ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

CƠ SỞ LẬP TRÌNH

Bài thực hành 07

CON TRỎ

Giảng viên thực hành: Lê Đức Khoan

Thành phố Hồ Chí Minh, 12/2025

Mục lục

1 Khái niệm **3**

2 Con trỏ và thao tác với con trỏ **4**

2.1 Toán tử & và * 4

2.2 Con trỏ NULL / nullptr 5

2.3 Phép toán trên con trỏ 5

2.4 Truyền con trỏ vào hàm 6

3 Mối quan hệ giữa con trỏ và mảng **7**

3.1 Truy cập phần tử: toán tử `[]` và số học con trỏ 8

3.2 Truyền mảng vào hàm 9

4 Cấp phát động trong C++ **10**

4.1 Toán tử `new` 10

4.1.1 Cấp phát một biến đơn 10

4.1.2 Cấp phát mảng động 10

4.2 Toán tử `delete` 11

4.2.1 Xoá một biến đơn 11

4.2.2 Xoá mảng động 11

5 Yêu cầu bài nộp **14**

6 Bài tập **15**

6.1 Hoán đổi giá trị hai biến a và b 15

6.2 Tính tổng hai số a và b 15

6.3 Nhập mảng 15

6.4 In mảng ra màn hình 15

6.5 Tìm phần tử lớn nhất trong mảng 15

6.6 Tạo một bản copy của mảng 15

6.7 Đếm số chẵn và tạo mảng số chẵn 16

6.8 Tìm dãy con có tổng lớn nhất trong mảng 16

6.9 Tìm dãy con tăng dài nhất trong mảng 16

6.10 Hoán đổi hai mảng động 16

6.11 Nối hai mảng 16
6.12 Gộp hai mảng (các phần tử phân biệt) và sắp xếp tăng dần 17
6.13 Tạo ma trận 17
6.14 Tạo ma trận tích outer-product 17
6.15 Hoán vị hai dòng / hai cột của ma trận 17
6.16 Tạo ma trận chuyển vị 18
6.17 Nối hai ma trận cùng kích thước theo dòng / theo cột 18
6.18 Nhân hai ma trận A ($m_a \times n_a$) và B ($m_b \times n_b$) 18
6.19 Tìm ma trận con có tổng phần tử lớn nhất 18

1 Khái niệm

Con trỏ là một khái niệm quan trọng trong lập trình, đặc biệt là trong các ngôn ngữ như C, C++, và các ngôn ngữ có khả năng truy cập trực tiếp bộ nhớ. Con trỏ là một biến lưu trữ địa chỉ của một biến khác trong bộ nhớ máy tính. Thay vì chứa giá trị trực tiếp, con trỏ lưu trữ địa chỉ của một ô nhớ mà tại đó giá trị của một biến thực sự được lưu trữ.

Con trỏ có kích thước bằng kích thước địa chỉ CPU:

- 4 byte trên hệ 32-bit
- 8 byte trên hệ 64-bit

```
1    int a = 10;
2    int *p = &a; // p chứa địa chỉ của a
```

2 Con trỏ và thao tác với con trỏ

2.1 Toán tử & và *

Lấy địa chỉ (&)

Toán tử `&` trong C/C++ được gọi là *toán tử lấy địa chỉ* (*address-of operator*). Nó được sử dụng để lấy địa chỉ ô nhớ nơi một biến đang được lưu trữ trong RAM. Mỗi biến khi được tạo trong chương trình đều có một địa chỉ duy nhất, và toán tử `&` cho phép truy xuất địa chỉ đó.

```
int a = 10;
int *p = &a;    // p nhận địa chỉ của biến a
```

Nếu biến `a` đang nằm tại địa chỉ `0x61ff0c`, ta có:

- `a = 10`
- `&a = 0x61ff0c`
- `p = 0x61ff0c`
- `*p = 10`

Toán tử `&` luôn trả về địa chỉ, không bao giờ trả về giá trị.

