

# Vec-tơ và vùng nhớ động

Soạn bởi: TS. Nguyễn Bá Ngọc

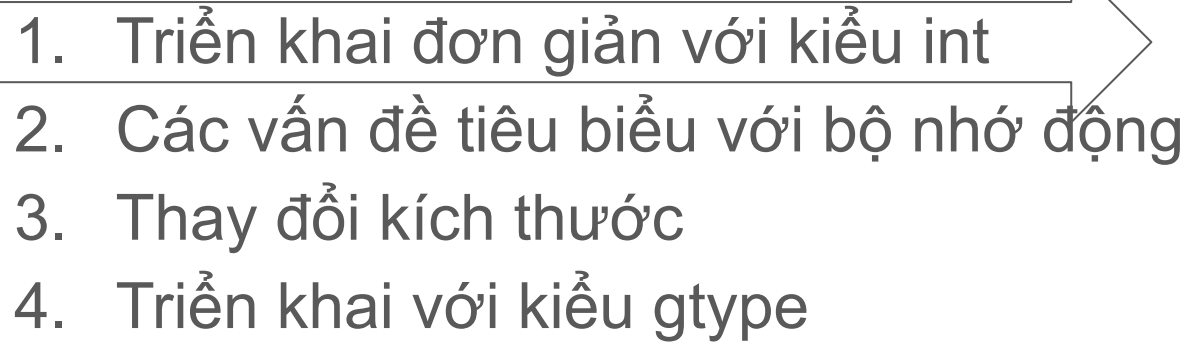
# Tổng quan

- Vec-tơ không chỉ là cấu trúc lưu trữ hữu ích nhất, nó còn là 1 triển khai tiêu biểu dựa trên mảng, có thể minh họa kỹ thuật đóng gói và tạo các giao diện để sử dụng dữ liệu an toàn hơn.
- Trong bài giảng này chúng ta sẽ phân tích một số triển khai vec-tơ từ đơn giản tới phức tạp, từ cụ thể tới khái quát với nhóm kiểu (kiểu gtype), qua đó có thể hiểu hơn vấn đề sử dụng bộ nhớ động và giải pháp.
  - Bài giảng này cũng sẽ đề cập đến bộ nhớ động (free store).
  - *Bộ nhớ động: Khi cần sử dụng 1 khối nhớ người lập trình gọi hàm cấp phát bộ nhớ, khi không cần sử dụng nữa người lập trình gọi hàm giải phóng khối nhớ đã được cấp phát.*
  - Vùng nhớ động còn được gọi là Heap

# Nội dung

1. Triển khai đơn giản với kiểu int
2. Các vấn đề tiêu biểu với bộ nhớ động
3. Thay đổi kích thước
4. Triển khai với kiểu gtype

# Nội dung

- 
1. Triển khai đơn giản với kiểu int
  2. Các vấn đề tiêu biểu với bộ nhớ động
  3. Thay đổi kích thước
  4. Triển khai với kiểu gtype

# Tổng quan về Vec-tơ

- Cấu trúc lưu trữ hữu ích nhất
  - Đơn giản
  - Lưu các phần tử thành khối liên tục
  - Truy cập ngẫu nhiên theo chỉ số
  - Có thể được mở rộng khi đầy
  - Có thể kiểm tra khoảng chỉ số khi truy cập phần tử
- Vec-tơ được triển khai như thế nào?
  - Chúng ta sẽ làm rõ từng bước
- Sử dụng vec-tơ như cấu trúc lưu trữ mặc định
  - Vec-tơ thường được sử dụng để lưu các phần tử trừ khi có lý do thích đáng để không sử dụng vec-tơ.

# Các tầng kiến trúc

- Phần cứng cung cấp bộ nhớ và địa chỉ
  - Bậc thấp
  - Không định kiểu
  - Các khối nhớ kích thước cố định
  - Không kiểm tra điều kiện
  - Tốc độ xử lý có thể đạt tới giới hạn của phần cứng
- Để phát triển chương trình ứng dụng chúng ta cần các cấu trúc bậc cao như vec-tơ
  - Các thao tác bậc cao
  - Kiểm tra kiểu
  - Kích thước động (có thể thay đổi trong thời gian thực thi)
  - Tính năng an toàn: Kiểm tra khoảng chỉ số
  - Tốc độ tiệm cận tốc độ tối đa.

# Các tầng kiến trúc<sub>(2)</sub>

- Lập trình ở tầng thấp, gần với phần cứng, chỉ có thể sử dụng các thao tác thô sơ
  - Người lập trình phải tự xử lý mọi thứ
  - Không có các cơ chế an toàn: Kiểm tra kiểu, kiểm tra khoảng chỉ số, v.v..
  - Các lỗi có thể chỉ được phát hiện khi phát sinh lỗi dữ liệu hoặc chương trình dừng hoạt động đột ngột.
- Chúng ta muốn chuyển lên các tầng cao hơn càng nhanh càng tốt
  - Xử lý vấn đề nhanh và ổn định.
  - Sử dụng ngôn ngữ *thân thiện* với người.

*Người ta thường tin tưởng ở những phép màu khi chưa hiểu*

# Vec-tơ

- Vec-tơ

- Có thể truy cập ngẫu nhiên đến 1 phần tử bất kỳ bằng chỉ số
  - Do các phần tử được lưu trong mảng cấp phát động
- Số lượng phần tử có thể thay đổi sau khi tạo
  - Cấp phát lại bộ nhớ cho mảng khi cần mở rộng
  - Có thể đạt đến giới hạn vật lý của bộ nhớ và hệ điều hành.

