Các bạn có thể copy nội dung của các đoạn code trong phần này ở link sau:

0.1 Con trỏ cơ bản

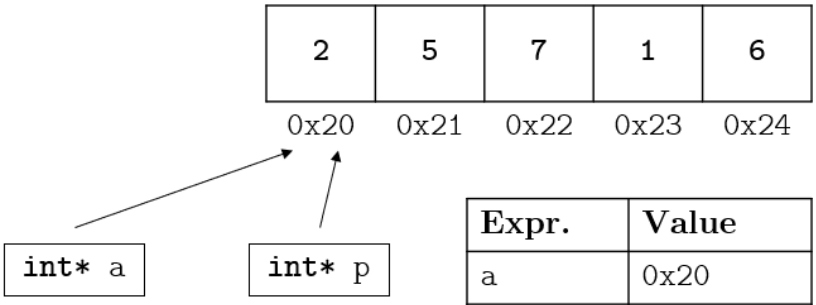
Lưu ý: các địa chỉ của các ô nhớ trong những hình dưới đây chỉ mang tính chất minh hoạ.

0.1.1 Câu a

Do `p` là một con trỏ, trỏ đến `a` nên `*p` chính là `a`. Phép gán `*p += 4` tương đương `a += 4` nên đáp án là `19.75`.

0.1.2 Câu b

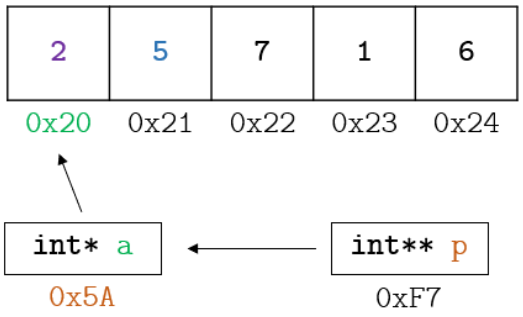
Do `p` trỏ đến cùng vùng nhớ với `a`, nên các thao tác truy xuất bộ nhớ thông qua `p` cũng hoàn toàn tương tự với các thao tác truy xuất bộ nhớ thông qua `a`.



Expr.	Value
a	0x20
a+1	0x21
*a	2
*(a+1)	5

Ta thấy `*(p+1)` trùng với `*(a+1)`, chính là `a[1]`. Còn `*a` chính là `a[0]`, có giá trị là `2`.
Vậy `*(p + 1) += *a;` tương đương với phép gán `a[1] += a[0]`.
Tương tự, `cout « *(a + 1);` là `cout « a[1]`, nên kết quả in ra màn hình sẽ là `7`.

0.1.3 Câu c



Expr.	Value
p	0x5A
a	0x20
*p	0x20
*a	2
*(*p)	2

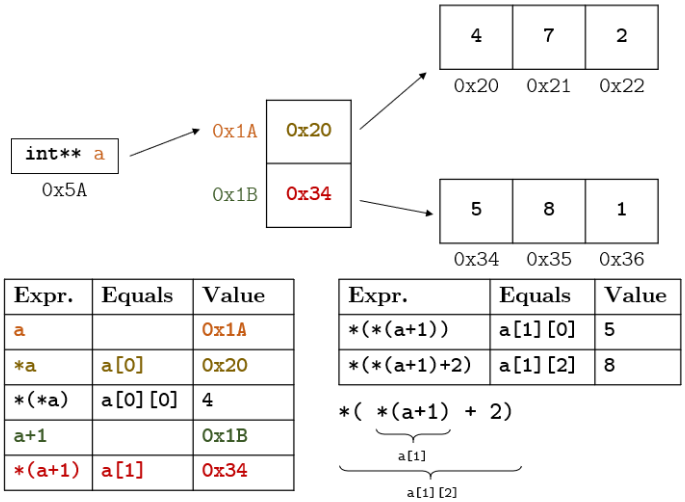
Expr.	Value
*(a+1)	5
*(*p+1)	5

Do `p` là một con trỏ, trỏ đến `a` nên `*p` chính là `a`.
`(*p)[4]` tương đương với `a[4]`, `*(*p + 1)` tương đương với `*(a + 1)`, chính là `a[1]`. Tự suy luận, ta có kết quả là 7.

