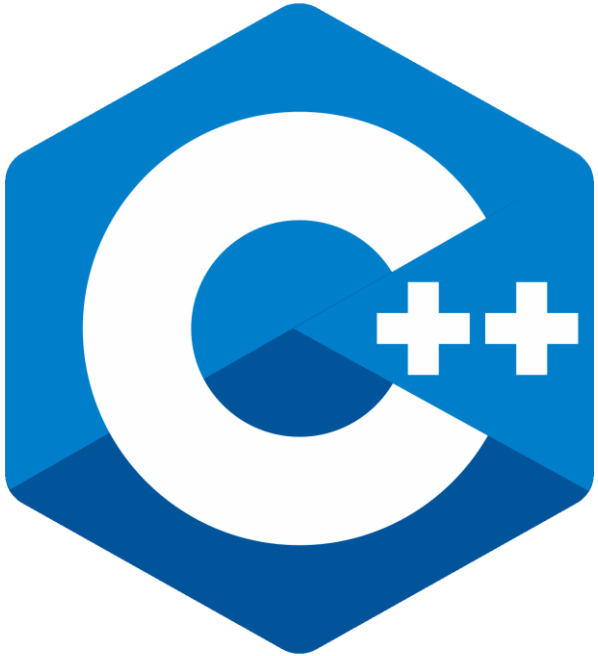
KỸ THUẬT LẬP TRÌNH

TRẦN ĐÌNH LUYỆN

Cấu trúc dữ liệu

Cung cấp kiến thức về các cấu trúc dữ liệu thường sử dụng và sử dụng để giải quyết các bài toán.



Giới thiệu

- Phần lớn các bài toán trong thực tế liên quan tới các dữ liệu phức hợp, những kiểu dữ liệu cơ bản trong ngôn ngữ lập trình không đủ biểu diễn
- Ví dụ:
  - Dữ liệu sinh viên: Họ tên, ngày sinh, quê quán, mã số SV,...
  - Mô hình hàm truyền: Đa thức tử số, đa thức mẫu số
  - Mô hình trạng thái: Các ma trận A, B, C, D
  - Đối tượng đồ họa: Kích thước, màu sắc, đường nét, font chữ, ...
- Phương pháp biểu diễn dữ liệu: định nghĩa kiểu dữ liệu mới sử dụng cấu trúc (struct, class, union, ...)

Biểu diễn dữ liệu

- Đa số những dữ liệu thuộc một ứng dụng có liên quan với nhau => cần biểu diễn trong một tập hợp có cấu trúc, ví dụ:
  - Danh sách sinh viên: Các dữ liệu sinh viên được sắp xếp theo thứ tự Alphabet
  - Mô hình tổng thể cho hệ thống điều khiển: Bao gồm nhiều thành phần tương tác
  - Dữ liệu quá trình: Một tập dữ liệu có thể mang giá trị của một đại lượng vào các thời điểm gián đoạn, các dữ liệu đầu vào liên quan tới dữ liệu đầu ra
  - Đối tượng đồ họa: Một cửa sổ bao gồm nhiều đối tượng đồ họa, một bản vẽ cũng bao gồm nhiều đối tượng đồ họa

Biểu diễn dữ liệu

- Thông thường, các dữ liệu trong một tập hợp có cùng kiểu, hoặc ít ra là tương thích kiểu với nhau
- Kiểu mảng không phải bao giờ cũng phù hợp!
- Sử dụng kết hợp một cách khéo léo kiểu cấu trúc và kiểu mảng đủ để biểu diễn các tập hợp dữ liệu bất kỳ

Quản lý dữ liệu

- Các giải thuật (hàm) thao tác với dữ liệu, nhằm quản lý dữ liệu một cách hiệu quả:
  - Bổ sung một mục dữ liệu mới vào một danh sách, một bảng, một tập hợp, ...
  - Xóa một mục dữ liệu trong một danh sách, bảng, tập hợp,..
  - Tìm một mục dữ liệu trong một danh sách, bảng tập hợp,...theo một tiêu chuẩn cụ thể
  - Sắp xếp một danh sách theo một tiêu chuẩn nào đó
  - ....

