Valerio Ceraudo

1)

1a) sia i che j mancano il tipo. unsigned int sarebbe l'ideale essendo usati come contatori sempre positivi.

1b) aggiungerei un controllo che i parametri della funzione non siano NULL e che le dimensioni siano maggiori di 0.

1c) dest[i + j \* N + y + x\*N] = src[i \* H + j]; altrimenti non tiene conto dell’offset iniziale. Inoltre l’algoritmo così costruito non tronca correttamente l’immagine di src in base ai confini di dst, ma la parte fuori schermo a destra, viene ricopiata, una riga più giù a sinistra.

1d) non tiene conto degli offset negativi e di quelli fuori schermo. La correzione che tiene conto anche degli offset è nella risposta del secondo esercizio.

1e)Non è un bug ma scorrere le matrici K\*H abbassa notevolmente le prestazioni. Si dovrebbe scorrere la matrice H\*K.

2)

Vengono calcolati a priori i limiti delle texture da blittare in maniera tale da avere i cicli for con meno interruzioni possibili e nessun if di controllo all’interno.

Inoltre anzichè scrivere per colonne scrivo per righe così da ottimizzare gli accessi in memoria dal momento che un array è salvato in celle di memorie contigue. Scorrendo per colonne invece è probabile che si verifichino page fault.

void BitBlt(unsigned char \*dest, unsigned int N, unsigned int M, unsigned char \*src, unsigned int H, unsigned int K, int x, int y){

//potrebbe essere un assert

if( src != NULL && dest != NULL && N>0 && M>0 && H >0 && K >0){

unsigned int leftLimit = 0;

if (y < 0){

leftLimit = -y;

}

unsigned int rightLimit = H;

if (H + y > N){

rightLimit = N - y;

}

unsigned int upperLimit = 0;

if (x < 0){

upperLimit = -x;

}

unsigned int lowerLimit = K;

if (K + x > M){

lowerLimit = M - x;

}

for (unsigned int i = upperLimit; i < lowerLimit; i++){

for (unsigned int j = leftLimit; j < rightLimit; j++){

dest[x\*N + i\*N + y +j] = src[i\*K + j];

}

}

}

}

3)

int pow(int n, int e) è una normale funzione che riceve come parametri due interi e restituisce un intero, in questo caso n elevato ad e.

In questa maniera la funzione è però limitata ad utilizzare solo tipi interi mentre con #define pow(n, e) n^e si crea una direttiva che indica al processore di sostituire ad ogni pow(n,e) n^e a prescindere dal tipo di n ed e (ipotizando che sia valido e corretto scrivere n^e),

Il problema della define in questo caso è che sostituirebbe in tutto il codice la parola pow, ad esempio non sarebbe più utilizzabile la funzione pow di math.h poichè colliderebbe.

Se la funzione da una parte ha un costo aggiuntivo in termini di overhead di spazio necessario all'allocazione della chiamata della funzione stessa, la define sostituisce a tempo di compilazione n^e ovunque sia scritto pow(n,e). Fatto comunque parzialmente vero dato che l'implementazione stessa potrebbe a sua volta richiamare altre funzioni che implicherebbero comunque un overhead ( ipotesi: la più semplice implementazione di una potenza sarebbe moltiplicare n per se stesso e volte, dunque si chiamerebbe l'operatore \*).

Vi è inoltre da considerare la poca mantenibilità del codice in debugging dato che non sarebbe possibile trovare la chiamata a funzione, quanto pure la scrittura stessa delle operazioni richiederebbe attenzione nell'uso delle parentesi al fine di evitare effetti collaterali della sostituzione. Inoltre le define non prevedono ricorsione e vanno scritte tutte sulla stessa riga.

Infine mentre una const o una funzione inline sono file scope, le #define rompono il concetto di scope e rimangono definite come tali in tutta la compilazione fino a che non vengono #undef.

4)

La define viene chiamata con n=4 e sostituisce il valore ++n.

Ipotizzando che la sostituzione sia ++n\*++n, perchè e=2, avvengono prima le operazioni prefisse di incremento, quindi n non vale più 4 ma 6, ed il risultato di pow(n,e) è 36 e non 25 come ci si aspetterebbe.