*Ví dụ vec-tơ v chứa 5 số nguyên*





# Một triển khai vec-tơ số nguyên

// Một giao diện vec-tơ rất tối giản

```
struct vector {  
    int sz; // Số lượng phần tử  
    int *elems; // con trỏ tới phần tử đầu tiên  
};
```

// Hàm tạo: cấp phát n phần tử

```
struct vector *vec_create(int n);
```

// Hàm hủy: giải phóng bộ nhớ

```
void vec_free(struct vector *v);
```

```
int vec_size(struct vector *v); // Trả về kích thước của vec-tơ
```

// Chúng ta cần lưu số lượng phần tử bởi vì không thể lấy số

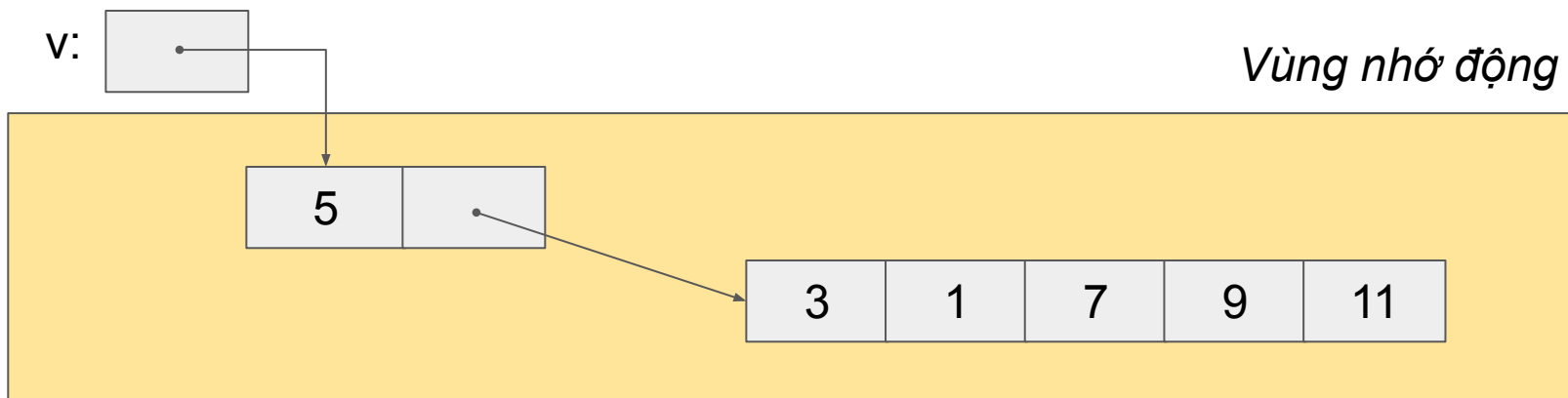
// lượng phần tử từ con trỏ đến khối nhớ.

Cấp phát bộ nhớ cho các phần tử như thế nào?

# Hàm tạo vec-tơ

```
struct vector *vec_create(int n) {  
    struct vector *v = malloc(sizeof(struct vector));  
    v->sz = n;  
    v->elems = malloc(n * sizeof(int));  
    return v;  
}
```

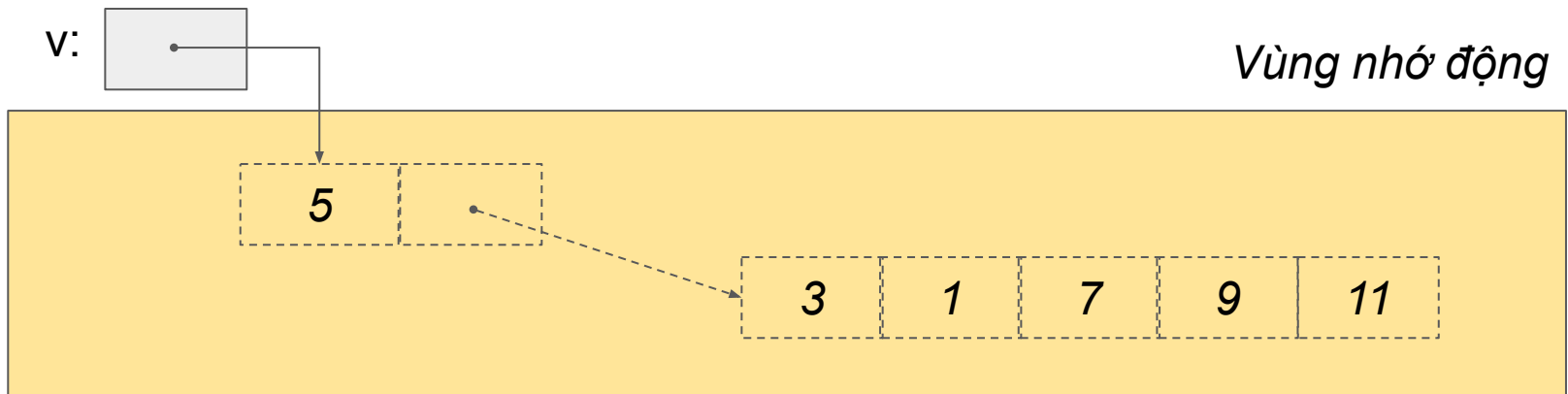
// Hàm malloc cấp phát vùng nhớ trong bộ nhớ động,  
// không khởi tạo giá trị



# Hàm hủy vec-tơ

```
void vec_free(struct vector *v) {  
    free(v->elems);  
    free(v);  
}
```

// Hàm free giải phóng vùng nhớ đã được cấp phát



# Mô hình bộ nhớ máy tính

*Bố cục bộ nhớ:*

Lệnh
Dữ liệu tĩnh
Vùng nhớ động
Ngăn xếp

*(Các thuật ngữ:*

*Lệnh - Code*

*Dữ liệu tĩnh - Static data*

*Bộ nhớ động - Free store*

*Ngăn xếp - Stack)*

- Các thành phần của chương trình
  - Các biến địa phương nằm trong phân vùng ngăn xếp
  - Các biến toàn cục - dữ liệu tĩnh
  - Mã thực thi - lệnh