Các lỗi thường gặp

- Lấy địa chỉ của biểu thức tạm thời:

```
int *p = &(a + b); // Sai
```

- Lấy địa chỉ của hằng số:

```
int *p = &10; // Sai
```

- Trả địa chỉ biến cục bộ:

```
int* foo() {
    int x = 20;
    return &x; // Sai: x bị hủy khi hàm kết thúc
}
```

Giải tham chiếu (dereference) (*)

Toán tử * trong C/C++ có hai vai trò:

1. Dùng để khai báo con trỏ.

```
int a = 10;
int *p = &a;
```

2. Dùng để *giải tham chiếu* (dereference) – tức truy cập giá trị tại ô nhớ mà con trỏ trỏ đến.

```
*p = 20; // thay đổi giá trị của a thành 20
```

Ví dụ minh hoạ con trỏ:

```
int a = 10;
int *p = &a;

cout << p;   // in địa chỉ
cout << *p;  // in 10
*p = 50;     // thay đổi a thành 50
```

2.2 Con trỏ NULL / nullptr

`nullptr` đại diện cho con trỏ trống, an toàn hơn `NULL` và số 0. Khi khởi tạo nếu chưa gán địa chỉ cho con trỏ ta nên gán cho con trỏ giá trị `nullptr` để đảm bảo con trỏ không chứa giá trị rác.

```
int *p;
*p = 10; // LỖI! p chứa giá trị rác
```

Luôn khởi tạo con trỏ:

```
int *p = nullptr;
```

2.3 Phép toán trên con trỏ

Trong C/C++, các con trỏ có thể tham gia vào một số phép toán số học đặc biệt.Điểm quan trọng là: **phép toán trên con trỏ không hoạt động theo byte**, mà theo **kích thước của kiểu dữ liệu** mà con trỏ trỏ đến.

Nếu `int` chiếm 4 byte, thì:

- `p + 1` dịch con trỏ sang địa chỉ lớn hơn 4 byte
- `p + 2` dịch 8 byte
- `p++` tương đương `p = p + 1`

Ví dụ:

```
int a[3] = {10, 20, 30};
int *p = a;

cout << *p << endl; // 10
p++;
cout << *p << endl; // 20
p++;
cout << *p << endl; // 30
```

Các phép toán hợp lệ

- **Dịch chuyển con trỏ**: `p + n`, `p - n`
- **Tăng/giảm**: `p++`, `p-`
- **Hiệu hai con trỏ**: `p2 - p1` (nếu cùng mảng)

Ví dụ:

```
int a[5] = {1,2,3,4,5};
int *p1 = &a[1];
int *p2 = &a[4];

cout << (p2 - p1); // 3
```

2.4 Truyền con trỏ vào hàm

Trong C/C++, truyền con trỏ vào hàm cho phép hàm làm việc trực tiếp với địa chỉ của biến. Điều này giúp hàm có khả năng đọc và thay đổi giá trị của biến gốc, thay vì chỉ làm việc với bản sao như cơ chế truyền tham trị.

```
void increase(int *p) {
    (*p)++;
}

int main() {
    int a = 10;
    increase(&a);
}
```

Ví dụ: Hoán đổi giá trị bằng con trỏ:

```
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

3 Mối quan hệ giữa con trỏ và mảng

Trong C/C++, mảng và con trỏ là hai khái niệm *khác nhau*, nhưng có mối quan hệ rất chặt chẽ.

- **Mảng** (`array`) là một *khối vùng nhớ liên tiếp* gồm nhiều phần tử cùng kiểu.
- **Con trỏ** là một *biến* dùng để lưu *địa chỉ* của một vùng nhớ.

Điểm quan trọng:

- Tên mảng **thường được chuyển ngầm** thành con trỏ tới phần tử đầu tiên khi dùng trong biểu thức.
- Tuy nhiên, bản thân "mảng" không phải là "con trỏ" và không thể gán cho nó một địa chỉ khác.

Ví dụ cơ bản:

```
int main() {
    int a[5] = {1, 2, 3, 4, 5};
```

```
    int *p = nullptr;

    p = a;          // hợp lệ: a suy biến thành &a[0]
    // a = p;       // không hợp lệ: không gán lại địa chỉ cho mảng

    std::cout << a      << std::endl; // địa chỉ phần tử đầu tiên
    std::cout << &a[0]  << std::endl; // giống với dòng trên
    std::cout << p      << std::endl; // giống với hai dòng trên

    return 0;
}
```

Ở ví dụ trên:

- `a` là `int[5]`: *mảng 5 số nguyên.*
- Khi gán `p = a`, tên mảng `a` được suy biến (decay) thành `int*`, trỏ tới phần tử đầu tiên.
- `a` và `&a[0]` có cùng giá trị địa chỉ, nhưng khác *kiểu.*

3.1 Truy cập phần tử: toán tử [] và số học con trỏ

Có hai cách hoàn toàn tương đương để truy cập một phần tử trong mảng qua con trỏ:

1. Dùng toán tử `[]`: `p[i]`
2. Dùng toán tử `*` kết hợp với cộng: `*(p + i)`

Ví dụ minh hoạ:

```
int main() {
    int a[4] = {10, 20, 30, 40};
    int *p = a; // p trỏ đến a[0]

    std::cout << a[1] << std::endl; // 20
    std::cout << p[1] << std::endl; // 20
    std::cout << *(a + 1) << std::endl; // 20
    std::cout << *(p + 1) << std::endl; // 20
```

```
9
10      // Thay đổi phần tử
11      a[2] += 5;    // a[2] = 35
12      *(p + 3) += 10;  // a[3] = 50
13      return 0;
14  }
```

Ghi nhớ:

$$\texttt{a[i]} \equiv \texttt{*(a + i)} \equiv \texttt{*(i + a)} \equiv \texttt{i[a]}$$

3.2 Truyền mảng vào hàm

Khi truyền mảng vào hàm, **tên mảng bị suy biến thành con trỏ** trỏ đến phần tử đầu tiên. Do đó, hai khai báo sau là tương đương:

```
1   void printArray(int a[], int n);
2   void printArray(int *a, int n);
```

Ví dụ: hàm in mảng

```
1   void printArray(int *a, int n) {
2       for (int i = 0; i < n; ++i) {
3           std::cout << a[i] << " ";
4           // hoặc std::cout << *(a + i) << " ";
5       }
6       std::cout << std::endl;
7   }
8
9   int main() {
10      int a[5] = {1, 2, 3, 4, 5};
11      printArray(a, 5); // truyền tên mảng -> suy biến thành int*
12      return 0;
13  }
```

Lưu ý quan trọng:

- Hàm `printArray` *không biết* kích thước thật của mảng, nên ta cần truyền thêm tham số `n`.

- Bên trong hàm, `a` chỉ còn là `int*`, không còn là `int[5]`.

4 Cấp phát động trong C++

Trong C++, cấp phát động cho phép chương trình xin cấp bộ nhớ từ vùng nhớ **Heap** tại thời điểm chạy (runtime), thay vì tại thời điểm biên dịch (static allocation). Cơ chế này rất quan trọng đối với các cấu trúc dữ liệu có kích thước thay đổi (dynamic size) như danh sách liên kết, cây, mảng động v.v.

C++ cung cấp hai toán tử để quản lý bộ nhớ động:

- `new`: cấp phát bộ nhớ
- `delete`: giải phóng bộ nhớ đã cấp phát

Khác với ngôn ngữ có bộ gom rác (garbage collector) như python, C++ yêu cầu lập trình viên tự quản lý bộ nhớ. Nếu quên `delete`, chương trình sẽ bị **rò rỉ bộ nhớ** (memory leak).

4.1 Toán tử `new`

4.1.1 Cấp phát một biến đơn

Cú pháp:

```
Kiểu_dữ_liệu *tên_con_trỏ = new Kiểu_dữ_liệu;
```

Ví dụ:

```
int *p = new int;      // cấp phát 1 số nguyên trên heap
*p = 10;               // gán giá trị
```

Hoặc cấp phát và khởi tạo:

```
double *x = new double(3.14);
char *c = new char('A');
```

4.1.2 Cấp phát mảng động

Cú pháp:

```
Kiểu *p = new Kiểu[kích_thước];
```

Ví dụ:

```
int n = 100;
int *a = new int[n];   // mảng int gồm 100 phần tử
```

Mảng được cấp phát liên tục trên Heap, tương tự mảng thường, nhưng kích thước được xác định tại runtime.

Lưu ý quan trọng

- Mảng cấp phát bằng `new[]` bắt buộc phải giải phóng bằng `delete[]`
- Mảng không có thông tin về kích thước -> lập trình viên phải tự quản lý biến `n`.

4.2 Toán tử `delete`

4.2.1 Xoá một biến đơn

Cú pháp:

```
delete tên_con_trỏ;
```

Ví dụ:

```
int *p = new int(10);
delete p;   // giải phóng vùng nhớ của p
p = nullptr;
```

4.2.2 Xoá mảng động

Cú pháp:

```
delete[] tên_con_trỏ;
```

Ví dụ:

```
int *a = new int[50];
// sử dụng mảng a tại đây...
delete[] a;     // Hợp lệ
a = nullptr;
```

Chú ý: Không được trộn lẫn `new` và `malloc`

- Bộ nhớ cấp phát bởi `new` phải được giải phóng bằng `delete`.
- Bộ nhớ cấp phát bởi `new[]` phải được giải phóng bằng `delete[]`.

Các trường hợp sau là sai:

```
int *p = new int;
free(p);          // sai

int *q = (int*) malloc(sizeof(int) * 10);
delete[] q;       // sai
```

Ví dụ minh hoạ

Ví dụ 1: Cấp phát biến đơn

```
int main() {
    int *p = new int;   // cấp phát
    *p = 42;            // sử dụng

    std::cout << *p << std::endl;

    delete p;           // giải phóng
    p = nullptr;

    return 0;
}
```

Ví dụ 2: Cấp phát mảng động

```
int main() {
    int n = 5;
    int *a = new int[n];

    for (int i = 0; i < n; ++i)
        a[i] = i * 10;

    for (int i = 0; i < n; ++i)
```

```
            std::cout << a[i] << " ";

        delete[] a;
        a = nullptr;

        return 0;
    }
```

Ví dụ 3: Truyền con trỏ để cấp phát mảng

```
    void createArray(int *&p, int n) {
        p = new int[n];
    }

    int main() {
        int *arr = nullptr;
        createArray(arr, 10);

        // sử dụng arr...

        delete[] arr;
        return 0;
    }
```

5 Yêu cầu bài nộp

Sinh viên được cung cấp một file mã nguồn **lab_07.cpp** chứa thư viện, các định nghĩa hàm. Mã nguồn đã bao gồm đầy đủ các định nghĩa hàm cho tất cả các bài tập trong Lab 07. Sinh viên thực hiện các đoạn mã còn thiếu trong từng hàm bắt đầu từ **TODO** và kết thúc ở **END TODO**.

Chú ý: Sinh viên không được tự ý sửa đổi bất kỳ thành phần nào của mã nguồn mẫu như **Thư viện, định nghĩa hàm**. Nếu mã nguồn biên dịch không thành công khi chấm thì sẽ nhận điểm 0 cho bài tập đó. Sinh viên có thể sửa đổi hàm **main** để có thể nhập xuất dữ liệu để kiểm thử các hàm.

Khi nộp bài sinh viên đổi tên file mã nguồn thành **MSSV.cpp** với MSSV là mã số sinh viên được nhà trường cung cấp.

Tên file cpp mẫu:

```
25123456.cpp
```

Tất cả các trường hợp làm sai yêu cầu sẽ nhận điểm 0 cho bài thực hành. Vì thế sinh viên cần đọc kỹ và thực hiện đúng yêu cầu.

6 Bài tập

6.1 Hoán đổi giá trị hai biến a và b

Viết hàm nhận hai con trỏ `int` và hoán đổi giá trị của hai biến mà hai con trỏ trỏ đến.

```
1    void swap(int *a, int *b);
```

6.2 Tính tổng hai số a và b

Viết hàm nhận hai con trỏ `int`, tính tổng giá trị của hai biến tương ứng và trả về con trỏ trỏ đến vùng nhớ chứa kết quả tổng (được cấp phát động bằng `new`).

```
1    int* sum(int *a, int *b);
```

6.3 Nhập mảng

Viết hàm yêu cầu người dùng nhập số lượng phần tử `n`, sau đó cấp phát động mảng `int` kích thước `n` bằng `new[]` và cho phép người dùng nhập vào từng phần tử.

```
1    void inputArray(int *&a, int &n);
```

6.4 In mảng ra màn hình

Viết hàm nhận con trỏ mảng `a` và số phần tử `n`, sau đó in toàn bộ giá trị trong mảng ra màn hình theo thứ tự từ trái sang phải.

```
1    void printArray(int *a, int n);
```

6.5 Tìm phần tử lớn nhất trong mảng

Viết hàm duyệt mảng `a` gồm `n` phần tử và trả về con trỏ trỏ đến phần tử lớn nhất trong mảng (không tạo bản sao mảng).

```
1    int* findMax(int *a, int n);
```

6.6 Tạo một bản copy của mảng

Viết hàm tạo một bản sao đầy đủ của mảng `a` kích thước `n`. Hàm phải cấp phát động mảng mới bằng `new[]` và sao chép từng phần tử. Trả về con trỏ trỏ đến mảng mới.

```
1    int* copyArray(int *a, int n);
```

6.7 Đếm số chẵn và tạo mảng số chẵn

Viết hàm đếm bao nhiêu phần tử trong mảng là số chẵn và lưu kết quả vào biến `count`. Sau đó viết hàm tạo mảng mới chỉ chứa các phần tử chẵn trong mảng ban đầu.

