

vector<QAbstractButton\*> fun(list<QWidget\*>& lista, const QSize& size) {

vector<QAbstractButton\*> vett;

list<QWidget\*> :: iterator it = lista.begin();

while(it != lista.end()) {

if(\*it && (\*it)->sizeHint() == size) lista.push\_front((\*it)->clone());

QAbstractSlider\* p = dynamic\_cast<QAbstractSlider\*>(\*it);

if(\*it && !p && (\*it)->sizeHint() == s) {

delete \*it;

it = lista.erase(it);

} else if(dynamic\_cast<QCheckBox\*>(\*it) || dynamic\_cast<QPushButton\*>(\*it)) {

vett.push\_back(static\_cast<QAbstractButton\*>(\*it));

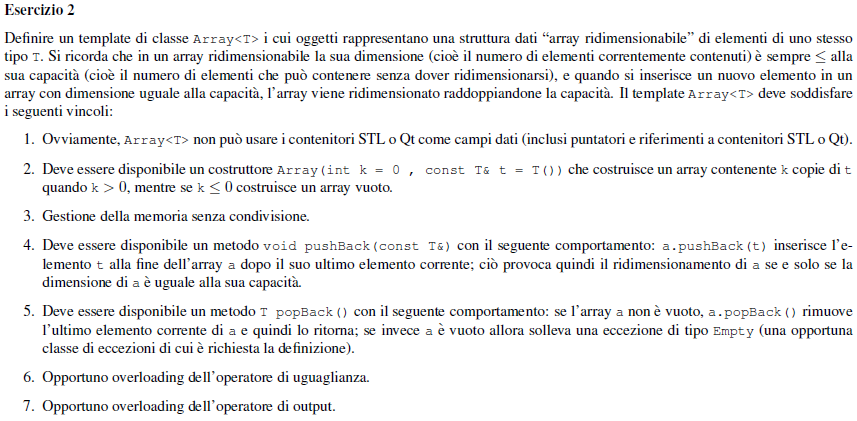
it = lista.erase(it);

} else ++it;

}

return vett;

}



class Empty{

public:

Empty();

};

template <class T>

class Array{

friend ostream& operator<<(ostream& os, const Array<T>&);

private:

T\* x;

int size;

int capacity;

static T\* copia(T\* j, int y, int z){

T\* a = new T[z];

for (int i=0; i<y; i++) a[i]=j[i];

return a;

}

public:

Array(int k=0, const T& t = T()): x(0), size(k), capacity(k\*2){

if (k) {

x=new T[k\*2];

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

x[i]=t;

}

}

Array(const Array& a) : x(copia(a.x,a.size,a.capacity)), size(a.size), capacity(a.capacity) { }

~Array() { delete [] x;}

Array& operator=(const Array& a){

if (this != &a) {

delete [] x;

x=copia(a.x,a.size,a.capacity);

size = a.size;

capacity = a.capacity;

}

return \*this;

}

continua..

..continua

void pushBack(const T& t){

if (size==capacity) {

x=copia(x, size, capacity\*2);

capacity=capacity\*2;

x[size]=t;

size++;

}

else { x[size]=t;

size++;

}

}

T popBack(){

if (!size) throw Empty();

else { T a=x[size-1];

delete x[size-1];

size--;

return a;

}

}

bool operator==(const Array& a) const {

if (size != a.size) return false;

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

return true;

}

};

template<class T>

ostream& operator<<(ostream& os, const Array<T>& a){

for (int i=0; i<a.size; i++) {

os << (a.x)[i] << " ";

}

return os;

}

int main(){

B\* pb=new B;

C\* pc = new C;

D\* pd = new D;

E\* pe = new E;

A \*pa1=pb, \*pa2=pc, \*pa3=pd, \*pa4=pe;

C \*pc1=pe;

(dynamic\_cast<B\*>(pa1))->*f*(1); // B::f(const int&)

