Danh sách liên kết kép

Thực hiện các thao tác sau bằng danh sách liên kết kép.

Input

Gồm nhiều dòng, số dòng không vượt quá 200, mỗi dòng theo qui cách sau, số đầu tiên là:

0, số tiếp theo là n (1<=n<=10^3), chèn n số vào cuối dánh sách, các số có giá trị từ -10^6 đến 10^6.

1, số tiếp theo là x: Chèn một phần tử x vào đầu danh sách

2, số tiếp theo là x: Chèn một phần tử x vào cuối danh sách

3, số tiếp theo là x và pos: Chèn phần tử x vào ví trị pos vào danh sách, trong trường hợp vị trí chèn không hợp lệ thì in ra -1.

4: Xóa phần tử đầu danh sách, trong trường hợp danh sách rỗng thì in ra -1.

5: Xóa phần tử cuối danh sách, trong trường hợp danh sách rỗng thì in ra -1.

6, số tiếp theo là pos: Xóa phần tử ở vị trí thứ pos danh sách, trong trường hợp danh sách rỗng thì in ra -1.

7: In danh sách, trong trường hợp danh sách rỗng thì in ra -1.

8: Sắp xếp không giảm (Dùng thuật toán selectionsort)

9: Sắp xếp không tăng (Dùng thuật toán selectionsort)

10: In ra giá trị nhỏ nhất, trong trường hợp danh sách rỗng thì in ra -1.

11: In giá trị lớn nhất, trong trường hợp danh sách rỗng thì in ra -1.

12: Xóa tất cả giá trị.

-1: Thoát

Output

Như yêu cầu của input.

Ví dụ

Input	Output
0 3 -7 8 4 7 8 7 1 6 7 3 12 6 3 12 2 7 12 7 0 4 5 8 7 -6 7 6 2 7 -1	-7 8 4 -7 4 8 6 -7 4 8 -1 6 12 -7 4 8 -1 5 8 7 -6 5 7 -6

CODE

```
#include<bits/stdc++.h>

using namespace std;

#define ll long long

#define pi pair<int,int>

#define all(a) a.begin(), a.end()

#define task "dslk_kep"

const int INF=1e9;

void file(){

	ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);

	freopen(task".inp","r",stdin);
```

```
//        freopen(task".out","w",stdout);
}
struct Node{
        int data;
        Node *next;
        Node *prev;
};
typedef Node* node;
node taomoi(int x){
        node tmp=new Node;
        tmp->data=x;
        tmp->next=NULL;
        tmp->prev=NULL;
        return tmp;
}
bool rong(node a){
        return a==NULL;
}
int kichthuoc(node a){
        int cnt=0;
        while(a!=NULL){
                ++cnt;
                a=a->next;
```

```
	}
	return cnt;
}
void chendau(node &a, int x){
	node tmp=taomoi(x);
	if(a==NULL)a=tmp;
	else{
		tmp->next=a;
		a->prev=tmp;
		a=tmp;
	}
}
void chencuoi(node &a, int x){
	node tmp=taomoi(x);
	if(a==NULL)a=tmp;
	else{
		node p=a;
		while(p->next!=NULL)p=p->next;
		p->next=tmp;
		tmp->prev=p;
		p=tmp;
	}
}
```

```
void chengiua(node &a, int x, int pos){
	int n=kichthuoc(a);
	if(pos<1||pos>n+1){
		cout<<-1<<endl;
		return;
	}
	if(pos==1){
		chendau(a, x);
		return;
	}
	if(pos==n+1){
		chencuoi(a, x);
		return;
	}
	node p=a;
	for(int i=1; i<pos; i++)p=p->next;
	node tmp=taomoi(x);
	tmp->next=p;
	p->prev->next=tmp;
	tmp->prev=p->prev;
	p->prev=tmp;
}
void in(node a){
```

```
	if(a==NULL){
		cout<<-1<<endl;
		return;
	}
	while(a!=NULL){
		cout<<a->data<<" ";
		a=a->next;
	}
	cout<<endl;
}
void xoadau(node &a){
	if(a==NULL){
		cout<<-1<<endl;
		return;
	}
	a=a->next;
}
void xoacuoi(node &a){
	if(a==NULL){
		cout<<-1<<endl;
		return;
	}
	if(kichthuoc(a)==1){
```

```
            a=NULL;
            return;
        }
        node p=a;
        for(int i=1; i<kichthuoc(a)-1; i++)p=p->next;
        p->next=NULL;
}
void xoagiua(node &a, int pos){
        int n=kichthuoc(a);
        if(pos==1){
            xoadau(a);
            return;
        }
        if(pos==n){
            xoacuoi(a);
            return;
        }
        node p=a, q=a;
        for(int i=1; i<pos; i++)q=p, p=p->next;
        q->next=p->next;
        p->next->prev=q;
}
void sapxep(node &a){
```

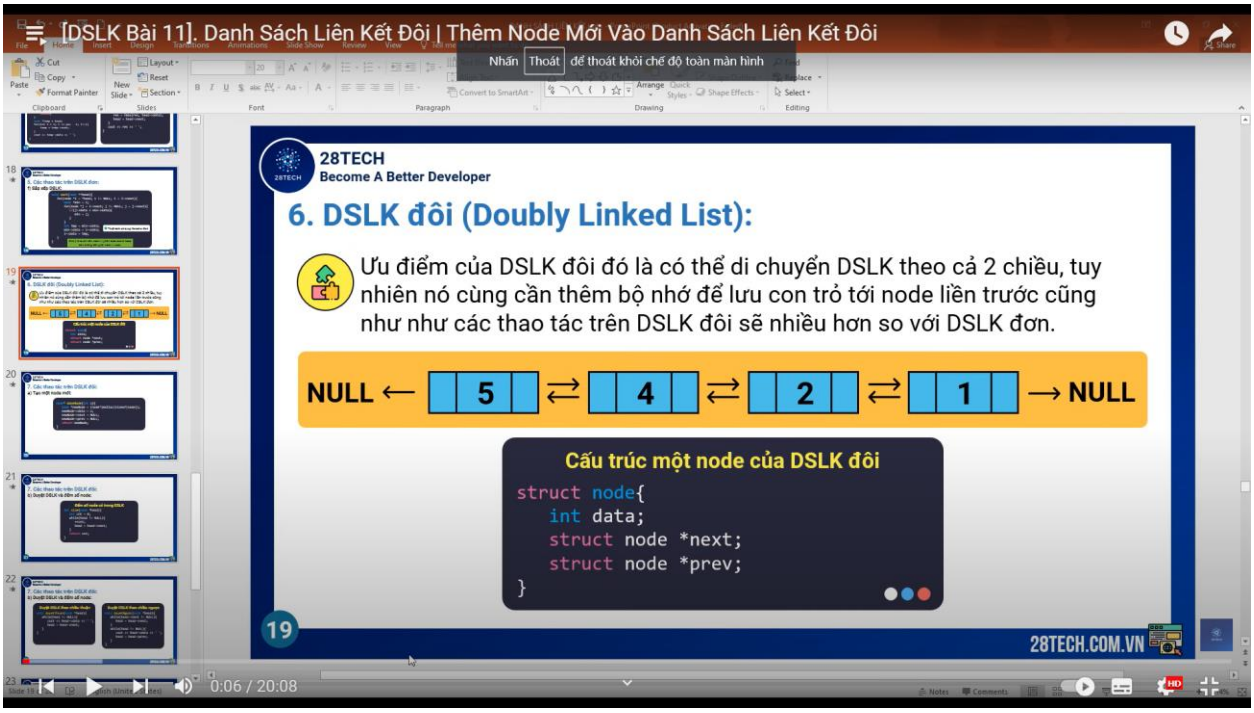
```
	if(a==NULL)return;
	for(node p=a; p!=NULL; p=p->next){
		node min=p;
		for(node q=p->next; q!=NULL; q=q->next)
			if(q->data<min->data)min=q;
		int tmp=min->data;
		min->data=p->data;
		p->data=tmp;
	}
}
void sapxepgiam(node &a){
	if(a==NULL)return;
	for(node p=a; p!=NULL; p=p->next){
		node max=p;
		for(node q=p->next; q!=NULL; q=q->next)
			if(q->data>max->data)max=q;
		int tmp=p->data;
		p->data=max->data;
		max->data=tmp;
	}
}
void timnn(node &a){
	if(rong(a)){
```

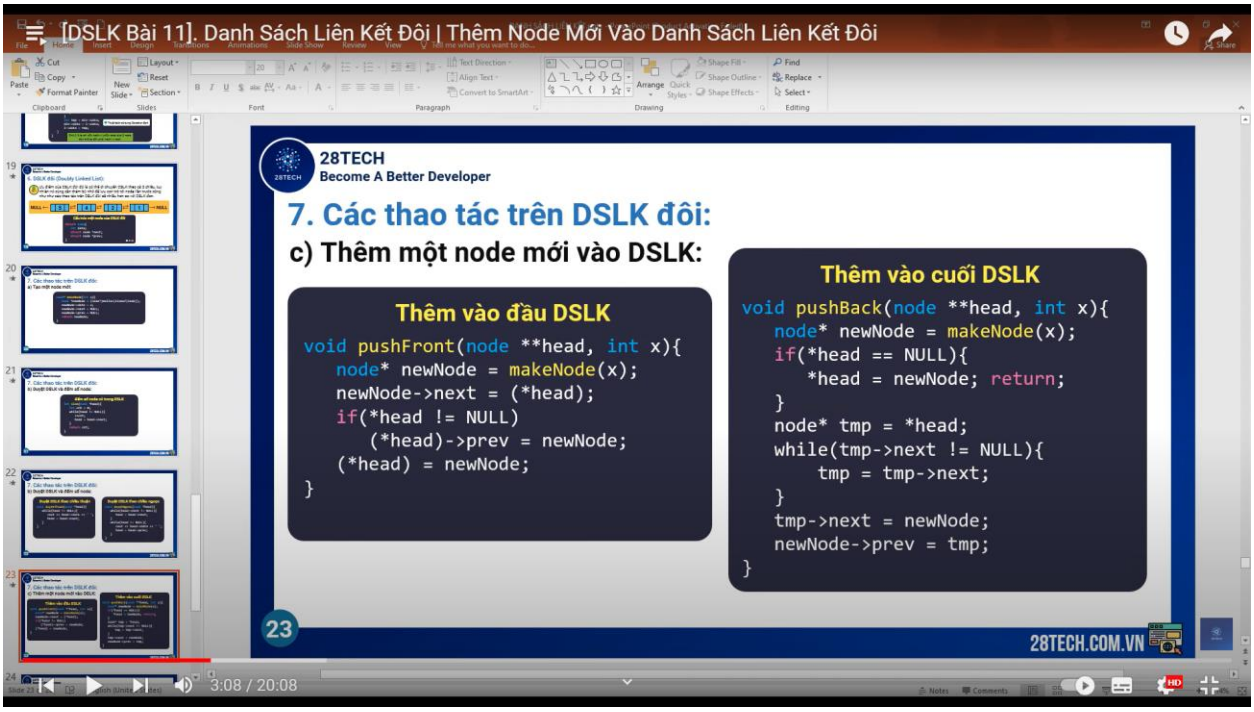
```
                cout<<-1<<endl;
                return;
        }
        sapxep(a);
        cout<<a->data<<endl;
}
void timln(node &a){
        if(rong(a)){
                cout<<-1<<endl;
                return;
        }
        sapxepgiam(a);
        cout<<a->data<<endl;
}
void xoa(node &a){
        while(a!=NULL)a=a->next;
}
void xuli(){
        node head=NULL;
        while(1){
                int t; cin>>t;
                if(t==-1)break;
                switch(t){
```

