Nguyễn Hữu Thắng

Mã sinh viên:

Trường đại học Phương Đông

***Cấu trúc dữ liệu và giải thuật***

*Mục lục*

Phần 1: *Giải thuật*

*Chương 1: Giới thiệu*

Các ký hiệu được dùng trong sách này:

Phần A: cỡ chữ 24

Chương A: cỡ chữ 20

1.1 ABC: Cỡ 18

Chương 2: *Giải thuật sắp xếp và tìm kiếm*

1. Các giải thuật sắp xếp

Input:

Output:

1. Sắp xếp chọn

Tư tưởng: Ở các bước thứ (i=0 -> n-2) ta tìm phần tử nhỏ nhất trong đoạn phần tử a [i]...a[n-1] rồi trao đôur chỗ của nó với phần tử đứng đầu của đoạn đang xét.

1. Sắp xếp tròn (infection\_sort)
2. Tư tưởng

Giả sử đoạn đầu của mảng (từ a[0]...a[i-1]) đã được sắp . Để không mất tính tổng quát ta giả sử i chạy từ 1. Ở bước thứ i ta tìm vị trí thích hợp trong đoạn đầu để chèn a [i] vào sao cho đoạn mới vẫn được sắp .

* Tìm vị trí thích hợp để tiến hành chèn a [i] như sau:

+) x = a[i]

+) chọn một j chạy từ i-1 về đầu mảng, cứ gặp a[j]>x => đẩy lùi a[j] về sau giảm j đi 1 đến khi gặp a[j]<x hoặc j<0 thì dừng và đặt x vào vị trí j+1

1. Giải thuật

*Void insertion\_sort(int a[], int n)*

*{*

*Int i, j, x;*

*for (i=1; i<n; i++)*

*{*

*x=a[j]; j=i-1;*

*while (j>=0 && a[j]>x)*

*{*

*a[j+1]*

*}*

*}*

*}*

1. Áp dụng

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 | 14 | 15 | 78 | 21 | 67 | 85 |

Bước 1: Tìm vị trí thích hợp để chèn a[1]

* X=a[1]=14
* J=0 ta thấy a[0]>x => đẩy lùi a [0] (a[1]=a[0]=23)

J = -1<0 => dừng và đặt x vào vị trí j+ 1 (a[0]=x)

Mảng trở thành

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 14 | 23 | 15 | 78 | 21 | 67 | 85 |

Bước 2: tìm vị trí thích hợp để chèn a[2]

* X=a[2]=15
* J= 1 ta thấy a[1]> x => đẩy lùi a[1 ] về sau (a[2]=a[1]=23)

J=0 ta thấy a[0]<x => dừng => đặt x vào vị trí j + 1 (a[1]=x=15)

Mảng trở thành

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 14 | 15 | 23 | 78 | 21 | 67 | 85 |

Bước 3: tìm vị trí thích hợp để chèn a[3]

* X=a[3]=78
* J=2 ta thấy a[2]<x => dừng vào vị trí j+1 (a[3]=x)

1. Giải thuật sắp xếp nổi bọt (bubble\_sort)
2. Tư tưởng: Đổi chỗ liên tiếp

*Bước1:* đi từ a[0]-> a[n-1] cứ gặp hai phần tử kề nhau mà không đúng trật tự thì đổi chỗ. Sau b1 thì ta nhân được phần tử a[n-1] lớn nhất.

*Bước 2:* đi từ a[0]-> đến a[n-2] và trao đổi 2 ptu kề nhau không đúng trật tự. Sau bước 2 ta được a[n-2] là lớn nhất đoạn a[0]...a[n-2]

Cứ như vậy đến khi mảng được sắp.

1. Ví dụ áp dụng

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| 23 | 56 | 21 | 15 | 45 | 17 | 16 |

Bước 1

23 56

23 21 56

23 21 15 56

23 21 15 45 56

23 21 15 45 17 56

23 21 15 45 17 16 56 a[n-1]

Bước 2

21 23

21 15 23 45

21 15 23 17 45

21 15 23 17 16 45 a[n-2]

1. Giải thuật

void b\_s (int a[], int n)

{

I,j;

for (i=1; i<n-1;i++){

for (j=0;j<n-1;j++){

if (a[j]>a[j+1])

traodoi(a[j],a[j+1])

}

BTVN: 1) Viết chương trình nhận vào mảng n số nguyên sắp xếp mảng bằng 1 in 3 phương pháp, in mảng trước và sau khi sắp.

2) Viết chương trình nhận danh sách n sinh viên gồm họ tên mã số và điểm tb, sắp xếp danh sách sv theo điểm trung bình tăng dần.

*Bài làm*

1)

Sắp xếp nổi bọt (bubble\_sort)

#include <iostream>

//hàm sắp xếp mảng bằng bubble sort

void bubble\_sort(int a[]; int n)

{

    for(i=1; i<n-1;i++ )

    {

        for(j=0;j<n-1;j++ )

        {

            // so sánh hai phần tử liền kề và đổi chỗ nếu cần

            if(a[j]>a[j+1])

            {

               int doicho = a[j];

               a[j]=a[j+1];

               a[j+1]=doicho;

            }

        }

    }

}

int main()

{

int n;

  //Nhập số lượng phần tử của mảng

  std::cout<<"Nhap so luong phan tu cua mang:\n";

  std::cin>>n;

int a[n];

//Nhập các phần tử của mảng

std::cout<<"Nhap"<<n<<"so nguyen:\n";

    for(int i=0; i<n; i++)

      {

          std::cin>>a[i];

      }

//In mảng ra trước khi sắp xếp

std::cout<<"Mang truoc khi sap xep:";

  for(int i=0; i<n; i++)

  {

    cout<<a[i]<<" "<<end;

  }

//Gọi hàm Bubble sort để sắp xếp

bubble\_sort(a, n);

cout<<"Mang sau khi sap xep: "<<end;

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

 {

   cout<<a[i]<<end;

 }

return 0;

}

Cách 2:sắp xếp chọn

void selectionSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Hoán đổi giá trị của phần tử nhỏ nhất với phần tử đầu tiên của dãy chưa sắp

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

2)

#include <iostream>

#include <string>

// Định nghĩa một cấu trúc để lưu thông tin của sinh viên

struct Student {

std::string hoTen;

int maSo;

double diemTrungBinh;

};

// Hàm để thực hiện sắp xếp nổi bọt

void bubbleSort(Student arr[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j].diemTrungBinh > arr[j + 1].diemTrungBinh) {

// Hoán đổi vị trí của hai sinh viên nếu điểm trung bình của họ không đúng thứ tự

Student temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int n;

std::cout << "Nhập số lượng sinh viên: ";

std::cin >> n;

Student danhSachSinhVien[n];

// Nhập thông tin của từng sinh viên

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

std::cout << "Nhập thông tin cho sinh viên " << i + 1 << ":\n";

std::cout << "Họ tên: ";

std::cin.ignore(); // Loại bỏ ký tự '\n' còn lại sau khi nhập n số nguyên

std::getline(std::cin, danhSachSinhVien[i].hoTen);

std::cout << "Mã số: ";

std::cin >> danhSachSinhVien[i].maSo;

std::cout << "Điểm trung bình: ";

std::cin >> danhSachSinhVien[i].diemTrungBinh;

}

// Sử dụng bubble sort để sắp xếp danh sách sinh viên theo điểm trung bình

bubbleSort(danhSachSinhVien, n);

// In danh sách sinh viên sau khi sắp xếp

std::cout << "Danh sách sinh viên sau khi sắp xếp theo điểm trung bình tăng dần:\n";

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

std::cout << "Họ tên: " << danhSachSinhVien[i].hoTen << std::endl;

std::cout << "Mã số: " << danhSachSinhVien[i].maSo << std::endl;

std::cout << "Điểm trung bình: " << danhSachSinhVien[i].diemTrungBinh << std::endl;

}

return 0;

}

Nhập chuỗi có dấu cách cin.get(s,n): Nhập dl cho biến số tối đa n-1 ký tự bao gồm cả dấu cách.

Lệnh cin.get(s,n) Lấy các ký tự ở dòng vào stdin đưa vào s, nếu ở stdin có ký tự enter, thì ký tự này cũng sẽ được đưa vào s => trôi lệnh nhập chuỗi

Để khắc phục: ta cho thêm lệnh cin.get() trước lệnh cin.get(s,n); để loại ký tự enter ra khỏi stdin

1. Các giải thuật tìm kiếm
2. Tìm kiếm tuần tự

Input: một dãy khóa chưa được sắp xếp, khóa k cần tìm

Output: vị trí tìm thấy

1. Tư tưởng: So sánh khóa k cần tìm với lần lượt từng phần tử trong dãy khóa ban đầu. Nếu có a[i] = k => return i; Nếu đã đi hết dãy mà không tìm thấy return -1.
2. Giải thuật

Timtuantu

Mảng a vị trí tìm thấy

Khóa k

*Int timtuantu(int a[], int n, int k)*

*{*

*Int i;*

*For(i=0;i<n;i++)*

*If(a[i]==k) return i;*

*Return -1;*

*}*

1. *Phương pháp tìm kiếm nhị phân*

*int: Dãy khóa đã được sắp xếp (g/s tăng dần ); khóa k cần tìm.*

*output: Vị trí tìm thấy*

1. *Tư tưởng: So sánh khóa k cần tìm với phần tử đứng ở giữa mảng*

*+) Nếu k = a[giua]=> tìm thấy ở vị trí giữa*

*+) Nếu k<a[giua]=> tìm ở nữa trái*

*+) Nếu k>a[giua]=> tìm ở nữa phải*

*Khi tìm ở nữa trái hoặc nữa phải ta lại áp dụng đệ quy tư tưởng trên*

*Gọi left và right là chỉ số của phầ tử bên trái và bên phải ngoài cùng của đoạn chúng ta xét.*

1. *Ví dự*

*Cho dãy khóa sau:*

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *45* | *55* | *56* | *67* | *78* | *79* | *81* | *83* | *88* | *100* |

*(k=81)*

*left=45; right=100*

*k>a[mid]*

1. Giải thuật tìm nhị phân sử dụng vòng lặp

Int timnp(int a[]; int n, int k)

{

Int left, right, mid;

Left = 0; right = n-1;

While(left<=right)

{

Mid=(left+right)/2;

If(k==a[mid]) return mid;

{

Else if(k<a[mid]) right=mid-1;

Else left = mid+1;

}

}

}

Giair thuật đệ qui:

1. Khái niệm: Nếu lời giải của bài toán T được thực hiện bởi lời giải của bài toán T’ CÓ DẠNG T nhưng kích thước nhỏ hơn T. Đến lượt mình lời giải của bài toán T’ LẠI được thực hiện bởi lời giải của bài toán T’’ có dạng giống T’ nhưng kích thước nhỏ hơn. Cứ như vậy cho đến khi tìm đc bài toán T’n đủ nhỏ , dễ dàng giải và đưa ra kết quả.

Giải thuật tương ứng với lời giải của bài toán T như trên đc gọi là GTDQ

Ví dụ: 5!= 1\*2\*3\*4\*5 = 120

4!\*5

3!\*4

2!\*3

1!\*2

= 1

1. Hàm đệ quy

HDQ là hàm là hàm mà bên trong thân của nó có lời gọi tới chính nó với phạm vi tham số nhỏ Trong hàm đệ quy gồm 2 thành phần chính

+) Phân neo (kết thúc đệ qui) : TRả về giá trị cho hàm tại 1 hay nhiều giá trị của tham số

+) Phần đệ qui là lời gọi tới chính nó

Viết hàm đệ qui tính n!

5, 4,3,2,1

Long int giaithua(int n)

{

If (n==1 || n==0) return 1; //điểm neo

Else 4

Return n\*giaithua(n-1);

} 5

Lời gọi đệ qui

Mô phỏng giaithua(5)

Return 5\*giaithua(4)

Return 4\*giaithua(3)

Return 3\*giaithua(2)

Return 2\*giaithua(1)

Return 1

Viết hàm đệ qui tìm nhị phân (mảng a đã được sắp xếp tăng dần ; k là khóa cần tìm)

int timnp(int left, int right)

{

If(left>right) return -1; //ko tìm thấy(điểm neo)

Else

{

Mid=(left+right)/2;

If (k==a[mid]) return mid;// điểm neo

Else if (k<a[mid]) right = mid-1;

Else left=mid+1;

Return timnp(left, right); //lời gọi đến chính nó

}

}

Chương 3: Cấu trúc dữ liệu danh sách

Danh sách: là tập hợp có các thứ tự các phần tử có cùng kiểu dữ liệu.

* Cài đặt danh sách bằng mảng hoặc bằng con trỏ

Các phép toán trên danh sách :

* Khởi tạo danh sách rỗng
* Kiểm tra ds có rỗng / có đầy ko
* Thêm một phần tử vào danh sách (thêm từ đầu, thêm từ cuối, thêm từ giữa)
* Lấy một phần tử ra khỏi ds (Lấy từ đầu, lấy từ cuối, Lấy từ giữa)

1. Danh sách kiểu ngăn xếp (stack)
2. Khái niệm: Ngăn xếp là 1 loại ds đặc biệt trong đó phép thêm 1 phần tử vào và lấy 1 ptuwr ra đều đc thực hiện ở đỉnh của ngăn xếp (top).

* Ngăn xếp hoạt động theo cơ chế LIFO (vào sau ra trc – last in first out)

1. Cài đặt ngăn xếp bằng mảng

Phần

Thêm một phần tử vào hàng đợi (thêm từ cuối)

+) nếu 1 hàng đang rỗng, sau khi thêm sẽ trở thành hàng có 1 phần tử Q.front=Q.rear=0

+) Nếu hàng không rỗng => thêm vào cuối Q.rear++

Void add\_Q(item Type x, Queue & Q)

{

If (is\_full (Q)==0)

{

If (is\_empty(Q)==1)

{ Q.front=Q.rear=0;

}

Else{Q.rear++;}

Q.phantu[Q.rear]=x;

}

}

//Lấy 1 phần tử ra khỏi hàng (lấy từ đầu)

Khi hàng không rỗng mới lấy

+) Khi hàng đang có một phần tử chúng ta đưa hàng về rỗng

+) Khi hàng đang có nhiều hơn một phần tử : Q.front++

Void delete Q (Queue &Q, item Type &x)

{

If (is\_empty(Q)==0)

{

X=Q.phantu[Q.front];

If (Q.front==Q.rear) khoitao(Q);

}

CÀI ĐẶT NGĂN XẾP, HÀNG ĐỢI BẰNG CON TRỎ

Khai báo bến x y

5

int x,y;

7

lcnslanl

2 byte

2 byte

X=5,y=7;

Con trỏ: là một biến mà chứa địa chỉ của biến khác

Khai báo con trỏ: TenkieuDL\*tenbiencontro;

Xin cấp bộ nhớ cho con trỏ: tenbiencontro=new TenkieuDL;

Truy cập vào thành phần con trỏ :

Tenbiencontro -> tenthanhphan

Cài đặt ngăn xếp bằng con trỏ:

Data

Data

Data

NULL

Top

struct Phantu

{

itemType data;

Phantu\* next; //con trỏ next chứa địa chỉ của phần tử tiếp theo

};

Struct Stack

{

Phantu\* top; //con trỏ top chứa địa chỉ ptu ở đỉnh

};

//khởi tạo một ngăn xếp rổng

Void khoitao (Stack &S)

{

S.top=NULL;

}

//kiem tra rỗng

int kt\_rong(Stack S)

{

If (S.top==NULL) return 1;

Else return 0;

}

//Thêm phần tử vào ngăn xếp

S.top

Temp NULL

X

S.top

Bài tập

Viết chương trình nhập từng sinh viên từ bàn phím (mỗi sinh viên gồm các thông tin ms, hoten, đtb). Đưa vào hàng đợi đến khi ko muốn nhập nữa. Lấy từ hàng đợi và hiển thị

#include<iostream>

struct sinhvien

{

    char ms[10], hoten[30];

    float dtb;

};

struct Phantu

{

    sinhvien data;

    Phantu\*next;

};

struct Queue

{

    Phantu\* front;

    Phantu\* rear;

};

//Khoi tao

void khoitao(Queue\*Q)

{

    Q.front=NULL; Q.rear=NULL;

}

int ktrong(Queue Q)

{

    if(Q.front==NULL && Q.rear==NULL) return 1;

    else return 0;

}

void them(Phantu\* temp, Queue&Q)

{

    //Khi hang dang rong

    if (ktrong(Q)==1)

    {Q.front=temp; Q.rear=temp}

    //khi hang ko rong

    else

    {

        Q.rear->next=temp; Q.rear=temp;

    }

}

void layra(Queue& Q, sinhvien &x)

{

    if(ktrong(Q)==0)

    {

     q=Q.front; x=p->data;

     if(Q.front==Q.rear)

        khoitao(Q);

    else

       Q.rear=p->next;

       delete (p);

    }

}

int main()

{

    sinhvien sv;

    Queue Q; char tloi;

    Phantu\*temp;

    khoitao(Q);

    do

    {

        cout<<"Nhap masv="; cin>>sv.maso;

        cout<<"Nhap hoten="; cin.get(); cin.get(sv.hoten,30);

        cout<<"Nhap dtb=";cin>>sv.dtb;

        temp=new Phantu;

        temp->data = sv;

        temp->next = NULL;

        them(temp,Q);

        cout<<"Ban co muon tiep tuc khong(c/k)?"; cin>>tloi;

    }

    while (tloi=='c'||tloi=='C')

    cout<<"Danh sach sv theo thu tu ban dau:\n";

    while(ktrong(Q)==0)

    {

        layra(Q,sv);

        cout<<sv.maso<<"\t"<<sv.hoten<<"\t"<<sv.dtb;

    }

}

CHƯƠNG 4: CTDL hình cây

* 1. Định nghĩa đệ qui về cây

Tập hợp có 1 ptu gọi là cây (Phần tử của cây còn gọi là nút (node) hoặc đỉnh).Nút duy nhất này của cây là gốc (Root) của cây

Nếu các nút n1,n2,... Lần lượt là gốc của các cây T1, T2,.., Tk Thì 1 cây T sẽ được hình thành bằng cách cho n làm cha của n1, n2,...,nk

1

T1 2 2 T2 2 Tk

Cấp: Của 1 nút là số nút con của nút đó

Cấp của cây là cấp của nút có nút cao nhất . Các nút có cấp bằng ko (ko có con ) => nút lá

Mức : Nút gốc có mức là 1; nếu nút cha có mức là i thì nút con có mức là i+1 ; Các nút có cùng mức => Gọi là anh em (siblings )

Mức của cây là mức của nút có mức cao nhất (gọi là chiều cao của cây )

II. Cây nhị phân

Là cây mà mỗi nút chỉ có tối đa 2 nút con, phân biệt nút con trái và nút con phải.

Cây nhị phân tìm kiếm: Là cây nhị phân mà mỗi nút thuộc cây con trái bé hơn nút gốc, Mỗi nút thuộc cây con phải lớn hơn nút gốc.

BT: Viết chương trình đọc dãy số từ từ tệp, đưa vào cây nhị phân tìm kiếm, duyệt cây theo thứ tự trước, giữa, sau. Đưa ra kq ra màn hình

BT: SX mảng, sd cây nhị phân tìm kiếm

Hàm main: - Đọc dãy số từ tệp (đọc từng phần tử); đẩy vào cây; duyệt cây theo thứ tự giữa (duyệt đến đâu đẩy vào mảng đó)

Giải thuật sắp xếp nhanh (Được đề xuất bởi Hoare)

1. Tư tưởng: Dựa theo chiến thuật chia để trị

Chọn 1 phần tử àm mốc sau đó đưa hết những phần tử nhỏ hơn mốc sang bên tay trái (giả sử xắp xếp tăng dần) , Những phần tử lớn hơn mốc sang phải. Sau quá trình phân hạch này ta thu được 2 nữa; Nửa trái gồm toàn những phần tử bé hơn mốc và nửa phải toàn phần tử lớn hơn mốc

Tiếp tục áp dụng đệ quy tư tưởng trên để phân hạch hai nửa.

Việc phân hạch a[left]...a[right] được tiến hành như sau:

+) Chọn phần tử ở giữa làm mốc

+) Cho một biến i đi từ bên trái sang (i xuất phát từ left) cứ gặp a[i]< mốc thì tăng i lên 1

+) Cho 1 biến j đi từ bên phải sang xuất phát từ right cứ gặp a[j]> mốc thì giảm j đi 1

Đến khi dừng i và j thì kiểm tra

Nếu i<=j thì trao đổi (a[i],a[j]) tiếp tục tăng i và giảm j đi 1 Đến khi i>=j thì DỪNG => ta thu được phân hạch a[left]...a[i]...a[right]

1. Giải thuật

Void quicksort( int left, int right)

{

Int i,j,moc,mid;// chỉ số phần tử đứng giữa mid

I=left; j=right; mid=(left+right)/2;

Moc=a[mid];

do

{

While(a[i]<x) i++;

While (a[j]>x)j--;

If(i<=j)

{

Swap(a[i], a[j])

I++; j--;

}

}while (i<j); if(left<j) quicksort(left,j); if(i<right) quicksort(i,right);

}

1. Ví dụ: sắp xếp mảng sau tăng dần theo pp quicksort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 45 | 23 | 47 | 42 | 40 | 78 | 41 |

Left right

Mốc là a[3]

+) Xuất phát từ trái sang a[0]=i (i>moc -> stop) => Xuất phát từ bên phải a[6] = j (j<moc -> stop)

Trao đổi giữa a[0] và a[6] (swap (a[i], a[j]))

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

Tăng i lên 1 lớn hơn mốc thì dừng a[1]->a[2]

Sang trái j lớn hơn mốc thì giảm 1 a[5]->a[4]

Trao đổi giữa a[2] và a[4] (swap (a[i], a[j]))

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 41 | 23 | 40 | 42 | 47 | 78 | 45 |

Moc=i,j => STOP

Left a[0] và right a[6]

Quicksort(0,3);

Quicksort(3,6);

Bài tập về nhà: Viết ctrinh đọc 1 mảng từ tệp sau đó sắp xếp mảng bằng phương pháp sắp xếp nhanh

*PHẦN TÀI LIỆU THAM KHẢO:*

Phần 1 : GIẢI THUẬT

Chương 1: Mở đầu

1.1 Giải thuật và cấu trúc dữ liệu

Giải thuật là các câu lệnh chặt chẽ và rõ ràng xác định một trình tự các thao tác trên cùng một số đối tượng nào đó sao cho sau một số hữu hạn bước thực hiện ta đạt được kết quả mong muốn.

Giải thuật là phép xử lý chính những dữ liệu và chúng biểu thị cho nhữn thông tin của bài toán:

* Các dữ liệu đưa vào, các kết quả trung gian,...

Như vậy: giữa CTDL và GT có mối quan hệ mật thiết. Có thể coi chúng như hình với bóng. Không thể nói tới cái này mà không nhắc tới cái kia.

Chính vì lẽ đó nghiên cứu CTDL đi đôi với việc xác lập các giải thuật xửu lý trên cấu trúc ấy.

* + 1. Cấu trúc dữ liệu và các vấn đề liên quan

1.2.1

Dữ liệu

Các phần tử cơ sở (Dữ liệu nguyên tử )

VD: Chữ số, một số ký tự,...

Tùy thuộc vào từng bài toán

* 1. Ngôn ngữ diễn đạt
* Phải luôn luôn tuân thủ các quy tắc chặt chễ về cú pháp của ngôn ngữ đo khiến cho việc trình bài về GT và cấu trúc dữ liệu có thiên hướng nặng nề và gò bó.
* Phải phụ thuộc vào CTDL tiền định của ngôn ngữ nên có lúc không thể hiện được đày đủ các ý về cấu trúc mà ta muốn biểu thị.
* Ngôn ngữ nào được chọn cũng không hẳng đã được mọi người yêu thích mà muốn sử dụng.
  + 1. Ký tự và biểu thức
* Giá trị logic
* Dấu phpes toán logic
* Tên biến
* Biến chỉ số có dạng a[i]; a[j]...
  1. Các câu lệnh chỉ thị
     1. Câu lệnh gán

Có lệnh là V:= E

Với V chỉ tên biến, tên hàm

E chỉ biểu thức

* + 1. Câu lệnh ghép

Có dạng:

Begin S1, S2,...; Sn; end

Với Sn, i=1,....,n là các câu lệnh.

Nó cho phép ghép nhiều câu lệnh lại đẻ được coi như một câu lệnh.

* + 1. Câu lệnh điều kiện

Có dạng

If B then S

Với B là biều thức logic

S là một câu lệnh khác

Có thể diễn tả bằng biểu đồ như sau:

S

True

False

Hoặc: if B then S1 else S2

S1

True

S2

False

* + 1. Câu lệnh tuyển

Case

B1:S1;

B2:S2;

....

....

Bn:Sn;

else: Sn+1

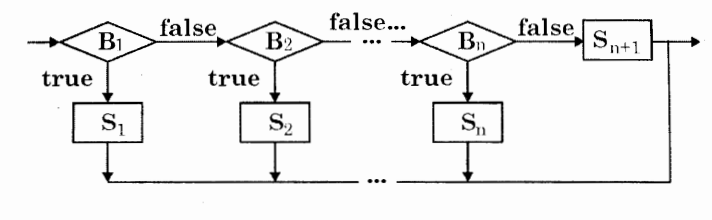
end case

Với B(i=1,2,..,n) là điều kiện

S(i=1,2,...,n) là các câu lệnh

\*Câu lệnh này cho phpes phân biệt các tình huống xử lý các tình huống xử lý khác nhau trong đièu kiệ khác nhau mà không phải dùng tới các câu lệnh if – then – else lồng nhau.

Có thể diễn giải bằng sơ đồ sau:



\*Và điểm linh động

else có thể không có mặt.

Si (i=1,2,...n) có thể được thay bằng một dãy các câu lệnh thể hirnj một dãy xử lý khi có điều kiện B mà không cần phải đặt giữa begin và end

* + 1. Câu lệnh lặp
* Với số lần lặp biết trước

for i:= m to n do S

Nhằm câu lệnh S với i lấy giá trị nguyên từ m tới n (n>= m), với bước nhảy tăng bằng 1.

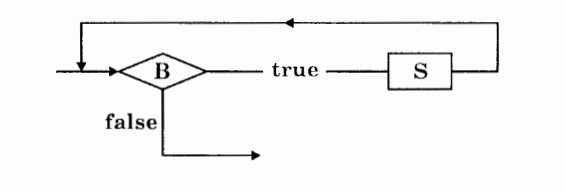
Hoặc:

for i := n down to m do S

tương tự như câu lệnh trên với bước nhảy giảm bằng 1.