# Vùng nhớ động

- Khối nhớ trong vùng nhớ động có thể được yêu cầu cấp phát ở thời gian thực thi, ví dụ bằng hàm malloc
  - Hàm malloc trả về con trỏ tới khối nhớ được cấp phát.
  - Khối nhớ được cấp phát động bằng malloc có thể được sử dụng như 1 mảng hoặc 1 đối tượng bất kỳ khác
    - (Mảng cấp phát động.)
  - Các giá trị không được khởi tạo.
- Khối nhớ được cấp phát trong vùng nhớ động vẫn tồn tại sau khi hoàn thành thực hiện hàm tạo nó.
  - Người lập trình có thể giải phóng khối nhớ đã được cấp phát bằng hàm **free** khi không cần dùng đến nữa.
  - Trường hợp có các khối nhớ bị *lãng quên*, không được giải phóng, gây lãng phí bộ nhớ được gọi là thất thoát bộ nhớ (*memory leak*).

# Con trỏ, mảng và vec-tơ

- Con trỏ và mảng là các công cụ truy cập và sử dụng bộ nhớ ở bậc thấp, nếu được sử dụng không đúng cách có thể gây ra những lỗi nghiêm trọng
  - Cần thận trọng với các thao tác xử lý bậc thấp, và chỉ nên viết các xử lý bậc thấp nếu thực sự cần thiết.
  - Sử dụng con trỏ không hợp lệ hoặc chưa được khởi tạo có thể gây ra các lỗi điển hình như: "Segmentation fault (core dumped)"
- Cấu trúc vec-tơ cho phép phát huy các điểm mạnh về hiệu năng của mảng trong điều kiện có nhiều hỗ trợ về mặt ngôn ngữ hơn: Ít lỗi hơn và rút ngắn thời gian tìm lỗi.


# Vec-tơ và một số hàm xử lý đơn giản

```
int vec_get(struct vector *v, int idx) { // Đọc
    return v->elems[idx];
}

void vec_set(struct vector *v, int idx, int value) { // Cập nhật
    v->elems[idx] = value;
}

int vec_size(struct vector *v) { // Kích thước hiện tại
    return v->sz;
}
```

# Nội dung

1. Triển khai đơn giản với kiểu int
  2. Các vấn đề tiêu biểu với bộ nhớ động
  3. Thay đổi kích thước
  4. Triển khai với kiểu gtype
- 



# Vấn đề thất thoát bộ nhớ

```
int *process(struct vector *v) {  
    int *tmp = malloc(vec_size(v) * sizeof(int));  
    int *result = malloc(2 * vec_size(v) * sizeof(int));  
    /* Thực hiện các tính toán ... */  
    return result;  
} /*...*/
```

int \*r = process(v); // oops! Chúng ta quên giải phóng bộ nhớ  
// được trả tới bởi tmp trong process

- Thất thoát bộ nhớ có thể gây ra các vấn đề nghiêm trọng trong thực tế.
- Các chương trình hoạt động liên tục trong thời gian dài không được phép để thất thoát bộ nhớ.

# Vấn đề thất thoát bộ nhớ<sub>(2)</sub>

```
int *process(struct vector *v) {  
    int *tmp = malloc(vec_size(v) * sizeof(int));  
    int *result = malloc(2 * vec_size(v) * sizeof(int));  
    /* Thực hiện các tính toán ... */  
    free(tmp);  
    return result;  
} /*...*/  
int *r = process(v);  
/* Sử dụng r */  
free(r); // Khá dễ quên
```

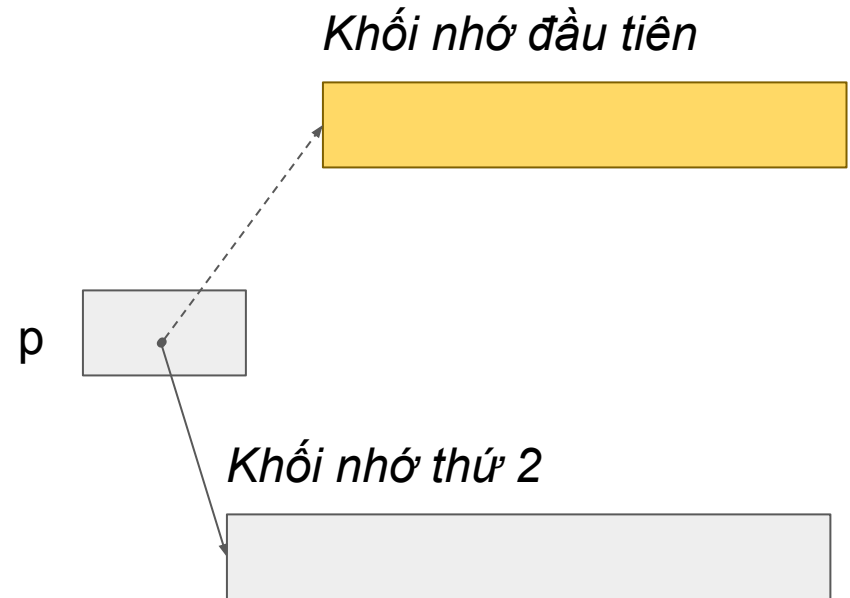
# Vấn đề thất thoát bộ nhớ<sub>(3)</sub>

- Chương trình **hoạt động liên tục trong thời gian dài** không được để thất thoát bộ nhớ
  - Nếu 1 hàm để thất thoát 8 bytes mỗi lần được gọi, thì sau bao nhiêu lần gọi nó sẽ làm thất thoát 1 megabyte?
- Tất cả bộ nhớ được trả cho hệ thống sau khi chương trình kết thúc
  - Nếu chương trình được chạy trong môi trường hệ điều hành (Linux, Unix, Windows, v.v..)
  - Chương trình sử dụng 1 lượng bộ nhớ có kiểm soát đến khi kết thúc có thể có **thất thoát bộ nhớ khi kết thúc** mà không gây ra vấn đề nghiêm trọng.
- => Thất thoát bộ nhớ có thể dẫn đến các vấn đề nghiêm trọng trong 1 số điều kiện nhất định.