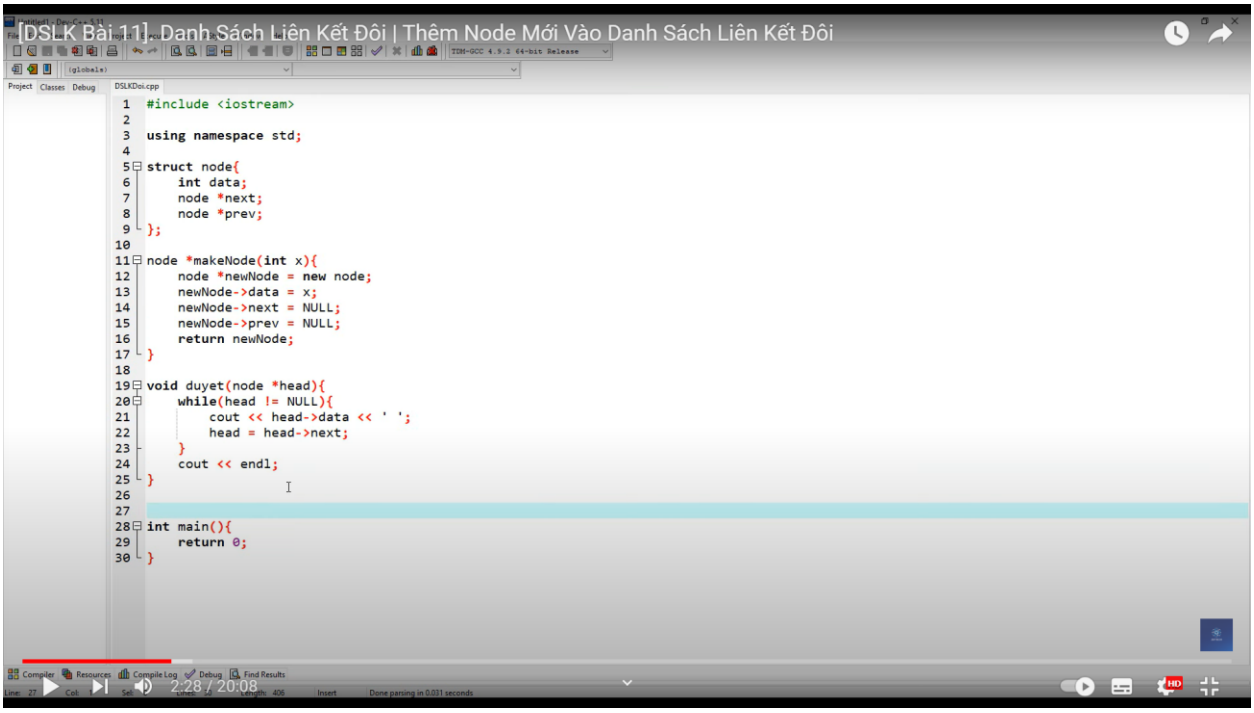
```
case 0:{
        int n; cin>>n;
        for(int i=1; i<=n; i++){
                int x; cin>>x;
                chencuoi(head, x);
        }
        break;
}
case 1:{
        int x; cin>>x; chendau(head, x);
        break;
}
case 2:{
        int x; cin>>x; chencuoi(head, x);
        break;
}
case 3:{
        int x, pos; cin>>x>>pos;
        chengiua(head, x, pos);
        break;
}
case 4: xoadau(head); break;
case 5: xoacuoi(head); break;
```

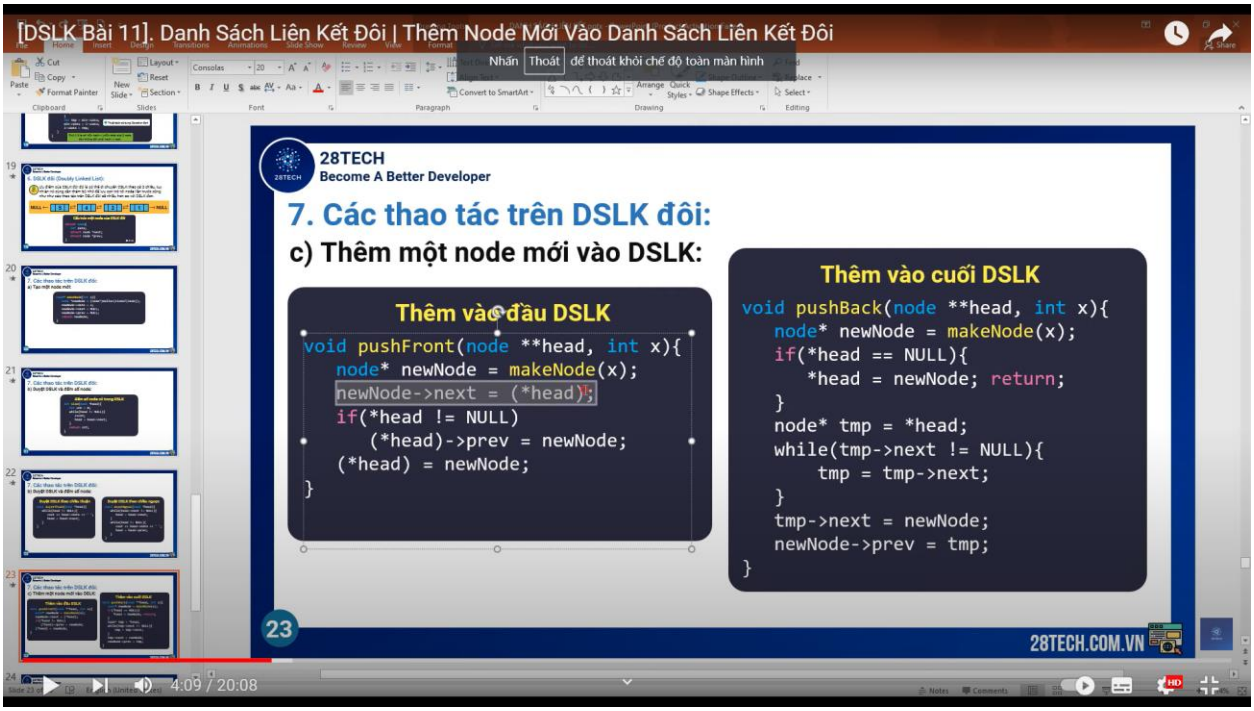
```
                case 6:{
                        int pos; cin>>pos; xoagiua(head, pos);
                        break;
                }
                case 7: in(head); break;
                case 8: sapxep(head); break;
                case 9: sapxepgiam(head); break;
                case 10: timnn(head); break;
                case 11: timln(head); break;
                case 12: xoa(head); break;
            }
        }
}
int main(){
        file();
        xuli();
}
```