Quản lý dữ liệu hiệu quả

- Tiết kiệm bộ nhớ
- Truy nhập nhanh, thuận tiện: Thời gian cần cho bổ sung, tìm kiếm và xóa bỏ các mục dữ liệu phải ngắn
- Linh hoạt: Số lượng các mục dữ liệu không (hoặc ít) bị hạn chế cố định, không cần biết trước khi tạo cấu trúc, phù hợp với cả bài toán nhỏ và lớn
- Hiệu quả quản lý dữ liệu phụ thuộc vào
  - Cấu trúc dữ liệu được sử dụng
  - Giải thuật được áp dụng cho bổ sung, tìm kiếm, sắp xếp, xóa bỏ

Các cấu trúc dữ liệu thông dụng

- Mảng : Tập hợp các dữ liệu có thể truy nhập tùy ý theo chỉ số
- Danh sách (list): Tập hợp các dữ liệu được móc nối đôi một với nhau và có thể truy nhập tuần tự
- Cây (tree): Tập hợp các dữ liệu được móc nối với nhau theo cấu trúc cây, có thể truy nhập tuần tự từ gốc. Nếu mỗi nút có tối đa hai nhánh: cây nhị phân (binary tree)
- Bìa, bảng (map): Tập hợp các dữ liệu có sắp xếp, có thể truy nhập rất nhanh theo mã khóa (key)
- Hàng đợi (queue): Tập hợp các dữ liệu có sắp xếp tuần tự, chỉ bổ sung vào từ một đầu và lấy ra từ đầu còn lại

Các cấu trúc dữ liệu thông dụng

- Tập hợp (set): Tập hợp các dữ liệu được sắp xếp tùy ý nhưng có thể truy nhập một cách hiệu quả
- Ngăn xếp (stack): Tập hợp các dữ liệu được sắp xếp tuần tự, chỉ truy nhập được từ một đầu
- Bảng hash (hash table): Tập hợp các dữ liệu được sắp xếp dựa theo một mã số nguyên tạo ra từ một hàm đặc biệt
- Bộ nhớ vòng (ring buffer): Tương tự như hàng đợi, nhưng dung lượng có hạn, nếu hết chỗ sẽ được ghi quay vòng
- Trong toán học và trong điều khiển: vector, ma trận, đa thức, phân thức, hàm truyền, ..

Mảng

- Mảng cho phép biểu diễn và quản lý dữ liệu một cách khá hiệu quả:
  - Đọc và ghi dữ liệu rất nhanh qua chỉ số hoặc qua địa chỉ
  - Tiết kiệm bộ nhớ

```
Student student_list[100];
```

- Số phần tử phải là hằng số (biết trước khi biên dịch, người sử dụng không thể nhập số phần tử, không thể cho số phần từ là một biến) => kém linh hoạt
- Chiếm chỗ cứng trong ngăn xếp (đối với biến cục bộ) hoặc trong bộ nhớ dữ liệu chương trình (đối với biến toàn cục) =>sử dụng bộ nhớ kém hiệu quả, kém linh hoạ

Mảng động

- Mảng động là một mảng được cấp phát bộ nhớ theo yêu cầu, trong khi chương trình chạy

```
#include <stdlib.h> /* C */
int n = 50;
float* p1= (float*) malloc(n*sizeof(float)); /* C */
double* p2= new double[n]; // C++
```

- Sử dụng con trỏ để quản lý mảng động: Cách sử dụng không khác so với mảng tĩnh

```
p1[0] = 1.0f; p2[0] = 2.0;
```

- Sau khi sử dụng xong giải phóng bộ nhớ

```
free(p1); /* C */
delete [] p2; // C+
```

Cấp phát bộ nhớ

- Hàm malloc() yêu cầu tham số là số byte, trả về con trỏ không kiểu (void*) mang địa chỉ vùng nhớ mới được cấp phát (nằm trong heap), trả về 0 nếu không thành công.
- Hàm free() yêu cầu tham số là con trỏ không kiểu (void*), giải phóng vùng nhớ có địa chỉ đưa vào
- Toán tử new chấp nhận kiểu dữ liệu phần tử kèm theo số lượng phần tử của mảng cần cấp phát bộ nhớ (trong vùng heap), trả về con trỏ có kiểu, trả về 0 nếu không thành công.
- Toán tử delete[] yêu cầu tham số là con trỏ có kiểu.
- Toán tử new và delete áp dụng cho mọi đối tượng con trỏ

Lưu ý

- Con trỏ có vai trò quản lý mảng (động), chứ con trỏ không phải là mảng (động)
- Cấp phát bộ nhớ và giải phóng bộ nhớ chứ không phải cấp phát con trỏ và giải phóng con trỏ
- Chỉ giải phóng bộ nhớ một lần