# Vấn đề thất thoát bộ nhớ<sub>(4)</sub>

- Một kịch bản gây thất thoát bộ nhớ khác

```
void f() {  
    int *p = malloc(10 * sizeof(int));  
    /* ... */  
    p = malloc(20 * sizeof(int));  
    /* ... */  
    free(p);  
}
```



// Thất thoát (khối nhớ) mảng 10 phần tử int đầu tiên

# Vấn đề thất thoát bộ nhớ<sub>(5)</sub>

- Làm cách nào để loại bỏ thất thoát bộ nhớ một cách có hệ thống
  - Tránh sử dụng các xử lý bậc thấp (malloc, free) trong chương trình ứng dụng
  - Sử dụng các cấu trúc bậc cao: vec-tơ, v.v..
  - Hoặc sử dụng các nền tảng quản lý bộ nhớ tự động (garbage collector)
    - Giám sát tất cả các cấp phát động
    - Hoàn trả các vùng nhớ khi không sử dụng đến nữa.

# Vấn đề với phép gán

- Phép gán có thể không hoạt động đúng như mong đợi

```
void f(int n) {
```

```
    struct vector *v = vec_create(n); // Tạo 1 vec-tơ
```

```
    struct vector v2 = *v; // Kết quả là gì?
```

```
    // Chúng ta mong đợi kết quả gì?
```

```
    struct vector *v3 = vec_create(n); // Tạo thêm 1 vec-tơ
```

```
    *v3 = *v; // Kết quả là gì?
```

```
    // Chúng ta mong đợi kết quả là gì?
```

```
    // ....
```

```
}
```

- Trong C chúng ta không sao chép được nội dung của vec-tơ bằng phép gán, tuy nhiên có thể triển khai thao tác sao chép nội dung bằng hàm.

# Vấn đề với phép gán<sub>(2)</sub>

- Phép gán = có kết quả giống như sao chép vùng nhớ của các biến

```
void f(int n) {
```

```
    struct vector *v1 = vec_create(n);
```

```
    struct vector *v2 = vec_create(n);
```

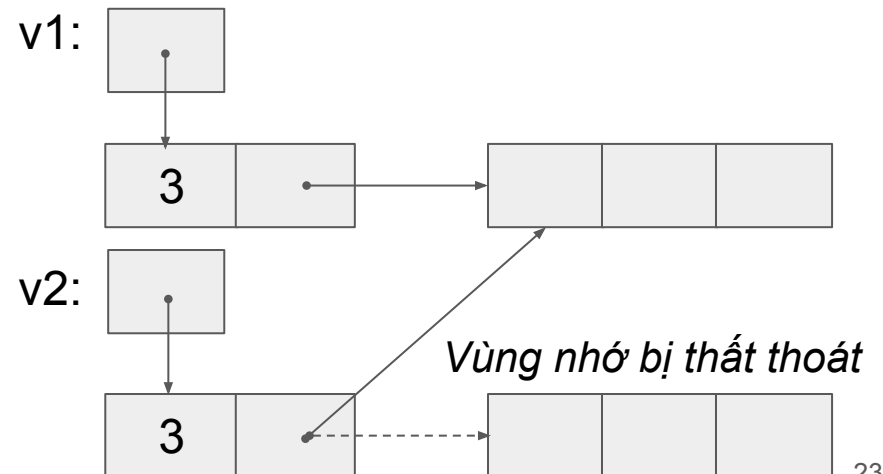
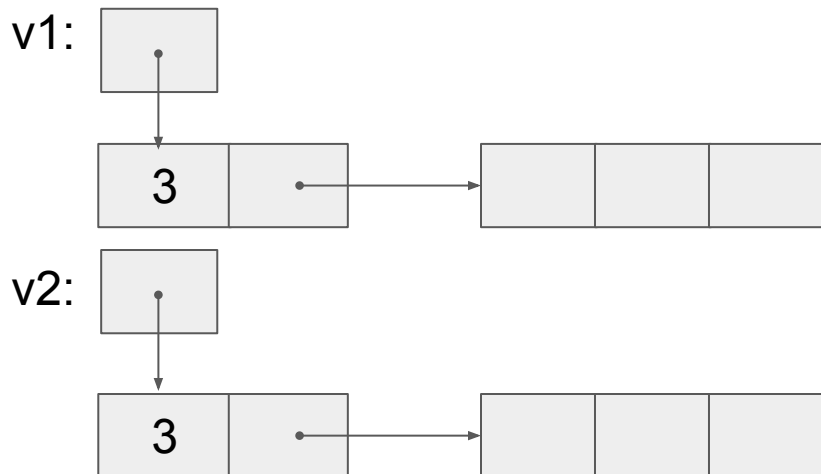
```
    *v2 = *v1;
```

```
    // ...
```

```
    vec_free(v1);
```

