh thứ tự với con trỏ

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

- Kết quả so sánh con trỏ phụ thuộc vào vị trí tương đối trong bộ nhớ của các đối tượng được trỏ tới .
- Nếu các đối tượng được trỏ tới đều là các trường của 1 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 tương ứng của các phần tử mảng .
- 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

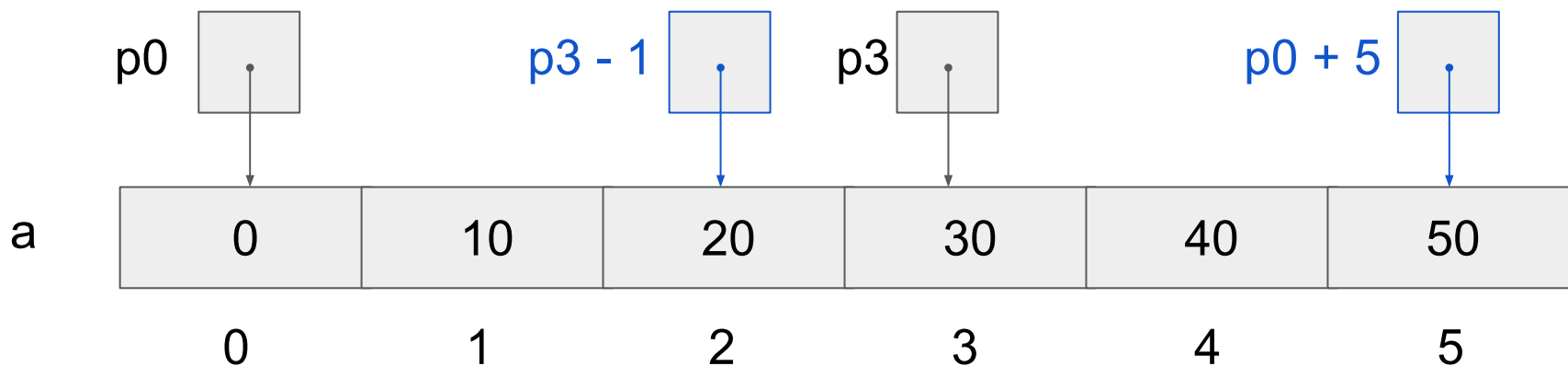
Các toán tử (==, !=)

- 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 2 con trỏ bằng nhau.
 - Ví dụ: Các mảng thành phần của 1 mảng nhiều chiều; 2 đối tượng được cấp phát liền nhau trong bộ nhớ; v.v..

Ví dụ 7.2. 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]; // Trỏ vào phần tử lô-gic liền sau a[5]
```