5)

1342 -> costruttore A, costruttore B, distruttore B, distruttore A

6)

1. Car(float speed, int kw);
2. Assegnamento dell’indirizzo di car1 al puntatore pCar1;
3. Car(Car &car);
4. Car(Car \*car);
5. Assegnamento dell’indirizzo di pCar1 al puntatore \*pCar3;
6. Car(Car &car);
7. Assegnamento dell’indirizzo di pCar3 all’indirizzo a pCar1;
8. Car &operator=(const Car &car); //not assign because same objs
9. virtual ~Car();
10. virtual ~Car();
11. virtual ~Car();
12. memory leak dell’oggetto new del quale non si fà la delete.

7)

MIA MODIFICATA

class Faa; //non necessito l’include ma solo la forward declaration

class Fii {

public:

Fii();

//aggiungere costruttore di copia e operatore =

Fii(const Fii &fii);

Fii& operator=(const Fii &fii); //aggiunto costruttore di copia e operator= per la regola dei 3.

virtual ~Fii(); //aggiunto il virtual

private:

float \*vec;

};

class Foo : public Fii {

public:

Foo();

~Foo();

Foo(const Foo&foo);

Foo& operator=(const Foo&foo); //aggiunto costruttore di copia e operator= per la regola dei 3.

void Data(const Faa \*f); //potrebbe essere usato per valorizzare foo, ma è const per non modificarlo.

int FastSum(int l, int r) const; //aggiunto il const perchè la funzione non dovrebbe modificare i membri interni

//int Sum(int l, int r); Ridondante dato che ho fast sum

inline int GetNumber() const; //aggiunto inline essendo probabilmente un semplice return ed il const finale dato che non modifica le variabili membro.

const Fii\* GetObject() const; //aggiunto il const perchè la funzione non dovrebbe modificare i membri interni, aggiunto \* essendo probabilmente riferito a Fii \*obj;

NOTA: Ha senso ritornare un oggetto const che non posso modificare e che non ha valori public?

friend Foo operator+(int a,const Foo &f); //aggiunto const a Foo perchè se in linea con l’uso delle friend, torna un nuovo Foo

Foo &operator+=(const Foo &f);

private:

int num;

Fii \*obj;

};

8)

template<typename T, typename U>class C{

//

//primary template

//

};

template<> class C<int, double>{

//

//full spec case <int,double>

//

};

I template specializzati possono essere scritti solo nello stesso namespace del template non specializzato e solo dopo esso. La specializzazione deve essere comunque dichiarata prima del suo utilizzo

La specializzazione si fà specificando particolari tipi al posto di tutti gli argomenti del template (in questo caso T = int, U = double) e lo si fà per specializzare particolari casi del template. Senza la specializzazione, il codice del template è generato per ogni tipo usato nell’istanziamento del template. Con la specificazione viene invece chiamato lo specifico template e solo quello primario se non vi è nessuna specializzazione disponibile.

9)

template<typename T, typename U>class C{

//

//primary template

//

};

template<typename T> class C<T, T\*>{

//

//partial spec case <T,T\*>

//

};

E’ possibile specificare solo alcuni argomenti del template, perchè magari ridondanti solo in una comune parte da applicare però a vari tipi, ad esempio:

template<typename T> class C<T, T\*>{};

specifica che qualsiasi sia il tipo passato, il secondo parametro sarà il suo puntatore: utile ad esempio per metodi che devono lavorare su strutture come i sort.