* Với số lần lặp ko biết trước

While B do S



Chừng nào mà thực hiện B có giá trị bằng true thì thực hiện S,

Hoặc:

Repeat S until B

Lặp lại S cho tới khi B có giá trị bằng true (S có thể là 1 dãy lệnh)

* + 1. Câu lệnh chuyển

go to n (n là kí hiệu của 1 bước trong chương trình )

Thường người ta hạn chế việc dùng go to. Tuy nhiêu với mục đích diễn đạt cho tự nhiên, trong một chừng mực nào đó ta vẫn sử dụng.

* + 1. Câu lệnh vào ra

Có dạng:

read (<danh sách biến>)

write (<danh sách biến hoặc dòng kí tự>)

Các biến trong dah sách cách nhau bởi dấu phẩy

Dòng ký tự là 1 dãy các ký tự đặt giữa hai dấu nháy ‘ ‘.

* + 1. Câu lệnh kết thúc chương trình

end

* 1. Chương trình con
* Chương trình con hàm

Có dạng:

function <tên hàm> (<danh sách tham số>)

S1,S2,...; Sn

Return

Câu lệnh kết thúc chương trình con ở đay là return thay cho end

* Chương trình con thủ tục

Tương tự như trên chỉ khác ở chỗ :

Từ khóa procedure thay cho function

Trong cấu tạo của chương trình con hàm bao giờ cũng có câu lệnh gán mà tên hàm nằm ở vế trái. Còn đối với chương trình con thủ tục thì không có.

Lời gọi chương trình con hàm thể hiện bằng tên hàm cùng danh sách tham số thực sự nằm trong biểu thức. Còn đối với chương trình con thủ tục lời gọi được thể hiện bằng câu lệnh call có dạng:

Call <tên thủ tục >(<danh sách tham số thực sự>)

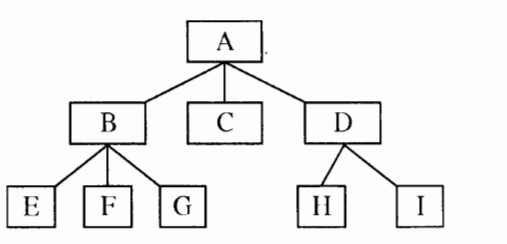
Chú ý: Chương trình diễn đạt một giải thuật ở đat phần khai báo dữ liệ được bỏ qua. Thay bởi phần mô tả CTDL bằng ngôn ngữ tự nhiên, mà ta sẽ nên ra trước khi bước vào phần giải thuật.

Chương 2: Thiết kế và phân tích giải thuật

2.1 Từ bài toán đến chương trình

2.1.1 Mô – đun hóa và việc giải quyết bài toán

Các bài ngày nay càng ngày càng phúc tạm. Các gải thuật có quy mô lớn và càng khó hiểu. Tuy nhiên mọi việc sẽ đơn giản nếu phân chia các bài toán lớn (mô-đun chính) thành các nhỏ (mô-đun con), như vậy sẽ được thể hiện theo một cấu trúc phân cấp, có dạng như sau:



Chiến thuật “chia để trị” người ta dùng cách thiết kế “từ đỉnh xuống”. Đi dần vào giải quyết các phần cụ thể một cách chi tiết hơn (từ khái quát đến chi tiết)

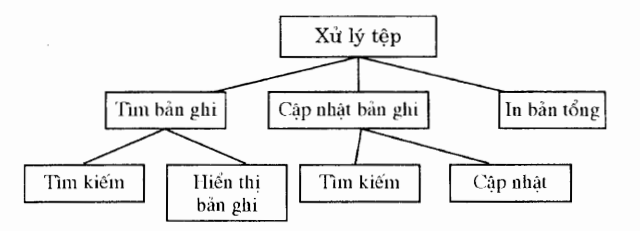
Ví dụ:

Hội đồng xét học bổng của trường yêu cầu:

“Dùng MT để quản lý và bảo trì các hồ sơ về học bổng của các sinh viên ở diện được tài trợ học bổng phải được lập báo cáo lên Bộ”

Giải thuật xử lý sẽ phải giải quyết ba nhiệm vụ chính như sau

1. Những thông tin về sv được học bổng , lưu trữ trên đĩa phải được đọc vào bộ nhớ trong để có thể xử lý (ta gọi là nhiệm vụ “đọc tệp”)
2. Xử lý các thông tin này để tạo ra được kết quả mông muốn (nhiệm vụ “xử lý tệp”)
3. Sao chép những thông tin đã được cập nhật vào tệp trên đĩa để lưu trữ cho việc xử lý sao này (nhiệm vụ “ghi tệp”)

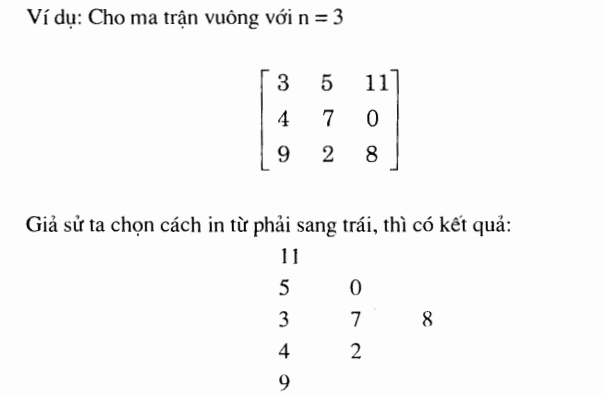
**

2.1.2Phương pháp tinh chế từng bước

Tinh chỉnh gứn liền với lâp trình . Nó phản ánh tinh thần mô đun hóa bài toán và thiết kế kiểu top-Down

Bắt đầu là trình bằng ngôn ngữ tự nhiên phản ánh đúng ý chính của công việc cần làm. Dần ta tinh chỉnh hướng về ngôn ngữ lập trình mà ta chọn. Người ta thường dùng pha cả ngôn ngữ tự nhiên lẫn ngôn ngữ lập trình, mà người ta còn gọi là giả ngôn.

Ví dụ 1: Cho một ma trận vuông nxn các số nguyên. Hãy in ra các phần tử thuộc các đường chéo song song và đường chéo chính



*Ta sẽ giải theo hướng C++*

*Giải thuật phác họa như sau:*

1. *Nhập n*
2. *Nhập các phần tử của ma trận*
3. *In các đường chéo song song với dường chéo chính*

*Lời giải*

**#include <iostream>**

**#include <cstdlib>**

**#include <ctime>**

**Using name std;**

**Int main ()**

**{**

**Scrand (time(0));**

**int n;**

**Cout<<”Nhap kich thuoc ma tran (n x n) :”;**

**Cin>>n;**

**Int\*\*matrix = new int\*[n]**

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

**{**

**Matrix[i] = new int[n];**

**for (int j=0; j<n; j++){**

**matrix[i][j] = rand() %100;**

**}**

**}**

**Cout<<”Ma tran”<<n<<”x”<<n<<”:\n‘ ;**

**for (int i = 0 ; i < n ; i++){**

**for (int j = 0; j <n ; j++)**

**cout <<matrix[i][j]<<”\t”**

**}**

**Cout<< ‘\n’ ;**

**}**

**Cout<<”Cac phan tut ren duong cheo chinh:\n”;**

**For (int i=0; i<n; i++){**

**Cout<<matrix[i][i]<<””;**

**}**

**Cout<<”Cac thanh phan tren duong cheo song song:\n”;**

**for (int i= 0; i<n; i++){**

**cout<<matrix[i][n – 1 - i]<<” ”;**

**}**

**Cout<<”\n”;**

**for (int I = 0; I < n; i++){**

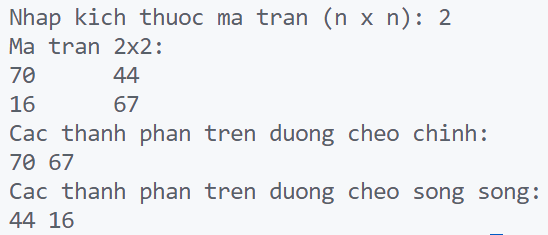
**delete[] matrix[i];**

**}**

**Delete [] matrix;**

**return 0;**

**}**

****

Khóa học cấu trúc dữ liệu và giải thuật – US San Diego

Phần 1: Hộp công cụ thuật toán

**Mục tiêu học tập**

* Thực hành triển khai thuật toán
* Các chương trình kiểm tra thực hành và kiểm tra căng thẳng
* So sánh chương trình nhanh và chậm
* Thực hành giải quyết các thách thức lập trình

Phần 1: Thuật toán

Chương 1: Phân tích thuật toán

Bài 1:THUẬT TOÁN VÀ CÁC VẤN ĐỀ LIÊN QUAN

Thuật toán được hiểu là sự đặc tả chính xác một dãy các bước có

thể thực hiện được một cách máy móc để giải quyết một vấn đề. Cần nhấn

mạnh rằng, mỗi thuật toán có một dữ liệu vào (Input) và một dữ liệu ra

(Output); khi thực hiện thuật toán (thực hiện các bước đã mô tả), thuật toán

cần cho ra các dữ liệu ra tương ứng với các dữ liệu vào.

Thuật toán yêu cầu tính đúng đắn, hiệu quả

1. Tính hiêu quả của thuật toán

Người ta thường xem xét thuật toán, lựa chọn thuật toán để áp dụng

dựa vào các tiêu chí sau:

1. Thuật toán đơn giản, dễ hiểu.

2. Thuật toán dễ cài đặt (dễ viết chương trình)

3. Thuật toán cần ít bộ nhớ

4. Thuật toán chạy nhanh

Thời gian chạy trong trường hợp xấu nhất (worst-case running time) của một thuật toán là thời gian chạy lớn nhất của thuật toán đó trên tất cả các dữ liệu vào cùng cỡ . Chúng ta sẽ ký hiệu thời gian chạy trong trường

hợp xấu nhất là T(n), trong đó n là cỡ của dữ liệu vào. Sau này khi nói tới

thời gian chạy của thuật toán chúng ta cần hiểu đó là thời gian chạy trong

trường hợp xấu nhất. Sử dụng thời gian chạy trong trường hợp xấu nhất để

biểu thị thời gian chạy của thuật toán có nhiều ưu điểm. Trước hết, nó đảm

bảo rằng, thuật toán không khi nào tiêu tốn nhiều thời gian hơn thời gian

chạy đó. Hơn nữa, trong các áp dụng, trường hợp xấu nhất cũng thường

xuyên xảy ra.

Chúng ta xác định thời gian chạy trung bình (average running time)

của thuật toán là số trung bình cộng của thời gian chạy của thuật toán đó trên

tất cả các dữ liệu vào cùng cỡ n. Thời gian chạy trung bình của thuật toán sẽ

được ký hiệu là Ttb(n). Đánh giá thời gian chạy trung bình của thuật toán là

công việc rất khó khăn, cần phải sử dụng các công cụ của xác suất, thống kê

và cần phải biết được phân phối xác suất của các dữ liệu vào. Rất khó biết

được phân phối xác suất của các dữ liệu vào. Các phân tích thường phải dựa

trên giả thiết các dữ liệu vào có phân phối xác suất đều. Do đó, sau này ít khi

ta đánh giá thời gian chạy trung bình.

**p. Do đó, chúng ta sẽ chỉ quan tâm**

**tới tốc độ tăng (rate of growth) của hàm T(n), tức là tốc độ tăng của thời**

**gian chạy khi cỡ dữ liệu vào tăng. Ví dụ, giả sử thời gian chạy của thuật toán**

**là T(n) = 3n2 + 7n + 5 (phép toán sơ cấp). Khi cỡ n tăng, hạng thức 3n2**

**quyết**

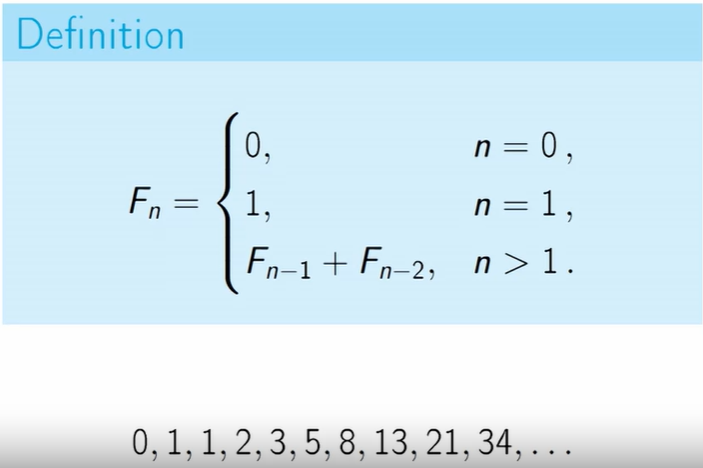
**định tốc độ tăng của hàm T(n), nên ta có thể bỏ qua các hạng thức khác và**

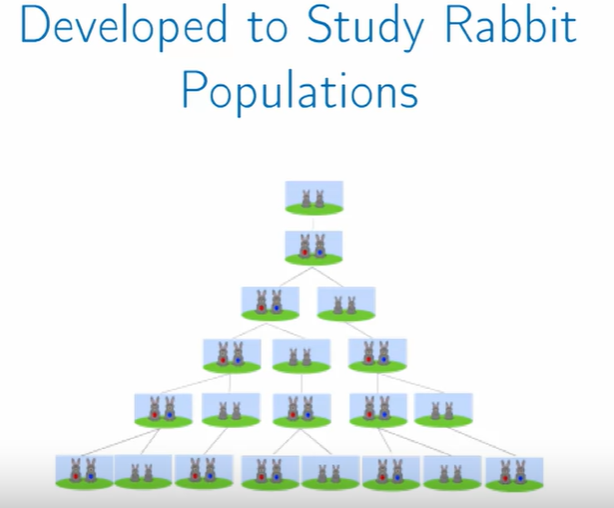
**có thể nói rằng thời gian chạy của thuật toán tỉ lệ với bình phương của cỡ dữ**

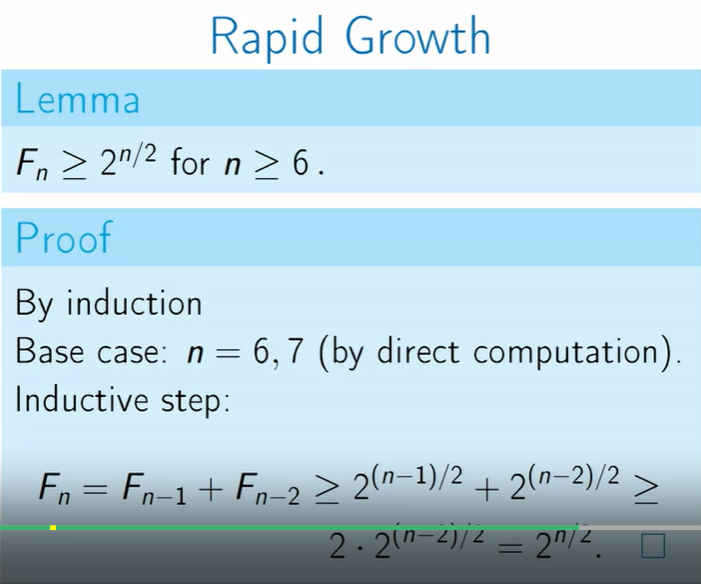
**liệu vào. Trong mục tiếp theo chúng ta sẽ định nghĩa ký hiệu ô lớn và sử**

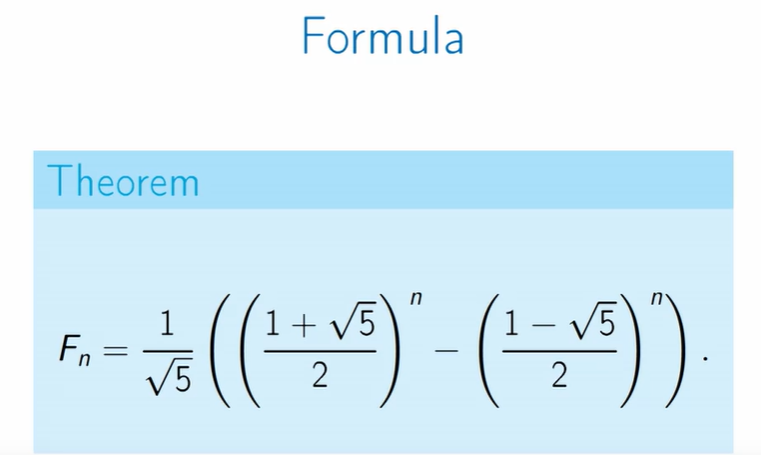
**dụng ký hiệu ô lớn để biểu diễn thời gian chạy của thuật toán.**

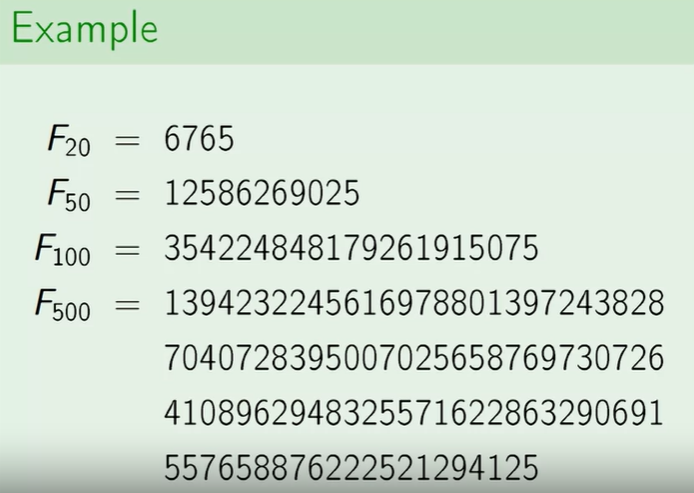
1. Số Fibanacci



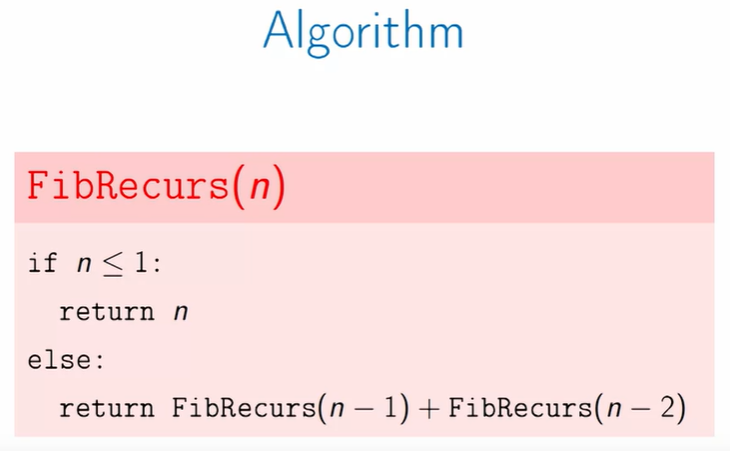


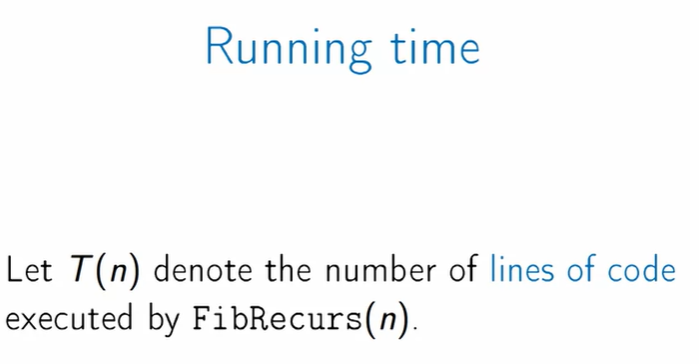


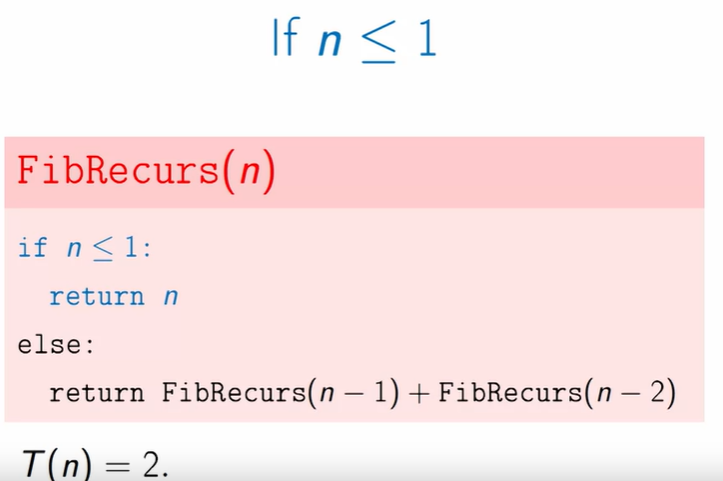


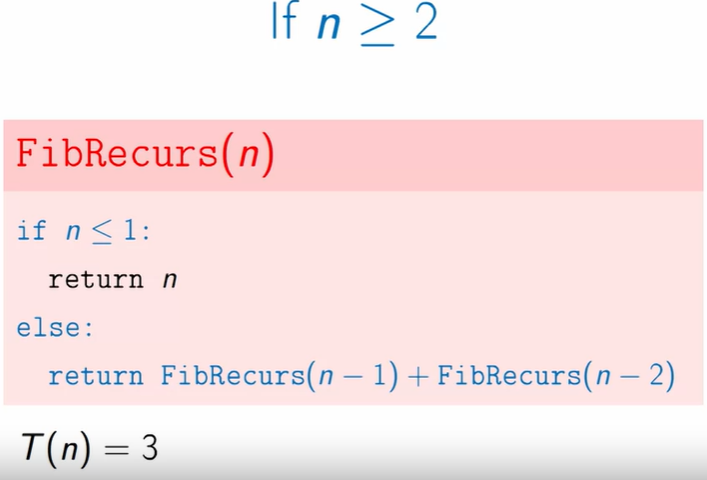


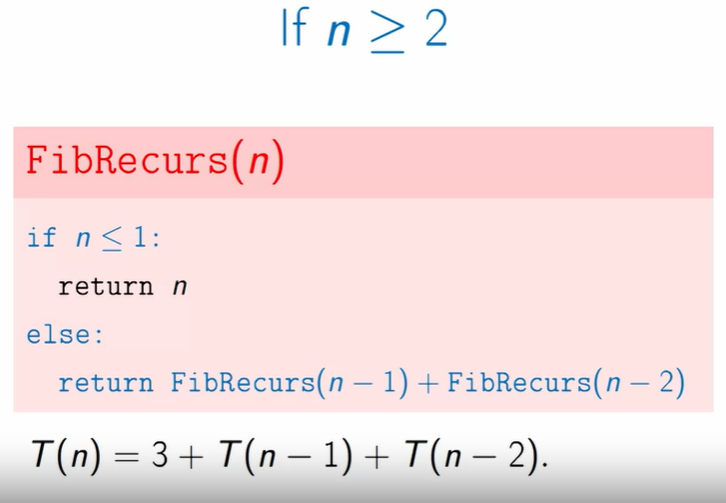
1. Naive Algorithm

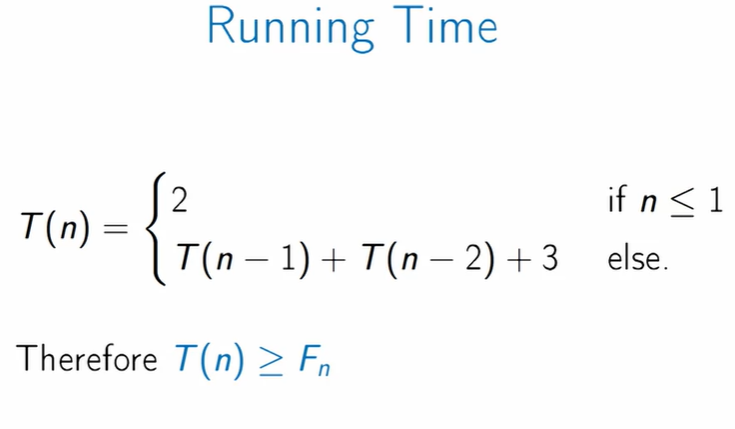


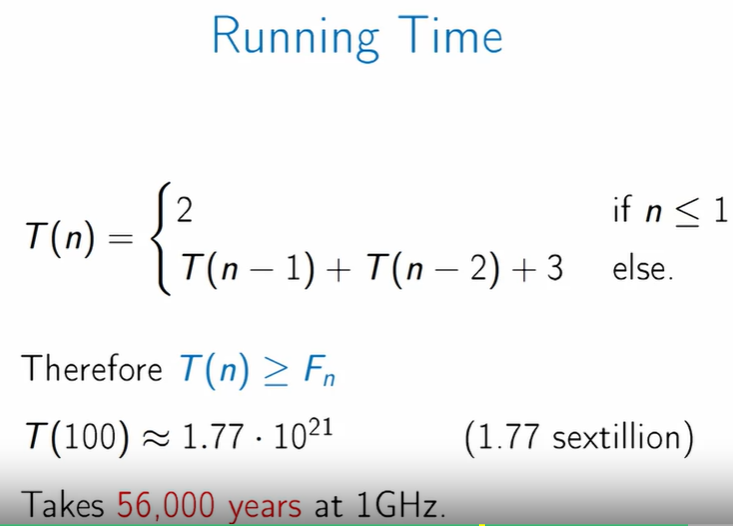


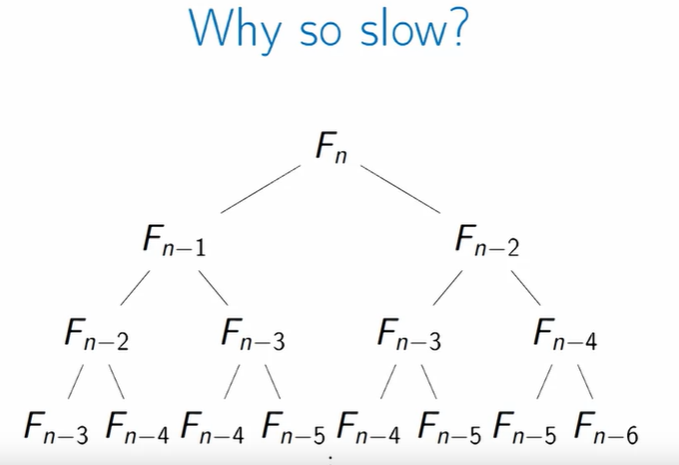


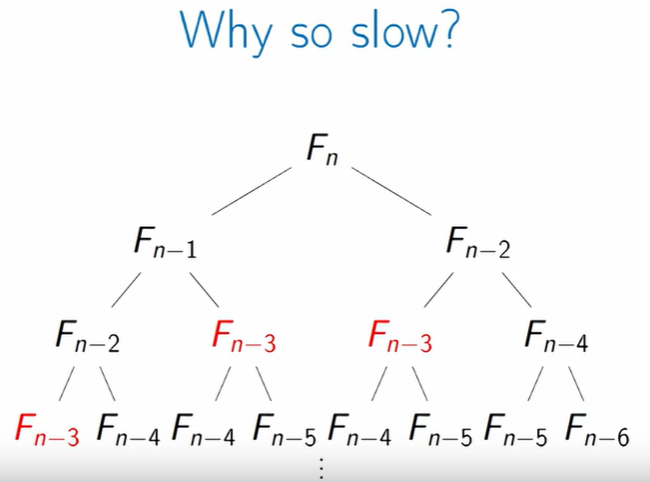


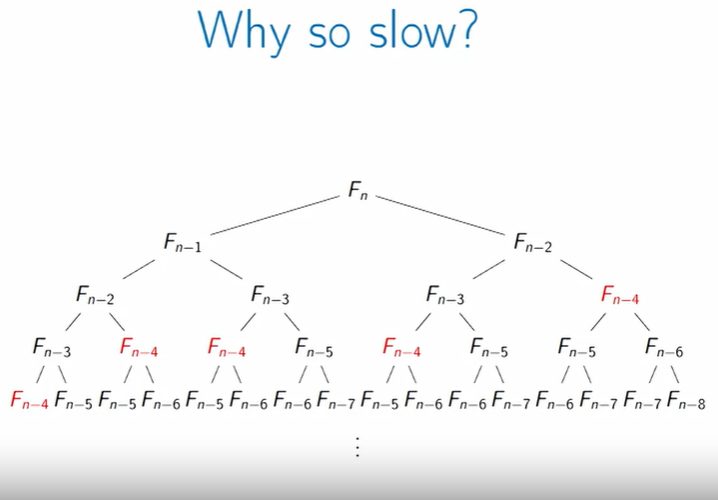












Bài 2: KÝ HIỆU Ô LỚN VÀ BIỂU DIỄN THỜI GIAN CHẠY BỞI KÝ HIỆU Ô LỚN

1/ Định nghĩa Ô lớn

Bây giờ chúng ta đưa ra định nghĩa khái niệm một hàm là “ô lớn” của

một hàm khác.

