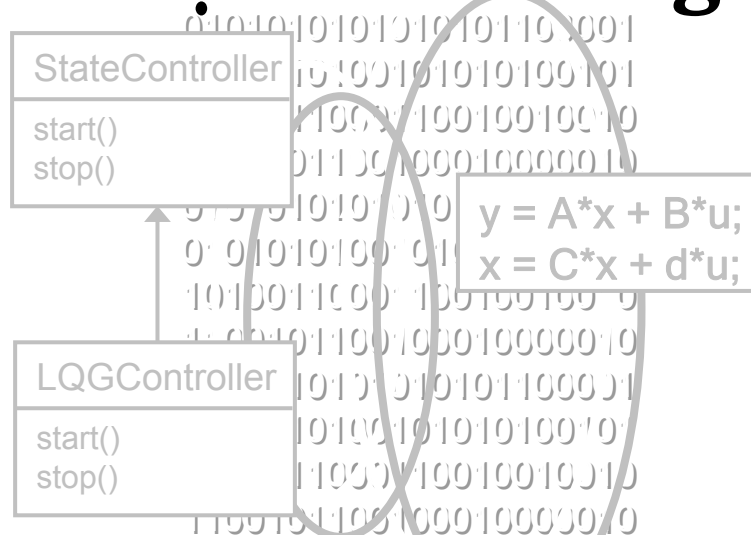


Kỹ thuật lập trình

Chương 10: Thuật toán tổng quát



Nội dung chương 10



- 10.1 Tổng quát hóa kiểu dữ liệu phần tử
- 10.2 Tổng quát hóa phép toán cơ sở
- 10.3 Tổng quát hóa phương pháp truy cập phần tử

10.1 Tổng quát hóa kiểu dữ liệu phần tử

- Thực tế:
 - Khoảng 80% thời gian làm việc của một người thư ký văn phòng trước đây (và hiện nay ở nhiều nơi) sử dụng cho công việc tìm kiếm, sắp xếp, đối chiếu, so sánh,... tài liệu và hồ sơ
 - Trung bình, khoảng 80% mã chương trình và thời gian thực hiện chương trình dành cho thực hiện các thuật toán ít liên quan trực tiếp tới bài toán ứng dụng cụ thể, mà liên quan tới tìm kiếm, sắp xếp, lựa chọn, so sánh... dữ liệu
- Dữ liệu được quản lý tốt nhất trong các cấu trúc dạng "container" (vector, list, map, tree, queue,...)
- Vấn đề xây dựng hàm áp dụng cho các "container": Nhiều hàm chỉ khác nhau về kiểu dữ liệu tham số áp dụng, không khác nhau về thuật toán
- Giải pháp: Xây dựng khuôn mẫu hàm, tổng quát hóa kiểu dữ liệu phần tử

- Ví dụ: Thuật toán tìm địa chỉ phần tử đầu tiên trong một mảng có giá trị lớn hơn một số cho trước:

```
template <typename T>
T* find_elem(T *first, T* last, T k) {
    while (first != last && !(*first > k))
        ++first;
    return first;
}

void main() {
    int a[] = { 1, 3, 5, 2, 7, 9, 6 };
    int *p = find_elem(a,a+7,4);
    if (p != a+7) {
        cout << "First number > 4 :" << *p;
        p = find_elem(p+1,a+7,4);
        if (p != a+7) cout << "Second number > 4:" << *p;
    }
    double b[] = { 1.5, 3.2, 5.1, 2.4, 7.6, 9.7, 6.5 };
    double *q = find_elem(b+2,b+6,7.0);
    *q = 7.0;
    ...
}
```

- Ví dụ: Thuật toán cộng hai vector, kết quả lưu vào vector thứ ba

```
#include <assert.h>
#include "myvector.h"
template <typename T>
void addVector(const Vector<T>& a, const Vector<T>& b,
               Vector<T>& c) {
    assert(a.size() == b.size() && a.size() == c.size());
    for (int i= 0; i < a.size(); ++i)
        c[i] = a[i] + b[i];
}

template <typename T>
Vector<T> operator+(const Vector<T>&a, const Vector<T>& b) {
    Vector<T> c(a.size());
    addVector(a,b,c);
    return c;
}
```