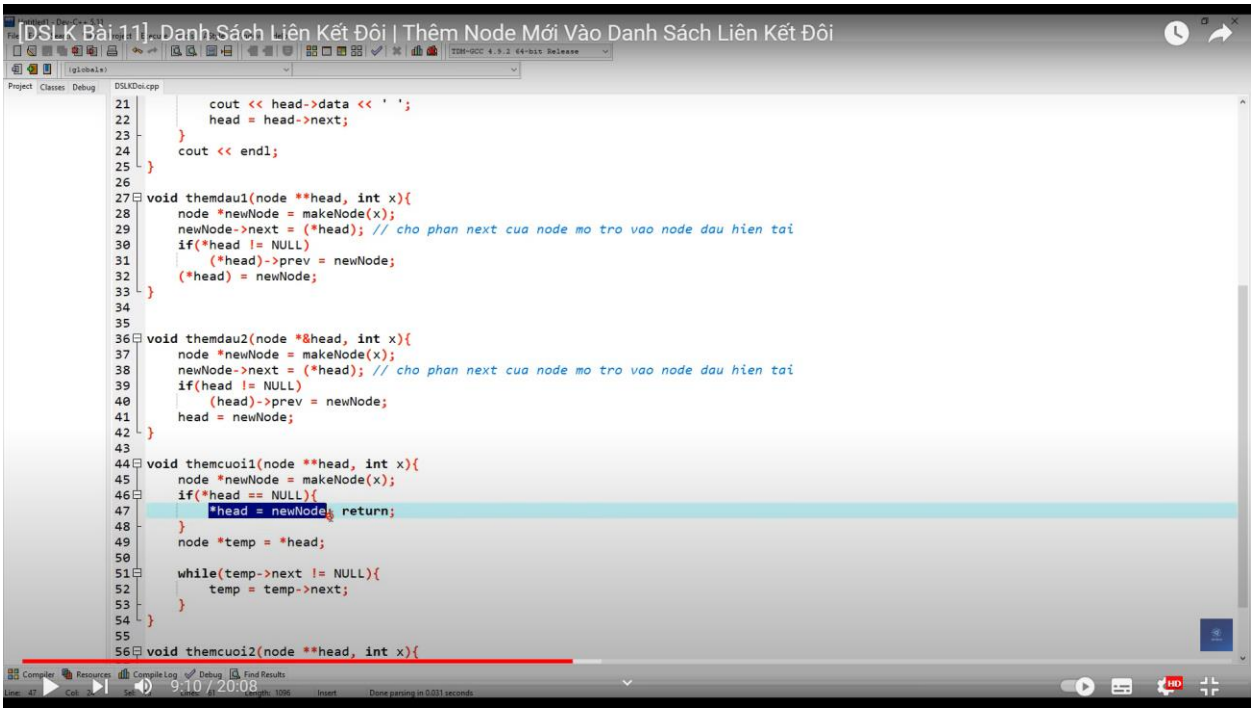
Previous: trước

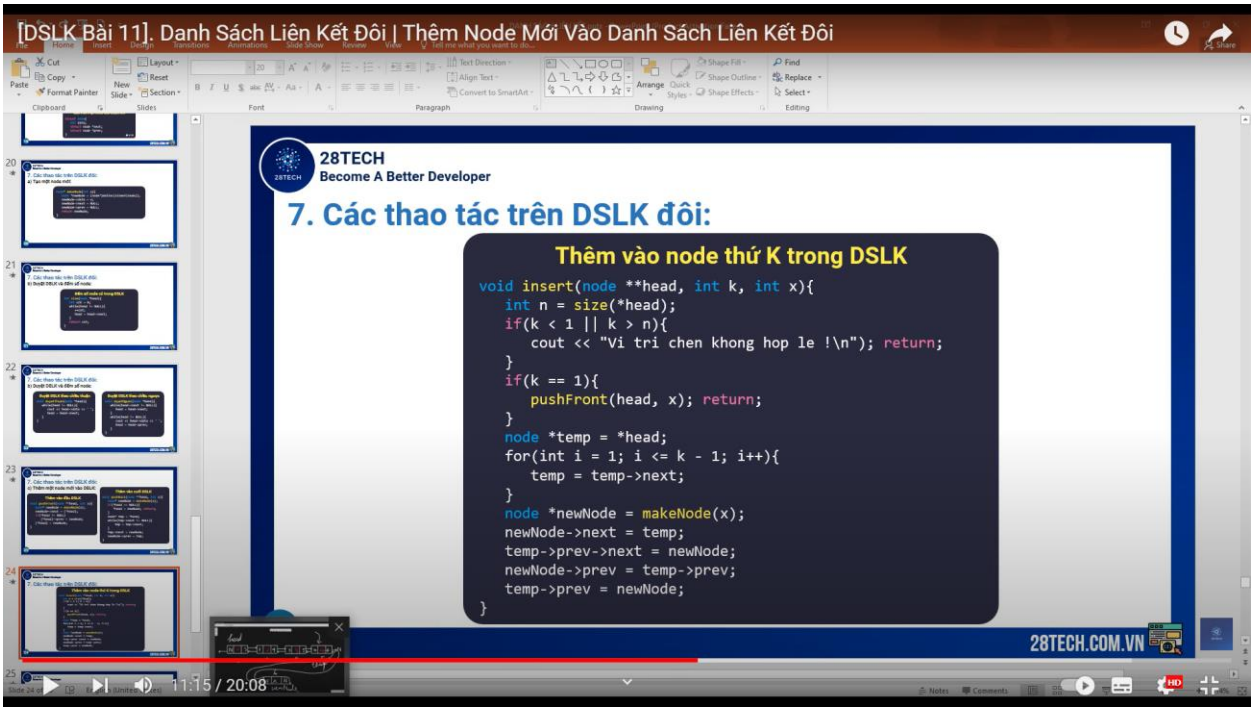

[DSLK Bài 11]. Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
Nhấn Thoát để thoát khỏi chế độ toàn màn hình
28TECH
Become A Better Developer
6. DSLK đôi (Doubly Linked List):
Ưu điểm của DSLK đôi đó là có thể di chuyển DSLK theo cả 2 chiều, tuy nhiên nó cùng cần thêm bộ nhớ để lưu con trỏ tới node liền trước cũng như như các thao tác trên DSLK đôi sẽ nhiều hơn so với DSLK đơn.
NULL ← 5 ⇄ 4 ⇄ 2 ⇄ 1 → NULL
Cấu trúc một node của DSLK đôi
struct node{
int data;
struct node *next;
struct node *prev;
}
19
28TECH.COM.VN
0:06 / 20:08

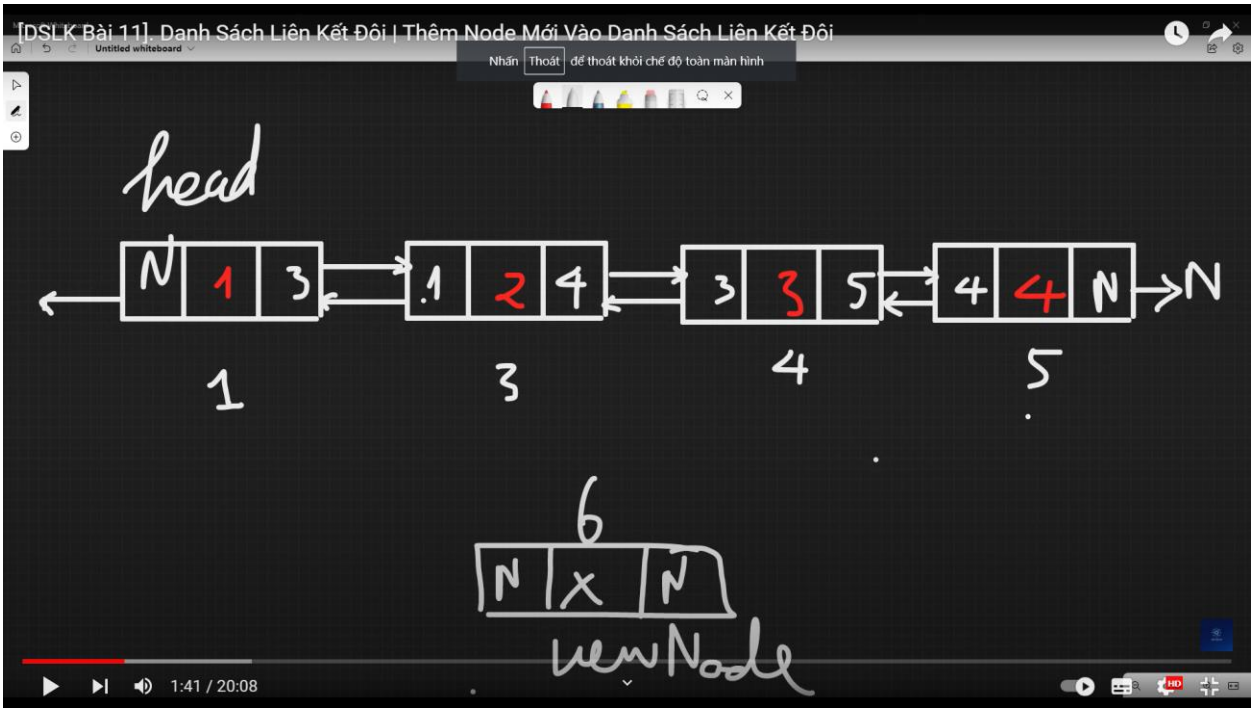

[DSLK Bài 11]. Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
28TECH
Become A Better Developer
7. Các thao tác trên DSLK đôi:
c) Thêm một node mới vào DSLK:
Thêm vào đầu DSLK
void pushFront(node **head, int x){
node* newNode = makeNode(x);
newNode->next = (*head);
if(*head != NULL)
(*head)->prev = newNode;
(*head) = newNode;
}
Thêm vào cuối DSLK
void pushBack(node **head, int x){
node* newNode = makeNode(x);
if(*head == NULL){
*head = newNode; return;
}
node* tmp = *head;
while(tmp->next != NULL){
tmp = tmp->next;
}
tmp->next = newNode;
newNode->prev = tmp;
}
23
28TECH.COM.VN
3:08 / 20:08

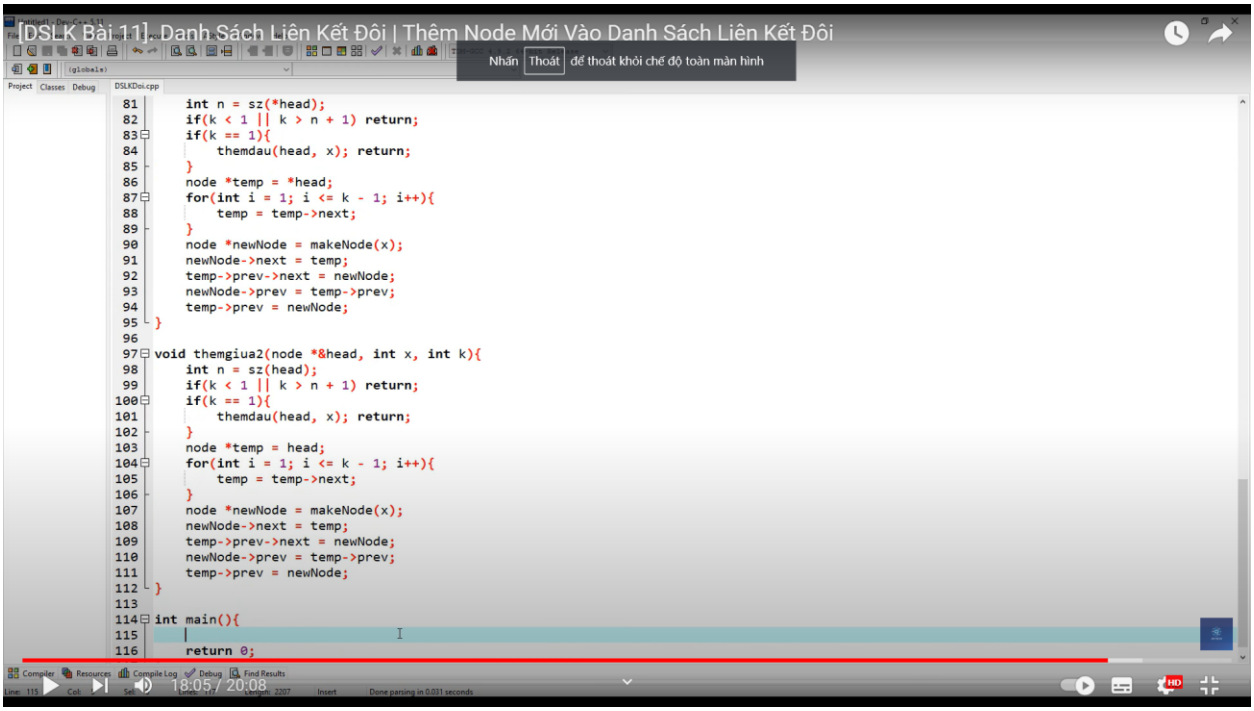

[DSLK Bài 11] Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
DSLKDoi.cpp
#include <iostream>
using namespace std;
struct node{
int data;
node *next;
node *prev;
};
node *makeNode(int x){
node *newNode = new node;
newNode->data = x;
newNode->next = NULL;
newNode->prev = NULL;
return newNode;
}
void duyet(node *head){
while(head != NULL){
cout << head->data << ' ';
head = head->next;
}
cout << endl;
}
int main(){
return 0;
}
2:28 / 20:08