Định nghĩa. Giả sử f(n) và g(n) là các hàm thực không âm của đối số

nguyên không âm n. Ta nói “f(n) là ô lớn của g(n)” và viết là

f(n) = O( g(n) )

nếu tồn tại các hằng số dương c và n0 sao cho f(n) <= cg(n) với mọi n >= n0.

Như vậy, f(n) = O(g(n)) có nghĩa là hàm f(n) bị chặn trên bởi hàm

g(n) với một nhân tử hằng nào đó khi n đủ lớn. Muốn chứng minh được f(n)

= O(g(n)), chúng ta cần chỉ ra nhân tử hằng c , số nguyên dương n0 và chứng

minh được f(n) <= cg(n) với mọi n >= no .

Ví dụ. Giả sử f(n) = 5n3

+ 2n2 + 13n + 6 , ta có:

f(n) = 5n3

+ 2n2 + 13n + 6 <= 5n3

+ 2n3 + 13n3 + 6n3 = 26n3

Bất đẳng thức trên đúng với mọi n >= 1, và ta có n0 = 1, c = 26. Do đó, ta có

thể nói f(n) = O(n3

). Tổng quát nếu f(n) là một đa thức bậc k của n:

f(n) = akn

k + ak-1n

k-1 + ... + a1n + a0 thì f(n) = O(nk

)

Sau đây chúng ta đưa ra một số hệ quả từ định nghĩa ký hiệu ô lớn, nó

giúp chúng ta hiểu rõ bản chất ký hiệu ô lớn. (Lưu ý, các hàm mà ta nói tới

đều là các hàm thực không âm của đối số nguyên dương)

• Nếu f(n) = g(n) + g1(n) + ... + gk(n), trong đó các hàm gi(n)

(i=1,...,k) tăng chậm hơn hàm g(n) (tức là gi(n)/g(n) --> 0, khi n-->0)

thì f(n) = O(g(n))

• Nếu f(n) = O(g(n)) thì f(n) = O(d.g(n)), trong đó d là hằng số

dương bất kỳ

• Nếu f(n) = O(g(n)) và g(n) = O(h(n)) thì f(n) = O(h(n)) (tính bắc

cầu)

Các kết luận trên dễ dàng được chứng minh dựa vào định nghĩa của

ký hiệu ô lớn. Đến đây, ta thấy rằng, chẳng hạn nếu f(n) = O(n2

) thì f(n) =

O(75n2

), f(n) = O(0,01n2

), f(n) = O(n2 + 7n + logn), f(n) = O(n3

),..., tức là có

vô số hàm là cận trên (với một nhân tử hằng nào đó) của hàm f(n).

Một nhận xét quan trọng nữa là, ký hiệu O(g(n)) xác định một tập hợp

vô hạn các hàm bị chặn trên bởi hàm g(n), cho nên ta viết f(n) = O(g(n)) chỉ

có nghĩa f(n) là một trong các hàm đó.

2/ Biểu diễn thời gian chạy của thuật toán

Thời gian chạy của thuật toán là một hàm của cỡ dữ liệu vào: hàm

T(n). Chúng ta sẽ biểu diễn thời gian chạy của thuật toán bởi ký hiệu ô lớn:

T(n) = O(f(n)), biểu diễn này có nghĩa là thời gian chạy T(n) bị chặn trên bởi

hàm f(n). Thế nhưng như ta đã nhận xét, một hàm có vô số cận trên. Trong

số các cận trên của thời gian chạy, chúng ta sẽ lấy cận trên chặt (tight

bound) để biểu diễn thời gian chạy của thuật toán.

Định nghĩa. Ta nói f(n) là cận trên chặt của T(n) nếu

• T(n) = O(f(n)), và

• Nếu T(n) = O(g(n)) thì f(n) = O(g(n)).

Nói một cách khác, f(n) là cận trên chặt của T(n) nếu nó là cận trên

của T(n) và ta không thể tìm được một hàm g(n) là cận trên của T(n) mà lại

tăng chậm hơn hàm f(n).

Sau này khi nói thời gian chạy của thuật toán là O(f(n)), chúng ta cần

hiểu f(n) là cận trên chặt của thời gian chạy.

Nếu T(n) = O(1) thì điều này có nghĩa là thời gian chạy của thuật toán

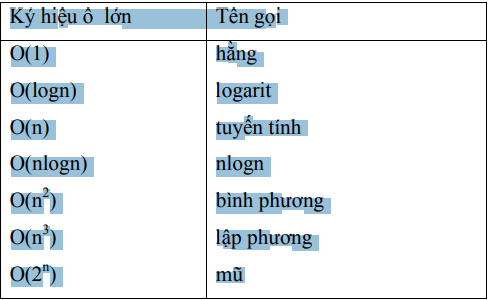
bị chặn trên bởi một hằng số nào đó, và ta thường nói thuật toán có thời gian

chạy hằng. Nếu T(n) = O(n), thì thời gian chạy của thuật toán bị chặn trên

bởi hàm tuyến tính, và do đó ta nói thời gian chạy của thuật toán là tuyến

tính. Các cấp độ thời gian chạy của thuật toán và tên gọi của chúng được liệt

kê trong bảng sau:



Đối với một thuật toán, chúng ta sẽ đánh giá thời gian chạy của nó

thuộc cấp độ nào trong các cấp độ đã liệt kê trên. Trong bảng trên, chúng ta

đã sắp xếp các cấp độ thời gian chạy theo thứ tự tăng dần, chẳng hạn thuật

toán có thời gian chạy là O(logn) chạy nhanh hơn thuật toán có thời gian

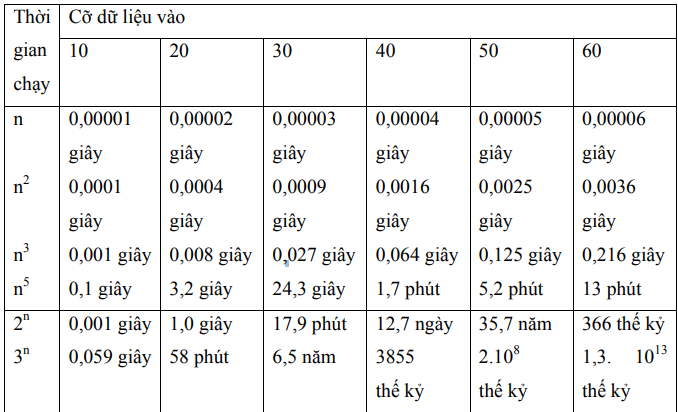
chạy là O(n),... Các thuật toán có thời gian chạy là O(nk

), với k = 1,2,3,...,

được gọi là các thuật toán thời gian chạy đa thức (polynimial-time

algorithm). Để so sánh thời gian chạy của các thuật toán thời gian đa thức và

các thuật toán thời gian mũ, chúng ta hãy xem xét bảng sau:



Trong bảng trên, ta giả thiết rằng mỗi phép toán sơ cấp cần 1 micro

giây để thực hiện. Thuật toán có thời gian chạy n

2

, với cỡ dữ liệu vào n = 20,

nó đòi hỏi thời gian chạy là 202

x10-6 = 0,004 giây. Đối với các thuật toán

thời gian mũ, ta thấy rằng thời gian chạy của thuật toán là chấp nhận được

chỉ với các dữ liệu vào có cỡ rất khiêm tốn, n < 30; khi cỡ dữ liệu vào tăng,

thời gian chạy của thuật toán tăng lên rất nhanh và trở thành con số khổng

lồ. Chẳng hạn, thuật toán với thời gian chạy 3n

, để tính ra kết quả với dữ liệu

vào cỡ 60, nó đòi hỏi thời gian là 1,3x1013 thế kỷ! Để thấy con số này khổng

lồ đến mức nào, ta hãy liên tưởng tới vụ nổ “big-bang”, “big-bang” được

ước tính là xảy ra cách đây 1,5x108

thế kỷ. Chúng ta không hy vọng có thể

áp dụng các thuật toán có thời gian chạy mũ trong tương lai nhờ tăng tốc độ

máy tính, bởi vì không thể tăng tốc độ máy tính lên mãi được, do sự hạn chế

của các quy luật vật lý. Vì vậy nghiên cứu tìm ra các thuật toán hiệu quả

(chạy nhanh) cho các vấn đề có nhiều ứng dụng trong thực tiễn luôn luôn là

sự mong muốn của các nhà tin học.

Bài 3: Đánh giá thời gian chạy của thuật toán

1. Luật tổng

Giả sử thuật toán gồm hai phần (hoặc nhiều phần), thời gian chạy của phần đầu là T1(n), phần sau là T2(n). Khi đó thời gian chạy của thuật toán là T1(n) + T2(n) sẽ được suy ra từ sự đánh giá của T1(n) và T2(n) theo luật sau: Luật tổng. Giả sử T1(n) = O(f(n)) và T2(n) = O(g(n)). Nếu hàm f(n) tăng nhanh hơn hàm g(n), tức là g(n) = O(f(n)), thì T1(n) + T2(n) = O(f(n)). Luật này được chứng minh như sau. Theo định nghĩa ký hiệu ô lớn, ta tìm được các hằng số c1, c2, c3 và n1, n2, n3 sao cho T1(n) <= c1f(n) với n >= n1 T2(n) <= c2g(n) với n >= n2 g(n) <= c3f(n) với n >= n3 Đặt n0 = max(n1, n2, n3). Khi đó với mọi n >= n0, ta có T1(n) + T2(n) <= c1f(n) + c2g(n) <= c1f(n) + c2c3f(n) = (c1+ c2 c3)f(n)

Như vậy với c = c1 + c2 c3 thì T1(n) + T2(n) <= cf(n) với mọi n >= n0 Ví dụ. Giả sử thuật toán gồm ba phần, thời gian chạy của từng phần được đánh giá là T1(n) = O(nlogn), T2(n) = O(n2 ) và T3(n) = O(n). Khi đó thời gian chạy của toàn bộ thuật toán là T(n) = T1(n) + T2(n) + T3(n) = O(n2 ), vì hàm n2 tăng nhanh hơn các hàm nlogn và n

1. Thời gian chạy các lệnh

Thời gian thực hiện các phép toán sơ cấp là O(1)

1. Lệnh gán

X = Thời gian chạy của lệnh gán là thời gian thực hiện biểu thức. Trường hợp hay gặp nhất là biểu thức chỉ chứa các phép toán sơ cấp, và thời gian thực hiện nó là O(1). Nếu biểu thức chứa các lời gọi hàm thì ta phải tính đến thời gian thực hiện hàm, và do đó trong trường hợp này thời gian thực hiện biểu thức có thể không là O(1).

1. Lệnh lựa chọn

Lệnh lựa chọn if-else có dạng

if (<điều kiện>)

lệnh 1

else

lệnh 2

Trong đó, điều kiện là một biểu thức cần được đánh giá, nếu điều kiện đúng thì lệnh 1 được thực hiện, nếu không thì lệnh 2 được thực hiện. Giả sử thời gian đánh giá điều kiện là T0(n), thời gian thực hiện lệnh 1 là T1(n), thời gian thực hiện lệnh 2 là T2(n). Thời gian thực hiện lệnh lựa chọn if-else sẽ là thời gian lớn nhất trong các thời gian T0(n) + T1(n) và T0(n) + T1(n). Trường hợp hay gặp là kiểm tra điều kiện chỉ cần O(1). Khi đó nếu T1(n) = O(f(n)), T2(n) = O(g(n)) và f(n) tăng nhanh hơn g(n) thì thời gian chạy của lệnh if-else là O(f(n)); còn nếu g(n) tăng nhanh hơn f(n) thì lệnh if-else cần thời gian O(g(n)). Thời gian chạy của lệnh lựa chọn switch được đánh giá tương tự như lệnh if-else, chỉ cần lưu ý rằng, lệnh if-else có hai khả năng lựa chọn, còn lệnh switch có thể có nhiều hơn hai khả năng lựa chọn.

1. Các lệnh lặp

Các lệnh lặp: for, while, do-while Để đánh giá thời gian thực hiện một lệnh lặp, trước hết ta cần đánh giá số tối đa các lần lặp, giả sử đó là L(n). Sau đó đánh giá thời gian chạy của mỗi lần lặp, chú ý rằng thời gian thực hiện thân của một lệnh lặp ở các lần lặp khác nhau có thể khác nhau, giả sử thời gian thực hiện thân lệnh lặp ở lần thứ i (i=1,2,..., L(n)) là Ti(n). Mỗi lần lặp, chúng ta cần kiểm tra điều kiện lặp, giả sử thời gian kiểm tra là T0(n). Như vậy thời gian chạy của lệnh lặp là: ∑( ) () () = + ( ) 1 0 L n i T n Ti n Công đoạn khó nhất trong đánh giá thời gian chạy của một lệnh lặp là đánh giá số lần lặp. Trong nhiều lệnh lặp, đặc biệt là trong các lệnh lặp for, ta có thể thấy ngay số lần lặp tối đa là bao nhiêu. Nhưng cũng không ít các

Bài 4: Phân tích hàm đệ quy tr 402

Phần 2: Các cấu trúc dữ liệu cơ bản

Chương 1: Sự trừu tượng hóa dữ liệu

Khi thiết kế thuật giải cho một vấn đề, chúng ta cần sử dụng sự trừu tượng hoá dữ liệu. Sự trừu tượng hoá dữ liệu được hiểu là chúng ta chỉ quan tâm tới một tập các đối tượng dữ liệu (ở mức độ trừu tượng) và các phép toán (các hành động) có thể thực hiện được trên các đối tượng dữ liệu đó. Với mỗi phép toán chúng ta cũng chỉ quan tâm tới điều kiện có thể sử dụng nó và hiệu quả mà nó mang lại, không cần biết nó được thực hiện như thế nào. Sự trừu tượng hoá dữ liệu được thực hiện bằng cách tạo ra các kiểu dữ liệu trừu tượng. Trong chương này chúng ta sẽ trình bày khái niệm kiểu dữ liệu trừu tượng, các phương pháp đặc tả và cài đặt kiểu dữ liệu trừu tượng.

Bài 1: Biểu diễn dữ liệu trong các ngôn ngữ lập trình

Trong khoa học máy tính, dữ liệu được hiểu là bất kỳ thông tin nào được xử lý bởi máy tính. Dữ liệu có thể là số nguyên, số thực, ký tự, … Dữ liệu có thể có cấu trúc phức tạp, gồm nhiều thành phần dữ liệu được liên kết với nhau theo một cách nào đó. Trong bộ nhớ của máy tính, mọi dữ liệu đều được biểu diễn dưới dạng nhị phân (một dãy các ký hiệu 0 và 1 ). Đó là dạng biểu diễn cụ thể nhất của dữ liệu (dạng biểu diễn vật lý của dữ liệu). Trong các ngôn ngữ lập trình bậc cao (Pascal, C, C+ +…), dữ liệu được biểu diễn dưới dạng trừu tượng, tức là dạng biểu diễn của dữ liệu xuất phát từ dạng biểu diễn toán học của dữ liệu (sử dụng các khái niệm toán học, các mô hình toán học để biểu diễn dữ liệu). Chẳng hạn, nếu dữ liệu là các điểm trong mặt phẳng, thì chúng ta có thể biểu diễn nó như một cặp số thực (x, y), trong đó số thực x là hoành độ, còn số thực y là tung độ của điểm. Do đó, trong ngôn ngữ C + +, một điểm được biểu diễn bởi cấu trúc:

Struct point

{

Double x;

Double y;

}

Trong các ngôn ngữ lập trình bậc cao, các dữ liệu được phân thành các lớp dữ liệu (kiểu dữ liệu ). Kiểu dữ liệu của một biến được xác định bởi một tập các giá trị mà biến đó có thể nhận và các phép toán có thể thực hiện trên các giá trị đó. Ví dụ, có lẽ kiểu dữ liệu đơn giản nhất và có trong nhiều ngôn ngữ lập trình là kiểu boolean, miền giá trị của kiểu này chỉ gồm hai giá trị false và true, các phép toán có thể thực hiện trên các giá trị này là các phép toán logic mà chúng ta đã quen biết.

Mỗi ngôn ngữ lập trình cung cấp cho chúng ta một số kiểu dữ liệu cơ

bản (basic data types). Trong các ngôn ngữ lập trình khác nhau, các kiểu

dữ liệu cơ bản có thể khác nhau. Ngôn ngữ lập trình Lisp chỉ có một kiểu cơ

bản, đó là các S-biểu thức. Song trong nhiều ngôn ngữ lập trình khác (chẳng

hạn Pascal, C / C + +, Ada, …), các kiểu dữ liệu cơ bản rất phong phú. Ví

dụ, ngôn ngữ C + + có các kiểu dữ liệu cơ bản sau:

Các kiểu ký tự ( char, signed char, unsigned char )

Các kiểu nguyên (int, short int, long int, unsigned)

Các kiểu thực (float, double, long double)

Các kiểu liệt kê (enum)

Kiểu boolean (bool)

Gọi là các kiểu dữ liệu cơ bản, vì các dữ liệu của các kiểu này sẽ được

sử dụng như các thành phần cơ sở để kiến tạo nên các dữ liệu có cấu trúc

phức tạp. Các kiểu dữ liệu đã cài đặt sẵn (build-in types) mà ngôn ngữ lập

trình cung cấp là không đủ cho người sử dụng. Trong nhiều áp dụng, người

lập trình cần phải tiến hành các thao tác trên các dữ liệu phức hợp. Vì vậy,

mỗi ngôn ngữ lập trình cung cấp cho người sử dụng một số quy tắc cú pháp

để tạo ra các kiểu dữ liệu mới từ các kiểu cơ bản hoặc các kiểu khác đã được

xây dựng. Chẳng hạn, C + + cung cấp cho người lập trình các luật để xác định các kiểu mới: kiểu mảng (array), kiểu cấu trúc (struct), kiểu con trỏ, …

Ví dụ: Từ các kiểu đã có T1, T2, …, Tn (có thể khác nhau), khai báo sau

struct S {

T1 M1 ;

T2 M2 ;

………….

Tn Mn ;

}

xác định một kiểu cấu trúc với tên là S, mỗi dữ liệu của kiểu này gồm n

thành phần, thành phần thứ i có tên là Mi và có giá trị thuộc kiểu Ti (i =

1,…, n).

Các kiểu dữ liệu được tạo thành từ nhiều kiểu dữ liệu khác (các kiểu

này có thể là kiểu cơ bản hoặc kiểu dữ liệu đã được xây dựng) được gọi là

kiểu dữ liệu có cấu trúc. Các dữ liệu thuộc kiểu dữ liệu có cấu trúc được

gọi là các cấu trúc dữ liệu (data structure). Ví dụ, các mảng, các cấu trúc,

các danh sách liên kết, … là các cấu trúc dữ liệu (CTDL).

Từ các kiểu cơ bản, bằng cách sử dụng các qui tắc cú pháp kiến tạo

các kiểu dữ liệu, người lập trình có thể xây dựng nên các kiểu dữ liệu mới

thích hợp cho từng vấn đề. Các kiểu dữ liệu mà người lập trình xây dựng

nên được gọi là các kiểu dữ liệu được xác định bởi người sử dụng (userdefined data types).

Như vậy, một CTDL là một dữ liệu phức hợp, gồm nhiều thành phần

dữ liệu, mỗi thành phần hoặc là dữ liệu cơ sở (số nguyên, số thực, ký tự,… )

hoặc là một CTDL đã được xây dựng. Các thành phần dữ liệu tạo nên một

CTDL được liên kết với nhau theo một cách nào đó. Trong các ngôn ngữ lập

trình thông dụng (Pascal, C/ C+ +), có ba phương pháp để liên kết các dữ

liệu:

1. Liên kết các dữ liệu cùng kiểu tạo thành mảng dữ liệu.

2. Liên kết các dữ liệu (không nhất thiết cùng kiểu) tạo thành cấu

trúc trong C/ C+ +, hoặc bản ghi trong Pascal.

3. Sử dụng con trỏ để liên kết dữ liệu. Chẳng hạn, sử dụng con trỏ

chúng ta có thể tạo nên các danh sách liên kết, hoăc các CTDL để biểu

diễn cây. (Chúng ta sẽ nghiên cứu các CTDL này trong các chương

sau)

Ví dụ. Giả sử chúng ta cần xác định CTDL biểu diễn các lớp học. Giả

sử mỗi lớp học cần được mô tả bởi các thông tin sau: tên lớp, số tổ của lớp,

danh sách sinh viên của mỗi tổ; mỗi sinh viên được mô tả bởi 3 thuộc tính:

tên sinh viên, tuổi và giới tính. Việc xây dựng một CTDL cho một đối tượng

dữ liệu được tiến hành theo nguyên tắc sau: từ các dữ liệu có kiểu cơ sở tạo

ra kiểu dữ liệu mới, rồi từ các kiểu dữ liệu đã xây dựng tạo ra kiểu dữ liệu

phức tạp hơn, cho tới khi nhận được kiểu dữ liệu cho đối tượng dữ liệu

mong muốn. Trong ví dụ trên, đầu tiên ta xác định cấu trúc Student

struct Student

{

string StName;

int Age;

bool Sex;

}

struct Cell

{

Student Infor;

Cell\* Next;

}

Danh sách sinh viên của mỗi tổ có thể lưu trong mảng, hoặc biểu diễn bởi danh sách liên kết. Ở đây chúng ta dùng danh sách liên kết, mỗi tế bào của nó là cấu trúc sau:

Chúng ta sử dụng một mảng để biểu diễn các tổ, mỗi thành phần của

mảng lưu con trỏ trỏ tới đầu một danh sách liên kết biểu diễn danh sách các

sinh viên của một tổ. Giả sử mỗi lớp có nhiều nhất 10 tổ, kiểu mảng

GroupArray được xác định như sau:

typedef Cell\* GroupArray[10];

Cuối cùng, ta có thể biểu diễn lớp học bởi cấu trúc sau:

struct StudentClass

{

string ClassName;

int GroupNumber;

GroupArray Group;

}

Bài 2: Sự trừu tượng hóa của dữ liệu

Thiết kế và phát triển một chương trình để giải quyết một vấn đề là một quá trình phức tạp. Thông thường quá trình này cần phải qua các giai đoạn chính sau:

1. Đặc tả vấn đề.

2. Thiết kế thuật toán và cấu trúc dữ liệu.

3. Cài đặt (chuyển dịch thuật toán thành các câu lệnh trong một

ngôn ngữ lập trình, chẳng hạn C+ +)

4. Thử nghiệm và sửa lỗi.

Liên quan tới nội dung của sách này, chúng ta chỉ đề cập tới hai giai đoạn đầu. Chúng ta muốn làm sang tỏ vai trò quan trọng của sự trừu tượng hoá (abstraction) trong đặc tả một vấn đề, đặc biệt là sự trừu tương hoá dữ liệu (data abstraction) trong thiết kế thuật toán.

Bài 3: Kiểu dữ liệu trừu tượng

1. Đặc điểm

Nhớ lại rằng, một KDLTT được định nghĩa là một tập các đối tượng

dữ liệu và một tập các phép toán trên các đối tượng dữ liệu đó. Do đó, đặc tả

một KDLTT gồm hai phần: đặc tả đối tượng dữ liệu và đặc tả các phép toán.

• Đặc tả đối tượng dữ liệu. Mô tả bằng toán học các đối tượng

dữ liệu. Thông thường các đối tượng dữ liệu là các đối tượng trong

thế giới hiện thực, chúng là các thực thể phức hợp, có cấu trúc nào

đó. Để mô tả chúng, chúng ta cần sử dụng sự trừu tượng hoá (chỉ

quan tâm tới các đặc tính quan trọng, bỏ qua các chi tiết thứ yếu).