```
p > q; // Đúng
```

```
q += 3; // q == &a[6]
```

```
p == q; // Đúng
```


Đối tượng chỉ đọc và con trỏ chỉ đọc

*Có thể giới hạn khả năng thay đổi giá trị của con trỏ giống như thay đổi giá trị của các đối tượng khác với từ khóa **const**.*

- Con trỏ chỉ đọc trỏ tới biến có thể thay đổi giá trị:
 - `int x, y;`
 - `int *p;` // p là con trỏ tới int
 - `int * const cp = &x;` // cp là con trỏ chỉ đọc và đang trỏ vào x
 - `*p = 100;` // OK, x = 100
 - `cp = &y;` // Lỗi thay đổi con trỏ chỉ đọc
- Con trỏ có thể thay đổi giá trị 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

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, định danh mảng của các phần tử kiểu T được ép kiểu thành con trỏ tới T và trỏ vào phần tử đầu tiên của mảng.
 - Nếu a là 1 mảng và i là 1 chỉ số mảng thì $a + i == \&a[i]$.
- Ví dụ:
 - `int x, a[100];`
 - `int *p = a; // p trỏ tới phần tử đầu tiên của mảng, $a == \&a[0]$`
 - `a = NULL; // lỗi biên dịch (thay đổi con trỏ chỉ đọc).`
 - `p + 10 == a + 10;`
 - `*(a + 10); // Phần tử có chỉ số 10 của mảng a`

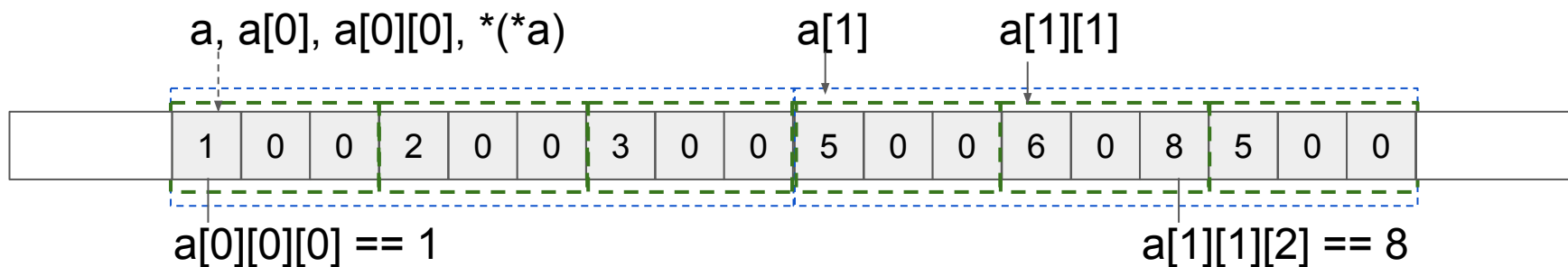
Toán tử chỉ số []

- Định dạng:
 - $E[i]$
- Yêu cầu:
 - E phải là con trỏ đối tượng và i phải có kiểu số nguyên.
- Ý nghĩa:
 - $E[i]$ tương đương với $((E) + (i))$.
 - Nếu E trỏ tới kiểu T thì $E[i]$ cho kết quả là đối tượng kiểu T.
 - Biểu thức dạng $\&E[i]$ được biến đổi thành $((E) + (i))$ (cả $\&$ và $*$ đều được lược bỏ).
- Ví dụ:
 - `int a[100];`
 - $a[0]$ tương đương với $(*(a + 0))$ và $(*a)$, có kiểu int
 - $\&a[0]$ tương đương với $(a + 0)$ và a , có kiểu int $*$
 - a trỏ vào phần tử đầu tiên của mảng, vì vậy chỉ số phải bắt đầu từ 0
 - $a[99]$ và $(*(a + 99))$ là phần tử cuối cùng của a, có kiểu int.

Chỉ số trong mảng nhiều chiều

- Đối với mảng nhiều chiều, ví dụ `int a[2][3][3];`
 - Biểu thức `a` cho kết quả là con trỏ kiểu `int (*)[3][3];`
 - `a[i]` là 1 mảng `3*3` và được chuyển thành con trỏ `int (*)[3];`
 - `a[i][j]` là 1 mảng 3 phần tử và được chuyển thành `int *`.
 - Với 3 chỉ số, `a[i][j][k]` là 1 đối tượng kiểu `int`.

- Ví dụ truy cập phần tử `a[1][1][2]`:



- Do tính chất liên tục của các phần tử mảng:
 - `(int*) a == &a[0][0][0]`
 - *Có thể coi mảng nhiều chiều như mảng 1 chiều với cùng số lượng phần tử và ngược lại.*

Mảng 1 chiều và mảng nhiều chiều

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

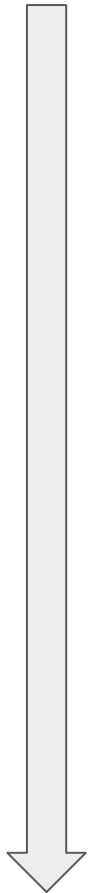
```
int a[12] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

	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 `6 * 2`:
 - `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 `2 * 2 * 3`
 - `c[1][1][1]` tương đương với `*(c + 1)[1][1]` và `== 10`