10)

a - 10

b - 8

c - 3

d - 2

e - 1

f - 9

g - 6

h - 7

i - 5

j - 4

k - 1

l - 3

11)

Il metodo UnknowFunc(int n) ritorna il numero di cifre che compongono n:  
n = 1, k = 1;

n = 10, K = 2;

n = 100, K = 3;

n = -100, K = 3;

12)

4 4 16

3 2 8

2 2 4

1 0 2

13)

//assuming char [5] { ‘C’,’i’,’a’,’o’,’\0’ };

reverse(char \*str,int length){

//one remove the endline, one to have array length - 1

length = length-2;

//count from length to 0

for (int i=0; i<(length+i)/2; length--,i++){

char c = str[length];

str[length] = str[(int)i];

str[(int)i] = c;

}

}

14)

//assuming char [5] { ‘C’,’i’,’a’,’o’,’\0’ };

reverse(char \*str,int length){

//To remove endline and move length on last char index

length = length - 2;

//Inversion

while (length > 0){

\*str = \*str - str[length];

str[length] = str[length] + \*str;

\*str = str[length] - \*str;

str++;

length = length-2;

}

}

15)

int factorial(int n){

//non si può fare il fattoriale di un negativo

if(n<0){

return 0;

}

int result = 1;

while (n > 1){

result \*= n;

n--;

}

return result;

}

16)

class Leaf{

public:

int value;

Leaf(int newValue){

value = newValue;

}

~Leaf();{

}

Leaf \*left;

Leaf \*right;

};

//leaf is the tree’s root

InOrder(Leaf\* leaf){

//initialize and set the root as visited

vector<Leaf\*> visited;

visited.push\_back(leaf);

//store nodes still not visited but discovered

vector<Leaf\*> toVisit;

toVisit.push\_back(leaf);

//store the actual root

Leaf\* root = leaf;

//temp store of root's leafes

vector<Leaf\*> leaves;

while (toVisit.size() > 0){

//get the first leaf to visit and remove it from the list

root = \*toVisit.begin();

toVisit.erase(toVisit.begin());

//search for leaves if exist and insert them iat the beginning of the list of leaves to visit

if (root->left != NULL){

leaves.push\_back(root->left);

}

if (root->right != NULL){

leaves.push\_back(root->right);

}

toVisit.insert(toVisit.begin(), leaves.begin(), leaves.end());

//clean leaves vector for next round

leaves.resize(0);

//add discovered leaves in visited list putting left before root and right after.

visited = InsertBeforeAndAfterPos(root, root->left, root->right, visited);

}

/\*il secondo metodo cerca la root ed inserisce i figli in ordine. Questa parte è una funzione a se stante semplicemente per leggibilità, ma aggiunge l’overhead in stack della chiamata. Spostando e sistemando il codice al posto della funzione si eliminerebbe l’overhead \*/

InsertBeforeAndAfterPos(Leaf\* root, Leaf\* left, Leaf\* right, vector<Leaf\*> visited){

//search for root element in visited

vector<Leaf\*>::iterator visitedIt;

for (visitedIt = visited.begin(); visitedIt != visited.end(); ++visitedIt){

if (\*visitedIt == root){

//if left exist add directly

if (left != NULL){

visitedIt = visited.insert(visitedIt, left);

visitedIt = visitedIt +1;

}

//if right exist increase iterator to point correct location and add

if (right != NULL){

visitedIt = visitedIt +1;

visitedIt = visited.insert(visitedIt, right);

}

//can break for, no more result expected

break;

}

}

return visited;

}

17)

Usando la stessa classe leaf dell’esercizio 16.

void CleanInOrder(Leaf\* startingRoot){

//store nodes still not visited but discovered

vector<Leaf\*> toVisit;

toVisit.push\_back(startingRoot);

//store the actual root

Leaf\* root = startingRoot;

//store last selected root

Leaf\* lastSelected = startingRoot;

//temp store of root's leafes

vector<Leaf\*> leaves;

//used to store if lastSelected has at least one child

bool oneChildLeast;

while (toVisit.size() > 0){

oneChildLeast = false;

//search for children if exist and insert them in the beginning of the list of leaves to visit

if (lastSelected->left != NULL){

leaves.push\_back(lastSelected->left);

oneChildLeast = true;

}

if (lastSelected->right != NULL){

leaves.push\_back(lastSelected->right);

oneChildLeast = true;

}

if (oneChildLeast){

//the root has at last a child, it becomes the new root to start search again if it has child

root = \*toVisit.begin();

//the child(children) is(are) added to toVisit list: left+right+oldList

toVisit.insert(toVisit.begin(), leaves.begin(), leaves.end());