Nói cụ thể hơn, để mô tả chính xác các đối tượng dữ liệu , chúng ta

cần sử dụng các khái niệm toán học, các mô hình toán học như tập

hợp, dãy, đồ thị, cây, … Chẳng hạn, đối tượng dữ liệu là sinh viên,

thì có thể biểu diễn nó bởi một tập các thuộc tính quan trọng như tên,

ngày sinh, giới tính, …

• Đặc tả các phép toán. Việc mô tả các phép toán phải đủ chặt

chẽ, chính xác nhằm xác định đầy đủ kết quả mà các phép toán mang

lại, nhưng không cần phải mô tả các phép toán được thực hiện như

thế nào để cho kết quả như thế. Cách tiếp cận chính xác để đạt được

mục tiêu trên là khi mô tả các phép toán, chúng ta xác định một tập

các tiên đề mô tả đầy đủ các tính chất của các phép toán. Chẳng hạn,

các phép toán cộng và nhân các số nguyên phải thoả mãn các tiên đề:

giao hoán, kết hợp, phân phối, … Tuy nhiên, việc xác định một tập

đầy đủ các tiên đề mô tả đầy đủ bản chất của các phép toán là cực kỳ

khó khăn, do đó chúng ta mô tả các phép toán một cách không hình

thức. Chúng ta sẽ mô tả mỗi phép toán bởi một hàm (hoặc thủ tục),

tên hàm là tên của phép toán, theo sau là danh sách các biến. Sau đó

chỉ rõ nhiệm vụ mà hàm cần phải thực hiện.

Ví dụ. Sau đây là đặc tả KDLTT số phức. Trong sách này, chúng ta sẽ đặc tả các KDLTT khác theo khuôn mẫu của ví dụ này. Mỗi số phức là một cặp số thực (x, y), trong đó x được gọi là phần thực (real), y được gọi là phần ảo (image) của số phức. Trên các số phức, có thể thực hiện các phép toán sau: 1. Create (a, b). Trả về số phức có phần thực là a, phần ảo là b. 2. GetReal (c). Trả về phần thực của số phức c. 3. GetImage (c). Trả về phần ảo của số phức c. 4. Abs (c). Trả về giá trị tuyệt đối (mođun) của số phức c. 5. Add (c1,c2). Trả về tổng của số phức c1 và số phức c2 . 6. Multiply (c1 , c2 ). Trả về tích của số phức c1 và số phức c2 .

7. Print (c). Viết ra số phức c dưới dạng a + i b trong đó a là phần thực, b là phần ảo của số phức c. Trên đây chỉ là một số ít các phép toán số phức. Còn nhiều các phép toán khác trên số phức, chẳng hạn các phép toán so sánh, các phép toán lượng giác, …, để cho ngắn ngọn chúng ta không liệt kê ra hết

2. Cài đặt kiểu dữ liệu trừu tượng

Công việc đầu tiên phải làm khi cài đặt một KDLTT là chọn một CTDL để biểu diễn các đối tượng dữ liệu. Cần lưu ý rằng, một CTDL là một dữ liệu phức hợp được tạo nên từ nhiều dữ liệu thành phần bằng các liên kết nào đó. Chúng ta có thể mô tả các CTDL trong một ngôn ngữ lập trình (chẳng hạn, C/ C + +) bằng cách sử dụng các phương tiện có sẵn trong ngôn ngữ lập trình đó, chẳng hạn sử dụng các qui tắc cú pháp mô tả mảng, cấu trúc, … Một CTDL cũng xác định cho ta cách lưu trữ dữ liệu trong bộ nhớ của máy tính.

Ví dụ. Chúng ta có thể biểu diễn một số phức bởi cấu trúc trong C + +

struct complex

{

float real;

float imag;

}

Cần chú ý rằng, một đối tượng dữ liệu có thể cài đặt bởi các CTDL

khác nhau. Chẳng hạn, một danh sách (a1, a2, …, an) có thể cài đặt bởi mảng

A, các thành phần của danh sách lần lượt được lưu trong các thành phần liên

tiếp của mảng A[0], A[1],…, A[n-1]. Nhưng chúng ta cũng có thể cài đặt

danh sách bởi CTDL danh sách liên kết sau:



Sau khi đã chọn CTDL biểu diễn đối tượng dữ liệu, bước tiếp theo

chúng ta phải thiết kế và cài đặt các hàm thực hiện các phép toán của

KDLTT.

Trong giai đoạn thiết kế một hàm thực hiện nhiệm vụ của một phép

toán, chúng ta cần sử dụng sự trừu tượng hoá hàm (functional

abstraction). Sự trừu tượng hoá hàm có nghĩa là cần mô tả hàm sao cho

người sử dụng biết được hàm thực hiện công việc gì, và sao cho họ có thể sử

dụng được hàm trong chương trình của mình mà không cần biết đến các chi

tiết cài đặt, tức là không cần biết hàm thực hiện công việc đó như thế nào.

Sự trừu tượng hoá hàm được thực hiện bằng cách viết ra mẫu hàm

(function prototype) kèm theo các chú thích.

Mẫu hàm gồm tên hàm và theo sau là danh sách các tham biến. Tên

hàm cần ngắn ngọn, nói lên được nhiệm vụ của hàm. Các tham biến cần phải

đầy đủ: các dữ liệu vào cần thiết để hàm có thể thực hiện được công việc của

mình và các dữ liệu ra sau khi hàm hoàn thành công việc.

Chú thích đưa ra sau đầu hàm là rất cần thiết (đặc biệt trong các đề án

lập trình theo đội). Trong chú thích này, chúng ta cần mô tả đầy đủ, chính

xác nhiệm vụ của hàm. Sau đó là hai phần: Preconditions (các điều kiện

trước) và Postconditions (các điều kiện sau).

• Preconditions gồm các phát biểu về các điều kiện cần phải thoả

mãn trước khi hàm thực hiện.

• Postconditions gồm các phát biểu về các điều kiện cần phải

thoả mãn sau khi hàm hoàn thành thực hiện.

Hai phần Preconditions và Postconditions tạo thành hợp đồng giữa

một bên là người sử dụng hàm và một bên là hàm. Preconditions là trách

nhiệm của người sử dụng, còn Postconditions là trách nhiệm của hàm. Một

khi sử dụng hàm (gọi hàm), người sử dụng phải có trách nhiệm cung cấp

cho hàm các dữ liệu vào thoả mãn các điều kiện trong Preconditions. Sau

khi hoàn thành thực hiện, hàm phải cho ra các kết quả thoả mãn các điều

kiện trong Postconditions. Sau đây là ví dụ một mẫu hàm:

void Sort (int A[ ], int n)

// Sắp xếp mảng A theo thứ tự không giảm.

// Preconditions: A là mảng số nguyên có cỡ Max ≥ n.

// Postconditions: A[0] ≤ A[1] ≤ …. ≤ A[n-1],

// n không thay đổi.

Bước tiếp theo, chúng ta phải thiết kế thuật toán thực hiện công việc

của hàm khi mà đối tượng dữ liệu được biểu diễn bởi CTDL đã chọn. Việc

cài đặt hàm bây giờ là chuyển dịch thuật toán thực hiện nhiệm vụ của hàm

sang dãy các khai báo biến địa phương cần thiết và các câu lệnh. Tất cả các

chi tiết mà hàm cần thực hiện này là công việc riêng tư của hàm, người sử

dụng hàm không cần biết đến, và không được can thiệp vào. Làm được như

vậy có nghĩa là chúng ta đã thực hành nguyên lý che dấu thông tin (the

principle of information hiding) - một nguyên lý quan trọng trong phương

pháp luận lập trình môđun.

Trên đây chúng ta mới chỉ trình bày các kỹ thuật liên quan đến thiết

kế CTDL cho đối tượng dữ liệu, thiết kế và cài đặt các hàm cho các phép

toán của KDLTT. Câu hỏi được đặt ra là: Chúng ta phải tổ chức CTDL và

các hàm đó như thế nào? Có hai cách: cách cài đặt cổ điển và cách cài đặt

định hướng đối tượng. Mục sau sẽ trình bày phương pháp cài đặt KDLTT

trong ngôn ngữ C. Cài đặt KDLTT bởi lớp trong C + + sẽ được trình bày

trong chương 3. Chương 2: Kiểu dữ liệu trừu tượng và các lớp C++

Mục đích của chương này là trình bày khái niệm lớp và các thành

phần của lớp trong C + +. Sự trình bày sẽ không đi vào chi tiết, mà chỉ đề

cập tới các vấn đề quan trọng liên quan tới các thành phần của lớp giúp cho

bạn đọc dễ dàng hơn trong việc thiết kế các lớp khi cài đặt các KDLTT.

Chương này cũng trình bày khái niệm lớp khuôn, lớp khuôn được sử dụng

để cài đặt các lớp côngtơnơ. Cuối chương chúng ta sẽ giới thiệu các KDLTT

quan trọng sẽ được nghiên cứu kỹ trong các chương sau. 1. Lớp và các thành phần của lớp

Các ngôn ngữ lập trình định hướng đối tượng, chẳng hạn C + +, cung

cấp các phương tiện cho phép đóng gói CTDL và các hàm thao tác trên

CTDL thành một đơn vị được gọi là lớp (class). Ví dụ, sau đây là định nghĩa

lớp số phức:

class Complex

{

public :

(1) Complex (double a = 0.0 , double b = 0.0) ;

(2) Complex (const Complex & c);

(3) double GetReal ( ) const ;

(4) double GetImag ( ) const ;

(5) double GetAbs ( ) const ;

(6) friend Complex & operator + (const Complex & c1, const Complex & c2) ;

(7) friend Complex & operator - (const Complex & c1,

const Complex & c2) ;

(8) friend Complex & operator \* (const Complex & c1,

const Complex & c2) ;

(9) friend Complex & operator / (const Complex & c1,

const Complex & c2) ;

(10) friend ostream & operator << (ostream & os,

const Complex & c);

// Các mẫu hàm cho các phép toán khác.

private:

double real ;

double imag ;

} ;

Từ ví dụ đơn giản trên, chúng ta thấy rằng, một lớp bắt đầu bởi đầu

lớp: đầu lớp gồm từ khoá class, rồi đến tên lớp. Phần còn lại trong định

nghĩa lớp (nằm giữa cặp dấu { và } ) là danh sách thành phần. Danh sách

thành phần gồm các thành phần dữ liệu (data member), hay còn gọi là

biến thành phần (member variable), chẳng hạn lớp Complex có hai biến

thành phần là real và imag. Các thành phần (1) – (5) trong lớp Complex là

các hàm thành phần (member functions hoặc methods).

Một lớp là một kiểu dữ liệu, ví dụ khai báo lớp Complex như trên, có

nghĩa là người lập trình đã xác định một kiểu dữ liệu Complex. Các đối

tượng dữ liệu thuộc một lớp được gọi là các đối tượng (objects).

Các thành phần của lớp điển hình được chia thành hai mục: mục

public và mục private như trong định nghĩa lớp Complex. Trong chương

trình, người lập trình có thể sử dụng trực tiếp các thành phần trong mục

public để tiến hành các thao tác trên các đối tượng của lớp. Các thành phần

trong mục private chỉ được phép sử dụng trong nội bộ lớp. Mục public (mục

private) có thể chứa các hàm thành phần và các biến thành phần. Tuy nhiên, khi cần thiết kế một lớp cài đặt một KDLTT, chúng ta nên đưa các biến

thành phần mô tả CTDL vào mục private, còn các hàm biểu diễn các phép

toán vào mục public. Trong định nghĩa lớp Complex cài đặt KDLTT số

phức, chúng ta đã làm như thế.

Nên biết rằng, các thành phần của lớp có thể khai báo là tĩnh bằng

cách đặt từ khoá static ở trước. Trong một lớp, chúng ta có thể khai báo các

hằng tĩnh, các biến thành phần tĩnh, các hàm thành phần tĩnh. Chẳng hạn:

static const int CAPACITY = 50; // khai báo hằng tĩnh

static double static Var; // khai báo biến tĩnh

Các thành phần tĩnh là các thành phần được dùng chung cho tất cả

các đối tượng của lớp. Trong lớp Complex không có thành phần nào cần

phải là tĩnh.

Nếu khai báo của hàm trong một lớp bắt đầu bởi từ khoá friend, thì

hàm được nói là bạn của lớp, chẳng hạn các hàm (6) – (10) trong lớp

Complex. Một hàm bạn (friend function) không phải là hàm thành phần,

song nó được phép truy cập tới các thành phần dữ liệu trong mục private của

lớp.

Một hàm thành phần mà khai báo của nó có từ khoá const ở sau cùng

được gọi là hàm thành phần hằng (const member function). Một hàm

thành phần hằng có thể xem xét trạng thái của đối tượng, song không được

phép thay đổi nó. Chẳng hạn, các hàm (3), (4), (5) trong lớp Complex. Các

hàm này khi áp dụng vào một số phức, không làm thay đổi số phức mà chỉ

cho ra phần thực, phần ảo và mođun của số phức, tương ứng.

Bài 2: Các hàm thành phần

1. Hàm kiến tạo và hàm tích lũy

Hàm kiến tạo có đặc điểm là tên của nó trùng với tên lớp và không có kiểu trả về, chẳng hạn hàm (1), (2) trong lớp Complex. Nếu trong một lớp, bạn không định nghĩa một hàm kiến tạo, thì chương trình dịch sẽ tự động tạo ra một hàm kiến tạo mặc định tự động (automatic default constructor). Hàm này chỉ tạo ra đối tượng với tất cả các thành phần dữ liệu đều bằng 0. Nói chung, rất ít khi người ta thiết kế một lớp không có hàm kiến tạo. Đặc biệt khi bạn thiết kế một lớp có chứa thành phần dữ liệu là đối tượng của một lớp khác, thì nhất thiết bạn phải viết hàm kiến tạo. Một loại hàm kiến tạo đặc biệt có tên gọi là hàm kiến tạo copy (copy constructor). Nhiệm vụ của hàm kiến tạo copy là khởi tạo ra một đối tượng mới là bản sao của một đối tượng đã có. Ví dụ, hàm (2) trong lớp Complex là hàm kiến tạo copy. Hàm kiến tạo copy chỉ có một tham biến tham chiếu hằng có kiểu là kiểu lớp đang định nghĩa. Nếu bạn không đưa vào một hàm kiến tạo copy trong định nghĩa lớp, thì chương trình dịch sẽ tự động tạo ra một hàm kiến tạo copy tự động (automatic copy constructor). Nó thực hiện sao chép tất cả các thành phần dữ liệu của đối tượng đã có sang đối tượng đang khởi tạo. Nói chung, trong nhiều trường hợp chỉ cần sử dụng hàm kiến tạo copy tự động là đủ. Chẳng hạn, trong lớp Complex, thực ra không cần có hàm kiến tạo copy (2). Song trong trường hợp lớp chứa các biến thành phần là biến con trỏ, thì cần thiết phải thiết kế hàm kiến tạo copy cho lớp. (Tại sao?) Sau đây là một số ví dụ sử dụng hàm kiến tạo trong khai báo các đối tượng thuộc lớp Complex:

Complex c1; // khởi tạo số phức c1 với c1.real = 0.0 và c1.imag = 0.0

Complex c2(2.6); // khởi tạo số phức c2 với c2.real = 2.6 // và c2.imag = 0.0 Complex c3(5.4, 3.7); // khởi tạo số phức c3 với c3.real =5.4 // và c3.imag = 3.7 Complex c4 = c2; // khởi tạo số phức c4 là copy của c2.

Ngược lại với hàm kiến tạo là hàm huỷ (destructor). Hàm huỷ thực hiện nhiệm vụ huỷ đối tượng (thu hồi vùng nhớ cấp phát cho đối tượng và trả lại cho hệ thống), khi đối tượng không cần thiết cho chương trình nữa. Hàm huỷ là hàm thành phần có tên trùng với tên lớp, không có tham biến và phía trước có dấu ngã ~. Hàm huỷ tự động được gọi khi đối tượng ra khỏi phạm vi của nó. Trong một định nghĩa lớp chỉ có thể có một hàm huỷ. Nói chung, trong một lớp không cần thiết phải đưa vào hàm huỷ (chẳng hạn, lớp Complex), trừ trường hợp lớp chứa thành phần dữ liệu là con trỏ trỏ tới vùng nhớ cấp phát động.

***hàm kiến tạo (constructor)***

1. Các tham biến của hàm

Các hàm thành phần của một lớp cũng như các hàm thông thường khác có một danh sách các tham biến ( danh sách này có thể rỗng) được liệt kê sau tên hàm trong khai báo hàm. Các tham biến này được gọi là tham biến hình thức (formal parameter). Khi gọi hàm, các tham biến hình thức được thay thế bởi các đối số (argument) hay còn gọi là các tham biến thực tế (actual parameter).

Chúng ta xem xét ba loại tham biến:

Tham biến giá trị: Tham biến giá trị (value parameter) được khai báo bằng cách viết tên kiểu theo sau là tên tham biến. Chẳng hạn, trong hàm kiến tạo của lớp Complex: Complex (double a = 0.0, double b = 0.0) ; thì a và b là các tham biến giá trị. Trong khai báo trên chúng ta đã xác định các đối số mặc định (default argument) cho các tham biến a và b, chúng đều là 0.0. Khi chúng ta gọi hàm kiến tạo không đưa vào đối số, thì có nghĩa là đã gọi hàm kiến tạo với đối số mặc định. Ví dụ, khi ta khai báo Complex c ; thì số phức c được khởi tạo bằng gọi hàm kiến tạo với các đối số mặc định (số phức c có phần thực và phần ảo đều là 0.0).

1. Định nghĩa phép toán

Giả sử trong định nghĩa lớp Complex, chúng ta xác định các hàm tính tổng và tích của hai số phức như sau: Complex & Add (const Complex & c1, const Complex & c2) ; Complex & Multiply (const Complex & c1, const Complex & c2) ; Khi đó trong chương trình muốn lấy số phức A cộng với tích của số phức B và số phức C, ta cần viết: D = Add (A, Multiply (B, C)) ; Cách viết này rất không sáng sủa, nhất là đối với các tính toán phức tạp hơn trên các số phức. Chúng ta mong muốn biểu diễn các tính toán trên các số phức trong chương trình bởi các biểu thức toán học. Chẳng hạn, dòng lệnh trên, nếu được viết thành: D = A + B \* C ; thì chương trình sẽ trở nên sáng sủa hơn, dễ đọc, dễ hiểu hơn. Sử dụng các công cụ mà C + + cung cấp, chúng ta có thể làm được điều đó. Trong ngôn ngữ lập trình C + + có rất nhiều các phép toán (toán tử). Chẳng hạn, các phép toán số học +, - , \* , / , % ; các phép toán so sánh = =, != , < , <= , > , >= , các toán tử gán và rất nhiều các phép toán khác. Các phép toán này có ngữ nghĩa đã được xác định trong ngôn ngữ. Chúng ta muốn sử dụng các ký hiệu phép toán trong C + +, nhưng với ngữ nghĩa hoàn toàn mới, chẳng hạn chúng ta muốn sử dụng ký hiệu + để chỉ phép cộng số phức hoặc phép cộng vectơ hoặc phép cộng ma trận … Việc xác định lại ngữ nghĩa của các phép toán (toán tử) trên các lớp đối tượng dữ liệu mới sẽ được gọi là định nghĩa lại các phép toán ( operator overloading). Các phép toán được định nghĩa lại bởi các hàm có tên hàm bắt đầu bởi từ khoá operator và đi sau là ký hiệu phép toán, chúng ta sẽ gọi các hàm này là hàm toán tử. Ví dụ, chúng ta có thể định nghĩa lại phép toán + cho các số phức. Có ba cách định nghĩa phép toán cộng số phức bởi hàm toán tử operator +

• Hàm toán tử không phải là hàm thành phần của lớp Complex: Complex & Operator + ( const Complex & c1, const Complex & c2); { double x , y ; x = c1. GetReal ( ) + c2. GetReal ( ) ; y = c1. GetImag ( ) + c2. GetImag ( ) ; Complex c(x,y) ; return c ; } Khi đó, trong chương trình muốn cộng hai số phức, ta có thể viết như sau: Complex A (3.5, 2.7) ; Complex B (-4.3, 5.8) ; Complex C ; C = A + B ; Cũng có thể viết C = operator + (A, B), nhưng không nên sử dụng cách này.

• Hàm toán tử là hàm thành phần của lớp Complex

Complex & Complex :: operator + (const Complex & c) { Complex temp ; temp.real = real + c.real ; temp.imag = imag + c. imag ; return temp ; } Trong cách này, khi ta viết C = A + B, thì toán hạng thứ nhất (số phức A) là đối tượng kích hoạt hàm toán tử, tức là C = A.operator + (B).

• Hàm toán tử là hàm bạn của lớp Complex. Đây là cách mà chúng ta đã lựa chọn trong định nghĩa lớp Complex (xem mục 2.1.). Hàm bạn này được cài đặt như sau: Complex & operator + (const Complex & c1, const Complex & c2); { Complex sum ; sum.real = c1.real + c2.real ; sum.imag = c1.imag + c2.imag ; return sum ; } Sử dụng hàm toán tử là bạn giống như sử dụng hàm toán tử không phải thành phần của lớp. Có sự khác nhau tinh tế giữa hàm toán tử thành phần và hàm toán tử bạn. Ví dụ, giả sử A và B là hai số phức, và hàm operator + là hàm bạn của lớp Complex. Khi đó, câu lệnh: A = 1 + B ; được chương trình dịch xem là: A = operator+ (1,B) ;

và để thực hiện, 1 được chuyển thành đối tượng Complex với phần thực bằng 1, phần ảo bằng 0 bởi toán tử chuyển kiểu được xác định trong lớp Complex, rồi được cộng với số phức A. Chúng ta xét xem điều gì sẽ xảy ra khi hàm toán tử operator + là hàm thành phần của lớp Complex. Trong trường hợp này, chương trình dịch sẽ minh họa A = 1 + B như là A = 1. operator (B) ; Nhưng 1 không phải là đối tượng của lớp Complex, do đó nó không thể kích hoạt một hàm thành phần của lớp Complex. Điều này dẫn tới lỗi! Vì những lý do trên, khi thiết kế một lớp cài đặt một KDLTT thì các phép toán hai toán hạng (chẳng hạn, các phép cộng, trừ, nhân, chia số phức) nên được cài đặt bởi hàm toán tử bạn của lớp. Trong một lớp cài đặt một KDLTT, nói chung ta cần đưa vào một hàm viết ra đối tượng dữ liệu trên các thiết bị ra chuẩn. C + + đã đưa vào toán tử << để viết ra các số nguyên, số thực, ký tự, … Chúng ta có thể định nghĩa lại toán tử << để viết ra các đối tượng dữ liệu phức hợp khác, chẳng hạn để viết ra các số phức trong lớp Complex. Trong lớp Complex, hàm operator << được thiết kế là hàm bạn với khai báo sau: ostream & operator << (ostream & os, const Complex & c) ; trong đó ostream là lớp các luồng dữ liệu ra (output stream), ostream là thành viên của thư viện iostream.h, và cout (thiết bị ra chuẩn) là một đối tượng của lớp ostream. Sau đây là cài đặt hàm toán tử bạn operator << trong lớp Complex: ostream & operator << (ostream & os, const Complex & c) // Postcondition. Số phức c được viết ra luồng os, dưới dạng a + ib, // trong đó a là phần thực và b là phần ảo của số phức c. // Giá trị trả về là luồng ra os. {

os << c.real << “ + i” << c.imag ; return os ; } Trên đây chúng ta đã xét cách cài đặt các hàm toán tử định nghĩa lại các phép toán + và << cho các số phức. Các ví dụ đó cũng cho bạn một phương pháp chung để khi thiết kế một lớp cài đặt một KDLTT, bạn có thể cài đặt một phép toán hai toán hạng bởi một hàm toán tử định nghĩa lại các phép toán trong C + +. Hầu hết các phép toán, các toán tử trong C + + đều có thể định nghĩa lại. Tuy nhiên, khi thiết kế các lớp cài đặt các KDLTT, thông thường chúng ta chỉ cần đến định nghĩa lại các phép toán số học: +, - , \* , / , các phép toán so sánh = = , ! = , < , < = , > , > = , các toán tử gán = , + = , - = , \* = , / = .

Bài 3: Cài đặt C++ Trong kiểu dữ liệu trừu tượng

Trong mục này chúng ta sẽ trình bày một ví dụ về lớp Complex, qua đó bạn đọc sẽ thấy cần phải làm gì để phát triển một lớp C + + cài đặt một KDLTT. Phần cuối của mục sẽ trình bày các hướng dẫn cài đặt KDLTT bởi lớp. Một lớp khi được khai báo sẽ là một kiểu dữ liệu được xác định bởi người sử dụng. Vì vậy, bạn có thể khai báo một lớp trong chương trình và sử dụng nó trong chương trình giống như khai báo và sử dụng các kiểu dữ liệu quen thuộc khác. Hình 2.1 là một chương trình demo cho việc khai báo và sử dụng lớp Complex. Chú ý rằng, khi cài đặt các hàm thành phần của một lớp, bạn cần sử dụng toán tử định phạm vi để chỉ nó thuộc lớp đó ở đây bạn phải đặt Complex :: trước tên hàm.

# include < math.h >

# include < iostream.h >

class Complex { public : Complex (double a = 0, double b = 0) ;

// Tạo ra số phức có phần thực a, phần ảo b

double GetReal( ) const ;

// Trả về phần thực của số phức.

double GetImag ( ) const ;

// Trả về phần ảo của số phức.

double GetAbs ( ) const ;

// Trả về giá trị tuyệt đối của số phức.

friend Complex & operator +(const Complex & c1,const Complex &c2);

// Trả về tổng của số phức c1 và c2.

friend Complex & operator -(const Complex & c1,const Complex & c2);

// Trả về hiệu của số phức c1 và c2.

friend Complex & operator \*(const Complex & c1,const Complex & c2);

// Trả về tích của số phức c1 và c2.

friend Complex & operator /(const Complex & c1, const Complex & c2);

// Trả về thương của số phức c1 và c2.

friend ostream & operator << (ostream & os, const Complex &c);

// In số phức c trên luồng ra os.

// Các mẫu hàm khác. private : double real ; double imag ;

} ;

int main ( )

{ Complex A (3.2, 5.7) ;

Complex B (6.3, -4.5) ;

Tuy nhiên chúng ta thiết kế một KDLTT và cài đặt nó bởi lớp C + + là để sử dụng trong một chương trình bất kỳ cần đến KDLTT đó, do đó khi phát triển một lớp cài đặt một KDLTT, chúng ta cần phải tổ chức thành hai file: file đầu và file cài đặt (tương tự như chúng ta đã làm khi cài đặt không định hướng đối tượng KDLTT, xem mục 1.4 ).