[DSLK Bài 11]. Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
Nhấn Thoát để thoát khỏi chế độ toàn màn hình
28TECH
Become A Better Developer
7. Các thao tác trên DSLK đôi:
c) Thêm một node mới vào DSLK:
Thêm vào đầu DSLK
void pushFront(node **head, int x){
node* newNode = makeNode(x);
newNode->next = (*head);
if(*head != NULL)
(*head)->prev = newNode;
(*head) = newNode;
}
Thêm vào cuối DSLK
void pushBack(node **head, int x){
node* newNode = makeNode(x);
if(*head == NULL){
*head = newNode; return;
}
node* tmp = *head;
while(tmp->next != NULL){
tmp = tmp->next;
}
tmp->next = newNode;
newNode->prev = tmp;
}
23
28TECH.COM.VN
4:09 / 20:08


[DSLK Bài 11]. Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
DSLKDoi.cpp
cout << head->data << ' ';
head = head->next;
}
cout << endl;
}
void themdau1(node **head, int x){
node *newNode = makeNode(x);
newNode->next = (*head); // cho phan next cua node mo tro vao node dau hien tai
if(*head != NULL)
(*head)->prev = newNode;
(*head) = newNode;
}
void themdau2(node *&head, int x){
node *newNode = makeNode(x);
newNode->next = (*head); // cho phan next cua node mo tro vao node dau hien tai
if(head != NULL)
(head)->prev = newNode;
head = newNode;
}
void themcuoi1(node **head, int x){
node *newNode = makeNode(x);
if(*head == NULL){
*head = newNode; return;
}
node *temp = *head;
while(temp->next != NULL){
temp = temp->next;
}
}
void themcuoi2(node **head, int x){
9:10 / 20:08

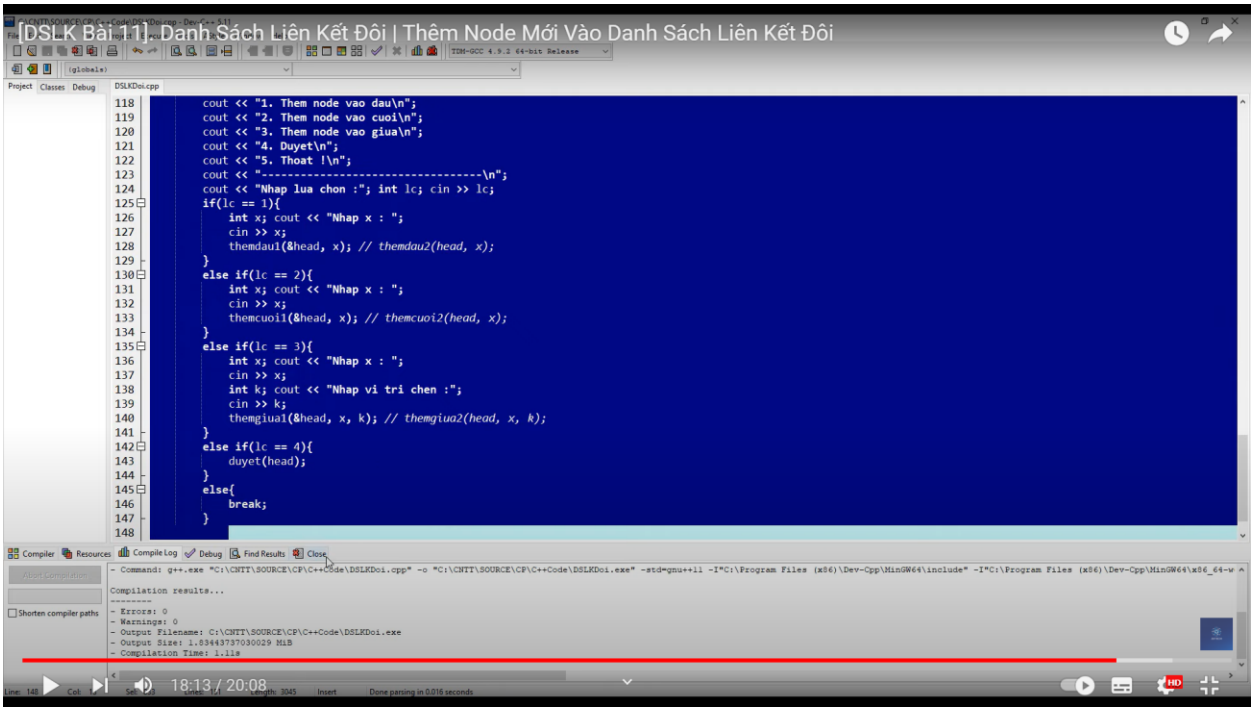

[DSLK Bài 11]. Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
28TECH
Become A Better Developer
7. Các thao tác trên DSLK đôi:
Thêm vào node thứ K trong DSLK
void insert(node **head, int k, int x){
int n = size(*head);
if(k < 1 || k > n){
cout << "Vi tri chen khong hop le !\n"); return;
}
if(k == 1){
pushFront(head, x); return;
}
node *temp = *head;
for(int i = 1; i <= k - 1; i++){
temp = temp->next;
}
node *newNode = makeNode(x);
newNode->next = temp;
temp->prev->next = newNode;
newNode->prev = temp->prev;
temp->prev = newNode;
}
28TECH.COM.VN
11:15 / 20:08

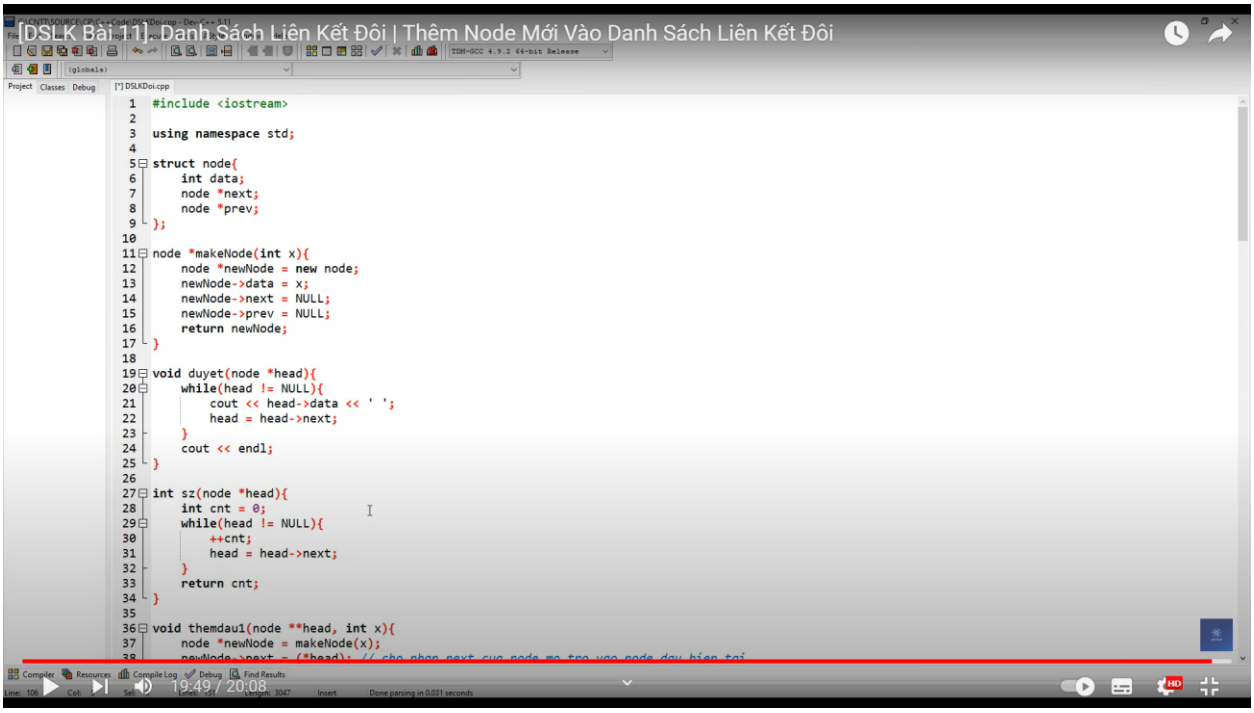


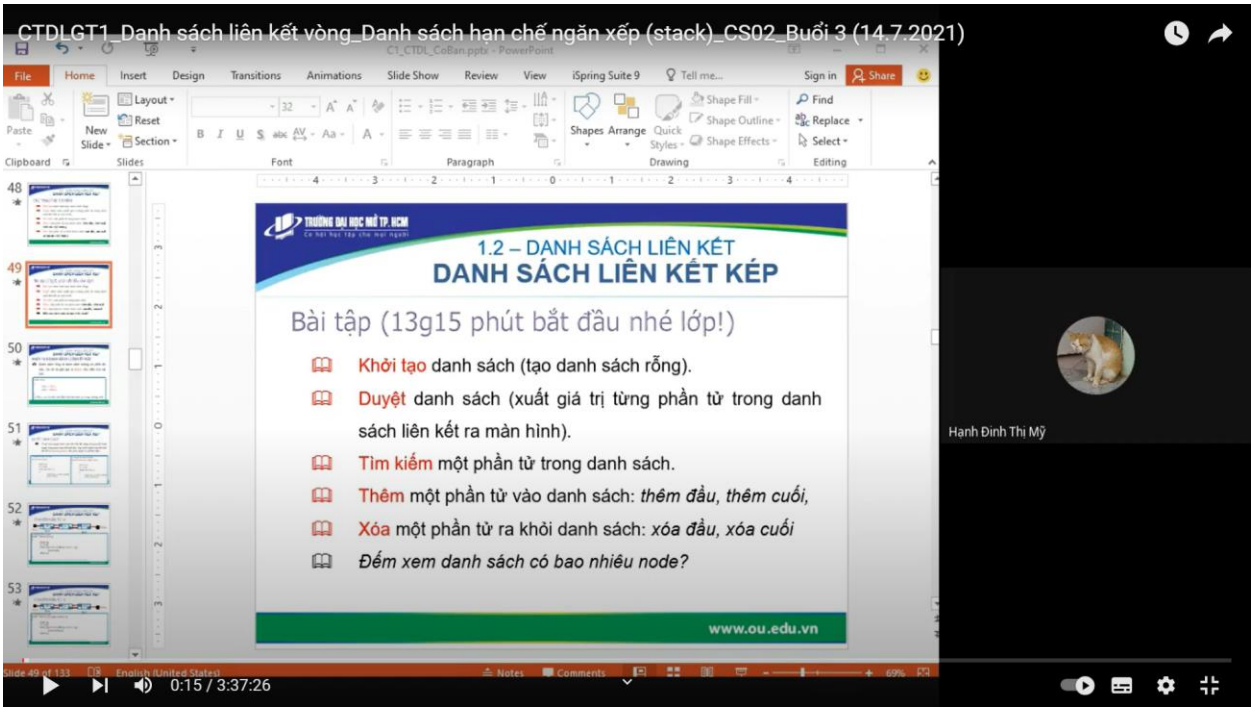



```
    int n = sz(*head);
    if(k < 1 || k > n + 1) return;
    if(k == 1){
        themdau(head, x); return;
    }
    node *temp = *head;
    for(int i = 1; i <= k - 1; i++){
        temp = temp->next;
    }
    node *newNode = makeNode(x);
    newNode->next = temp;
    temp->prev->next = newNode;
    newNode->prev = temp->prev;
    temp->prev = newNode;
}

void themgiua2(node *&head, int x, int k){
    int n = sz(head);
    if(k < 1 || k > n + 1) return;
    if(k == 1){
        themdau(head, x); return;
    }
    node *temp = head;
    for(int i = 1; i <= k - 1; i++){
        temp = temp->next;
    }
    node *newNode = makeNode(x);
    newNode->next = temp;
    temp->prev->next = newNode;
    newNode->prev = temp->prev;
    temp->prev = newNode;
}

int main(){
    
    return 0;
```