```
int* countEvens(int *arr, int n, int *evens);
int* generateEvenArray(int *arr, int n, int *count);
```

6.8 Tìm dãy con có tổng lớn nhất trong mảng

Viết hàm tìm dãy con liên tiếp có tổng lớn nhất trong mảng `a`. Hàm trả về con trỏ trỏ đến phần tử đầu tiên của dãy con, đồng thời trả ra tổng lớn nhất và số lượng phần tử của dãy con thông qua tham chiếu.

```
int* findLargestSumSubarray(int *a, int n,
                            int &largestSum,
                            int &subarrayLength);
```

6.9 Tìm dãy con tăng dài nhất trong mảng

Viết hàm duyệt mảng `a` và tìm dãy con liên tiếp tăng dần có độ dài lớn nhất. Hàm trả về con trỏ đến phần tử đầu tiên của dãy con này và trả độ dài thông qua tham chiếu.

```
int* findLongestAscendingSubarray(int *a, int n,
                                  int &subarrayLength);
```

6.10 Hoán đổi hai mảng động

Viết hàm hoán đổi hai mảng động `a` và `b` cùng với hai biến `na` và `nb`. Sau khi hoán đổi, `a` trở thành mảng `b` trước đó và ngược lại.

```
void swapArrays(int *&a, int *&b,
                int &na, int &nb);
```

6.11 Nối hai mảng

Viết hàm nhận hai mảng `a` và `b` với kích thước `na` và `nb`. Tạo mảng mới chứa toàn bộ phần tử của `a` sau đó đến `b`.

```
int* concatenateTwoArrays(int *a, int *b,
                          int na, int nb);
```

6.12 Gộp hai mảng (các phần tử phân biệt) và sắp xếp tăng dần

Viết hàm gộp hai mảng `a` và `b` thành một mảng mới theo thứ tự tăng dần. Trả về mảng mới và số phần tử thông qua biến `nc`.

```
int* mergeTwoArrays(int *a, int *b,
                    int na, int nb,
                    int &nc);
```

6.13 Tạo ma trận

Viết hàm cấp phát động ma trận kích thước `m × n` bằng con trỏ cấp 2 (`int**`) và khởi tạo các phần tử trong ma trận với các số nguyên ngẫu nhiên trong khoảng từ 0 đến 100.

```
void generateRandomMatrix(int **&A,
                          int &m, int &n);
```

6.14 Tạo ma trận tích outer-product

Cho hai mảng `a` và `b`. Tạo ma trận kích thước `na × nb` sao cho:

$$M[i][j] = a[i] \cdot b[j]$$

```
int** calculateProductMatrix(int *a, int *b,
                             int na, int nb);
```

6.15 Hoán vị hai dòng / hai cột của ma trận

Viết hai hàm:

1. Hoán đổi hai dòng của ma trận động
2. Hoán đổi hai cột của ma trận động

```
void swapRows(int **a, int rows, int cols,
              int firstRow, int secondRow);
```

```

    void swapColumns(int **a, int rows, int cols,
                     int firstCol, int secondCol);
```

6.16 Tạo ma trận chuyển vị

Viết hàm tạo ma trận chuyển vị kích thước `n × m` của ma trận gốc `m × n`, trong đó phần tử mới thỏa mãn `B[j][i] = A[i][j]`.

```
int** transposeMatrix(int **a, int m, int n);
```

6.17 Nối hai ma trận cùng kích thước theo dòng / theo cột

Viết các hàm nối hai ma trận có cùng số cột (nối dòng) hoặc cùng số dòng (nối cột).

```
    int** concatenateMatricesByRow(int **a, int **b,
                                   int m, int n);

    int** concatenateMatricesByCol(int **a, int **b,
                                   int m, int n);
```

6.18 Nhân hai ma trận A ($m_a \times n_a$) và B ($m_b \times n_b$)

Viết hàm thực hiện phép nhân hai ma trận A và B với kích thước hợp lệ. Ma trận kết quả C có kích thước $m_a \times n_b$.

```
    int** multiplyMatrices(int **a, int **b,
                           int ma, int na,
                           int mb, int nb);
```

6.19 Tìm ma trận con có tổng phần tử lớn nhất

Viết hàm tìm ma trận con A' trong ma trận A sao cho tổng các phần tử trong ma trận con là lớn nhất. Trả về ma trận con được cấp phát động cùng kích thước $m\_sub \times n\_sub$.

```
    int** findLargestSubmatrix(int **a,
                               int m, int n,
                               int &m_sub, int &n_sub);
```