Pointer arithmetic:

type X [10][5][10];

d = { "bool": 1, "char": 1, "int": 4, "float": 4, "double": 8 };

X:

T=type(\*)[5][10],

sizeof(T) = 5\*10,

I

X+k (X[k]):

T=type(\*)[5][10],

sizeof(T) = 5\*10\*d[type],

I+k(5\*10)\*d[type]

\*(X+k)+j (X[k][j]):

T=type(\*)[10],

sizeof(T) = 10\*d[type],

I+k(5\*10)\*d[type]+j(10)\*d[type]

\*(\*(X+k)+j)+i (X[k][j][i]):

T=type(\*),

sizeof(T) = d[type],

I+k(5\*10)\*d[type]+j(10)\*d[type]+i\*d[type]

int ex\_1(){

char X[10][5][10];

cout << (int) X << " " << typeid(X).name() << " char (\*)[5][10]\n";

cout << (int) (\*(\*X-4)+2) << " == " << (((int) X)-4\*10\*1+2\*1)

<< " " << typeid((\*(\*X-4)+2)).name() << " char (\*)\n";

cout << (int) X[-1] << " == " << (((int) X)-1\*5\*10\*1)

<< " " << typeid(X[-1]).name() << " char (\*)[10]\n";

// both are undefined

}

Javascript solving algorithm:

console.log(parse("X+3=", "char", [10,5,10]));

console.log(parse("\*(X+3)+4=", "char", [10,5,10]));

console.log(parse("\*(\*(X+3)+4)+5=", "char", [10,5,10]));

console.log(parse("\*(\*(X)-4)+2=", "char", [10,5,10]));

function parse(string, type, array){

var d = { "bool": 1, "char": 1, "int": 4, "float": 4, "double": 8 };

var ref = string.indexOf("\*(");

array.shift();

var re = (ref!=-1) ?

/\\*\(X([\+\-]{0,1}[0-9]\*)\)/gi :

/X([\+\-][0-9]\*)/gi;

while (match = re.exec(string))

string += [match[1] || '+0', array.join('\*'), d[type]].join('\*');

return (ref!=-1) ?

parse(string.replace(re, 'X'), type, array) :

string .replace(re, 'I')

.replace(/X/gi, 'I')

.replace('=', '');

}

Reverse engineering:

Find out errors, read the code.

Errors:

1) int & f(int a){ return a; } // a is local

2) int & f(){ int b; return b; } // b is local

3) int & f(int & a){ return a+1; } // a+1 is const, can't be referenced

4) int \* F(int \*a){ int b; a=&b; return a; } // b is local

int \*p

1) \*p; // segmentation fault

int a=6, \*p = &a;

1) p = 6; // p is int \*, 6 is int

2) p = a-6; // 0 is legal but compiler doesn't know

p = 0;

1) \*p = a // segmentation fault

int \*f(int \*\*p){

int b=3,\*x=&b;

\*\*p=b;

x=\*p;

return x;

}

int ex\_2(){

int y=5, b=2,\*q=&b;

\*f(&q)=y\*2;

cout << b << "\n";

/\* G -> global (main), L -> local (f)

L-value Name R-value

10 Gy 5

11 Gb 2

12 Gq 11

13 Lp 12

14 Lb 3

15 Lx 14

\*\*p = b

Gb = Lb = 3

x = \*p

Lx = &q

\*f(&q) = y\*2

\*q = y\*2 = 10

\*/

}

Iterative functions:

Write the code, write PRE, POST and R

void M(int(\*A)[20], int \* Inizio, int limite, int &r, int &k)

{

/\* PRE: A Ã¨ un array A[limite][20] con limite\*20 elementi definiti,

\* Inizio Ã¨ un array Inizio[limite] con limite elementi definiti,

\* limite>0

\*/

/\* R: min Ã¨ l'elemento con il numero minore trovato in A[0..i-1],

\* r e k sono i suoi indici

\*/

int min = INT\_MAX;

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

{

if (Inizio[i]<20 && A[i][Inizio[i]]<min){

min = A[i][Inizio[i]];

r = i;

k = Inizio[i];

}

}

Inizio[r]++;

/\* POST: min Ã¨ l'elemento con il numero minore trovato in A,

\* r e k sono i suoi indici

\* Inizio[r] viene incrementato di 1

\*

\*/

}

int \* F(int(\*A)[20],int limite)