[DSLK Bài 11] Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
}
int main(){
node *head = NULL;
while(1){
cout << "----------------DSLK---------------\n";
cout << "1. Them node vao dau\n";
cout << "2. Them node vao cuoi\n";
cout << "3. Them node vao giua\n";
cout << "4. Duyet\n";
cout << "5. Thoat !\n";
cout << "----------------------------------\n";
cout << "Nhap lua chon :"; int lc; cin >> lc;
if(lc == 1){
int x; cout << "Nhap x : ";
cin >> x;
themdau1(&head, x); // themdau2(head, x);
}
else if(lc == 2){
int x; cout << "Nhap x : ";
cin >> x;
themcuoi1(&head, x); // themcuoi2(head, x);
}
else if(lc == 3){
int x; cout << "Nhap x : ";
cin >> x;
int k; cout << "Nhap vi tri chen :";
cin >> k;
Compiler
Resources
Compile Log
Debug
Find Results
Close
Compilation results...
- Errors: 0
- Warnings: 0
- Output Filename: C:\CNTT\SOURCE\CP\C++Code\DSLKDoi.exe
- Output Size: 1.83443737030029 MiB
- Compilation Time: 1.11s
18:09 / 20:08


[DSLK Bài 11] Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
cout << "1. Them node vao dau\n";
cout << "2. Them node vao cuoi\n";
cout << "3. Them node vao giua\n";
cout << "4. Duyet\n";
cout << "5. Thoat !\n";
cout << "----------------------------------\n";
cout << "Nhap lua chon :"; int lc; cin >> lc;
if(lc == 1){
int x; cout << "Nhap x : ";
cin >> x;
themdau1(&head, x); // themdau2(head, x);
}
else if(lc == 2){
int x; cout << "Nhap x : ";
cin >> x;
themcuoi1(&head, x); // themcuoi2(head, x);
}
else if(lc == 3){
int x; cout << "Nhap x : ";
cin >> x;
int k; cout << "Nhap vi tri chen :";
cin >> k;
themgiua1(&head, x, k); // themgiua2(head, x, k);
}
else if(lc == 4){
duyet(head);
}
else{
break;
}
Compiler
Resources
Compile Log
Debug
Find Results
Close
Compilation results...
- Errors: 0
- Warnings: 0
- Output Filename: C:\CNTT\SOURCE\CP\C++Code\DSLKDoi.exe
- Output Size: 1.83443737030029 MiB
- Compilation Time: 1.11s
18:13 / 20:08


[DSLK Bài 11] Danh Sách Liên Kết Đôi | Thêm Node Mới Vào Danh Sách Liên Kết Đôi
#include <iostream>
using namespace std;
struct node{
int data;
node *next;
node *prev;
};
node *makeNode(int x){
node *newNode = new node;
newNode->data = x;
newNode->next = NULL;
newNode->prev = NULL;
return newNode;
}
void duyet(node *head){
while(head != NULL){
cout << head->data << ' ';
head = head->next;
}
cout << endl;
}
int sz(node *head){
int cnt = 0;
while(head != NULL){
++cnt;
head = head->next;
}
return cnt;
}
void themdau1(node **head, int x){
node *newNode = makeNode(x);
19:49 / 20:08


CTDLGT1_Danh sách liên kết vòng_Danh sách hạn chế ngăn xếp (stack)_CS02_Buổi 3 (14.7.2021)
1.2 – DANH SÁCH LIÊN KẾT
DANH SÁCH LIÊN KẾT KÉP
Bài tập (13g15 phút bắt đầu nhé lớp!)
Khởi tạo danh sách (tạo danh sách rỗng).
Duyệt danh sách (xuất giá trị từng phần tử trong danh sách liên kết ra màn hình).
Tìm kiếm một phần tử trong danh sách.
Thêm một phần tử vào danh sách: thêm đầu, thêm cuối,
Xóa một phần tử ra khỏi danh sách: xóa đầu, xóa cuối
Đếm xem danh sách có bao nhiêu node?
www.ou.edu.vn
Hạnh Đinh Thị Mỹ
0:15 / 3:37:26

3. Độ phức tạp của các thao tác với mảng và DSLK:

Thao tác	DSLK	Mảng
Truy xuất phần tử	O(n)	O(1)
Chèn/Xóa ở đầu	O(1)	O(n) nếu mảng chưa full
Chèn ở cuối	O(n)	O(1) nếu mảng chưa full
Xóa ở cuối	O(n)	O(1)
Chèn giữa	O(n)	O(n) nếu mảng chưa full
Xóa giữa	O(n)	O(n)

CHƯƠNG 5: CON TRỎ VÀ HÀM

-----ᘓ✧ᘒ-----

Con trỏ là một biến dùng để giữ địa chỉ của biến khác. Nếu p là con trỏ giữ địa chỉ của biến x ta gọi p trỏ tới x và x được trỏ bởi p. Thông qua con trỏ ta có thể làm việc được với nội dung của những ô nhớ mà p trỏ đến.

Con trỏ là một đặc trưng mạnh của C++, nó cho phép chúng ta thâm nhập trực tiếp vào bộ nhớ để xử lý các bài toán khó chỉ bằng vài câu lệnh đơn giản của chương trình.

5.1.1 Khai báo

<Kiểu được trỏ> <*Tên biến con trỏ> ;

<Kiểu được trỏ*> <Tên biến con trỏ> ;

Địa chỉ của một biến là địa chỉ byte nhớ đầu tiên của biến đó. Vì vậy để lấy được nội dung của biến, con trỏ phải biết được số byte của biến, tức là kiểu của biến mà con trỏ sẽ trỏ tới. Kiểu này cũng được gọi là kiểu của con trỏ. Như vậy khai báo biến con trỏ cũng giống như khai báo một biến thường ngoại trừ cần thêm dấu * trước tên biến (hoặc sau tên kiểu).

Ví dụ:

int *p ; // khai báo biến p là biến con trỏ trỏ đến kiểu số nguyên

float *q, *r ; // khai báo 2 con trỏ thực q và r

5.1.2 Cách sử dụng

- Để con trỏ p trỏ đến biến x ta phải dùng phép gán p = địa chỉ của x. Nếu x không phải là biến mảng thì ta viết: p = &x. Còn nếu x là biến mảng thì ta viết: p = x hoặc p = &x[0].

- Ta không thể gán p cho một hằng địa chỉ cụ thể. Chẳng hạn ta viết p = 200 là sai.

- Phép toán * cho phép lấy nội dung nơi p trỏ đến, ví dụ để gán nội dung nơi p trỏ đến

cho biến f ta viết f = *p.

- Phép toán & và phép toán * là 2 phép toán ngược nhau. Cụ thể nếu p = &x thì x = *p. Từ đó nếu p trỏ đến x thì bất kỳ nơi nào xuất hiện x đều có thể thay được bởi *p và ngược lại.

Ví dụ 1:

```
#include <iostream>
using namespace std;
main()
{    int i, j; // khai báo 2 biến nguyên i, j
     int *p, *q; // khai báo 2 con trỏ nguyên p, q
     p = &i; // cho p trỏ tới i
     q = &j; // cho q trỏ tới j
     cout << &i; // hỏi địa chỉ biến i
     cout << q; // hỏi địa chỉ biến j (thông qua q)
     i = 2; // gán i bằng 2
     *q = 5; // gán j bằng 5 (thông qua q)
     i++ ; cout << i;   // tăng i và in ra i = 3
     (*q)++ ; cout << j;// tăng j (thông qua q) và in j = 6
     (*p) = (*q) * 2 + 1; // gán lại i (thông qua p)
     cout << i; // i = 13
}
```

Qua ví dụ trên ta thấy mọi thao tác với i là tương đương với *p, với j là tương đương với *q và ngược lại.

5.1.3 Các phép toán

5.1.3.1 Phép toán gán

- Gán con trỏ với địa chỉ một biến: p = &x ;

- Gán con trỏ với con trỏ khác: p = q ; (sau phép toán gán này p, q chứa cùng một địa chỉ hay cùng trỏ đến một nơi).

Ví dụ 2:

```
#include <iostream>
```

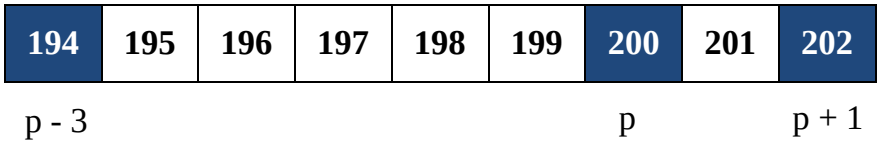
```
using namespace std;
main()
{    int i = 10;
     int *p, *q, *r;
     p = q = r = &i;
     *p = *q + 3;
     *q = *r * 2;
     cout << i << endl;
}
```

5.1.3.2 Phép toán tăng, giảm địa chỉ

p ± n: con trỏ trỏ đến thành phần thứ n sau (trước) p.