(dynamic\_cast<B\*>(pa1))->*f*(true); // B::f(const bool&)

pa1->*f*(true); // A::f(bool)

pa2->*f*(1); // A:.f(int)

(dynamic\_cast<C\*>(pa2))->*f*(1); // C::f(Z)

(dynamic\_cast<E\*>(pa2))->f(1); ERRORE RUN-TIME

(dynamic\_cast<C\*>(pa3))->*f*(0); // C::f(Z)

(dynamic\_cast<D\*>(pa3))->*f*(0); // D::f(int\*)

pa4->*f*(1); // A::f(int)

(dynamic\_cast<C\*>(pa4))->*f*(1); // E::f(Z)

pc1->*f*(1); // E::f(Z)

(static\_cast<E\*>(pc1))->*f*(1); // E::f(Z)

(static\_cast<A\*>(pc1))->*f*(1); // A::f(int)

}

class Z {

public:

Z(int x) {}

};

class A {

public:

void *f*(int){cout << "A::f(int) ";}

virtual void *f*(bool){cout <<"A::f(bool) ";}

virtual void *f*(Z){cout <<"A::f(Z) ";}

};

class B: virtual public A {

public:

void *f*(const bool&){cout<< "B::f(const bool&) ";}

void *f*(const int&){cout<< "B::f(const int&) ";}

};

class C: virtual public A {

public:

virtual void *f*(Z){cout <<"C::f(Z) ";}

};

class D: public B, public C {

public:

virtual void *f*(int\*){cout<< "D::f(int\*) ";}

void *f*(int&){cout <<"D::f(int&) ";}

};

class E: public D {

public:

void *f*(Z){cout <<"E::f(Z) ";}

};

#include <iostream>

using namespace std;

class Z {

public:

Z(int x) {}

};

class A {

public:

void *f*(int) {cout << "A::f(int) "; *f*(true);}

virtual void *f*(bool) {cout <<"A::f(bool) ";}

virtual A\* *f*(Z) {cout <<"A::f(Z) "; *f*(2); return this;}

A() {cout <<"A() "; }

};

class C: virtual public A {

public:

C\* *f*(Z){cout <<"C::f(Z) "; return this;}

C() {cout <<"C() "; }

};

class B: virtual public A {

public:

void *f*(const bool&){cout<< "B::f(const bool&) ";}

void *f*(const int&){cout<< "B::f(const int&) ";}

virtual B\* *f*(Z) {cout <<"B::f(Z) "; return this;}

virtual ~*B*() {cout << " ̃B ";}

B() {cout <<"B() "; }

};

class D: public B {

public:

virtual void *f*(bool) const {cout <<"D::f(bool) ";}

B\* *f*(Z) {cout << "D::f(Z) "; return this;}

~*D*() {cout <<" ̃D ";}

D() {cout <<"D() "; }

};

class E: public D, public C {

public:

void *f*(bool){cout<< "E::f(bool) ";}

E\* *f*(Z){cout <<"E::f(Z) "; return this;}

E() {cout <<"E() "; }

~*E*() {cout <<" ̃E ";}

};

int main(){

B\* pb = new B; C\* pc = new C; D\* pd = new D; E\* pe = new E;

A\* pa1 = pb, \*pa2=pc, \*pa3=pd, \*pa4=pe; B \*pb1=pe;

E\* puntE = new E; | A() B() D() C() E()

D\* puntD = new D; | A() B() D()

pa3 -> *f*(3); | A::f(int) A::f(bool) Perchè D::f(bool) è marchiata costante

pa4->*f*(3); | A::f(int) E::f(bool)

pb1->*f*(true); | B::f(const bool&)

pa4->*f*(true); | E::f(bool)

pa2->*f*(Z(2)); | C::f(Z)

pa4->*f*(Z(2)); | E::f(Z)

pb->*f*(3); | B::f(const int&)

pc->*f*(3);\ | C::f(Z)