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	Ví dụ	Chuỗi
Chỉ số Gọi hàm Tăng, giảm 1 (hậu tố)	<code>[]</code> <code>()</code> <code>++, --</code>	<code>a[9]</code> <code>f(3, 5)</code> <code>x++, x--</code>	\Rightarrow
Tăng, giảm 1 (tiền tố) Địa chỉ, truy cập Dấu (tiền tố) Đảo dãy bit, phủ định Dung lượng	<code>++, --</code> <code>&, *</code> <code>+, -</code> <code>~, !</code> <code>sizeof</code>	<code>++x, --x</code> <code>&x, *p</code> <code>+x, -x</code> <code>~x, !x</code> <code>sizeof(int)</code>	\Leftarrow
Ép kiểu	<code>()</code>	<code>(double)n</code>	\Leftarrow
Nhân, Chia, Phần dư	<code>*, /, %</code>	<code>x * y, x / y, x % y</code>	\Rightarrow
Cộng, trừ	<code>+, -</code>	<code>x + y, x - y</code>	\Rightarrow
Dịch trái, dịch phải	<code><<, >></code>	<code>x << y, x >> y</code>	\Rightarrow
So sánh thứ tự	<code><, >, <=, >=</code>	<code>x < y</code>	\Rightarrow
So sánh bằng	<code>==, !=</code>	<code>x == y</code>	\Rightarrow
AND theo bit	<code>&</code>	<code>x & y</code>	\Rightarrow
XOR theo bit	<code>^</code>	<code>x ^ y</code>	\Rightarrow
OR theo bit	<code> </code>	<code>x y</code>	\Rightarrow
AND lô-gic	<code>&&</code>	<code>x && y</code>	\Rightarrow
OR lô-gic	<code> </code>	<code>x y</code>	\Rightarrow
Lựa chọn	<code>?:</code>	<code>x ? y : z</code>	\Leftarrow
Gán đơn giản Gán kết hợp	<code>=</code> <code>*, /=, %=, +=, -=, <=<, >>=, &=, ^=, =</code>	<code>x = y</code> <code>x *= y, x /= y, ...</code>	\Leftarrow

Nội dung

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

Hàm với tham số mảng

Hàm có thể được sử dụng để triển khai xử lý mảng như thao tác bậc cao

- 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]`
- 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 kích thước `m * n` - phức tạp hơn nhưng khái quát.

Ví dụ 7-3a. Tính tổng các phần tử của mảng

Chương trình tính tổng các phần tử của mảng bằng hàm

```
1  #include <stdio.h>
2  long sum(const long n, long *a) {
3      long sum = 0;
4      for (long i = 0; i < n; ++i) {
5          sum += a[i];
6      }
7      return sum;
8  }
9  int main() {
10     long a[] = {1, 3, 5, 8, 9};
11     long n = sizeof(a) / sizeof(a[0]);
12     printf("sum = %ld\n", sum(n, a));
13 }
```

sum = 26

Ví dụ 7-3b. Duyệt mảng 1 chiều bằng con trỏ

Duyệt mảng bằng con trỏ

```
1  #include <stdio.h>
2  long sum(long *beg, long *end) {
3      long sum = 0;
4      for (long *p = beg; p < end; ++p) {
5          sum += *p;
6      }
7      return sum;
8  }
9  int main() {
10     long a[] = {1, 3, 5, 8, 9};
11     long n = sizeof(a) / sizeof(a[0]);
12     printf("sum = %ld\n", sum(a, a + n));
13 }
```

sum = 26

Ví dụ 7-3c. Xử lý mảng nhiều chiều

Biểu diễn mảng 2 chiều như mảng 1 chiều

```
1  #include <stdio.h>
2  long sum(const long n, long *a) {
3      long sum = 0;
4      for (long i = 0; i < n; ++i) {
5          sum += a[i];
6      }
7      return sum;
8  }
9  int main() {
10     long a[2][3] = {1, 3, 5, 8, 9};
11     printf("sum = %ld\n", sum(2 * 3, a[0]));
12 }
```

sum = 26

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:

- Hàm với tham số là VLA 1 chiều:
 - `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
- Hàm với tham số là VLA nhiều chiều:
 - `int f(int m, int n, int a[m][n]);`
 - `int f(const int m, const int n, int a[m][n]);`
 - *(Có thể truyền các mảng nhiều chiều hơn theo cách tương tự)*