Một đơn vị tăng giảm của con trỏ bằng kích thước của biến được trỏ.

Ví dụ: Giả sử p là con trỏ nguyên (2 bytes) đang trỏ đến địa chỉ 200 thì p + 1 là con trỏ trỏ đến địa chỉ 202; tương tự, p + 5 là con trỏ trỏ đến địa chỉ 210; và p - 3 chứa địa chỉ 194.



Như vậy, phép toán tăng, giảm con trỏ cho phép làm việc thuận lợi trên mảng. Nếu con trỏ đang trỏ đến mảng (tức đang chứa địa chỉ đầu tiên của mảng), việc tăng con trỏ lên 1 đơn vị sẽ dịch chuyển con trỏ trỏ đến phần tử thứ hai, và cứ thế tiếp tục. Từ đó ta có thể cho con trỏ chạy từ đầu đến cuối mảng bằng cách tăng con trỏ lên từng đơn vị như trong câu lệnh for trong ví dụ dưới đây.

Ví dụ 3:

```
#include <iostream>
using namespace std;
main()
{    int a[10] = { 1, 2, 3, 4, 5, 6, 7 }, *p, *q;
     p = a; // cho p trỏ đến mảng a
     cout << *p ;  // *p = a[0] = 1
     p += 5;
     cout << *p ; // *p = a[5] = 6
```

```
    q = p - 4 ;
    cout << *q ; // q = a[1] = 2
    p = a;
    for (int i = 0;  i < 10;  i++)
        cout << *(p + i) << '\t' ;  // in toàn bộ mảng a
}
```

5.1.3.3 Phép toán tự tăng, tự giảm

p++, p--, ++p, --p: Tương tự p + 1 và p - 1, có chú ý đến tăng (giảm) sau hay trước.

Ví dụ sau minh họa kết quả kết hợp phép tự tăng, tự giảm với việc lấy giá trị nơi con trỏ trỏ đến. a là một mảng gồm 2 số, p là con trỏ trỏ đến mảng a.

Ví dụ 4:

```
#include <iostream>
using namespace std;
main()
{   int a[2] = {3, 7}, *p = a;
    cout << (*p)++ << endl; // in ra 3
    cout << *p << endl;     // in ra 4
    cout << ++(*p) << endl; // in ra 5
    cout << *p << endl;     // in ra 5
    cout << *(p++) << endl; // in ra 5
    cout << *p << endl;     // in ra 7
}
```

✧ ***Chú ý***: Ta phải phân biệt giữa p + 1 và p++ (hoặc ++p):

• p + 1 được xem như một con trỏ khác với con trỏ p. Con trỏ p + 1 trỏ đến phần tử sau p.

• p++ là con trỏ p nhưng trỏ đến phần tử khác. Con trỏ p++ trỏ đến phần tử đứng sau phần tử p trỏ đến ban đầu.

5.1.3.4 Phép hiệu của 2 con trỏ

Phép toán này chỉ thực hiện được khi p và q là 2 con trỏ cùng trỏ đến các phần tử của một dãy dữ liệu nào đó trong bộ nhớ (chẳng hạn cùng trỏ đến 1 mảng dữ liệu).

Khi đó hiệu p - q là số phần tử giữa p và q (chú ý p - q không phải là hiệu của 2 địa chỉ

mà là số phần tử giữa p và q).

Ví dụ:

Giả sử p và q là 2 con trỏ số nguyên có kích thước 2 bytes, p có địa chỉ 200 và q có địa chỉ 208. Khi đó p - q = - 4 và q - p = 4 (4 là số phần tử nguyên từ địa chỉ 200 đến 208).

5.1.3.5 Phép toán so sánh

Các phép toán so sánh cũng được áp dụng đối với con trỏ, thực chất là so sánh giữa địa chỉ của hai nơi được trỏ bởi các con trỏ này. Thông thường các phép so sánh <, <=, >, >= chỉ áp dụng cho hai con trỏ trỏ đến phần tử của cùng một mảng dữ liệu nào đó. Thực chất của phép so sánh này chính là so sánh chỉ số của 2 phần tử được trỏ bởi 2 con trỏ đó.

Ví dụ 5:

```
#include <iostream>
using namespace std;
main()
{    float a[5] = {1, 2, 3, 4, 5}, *p, *q;
     p = a; // p trỏ đến mảng (tức p trỏ đến a[0])
     q = &a[3]; // q trỏ đến phần tử thứ 3 (a[3]) của
     mảng cout << (p < q) << endl;   // in ra 1 (true)
     cout << (p + 3 == q) << endl; // in ra 1 (true)
     cout << (p > q - 1) << endl;  // in ra 0 (false)
     cout << (p >= q - 2) << endl; // in ra 0 (false)
     for (p = a ; p < a + 5; p++)
          cout << *p << '\t'; // in toàn bộ mảng a
     cout << endl;
}
```

5.1.5 Cấp phát động, toán tử cấp phát và thu hồi vùng nhớ

5.1.5.1 Cấp phát động

Khi tiến hành chạy chương trình, chương trình dịch sẽ bố trí các ô nhớ cụ thể cho các biến được khai báo trong chương trình. Vị trí cũng như số lượng các ô nhớ này tồn tại và cố định trong suốt thời gian chạy chương trình, chúng xem như đã bị chiếm dụng và sẽ không được sử dụng vào mục đích khác và chỉ được giải phóng sau khi chấm dứt chương trình. Việc phân bổ bộ nhớ như vậy được gọi là ***cấp phát tĩnh*** (vì được cấp sẵn

trước khi chạy chương trình và không thể thay đổi tăng, giảm kích thước hoặc vị trí trong suốt quá trình chạy chương trình).

Ví dụ nếu ta khai báo một mảng nguyên chứa 1000 phần tử thì trong bộ nhớ sẽ có một vùng nhớ liên tục 2000 bytes để chứa dữ liệu của mảng này. Khi đó dù trong chương trình ta chỉ nhập vào mảng và làm việc với một vài phần tử thì phần mảng rỗi còn lại vẫn không được sử dụng vào việc khác. Đây là hạn chế thứ nhất của kiểu mảng. Ở một hướng khác, một lần nào đó chạy chương trình ta lại cần làm việc với hơn 1000 số nguyên. Khi đó vùng nhớ mà chương trình dịch đã dành cho mảng là không đủ để sử dụng. Đây chính là hạn chế thứ hai của mảng được khai báo trước.

Để khắc phục các hạn chế trên của kiểu mảng, bây giờ chúng ta sẽ không khai báo (bố trí) trước mảng dữ liệu với kích thước cố định như vậy. Kích thước cụ thể sẽ được cấp phát trong quá trình chạy chương trình theo đúng yêu cầu của NLT. Nhờ vậy chúng ta có đủ số ô nhớ để làm việc mà vẫn tiết kiệm được bộ nhớ, và khi không dùng nữa ta có thể thu hồi (còn gọi là giải phóng) số ô nhớ này để chương trình sử dụng vào việc khác. Hai công việc cấp phát và thu hồi này được thực hiện thông qua các toán tử new và delete, để thực hiện được điều này ta cần phải sử dụng biến kiểu con trỏ. Thông qua biến con trỏ ta có thể làm việc với bất kỳ địa chỉ nào của vùng nhớ được cấp phát. Cách thức bố trí bộ nhớ như thế này được gọi là ***cấp phát động***.

5.1.5.2 Toán tử cấp phát vùng nhớ (new)

Cú pháp:

p = new <kiểu> ;

p = new <kiểu>[n] ;

Với p là biến con trỏ.

Ví dụ:

```
int *p;
p = new int;
p = new int[10];
```

Khi gặp toán tử new, chương trình sẽ tìm trong bộ nhớ một lượng ô nhớ còn rỗi và liên tục với số lượng đủ theo yêu cầu và cho p trỏ đến địa chỉ (byte đầu tiên) của vùng nhớ này. Nếu không có vùng nhớ với số lượng như vậy thì việc cấp phát là thất bại và khi đó p bằng NULL (NULL là một địa chỉ rỗng, không xác định). Do vậy ta có thể kiểm tra việc cấp phát có thành công hay không thông qua việc kiểm tra con trỏ p bằng hay khác NULL.

Ví dụ 6:

```
#include <iostream>
using namespace std;
main()
{   float *p; int n;
    cout << "So luong can cap phat = ";
    cin >> n;
    p = new float[n];
    if (p != NULL)
          cout << "Cap phat thanh cong!";
}
```

5.1.5.3 Toán tử thu hồi vùng nhớ (delete)

Để thu hồi hay giải phóng vùng nhớ đã cấp phát cho một biến (khi không cần sử dụng nữa) ta sử dụng toán tử delete.

Cú pháp:

delete p ;

Với p là con trỏ được cấp phát vùng nhớ bởi toán tử new.

Để giải phóng toàn bộ mảng được cấp phát thông qua con trỏ p ta dùng câu lệnh:

delete[ ] p ;

Với p là con trỏ trỏ đến mảng.

5.1.6 Ví dụ minh họa

Viết chương trình nhập vào dãy số nguyên (không dùng mảng). In ra dãy đã nhập. Sắp xếp dãy tăng dần và in ra dãy kết quả.

Trong ví dụ này chương trình yêu cầu cấp phát bộ nhớ đủ chứa n số nguyên và được trỏ bởi con trỏ head. Khi đó địa chỉ của số nguyên đầu tiên và cuối cùng sẽ là head và

head + n - 1. p và q là 2 con trỏ chạy trên dãy số này, so sánh và đổi nội dung của các số này với nhau để sắp thành dãy tăng dần và cuối cùng in kết quả.

```
#include <iostream>
using namespace std;
main()
```

```
{
     int *head, *p, *q, n, tam;
                         // head trỏ đến (đánh dấu) đầu dãy
     cout << "Cho biet so phan tu cua day: ";
     cin >> n ;
     head = new int[n] ;// cấp phát bộ nhớ chứa n số nguyên
     for (p = head; p < head + n; p++)    // nhập dãy
     {
          cout << "Nhap p.tu thu " << p-head+1 << ": " ;
          cin >> *p ;
     }
     cout << "Day vua nhap la : " << endl;
     for (p = head; p < head + n; p++)
          cout << *p << '\t';
    cout << endl;
     // Sắp xếp dãy tăng dần
    for (p = head; p < head + n - 1; p++)
          for (q = p + 1; q < head + n; q++)
               if (*q < *p)
               {
                    tam = *p; *p = *q; *q = tam;
               }
     cout << "Day sau khi sap xep la : " << endl;
    for (p = head; p < head + n; p++)
        cout << *p << '\t';
    cout << endl;
}
```

5.1.7 Con trỏ và chuỗi ký tự

Một con trỏ ký tự có thể xem như một biến chuỗi ký tự, trong đó chuỗi chính là tất cả

các ký tự kể từ byte con trỏ trỏ đến cho đến byte '\0' gặp đầu tiên. Vì vậy ta có thể khai báo các chuỗi dưới dạng con trỏ ký tự như sau:

char *s ;

char *s = "Hello" ;

Các hàm trên chuỗi vẫn được sử dụng như khi ta khai báo nó dưới dạng mảng ký tự. Bên cạnh đó ta còn được phép sử dụng phép gán cho 2 chuỗi dưới dạng con trỏ.