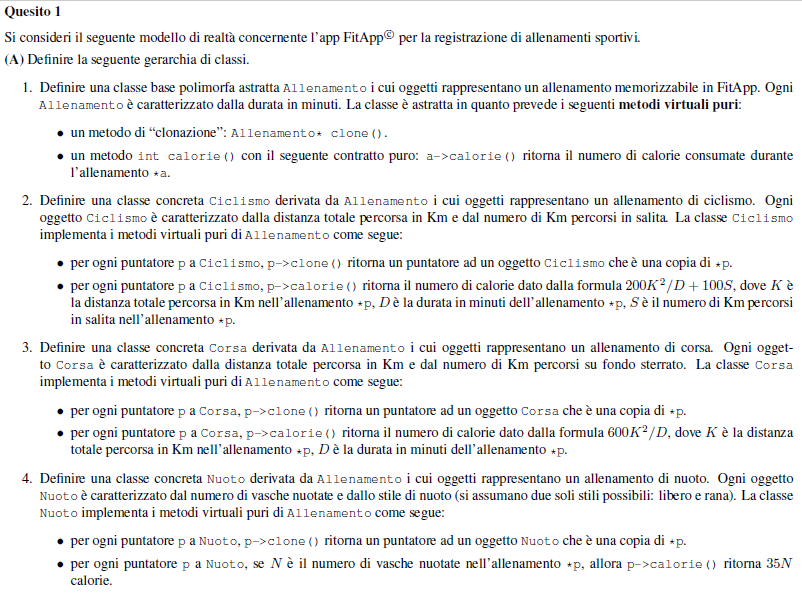
(pa4->*f*(Z(3)))->*f*(4); | E::f(Z) A::f(int) E::f(bool)

(pc->*f*(Z(3)))->*f*(4); | C::f(Z) C::f(Z)

delete pa4;

delete pd; | ~D ~B dovrebbe stampare A ma non ha ridefinito il distruttore

}



class Corsa: public Allenamento {

private:

double distanza, sterrato;

public:

Allenamento\* *clone*() const {

return new Corsa(\*this);

}

int *calorie*() const {

return (600\*pow(distanza,2))/durata;

}

double getSterrato() const { return sterrato; }

};

#include <string>

#include <vector>

#include <math.h>

using std::string;

using std::vector;

class Allenamento {

protected:

double durata;

public:

virtual Allenamento\* *clone*() const = 0;

virtual int *calorie*() const = 0;

virtual ~*Allenamento*() {};

};

class Nuoto: public Allenamento {

private:

double vasche;

string stile;

public:

Allenamento\* *clone*() const { return new Nuoto(\*this); }

int *calorie*() const { return 35\*vasche; }

string getStile() const { return stile; }

};

class Ciclismo: public Allenamento {

private:

double distanza, salita;

public:

Allenamento\* *clone*() const { return new Ciclismo(\*this); }

int *calorie*() const {

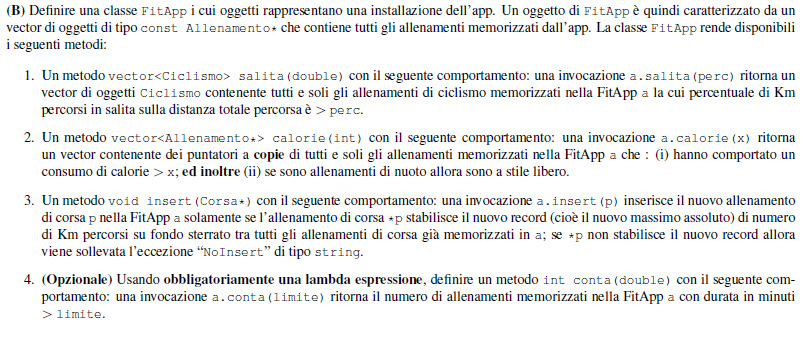
return (200\*pow(distanza,2))/durata + 100\*salita;

}

double getDistanza() const { return distanza; }

double getSalita() const { return salita; }

};