Ví dụ 7-4a. Nhân ma trận với con trỏ

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

```
1  #include <stdio.h>
2
3  // c(mxp) = a(mxn) %*% b(nxp)
4  void matrix_mult(int m, int n, int p,
5                  double *a, double *b, double *c) {
6      for (int i = 0; i < m; ++i) {
7          for (int j = 0; j < p; ++j) {
8              double s = 0;
9              for (int t = 0; t < n; ++t) {
10                 s += a[i * n + t] * b[t * p + j];
11             }
12             c[i * p + j] = s;
13         }
14     }
15 }
```

Ví dụ 7-4a. Nhân ma trận với con trỏ⁽²⁾

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

```
16 int main() {
17     double a[3][2] = {{1, 1}, {1, 1}, {1, 1}},
18                   b[2][3] = {{1, 2, 3}, {7, 6, 5}},
19                   c[3][3];
20     matrix_mult(3, 2, 3, a[0], b[0], c[0]);
21     for (int i = 0; i < 3; ++i) {
22         for (int j = 0; j < 3; ++j) {
23             printf("\t%.0f", c[i][j]);
24         }
25         printf("\n");
26     }
27 }
```

8	8	8
8	8	8
8	8	8

Ví dụ 7-4b. Nhân ma trận với VLA

```
1  #include <stdio.h>
2
3  // c(mxp) = a(mxn) %*% b(nxp)
4  void matrix_mult(int m, int n, int p,
5                  double a[m][n], double b[n][p],
6                  double c[m][p]) {
7      for (int i = 0; i < m; ++i) {
8          for (int j = 0; j < p; ++j) {
9              double s = 0;
10             for (int t = 0; t < n; ++t) {
11                 s += a[i][t] * b[t][j];
12             }
13             c[i][j] = s;
14         }
15     }
16 }
```

Đễ đọc hơn?

Ví dụ 7-4b. Nhân ma trận với VLA

```
17 int main() {
18     double a[3][2] = {{1, 1}, {1, 1}, {1, 1}},
19         b[2][3] = {{1, 2, 3}, {7, 6, 5}},
20         c[3][3];
21     matrix_mult(3, 2, 3, a, b, c);
22     for (int i = 0; i < 3; ++i) {
23         for (int j = 0; j < 3; ++j) {
24             printf("\t%.0f", c[i][j]);
25         }
26         printf("\n");
27     }
28 }
```

8	8	8
8	8	8
8	8	8

Ví dụ 7.5. Mở rộng hàm với con trỏ hàm

```
1  #include <stdio.h>
2  void process(const int n, int *a,
3              void (*op) (int*)) {
4      for (int i = 0; i < n; ++i) {
5          op(a + i);
6      }
7  }
8  void add1(int *p) { ++(*p); }
9  void mul3(int *p) { (*p) *= 3; }
10 void print_i(int *p) { printf("%d", *p); };
11 void print_a(const int n, int *a) {
12     process(n, a, print_i);
13     printf("\n");
14 }
```

Ví dụ 7.5. Mở rộng hàm với con trỏ hàm⁽²⁾

```
15 int main() {
16     int a[5] = {1, 2, 3, 4, 5};
17     print_a(5, a);
18     process(5, a, add1);
19     print_a(5, a);
20     process(5, a, mul3);
21     print_a(5, a);
22     int b[2][3] = {1, 2, 3, 5, 6, 7};
23     process(2 * 3, b[0], mul3);
24     for (int i = 0; i < 2; ++i) {
25         print_a(3, b[i]);
26     }
27 }
```

Cộng 1 vào mỗi phần tử mảng

Nhân 3 với mỗi phần tử mảng

1	2	3	4	5
2	3	4	5	6
6	9	12	15	18
3	6	9		
15	18	21		

Ví dụ 7.6. Sắp xếp mảng

Viết chương trình:

Nhập 1 số nguyên dương n (<100), sau đó tiếp tục nhập n số nguyên.

Cuối cùng in ra các số nguyên đã nhập theo thứ tự tăng dần trên 1 dòng, mỗi số cách nhau 1 dấu cách.