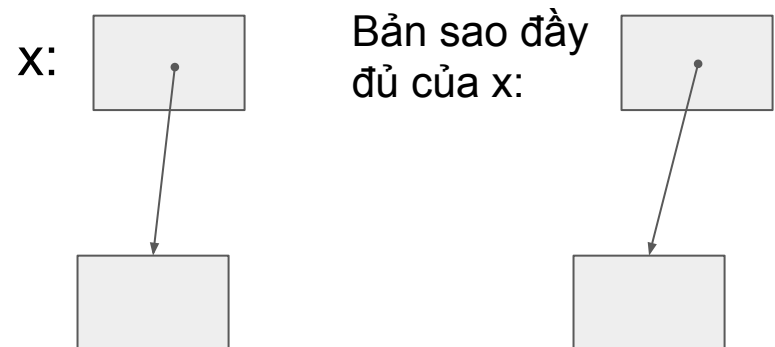
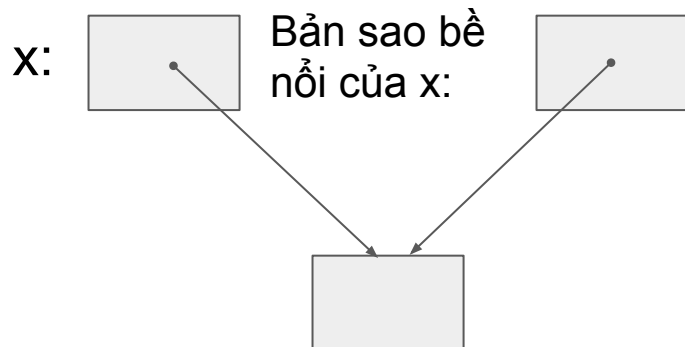
```
    vec_free(v2); // Lỗi giải phóng bộ nhớ 2 lần
```

```
}
```



# Các mức sao chép

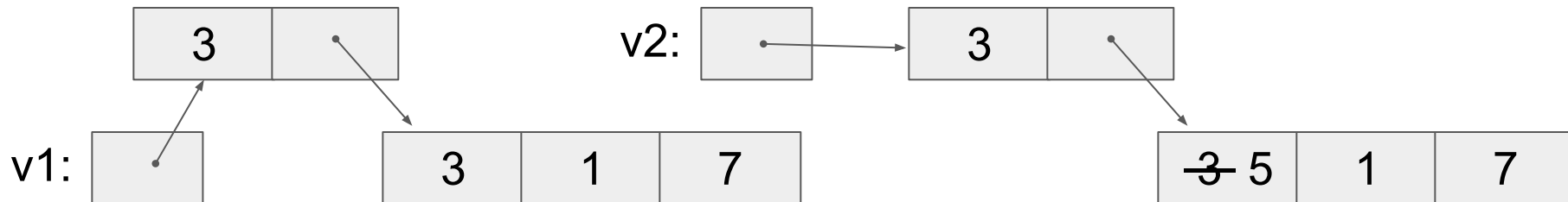
- Sao chép bề nổi: Chỉ sao chép con trỏ, các con trỏ cùng trỏ tới 1 đối tượng
- Sao chép đầy đủ: Sao chép đối tượng được trỏ tới, các con trỏ trỏ tới các đối tượng khác nhau
  - Sao chép tất cả các mức nếu có nhiều mức liên kết trong đối tượng



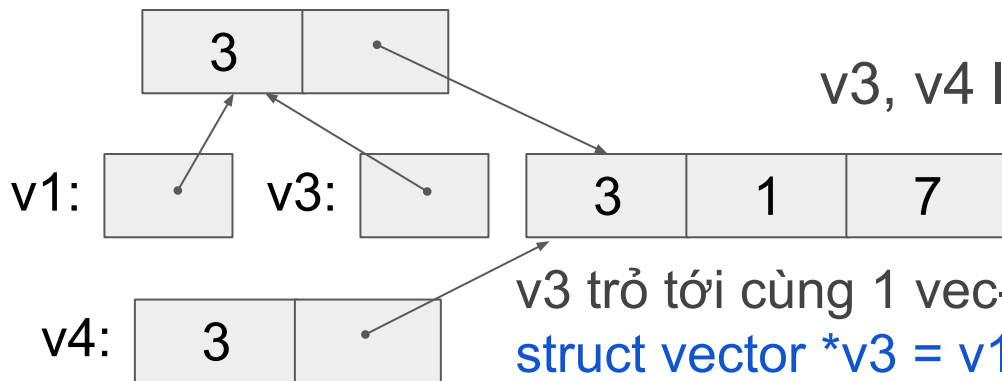
*Phép gán chỉ thực hiện sao chép bề nổi đối với các cấu trúc chứa con trỏ trỏ tới vùng nhớ nằm ngoài vùng nhớ của nó*



# Các mức sao chép<sub>(2)</sub>



v2 là bản sao đầy đủ của v1, có thể được tạo bằng hàm.  
`vec_set(v2, 0, 5);` // Phần tử đầu tiên của v2 == 5, phần tử  
// đầu tiên của v1 không thay đổi



v3, v4 là các bản sao bề nổi của v1.

v3 trỏ tới cùng 1 vec-tơ như v1, ví dụ  
`struct vector *v3 = v1;` // OK: Thường dùng để  
// truyền dữ liệu cho hàm  
v4 chỉ sao chép phần cấu trúc mô tả, ví dụ:  
`struct vector v3 = *v1;` // Không nên truyền dữ liệu  
// theo cách này

# Phép gán với kiểu vec-tơ

- Các trường hợp tiêu biểu

```
struct vector *v1 = vec_create(10);
```

1) `struct vector *v2 = v1; // Ok: Gán con trỏ`

2) `struct vector v3 = *v1; // Có khả năng lỗi, bạn muốn làm gì?`

3) `struct vector *v4 = vec_create(20);`

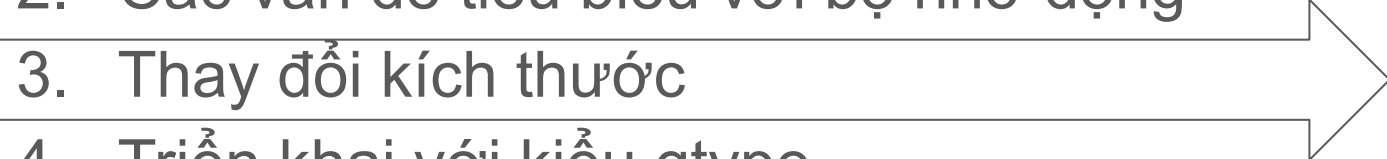
`*v4 = *v1; // Các vấn đề: Chỉ sao chép cấu trúc mô tả như 2 &`