0.1.4 Câu d

Ta thấy `*(a+2)` chính là `a[2]`.
Nếu ta đặt `u = *(a+2)` thì `(*(a+2)+1)` sẽ trở thành `*(u+1)`, chính là `u[1]`.
Mà vì `u = *(a+2) = a[2]` nên `u[1]` chính là `a[2][1]`.
Giá trị tại vị trí `a[2][1]` chính là `1`.

Trên đây là một cách giải thích theo phong cách "toán học". Nếu các bạn muốn một cách giải thích theo bản chất của con trỏ có thể tham khảo hình dưới đây (tự tham khảo):



Expr.	Equals	Value
a		0x1A
*a	a[0]	0x20
*(*a)	a[0][0]	4
a+1		0x1B
*(a+1)	a[1]	0x34

Expr.	Equals	Value
((a+1))	a[1][0]	5
((a+1)+2)	a[1][2]	8

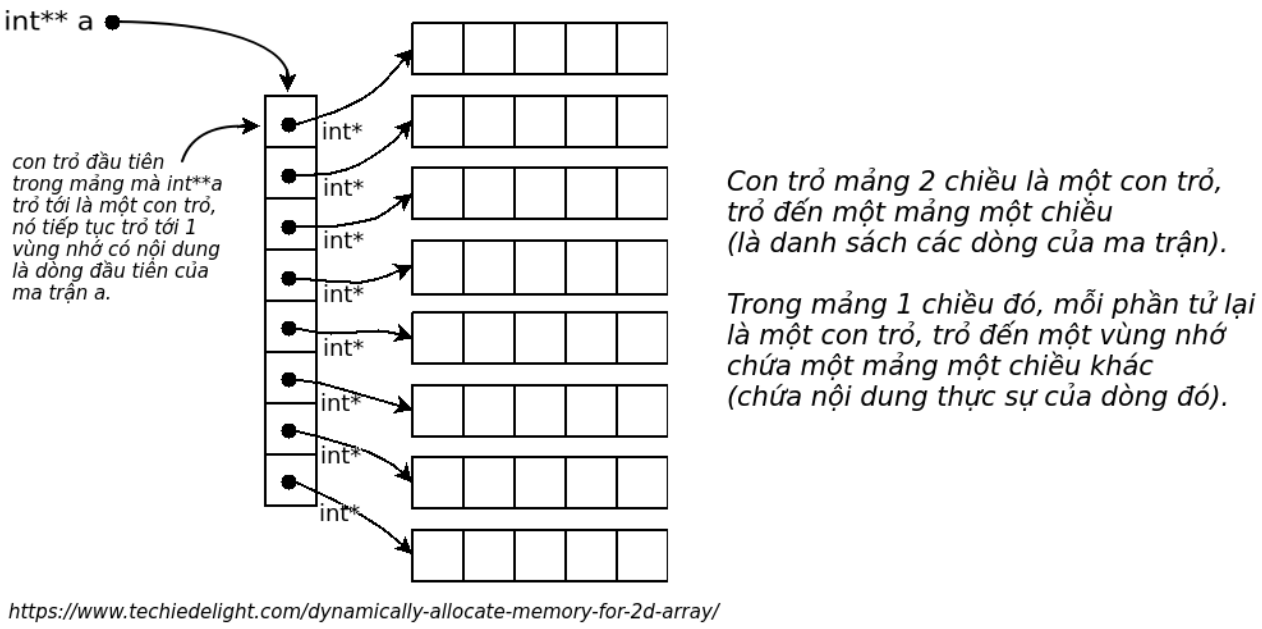
0.2 Con trỏ nâng cao

0.2.1 Câu a

```cpp
// Cấp phát
int n;
cin >> n;
int* a = new int[n];

// Nhập
for (int i = 0; i < n; i++) {
    cin >> a[i];
}

// Giải phóng
delete[] a;
```

0.2.2 Câu b

Nhắc lại kiến thức về con trỏ 2 cấp:

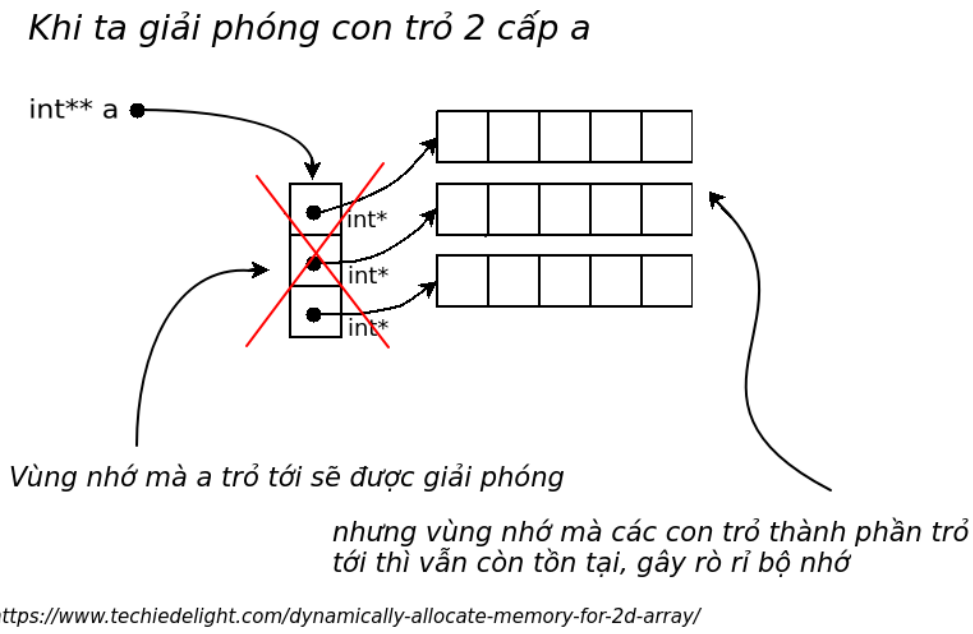


https://www.techiedelight.com/dynamically-allocate-memory-for-2d-array/

```cpp
// Cấp phát
int m, n;
cin >> m >> n;

// Một ma trận có m hàng
float** a = new (float*)[m];

// Mỗi hàng có n cột
for (int i = 0; i < m; i++) {
    a[i] = new float[n];
```

```
}

// Nhập
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        cin >> a[i][j];
    }
}

// Giải phóng
for (int i = 0; i < m; i++) {
    delete[] a[i];
}
delete[] a;
```

Lưu ý: Ta phải giải phóng các phần tử của a trước khi giải phóng a, vì bản thân a chỉ là một con trỏ 2 cấp, nó không thực sự chứa nội dung của ma trận. Khi ta giải phóng a thì các con trỏ thành phần bên trong a vẫn còn tồn tại, gây rò rỉ bộ nhớ nếu như ta không giải phóng trước.



https://www.techiedelight.com/dynamically-allocate-memory-for-2d-array/

0.2.3 Câu c

```
int n;
cin >> n;

// Mảng của các chuỗi
char** a = new (char*)[n];

// Cấp phát và nhập từng chuỗi
for (int i = 0; i < n; i++) {
    int size;
```

```
    cin >> size;

    a[i] = new char[size];
    gets(a[i]);
}

// Giải phóng
for (int i = 0; i < n; i++) {
    delete[] a[i];
}
delete[] a;
```