Ví dụ 7.6. Sắp xếp mảng₍₂₎

```
1  #include <stdio.h>
2
3  // Sắp xếp chọn
4  void selsort(int n, int *a) {
5      for (int i = 0; i < n - 1; ++i) {
6          for (int j = i + 1; j < n; ++j) {
7              if (a[i] > a[j]) {
8                  int tmp = a[i];
9                  a[i] = a[j];
10                 a[j] = tmp;
11             }
12         }
13     }
14 }
```

Ví dụ 7.6. Sắp xếp mảng₍₃₎

```
15 int main() {
16     int n, a[100];
17     scanf("%d", &n);
18     for (int i = 0; i < n; ++i) {
19         scanf("%d", &a[i]);
20     }
21     selsort(n, a);
22     for (int i = 0; i < n; ++i) {
23         printf("%d ", a[i]);
24     }
25     printf("\n");
26 }
```

5				
3	1	2	9	8
1	2	3	8	9

Bài tập 7.1. Nhập/xuất mảng

Viết chương trình:

- + Nhập 1 số nguyên dương n (<100) sau đó tiếp tục nhập n số nguyên.
- + Xuất ra màn hình các số nguyên theo thứ tự ngược lại.
- + Sau đó nhập 1 số nguyên x - giá trị ngưỡng, rồi xuất ra màn hình tất cả các số nguyên trong n số đã nhập có giá trị $< x$ theo thứ tự nhập ban đầu.

Bài tập 7.2. Mảng và hàm toán học

Nhập 1 số nguyên dương n (< 100) sau đó nhập 1 dãy **n số thực** là các giá trị tọa độ của 1 điểm trong không gian n chiều.

Sau đó tính và in ra màn hình khoảng cách đến gốc tọa độ của điểm đã nhập, làm tròn (mặc định) đến 5 chữ số phần thập phân.

Bài tập 7.3. Sắp xếp mảng

Viết chương trình:

Nhập 1 số nguyên dương n (<100), sau đó tiếp tục nhập 1 mảng n số nguyên.

Sau đó sắp xếp tăng dần các phần tử mảng bằng giải thuật sắp xếp chèn.

Cuối cùng in ra các số nguyên đã nhập theo thứ tự tăng dần, mỗi số cách nhau 1 dấu cách.