Ví dụ:

```
char *s, *t = "Cong nghe thong tin";
s = t;    // thay cho hàm strcpy(s, t);
```

Thực chất phép gán trên chỉ là gán 2 con trỏ với nhau, nó cho phép s bây giờ cũng được trỏ đến nơi mà t đang trỏ (tức là dãy ký tự "Cong nghe thong tin" đã bố trí sẵn trong bộ nhớ).

Khi khai báo chuỗi dạng con trỏ nó vẫn chưa có bộ nhớ cụ thể, vì vậy thông thường kèm theo khai báo ta cần phải xin cấp phát bộ nhớ cho chuỗi với độ dài cần thiết.

Ví dụ:

```
#include <iostream>
#include <string.h>
using namespace std;
main()
{   char *s = new char[30], *t;
    strcpy(s, "Hello");
    t = s;
    cout << "Chuoi s la : " << s << endl;
    cout << "Chuoi t la : " << t << endl;
}
```

Nếu không có lệnh cấp phát vùng nhớ cho t thì ta chỉ có thể gán s vào t mà không thể gọi hàm strcpy. Vì vậy để gọi được `strcpy(t, s);` ta cần phải có thêm câu lệnh `t = new char[30];` vào chương trình.

Chương trình trên được viết lại như sau:

```
#include <iostream>
#include <string.h>
```

```
using namespace std;
main()
{   char *s = new char[30], *t;
    strcpy(s, "Hello");
    t = new char[30];
    strcpy(t, s);
    cout << "Chuoi s la : " << s << endl;
    cout << "Chuoi t la : " << t << endl;
}
```

5.2 HÀM

5.2.1 Định nghĩa

Hàm (Function) là một chương trình con trong chương trình lớn. Hàm nhận (hoặc không) các đối số (tham số) và trả lại (hoặc không) một giá trị cho chương trình gọi nó. Trong trường hợp không trả lại giá trị, hàm hoạt động như một thủ tục trong các NNLT khác. Một chương trình là một tập hợp các hàm, trong đó có một hàm chính với tên gọi là main(), khi chạy chương trình, hàm main() sẽ được chạy đầu tiên và gọi đến các hàm khác. Kết thúc hàm main() cũng là kết thúc chương trình.

Hàm giúp cho việc phân đoạn chương trình thành những mô-đun riêng lẻ, hoạt động độc lập với ngữ nghĩa của chương trình lớn, có nghĩa là một hàm có thể được sử dụng trong chương trình này mà cũng có thể được sử dụng trong chương trình khác, dễ dàng cho việc kiểm tra và bảo trì chương trình.

5.2.2 Các đặc trưng

- Nằm trong hoặc ngoài văn bản có chương trình gọi đến hàm, trong một văn bản có thể chứa nhiều hàm.

- Được gọi từ chương trình chính (hàm main()), từ hàm khác hoặc từ chính nó (tính đệ quy).

- Không lồng nhau.

5.2.3 Khai báo

Một hàm thường thực hiện chức năng: tính toán trên các đối số và trả lại kết quả, hoặc chỉ đơn thuần thực hiện một chức năng nào đó, không trả lại kết quả tính toán. Thông thường kiểu của kết quả trả về được gọi là kiểu của hàm. Các hàm thường được khai báo ở đầu chương trình. Các hàm viết sẵn được khai báo trong các tệp nguyên mẫu

(tập tin tiêu đề) *.h.

Do đó, để sử dụng được các hàm này, cần có chỉ thị `#include <*.h>` ở ngay đầu chương trình, trong đó *.h là tên file cụ thể có chứa khai báo của các hàm được sử dụng (chẳng hạn để sử dụng các hàm toán học ta cần khai báo `#include <math.h>`). Đối với các hàm do người lập trình tự viết, cũng cần phải khai báo.

Khai báo một hàm như sau:

<Kiểu giá trị trả về> <Tên hàm> (Danh sách đối số) ;

Trong đó, kiểu giá trị trả về còn gọi là kiểu của hàm và có thể nhận bất kỳ kiểu chuẩn nào của C++ và cả kiểu do người lập trình khai báo. Đặc biệt nếu hàm không trả về giá trị thì kiểu giá trị trả về được khai báo là **void**.

Ví dụ:

```
long LapPhuong(int);
int NgauNhien();
void ChuoiHoa(char[ ]);
int  Cong(int, int);
```

5.2.4 Cấu trúc chung

Cấu trúc chung của một hàm bất kỳ được bố trí cũng giống như hàm main() trong các phần trước.

```
<Kiểu giá trị trả về> <Tên hàm> (Danh sách đối số hình thức)
    {
        Các khai báo cục bộ của hàm ; // chỉ dùng riêng cho hàm này
        Dãy lệnh của hàm ; // các câu lệnh xử lý
        return <Biểu thức trả về> ; // có thể nằm đâu đó trong dãy lệnh.
    }
```

- **Danh sách đối số hình thức** còn được gọi ngắn gọn là danh sách đối số gồm dãy các đối số cách nhau bởi dấu phẩy, đối số có thể là một biến thường, biến tham chiếu hoặc biến con trỏ. Mỗi đối số được khai báo giống như khai báo biến, tức là một cặp gồm **<kiểu đối số> <tên đối số>**.

- Với **hàm có trả lại giá trị** cần có câu lệnh return kèm theo sau là một biểu thức. Kiểu của giá trị biểu thức này chính là kiểu của hàm đã được khai báo ở phần tên hàm. Câu

lệnh return có thể nằm ở vị trí bất kỳ trong phần dãy lệnh của hàm. Khi gặp câu lệnh return chương trình tức khắc thoát khỏi hàm và trả lại giá trị của biểu thức sau return như giá trị của hàm.

- Với **hàm không có trả lại giá trị** (tức kiểu hàm là void) thì không cần có câu lệnh return hoặc có câu lệnh return nhưng phía sau return không có <Biểu thức trả về>.

Ví dụ 1: Ví dụ sau định nghĩa hàm tính lập phương của số nguyên n (với n nguyên) và hàm đổi chuỗi bất kỳ về dạng chuẩn upper.

```
#include <iostream>
#include <string.h>
using namespace std;
long LapPhuong(int);
void ChuoiHoa(char *s);
main()
{   int n;
    cout<<"Nhap so nguyen n = "; cin>>n;
    cout<<"Lap phuong cua "<<n<<" la : ";
    cout<<LapPhuong(n)<<endl;
    cin.ignore(1);
    char *s = new char[100];
    cout<<"Nhap vao chuoi bat ky : ";
    cin.getline(s, 100);
    ChuoiHoa(s);
    cout<<"Chuoi dang upper la : "<<s<<endl;
}
long LapPhuong(int n)
{
    return n*n*n;
}
void ChuoiHoa(char *s)
{   int i;
```

```
    for(i = 0; i < strlen(s); i++)
        s[i] = toupper(s[i]);
}
```

✧ ***Các chú ý***:

- Danh sách đối số trong khai báo hàm có thể chứa hoặc không chứa tên đối số, thông thường ta chỉ khai báo kiểu đối số chứ không cần khai báo tên đối số, trong khi ở hàng đầu tiên của định nghĩa hàm phải có tên đối số đầy đủ.

- Cuối khai báo hàm phải có dấu chấm phẩy (;), trong khi cuối hàng đầu tiên của định nghĩa hàm không có dấu chấm phẩy.

- Hàm có thể không có đối số(danh sách đối số rỗng), tuy nhiên cặp dấu ngoặc sau tên hàm vẫn phải được viết. Chẳng hạn như: NgauNhien(), InTho(), …

- Một hàm có thể không cần phải khai báo nếu nó được định nghĩa trước khi có hàm nào đó gọi đến nó.

5.2.5 Lời gọi hàm

Lời gọi hàm được phép xuất hiện trong bất kỳ biểu thức, câu lệnh của hàm khác … Nếu lời gọi hàm lại nằm trong chính bản thân hàm đó thì ta gọi là đệ quy. Để gọi hàm ta chỉ cần viết tên hàm và danh sách các giá trị cụ thể truyền cho các đối số đặt trong cặp dấu ngoặc ( ).

Tên hàm(danh sách đối số thực tế);

- Danh sách đối số (tham số) thực tế còn gọi là danh sách giá trị gồm các giá trị cụ thể để gán lần lượt cho các đối số hình thức của hàm. Khi hàm được gọi thực hiện thì tất cả những vị trí xuất hiện của đối số hình thức sẽ được gán cho giá trị cụ thể của đối số thực tế tương ứng trong danh sách, sau đó hàm tiến hành thực hiện các câu lệnh của hàm (để tính kết quả).

- Danh sách đối số thực tế truyền cho đối số hình thức có số lượng bằng với số lượng đối số trong hàm và được truyền cho đối số theo thứ tự tương ứng. Các đối số thực tế có thể là các hằng, các biến hoặc biểu thức. Biến trong giá trị có thể trùng với tên đối số. Ví dụ ta có hàm in n lần ký tự c với khai báo `void inkitu(int n, char c);` và lời gọi hàm `inkitu(12, 'A');` thì n và c là các đối số hình thức, 12 và 'A' là các đối số thực tế hoặc giá trị. Các đối số hình thức n và c sẽ lần lượt được gán bằng các giá trị tương ứng là 12 và 'A' trước khi tiến hành các câu lệnh trong phần thân hàm. Giả sử hàm in ký tự được khai báo lại thành `inkitu(char c, int n);` thì lời gọi hàm cũng phải được thay lại thành `inkitu('A', 12);`

- Các giá trị tương ứng được truyền cho đối số phải có kiểu cùng với kiểu đối số (hoặc

C++ có thể tự động chuyển về kiểu của đối số).

- Khi một hàm được gọi, nơi gọi tạm thời chuyển điều khiển đến thực hiện câu lệnh đầu tiên trong hàm được gọi. Sau khi kết thúc thực hiện hàm, điều khiển lại được trả về thực hiện tiếp câu lệnh sau lệnh gọi hàm của nơi gọi.