Lưu ý

```
int* p;
p[0] = 1;          // never do it
new(p);            // access violation!
p = new int[100];        // OK
p[0] = 1;          // OK
int* p2=p;         // OK
delete[] p2;       // OK
p[0] = 1;          // access violation!
delete[] p;              // very bad!
p = new int[50]; // OK, new array
```

Biến con trỏ

- Ý nghĩa: Các đối tượng có thể được tạo ra động, trong khi chương trình chạy (bổ sung sinh viên vào danh sách, vẽ thêm một hình trong bản vẽ, bổ sung một khâu trong hệ thống,...)

```
int* p = new int;
*p = 1;
p[0]= 2; // the same as above
p[1]= 1; // access violation!
int* p2 = new int(1); // with initialization
delete p;
delete p2;
Student* ps = new Student;
ps->code = 1000;
delete ps;
```

Ý nghĩa

- Hiệu suất:
  - Bộ nhớ được cấp phát đủ dung lượng theo yêu cầu và khi được yêu cầu trong khi chương trình đã chạy
  - Bộ nhớ được cấp phát nằm trong vùng nhớ tự do còn lại của máy tính (heap), chỉ phụ thuộc vào dung lượng bộ nhớ của máy tính
  - Bộ nhớ có thể được giải phóng khi không sử dụng tiếp.
- Linh hoạt:
  - Thời gian "sống" của bộ nhớ được cấp phát động có thể kéo dài hơn thời gian "sống" của thực thể cấp phát nó.
  - Có thể một hàm gọi lệnh cấp phát bộ nhớ, nhưng một hàm khác giải phóng bộ nhớ.
  - Sự linh hoạt cũng dễ dẫn đến những lỗi "rò rỉ bộ nhớ"

Ví dụ

```
1   #include <bits/stdc++.h>
2   using namespace std;
3   struct Date{
4     int day, month, year;
5     void display(){
8   };
9   Date* createDateList(int n) {
10    Date* p = new Date[n];
11    return p;
12  }
13  int main() {
14    int n;
15    cout << "Enter the number of your favorite dates:";
16    cin >> n;
17    Date* date_list = createDateList(n);
18    for (int i=0; i < n; ++i){
23    for (int i=0; i < n; ++i)
24      date_list[i].display();
25    delete [] date_list;
26    return 1;
27  }
```

Tham số đầu ra là con trỏ

```
void createDateList(int n, Date* &p) {
  p = new Date[n];
}
int main() {
  int n;
  cout << "Enter the number of your favorite dates:";
  cin >> n;
  Date* date_list;// = createDateList(n);
  createDateList(n, date_list);
  for (int i=0; i < n; ++i){
  for (int i=0; i < n; ++i)
    date_list[i].display();
  delete [] date_list;
  return 1;
}
```

Bài tập

1. Định nghĩa cấu trúc dữ liệu lưu trữ một danh sách các sinh viên (gồm mã sinh viên, họ tên, năm sinh và lớp). Danh sách phải thể hiện được các yếu tố sau:
    - Số sinh viên tối đa lưu trữ là 1 triệu
    - Số sinh viên hiện có
    - Danh sách có rỗng hay không
    - Danh sách đã đầy hay chưa
2. Viết các hàm sau:
    - Thêm 1 sinh viên vào cuối danh sách
    - Thêm 1 sinh viên vào vị trí k
    - Xóa 1 sinh viên
    - Cho một mã sinh viên, lấy thông tin của 1 sinh viên có mã đó.

string

- Cho phép lập trình với các chuỗi ký tự một cách rất thuận tiện
  - Không cần quan tâm tới quản lý bộ nhớ động
  - Có thể sao chép, gán giống như các kiểu dữ liệu cơ bản
  - Có thể truy nhập ký tự qua chỉ số toán tử [] giống như chuỗi ký tự thô
  - Có thể áp dụng các phép toán +, ==, !=, > , <, ...
  - Có thể truy nhập chuỗi con, tìm kiếm, thay thế ký tự,...
  - Có thể áp dụng các thuật toán tổng quát (string, wstring cũng được coi là các container, có các hàm begin(), end())

string

- Khai báo biến string
  - `string s1( "Hello" );`
  - `string s2( 8, 'x' );`
  - `string month = "March";`
- Các hàm truy nhập thuộc tính
  - `n=s1.size(); // Number of characters in string`
  - `n=s1.length(); // Same as size()`
  - `n=s1.capacity(); // Number of elements that can be stored without reallocation`
  - `n=s1.max_size(); // Maximum possible string size`
  - `if (s1.empty()) // Returns true if empty`
  - `s1.resize(newlength); // Resizes string to newlength`