10.2 Tổng quát hóa phép toán cơ sở

- Vấn đề: Nhiều thuật toán chỉ khác nhau ở một vài phép toán (cơ sở) trong khi thực hiện hàm
- Ví dụ:
 - Các thuật toán tìm địa chỉ phần tử đầu tiên trong một mảng số nguyên có giá trị lớn hơn, nhỏ hơn, lớn hơn hoặc bằng, nhỏ hơn hoặc bằng, ... một số cho trước
 - Các thuật toán cộng, trừ, nhân, chia,... từng phần tử của hai mảng số thực, kết quả lưu vào một mảng mới
 - Các thuật toán cộng, trừ, nhân, chia,... từng phần tử của hai vector (hoặc của hai danh sách, hai ma trận, ...)
- Giải pháp: Tổng quát hóa thuật toán cho các phép toán cơ sở khác nhau!

```

template <typename COMP>
int* find_elem(int* first, int* last, int k, COMP comp) {
    while (first != last && !comp(*first, k))
        ++first;
    return first;
}
bool is_greater(int a, int b) { return a > b; }
bool is_less(int a, int b)    { return a < b; }
bool is_equal(int a, int b)   { return a == b; }
void main() {
    int a[] = { 1, 3, 5, 2, 7, 9, 6 };
    int* alast = a+7;
    int* p1 = find_elem(a,alast,4,is_greater);
    int* p2 = find_elem(a,alast,4,is_less);
    int* p3 = find_elem(a,alast,4,is_equal);
    if (p1 != alast) cout << "First number > 4 is " << *p1;
    if (p2 != alast) cout << "First number < 4 is " << *p2;
    if (p3 != alast) cout << "First number = 4 is at index "
                           << p3 - a;

    char c; cin >> c;
}

```

Tham số khuôn mẫu cho phép toán

- Có thể là một hàm, ví dụ

```
bool is_greater(int a, int b){ return a > b; }
bool is_less(int a, int b)   { return a < b; }
int  add(int a, int b)       { return a + b; }
int  sub(int a, int b)       { return a - b; }
...
```

- Hoặc tốt hơn hết là một đối tượng thuộc một lớp có hỗ trợ (nạp chồng) toán tử gọi hàm => **đối tượng hàm**, ví dụ

```
struct Greater {
    bool operator()(int a, int b) { return a > b; }
};
struct Less {
    bool operator()(int a, int b) { return a < b; }
};
struct Add {
    int operator()(int a, int b) { return a + b; }
};
...
```


■ Ví dụ sử dụng **đối tượng hàm**

```
void main() {  
    int a[] = { 1, 3, 5, 2, 7, 9, 6 };  
    int* alast = a+7;  
    Greater greater;  
    Less less;  
    int* p1 = find_elem(a,alast,4,greater);  
    int* p2 = find_elem(a,alast,4,less);  
    if (p1 != alast) cout << "First number > 4 is " << *p1;  
    if (p2 != alast) cout << "First number < 4 is " << *p2;  
  
    p1 = find_elem(a,alast,4,Greater());  
    p2 = find_elem(a,alast,4,Less());  
    char c; cin >> c;  
}
```

Ưu điểm của đối tượng hàm

- Đối tượng hàm có thể chứa trạng thái
- Hàm toán tử () có thể định nghĩa inline => tăng hiệu suất

```
template <typename OP>
void apply(int* first, int* last, OP& op) {
    while (first != last) {
        op(*first);
        ++first;
    }
}

class Sum {
    int val;
public:
    Sum(int init = 0) : val(init) {}
    void operator()(int k) { val += k; }
    int value() const { return val; }
};
```

```

class Prod {
    int val;
public:
    Prod(int init=1): val(init) {}
    void operator()(int k) { val *= k; }
    int value() const { return val; }
};

struct Negate {void operator()(int& k) { k = -k;} };
struct Print { void operator()(int& k) { cout << k << ' ';} };

void main() {
    int a[] = {1, 2, 3, 4, 5, 6, 7};
    Sum sum_op;
    Prod prod_op;
    apply(a,a+7,sum_op); cout << sum_op.value() << endl;
    apply(a,a+7,prod_op); cout << prod_op.value() << endl;
    apply(a,a+7,Negate());
    apply(a,a+7,Print());
    char c; cin >> c;
}