Ví dụ 2: Giả sử ta cần tính giá trị của hàm $f = 2x^3 - 5x^2 - 4x + 1$, thay cho việc tính trực tiếp x^3 và x^2, ta có thể gọi hàm LuyThua() trong ví dụ trên để tính các giá trị này bằng cách gọi nó trong hàm main() như sau:

```
#include <iostream>
using namespace std;
double LuyThua(float x, int n)
{   int i;
    double kq = 1;
    for (i = 1; i <= n; i++)
        kq *= x;
    return kq;
}
main()
{   float x;
    double
    f;
    cout << "Nhap x = ";
    cin >> x;
    f = 2*LuyThua(x,3) - 5*LuyThua(x,2) - 4*x + 1;
    cout << "Gia tri cua ham f = " << f << endl;
}
```

5.2.6 Hàm với đối số mặc định

Trong phần trước chúng ta đã khẳng định số lượng đối số thực tế phải bằng số lượng đối số hình thức khi gọi hàm. Tuy nhiên, trong thực tế rất nhiều lần hàm được gọi với các giá trị của một số đối số hình thức được lặp đi lặp lại. Trong trường hợp như vậy lúc nào cũng phải viết một danh sách dài các đối số thực tế giống nhau cho mỗi lần gọi là một công việc không mấy thú vị. Từ thực tế đó C++ đưa ra một cú pháp mới về hàm sao cho một danh sách đối số thực tế trong lời gọi không nhất thiết phải viết đầy đủ nếu một

số trong chúng đã có sẵn những giá trị định trước. Cú pháp này được gọi là hàm với đối số mặc định và được khai báo với cú pháp như sau:

<Kiểu hàm> <Tên hàm>(đs1, …, đsn, đsmđ1 = gt1, …, đsmđm = gtm) ;

- Các đối số đs1, …, đsn và các đối số mặc định đsmđ1, …, đsmđm đều được khai báo như cũ nghĩa là gồm có kiểu đối số và tên đối số.

- Riêng các đối số mặc định đsmđ1, …, đsmđm có gán thêm các giá trị mặc định gt1, …, gtm. Một lời gọi bất kỳ khi gọi đến hàm này đều phải có đầy đủ các đối số thực tế ứng với các đs1, …, đsn nhưng có thể có hoặc không các đối số thực tế ứng với các đối số mặc định đsmđ1, …, đsmđm. Nếu đối số nào không có đối số thực tế thì nó sẽ được tự động gán giá trị mặc định đã khai báo.

Ví dụ: Xét hàm `double LuyThua(float x, int n = 2);` Hàm này có một đối số mặc định là số mũ n, nếu lời gọi hàm bỏ qua số mũ này thì chương trình hiểu là tính bình phương của x (n = 2). Chẳng hạn lời gọi `LuyThua(4, 3)` được hiểu là tính 4^3, lời gọi `LuyThua(4, 2)` hay `LuyThua(4)` được hiểu là tính 4^2

✧ ***Lưu ý***: Các đối số mặc định phải nằm bên phải các đối số không mặc định.

5.2.7 Khai báo hàm trùng tên (Quá tải hàm)

Hàm trùng tên còn gọi là hàm chồng (đè) hay quá tải hàm (function overload). Đây là một kỹ thuật cho phép sử dụng cùng một tên gọi cho các hàm "***giống nhau***" (cùng mục đích) nhưng xử lý trên các kiểu dữ liệu khác nhau hoặc trên số lượng dữ liệu khác nhau.

Ví dụ:

Hàm sau tìm số lớn hơn trong 2 số nguyên:

```
int Max(int a, int b)
{
    return (a > b) ? a: b;
}
```

Chúng ta có thể định nghĩa chồng hàm trên để có thể tìm số lớn hơn trong 2 số thực:

```
float Max(float a, float b)
{
    return (a > b) ? a: b;
}
```

Khi đó tùy theo giá trị nhận vào của các đối số a và b thuộc kiểu int hay float mà phiên bản của hàm Max nào sẽ được chọn để thực hiện.

5.2.8 Các cách truyền đối số

Có 3 cách truyền đối số thực tế cho các đối số hình thức trong lời gọi hàm. Trong đó cách ta đã dùng cho đến thời điểm hiện nay được gọi là truyền bằng giá trị, còn được gọi là *tham trị* hay *truyền bằng giá trị*, tức các đối số hình thức sẽ nhận các giá trị cụ thể từ lời gọi hàm và tiến hành tính toán rồi trả lại giá trị.

5.2.8.1 Truyền bằng giá trị

Ta xét lại ví dụ hàm `LuyThua(float x, int n)` dùng để tính tính x^n. Giả sử trong chương trình chính (hàm main()) ta có các biến a, b, f đang chứa các giá trị a = 2, b = 3, và f chưa có giá trị. Để tính a^b và gán giá trị trả về vào f, ta có thể gọi `f = LuyThua(a,b);`

Khi gặp lời gọi này, chương trình sẽ tổ chức như sau:

- Tạo 2 biến mới (tức 2 ô nhớ trong bộ nhớ) có tên x và n. Gán nội dung các ô nhớ này bằng các giá trị trong lời gọi, tức gán 2 (a) cho x và 3 (b) cho n.

- Tới phần khai báo (của hàm), chương trình tạo thêm các ô nhớ mang tên là i và kq.

- Tiến hành tính toán (gán lại kết quả cho kq).

- Cuối cùng lấy kết quả trong kq gán cho ô nhớ f (là ô nhớ có sẵn đã được khai báo trước, nằm bên ngoài hàm).

- Kết thúc hàm quay về chương trình gọi. Do hàm LuyThua đã hoàn thành xong việc tính toán nên các ô nhớ được tạo ra trong khi thực hiện hàm (x, n, i, kq) sẽ được xóa khỏi bộ nhớ. Kết quả tính toán được lưu giữ trong ô nhớ f (không bị xóa vì không liên quan gì đến hàm).

Truyền đối số theo giá trị là cách truyền phổ biến. Tuy nhiên vấn đề đặt ra là giả sử ngoài việc tính f, ta còn muốn thay đổi các giá trị của các ô nhớ a, b (khi truyền nó cho hàm) thì có thể thực hiện được không? Để giải quyết vấn đề này ta cần phải truyền đối số bằng địa chỉ hoặc con trỏ. Hai cách truyền này có thể được gọi chung là *tham biến*.

5.2.8.2 Truyền bằng con trỏ

Xét ví dụ hoán đổi giá trị của 2 biến. Đây là một yêu cầu nhỏ nhưng được gặp nhiều lần trong chương trình, ví dụ để sắp xếp một mảng hay danh sách. Do vậy cần viết một hàm để thực hiện yêu cầu trên. Hàm không trả kết quả mà chỉ hoán đổi giá trị giữa 2 đối số. Do các biến cần hoán đổi là chưa được biết trước tại thời điểm viết hàm, nên ta phải đưa chúng vào hàm như các đối số, tức là hàm có 2 đối số (có thể đặt là x, y) đại diện cho các biến sẽ thay đổi giá trị sau này.

Từ các nhận xét trên, nếu hàm hoán đổi ngay sau đây sẽ không đáp ứng được yêu cầu.

```
void Swap(int x, int y)
```

```
{
    int t = x;
    x = y; y = t;
}
```

Hàm Swap trên không đáp ứng được yêu cầu vì giả sử trong chương trình chính ta có 2 biến x, y chứa các giá trị lần lượt là 2 và 5. Ta cần hoán đổi nội dung 2 biến này sao cho x = 5 còn y = 2 bằng cách gọi đến hàm Swap(x, y).

```
main()
{   int x = 2, y = 5;
    Swap(x, y);
    cout << "x = " << x << "va y = " << y << endl;
            // in ra x = 2 va y = 5 (x, y vẫn không đổi)
}
```

Thực tế sau khi chạy xong chương trình ta thấy giá trị của x và y vẫn không thay đổi !?. Như đã giải thích trong mục trên (gọi hàm LuyThua), việc đầu tiên khi chương trình thực hiện một hàm là tạo ra các biến mới (các ô nhớ mới, độc lập với các ô nhớ x, y đã có sẵn) tương ứng với các đối số, trong trường hợp này cũng có tên là x, y và gán nội dung của x, y (ngoài hàm) cho x, y (mới). Và việc cuối cùng của chương trình sau khi thực hiện xong hàm là xóa các biến mới này. Do vậy nội dung của các biến mới thực tế là có thay đổi, nhưng không ảnh hưởng gì đến các biến x, y cũ (ngoài hàm).

Như vậy ***hàm Swap cần được viết lại*** sao cho việc thay đổi giá trị không thực hiện trên các biến tạm mà phải thực hiện trên các biến ngoài. Muốn vậy thay vì truyền giá trị của các biến ngoài cho đối số, bây giờ ta sẽ truyền địa chỉ của nó cho đối số, và các thay đổi sẽ phải thực hiện trên nội dung của các địa chỉ này. Đó chính là lý do ta phải sử dụng con trỏ để làm đối số thay cho biến thường.

Cụ thể hàm Swap được viết lại như sau:

```
void Swap(int *p, int *q)
{
    int t = *p;
    *p = *q; *q = t;
}
```

Với cách tổ chức hàm như vậy rõ ràng nếu ta cho p trỏ tới biến x và q trỏ tới biến y thì hàm Swap sẽ làm thay đổi nội dung của x, y chứ không phải của p, q.

Từ đó lời gọi hàm sẽ là Swap(&x, &y) (tức truyền địa chỉ của x cho p để p trỏ tới x và tương tự q trỏ tới y).

5.2.8.3 Truyền bằng địa chỉ

Một hàm viết dưới dạng đối số được truyền bằng địa chỉ sẽ đơn giản hơn so với truyền bằng con trỏ và nó giống với cách truyền bằng giá trị hơn, trong đó chỉ có một điểm khác biệt đó là các đối số khai báo theo dạng địa chỉ.

Như vậy ***hàm Swap trên cũng có thể được viết*** theo cách truyền địa chỉ trực tiếp mà không thông qua con trỏ như sau:

```
void Swap(int &x, int &y)
{
    int t = x;
    x = y; y = t;
}
```

Và lời gọi hàm cũng đơn giản như cách truyền đối số bằng giá trị.

```
main()
{   int x = 2, y = 5;
    Swap(x, y);
    cout << "x = " << x << "va y = " << y << endl;
            // in ra x = 5 va y = 2 (x, y đã không đổi)
}
```