`// thất thoát bộ nhớ đã cấp phát cho vec-tơ v4, bạn muốn làm gì?`

4) `struct vector *v5 = vec_clone(v1); // Ok: Giả sử vec_clone là`

`// hàm tạo bản sao đầy đủ của vec-tơ v1`

# Nội dung

1. Triển khai đơn giản với kiểu int
  2. Các vấn đề tiêu biểu với bộ nhớ động
  3. Thay đổi kích thước
  4. Triển khai với kiểu gtype
- 

# Thay đổi kích thước vec-tơ

- Chúng ta muốn thay đổi kích thước vec-tơ sau khi cấp phát để có được kích thước bộ nhớ phù hợp nhất trong nhiều trường hợp: Không cấp phát thiếu bộ nhớ và cũng không cấp phát quá nhiều.

- Cho:

```
struct vector *v = vec_create(n);
```

- Chúng ta có thể thay đổi kích thước vec-tơ với các hàm:

```
vec_resize(v, 10); // Kích thước mới của v là 10
```

```
vec_append(v, 100); // Thêm phần tử 100 vào sau phần tử cuối cùng đồng thời tăng kích thước vec-tơ lên 1.
```

# Vec-tơ với khả năng thay đổi kích thước

- Để triển khai khả năng thay đổi kích thước chúng ta cập nhật triển khai vec-tơ đơn giản:

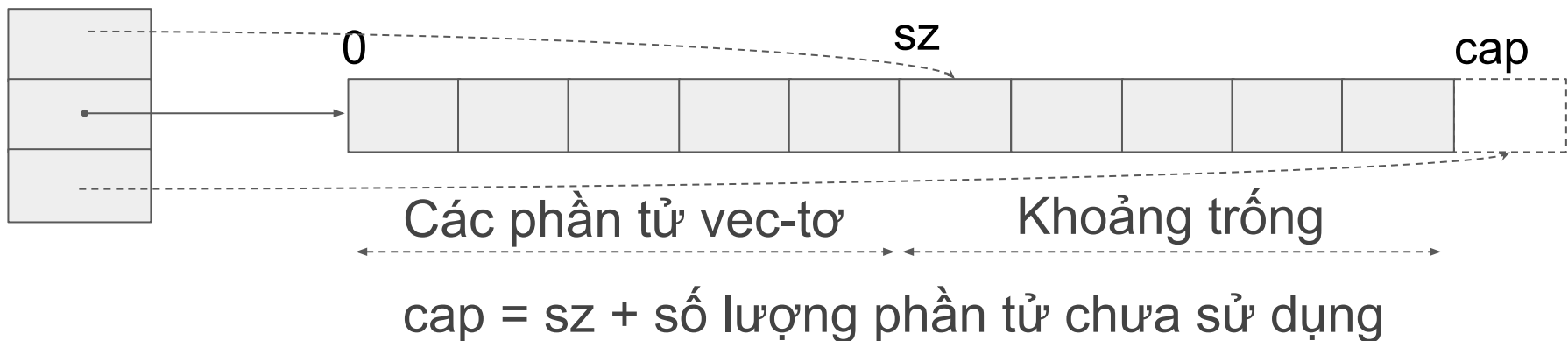
```
struct vector {
```

```
    int sz; // Kích thước: Số lượng phần tử đã có trong vec-tơ
```

```
    int *elems; // con trỏ tới phần tử đầu tiên
```

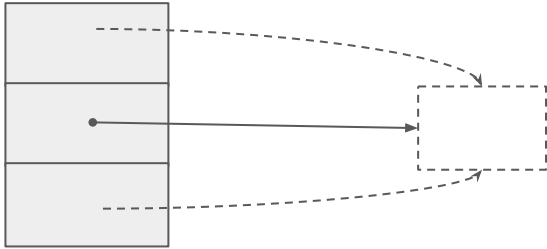
```
    int cap; // Dung lượng: Số lượng phần tử được cấp phát
```

```
};
```

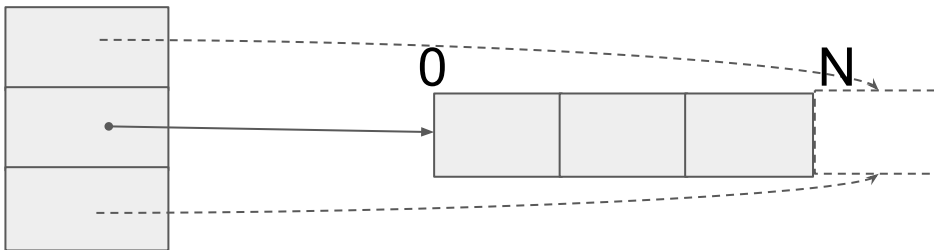


# Biểu diễn vec-tơ

- Vec-tơ rỗng (không được cấp phát bộ cho phần tử)



- Vec-tơ đầy (đã dùng hết bộ nhớ được cấp phát)



# Điều chỉnh dung lượng

- Cấp phát lại vùng nhớ mảng
  - Không thay đổi kích thước hoặc giá trị phần tử đã có

```
void vec_reserve(struct vector *v, int newcap) {  
    if (newcap <= v->size) {  
        return; \\ Không giải phóng các phần tử đã sử dụng  
    }  
    v->elems = realloc(v->elems, newcap * sizeof(int));  
    v->cap = newcap;  
}
```

# Điều chỉnh kích thước

- Thay đổi các phần tử đã có trong vec-tơ và có thể dẫn đến thay đổi dung lượng

```
void vec_resize(struct vector *v, int newsize) {  
    if (newsize > v->cap) {  
        vec_reserve(v, newsize);  
    }  
    v->sz = newsize;  
}
```