..continua

void insert(Corsa\* p) {

double record = 0;

for (int i = 0; i < a.size(); ++i) {

Corsa\* c = dynamic\_cast<Corsa\*>(const\_cast<Allenamento\*>( a[i] ));

if (c && c -> getSterrato() > record)

record = c -> getSterrato(); }

if (p -> getSterrato() > record) a.push\_back(p);

}

int conta(double lim) {}

};

class FitApp {

private:

vector<const Allenamento\*> a;

public:

vector<Ciclismo> salita(double perc) const {

vector<Ciclismo> v;

for (int i = 0; i < a.size(); ++i) {

Ciclismo\* c = dynamic\_cast<Ciclismo\*>(const\_cast<Allenamento\*>(a[i]));

if (c && (100/c->getDistanza())\*c -> getSalita() > perc)

v.push\_back(\*c);

}

return v;

}

vector<Allenamento\*> calorie(int x) const {

vector<Allenamento\*> v;

for (int i = 0; i < a.size(); ++i) {

if (a[i] -> calorie() > x) {

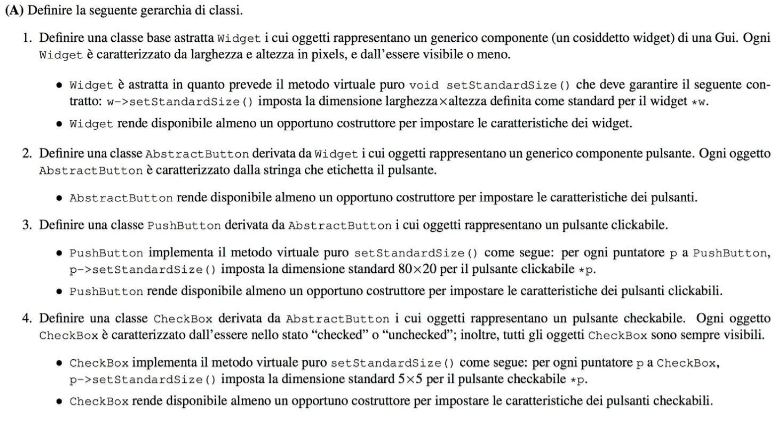
Nuoto\* n = dynamic\_cast<Nuoto\*>(const\_cast<Allenamento\*>( a[i] ));

if (n && n -> getStile() == "libero") v.push\_back(a[i] -> clone());

else v.push\_back(a[i] -> clone());

} } }

continua…



#include <iostream>

#include <vector>

#include <list>

using namespace std;

class Widget{

private:

unsigned int lung, alt;

bool visibile;

public:

Widget(unsigned int lu, unsigned int al, bool vis) : lung(lu), alt(al), visibile(vis){ }

virtual void *setStandardSize*() = 0;

virtual ~*Widget*();

unsigned int getLung() const{ return lung; }

unsigned int getAlt() const{ return alt; }

bool isVisible() const{ return visibile; }

void setLung(unsigned int lun){ lung = lun; }

void setAlt(unsigned int al){ alt = al; }

};

class AbstractButton : public Widget{

private:

std::string etichetta;

public:

AbstractButton(unsigned int lun = 1, unsigned int al = 1, bool vis = true, std::string et = "")

: Widget(lun, al, vis), etichetta(et){ }

std::string getEtichetta(){ return etichetta; }

};

class PushButton : public AbstractButton{

public:

PushButton(unsigned int lun = 80, unsigned int al = 20, bool vis = true, std::string et = "")

: AbstractButton(lun, al, vis, et){ }

void *setStandardSize*(){ setLung(80); setAlt(20); }

};

class CheckBox : public AbstractButton{

private:

bool checked;

public:

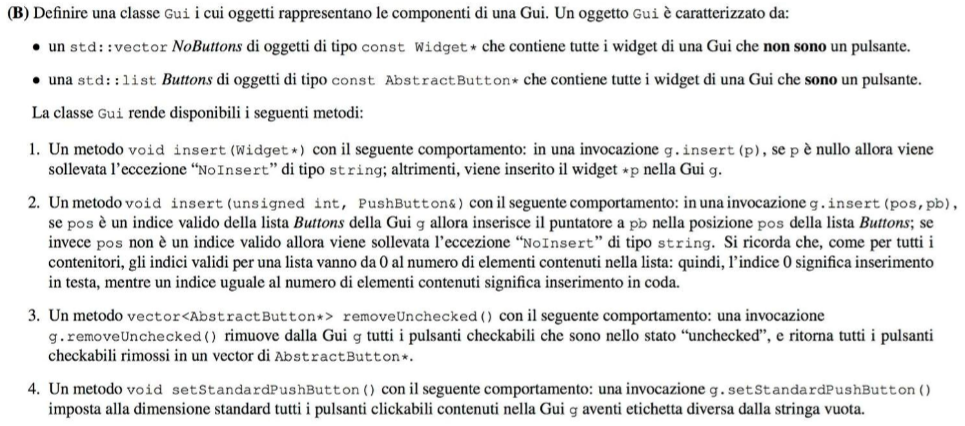
CheckBox(unsigned int lun = 5, unsigned int al = 5, bool vis = true, bool check = false)

: AbstractButton(lun, al, vis), checked(check){ }

void *setStandardSize*(){ setLung(5); setAlt(5); }

bool isChecked() const{ return checked; }

};



class Gui{

private:

vector<const Widget\*> NoButtons;

list<const Widget\*> Buttons;

public:

void insert(Widget\* p){

if(!p) throw string("NoInsert"); //eccezione di tipo string

else{ AbstractButton\* pab = dynamic\_cast<AbstractButton\*> (p);

if(pab) Buttons.push\_back(pab);

else NoButtons.push\_back(p);

}

}

void insert(unsigned int pos, PushButton& pb){

if(pos < 0 || pos > Buttons.size()) throw string("NoInsert");

if(pos == 0) Buttons.push\_front(&pb);

else if(pos == Buttons.size()) Buttons.push\_back(&pb);

else{ list<const AbstractButton\*> :: iterator it;

int i = 1;

for(it = ++Buttons.begin(); it != Buttons.end(); ++it, ++i){

if(pos == i) Buttons.insert(it, &pb);

}

}

}

vector<AbstractButton\*> removeUnchecked(){

vector<AbstractButton\*> uncheckedButtons;

list<const AbstractButton\*> :: iterator it;

for(it = Buttons.begin(); it < Buttons.end(); ++it){

AbstractButton\* aux = const\_cast<AbstractButton\*>(\*it);

CheckBox\* cb = dynamic\_cast<CheckBox\*>(aux);

if(cb && !cb -> isChecked()){

uncheckedButtons.push\_back(cb);

Buttons.erase(it);

}

}

return uncheckedButtons;

}

void setStandardPushButton() const{

list<const AbstractButton\*> :: const\_iterator it;

for(it = Buttons.begin(); it != Buttons.end(); ++it){

AbstractButton\* aux = const\_cast<AbstractButton\*>(\*it);

CheckBox\* cb = dynamic\_cast<CheckBox\*>(aux);

if(cb && cb -> getEtichetta() != "")

cb -> *setStandardSize*();

}

}

};

| TS | TD|

p | B\* | T\* |

pa | A\* | B\*|

pb | B\* | F\*|

pc | C\* | B\*|

pd | D\*| E\*|

pe | E\* | E\*|

using std::cout;

using std::cin;

class A {

public:

virtual ~*A*() {delete this; };

};

class B: virtual public A {};

class C: virtual public A {};

class D: public B, public C {};

class E: public D {};

class F: public E {};

template<class T>

void Fun(T& ref){

bool b = 0;