//starting to search again from the first element of toVisit list

lastSelected = \*toVisit.begin();

}

else{

//last selected hasn't child, can search in old root to remove reference pointing

if (root->left!=NULL && root->left == lastSelected){

//delete the leaf and the reference from root to it

delete(root->left);

root->left = NULL;

//remove the lastSelected from toVisit

toVisit.erase(toVisit.begin());

//at the beginning of toVisit there is now or the right child or the root

lastSelected = \*toVisit.begin();

}

if (root->right != NULL && root->right == lastSelected){

//delete the leaf and the reference from root to it

delete(root->right);

root->right = NULL;

//remove the lastSelected from toVisit

toVisit.erase(toVisit.begin());

//at the beginning of toVisit there is now the root

lastSelected = \*toVisit.begin();

}

while (toVisit.size() > 0 && root->left == NULL && root->right == NULL){

//root now hasn't children, can be removed from toVisit

toVisit.erase(toVisit.begin());

if (toVisit.size() == 0){

//if list is empty the root is the original root and can be deleted

delete(root);

}

else{

//if list is empty the root is the next node. the next node can be a right child still not explored or the root

root = \*toVisit.begin();

lastSelected = \*toVisit.begin();

}

}

}

//clean leaves vector for next round

leaves.resize(0);

}

}

18)

La traccia mostra un esempio con char anzichè int ma sia la struct che la funzione prevedono un int, per tanto l’esercizio è stato adattato a lavorare su int dal momento che ciò non cambia l’algoritmo di risoluzione.

Assunzione: il nodo da reversare non è mai il primo ed il primo non è mai null. Se dovessi reversare dal primo si tratterebbe di una funzione swap.

void ReverseAfter(Node\* head, int value){

while (head->next != NULL){

if (head->next->value == value){

//store in "value" the node before the first node to reverse

value = (int)head;

head = head->next;

//search the last node to move after “value”

while (head->next != NULL){

head = head->next;

if (head->next == NULL){

//last node is founded, now it will be moved before the reversing point

head->next = ((Node\*)(value))->next;

((Node\*)(value))->next = head;

head = head->next;

value = head->value;

//but now there is a cicle to break

while (head->next->next->value != value){

head = head->next;

}

value = (int)head->next;

head->next = NULL;

//cicle is break, head is now again at the reversing point

head = (Node\*)(value);

}

}

}

else{

head = head->next;

}

}

}

19)

L’assunzione sbagliata alla base dell’algoritmo è che nessuno garantisce che dst sia almeno capiente nBytes, altrimenti scrive su memoria che potrebbe essere utilizzata da altre cose.

Un’errore di portabilità è che si ci sta basando su l’assunzione di valori di int, char, int\* e char\* “standard” ma su altre architetture potrebbero essere rappresentati con un diverso numero di bit e byte.

Da ciò deriva il fatto che nBytes non è detto che shiftando di due a destra ( dividendo quindi per 4) contenga il corretto numero di char.

Ciò è vero sia per il cast da char\* a int\* che da int\* a char\* che si basa su rappresentazioni “standard”.

L’errore di sintassi sono il ; dopo il secondo for e prima della graffa, ed il <= ( che dovrebbe essere solo <) sempre del secondo for.

void myMemcpy(char\* dst, const char\* src, int nBytes){

// Try to be fast and copy a word at a time instead of byte by byte

int\* wordDst = (int\*)dst;

int\* wordSrc = (int\*)src;

int numWords = nBytes / (sizeof(int) /sizeof(char));

int i = numWords;

while(i>0){

\*wordDst++ = \*wordSrc++;

i--;

}

i = nBytes - (numWords \* (sizeof(int) /sizeof(char));

dst = (char\*)wordDst;

src = (char\*)wordSrc;

while(i>0){

\*dst++ = \*src++;

i--;

}

}

20)

void \*p = malloc(1024+15);

void \*alignedPointer = (void\*)(((uintptr\_t)p + 15) & ~(uintptr\_t)0x0F);

…...

alignedPointer = nullptr;

free(p);