# Thêm phần tử vào phía sau

- Thêm 1 phần tử vào sau phần tử cuối đồng thời tăng kích thước lên 1

```
void vec_append(struct vector *v, int value) {  
    if (v->sz == 0) {  
        vec_reserve(v, 16);  
    } else if (v->sz == v->cap) {  
        vec_reserve(v, 2 * v->cap);  
    }  
    v->elems[v->sz] = value;  
    ++v->sz;  
}
```

# Vấn đề thay đổi địa chỉ

- Sau khi cấp phát lại bộ nhớ động địa chỉ vùng nhớ có thể thay đổi.
- Ví dụ tình huống lỗi:

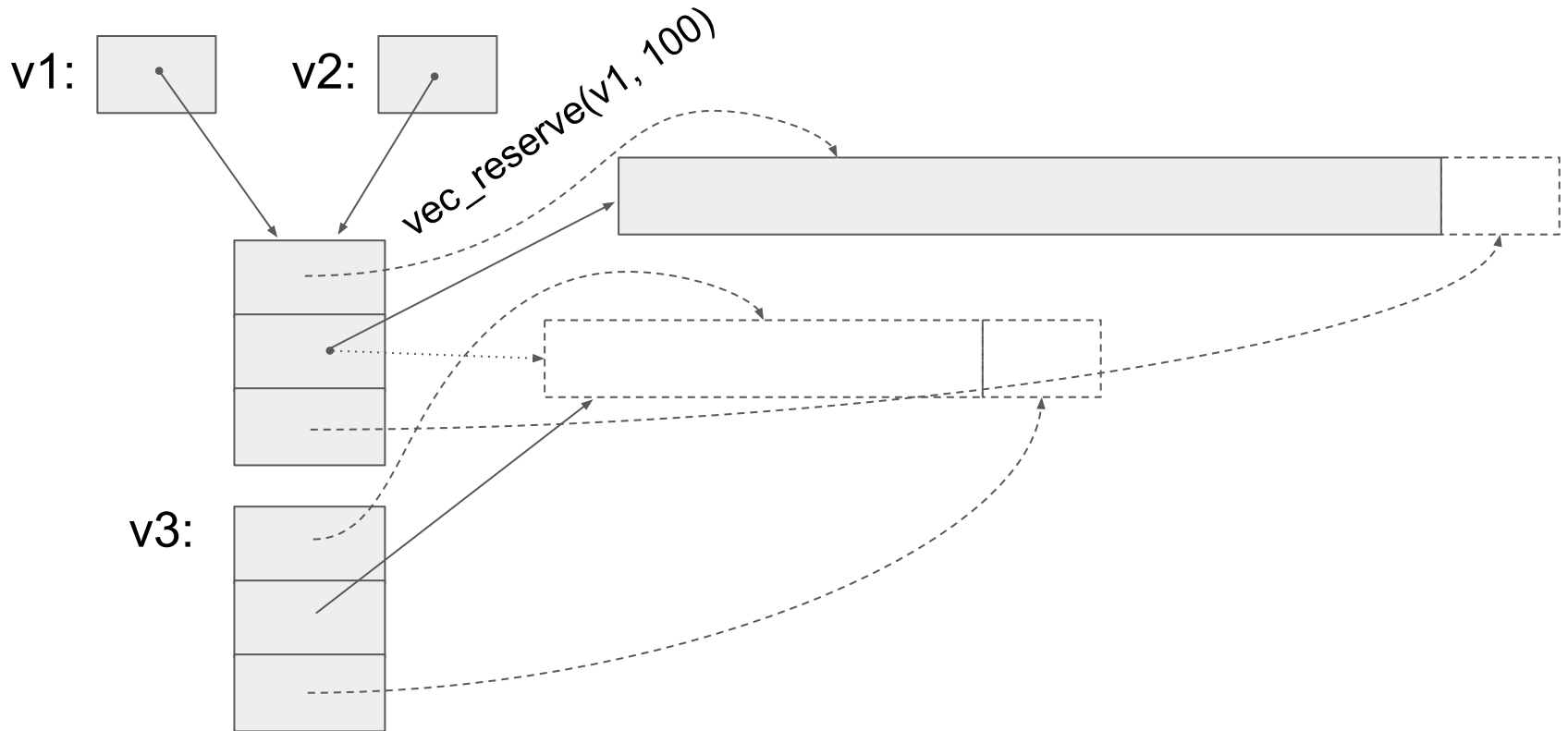
```
void f(int n) {  
    struct vector *v = vec_create(10);  
    struct vector v2 = *v;  
    vec_set(&v2, 1, 111); // OK  
    vec_reserve(v, 100);  
    vec_set(&v2, 1, 222); // NOK: Địa chỉ mảng động có thể đã  
    // thay đổi => Lỗi truy cập vùng nhớ đã bị giải phóng.  
    // ...  
}
```

# Các bản sao bề nổi

- Mảng cấp phát động giữ vai trò cơ sở của vec-tơ có thể thay đổi địa chỉ sau khi thay đổi dung lượng vec-tơ

```
void f(int n) {  
    struct vector *v1 = vec_create(10);  
    struct vector *v2 = v1; // Bản sao bề nổi của v1  
    struct vector v3 = *v1; // Bản sao bề nổi của cấu trúc điều  
    // khiển được trỏ tới bởi v1  
    vec_reserve(v1, 100);  
    // Có thể tiếp tục sử dụng v2.  
    // Không nên tiếp tục sử dụng v3 để truy cập tới các phần tử  
    // ...  
}
```

# Các bản sao bề nổi<sub>(2)</sub>



**Vấn đề:** Sau khi thay đổi dung lượng của v1 kéo theo thay đổi địa chỉ mảng động của vec-tơ, con trỏ v3.elems vẫn trỏ tới địa chỉ cũ.

