HƯỚNG DẪN XỬ LÝ FILE NHỊ PHÂN

Dương Nguyễn Thái Bảo - Bùi Huy Thông

Mục lục

1 Các kiểu dữ liệu xử lý file nhị phân trong C++

Đối với file text thông thường, C++ hỗ trợ 3 kiểu dữ liệu chính để xử lý: fstream, ifstream và ofstream. Tuy nhiên 3 kiểu này cũng có thể được dùng để xử lý file nhị phân, bằng cách thêm tham số chế độ đọc (mode) là binary file.

1.1 Các thư viện cần thiết

```
#include <iostream>
#include <fstream>
using namespace std;
```

1.2 fstream

Đây là kiểu dữ liệu đa dụng cho cả việc đọc và ghi file.

Cú pháp khai báo:

```
fstream f(file_name, mode);
```

Trong đó:

- *file_name*: chuỗi tên file nhị phân.
- *mode*: chế độ xử lý file. Có một số chế độ đáng chú ý như sau:
  - **ios::in**: chế độ đọc.
  - **ios::out**: chế độ ghi mới.
  - **ios::app**: chế độ ghi thêm.
  - **ios::binary**: chế độ file nhị phân.
  - Có thể kết hợp các chế độ trên bằng toán tử '|'. Ví dụ **ios::in | ios::binary** nghĩa là chế độ đọc file nhị phân.

Đọc file nhị phân:

```
fstream f(file_name, ios::in | ios::binary);
```

Ghi file nhị phân:

```
fstream f(file_name, ios::out | ios::binary);
```

hoặc:

```
fstream f(file_name, ios::app | ios::binary);
```

1.3 ifstream

Đây là cấu trúc đặc thù để đọc file, không có chức năng ghi file.

```
ifstream f(file_name, ios::binary);
```

1.4 ofstream

Đây là cấu trúc đặc thù để file file, không có chức năng đọc file.

```
ofstream f(file_name, ios::binary);
```

2 Một số hàm xử lý file nhị phân

2.1 open

Để mở một file nhị phân ta có thể truyền đường dẫn file vào hàm khởi tạo của biến dùng để xử lý file, hoặc dùng hàm open để mở một cách thủ công:

```
f.open(file_name, mode);
```

Trong đó:

- *f*: một biến kiểu fstream, ifstream hoặc ofstream.
- *file_name*: chuỗi tên file.
- *mode*: chế độ xử lý.

2.2 close

Sau khi mở một file để đọc/ghi và hoàn thành công việc, ta cần đóng file đó lại để chương trình khác có thể truy cập vào file đó tiếp theo.

```
f.close();
```

2.3 read

Với file nhị phân, không có cách đọc tường minh giống như file text (cin, getline, v.v) mà thay vào đó bạn phải đọc dữ liệu theo **từng byte**.

```
f.read(address, size);
```

Lệnh trên sẽ đọc *size* byte tiếp theo trong file đang được mở bởi biến *f* và lưu vào vùng nhớ có địa chỉ bắt đầu là *address*.
Lưu ý: *address* phải được chuyển đổi về kiểu **char***.

Ví dụ: bạn muốn đọc 4 byte tiếp theo trong file nhị phân và lưu nó vào biến A kiểu int:

```
f.read((char*)&A, 4);
```

2.4 write

Cũng giống như hàm đọc, C++ không có cách ghi file nhị phân tường minh như file text (cout, endl, v.v) mà thay vào đó bạn cũng phải ghi dữ liệu theo **từng byte**.

```
f.write(address, size);
```

Lệnh trên sẽ ghi *size* byte trên vùng nhớ bắt đầu từ địa chỉ *address* vào file đang được mở bởi biến *f*. Lưu ý: *address* phải được chuyển đổi về kiểu **char***.

Ví dụ: bạn muốn ghi giá trị của biến A kiểu int vào file nhị phân:

```
f.write((char*)&A, 4);
```

2.5 seekg/seekp

Dùng để di chuyển đầu đọc/ghi tới 1 vị trí mong muốn trong file.

```
seekg(offset, start); // for reading
seekp(offset, start); // for writing
```

Trong đó: Lệnh trên di chuyển đầu đọc/ghi tới vị trí $start + offset$. Trong đó *offset* là một số nguyên và *start* có thể nhận 1 trong 3 giá trị sau:

- **ios::beg**: vị trí bắt đầu của file.
- **ios::cur**: vị trí đầu đọc/ghi hiện tại.
- **ios::end**: vị trí kết thúc file (dữ liệu trong file được lưu từ đầu cho tới trước vị trí này, hay nói cách khác thì vị trí này không chứa dữ liệu).