• File đầu: File đầu có tên kết thúc bởi .h. File đầu chứa tất cả các thông tin cần thiết mà người lập trình cần biết để sử dụng KDLTT trong chương trình của mình. Chúng ta sẽ tổ chức file đầu như sau: Đầu tiên là các chú thích về các hàm trong mục public của lớp. Mỗi chú thích về một hàm bao gồm mẫu hàm và các điều kiện trước, điều

kiện sau kèm theo mỗi hàm. Người sử dụng lớp chỉ cần đọc các

thông tin trong phần chú thích này. Tiếp theo là định nghĩa lớp. Chú

ý rằng, định nghĩa lớp cần đặt giữa các định hướng tiền xử lý # ifndef

# define … # endif. Chẳng hạn, định nghĩa lớp Complex như sau:

# ifndef COMPLEX\_H

# define COMPLEX\_H

class Complex

{

// danh sách thành phần

} ;

# endif

File đầu của lớp Complex được cho trong hình 2.2. Cần lưu ý rằng,

trong lớp Complex đó, chúng ta mới chỉ đưa vào một số ít phép toán,

để thuận lợi cho việc tiến hành các thao tác số phức, lớp Complex

thực tế cần phải chứa rất nhiều phép toán khác, song để cho ngắn

gọn, chúng ta đã không đưa vào.

// File đầu Complex.h

// Các hàm kiến tạo :

// Complex (double a = 0.0, double b = 0.0) ;

// Postcondition: số phức được tạo ra có phần thức là a, phần ảo là b.

// Các hàm thành phần khác:

// double GetReal ( ) const ;

// Trả về phần thực của số phức.

// double GetImag ( ) const ;

// Trả về phần ảo của số phức.

// double GetAbs ( ) const ;

// Trả về giá trị tuyệt đối của số phức.

// Các hàm bạn:

// friend Complex & operator + (const Complex & c1,

// const Comple & c2) ;

// Trả về tổng c1 + c2 của số phức c1 và c2.

// friend Complex & operator - (const Complex & c1,

// const Complex & c2);

// Trả về hiệu c1 – c2 của số phức c1 và c2.

// friend Complex & operator \* (const Complex & c1,

// const Complex & c2);

// Trả về tích c1 \* c2 của số phức c1 và c2.

// friend Complex & operator / (const Complex & c1,

// const Complex &c2);

// Trả về thương c1 / c2 của số phức c1 và c2.

// friend ostream & operator << (ostream & os, const Complex &c);

// Postcondition: số phức c được in ra dưới dạng a + ib, trong đó a là

// phần thực, b là phần ảo của c.

Bài 4: Lớp khuôn

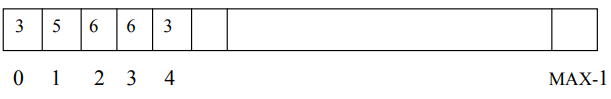
Trong mục này chúng ta sẽ trình bày khái niệm lớp khuôn (template class). Lớp khuôn là một công cụ quan trọng trong C + + được sử dụng để cài đặt các lớp phụ thuộc tham biến kiểu dữ liệu. Các KDLTT quan trọng mà chúng ta nghiên cứu trong các chương sau đều được cài đặt bởi lớp khuôn. Trước hết chúng ta xét một ví dụ về lớp côngtơnơ và cách cài đặt không sử dụng lớp khuôn.

1. Lớp côngtono

Trong nhiều ứng dụng, chúng ta cần sử dụng các KDLTT mà mỗi đối tượng dữ liệu của nó là một bộ sưu tập các phần tử dữ liệu cùng kiểu nào đó. Lớp cài đặt các KDLTT như thế được gọi là lớp côngtơnơ (container class). Như vậy, lớp côngtơnơ là một thuật ngữ để chỉ các lớp mà mỗi đối tượng của lớp là một “côngtơnơ” chứa một bộ sưu tập các dữ liệu cùng kiểu. Các lớp danh sách, hàng đội, ngăn xếp, … được nghiên cứu sau này đều là lớp côngtơnơ. Trong một chương trình của mình, bạn có thể cần đến lớp côngtơnơ các số nguyên, người khác lại cần sử dụng chính lớp côngtơnơ đó, chỉ khác một điều là các côngtơnơ của anh ta không phải là côngtơnơ các số nguyên mà là côngtơnơ các số thực, hoặc côngtơnơ các ký tự. Do đó, vấn đề đặt ra cho việc thiết kế các lớp côngtơnơ là: lớp cần được thiết kế như lớp phụ thuộc tham biến kiểu dữ liệu Item sao cho trong một chương trình bất kỳ, bạn có thể dễ dàng sử dụng lớp bằng cách chỉ ra Item được thay bởi một kiểu dữ liệu cụ thể, chẳng hạn int, double hoặc char… Sau đây là một ví dụ về một lớp côngtơnơ và cách cài đặt sử dụng mệnh đề định nghĩa kiểu typedef.

Ví dụ (KDLTT Túi). Trước hết chúng ta đặc tả KDLTT Túi. Mỗi đối tượng dữ liệu là một “túi” chứa một bộ sưu tập các phần tử cùng kiểu. Chẳng hạn, đối tượng dữ liệu có thể là túi bi, một túi có thể chứa 3 bi xanh, 1 bi đỏ, 2 bi vàng, … Một ví dụ khác, đối tượng dữ liệu có thể là côngtơnơ áo sơ mi, mỗi côngtơnơ có thể chứa nhiều áo thuộc cùng một loại áo (mỗi loại áo đặc trưng bởi kiểu dáng, loại vải, cỡ áo). Sau đây là các phép toán có thể thực hiện trên các túi, Trong các phép toán này, B là ký hiệu một túi, element là một phần tử cùng kiểu với kiểu của các phần tử trong túi. 1. Sum (B). Hàm trả về số phần tử trong túi B. 2. Insert (B, element). Thêm phần tử element vào túi B. 3. Remove (B, element). Lấy ra một phần tử là bản sao của element khỏi túi B. 4. Occurr (B, element). Hàm trả về số phần tử là bản sao của element trong túi B. 5. Union (B1, B2). Hợp của túi B1và túi B2. Hàm trả về một túi chứa tất cả các phần tử của B1 và tất cả các phần tử của túi B2. Dưới đây chúng ta đưa ra một cách cài đặt KDLTT Túi đã đặc tả ở trên.

Chúng ta sẽ sử dụng một mảng (tĩnh) data có cỡ là MAX để lưu các phần tử của túi. Các phần tử của túi được lưu trong các thành phần của mảng: data[0], data[1], …, data[size-1], trong đó size là số phần tử của túi. Chẳng hạn, một túi chứa 5 số nguyên: 2 số nguyên 3, 1 số nguyên 5, 2 số nguyên 6 có thể lưu trong mảng như sau:



Trong khai báo mảng data, cỡ của mảng MAX cần phải là hằng, MAX là số tối đa các phần tử mà túi có thể chứa, và nó là chung cho tất cả các đối tượng của lớp. Vì vậy, chúng ta sẽ khai báo MAX là hằng tĩnh trong lớp. Chẳng hạn, static const int MAX = 50 ; Mục tiêu thiết kế lớp túi (class bag) của chúng ta là sao cho người lập trình sử dụng lớp túi trong các hoàn cảnh khác nhau một cách thuận tiện: chỉ cần một vài thay đổi nhỏ, không cần viết lại toàn bộ mã của lớp. Trong một chương trình, người lập trình có thể sử dụng lớp túi số nguyên, trong chương trình khác lại có thể sử dụng lớp túi bi hoặc lớp túi áo sơ mi, … Các phép toán trên các đối tượng của các loại túi khác nhau là hoàn toàn như nhau, không phụ thuộc gì vào kiểu dữ liệu của các phần tử trong túi. Do đó, chúng ta đưa vào lớp mệnh đề định nghĩa kiểu, mệnh đề này xác định kiểu dữ liệu Item, Item là kiểu dữ liệu của các phần tử trong túi . Chẳng hạn, trong định nghĩa lớp túi số nguyên, bạn đưa vào mệnh đề:

typedef int Item ;

Lớp túi chứa hai biến thành phần: mảng data lưu các phần tử của túi

và biến size lưu số phần tử trong túi. Thiết kế ban đầu của lớp túi như sau:

class Bag

{

public :

static const int MAX = 50 ;

typedef int Item ;

// Các hàm thành phần

private :

Item data[MAX] ;

int size ;

} ;

Trong các hàm thành phần của lớp Bag, bất kỳ chỗ nào có mặt Item,

chương trình dịch sẽ hiểu đó là int. Với cách thiết kế này, nếu chương trình

của bạn cần đến lớp túi mà các đối tượng của nó chứa các ký tự, bạn chỉ cần

thay int trong mệnh đề typedef bởi char. Còn nếu bạn muốn sử dụng các túi

có cỡ lớn hơn, bạn chỉ cần xác định lại hằng MAX. Cần lưu ý rằng, hằng

MAX và kiểu Item được xác định trong phạm vi lớp Bag. Vì vậy, ở ngoài

phạm vi lớp Bag, để truy cập tới chúng, bạn cần sử dụng toán tử định phạm

vi. Chẳng hạn, trong chương trình để in ra cỡ tối đa của túi, bạn cần viết

cout << “cỡ tối đa của túi là:” << Bag :: MAX << endl ;

Bây giờ chúng ta thiết kế các hàm thành phần của lớp Bag. Trước hết,

cần có hàm kiến tạo mặc định sau

Bag ( ) ;

Hàm này khởi tạo ra một túi rỗng.

Các phép toán (1)-(4) của KDLTT Túi được cài đặt bởi các hàm thành

phần tương ứng của lớp Bag như sau:

int Sum ( ) const ;

void Insert (const Item & element) ;

void Remove (const Item & element) ;

int Occurr (const Item & element) ;

Phép toán (5) được cài đặt bởi hàm toán tử bạn:

friend Bag& operator + (const Bag & B1, const Bag & B2);

Với hàm toán tử này, chúng ta đã xác định phép + trên các túi. Do đó,

để thuận tiện khi cần cộng thêm một túi vào một túi khác, chúng ta đưa vào

lớp Bag một hàm toán tử thành phần operator += như sau:

void operator += (const Bag & B1) ;

Không cần đưa vào lớp Bag hàm kiến tạo copy, hàm toán tử gán, vì

chỉ cần sử dụng hàm kiến tạo copy tự động, toán tử gán tự động là đủ.

Đến đây chúng ta có thể viết ra file đầu của lớp Bag. File đầu được

cho trong hình 2.6.

// File đầu bag.h

// Sau đây là các thông tin người lập trình cần biết

// khi sử dụng lớp Bag.

// MAX là số tối đa các phần tử mà một Bag có thể chứa.

// Item là kiểu dữ liệu của các phần tử trong túi.

// Item có thể là một kiểu bất kỳ có sẵn trong C + + (chẳng hạn int,

// double, char, …), hoặc có thể là một lớp trong đó được cung cấp

// hàm kiến tạo mặc định, các hàm toán tử =, == và !=

// Sau đây là các hàm thành phần của lớp Bag:

// Bag( ) ;

// Postcondition: một Bag rỗng được khởi tạo.

// int Sum ( ) const ;

// Postcondition: Trả về tổng số phần tử của túi.

// void Insert(const Item& element) ;

// Hàm thực hiện thêm element vào túi B.

// Precondition: B.Sum ( ) < MAX

// Postcondition: một bản sao của element được thêm vào túi B.

// void Remove (const Item& element) ;

// Hàm loại một phần tử khỏi túi B.

// Postcondition: một phần tử là bản sao của element bị loại khỏi túi B,

// nếu trong B có chứa, và B không thay đổi nếu trong B không chứa.

// int Occurr (const Item& element) ;

// Postcondition: Trả về số lần xuất hiện của element trong túi B.

// void operator += (const Bag& B1)

// Hàm thực hiện cộng túi B1 vào túi B, B += B1

// Postcondition: tất cả các phần tử của túi B1 được thêm vào túi B.

// Hàm toán tử bạn của lớp Bag:

// friend Bag& operator + (const Bag& B1, const Bag& B2) ;

// Precondition: B1.Sum ( ) + B2.Sum( ) <= MAX

// Postcondition: Trả về túi B là hợp của túi B1 và B2.

// Bạn có thể sử dụng toán tử gán = và hàm kiến tạo copy cho các

// đối tượng của lớp Bag.

# ifdef BAG\_H

# define BAG\_H

class Bag

{

public:

static const int MAX = 50 ;

typdef int Item ;

Bag ( ) {size = 0 ;}

int Sum( ) const

{return size ;}

int Occurr (const Item& element) const ;

void Insert (const Item& element) ;

void Remove (const Item& element) ;

void operator += (const Bag & B1) ;

friend Bag & operator + (const Bag & B1,const Bag & B2);

private :

Item data[MAX];

int size ;

} ;

# endif

Hàm Remove: Để loại được phần tử element khỏi túi, trước hết ta

tìm vị trí đầu tiên trong mảng data, tại đó có lưu element. Điều này

được thực hiện bằng cách cho biến i chạy từ 0 tới size – 1, nếu gặp

một thành phần của mảng mà data[i] == element thì dừng lại, còn

nếu i chạy tới size thì điều đó có nghĩa là túi không chứa element.

Bước tìm vị trí đầu tiên lưu element được thực hiện bởi vòng lặp:

For ( i = 0; ( i < size ) && (data[i] != element ); i + + ) ;

Việc loại element ở vị trí thứ i trong mảng data được thực hiện

bằng cách chuyển phần tử lưu ở vị trí cuối cùng data[size - 1] tới vị

trí i, và giảm số phần tử của túi i đi 1. Tức là:

data[i] = data[size - 1] ;

size -- ;

Hàm Occurr: Chúng ta cần đếm số lần xuất hiện của phần tử

element trong túi. Muốn vậy, chúng ta sử dụng một biến đếm count,

cho biến chỉ số i chạy từ đầu mảng tới vị trí size – 1, cứ mỗi lần gặp

phần tử element thì biến đếm count được tăng lên 1.

count = 0 ;

for (i = 0; i < size; i + +)

if ( data[i] = = element)

count + +;

Hàm toán tử operator += : Nhiệm vụ của hàm là “đổ” nội dung của

túi B1 vào túi B. Chỉ có thể thực hiện được điều đó khi mà tổng số

phần tử của hai túi không vượt quá dung lượng MAX. Nếu điều kiện

đó đúng, chúng ta sao chép lần lượt các phần tử của túi B1 vào các

thành phần mảng data của túi B, bắt đầu từ vị trí size. Hành động đó

được thực hiện bởi vòng lặp:

for (i =0; i < B1.size; i + +)

{

data[size] = B1.data[i] ;

size + + ;

}

• Hàm toán tử bạn operator +. Muốn gộp hai túi thành một túi,

đương nhiên cũng cần điều kiện tổng số phần tử của hai túi không

lớn hơn MAX. Chúng ta khởi tạo một túi B rỗng, rồi áp dụng toán tử

+= đổ lần lượt túi B1, B2 vào túi B, tức là:

Bag B;

B += B1;

B += B2;

Sử dụng hàm tiện ích assert.

Trong nhiều hàm, để hàm thực hiện được nhiệm vụ của mình, các dữ

liệu vào cần thoả mãn các điều kiện được liệt kê trong phần chú thích

Precondition ở ngay sau mẫu hàm. Khi cài đặt các hàm, chúng ta có thể sử

dụng mệnh đề if (Precondition) ở đầu thân hàm, như chúng ta đã làm khi nói

về hàm Insert. Một cách lựa chọn khác là sử dụng hàm assert trong thư viện

chuẩn assert.h. Hàm assert có một đối số, đó là một biểu thức nguyên, hoặc

thông thường là một biểu thức logic. Hàm assert sẽ đánh giá biểu thức, nếu

biểu thức có giá trị true, các lệnh tiếp theo trong thân hàm sẽ được thực hiện.

Nếu biểu thức có giá trị false, assert sẽ in ra một thông báo lỗi và chương

1. Hàm khuôn

Xét vấn đề tìm giá trị lớn nhất (nhỏ nhất) trong một mảng. Thuật toán đơn giản là đọc tuần tự các thành phần của mảng và lưu lại vết của phần tử lớn nhất (nhỏ nhất). Rõ ràng, logic của thuật toán này chẳng phụ thuộc gì vào kiểu dữ liệu của các phần tử trong mảng. Các thuật toán sắp xếp mảng cũng có đặc trưng đó. Một ví dụ khác: vấn đề trao đổi giá trị của hai biến. Thuật toán trao đổi giá trị của hai biến mà chúng ta đã quen biết cũng không phụ thuộc vào kiểu dữ liệu của hai biến đó. Chúng ta mong muốn viết ra các hàm cài đặt các thuật toán đó sao cho hàm có thể sử dụng được cho các loại kiểu dữ liệu khác nhau. Một cách lựa chọn là sử dụng mệnh đề định nghĩa kiểu typedef. Chẳng hạn, hàm trao đổi giá trị của hai biến có thể viết như sau:

typedef int Item ;

void Swap (Item & x, Item & y)

{ Item temp;

temp = x ;

x = y ;

y = temp ;

}

Hàm trên để trao đổi hai biến nguyên. Nếu cần trao đổi hai biến ký tự,

bạn thay int bởi char trong mệnh đề typedef. Nhưng cách tiếp cận này có vấn

đề, khi trong cùng một chương trình bạn vừa cần trao đổi hai biến nguyên,

vừa cần trao đổi hai biến ký tự, bởi vì bạn không thể hai lần định nghĩa kiểu

Item trong cùng một chương trình.

Một cách tiếp cận khác khắc phục được hạn chế của cách tiếp cận trên

là sử dụng hàm khuôn. Chẳng hạn, hàm khuôn Swap được viết như sau:

template <class Item>

void Swap (Item & x, Item & y)

// Item cần phải là kiểu bất kỳ có sẵn trong C + + (int, char, …)

// hoặc là lớp có hàm kiến tạo mặc định và toán tử gán.

{

Item temp ;

temp = x ;

x = y ;

y = temp;

}

Để xác định một hàm khuôn, bạn chỉ cần đặt biểu thức template

<class Item> ngay trước mẫu hàm. Biểu thức đó nói rằng, ở mọi chỗ Item

xuất hiện trong định nghĩa hàm, Item là một kiểu dữ liệu nào đó chưa được

xác định. Nó sẽ được hoàn toàn xác định khi ta sử dụng hàm (gọi hàm).

Chương trình đơn giản sau minh họa cách sử dụng hàm khuôn Swap.

#include <iostream>

template <typename T>

void Swap(T& a, T& b)

{

T temp = a;

a = b;

b = temp;

}

int main() {

int a = 3;

int b = 5;

double x = 4.2;

double y = 3.6;

Swap(a, b); // Hoán đ?i giá tr? c?a hai s? nguyên

Swap(x, y); // Hoán đ?i giá tr? c?a hai s? th?c

std::cout << "a = " << a << ", b = " << b << std::endl;

std::cout << "x = " << x << ", y = " << y << std::endl;

return 0;

}

Một ví dụ khác về hàm khuôn: hàm tìm giá trị lớn nhất trong mảng

#include <iostream>

#include <cassert>

template <class Item>

Item& FindMax(Item data[], int n) {

assert(n > 0); // Đ?m b?o r?ng m?ng có ít nh?t m?t ph?n t?.

int index = 0; // Ch? s? c?a ph?n t? l?n nh?t.

for (int i = 1; i < n; i++) {

if (data[index] < data[i]) {

index = i;

}

}

return data[index];

}

int main() {

int intArray[] = {3, 7, 1, 9, 5};

double doubleArray[] = {2.3, 4.1, 1.7, 5.5};

int intMax = FindMax(intArray, 5);

double doubleMax = FindMax(doubleArray, 4);

std::cout << "Max int value: " << intMax << std::endl;

std::cout << "Max double value: " << doubleMax << std::endl;

return 0;

}

Điểm quan trọng ở đây là bạn đã thêm thư viện **#include <iostream>** và **#include <cassert>** để sử dụng **assert**. Ngoài ra, bạn cũng đã thay đổi vòng lặp **for** để đảm bảo rằng nó hoạt động đúng và sử dụng **data[index]** để truy xuất giá trị lớn nhất. Cuối cùng, trong hàm **main()**, bạn đã thử nghiệm việc tìm giá trị lớn nhất cho cả mảng số nguyên và mảng số thực.

1. Lớp khuôn

Hàm khuôn là hàm phụ thuộc tham biến kiểu dữ liệu. Tương tự, nhờ lớp khuôn chúng ta xây dựng được lớp phụ thuộc tham biến kiểu dữ liệu.

Định nghĩa lớp khuôn

Xác định một lớp khuôn cũng tương tự như xác định một hàm khuôn. Cần đặt biểu thức template ngay trước định nghĩa lớp, Item là tham biến kiểu của lớp. Trong định nghĩa lớp, chỗ nào có mặt Item, chúng ta chỉ cần hiểu Item là một kiểu dữ liệu nào đó. Item là kiểu cụ thể gì (là int hay double, …) sẽ được xác định khi chúng ta sử dụng lớp trong chương trình

Ví dụ ( Lớp khuôn Bag). Từ lớp Bag được xây dựng bằng cách sử dụng mệnh đề định nghĩa kiểu typedef int Item (xem hình 2.6), chúng ta có thể xây dựng lớp khuôn Bag như sau: template class Bag { // Bỏ mệnh đề typedef int Item; } ;

Trong định nghĩa lớp khuôn Bag, tất cả các hàm thành phần của nó là các hàm thành phần của lớp Bag cũ (lớp trong hình 2.6), không có gì thay đổi. Riêng hàm operator +, nó không phải là hàm thành phần của lớp, nên trong mẫu hàm của nó, Bag (với tư cách là tên kiểu dữ liệu) cần phải được viết là Bag . Cụ thể là: friend Bag & operator + (const Bag & B1, const Bag & B2); Tổng quát, ở bên ngoài định nghĩa lớp khuôn Bag ( chẳng hạn, ở trong file cài đặt lớp khuôn Bag), bất kỳ chỗ nào Bag xuất hiện với tư cách là tên lớp ( trong toán tử định phạm vi Bag ::). hoặc với tư cách là tên kiểu dữ liệu, chúng ta cần phải viết là Bag để chỉ rằng, Bag là lớp phụ thuộc tham biến kiểu Item. Cài đặt các hàm. Tất cả các hàm thao tác trên các đối tượng của lớp khuôn ( bao gồm các hàm thành phần, các hàm bạn, các hàm không thành phần) đều phải được cài đặt là hàm khuôn. Chẳng hạn, hàm toán tử += của

lớp Bag cần được cài đặt như sau:

template <class Item>

void Bag <Item> :: operator += (const Bag <Item> & B1)

{

int i;

int a = B1. size;

assert (Sum( ) + B1. Sum( ) < = MAX);

for (i = 0; i < a; i + +)

{

data [size] = B1.data [i];

size + + ;

}

}

Tổ chức các file. Như khi cài đặt lớp bình thường, cài đặt lớp khuôn

cũng được tổ chức thành hai file: file đầu và file cài đặt. Chỉ có một điều cần

lưu ý là, hầu hết các chương trình dịch đòi hỏi phải đưa tên file cài đặt vào

trong file đầu bởi mệnh đề # include “tên file cài đặt”. Hình 2.8 là file đầu

của lớp khuôn Bag, file cài đặt ở trong hình 2.9.

// File đầu: bagt.h

// Các thông tin cần thiết để sử dụng lớp Bag, các chú thích về các

// hàm hoàn toàn giống như trong file đầu bag.h (xem hình 2.6).

# ifndef BAG\_H

# define BAG\_H

template <class Item>

class Bag

{

public :

static const int MAX = 50;

Bag( ) {size = 0; ]

int Sum( ) const

{ return size; }

int Occurr (const Item & element) const;

void Insert (const Item & element);

void Remove (const Item & element);

void operator += (const Bag & B1);

friend Bag< Item> & operator + (const Bag<Item> & B1,

const Bag<Item> & B2);

private :

Item data[MAX];

int size;

};

# include “bag.template”

# endif

// File cài đặt: bag.template

#include <assert.h>

template <class Item>

int Bag<Item>::Occurr(const Item &element) const

{

int count = 0;

for (int i = 0; i < size; i++)

{

if (data[i] == element)

{

count++;

}

}

return count;

}

template <class Item>

void Bag<Item>::Insert(const Item &element)

{

assert(size < MAX);

data[size] = element;

size++;

}

template <class Item>

void Bag<Item>::Remove(const Item &element)

{

int i = 0;

while (i < size && data[i] != element)

{

i++;

}

if (i == size)

{

return;

}

data[i] = data[size - 1];

size--;

}

template <class Item>

void Bag<Item>::operator+=(const Bag<Item> &B1)

{

int a = B1.size;

assert(size + a <= MAX);

for (int i = 0; i < a; i++)

{

data[size] = B1.data[i];

size++;

}

}

template <class Item>

Bag<Item> operator+(const Bag<Item> &B1, const Bag<Item> &B2)

{

assert(B1.Sum() + B2.Sum() <= Bag<Item>::MAX);

Bag<Item> B;

B += B1;

B += B2;

return B;

}

Sử dụng lớp khuôn. Trong một chương trình, để sử dụng một lớp

khuôn, trước hết bạn cần đưa vào mệnh đề # include “tên file đầu”, chẳng

hạn # include “bag.h”. Khi khai báo các đối tượng của lớp, bạn cần thay thế

tham biến kiểu bởi kiểu thực tế. Chẳng hạn, biểu thức Bag<Item> nói rằng,

Bag là lớp khuôn với tham biến kiểu Item, nên trong chương trình nếu bạn

muốn sử dụng đối tượng A là túi số nguyên, bạn cần khai báo:

Bag<int> A;

Chương trình demo sử dụng lớp khuôn Bag được cho trong hình 2.10.\

Bài 5: Các kiểu dữ liệu trừu tượng quan trọng

Trong mục này chúng ta sẽ giới thiệu các KDLTT quan trọng nhất.

Các KDLTT này được sử dụng thường xuyên trong thiết kế các thuật toán.

Các KDLTT này sẽ được lần lượt nghiên cứu trong các chương tiếp theo.

Trong chương 1 chúng ta đã thảo luận về tầm quan trọng của sự trừu

tượng hoá dữ liệu trong thiết kế thuật toán. Sự trừu tượng hoá dữ liệu được

thực hiện bằng cách xác định các KDLTT. Một KDLTT là một tập các đối

tượng dữ liệu cùng với một tập phép toán có thể thực hiện trên các đối tượng

dữ liệu đó. Các đối tượng dữ liệu trong thế giới hiện thực rất đa dạng và có

thể rất phức tạp. Để mô tả chính xác các đối tượng dữ liệu, chúng ta cần sử

dụng các khái niệm toán học, các mô hình toán học.