{

int \*R=new int[limite\*20], \*Inizio=new int[limite], r=0,k=0;

for(int i=0; i<limite; i++) Inizio[i]=0;

for(int i=0; i<limite\*20; i++)

{

M(A,Inizio,limite,r,k);

R[i]=A[r][k];

}

delete[] Inizio;

return R;

}

int main(){

int A[5][20];

int limite = 5;

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

{

for (int k=0; k<20; k++)

{

A[i][k] = (k+1)\*10 + (i+1);

cout << A[i][k] << " ";

}

cout << "\n";

}

cout << "\n";

int \* R = F(A, 5);

for (int i=0; i<limite\*20; i++)

cout << \*(R + i) << (((i+1)%20) ? " " : " \n");

cout << "\n";

}

struct nodo{int info; nodo\* next;};

void append(nodo \*& L, int value)

{

if (L)

{

append(L->next, value);

}else{

L = new nodo;

L->info = value;

}

}

void append(nodo \*& L, nodo \* value)

{

if (L)

append(L->next, value);

else

L = value;

}

int count(nodo \*L)

{

if (L)

return 1+count(L->next);

else

return 0;

}

void print(nodo \* L)

{

if (L)

{

cout << L->info << ((L->next) ? "->" : "");

print(L->next);

}else{

cout << "\n";

}

}

void estrae(nodo\*&L, nodo\*N)

{

/\*

\* PRE: L Ã¨ una lista valida != 0,

\* N Ã¨ un nodo di L

\*/

nodo \*\* myL = &L;

/\*

\* R: N non Ã¨ uno degli elementi precedenti a myL

\* Condizione d'uscita: myL Ã¨ N

\*/

while (N!=\*myL)

myL = &((\*myL)->next);

\*myL = (\*myL)->next;

/\*

\* POST: L non contiene piÃ¹ l'elemento N

\*/

}

nodo\* MI(nodo\*&L, int\*P, int dim\_P)

{

nodo\*\* M=new nodo\*[dim\_P];

int n=0;

nodo\* origin=L;

while(origin && n<dim\_P)

{

if(origin->info==P[n])

{

M[n]=origin;

n++;

}

origin=origin->next;

}

nodo\* E= 0,\*fine;

if(n==dim\_P)

{

E=fine=M[0];

estrae(L,M[0]);

for(int i=1; i<dim\_P;i++)

{

estrae(L,M[i]);

fine->next=M[i];

fine=M[i];

}

fine->next=0;

}

delete[] M;

return E;

}

int main()

{

nodo \* L = 0;

append(L, 1);

append(L, 2);

append(L, 3);

append(L, 1);

append(L, 2);

int P[] = {1, 1};

int dim\_P = 2;

print(MI(L, P, dim\_P));

print(L);

}

Recursive functions:

Write the code, write PRE, POST and demostrate by induction

Esercizio cancellato perchÃ¨ considerato da FilÃ¨ â€œeccessivamente complicatoâ€

/\* Dimostrazione:

\* Casi base:

\* 1) A contiene una lista vuota e non ha elementi successivi:

\* viene ritornato 0 (rispettando la POST)

\* 2) A contiene una lista vuota:

\* la ricerca prosegue sull'elemento successivo;

\* abbiamo escluso il caso in cui A->next non sia valido quindi

\* la PRE Ã¨ rispettata.

\* 3) A non ha elementi successivi:

\* A Ã¨ l'unico elemento e viene ritornato A (rispettando la POST)

\*

\* In tutti gli altri casi:

\* Se A ha il primo numero minore del corrispettivo in A->next

\* A e A->next vengono scambiati in modo che A->next sia sempre

\* la lista con il numero minore. Anche nel caso in cui A->next

\* contenga una lista vuota, i due numeri vengono scambiati:

\* A->next, quindi, contiene sempre una lista non vuota.

\*

\* In seguito viene chiamata la funzione ricorsivamente passando

\* come parametro A->next. La PRE Ã¨ rispettata in quanto sia A che

\* A->next sono liste di nodoP valide e diverse da zero.

\*

\* Il caso piÃ¹ semplice, quello in cui A->next ha solo un elemento

\* successivo, puÃ² essere ricondotto al caso base 3: la funzione

\* ricorsiva si limiterÃ  a ritornare A->next stesso.