0.3 Danh sách liên kết

```
Node* initNode(int data) {
    Node* node = new Node;
    node->data = data;
    return node;
}

void deleteNode(Node* node) {
    delete node;
}

void connectNode(Node* backNode, Node* frontNode) {
    if (backNode == NULL) return;
    backNode->next = frontNode;
}

void makeOdd(List& list) {
    // tách số chẵn
    int tmp;
    Node* tmpNode;
    for (Node* node = list.head; node != NULL; node = node->next) {
        if (node->data % 2 != 0) continue;
        tmp = node->data / 2;
        if (tmp % 2 != 0) node->data = tmp;
        else {
            node->data = tmp - 1;
            tmpNode = initNode(tmp + 1);
            connectNode(tmpNode, node->next);
            connectNode(node, tmpNode);
        }
    }

    // loại node trùng nhau liên tiếp
    int lastData = -1;
    Node* prevNode = NULL;
```

```
    Node* nextNode;
    Node* node = list.head;
    while (node != NULL) {
        if (node->data == lastData) {
            nextNode = node->next;
            connectNode(prevNode, nextNode);
            deleteNode(node);
            node = nextNode;
        }
        else {
            lastData = node->data;
            prevNode = node;
            node = node->next;
        }
    }
}
```

0.4 Ngăn xếp, hàng đợi

0.4.1 Ngăn xếp

Ta để ý là các cặp dấu ngoặc xuất hiện theo thứ tự first in first out, có nghĩa là dấu mở ngoặc nào tới sau thì nó phải được đóng trước. Do đó ta sử dụng stack để giải bài toán này. Cụ thể như sau:

- Ta duyệt qua từng kí tự của chuỗi đó.
- Nếu ta gặp một dấu mở ngoặc, ta đẩy nó vào stack.
- Nếu ta gặp một dấu đóng ngoặc, ta kiểm tra thử nó có cùng loại với dấu mở ngoặc mà ta gặp gần nhất hay không (dấu mở ngoặc gần nhất là phần tử trên đỉnh stack). Nếu có, ta tiếp tục xét kí tự tiếp theo. Nếu khác loại thì coi như chuỗi là không hợp lệ (trường hợp này là bị thiếu dấu mở ngoặc, dư dấu đóng ngoặc).
- Sau khi duyệt hết các kí tự trong chuỗi, ta kiểm tra xem trong stack còn dư phần tử nào hay không, nếu có thì ta đã rơi vào trường hợp dư dấu mở ngoặc, thiếu dấu đóng ngoặc.
- Nếu chuỗi nhập vào không vi phạm bất cứ tiêu chuẩn nào thì nó là hợp lệ.

```
bool checkBracketPairs(string s) {
    stack<char> history;

    // Ta duyệt qua từng kí tự
    for (char c : s) {
        // Nếu ta gặp dấu mở ngoặc thì push vào stack
        if (c == '(' || c == '[' || c == '{') {
            history.push(c);
        }
        // Nếu ta gặp dấu đóng ngoặc
        else if (c == ')' || c == ']' || c == '}') {
```

```
            // Nếu dấu đóng ngoặc đó không phải đồng loại
            // của dấu mở ngoặc trên đỉnh stack -> sai
            if (history.empty() || !sameType(history.top(), c) {
                return false;
            }
        }
    }

    // Nếu ta đã duyệt hết chuỗi rồi mà trong stack vẫn còn
    // dư phần tử -> sai
    if (!history.empty()) {
        return false;
    }

    // Nếu không sai thì là đúng :)
    return true;
}

// Hàm kiểm tra xem một cặp ngoặc có cùng loại hay không
bool sameType(char open, char close) {
    if (open == '(' && close == ')') return true;
    if (open == '[' && close == ']') return true;
    if (open == '{' && close == '}') return true;
    return false;
}
```

0.4.2 Hàng đợi

0.5 Đệ quy

Nhắc lại về thuật toán Selection Sort:

- Chọn ra phần tử nhỏ nhất trong n phần tử ban đầu.
- Hoán đổi phần tử đó lên đầu mảng.
- Bỏ qua phần tử đầu tiên, ta xem phần tử tiếp theo như là đầu mảng và tiếp tục lặp lại thuật toán cho đến khi hết mảng.

Ta có thể cài đặt một thuật toán đệ quy trên DSLK như sau:

```
void recursiveSelectionSort(Node* head);
```

- Tìm phần tử nhỏ nhất trong danh sách liên kết.
- Hoán đổi node nhỏ nhất đó lên đầu dãy.
- Gọi hàm sắp xếp đệ quy cho dãy có `head` là `head->next`.