Tập hợp là khái niệm cơ bản trong toán học, nó là cơ sở để xây dựng nên nhiều khái niệm toán học khác. Tuy nhiên, tập hợp trong toán học là cố định. Còn trong các chương trình, thông thường chúng ta cần phải lưu một tập các phần tử dữ liệu, tập này sẽ thay đổi theo thời gian trong quá trình xử lý, do chúng ta thêm các phần tử dữ liệu mới vào và loại các phần tử dữ liệu nào đó khỏi tập. Chúng ta sẽ gọi các tập như thế là tập động (dynamic set). Giả sử rằng, các phần tử của tập động chứa một trường được gọi là khoá (key) và trên các giá trị khoá có quan hệ thứ tự (chẳng hạn khoá là số nguyên, số thực, ký tự,…). Chúng ta cũng giả thiết rằng, các phần tử khác nhau của tập động có khoá khác nhau.

KDLTT tập động (dynamic set ADT) được xác định như sau: Mỗi

đối tượng dữ liệu của kiểu này là một tập động. Dưới đây là các phép toán

của KDLTT tập động, trong các phép toán đó, S ký hiệu một tập, k là một

giá trị khoá và x là một phần tử dữ liệu.

1. Insert(S, x). Thêm phần tử x vào tập S.

2. Delete(S, k). Loại khỏi tập S phần tử có khoá k.

3. Search(S, k). Tìm phần tử có khoá k trong tập S.

4. Max(S). Trả về phần tử có khoá lớn nhất trong tập S.

5. Min(S). Trả về phần tử có khoá nhỏ nhất trong tập S.

Ngoài các phép toán chính trên, còn có một số phép toán khác trên tập

động.

Trong nhiều áp dụng, chúng ta chỉ cần đến ba phép toán Insert, Delete

và Search. Các tập động với ba phép toán này tạo thành KDLTT từ điển

(dictionary ADT).

Một khái niệm quan trọng khác trong giải tích toán học là khái niệm

dãy (a1, …, ai, …). Dãy khác tập hợp ở chỗ, các phần tử của dãy được sắp

xếp theo một thứ tự xác định: a1 là phần tử đầu tiên, ai là phần tử ở vị trí thứ

i (i = 1, 2, …), một phần tử có thể xuất hiện ở nhiều vị trí khác nhau trong

dãy. Chúng ta sẽ gọi một dãy hữu hạn là một danh sách. Sử dụng danh sách

với tư cách là đối tượng dữ liệu, chúng ta cũng cần đến các thao tác Insert,

Delete. Tuy nhiên, các phép toán Insert và Delete trên danh sách có khác với

các phép toán này trên tập động. Nếu ký hiệu L là danh sách thì phép toán

Insert (L, x, i) có nghĩa là xen phần tử x và vị trí thứ i trong danh sách L, còn

Delete (L, i) là loại phần tử ở vị trí thứ i khỏi danh sách L. Trên danh sách

còn có thể tiến hành nhiều phép toán khác. Các danh sách cùng với các phép

toán trên danh sách tạo thành KDLTT danh sách (list ADT). KDLTT danh

sách sẽ được nghiên cứu trong chương 4 và 5.

Trong nhiều trường hợp, khi thiết kế thuật toán, chúng ta chỉ cần sử

dụng hạn chế hai phép toán Insert và Delete trên danh sách. Các danh sách

với hai phép toán Insert và Delete chỉ được phép thực hiện ở một đầu danh

sách lập ra một KDLTT mới: KDLTT ngăn xếp (stack ADT). Nếu chúng

ta xét các danh sách với phép toán Insert chỉ được phép thực hiện ở một đầu

danh sách, còn phép toán Delete chỉ được phép thực hiện ở một đầu khác

của danh sách, chúng ta có KDLTT hàng đợi (queue ADT). Ngăn xếp

được nghiên cứu trong chương 6, hàng đợi trong chương 7.

Cây là một tập hợp với cấu trúc phân cấp. Một dạng cây đặc biệt là

cây nhị phân, trong cây nhị phân mỗi đỉnh chỉ có nhiều nhất hai đỉnh con

được phân biệt là đỉnh con trái và đỉnh con phải. KDLTT cây nhị phân

(binary tree ADT) sẽ được nghiên cứu trong chương 9.

KDLTT hàng ưu tiên (priority queue ADT) là một biến thể của

KDLTT từ điển. Hàng ưu tiên là một tập động với ba phép toán Insert,

FindMin, DeleteMin. Hàng ưu tiên sẽ được trình bày trong chương 11.

Các KDLTT trên đây đóng vai trò cực kỳ quan trọng trong thiết kế

chương trình, đặc biệt KDLTT từ điển, bởi vì trong phần lớn các chương

trình chúng ta cần lưu một tập dữ liệu rồi thì tiến hành tìm kiếm dữ liệu và

cập nhật tập dữ liệu đó (bởi xen, loại dữ liệu).

Như chúng ta đã nhấn mạnh trong chương 2, một KDLTT có thể có

nhiều cách cài đặt. Chúng ta sẽ cài đặt các KDLTT bởi các lớp C + +, và sẽ

phân tích, đánh gía hiệu quả của các phép toán trong mỗi cách cài đặt.

Chúng ta có thể cài đặt danh sách bởi mảng tĩnh, bởi mảng động hoặc bởi

CTDL danh sách liên kết (chương 4).

Có nhiều cách cài đặt tập động. Có thể cài đặt tập động bởi danh sách,

danh sách được sắp (chương 4). Cài đặt tập động bởi cây tìm kiếm nhị

phân (binary search tree) sẽ được trình bày trong chương 9. Đó là một

trong các cách cài đặt tập động đơn giản và hiệu quả.

Bảng băm (hashing table) là một CTDL đặc biệt thích hợp cho cài

đặt từ điển. Cài đặt từ điển bởi bảng băm là nội dung của chương 10.

***Bài tập***

1. Hãy nói rõ sự khác nhau giữa hai mục public và private của một lớp

2. Hãy mô tả vai trò của hàm kiến tạo và hàm huỷ? Có các loại hàm kiến tạo nào?

3. Sự khác nhau giữa hàm kiến tạo copy và toán tử gán?

4. Nếu ta không cung cấp cho lớp hàm kiến tạo và hàm huỷ thì kết quả

sẽ như thế nào?

5. Khi nào trong một lớp nhất thiết phải đưa vào hàm huỷ, hàm kiến tạo

copy, toán tử gán?

6. Sự khác nhau giữa hàm thành phần và hàm bạn của một lớp?

7. Hãy cài đặt lớp xâu ký tự theo các chỉ dẫn sau. Xâu ký tự được biểu

diễn bởi mảng động. Lớp cần chứa các hàm thực hiện các công việc

sau: xác định độ dài của xâu, truy cập tới ký tự thứ i trong xâu, kết nối

hai xâu thành một xâu, các phép toán so sánh hai xâu, đọc và viết ra

xâu. Tất cả các hàm cần phải cài đặt là hàm toán tử, chỉ trừ hàm xác

định độ dài của xâu.

Chương 3: Sự thừa kế

Một trong các đặc trưng quan trọng nhất của C + + và các ngôn ngữ lập trình định hướng đối tượng khác là cho phép chúng ta có thể sử dụng lại các thành phần mềm. Trong mục 2.4 chúng ta đã trình bày một phương pháp thực hiện sử dụng lại các thành phần mềm bằng cách xây dựng các lớp khuôn. Chương này sẽ trình bày một phương pháp khác: sử dụng lại các thành phần mềm thông qua tính **thừa kế (inheritance)**. Sử dụng tính thừa kế, chúng ta có thể xây dựng nên các lớp mới từ các lớp đã có, tránh phải viết lại các thành phần mềm đã có.

Bài 1: Các lớp dẫn xuất

Khi xây dựng một lớp mới, trong nhiều trường hợp lớp mới cần xây dựng có nhiều điểm giống một lớp đã có. Khi đó trên cơ sở lớp đã có, bằng cách sử dụng tính thừa kế, chúng ta có thể xây dựng nên lớp mới. Lớp đã có được gọi là **lớp cơ sở (base class)**, lớp mới được xây dựng nên từ lớp cơ sở được gọi là **lớp dẫn xuất (derived class)**. Một lớp dẫn xuất có thể được thừa kế từ nhiều lớp cơ sở, điều này được gọi là **tính đa thừa kế (multiple inheritance)**. Song để đơn giản cho trình bày, sau đây chúng ta chỉ đề cập tới sự thiết kế lớp dẫn xuất thừa kế từ một lớp cơ sở. Tính thừa kế cho phép ta sử dụng lại các thành phần mềm khi chúng ta xây dựng một lớp mới. Lớp dẫn xuất có thể thừa kế các thành phần dữ liệu và các hàm thành phần từ lớp cơ sở, trừ các hàm kiến tạo và hàm huỷ. Lớp dẫn xuất có thể thêm vào các thành phần dữ liệu mới và các hàm thành phần mới cần thiết cho các phép toán của nó. Ngoài ra, lớp dẫn xuất còn có thể xác định lại bất kỳ hàm thành phần nào của lớp cơ sở cho phù hợp với

các đặc điểm của lớp dẫn xuất.

Cú pháp xác định một lớp dẫn xuất như sau: Đầu lớp bắt đầu bởi từ

khoá class, sau đó là tên lớp dẫn xuất, rồi đến dấu hai chấm, theo sau là từ

khoá chỉ định dạng thừa kế (public, private, protected), và cuối cùng là tên

lớp cơ sở. Chẳng hạn, nếu ta muốn xác định một lớp dẫn xuất D từ lớp cơ sở

B thì có thể sử dụng một trong ba khai báo sau:

class D : public B { … } ;

class D : private B { … } ;

class D : protected B { … } ;

Chúng ta sẽ nói tới đặc điểm của các dạng thừa kế ở cuối mục này,

còn bây giờ chúng ta sẽ xét một ví dụ minh hoạ. Giả sử chúng ta muốn xây

dựng lớp Ball (lớp quả bóng) từ lớp Sphere (lớp hình cầu). Giả sử lớp hình

cầu được xác định như sau:

class Sphere

{

public :

Sphere (double R = 1) ;

double Radius ( ) const ;

double Area ( ) const ;

double Volume ( ) const ;

void WhatIsIt ( ) const ;

private :

double radius ;

}

Lớp Sphere chỉ có một thành phần dữ liệu radius là bán kính của hình

cầu, và các hàm thành phần: hàm kiến tạo ra hình cầu có bán kính R, hàm

cho biết bán kính hình cầu Radius ( ), hàm tính diện tích hình cầu Area ( ) và

hàm tính thể tích hình cầu Volume ( ), cuối cùng là hàm WhatIsIt ( ) cho ta

câu trả lời rằng đối tượng được hỏi là hình cầu có bán kính là bao nhiêu.

Hàm WhatIsIt ( ) được cài đặt như sau:

void Sphere :: WhatIsIt ( ) const

{

cout << “It is the sphere with the radius”

<< radius ;

}

Bởi vì mỗi quả bóng là một hình cầu, nên chúng ta có thể xây dựng

lớp Ball như là lớp dẫn xuất từ lớp Sphere. Lớp Ball thừa kế tất cả các thành

phần của lớp Sphere, trừ ra hàm kiến tạo và xác định lại hàm WhatIsIt ( ).

Chúng ta sẽ thêm vào lớp Ball một thành phần dữ liệu mới madeof để chỉ

quả bóng được làm bằng chất liệu gì: cao su, nhựa hay gỗ. Một hàm thành

phần mới cũng được thêm vào lớp Ball, đó là hàm Madeof ( ) trả về chất liệu

tạo thành quả bóng. Lớp Ball được khai báo như sau:

class Ball : public Sphere

{

public :

enum Materials {RUBBER, PLASTIC, WOOD};

Ball (double R = 1, Materials M = RUBBER) ;

Materials MadeOf ( ) const ;

void WhatIsIt ( ) const ;

private :

Materials madeOf ;

} ;

Lớp Ball được định nghĩa như trên sẽ có hai thành phần dữ liệu:

radius được thừa kế từ lớp Sphere và madeof mới được đưa vào. Ngoài hàm

kiến tạo, lớp Ball có ba hàm thành phần được thừa kế từ lớp Sphere, đó là

các hàm Radius ( ), Area ( ) và Volume( ), một hàm thành phần mới là hàm

MadeOf( ), và hàm thành phầnWhatIsIt( ) mới, nó định nghĩa lại một hàm

cùng tên đã có trong lớp cơ sở Sphere. Hàm WhatIsIt( ) trong lớp Ball được

định nghĩa như sau:

void Ball :: WhatIsIt ( ) const

{

Sphere :: WhatIsIt( ) ;

cout << “ and made of ” << madeof ;

}

class Ball : public Sphere

{

public :

enum Materials {RUBBER, PLASTIC, WOOD};

Ball (double R = 1, Materials M = RUBBER) ;

Materials MadeOf ( ) const ;

void WhatIsIt ( ) const ;

private :

Materials madeOf ;

} ;

Lớp Ball được định nghĩa như trên sẽ có hai thành phần dữ liệu:

radius được thừa kế từ lớp Sphere và madeof mới được đưa vào. Ngoài hàm

kiến tạo, lớp Ball có ba hàm thành phần được thừa kế từ lớp Sphere, đó là

các hàm Radius ( ), Area ( ) và Volume( ), một hàm thành phần mới là hàm

MadeOf( ), và hàm thành phầnWhatIsIt( ) mới, nó định nghĩa lại một hàm

cùng tên đã có trong lớp cơ sở Sphere. Hàm WhatIsIt( ) trong lớp Ball được

định nghĩa như sau:

void Ball :: WhatIsIt ( ) const

{

Sphere :: WhatIsIt( ) ;

cout << “ and made of ” << madeof ;

}

**Hàm kiến tạo của lớp dẫn xuất**. Nếu chúng ta không cung cấp cho

lớp dẫn xuất hàm kiến tạo, thì chương trình dịch sẽ tự động cung cấp hàm

kiến tạo mặc định tự động. Nhưng cũng như đối với một lớp bất kỳ, khi thiết

kế một lớp dẫn xuất, nói chung chúng ta cần phải cung cấp cho lớp dẫn xuất

hàm kiến tạo. Bây giờ chúng ta xét xem hàm kiến tạo của lớp dẫn xuất được

cài đặt như thế nào? Chú ý rằng, lớp dẫn xuất chứa các thành phần dữ liệu

được thừa kế từ lớp cơ sở, ngoài ra nó còn chứa các thành phần dữ liệu mới,

trong các thành phần dữ liệu mới này có thể có thành phần dữ liệu là đối

tượng của một lớp khác. Nhưng trong lớp dẫn xuất, chúng ta không được

quyền truy cập trực tiếp đến các thành phần dữ liệu của lớp cơ sở (và các

thành phần dữ liệu của lớp khác). Vậy làm thế nào để khởi tạo các thành

phần dữ liệu của lớp cơ sở (và các thành phần dữ liệu của lớp khác). Vấn đề

này được giải quyết bằng cách cung cấp một **danh sách khởi tạo**

**(initiazation list)**. Danh sách khởi tạo là danh sách các lời gọi hàm kiến tạo

của lớp cơ sở (và các hàm kiến tạo của các lớp khác). Danh sách này được

đặt ngay sau đầu hàm kiến tạo của lớp dẫn xuất. Ví dụ, hàm kiến tạo của lớp

dẫn xuất Ball được cài đặt như sau:

Ball :: Ball (double R, Materials M)

: Sphere (R)

{ madeof = M; }

Lưu ý rằng, ngay trước danh sách khởi tạo phải có dấu hai chấm :,

trong ví dụ trên danh sách khởi tạo chỉ có một lời gọi hàm kiến tạo lớp cơ sở

Sphere (R), nếu có nhiều lời gọi hàm thì cần có dấu phẩy giữa các lời gọi

hàm.

***Các mục public, private và protected của một lớp***

Trong các ví dụ mà chúng ta đưa ra từ trước tới nay, các thành phần

của lớp được đưa vào hai mục: public và private. Các thành phần nằm trong

mục public là các thành phần công khai, khách hàng của lớp có thể sử dụng

trực tiếp các thành phần này. Các thành phần nằm trong mục private là các

thành phần cá nhân của lớp, chỉ được phép sử dụng trong phạm vi lớp. Song

khi chúng ta thiết kế một lớp làm cơ sở cho các lớp dẫn xuất khác, chúng ta

mong muốn rằng một số thành phần của lớp, khách hàng không được quyền

sử dụng, nhưng cho phép các lớp dẫn xuất được quyền sử dụng. Muốn vậy

chúng ta đưa các thành phần đó vào mục protected. Như vậy các thành phần

nằm trong mục protected là các thành phần được bảo vệ đối với khách hàng,

nhưng các lớp dẫn xuất được quyền truy cập. Hình 3.1 minh hoạ quyền truy

cập đến các mục public, protected và private của một lớp. Đến đây chúng ta

có thể đưa ra cấu trúc tổng quát của một định nghĩa lớp:

class tên\_lớp

{

public:

danh sách các thành phần công khai

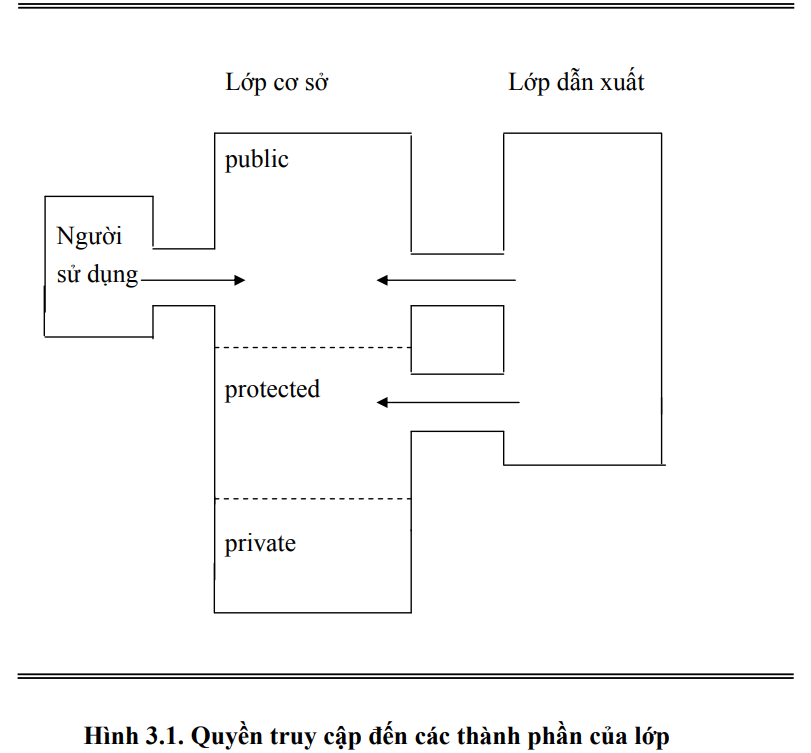
protected :

danh sách các thành phần được bảo vệ

private :

danh sách các thành phần cá nhân

};



***Các dạng thừa kế public, private và protected***

Khi xây dựng một lớp dẫn xuất từ một lớp cơ sở, chúng ta có thể sử

dụng một trong ba dạng thừa kế: public, private hay protected. Tức là, định

nghĩa một lớp dẫn xuất được bắt đầu bởi:

class tên\_lớp\_dẫn\_xuất : dạng\_thừa\_kế tên\_lớp\_cơ\_sở

Trong đó, dạng\_thừa\_kế là một trong ba từ khoá chỉ định dạng thừa

kế: public, private, protected. Dù dạng thừa kế là gì (là public hoặc private

hoặc protected) thì lớp dẫn xuất đều có quyền truy cập đến các thành phần

trong mục public và protected của lớp cơ sở. Như chúng ta đã nhấn mạnh,

lớp dẫn xuất được thừa kế các thành phần của lớp cơ sở, trừ ra các hàm kiến

tạo, hàm huỷ và các hàm được định nghĩa lại. Vấn đề được đặt ra là, quyền

truy cập đến các thành phần được thừa kế của lớp dẫn xuất như thế nào? Câu

trả lời phụ thuộc vào dạng thừa kế. Sau đây là các quy tắc quy định quyền

truy cập đến các thành phần được thừa kế của lớp dẫn xuất.

1. Thừa kế public: các thành phần public và protected của lớp

cơ sở trở thành các thành phần public và protected tương ứng

của lớp dẫn xuất.

2. Thừa kế protected: các thành phần public và protected của

lớp cơ sở trở thành các thành phần protected của lớp xuất.

3. Thừa kế private: các thành phần public và protected của lớp

cơ sở trở thành các thành phần private của lớp dẫn xuất.

4. Trong bất kỳ trường hợp nào, các thành phần private của lớp

cơ sở, lớp dẫn xuất đều không có quyền truy cập tới mặc dầu

nó được thừa kế.

Trong ba dạng thừa kế đã nêu, thì thừa kế public là quan trọng nhất.

Nó được sử dụng để mở rộng một định nghĩa lớp, tức là để cài đặt một lớp

mới (lớp dẫn xuất) có đầy đủ các tính chất của lớp cơ sở và được bổ sung

thêm các tính chất mới. Thừa kế private được sử dụng để cài đặt một lớp mới

bằng các phương tiện của lớp cơ sở. Thừa kế protected ít được sử dụng.

***Sự tương thích kiểu.***

Khi lớp cơ sở là lớp cơ sở public, thì một lớp dẫn xuất có thể xem như

lớp con của lớp cơ sở theo nghĩa thuyết tập, tức là mỗi đối tượng của lớp dẫn

xuất là một đối tượng của lớp cơ sở, song điều ngược lại thì không đúng. Do

đó, khi ta khai báo một con trỏ trỏ tới đối tượng của lớp cơ sở, thì trong thời gian chạy con trỏ này có thể trỏ tới đối tượng của lớp dẫn xuất. Ví dụ, giả sử

chúng ta có các khai báo sau:

class Polygon { … };

class Rectangle : public Polygon { … };

Chú ý rằng, mỗi lớp là một kiểu dữ liệu. Do đó, ta có thể khai báo các

biến sau:

Polygon \*P ;

Polygon Pobj;

Rectangle \*R;

Rectangle Robj;

Khi đó:

P = new Polygon( ); // hợp lệ

P = new Rectangle ( ); // hợp lệ

R = new Polygon( ); // không hợp lệ

R = new Rectangle( ); // hợp lệ

P = R; // hợp lệ

R = P; // không hợp lệ

Chúng ta cũng có thể gán đối tượng lớp dẫn xuất cho đối tượng lớp cơ

sở, song ngược lại thì không được phép, chẳng hạn:

Pobj = Robj; // hợp lệ

Robj = Pobj; // không hợp lệ

1. Hàm ảo và tính đa hình

***Tính đa hình (polymorphism)*** để chỉ một hàm khai báo trong một

lớp cơ sở có thể có nhiều dạng khác nhau trong các lớp dẫn xuất, tức là hàm

có nội dung khác nhau trong các lớp dẫn xuất. Để hiểu được tính đa hình,

chúng ta hãy xét một ví dụ đơn giản. Giả sử chúng ta có một lớp cơ sở:

class Alpha

{

public :

……

void Hello( ) const

{ cout << “I am Alpha” << endl ; }

……

} ;

Lớp Alpha chứa hàm Hello( ), hàm này in ra thông báo “I am Alpha”.

Chúng ta xây dựng lớp Beta dẫn xuất từ lớp Alpha, lớp Beta cũng chứa hàm

Hello( ) nhưng với nội dung khác: nó in ra thông báo “I am Beta”.

class Beta : public Alpha

{

public :

…….

void Hello( ) const

{ cout << “I am Beta” << endl ; }

…….

} ;

Bây giờ, ta khai báo

Alpha Obj;

Khi đó, lời gọi hàm Obj.Hello( ) sẽ cho in ra “I am Alpha”, tức là bản

hàm Hello( ) trong lớp Alpha được thực hiện. Còn nếu ta khai báo

Beta Obj ;

thì lời gọi hàm obj.Hello( ) sẽ sử dụng bản hàm trong lớp Beta và sẽ cho in

ra “I am Beta”. Như vậy, bản hàm nào của hàm Hello( ) được sử dụng khi

thực hiện một lời gọi hàm được quyết định bởi kiểu đã khai báo của đối

tượng, tức là được quyết định trong thời gian dịch. Hoàn cảnh này được gọi là sự ràng buộc tĩnh (static binding). Song sự ràng buộc tĩnh không đáp

ứng được mong muốn của chúng ta trong tình huống sau đây:

Giả sử, Aptr là con trỏ trỏ tới đối tượng lớp Alpha:

Alpha \*Aptr ;

Kiểu của con trỏ Aptr khi khai báo là kiểu tĩnh. Trong thời gian chạy,

con trỏ Aptr có thể trỏ tới một đối tượng của lớp dẫn xuất. Kiểu của con trỏ

Aptr lúc đó là kiểu động. Chẳng hạn, khi gặp các dòng lệnh:

Aptr = new Beta ;

Aptr ÆHello( );

Aptr trỏ tới đối tượng của lớp Beta. Do đó, chúng ta mong muốn rằng,

lời gọi hàm Aptr ÆHello( ) sẽ cho in ra “I am Beta”. Song đáng tiếc là

không phải như vậy, kiểu tĩnh của con trỏ Aptr đã quyết định bản hàm

Hello( ) trong lớp Alpha được thực hiện và cho in ra “I am Alpha”.

Chúng ta có thể khắc phục được sự bất cập trên bằng cách khai báo

hàm Hello( ) trong lớp cơ sở Alpha là hàm ảo (virtual function). Để chỉ

một hàm là ảo, chúng ta chỉ cần viết từ khoá virtual trước khai báo hàm

trong định nghĩa lớp cơ sở. Chẳng hạn, lớp Alpha được khai báo lại như sau:

class Alpha

{

public :

………

virtual void Hello( ) const

{ cout << “I am Alpha << endl; }

………

};

Khi một hàm được khai báo là ảo trong lớp cơ sở, như hàm Hello( )

trong lớp Alpha, thì nó có thể được định nghĩa lại với các nội dung mới

trong các lớp dẫn xuất. Chẳng hạn, trong lớp Beta:

class Beta : public Alpha

{

public :

……..

virtual void Hello( ) const

{ cout << “I am Beta” << endl; }

……..