*Nhắc lại thuật toán sắp xếp chèn
(Đã học trong phần về thuật toán)*

Thuật toán sắp xếp chèn: NNTN

Sắp xếp dãy số a gồm n phần tử theo thứ tự tăng dần

B1. Khởi tạo chỉ số $j = 1$

B2. Nếu $j < n$ thì

B2.1. Gán $key = a[j]$ và $i = j - 1$

B2.2. Nếu $i \geq 0$ và $a[i] > key$ thì

B2.2.1. Gán $a[i + 1] = a[i]$

B2.2.2. Giảm i đi 1 đơn vị

B2.2.3. Lặp lại khối B2.2

B2.3. Gán $a[i + 1] = key$

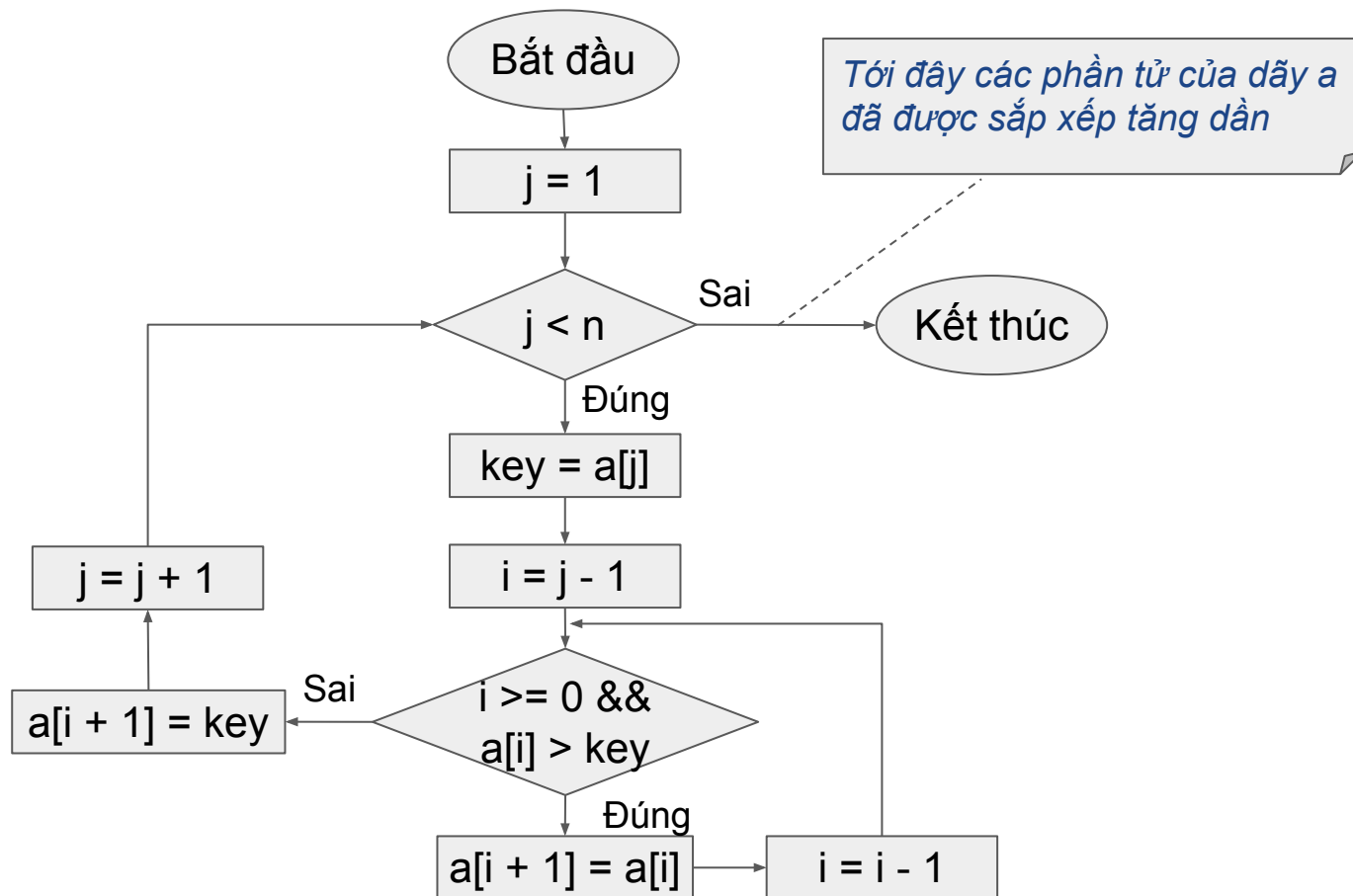
B2.4. Tăng j lên 1 đơn vị

B2.5. Lặp lại khối B2

** Ghi chú: Sau khi khối B2 kết thúc các phần tử của dãy a đã được sắp xếp tăng dần.*

Thuật toán sắp xếp chèn: Lưu đồ

Sắp xếp dãy số a gồm n phần tử theo thứ tự tăng dần



Thuật toán sắp xếp chèn: Mã giả

Sắp xếp dãy số a gồm n phần tử theo thứ tự tăng dần

```
sap_xep_chen(n, a)
```

```
  j = 1
```

```
  while j < n
```

```
    key = a[j]
```

```
    i = j - 1
```

```
    while i >= 0 && a[i] > key
```

```
      a[i + 1] = a[i]
```

```
      i = i - 1
```

```
    a[i + 1] = key
```

```
    j = j + 1
```

```
sap_xep_chen(n, a)
```

```
  for j = 1 to n - 1
```

```
    key = a[j]
```

```
    i = j - 1
```

```
    while i >= 0 && a[i] > key
```

```
      a[i + 1] = a[i]
```

```
      i = i - 1
```

```
    a[i + 1] = key
```

**Ghi chú:*

&& là phép toán lô-gic và/phép hội.