0.6 Quy hoạch động

Lưu ý: khi giải một bài toán có sử dụng quy hoạch động, ta cần làm 2 bước sau:

- Làm theo phương pháp vét cạn trước:
  - Thử làm bài toán đó bằng tay và rút ra thuật toán thực hiện.
  - Mô hình hoá thứ tự các bước thực hiện dưới dạng đồ thị.
  - Cài đặt thuật toán bằng đệ quy.
- Tối ưu hoá:
  - Thêm một "bảng ghi nhớ" vào hàm đệ quy đã viết ở bước trước.
  - Mỗi khi gọi hàm, tra bảng và trả về giá trị đã được lưu, nếu không có thì bắt đầu thực thi quá trình tính toán và lưu giá trị vào bảng ghi nhớ đó.

Bạn có thể tham khảo thêm trong khoá học dài 5 tiếng về quy hoạch động của `freeCodeCamp` (tiếng Anh): Youtube: Dynamic Programming - Learn to Solve Algorithmic Problems & Coding Challenges.

Phân tích bài toán: giả sử ta có `a = {5, 3, 4, 7}` và `n = 7`.

0.7 Bài tập tự luyện ở nhà

Lưu ý: trong file pdf, phần giải thích này chỉ có giải thích, không có code. Các bạn vui lòng vào link được đính kèm ở đầu section để xem code.

0.7.1 Cấp phát động

Không có gì phải giải thích nhiều, thực hiện tương tự như khi thao tác với ma trận số nguyên. Cần lưu ý khi truyền một con trỏ trỏ đến kiểu dữ liệu `T` vào hàm, mà ta muốn thay đổi giá trị của con trỏ đó (như cấp phát hay xoá) thì ta cần phải truyền dưới dạng tham chiếu:

```
void foo(T* &ptr) {
    ptr = new T[n];
    delete[] ptr;
}
```

Hoặc dưới dạng tham trỏ

```
void foo(T** ptr) {
    *ptr = new T[n];
    delete[] *ptr;
}
```

0.7.2 Ngăn xếp

Nhắc lại về 2 cấu trúc:

- Stack: phần tử nào vào sau thì ra trước (như một chồng dĩa).

- Queue: phần tử nào vào sau thì ra sau (như một hàng người đợi mua trà sữa).

Để implement một queue bằng 2 stack, ta có thể làm như sau:

- Giả sử ta có 2 stack gọi là stack (1) và stack (2).
- Khi push một phần tử vào queue, ta push vào (1).
- Khi muốn xem một phần tử ở đầu queue, vì phần tử ở đầu queue lại nằm dưới đáy của stack (1), nên ta pop tất cả phần tử từ stack (1) và chuyển qua stack (2). Lúc này, phần tử ở đầu queue đã nằm trên đỉnh của stack (2). Sau khi xem xong thì ta lại pop tất cả phần tử từ stack (2) bỏ về stack (1).
- Khi muốn xóa một phần tử khỏi queue, ta cũng chuyển tất cả phần tử từ (1) sang (2), nhưng sau đó phải bỏ phần tử trên đỉnh của stack (2) đi, rồi mới chuyển về lại stack (1).

Ảnh động minh hoạ:

- Link ảnh GIF xem đỉnh queue: https://ibb.co/64Hv1bH
- Link ảnh GIF xóa đỉnh queue: https://ibb.co/syPCBFS

Tối ưu hóa: Vì mỗi lần xem đỉnh queue ta phải pop (1) bỏ sang (2) rồi lại phải bỏ về (1), do đó ta nên tạo một biến tên là `top` bên trong struct queue của chúng ta để lưu giá trị của đỉnh queue để truy xuất thuận tiện hơn. Biến này sẽ được cập nhật giá trị khi và chỉ khi:

- Khi queue đang rỗng mà chúng ta thêm một phần tử mới vào, thì phần tử đó sẽ là đỉnh của queue (các phần tử khác sau đó nằm ở phía sau phần tử đầu).
- Khi ta pop đỉnh của queue mà trong queue vẫn còn dữ liệu thì đỉnh queue sẽ là phần tử ngay kế sau.