};

Chúng ta xây dựng một lớp dẫn xuất khác Gama từ lớp cơ sở Alpha.

Trong lớp Gama, hàm Hello( ) cũng được định nghĩa lại:

class Gama : public Alpha

{

public :

……..

virtual void Hello( ) const

{ cout << “I am Gama” << endl; }

……..

};

Như vậy, hàm ảo Hello( ) có ba dạng khác nhau. Chúng ta thử xem

bản hàm nào được sử dụng trong chương trình sau: #include <iostream>

class Alpha {

public:

virtual void Hello() {

std::cout << "Hello from Alpha" << std::endl;

}

};

class Beta : public Alpha {

public:

void Hello() override {

std::cout << "Hello from Beta" << std::endl;

}

};

class Gama : public Alpha {

public:

void Hello() override {

std::cout << "Hello from Gama" << std::endl;

}

};

int main() {

Alpha Aobj;

Beta Bobj;

Alpha \*Aptr;

Aptr = &Aobj;

Aptr->Hello(); // Bản hàm Hello() trong lớp Alpha được sử dụng.

Aptr = &Bobj;

Aptr->Hello(); // Bản hàm Hello() trong lớp Beta được sử dụng.

Aptr = new Gama();

Aptr->Hello(); // Bản hàm Hello() trong lớp Gama được sử dụng.

delete Aptr; // Don't forget to free the dynamically allocated memory.

return 0;

}

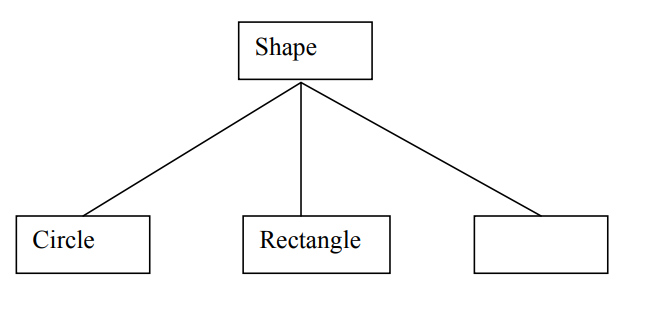
Tóm lại, một hàm được khai báo là ảo trong một lớp cơ sở, nó có thể được định nghĩa lại trong các lớp dẫn xuất và do đó nó có thể có nhiều dạng khác nhau trong các lớp dẫn xuất, chẳng hạn hàm Hello( ) có ba bản hàm khác nhau. Bản hàm nào được sử dụng trong lời gọi hàm (chẳng hạn, Aptr → Hello( ) ) được quyết định bởi kiểu của đối tượng mà con trỏ lớp cơ sở trỏ tới, tức là được xác định trong thời gian chạy. Điều này được gọi là sự ràng buộc động (dynamic binding). Như vậy, một hàm được khai báo là ảo trong lớp cơ sở là hàm có tính đa hình, tức là hàm có nhiều dạng khác nhau. Dạng hàm thích hợp được lựa chọn để thực hiện phụ thuộc vào kiểu động của đối tượng kích hoạt hàm. Khi thiết kế một lớp làm cơ sở cho các lớp dẫn xuất khác, chúng ta cần chú ý đến các luật tổng quát sau:

• Các hàm cần định nghĩa lại trong các lớp dẫn xuất cần phải là ảo.

• Hàm kiến tạo không thể là ảo, song hàm huỷ cần phải là ảo.

1. Lớp cơ sở trừu tượng

Giả sử chúng ta cần thiết kế các lớp sau: lớp các hình tròn (Circle), lớp các hình chữ nhật (Rectangle), và nhiều lớp các hình phẳng có dạng đặc biệt khác. Trong các lớp đó chúng ta cần phải đưa vào các hàm thành phần thực hiện các hành động có đặc điểm chung cho tất cả các loại hình, chẳng hạn tính chu vi, tính diện tích, vẽ hình, … Trong tình huống này, chúng ta cần thiết kế một lớp, lớp các hình (Shape), làm cơ sở để dẫn xuất ra các lớp Circle, Rectangle, …, như được minh hoạ trong hình sau:



Lớp Shape sẽ chứa các khai báo các hàm thực hiện các xử lý có đặc điểm chung cho các lớp dẫn xuất, chẳng hạn các hàm tính chu vi Perimeter( ), hàm tính diện tích Area( ), … Song chúng ta không thể cài đặt được các hàm này trong lớp Shape ( bởi vì chúng ta không thể tính được chu vi và diện tích của một hình trừu tượng), các hàm này sẽ được cài đặt trong các lớp dẫn xuất từ lớp Shape, chẳng hạn được cài đặt trong các lớp Circle, Rectangle, …Và do đó, trong lớp Shape chúng được khai báo là hàm ảo thuần tuý (pure virtual function) hay còn gọi là hàm trừu tượng (abstract function). Một hàm thành phần trong một lớp được gọi là ảo thuần tuý (hay trừu tượng) nếu nó chỉ được khai báo, nhưng không được định nghĩa trong lớp đó. Một hàm ảo thuần tuý được khai báo bằng cách khai báo nó là ảo và đặt = 0 ở sau mẫu hàm. Ví dụ, để khai báo các hàm Permeter( ) và Area( ) là ảo thuần tuý, ta viết:

virtual double Perimeter( ) const = 0;

virtual double Area( ) const = 0;

Một lớp chứa ít nhất một hàm ảo thuần tuý được gọi là lớp cơ sở trừu tượng (abstract base class). Lớp cơ sở trừu tượng không có đối tượng nào cả, nó chỉ được sử dụng làm cơ sở để xây dựng các lớp khác. Lớp cơ sở trừu tượng chứa các hàm ảo thuần tuý biểu diễn các xử lý có đặc điểm chung cho các lớp đối tượng khác nhau. Các hàm đó sẽ được định nghĩa trong các lớp dẫn xuất. Và như vậy, các hàm ảo thuần tuý trong một lớp cơ sở trừu tượng là các hàm có tính đa hình. Sử dụng tính đa hình, người lập trình có thể viết các phần mềm dễ dàng hơn khi mở rộng, có tính khái quát cao, dễ đọc, dễ hiểu,… Mô hình thiết kế các lớp dẫn xuất từ lớp cơ sở trừu tượng là mô hình thiết kế mà chúng ta cần sử dụng trong các hoàn cảnh tương tự như khi thiết kế các lớp đối tượng hình học phẳng. Để làm ví dụ minh hoạ cho khái niệm lớp cơ sở trừu tượng, chúng ta xây dựng lớp Shape. Một lớp cơ sở trừu tượng có thể chứa các thành phần dữ liệu là chung cho tất cả các lớp dẫn xuất. Chẳng hạn, lớp Shape chứa một biến thành phần private có tên là name để lưu tên các loại hình. Chúng ta đưa vào lớp Shape một hàm kiến tạo, nó không được gọi trực tiếp để khởi tạo ra đối tượng của lớp Shape (vì Shape là lớp trừu tượng, nên không có đối tượng nào cả), song nó sẽ được gọi để khởi tạo các đối tượng của các lớp dẫn xuất. Lớp Shape chứa hai hàm ảo thuần tuý: hàm Perimeter( ) và hàm Area( ). Ngoài các hàm ảo thuần tuý, lớp cơ sở trừu tượng còn có thể chứa các hàm ảo và các hàm thành phần khác. Chẳng hạn, lớp Shape chứa hàm operator <= để so sánh các hình theo diện tích và hàm ảo print để in ra một số thông tin về các hình. Định nghĩa lớp trừu tượng Shape được cho trong hình 3.2.

class Shape

{

public:

Shape (const string & s = “ ”)

: name (s) { }

~ Shape( ) { }

virtual double Perimeter( ) const = 0;

virtual double Area( ) const = 0;

bool operator <=(const Shape & sh) const

{ return Area( ) <= sh. Area( ); }

virtual void Print (ostream & out = cout) const

{ out << name << “of area” << Area( ) ; }

private:

string name;

};

Chú ý rằng, hàm Print đã được cài đặt trong lớp Shape, nhưng nó được khai báo là ảo, mục đích là để bạn có thể cài đặt lại hàm này trong các lớp dẫn xuất để in ra các thông tin khác nếu bạn muốn. Sử dụng hàm Print, bạn có thể định nghĩa lại toán tử << để nó in ra các thông tin về các hình như sau: ostream & operator << (ostream & out, const Shape & sh) { sh.Print (out); return out; } Bây giờ dựa trên lớp cơ sở trừu tượng Shape, chúng ta sẽ xây dựng một loạt lớp dẫn xuất: lớp các hình có dạng đặc biệt, chẳng hạn các lớp Circle, Rectangle,… Lớp Rectangle được thiết kế như sau: ngoài thành phần dữ liệu name được thừa kế từ lớp Shape, nó chứa hai thành phần dữ liệu khác là length (chỉ chiều dài) và width (chỉ chiều rộng của hình chữ nhật). Lớp chứa hàm kiến tạo để khởi tạo nên hình chữ nhật có chiều dài là l, chiều rộng là w.

Chú ý rằng, hàm Print đã được cài đặt trong lớp Shape, nhưng nó được khai báo là ảo, mục đích là để bạn có thể cài đặt lại hàm này trong các lớp dẫn xuất để in ra các thông tin khác nếu bạn muốn. Sử dụng hàm Print, bạn có thể định nghĩa lại toán tử << để nó in ra các thông tin về các hình như sau: ostream & operator << (ostream & out, const Shape & sh) { sh.Print (out); return out; } Bây giờ dựa trên lớp cơ sở trừu tượng Shape, chúng ta sẽ xây dựng một loạt lớp dẫn xuất: lớp các hình có dạng đặc biệt, chẳng hạn các lớp Circle, Rectangle,… Lớp Rectangle được thiết kế như sau: ngoài thành phần dữ liệu name được thừa kế từ lớp Shape, nó chứa hai thành phần dữ liệu khác là length (chỉ chiều dài) và width (chỉ chiều rộng của hình chữ nhật). Lớp chứa hàm kiến tạo để khởi tạo nên hình chữ nhật có chiều dài là l, chiều rộng là w.

Chú ý rằng, trong hàm kiến tạo trên, chúng ta đã gọi hàm kiến tạo của

lớp Shape để đặt tên cho hình. Trong lớp Rectangle, chúng ta cần cung cấp

định nghĩa cho các hàm Perimeter( ) và Area( ) được khai báo là trừu tượng

trong lớp Shape. Chúng ta cũng định nghĩa lại hàm ảo Print để nó cho biết

thêm một số thông tin khác về hình chữ nhật. Hàm Print được cài đặt như

sau:

void Rectangle :: Print (ostream & out = cout) const

{

Shape :: Print(out);

out << “and of length” << length

<< “and of width” << width;

}

Lớp Circle được thiết kế một cách tương tự. Định nghĩa lớp Rectangle

và lớp Circle được cho trong hình 3.3.

--------------------------------------------------------------------------

class Rectangle : public Shape

{

public:

Rectangle (double l = 0.0, double w = 0.0)

: Shape (“rectangle”), length(l), width(w) { }

double GetLength( ) const

{ return length; }

double GetWidth( ) const

{ return width; }

double Perimeter( ) const

{return 2\*(length + width); }

double Area( ) const

{ return length \* width; }

void Print(ostream & out = cout)

{

// như đã trình bày ở trên.

}

private :

double length;

double width;

};

Tóm lại, trong một lớp các hàm ảo thuần tuý (hay các hàm trừu tượng)

là các hàm chỉ được khai báo, nhưng không được định nghĩa. Một lớp chứa

ít nhất một hàm ảo thuần tuý được gọi là lớp trừu tượng, nó không có đối

tượng nào cả, nhưng được sử dụng làm cơ sở để xây dựng các lớp dẫn xuất.

Nó cung cấp cho người sử dụng một dao diện chung cho tất cả các lớp dẫn

xuất. Trong một lớp dẫn xuất từ lớp cơ sở trừu tượng, chúng ta có thể cung

cấp định nghĩa cho các hàm trừu tượng của lớp cơ sở. Một hàm ảo thuần tuý

cũng giống như hàm ảo thông thường ở đặc điểm là hàm có tính đa hình, bản

hàm nào ( trong số các bản hàm được cài đặt ở các lớp dẫn xuất) được thực

hiện phụ thuộc vào đối tượng kích hoạt hàm thuộc lớp dẫn xuất nào, đối

tượng đó được xác định trong thời gian chạy (sự ràng buộc động).

Mặc dù lớp cơ sở trừu tượng không có đối tượng nào cả, song chúng

ta có thể khai báo một biến tham chiếu đến lớp cơ sở trừu tượng, chẳng hạn

trong khai báo hàm sau:

ostream & operator << (ostream & out, const Shape & sh);

Chúng ta cũng có thể khai báo các con trỏ trỏ tới lớp cơ sở trừu tượng,

chẳng hạn

Shape \* aptr, \* bptr;

aptr = new Rectangle (2.3, 5.4); // hợp lệ

bptr = new Shape( “circle” ) ; // không hợp lệ

Chương 4: DANH SÁCH

Danh sách là cấu trúc dữ liệu tuyến tính, danh sách được tạo nên từ các phần tử dữ liệu được sắp xếp theo một thứ tự xác định. Danh sách là một trong các cấu trúc dữ liệu cơ bản được sử dụng thường xuyên nhất trong các thuật toán. Danh sách còn được sử dụng để cài đặt nhiều KDLTT khác. Trong chương này, chúng ta sẽ xác định KDLTT danh sách và nghiên cứu phương pháp cài đặt KDLTT danh sách bởi mảng. Sau đó, chúng ta sẽ sử dụng danh sách để cài đặt KDLTT tập động.

Bài 1: Kiểu dữ liệu trừu tượng danh sách

Danh sách là một khái niệm được sử dụng thường xuyên trong thực

tiễn. Chẳng hạn, chúng ta thường nói đến danh sách sinh viên của một lớp, danh sách các số điện thoại, danh sách thí sinh trúng tuyển, …

Danh sách được định nghĩa là một dãy hữu hạn các phần tử:

L = (a1, a2, … , an)

trong đó ai (i = 1, …, n) là phần tử thứ i của danh sách. Cần lưu ý rằng, số

phần tử của danh sách, được gọi là độ dài của danh sách, có thể thay đổi

theo thời gian. Và một phần tử dữ liệu có thể xuất hiện nhiều lần trong danh

sách ở các vị trí khác nhau. Chẳng hạn, trong danh sách các số nguyên sau:

L = (3, 5, 5, 0, 7, 5)

số nguyên 5 xuất hiện 3 lần ở các vị trí 2, 3, và 6. Một đặc điểm quan trọng

khác của danh sách là các phần tử của nó có thứ tự tuyến tính, thứ tự này

được xác định bởi vị trí của các phần tử trong danh sách. Khi độ dài của

danh sách bằng không (n = 0), ta nói danh sách rỗng. Nếu danh sách không

rỗng (n ≥ 1), thì phần tử đầu tiên a1 được gọi là đầu của danh sách, còn phần

tử cuối cùng an được gọi là đuôi của danh sách.

Không có hạn chế nào trên kiểu dữ liệu của các phần tử trong danh sách. Khi mà tất cả các phần tử của danh sách cùng một kiểu, ta nói danh sách là danh sách thuần nhất. Trong trường hợp tổng quát, một danh sách có thể chứa các phần tử có kiểu khác nhau, đặc biệt một phần tử của danh sách có thể lại là một danh sách. Chẳng hạn

L = (An, (20, 7, 1985), 8321067)

Trong danh sách này, phần tử đầu tiên là một xâu ký tự, phần tử thứ hai là danh sách các số nguyên, phần tử thứ ba là số nguyên. Danh sách này có thể sử dụng để biểu diễn, chẳng hạn, một sinh viên có tên là An, sinh ngày 20/7/1985, có số điện thoại 8321067. Danh sách (tổng quát) là cấu trúc dữ liệu cơ bản trong các ngôn ngữ lập trình chuyên dụng cho các xử lý dữ liệu phi số, chẳng hạn Prolog, Lisp. Trong sách này, chúng ta chỉ quan tâm tới các danh sách thuần nhất, tức là khi nói đến danh sách thì cần được hiểu đó là danh sách mà tất cả các phần tử của nó cùng một kiểu. Khi sử dụng danh sách trong thiết kế thuật toán, chúng ta cần dùng đến một tập các phép toán rất đa dạng trên danh sách. Sau đây là một số phép toán chính. Trong các phép toán này, L ký hiệu một danh sách bất kỳ có độ dài n ≥ 0, x ký hiệu một phần tử bất kỳ cùng kiểu với các phần tử của L và i là số nguyên dương chỉ vị trí

1. Empty(L). Hàm trả về true nếu L rỗng và false nếu L không rỗng.

2. Length(L). Hàm trả về độ dài của danh sách L.

3. Insert(L, x, i). Thêm phần tử x vào danh sách L tại vị trí i. Nếu thành công thì các phần tử ai, ai+1, … , an trở thành các phần tử ai+1, ai+2, … , an+1 tương ứng, và độ dài danh sách là n+1. Điều kiện để phép toán xen có thể thực hiện được là i phải là vị trí hợp lý, tức 1≤ i ≤ n+1, và không gian nhớ dành để lưu danh sách L còn chỗ.

4. Append(L, x). Thêm x vào đuôi danh sách L, độ dài danh sách tăng lên 1.

5. Delete(L, i). Loại phần tử ở vị trí thứ i trong danh sách L. Nếu thành công, các phần tử ai+1, ai+2, … , an trở thành các phần tử ai, ai+1, … , an-1 tương ứng, và độ dài danh sách là n-1. Phép toán loại chỉ được thực hiện thành công khi mà danh sách không rỗng và i là vị trí thực sự trong danh sách, tức là 1 ≤ i ≤ n. 6. Element(L, i). Tìm biết phần tử ở vị trí thứ i của L. Nếu thành công hàm trả về phần tử ở vị trí i. Điều kiện để phép toán tìm thực hiện thành công cũng giống như đối với phép toán loại

Chúng ta quan tâm đến các phép toán trên, vì chúng là các phép toán được sử dụng thường xuyên khi xử lý danh sách. Hơn nữa, đó còn là các phép toán sẽ được sử dụng để cài đặt nhiều KDLTT khác như tập động, từ điển, ngăn xếp, hàng đợi, hàng ưu tiên. Nhưng trong các chương trình có sử dụng danh sách, nhiều trường hợp chúng ta cần thực hiện các phép toán đa dạng khác trên danh sách, đặc biệt chúng ta thường phải đi qua danh sách (duyệt danh sách) để xem xét lần lượt từng phần tử của danh sách từ phần tử đầu tiên đến phần tử cuối cùng và tiến hành các xử lý cần thiết với mỗi phần tử của danh sách. Để cho quá trình duyệt danh sách được thực hiện thuận tiện, hiệu quả, chúng ta xác định các phép toán sau đây. Các phép toán này tạo thành bộ công cụ lặp (iteration). Tại mỗi thời điểm, phần tử đang được xem xét của danh sách được gọi là phần tử hiện thời và vị trí của nó trong danh sách được gọi là vị trí hiện thời.

7. Start(L). Đặt vị trí hiện thời là vị trí đầu tiên trong danh sách L.

8. Valid(L). Trả về true, nếu vị trí hiện thời có chứa phần tử của danh sách L, nó trả về false nếu không.

9. Advance(L). Chuyển vị trí hiện thời tới vị trí tiếp theo trong danh sách L.

10.Current(L). Trả về phần tử tại vị trí hiện thời trong L.

11.Add(L, x). Thêm phần tử x vào trước phần tử hiện thời, phần tử

hiện thời vẫn còn là phần tử hiện thời.

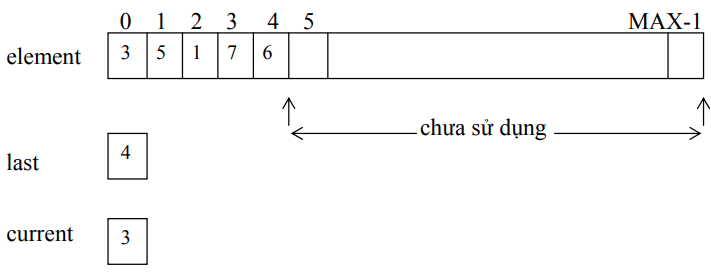
12.Remove(L). Loại phần tử hiện thời khỏi L. Phần tử đi sau phần bị loại trở thành phần tử hiện thời.

Chúng ta đã đặc tả KDLTT danh sách. Bây giờ chuyển sang giai đoạn cài đặt danh sách.

Bài 2: Cài đặt danh sách bởi mảng

Chúng ta sẽ cài đặt KDLTT danh sách bởi các lớp C + +. Có nhiều cách thiết kế lớp cho KDLTT danh sách, điều đó trước hết là do danh sách có thể biểu diễn bởi các cấu trúc dữ liệu khác nhau. Các thiết kế lớp khác nhau cho danh sách sẽ được trình bày trong chương này và chương sau. Trong mục này chúng ta sẽ trình bày cách cài đặt danh sách bởi mảng (mảng tĩnh). Đây là phương pháp cài đặt đơn giản và tự nhiên nhất.

Chúng ta sẽ sử dụng một mảng element có cỡ là MAX để lưu các phần tử của danh sách. Các phần tử của danh sách sẽ được lần lượt lưu trong các thành phần của mảng element[0], element[1], … , element[n-1], nếu danh sách có n phần tử. Tức là, danh sách được lưu trong đoạn đầu element[0…n-1] của mảng, đoạn sau của mảng, element[n.. MAX-1], là không gian chưa được sử dụng. Cần lưu ý rằng, phần tử thứ i của danh sách (i = 1, 2, …) được lưu trong thành phần element[i-1] của mảng. Cần có một biến last ghi lại chỉ số sau cùng mà tại đó mảng có chứa phần tử của danh sách. Vị trí hiện thời được xác định bởi biến current, nó là chỉ số mà element[current] chứa phần tử hiện thời của danh sách. Chẳng hạn, giả sử chúng ta có danh sách các số nguyên L = (3, 5, 1, 7, 6) và phần tử hiện thời đứng ở vị trí thứ 4 trong danh sách, khi đó danh sách L được biểu diễn bởi cấu trúc dữ liệu được mô tả trong hình 4.1.



Chúng ta muốn thiết kế lớp danh sách sao cho người lập trình có thể sử dụng nó để biểu diễn danh sách với các phần tử có kiểu tuỳ ý. Do đó, lớp danh sách được thiết kế là lớp khuôn phụ thuộc tham biến kiểu Item như sau:

***template*** <class Item>

***class*** List

{

***public*** :

static const int MAX = 50; // khai báo cỡ của mảng.

// các hàm thành phần.

***private*** :

Item element[MAX] ;

int last ;

int current ;

} ;

Bây giờ cần phải thiết kế các hàm thành phần của lớp List. Ngoài các hàm thành phần tương ứng với các phép toán trên danh sách, chúng ta đưa vào một hàm kiến tạo mặc định, hàm này khởi tạo nên danh sách rỗng. Lớp List chứa ba biến thành phần đã khai báo như trên, nên không cần thiết phải đưa vào hàm kiến tạo copy, hàm huỷ và hàm toán tử gán, vì chỉ cần sử dụng các hàm kiến tạo copy tự động, … do chương trình dịch cung cấp là đủ. Định nghĩa đầy đủ của lớp List được cho trong hình 4.2.

// File đầu list.h

# ifndef LIST\_H

# define LIST\_H

# include <assert.h>

template <class, Item>

class List

{

public :

static const int MAX = 50 ;

List ( )

// Khởi tạo danh sách rỗng.

{ last = -1; current = -1; }

bool Empty( ) const

// Kiểm tra danh sách có rỗng không.

// Postcondition: Hàm trả về true nếu danh sách rỗng và false

// nếu không

{ return last < 0; }

int Length( ) const

// Xác định độ dài danh sách.

// Postcondition: Trả về số phần tử trong danh sách.

{return last+1; }

void Insert(const Item&x, int i);

// Xen phần tử x vào vị trí thứ i trong danh sách.

// Precondition: Length( ) < MAX và 1 ≤ i ≤ Length( )

// Postcondition: các phần tử của danh sách kể từ vị trí thứ i

// được đẩy ra sau một vị trí, x nằm ở vị trí i.

void Append(const Item&x);

// Thêm phần tử x vào đuôi danh sách.

// Precondition : Length( ) < MAX

// Postcondition : x là đuôi của danh sách.

void Delete(int i)

// Loại khỏi danh sách phần tử ở vị trí i.

// Precondition: 1 ≤ i ≤ Length( )

// Postcondition: phần tử ở vị trí i bị loại khỏi danh sách,

// các phần tử đi sau được đẩy lên trước một vị trí.

Item & Element(int i) const

// Tìm phần tử ở vị trí thứ i.

// Precondition: 1 ≤ i ≤ Length( )

// Postcondition: Trả về phần tử ở vị trí i.

{ assert (1<= i && i <= Length( ) ); return element[i - 1]; }

// Các hàm bộ công cụ lặp:

void start( )

// Postcondition: vị trí hiện thời là vị trí đầu tiên của danh sách.

{current = 0; }

bool Valid( ) const

// Postcondition: Trả về true nếu tại vị trí hiện thời có phần tử

// trong danh sách, và trả về false nếu không.

{return current >= 0 && current <= last; }

void Advance( )

// Precondition: Hàm Valid( ) trả về true.

// Postcondition: Vị trí hiện thời là vị trí tiếp theo trong danh

// sách.

{ assert (Valid( )); assert (Valid( )); current + +;}

Item & Current( ) const

// Precondition: Hàm Valid( ) trả về true.

// Postcondition: Trả về phần tử hiện thời của danh sách.

{ assert (Valid( )); return element[current]; }

void Add(const Item& x);

// Precondition: Length( ) < MAX và hàm Valid( ) trả về true

// Postcondition: Phần tử x được xen vào trước phần tử

// hiện thời, phần tử hiện thời vẫn còn là phần tử hịên thời.

void Remove( );

// Precondition: hàm Valid( ) trả về true.