Ví dụ 1: bỏ qua 8 byte tiếp theo khi đọc file:

```
f.seekg(8, ios::cur);
```

Ví dụ 2: đọc ký tự cuối cùng trong file và in ra màn hình:

```
f.seekg(-1, ios::end);
char c;
f.read(&c, 1);
cout << c;
```

2.6 tellg/tellp

Trả về vị trí hiện tại của đầu đọc/ghi tính theo byte và tính từ đầu file (đầu file = vị trí 0).

```
int pos = f.tellg(); // for reading
int pos = f.tellp(); // for writing
```

Ví dụ: tính kích thước của file trước khi bắt đầu đọc file:

```
f.seekg(0, ios::end);
int file_size = f.tellg(); // size of file in bytes
f.seekg(0, ios::beg);
```

3 Ví dụ minh họa

1. Ghi giá trị của các biến vào file nhị phân tên "*vars.dat*".

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ofstream f("vars.dat", ios::binary);
    int A = 123;
    double B = 100.142;
    char S[16] = "Hello 20CLC910!"; // 15 chars + '\0'

    f.write((char*)&A, 4); // int = 4 bytes
    f.write((char*)&B, 8); // double = 8 bytes
    f.write(S, 16);

    f.close();
}
```

Bây giờ nếu mở file "*vars.dat*" lên bằng 1 text editor bất kì bạn sẽ thấy một số ký tự kì lạ và chuỗi "Hello 20CLC910!" chứ không hề thấy 2 số A và B.

2. Đọc các biến trong file "*vars.dat*" lên lại vào bộ nhớ:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ifstream f("vars.dat", ios::binary);
    int A;
    double B;
    char S[16];

    f.read((char*)&A, 4); // int = 4 bytes
    f.read((char*)&B, 8); // double = 8 bytes
    f.read(S, 16);

    cout << A << endl;
    cout << B << endl;
    cout << S << endl;

    f.close();
}
```

Kết quả màn hình console:

```
123
100.142
Hello 20CLC910!
```

3. Cho cấu trúc sinh viên gồm 3 thông tin: MSSV (số nguyên dương 2 byte), tên (chuỗi tối đa 29 ký tự) và điểm (số thực 8 byte). Viết các chương trình ghi một mảng sinh viên vào một file nhị phân và đọc nó lên lại.

Ghi file:

```
#include <iostream>
#include <fstream>
using namespace std;

struct Student {
    unsigned short id; // 2 byte positive integer
    char name[30]; // 29 chars + '\0'
    double score; // 8 byte floating point number
};

int main() {
    Student students[] = {
        {1, "Nguyen Van A", 5.0},
        {2, "Tran Duc B", 6.0},
        {3, "Le Thi C", 10.0},
    };
    int num = 3;
    ofstream f("students.dat", ios::binary);

    for (int i = 0; i < num; ++i) {
        f.write((char*)&students[i].id, 2);
        f.write(students[i].name, 30);
        f.write((char*)&students[i].score, 8);
    }

    f.close();
}
```

Nhận thấy rằng 3 biến id, name và score sẽ luôn nằm cạnh nhau trên vùng nhớ nên bạn có thể coi 1 biến Student là 1 khối liên tục. Vì vậy, dòng 20-22 ở trên có thể viết gọn lại thành 1 dòng như sau:

```
f.write((char*)&students[i], 40); // 2 + 30 + 8 = 40 bytes
```

Đọc file:

```
#include <iostream>
#include <fstream>
using namespace std;

```

```
5  struct Student {
6      unsigned short id; // 2 byte positive integer
7      char name[30]; // 29 chars + '\0'
8      double score; // 8 byte floating point number
9  };
10
11 int main() {
12     ifstream f("students.dat", ios::binary);
13
14     // we can't declare a student array here because we don't know its size
15     // so now we compute the size before reading the file
16     f.seekg(0, ios::end);
17     int num = f.tellg() / 40; // file size / struct size
18     f.seekg(0, ios::beg);
19
20     Student* students = new Student[num]; // now we can declare an array
21     for (int i = 0; i < num; ++i) {
22         f.read((char*)&students[i].id, 2);
23         f.read(students[i].name, 30);
24         f.read((char*)&students[i].score, 8);
25         cout << students[i].id << "\t" << students[i].name << "\t" << students[i].score << endl;
26     }
27
28     delete[] students;
29     f.close();
30 }
```

Tương tự với việc ghi file, bạn cũng có thể rút gọn dòng 22-24 thành 1 dòng như sau:

```
f.read((char*)&students[i], 40);
```

Kết quả màn hình console:

```
1       Nguyen Van A    5
2       Tran Duc B      6
3       Le Thi C        10
```

4 Một số lưu ý

- File nhị phân không có phần mở rộng thống nhất mà tùy vào chức năng của nó (ví dụ *.mp4 là file nhị phân lưu trữ video, *.png là file nhị phân chứa ảnh, *.exe là file nhị phân chứa chương trình thực thi, v.v). Với mục đích chứa dữ liệu đơn giản, có thể dùng một số phần mở rộng thông dụng như .dat, .bin, v.v.
- Lấy kích thước của 1 biến/mảng/struct/... bằng cách sử dụng hàm **sizeof**. Thay vì phải ghi nhớ kiểu int kích thước 4 byte, double 8 byte, v.v bạn có thể sử dụng **sizeof(biến/mảng/struct/...)** để lấy kích thước một cách tự động.
- Kích thước của struct không phải lúc nào cũng bằng tổng kích thước các biến thành viên của nó. Mặc định, các biến

thành viên trong struct sẽ được đặt ở 1 địa chỉ mới **chia hết cho 8** để tăng tốc độ truy cập, dẫn tới việc chúng có thể không nằm liên tục nhau như trong ví dụ minh họa 3, và kích thước của struct có thể lớn hơn tổng kích thước các biến thành viên. Để đảm bảo điều này không xảy ra, bạn chỉ cần thêm chỉ thị **#pragma pack(1)** vào đầu mã nguồn.