string

```
s2 = s1;
s2.assign(s1);
myString.assign(s, start, N);
s2[0] = s3[2];
s3.append( "pet" );
s3 += "pet";
s3.append( s1, start, N );
So sánh chuỗi ký tự: ==, !=, <, >, <=,>=
s = s1.substr( start, N );
s1.swap(s2);
```

string

- Các hàm tìm kiếm: trả về chỉ số nếu tìm thấy và string::npos nếu không tìm thấy
  - `i=s1.find(s2);`
  - `i=s1.rfind(s2);`
  - `i=s1.find_first_of(s2);`
  - `i=s1.find_last_of(s2);`
  - `i=s1.find_first_not_of(s2);`
  - `i=s1.find_last_not_of(s2);`
- Các hàm xóa và thay thế
  - `s1.erase(start);`
  - `s1.replace(begin,N,s2);`

STL (Standard Template Library) Containers

- vector
- List
- deque
- map, multimap,
- set, multiset
- queue, priority_queue,
- stack

Hàm thành viên

- Hàm thành viên cho tất cả các cấu trúc
  - Hàm tạo mặc định, hàm tạo bản sao
  - Hàm hủy
  - empty
  - max_size, size
  - = < <= > >= == !=
  - swap
- Hàm thành viên chỉ cho các cấu trúc dãy
  - begin, end
  - rbegin, rend
  - erase, clear

Vector

- #include <vector>
- Một mảng động thực sự: truy nhập tùy ý, co dãn được
- Hàm tạo:
  - `vector<T>()`
  - `vector<T>(size_t num_elements)`
  - `vector<T>(size_t num, T init)`
- Thuộc tính:
  - `v.empty()`
  - `v.size()`
  - `v.capacity()`
  - `v.begin(); v.end()`

Vector

```
vector<int> v;
v.push_back(42);
v.insert(iter before, T val)
v.insert(iter before, iter start, iter end)
v.at(i)
v[i]
v.front()
v.back()
v.pop_back()
v.clear()
v.erase(iterator i)
v.erase(iter start, iter end)
```

deque

- Deque là từ viết tắt của double-ended queue (hàng đợi hai đầu). Deque có các ưu điểm như:
  - Các phần tử có thể truy cập thông qua chỉ số vị trí của nó.
  - Chèn hoặc xóa phần tử ở cuối hoặc đầu của dãy.
  - #include <deque>
- push_back : thêm phần tử vào ở cuối deque.
- push_front : thêm phần tử vào đầu deque.
- pop_back : loại bỏ phần tử ở cuối deque.
- pop_front : loại bỏ phần tử ở đầu deque.
- ...

iterator

- Các lớp chứa của STL (vector, list,...) có định nghĩa kiểu iterator tương ứng để duyệt các phần tử (theo thứ tự xuôi)
  - Mỗi iterator chứa vị trí của một phần tử
  - Các hàm begin() và end() trả về một iterator tương ứng với các vị trí đầu và cuối
  - Các toán tử với iterator:
    i++ phần tử kế tiếp
    i-- phần tử liền trước
    *i giá trị của phần tử
- Tương tự, có reverse_iterator để duyệt theo thứ tự ngược. Các hàm rbegin() và rend()

stack

- Stack là một loại container adaptor, được thiết kế để hoạt động theo kiểu LIFO

  #include <stack>
- Các hàm:
  - size : trả về kích thước hiện tại của stack.
  - empty : true stack nếu rỗng, và ngược lại.
  - push : đẩy phần tử vào stack.
  - pop : loại bỏ phẩn tử ở đỉnh của stack.
  - top : truy cập tới phần tử ở đỉnh stack.

queue

- Queue là một loại container adaptor, được thiết kế để hoạt động theo kiểu FIFO
- Trong queue, có hai vị trí quan trọng là vị trí đầu danh sách (front), nơi phần tử được lấy ra, và vị trí cuối danh sách (back), nơi phần tử cuối cùng được thêm vào.

  #include <queue>
- Các hàm: size, empty, push, pop, front, back

priority_queue

- Priority queue là một loại container adaptor, được thiết kế đặc biệt để phần tử ở đầu luôn luôn lớn nhất (theo một quy ước về độ ưu tiên nào đó) so với các phần tử khác.
- Nó giống như một heap, mà ở đây là heap max, tức là phần tử có độ ưu tiên lớn nhất có thể được lấy ra và các phần tử khác được chèn vào bất kì.

  #include <queue>
- Các hàm: size, empty, push, pop, top

Q&A