// Postcondition: Phần tử hiện thời bị loại khỏi danh sách,

// phần tử đi sau nó trở thành phần tử hiện thời.

private :

Item element[MAX];

int last;

int current;

};

# include “list.template”

# endif

#ifndef LIST\_H

#define LIST\_H

#include <assert.h>

template <class Item>

class List

{

public:

static const int MAX = 50;

List()

{

last = -1;

current = -1;

}

bool Empty() const

{

return last < 0;

}

int Length() const

{

return last + 1;

}

void Insert(const Item &x, int i)

{

assert(1 <= i && i <= Length());

if (Length() < MAX)

{

for (int j = last; j >= i - 1; j--)

{

element[j + 1] = element[j];

}

element[i - 1] = x;

last++;

}

}

void Append(const Item &x)

{

if (Length() < MAX)

{

element[++last] = x;

}

}

void Delete(int i)

{

assert(1 <= i && i <= Length());

for (int j = i - 1; j < last; j++)

{

element[j] = element[j + 1];

}

last--;

}

Item &Element(int i) const

{

assert(1 <= i && i <= Length());

return element[i - 1];

}

void Start()

{

current = 0;

}

bool Valid() const

{

return current >= 0 && current <= last;

}

void Advance()

{

assert(Valid());

current++;

}

Item &Current() const

{

assert(Valid());

return element[current];

}

void Add(const Item &x)

{

assert(Length() < MAX && Valid());

for (int j = last; j >= current; j--)

{

element[j + 1] = element[j];

}

element[current] = x;

last++;

}

void Remove()

{

assert(Valid());

for (int j = current; j < last; j++)

{

element[j] = element[j + 1];

}

last--;

}

private:

Item element[MAX];

int last;

int current;

};

#endif

Bước tiếp theo chúng ta cần cài đặt các hàm thành phần của lớp List. Trước hết, nói về hàm kiến tạo mặc định, hàm này cần tạo ra một danh sách rỗng, do vậy chỉ cần đặt giá trị cho biến last là –1, giá trị của biến current cũng là –1. Hàm này được cài đặt là hàm inline. Với cách khởi tạo này, mỗi khi cần thêm phần tử mới vào đuôi danh sách (kể cả khi danh sách rỗng), ta chỉ cần tăng chỉ số last lên 1 và đặt phần tử cần thêm vào thành phần mảng element[last].

***Hàm Append*** được định nghĩa như sau:

***template*** <class Item>

void List<Item> :: Append (const Item& x)

{ assert (Length( ) < MAX);

last + + ;

element[last] = x;

}

***Hàm Insert***. Để xen phần tử x vào vị trí thứ i của danh sách, tức là x

cần đặt vào thành phần element[i - 1] trong mảng, chúng ta cần dịch chuyển

đoạn element[i – 1 … last] ra sau một vị trí để có chỗ cho phần tử x. Hàm

Insert có nội dung như sau:

***template*** <class Item>

void List<Item> :: Insert(const Item& x, int i)

{

assert (Length( ) < MAX && 1 <= i && i <= Length( ));

last + +;

***for*** (int k = last; k >= i; k - -)

element[k] = element[k - 1];

element[i - 1] = x;

}

***Hàm Add***. Hàm này cũng tương tự như hàm Insert, nó có nhiệm vụ xen phần tử mới x vào vị trí của phần tử hiện thời. Vì phần tử hiện thời được đẩy ra sau một vị trí, nên chỉ số hiện thời phải được tăng lên 1.

template <class Item>

void List<Item> :: Add(const Item& x)

{

assert (Length( ) < MAX && Valid( ));

last + +;

for(int k = last; k > current; k - -)

element[k] = element[k - 1];

element[current] = x;

current + +;

}

***Hàm Delete.*** Muốn loại khỏi danh sách phần tử ở vị trí thứ i, chúng ta cần phải đẩy đoạn cuối của danh sách kể từ vị trí i + 1 lên trước 1 vị trí, và giảm chỉ số last đi 1.

***template*** <class Item>

void List<Item> :: Delete(int i)

{

assert (1 <= i && i <= Length( ));

***for*** (int k = i – 1; k < last; k + +)

element[k] = element[k + 1];

last - - ;

}

***Hàm Remove.*** Hàm này được cài đặt tương tự như hàm Delete.

***template*** <class Item>

void List<Item> :: Remove( )

{

assert(Valid( ));

***for*** (int k = current; k < last; k + +)

element[k] = element[k + 1];

last - - ;

}

Tất cả các hàm còn lại đều rất đơn giản và được cài đặt là hàm inline. Bây giờ chúng ta đánh giá hiệu quả của các phép toán của KDLTT danh sách, khi mà danh sách được cài đặt bởi mảng. Ưu điểm của cách cài đặt này là nó cho phép ta truy cập trực tiếp tới từng phần tử của danh sách, vì phần tử thứ i của danh sách được lưu trong thành phần mảng element[i - 1]. Nhờ đó mà thời gian của phép toán Element(i) là O(1). Giả sử danh sách có độ dài n, để xen phần tử mới vào vị trí thứ i trong danh sách, chúng ta cần đẩy các phần tử lưu ở các thành phần mảng từ element[i -1] đến element[n - 1] ra sau một vị trí. Trong trường hợp xấu nhất (khi xen vào vị trí đầu tiên trong danh sách), cần n lần đẩy, vì vậy thời gian của phép toán Insert là O(n). Phân tích một cách tượng tự, chúng ta thấy rằng thời gian chạy của các phép toán Delete, Add và Remove cũng là O(n), trong đó n là độ dài của danh sách. Thời gian của tất cả các phép toán còn lại đều là O(1). Như chúng ta đã nói, các phép toán trong bộ công cụ lặp cho phép chúng ta có thể tiến hành dễ dàng các xử lý danh sách cần đến duyệt danh sách. Chẳng hạn, giả sử L là danh sách các số nguyên, để loại khỏi danh sách L tất cả các số nguyên bằng 3, chúng ta có thể viết:

L. Start( );

***while*** (L.Valid( ))

***if*** (L.Current( ) = = 3)

L.Remove( ) ;

***else*** L.Advance( ) ;

Chúng ta cũng có thể in ra tất cả các phần tử của danh sách bằng cách sử dụng vòng lặp for như sau:

***for*** (L.Start( ); L.Valid( ); L.Advance( )) cout << L.Current( ) << endl;

Nhận xét. Cài đặt danh sách bởi mảng có ưu điểm cơ bản là nó cho phép truy cập trực tiếp tới từng phần tử của danh sách. Nhờ đó chúng ta có thể cài đặt rất thuận tiện phép toán tìm kiếm trên danh sách, đặc biệt khi danh sách là danh sách được sắp (các phần tử của nó được sắp xếp theo thứ tự không tăng hoặc không giảm), nếu lưu danh sách được sắp trong mảng, chúng ta có thể cài đặt dễ dàng phương pháp tìm kiếm nhị phân. Phương pháp tìm kiếm nhị phân là một kỹ thuật tìm kiếm rất hiệu quả và sẽ được nghiên cứu trong mục chương 4 bài 4.

Chúng ta đã cài đặt danh sách bởi mảng tĩnh, mảng có cỡ MAX cố định. Khi danh sách phát triển, tới lúc nào đó mảng sẽ đầy, và lúc đó các phép toán Insert, Append, Add sẽ không thể thực hiện được. Đó là nhược điểm chính của cách cài đặt danh sách bởi mảng tĩnh. Bây giờ nói về cách thiết kế lớp List. Trong lớp List này, tất cả các hàm trong bộ công cụ lặp được đưa vào các hàm thành phần của lớp và biến lưu vị trí hiện thời cũng là một biến thành phần của lớp. Thiết kế này có vấn đề: chỉ có một biến hiện thời, do đó chúng ta không thể tiến hành đồng thời hai hoặc nhiều phép lặp khác nhau trên cùng một danh sách. Mục sau sẽ trình bày một phương pháp thiết kế lớp List khác, nó khắc phục được các nhược điểm đã nêu trên.

Bài 3: Cài đặt danh sách bởi mảng động

Trong mục này chúng ta sẽ thiết kế một lớp khác cài đặt KDLTT danh

sách, lớp này được đặt tên là Dlist (Dynamic List). Lớp Dlist khác với lớp

List đã được trình bày trong mục 4.2 ở hai điểm. Thứ nhất, danh sách được

lưu trong mảng được cấp phát động. Thứ hai, các hàm trong bộ công cụ lặp

được tách ra và đưa vào một lớp riêng: lớp công cụ lặp trên danh sách,

chúng ta sẽ gọi lớp này là DlistIterator. Với cách thiết kế này, chúng ta sẽ

khắc phục được các nhược điểm của lớp List đã được nêu ra trong nhận xét

ở cuối mục 4.2.

Lớp Dlist chưa ba thành phần dữ liệu: Biến con trỏ element trỏ tới

mảng được cấp phát động để lưu các phần tử của danh sách. Biến size lưu cỡ

của mảng, và biến last lưu chỉ số cuối cùng mà tại đó mảng chứa phần tử của

danh sách.

Lớp Dlist chứa tất cả các hàm thành phần thực hiện các phép toán trên

danh sách giống như trong lớp List, trừ ra các hàm công cụ lặp (các hàm này

sẽ được đưa vào lớp DlistIterator). Chúng ta đưa vào lớp Dlist hai hàm kiến

tạo. Hàm kiến tạo một tham biến nguyên là cỡ của mảng được cấp phát động

và hàm kiến tạo copy. Chúng ta cần phải đưa vào lớp Dlist hàm huỷ để thu hồi bộ nhớ đã cấp phát cho mảng element, khi mà đối tượng không còn cần thiết nữa. Lớp Dlist cũng cần có hàm toán tử gán. Định nghĩa lớp Dlist được cho trong hình 4.3.

template <class Item>

class DlistIterator ;

// Khai báo trước lớp DlistIterator.

template <class Item>

class Dlist

{

public:

friend class DlistIterator<Item>;

Dlist( )

{element = NULL; size = 0; last = -1; }

DList (int m);

Dlist (const Dlist & L);

~ Dlist( )

{delete [ ] element; }

Dlist & operator = (const Dlist & L);

inline bool Empty( ) const;

inline int Length( ) const;

void Insert(const Item & x, int i);

void Append(const Item & x);

void Delete(int i);

inline Item & Element(int i);

private:

Item\* element;

Int size;

Int last;

};

Chú ý rằng, trước khi định nghĩa lớp Dlist, chúng ta đã khai báo trước lớp DlistIterator. Khai báo này là cần thiết, bởi vì trong định nghĩa lớp Dlist, chúng ta xác định lớp DlistIterator là bạn của nó. Sau đây chúng ta sẽ xem xét sự cài đặt các hàm thành phần của lớp Dlist. Các hàm Empty, Length, Delete và Retrieve là hàm hoàn toàn giống như trong lớp List.

***Các hàm kiến tạo.*** Trước hết nói đến hàm kiến tạo có một tham biến nguyên m. Nhiệm vụ chính của nó là cấp phát một mảng động có cỡ là m để lưu các phần tử của danh sách.

template <class Item>

Dlist<Item> :: Dlist(int m)

// Precondition. m là số nguyên dương.

// Postcondition. Một danh sách rỗng được khởi tạo, với khả năng tối

// đa chứa được m phần tử.

{

element = new Item[m];

assert (element != NULL);

size = m;

last = -1;

}

Hàm kiến tạo copy có trách nhiệm tạo ra một danh sách mới là bản sao của danh sách đã có L. Trước hết ta cần cấp phát một mảng động có cỡ là cỡ của mảng trong danh sách L, sau đó sao chép từng thành phần của mảng trong danh sách L sang mảng mới.

***template*** <class Item>

Dlist<Item> :: Dlist(const Dlist<Item> & L)

{

element = new Item[L.size];

size = L.size;

last = L.last;

***for*** (int i = 0; i <= last; i + +)

element[i] = L.element[i];

}

***Toán tử gán***. Toán tử gán được cài đặt gần giống như hàm kiến tạo copy. Chỉ có điều cần lưu ý là, khi cỡ của mảng trong hai danh sách khác nhau, chúng ta mới cần cấp phát một mảng động có cỡ là cỡ của mảng trong danh sách nguồn, rồi thực hiện sao chép giống như trong hàm kiến tạo copy.

Dlist<Item> & Dlist<Item> :: operator = (const Dlist<Item> & L)

{

***if*** (size != L.size)

{

delete [ ] element;

element = new Item[L.size];

size = L.size;

}

last = L.last;

***for***(int i = 0; i <= last; i + +)

element[i] = L.element[i];

***return*** \*this;

}

***Hàm Insert.*** Ưu điểm của cách cài đặt danh sách bởi mảng động là các hành động thêm phần tử mới vào danh sách luôn luôn được thực hiện. Bởi vì khi mà mảng đầy, chúng ta sẽ cấp phát một mảng động mới có cỡ gấp đôi cỡ của mảng cũ. Sau đó sao chép đoạn đầu [0… i-1] của mảng cũ sang mảng mới, đưa phần tử cần xen vào mảng mới, rồi lại chép đoạn còn lại của mảng cũ sang mảng mới. Hàm Insert được định nghĩa như sau:

template <class Item>

void Dlist<Item> :: Insert(const Item&x, int i)

{

assert(i >= 0 && i <= last);

if (Length( ) < size)

{

last + +;

for (int k = last; k >= i; k - -)

element[k] = element[k -1];

element[i-1] = x;

}

else // mảng element đầy

{

Item\* newArray = new Item[2 \* size + 1]

assert (newArray != NULL);

for (int k = 0; k < i –1; k + + )

newArray[k] = element[k];

newArray[i - 1] = x;

for (int k = i; k <= last + 1; k + + )

newArray[k] = element[k - 1];

delete [ ] element;

element = newArray;

last + +;

size = 2 \* size + 1;

}

}

Hàm Append được cài đặt tương tự như hàm Insert, nhưng đơn giản hơn. Chúng tôi để lại cho độc giả, xem như bài tập. Hàm Delete được cài đặt như trong lớp List (xem mục 4.2 ). Bây giờ chúng ta thiết kế lớp công cụ lặp trên danh sách được cài đặt bởi mảng: lớp DlistIterator. Khai báo lớp công cụ lặp được sử dụng với lớp Dlist được cho trong hình 4.4. Lớp này chứa các phép toán của KDLTT danh sách liên quan tới phép lặp qua các phần tử của danh sách. Lớp DlistIterator chứa hai thành phần dữ liệu: biến current lưu số nguyên biểu diễn vị trí hiện thời, nó là chỉ số mà tại đó mảng lưu phần tử hiện thời; và biến LPtr lưu một con trỏ hằng trỏ tới đối tượng của lớp Dlist mà các phép toán công cụ lặp sẽ thực hiện trên nó.

# include <assert.h>

template <class Item>

class DlistIterator

{

public :

DlistIterator (const Dlist<Item> & L) // Hàm kiến tạo

{ LPtr = &L; current = -1; }

void Start( )

{ current = 0; }

bool Valid( ) const

{ return 0 <= current && current <= LPtr last; }

void Advance( )

{ assert (Valid( )); current + +;}

Item & Current( ) const

{ assert(Valid( )); return LPtr element[current]; }

void Add(const Item & x);

void Remove( );

private :

const Dlist<Item>\* LPtr;

int current;

}; ĐỊNH NGHĨA LỚP CÔNG CỤ LẶP

Chú ý rằng, chúng ta đã khai báo lớp DlistIterator là bạn của lớp Dlist. Điều này là để khi cài đặt các hàm thành phần của lớp DlistIterator chúng ta có quyền truy cập trực tiếp tới các thành phần dữ liệu của lớp Dlist. Bây giờ chúng ta cài đặt hàm Add, hàm này được cài đặt hoàn toàn tương tự như hàm Insert trong lớp Dlist. Chỉ cần để ý rằng, ở đây chúng ta cần xen một phần tử mới vào vị trí hiện thời trong danh sách mà con trỏ LPtr trỏ tới.

template <class Item>

void DlistIterator<Item> :: Add(const Item & x)

{

if (LPtr ÆLength( ) < LPtr Æ size)

{

LPtr Æ last + +;

for (int k = LPtr Æ last; k > current; k - - )

LPtr Æ element[k] = LPtr Æ element[k - 1];

LPtr Æelement[current] = x;

}

else {

Item\* newArray = new Item[ 2 \* (LPtr Æ size + 1) ];

assert(newArray != NULL);

for(int i = 0; i < current; i + + )

newArray[i] = LPtr Æelement[i];

newArray[current] = x;

for(int i = current + 1; i <= LPtr ÆLength( ); i + +)

newArray[i] = LPtr Æelement[i - 1];

delete [ ] LPtr Æ element;

LPtr Æ element = newArray;

LPtr Æ size = 2 \* (LPtr Æ size) + 1;

LPtr Æ last + +;

}

current + +;

}

Dễ dàng thấy rằng, mặc dầu các phép toán Insert, Append và Add cài đặt phức tạp hơn các phép toán tương ứng khi danh sách được cài đặt bởi mảng tĩnh, song thời gian thực hiện các phép toán đó vẫn chỉ là O(n).

Chúng ta đưa ra một ví dụ minh hoạ cách sử dụng lớp công cụ lặp. Giả sử L là danh sách các số nguyên. Chúng ta cần thêm số nguyên 4 vào trước số nguyên 5 xuất hiện lần đầu trong danh sách, nếu L không chứa 5 thì thêm 4 vào đuôi danh sách. Trước hết, chúng ta phải tạo ra một đối tượng của lớp công cụ lặp gắn với danh sách L, và cần nhớ rằng, mọi phép lặp cần đến thao tác đầu tiên là Start. Xem đoạn mã sau:

Dlist<int> L(100); // Danh sách L có thể chứa được tối đa 100

// số nguyên.

….

DlistIterator<Int> itr(L); // khởi tạo đối tượng lặp itr gắn với

// danh sách L.

itr.Start( );

while(itr.Valid( ))

if (itr.Current( ) = = 5)

{ itr.Add(4); break; }

else itr.Advance( );

if (!itr.Valid( ))

L. Append(4);

Bài 4: Cài đặt tập động bởi danh sách. Tìm kiếm tuần tự và tìm kiếm nhị phân

Nhớ lại rằng, mỗi đối tượng của KDLTT tập động là một tập các phần tử dữ liệu , và các phần tử dữ liệu chứa một thành phần dữ liệu được gọi là khoá. Chúng ta giả thiết rằng, trên các giá trị khoá có quan hệ thứ tự và các phần tử dữ liệu khác nhau có giá trị khoá khác nhau. Trên tập các phần tử dữ liệu S, chúng ta xác định các phép toán cơ bản sau:

1. Insert(S, x). Xen phần tử dữ liệu x vào tập S.

2. Delete(S, k). Loại khỏi tập S phần tử dữ liệu có khoá k.

3. Search(S, k). Tìm phần tử dữ liệu có giá trị khoá là k trong tập S. Hàm trả về true nếu tìm thấy và false nếu ngược lại.

4. Max(S). Hàm trả về phần tử dữ liệu có giá trị khoá lớn nhất trong tập S.

5. Min(S). Hàm trả về phần tử dữ liệu có giá trị khoá nhỏ nhất trong tập S.

Để viết cho ngắn gọn, chúng ta giả sử rằng các phần tử dữ liệu có kiểu là Item và có thể truy cập trực tiếp tới thành phần khoá của Item, chẳng hạn trong các áp dụng thông thường Item là một cấu trúc:

***struct*** Item

{

keyType key; // khoá của dữ liệu.

// các thành phần khác.

};

1. Cài đặt bởi danh sách không được sắp. Tìm kiếm tuần tự

Chúng ta có thể sắp xếp các phần tử của tập động (theo một thứ tự tuỳ ý) thành một danh sách, và do đó dễ dàng thấy rằng, ta có thể sử dụng cách cài đặt danh sách bởi mảng động để cài đặt KDLTT tập động. Lớp tập động Dset (Dynamic Set) được thiết kế bằng cách sử dụng lớp Dlist (xem mục 4.3) làm lớp cơ sở với dạng thừa kế private. Với cách thiết kế này , các hàm thành phần của lớp Dlist trở thành các hàm thành phần private của lớp DSet và chúng ta sẽ sử dụng chúng để cài đặt các phép toán tập động. Lớp Dset chỉ chứa các thành phần dữ liệu được thừa kế từ lớp Dlist. Định nghĩa lớp Dset được cho trong hình 4.5.

template <class Item>

class Dset : private Dlist<Item>

{

public :

DSet(int m = 1)

// khởi tạo ra tập rỗng, có thể chứa được nhiều nhất là m phần tử

// dữ liệu.

: Dlist(m) { }

void DSetInsert(const Item & x);

// Postcondition: phần tử x trở thành thành viên của tập động.

void DSetDelete(keyType k);

// Postcondition: phần tử với khoá k không có trong tập động.

bool Search(keyType k);

// Postcondition: Trả về true nếu phần tử với khoá k có trong

// tập động và trả về false nếu không.

Item & Max( );

// Precondition: Danh sách không rỗng.

// Postcondition: Trả về phần tử có khoá lớn nhất trong tập động.

Item & Min( );

// Precondition: Danh sách không rỗng.

// Postcondition: Trả về phần tử có khoá nhỏ nhất trong tập động.

};

ĐỊNH NGHĨA LỚP DSET

Sau đây chúng ta cài đặt các phép toán trên n tập động bằng cách sử dụng các hàm được thừa kế từ lớp danh sách.

***Hàm DSetInsert***. Hàm này được thực hiện bằng cách thêm phần tử mới vào đuôi danh sách:

***template*** <class Item>

void DSet<Item> :: DSetInsert(const Item & x)

{

***if*** (! Search(x.key))

Append(x);

}

***Hàm DSetDelete***. Để loại phần tử có khóa k khỏi tập động, chúng ta cần xem xét từng phần tử trong danh sách, khi gặp phần tử cần loại ở vị trí thứ i trong danh sách thì sử dụng hàm Delete(i) thừa kế từ lớp Dlist.

***template*** <class Item>

void DSet<Item> : DSetDelete(keyType k)

{

***for***(int i = 1; i <= Length( ); i + +)

***if*** (Element(i).key = = k)

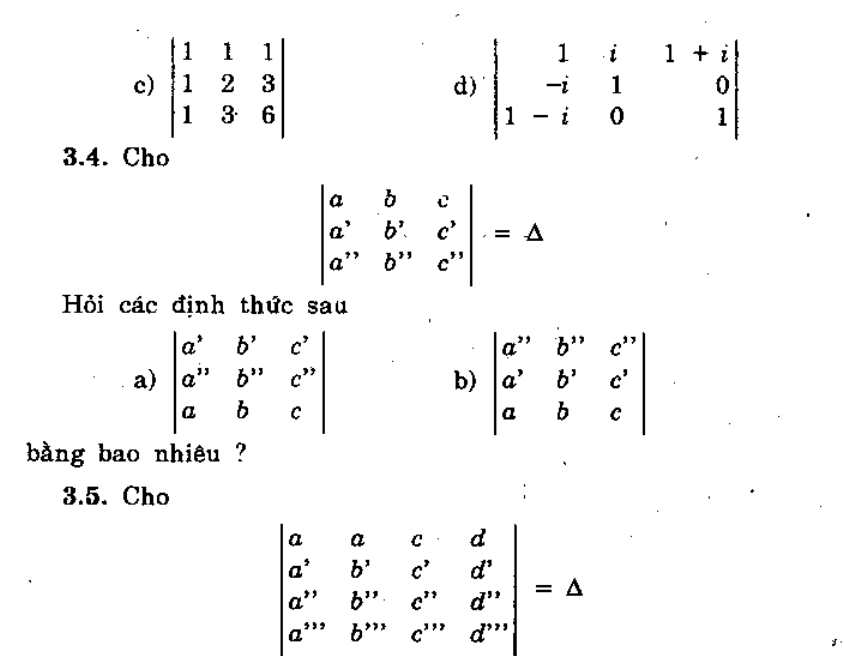
{

Delete(i);

break;

}

}



Chương trình tiếng anh 🪴

1. Vocabulary
2. Grammar
3. Pronanciation
4. Listenning – reading
5. Speking – writing

Cấu trúc dữ liệu và giải thuật

Chương 1

Chương 2

Chương 3

Chương 4

Chương 5

Xác suất thống kê

Chương 1

Chương 2

Chương 3

Chương 4

Chương 5

Chương 6

Đại số tuyến tính

Chương 1

Chương 2

Chương3

Chương 4

Chương 5

Chương 6

Cơ sở dữ liệu

Chương 1

Chương 2

Chương 3

Chương 4

Chương 5

Chương 6

Python

Chương 1

Chương 2

Chương 3

Chương 4

Chương 5

Chương 6

Giải tích 1

Chương 1

Chương 2

Chương 3

Chương 4

Chương 5

Chương 6

C++

Chương 1

Chương 2

Chương 3

Chương 4

Chương 5

Chương 6

Java

Chương 1

Chương 2

Chương 3

Chương 4

Chương 5

Chương 6

Html – Css – Javacrip

Chương 1

Chương 2

Chương 3

Chương 4

Chương 5

Chương 6

Nhập môn trí tuệ nhân tạo

I, Học máy

Nhập môn khoa học dữ liệu

Viết chương trình nhập mảng n hàng hóa(mỗi mặt hàng gồm mahang, dongia, soluong). Tính thành tiền mỗi mặt hàng =dongia\*soluong. Sắp xếp danh sách hàng hóa theo thành tiền tăng dần

Cho 1 dãy số gồm các phần tử đôi một khác nhau a1, a2,...,aN và một giá trị b. Đếm số cặp (ai, aj) sao cho ai+aj=b(i<j)

SÁCH CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN - TRẦN THÔNG QUẾ

Chương 1: Bìa tập về cấu trúc dữ liệu kiểu trừu tượng

Bài 1: Danh sách tuyến tính

Cho trước kiểu dữ liệu T={V,O}, Danh sách là tập các phần tử cùng kiểu TL được xác định như sau : TL = {VL,OL}

Trong đó:

VL: Tập các ptu co thứ tự tuyến tính và cùng kiểu T. Ta nói “thứ tự tuyến tính ” nghĩa là các phần tử của TL được sắp tuyến tính: Phần tử ai đứng trước (kể từ trái sang) phần tử ai+1 với i=1,2,3,...n(n là số phần tử thuộc danh sách TL còn gọi là độ dài của TL).Chỉ số (index) i biểu thị cho các vị trí của các phần tử thuộc TL, nghĩa là muốn truy cập vào 1 phần tử tùy ý của danh sách ta truy cập theo chỉ số i.

OL: Tập khởi tạo các thao tác (operations) trên các phần tử TL. Cca thao tác ấy là: Khởi tạo danh sách, tìm một phần tử trong danh sách, chèn thêm phần tử vào đầu , giữa, cuối danh sách, xóa 1 phần tử thuộc danh sách, sắp xếp danh sách, xem (duyệt) danh sách....

1. Danh sách liên kết đơn

Danh sách liên