```

Kết hợp 2 bước tổng quát hóa

```
template <typename T, typename COMP>
T* find_elem(T* first, T* last, T k, COMP comp) {
    while (first != last && !comp(*first, k))
        ++first;
    return first;
}

template <typename T, typename OP>
void apply(T* first, T* last, OP& op) {
    while (first != last) {
        op(*first);
        ++first;
    }
}
```

Khuôn mẫu lớp cho các đối tượng hàm

```
template <typename T> struct Greater{
    bool operator()(const T& a, const T& b)
    { return a > b; }
};

template <typename T> struct Less{
    bool operator()(const T& a, const T& b)
    { return a > b; }
};

template <typename T> class Sum {
    T val;
public:
    Sum(const T& init = T(0)) : val(init) {}
    void operator()(const T& k) { val += k; }
    T value() const { return val; }
};
```

```

template <typename T> struct Negate {
    void operator()(T& k) { k = -k;}
};

template <typename T> struct Print {
    void operator()(const T& k) { cout << k << ' ' ;}
};

void main() {
    int a[] = { 1, 3, 5, 2, 7, 9, 6 };
    int* alast = a+7;
    int* p1 = find_elem(a,alast,4,Greater<int>());
    int* p2 = find_elem(a,alast,4,Less<int>());
    if (p1 != alast) cout << "\nFirst number > 4 is " << *p1;
    if (p2 != alast) cout << "\nFirst number < 4 is " << *p2;
    Sum<int> sum_op; apply(a,a+7,sum_op);
    cout<< "\nSum of the sequence " << sum_op.value() << endl;
    apply(a,a+7,Negate<int>());
    apply(a,a+7,Print<int>());
    char c; cin >> c;
}

```

10.3 Tổng quát hóa truy lặp phần tử

- Vấn đề 1: Một thuật toán (tìm kiếm, lựa chọn, phân loại, tính tổng, ...) áp dụng cho một mảng, một vector, một danh sách hoặc một cấu trúc khác thực chất chỉ khác nhau ở cách truy lặp phần tử
- Vấn đề 2: Theo phương pháp truyền thống, để truy lặp phần tử của một cấu trúc "container", nói chung ta cần biết cấu trúc đó được xây dựng như thế nào
 - Mảng: Truy lặp qua chỉ số hoặc qua con trỏ
 - Vector: Truy lặp qua chỉ số
 - List: Truy lặp qua quan hệ móc nối (sử dụng con trỏ)
 - ...

Ví dụ thuật toán copy

- Áp dụng cho kiểu mảng thô

```
template <class T> void copy(const T* s, T* d, int n) {  
    while (n--) { *d = *s; ++s; ++d; }  
}
```

- Áp dụng cho kiểu Vector

```
template <class T>  
void copy(const Vector<T>& s, Vector<T>& d) {  
    for (int i=0; i < s.size(); ++i) d[i] = s[i];  
}
```

- Áp dụng cho kiểu List

```
template <class T>  
void copy(const List<T>& s, List<T>& d) {  
    ListItem<T> *sItem=s.getHead(), *dItem=d.getHead();  
    while (sItem != 0) {  
        dItem->data = sItem->data;  
        dItem = dItem->getNext(); sItem=sItem->getNext();  
    }  
}
```


Ví dụ thuật toán find_max

- Áp dụng cho kiểu mảng thô

```
template <typename T> T* find_max(T* first, T* last) {  
    T* pMax = first;  
    while (first != last) {  
        if (*first > *pMax) pMax = first;  
        ++first;  
    }  
    return pMax;  
}
```

- Áp dụng cho kiểu Vector

```
template <typename T> T* find_max(const Vector<T>& v) {  
    int iMax = 0;  
    for (int i=0; i < v.size(); ++ i)  
        if (v[i] > v[iMax]) iMax = i;  
    return &v[iMax];  
}
```