\* La funzione ritornerÃ  quindi A->next che, come abbiamo giÃ  visto

\* Ã¨ minore in ogni caso ad A. La POST Ã¨ rispettata.

\*

\* Ogni caso piÃ¹ complesso, con un diverso numero elementi

\* successivi, puÃ² essere ricondotto al caso base 3.

\*/

void H(nodoP\* A, nodo\*& R)

{

nodoP\* x=G(A);

if(x)

{

nodo\*y=x->info;

x->info=y->next;

R=y;

H(A,R->next);

}

else

R=0;

}

int main(){

nodoP \* A = 0;

for (int i=0; i<5; i++)

{

nodo \* B = 0;

for (int k=0; k<20; k++)

{

append(B, (i+1)+(k+1)\*10);

}

print(B);

append(A, B);

}

nodo \* R = 0;

H(A, R);

print(R);

}

struct nodo{char info; nodo\* next;};

void append(nodo \*& L, char value)

{

if (L)

{

append(L->next, value);

}else{

L = new nodo;

L->info = value;

}

}

void append(nodo \*& L, nodo \* value)

{

if (L)

append(L->next, value);

else

L = value;

}

int count(nodo \*L)

{

if (L)

return 1+count(L->next);

else

return 0;

}

void print(nodo \* L)

{

if (L)

{

cout << L->info << ((L->next) ? "->" : "");

print(L->next);

}else{

cout << "\n";

}

}

nodo\* M(nodo\*&L, char\*P, int dim\_P, bool &ok){

/\*PRE: L Ã¨ una lista valida !=0

\* P Ã¨ un array che contiene dim\_P elementi validi

\* dim\_P>0

\*/

nodo \* K = 0;

if (L->info == \*P)

{

append(K, L->info);

if (ok)

L = L->next;

if (dim\_P!=1 && L->next)

append(K, M(L->next, P+1, dim\_P-1, ok));

}else{

if (L->next)

append(K, M(L->next, P, dim\_P, ok));

}

if (!ok && count(K)==dim\_P)

{

ok = !ok;

M(L, P, dim\_P, ok);

ok = !ok;

return K;

}else

return 0;

/\* POST: L non contiene gli elementi verificati dal pattern (non

\* necessariamente contiguo) P, K contiene gli elementi verificati.

\*/

}

/\* Dimostrazione:

\* Casi base:

\* 1) dim\_P = 1 e avviene un match: K contiene soltanto l'elemento

\* corrente che viene ritornato dalla funzione. Subito prima del return

\* viene chiamata la funzione M con gli stessi L, P e dim\_P della

\* funzione corrente (la PRE Ã¨ rispettata) con il parametro ok true.

\* Quando ok Ã¨ true la funzione si limita ad eliminare i valori presenti

\* P. In questo caso provvede ad eliminare l'unico nodo contenuto in K.

\* Anche La POST Ã¨ quindi rispettata.

\*

\* 2) L->next == 0: non ci sono elementi successivi nella lista; il

\* controllo termina. Se non Ã¨ avvenuto un match count(K) non potrÃ  mai

\* essere uguale a dim\_P. La funzione ritornerÃ  zero, rispettando la

\* POST.

\*

\* In tutti gli altri casi:

\* La funzione puÃ² essere ricondotta sempre al caso 1 o al caso 2,

\* rispettivamente quando la funzione verifica l'ultimo elemento del

\* pattern e quando la funzione termina di controllare L senza aver

\* trovato tutti i matches necessari.

\* La funzione infatti si invoca ricorsivamente modificando P e dim\_P

\* in base al match del valore corrente (andando quindi a controllare

\* l'elemento successivo del pattern nell'invocazinoe ricorsiva) e

\* controllando che il match non sia l'ultimo match richiesto.

\* I match presenti nei nodi successivi vengono aggiunti a K.

\* Prima del termine di esecuzione della funzione viene confrontato

\* il numero di elementi contenuti in K e il numero di elementi del

\* pattern; se il pattern Ã¨ stato verificato viene invocata M con

\* ok==true e i valori verificati dal pattern vengono eliminati.

\*/

int main()

{

nodo \* L = 0;

append(L, 'a');

append(L, 'b');

append(L, 'c');

append(L, 'a');

append(L, 'b');

char P[] = {'a', 'd', 'a'};

int dim\_P = 2;

bool ok = false;

print(M(L, P, dim\_P, ok));

print(L);

}