Trích đoạn mã nguồn C

Sắp xếp dãy số a gồm n phần tử theo thứ tự tăng dần

// giả sử a là dãy số nguyên

```
int j = 1;
while (j < n) {
    int key = a[j];
    int i = j - 1;
    while (i >= 0 && a[i] > key) {
        a[i + 1] = a[i];
        i = i - 1;
    }
    a[i + 1] = key;
    j = j + 1;
}
```

Trích đoạn mã nguồn C₍₂₎

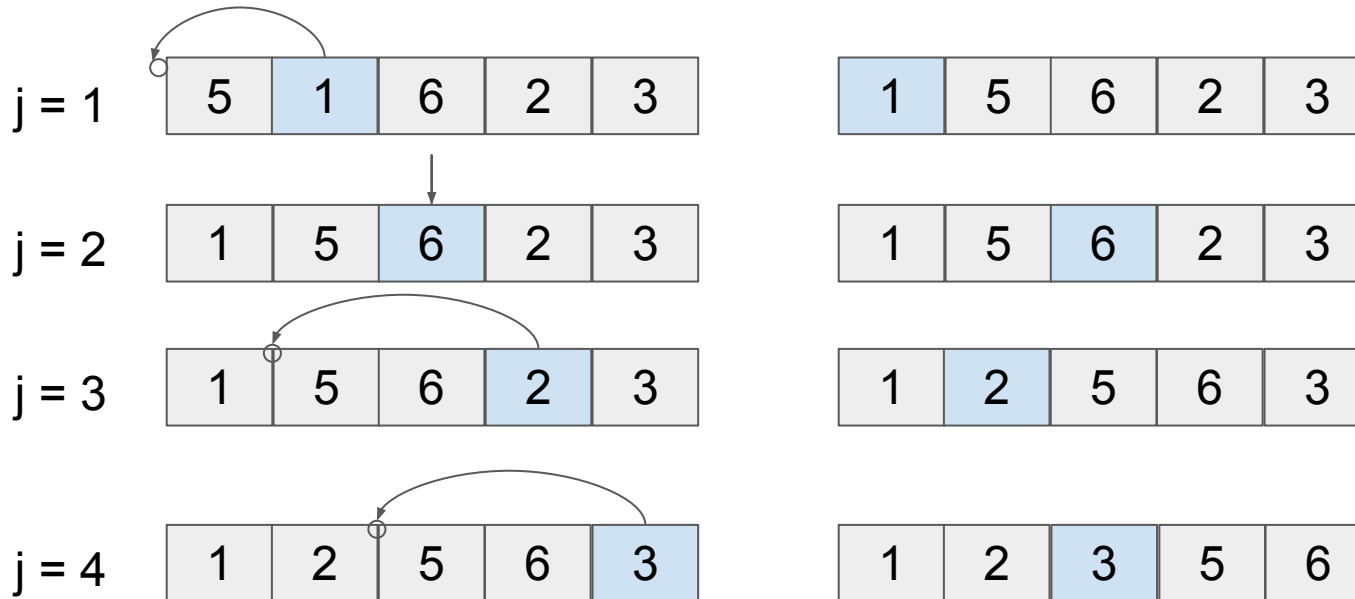
Sắp xếp dãy số a gồm n phần tử theo thứ tự tăng dần

```
// giả sử a là dãy số nguyên
for (int j = 1; j < n; ++j) {
    int key = a[j];
    int i = j - 1;
    for (; i >= 0 && a[i] > key; --i) {
        a[i + 1] = a[i];
    }
    a[i + 1] = key;
}
```

Sắp xếp chèn hoạt động như thế nào?

Xét ví dụ dãy a gồm 5 phần tử:

5	1	6	2	3
---	---	---	---	---



**Ghi chú: Ở mỗi bước lặp với chỉ số j chúng ta có phần $a[0..j - 1]$ của a đã sắp xếp. Vòng lặp bên trong với chỉ số i có tác dụng chèn phần tử $a[j]$ vào vị trí thích hợp trong đoạn $a[0..j]$*