**Khuyến cáo:** Chỉ nên sử dụng con trỏ tới cấu trúc mô tả vec-tơ (tương tự như v2) để trao đổi dữ liệu.

# Triển khai vec-tơ với gtype

- Thay vì tạo nhiều triển khai riêng cho từng kiểu dữ liệu của phần tử (int, long, double, các kiểu con trỏ, v.v...) chúng ta có thể sử dụng 1 triển khai với gtype cho nhiều trường hợp.
- Phần tử của vec-tơ có thể là con trỏ tới đối tượng được tạo trong vùng nhớ động
- Các xử lý phần tử gtype có thể được thực hiện bên ngoài triển khai vec-tơ.
- ... Tuy nhiên để thực hiện cấp phát và giải phóng bộ nhớ động 1 cách có hệ thống chúng ta có thể tích hợp các thao tác này vào bên trong triển khai vec-tơ.

# Nội dung

1. Triển khai đơn giản với kiểu int
2. Các vấn đề tiêu biểu với bộ nhớ động
3. Thay đổi kích thước
4. Triển khai với kiểu gtype



# Triển khai vec-tơ với gtype<sub>(2)</sub>

- Cấu trúc mô tả:

```
struct gvector {  
    int sz; // Kích thước: Số lượng phần tử đã có trong vec-tơ  
    gtype *elems; // con trỏ tới phần tử đầu tiên  
    int cap; // Dung lượng: Số lượng phần tử được cấp phát  
    gtype_free_t fv; // Con trỏ hàm: Hàm giải phóng bộ nhớ  
    // động được gán với phần tử gtype, = NULL nếu không có.  
};  
  
// Hàm tạo và hàm hủy có khác biệt so với triển khai đơn giản  
// (với kiểu int) do các xử lý liên quan đến con trỏ hàm fv
```

Cấp phát và giải phóng bộ nhớ của vec-tơ như thế nào?

# Hàm tạo vec-tơ với kiểu gtype

```
struct gvector *gvec_create(int n, gtype_free_t fv) {  
    struct gvector *v = malloc(sizeof(struct gvector));  
    v->sz = n;  
    v->cap = n;  
    v->elems = malloc(n * sizeof(gtype));  
    v->fv = fv;  
    return v;  
}
```



# Hàm hủy vec-tơ với kiểu gtype

```
void gvec_free(struct gvector *v) {  
    if (v->fv) { // Giải phóng các vùng nhớ động của các phần tử  
        for (int i = 0; i < v->sz; ++i) {  
            v->fv(v->elems[i]);  
        }  
    }  
    free(v->elems);  
    free(v);  
}
```

Các thao tác đơn giản như **get**, **set**, **size**, **capacity**, **setsize**, **reserve** có thể được triển khai tương tự triển khai với kiểu **int**

# Bài tập 1a. Đọc dãy số nguyên từ tệp

**Yêu cầu:** Viết hàm đọc 1 dãy số nguyên từ 1 tệp văn bản và trả về 1 vec-tơ chứa dãy số đó. Dữ liệu được cho theo định dạng: Đầu tiên là 1 số nguyên  $n$  và tiếp theo là  $n$  số nguyên.

Hàm nhận 1 tham số là tên tệp văn bản chứa dãy số cần đọc.

Lưu số nguyên vào trường `l (long)` của `gtype`.

Nguyên mẫu hàm:

```
struct gvector *read_long(const char *fname);
```

## Bài tập 1b. Đọc dãy số thực từ tệp

**Yêu cầu:** Viết hàm đọc 1 dãy số thực từ 1 tệp văn bản và trả về 1 vec-tơ chứa dãy số đó. Dữ liệu được cho theo định dạng: Đầu tiên là 1 số nguyên  $n$  và tiếp theo là  $n$  số thực.

Hàm nhận 1 tham số là tên tệp văn bản chứa dãy số cần đọc.

Lưu số thực vào trường `d` (double) của `gtype`.

Nguyên mẫu hàm:

```
struct gvector *read_double(const char *fname);
```

# Bài tập 1c. Đọc tệp văn bản theo dòng

**Yêu cầu:** Viết hàm đọc 1 tệp văn bản theo từng dòng và trả về 1 vec-tơ chứa các dòng đọc được.

Hàm nhận 1 tham số là tên tệp văn bản.

Lưu dòng vào trường s (char \*) của gtype.

Nguyên mẫu hàm:

```
struct gvector *read_lines(const char *fname);
```

## Bài tập 2a. Tìm số nguyên nhỏ nhất

**Yêu cầu:** Viết hàm tìm phần tử nhỏ nhất trong 1 vec-tơ gtype, biết rằng các phần tử chứa số nguyên kiểu long ở trường l.

Hàm nhận 1 tham số là 1 vec-tơ gtype.

Hàm trả về 1 giá trị gtype là phần tử nhỏ nhất tìm được.

Nguyên mẫu:

```
gtype gvec_min_l(struct gvector *v);
```

## Bài tập 2b. Tìm số thực lớn nhất

**Yêu cầu:** Viết hàm tìm phần tử lớn nhất trong 1 vec-tơ gtype, biết rằng các phần tử chứa số thực kiểu double ở trường d.

Hàm nhận 1 tham số là 1 vec-tơ gtype.

Hàm trả về 1 giá trị gtype là phần tử lớn nhất tìm được.

Nguyên mẫu:

```
gtype max_double(struct gvector *v);
```

## Bài tập 2c. Đảo ngược các phần tử

**Yêu cầu:** Viết hàm đảo ngược thứ tự các phần tử của 1 vec-tơ gtype.

Hàm nhận 1 tham số là 1 vec-tơ gtype.

Hàm không trả về kết quả.

Nguyên mẫu:

```
void gvec_revert(struct gvector *v);
```