- Áp dụng cho kiểu List (đã làm quen):

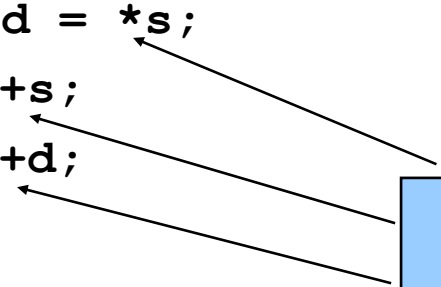
```
template <typename T>
ListItem<T>* find_max(List<T>& l) {
    ListItem<T> *pItem = l.getHead();
    ListItem<T> *pMaxItem = pItem;
    while (pItem != 0) {
        if (pItem->data > pMaxItem->data) pMaxItem = pItem;
        pItem = pItem->getNext();
    }
    return pMaxItem;
}
```

- ➡ Cần tổng quát hóa phương pháp truy lặp phần tử!

Bộ truy lặp (iterator)

- Mục đích: Tạo một cơ chế thống nhất cho việc truy lặp phần tử cho các cấu trúc dữ liệu mà không cần biết chi tiết thực thi bên trong từng cấu trúc
- Ý tưởng: Mỗi cấu trúc dữ liệu cung cấp một kiểu bộ truy lặp riêng, có **đặc tính tương tự như một con trỏ** (trong trường hợp đặc biệt có thể là một con trỏ thực)
- Tổng quát hóa thuật toán **copy**:

```
template <class Iterator1, class Iterator2>
void copy(Iterator1 s, Iterator2 d, int n) {
    while (n-- > 0) {
        *d = *s;
        ++s;
        ++d;
    }
}
```



Các phép toán áp dụng được tương tự con trỏ

- Tổng quát hóa thuật toán **find_max**:

```
template <typename ITERATOR>
ITERATOR find_max(ITERATOR first, ITERATOR last) {
    ITERATOR pMax = first;
    while (first != last) {
        if (*first > *pMax) pMax = first;
        ++first;
    }
    return pMax;
}
```

Các phép toán áp dụng
được tương tự con trỏ

Bổ sung bộ truy lặp cho kiểu Vector

- Kiểu Vector lưu trữ dữ liệu dưới dạng một mảng => có thể sử dụng bộ truy lặp dưới dạng con trỏ!

```
template <class T> class Vector {
    int nelem;
    T* data;
public:
    ...
    typedef T* Iterator;
    Iteratator begin() { return data; }
    Iteratator end()   { return data + nElem; }
};

void main() {
    Vector<double> a(5,1.0),b(6);
    copy(a.begin(),b.begin(),a.size());
    ...
}
```

Bổ sung bộ truy lặp cho kiểu List

```
template <class T> class ListIterator {
    ListItem<T> *pItem;
    ListIterator(ListItem<T>* p = 0) : pItem(p) {}
    friend class List<T>;
public:
    T& operator*() { return pItem->data; }
    ListIterator<T>& operator++() {
        if (pItem != 0) pItem = pItem->getNext();
        return *this;
    }
    friend bool operator!=(ListIterator<T> a,
                           ListIterator<T> b) {
        return a.pItem != b.pItem;
    }
};
```

Khuôn mẫu List cải tiến

```
template <class T> class List {  
    ListItem<T> *pHead;  
public:  
    ...  
    ListIterator<T> begin() {  
        return ListIterator<T>(pHead) ;  
    }  
    ListIterator<T> end() {  
        return ListIterator<T>(0) ;  
    }  
};
```

Bài tập về nhà

- Xây dựng thuật toán sắp xếp tổng quát để có thể áp dụng cho nhiều cấu trúc dữ liệu tập hợp khác nhau cũng như nhiều tiêu chuẩn sắp xếp khác nhau. Viết chương trình minh họa.
- Xây dựng thuật toán cộng/trừ/nhân/chia từng phần tử của hai cấu trúc dữ liệu tập hợp bất kỳ. Viết chương trình minh họa.