B\* p = &ref; ⇒ conv B\*⇒T\*

try{ throw ref; }

catch(E) { cout << "E "; b=1; }

catch(D) { cout << "D "; b=1; }

catch(B) { cout << "B "; b=1; }

catch(A) { cout << "A "; b=1; }

catch(C) { cout << "C "; b=1; }

if(b==0) cout << "ZERO ";

}

int main(){

/\*

T(S) = Tipo Statico

T(D) = Tipo Dinamico

T(S) p = B\* T(D) p = T\*

T(S) pa = A\* T(D) pa = B\*

T(S) pb = B\* T(D) pb = B\*

T(S) pc = C\* T(D) pc = E\*

T(S) pd = D\* T(D) pd = F\*

T(S) pe = E\* T(D) pe = E\*

\*/

A a; B b; C c; D d; E e; F f;

A\* pa = &b; D\* pd = &f;

B\* pb = dynamic\_cast<B\*>(pa); C\* pc = dynamic\_cast<E\*>(pd); E\* pe = static\_cast<E\*>(pd);

//Fun(a); //NON COMPILA invalid conversion from A\* to B\*

Fun(b); B

//Fun(c); NON COMPILA cannot convert C\* to B\* in initialization B\*p = &ref

//Fun(d); D

//Fun(e); E

//Fun(f); E (quando fa la throw di ref, ref ha tipo statico F.

// Lui guarda la prima catch, che prevede un E. Essendo F

// sottotipo di E, estrae il sottooggetto di F, che è E, ed entra dentro la catch.)

//Fun(\*pa); NON COMPILA invalid conversion from A\* to B\*

//Fun(\*pb); B

//Fun(\*pc); NON COMPILA cannot convert C\* to B\* in initialization B\*p = &ref

//Fun(\*pd); D

//Fun(\*pe); E

//Fun<B>(\*pd); B

//Fun<D>(\*pd); D

//Fun<E>(\*pd); NON COMPILA cannot convert \*pd (type D) to type E&

}

Fun(a); ⇒ prova conv B\*⇒ TS(pa) = A\* senza cast con B<A ⇒ NO

Fun(b); ⇒ prova conv B\*⇒ TS(pb) = B\* ⇒ OK stampa B

Fun(c); ⇒ prova conv B\*⇒C\* ⇒ C non è in gerarchia con B ⇒ NO

Fun(d); ⇒ prova conv B\*⇒D\* ⇒ D<B ⇒ OK stampa D

Fun(e); ⇒ prova conv B\*⇒E\*⇒ E<B ⇒ OK stampa E

Fun(f); ⇒ prova conv B\*⇒F\* ⇒ quando fa la throw, TS(ref)=F e in questo caso il comp prende la prima

catch con E in quanto estrae da F il relativo sottooggetto E ⇒ OK stampa E

Fun(\*pa); ⇒ prova conv B\*⇒A\* senza cast ⇒ NO

Fun(\*pb); ⇒ prova conv B\*⇒B\* ⇒ OK stampa B

Fun(\*pc); ⇒ prova conv B\*⇒C\* ⇒ C non è in gerarchia con B ⇒ NO

Fun(\*pd); ⇒ prova conv B\*⇒D\* ⇒ D<B ⇒ OK stampa D

Fun(\*pe); ⇒ prova conv B\*⇒E\*⇒ E<B ⇒ OK stampa E

Fun<B>(\*pd); ⇒ istanzia Fun<B> ⇒ prima fa conv TS(pd)= D\*⇒B\* e poi nella funz conv B\* ⇒ B\* OK stampa B

Fun<D>(\*pd); ⇒ istanzia Fun<D> ⇒ prima fa conv TS(pd)= D\*⇒D\* e poi nella funz conv B\* ⇒ D\* OK stampa D

Fun<E>(\*pd); ⇒ istanzia Fun<E> ⇒ prima conv TS(pd) = D\*⇒E\* senza cast con E<D ⇒ NO

int main(){

A a;

B b;

C c;

D d;

E e;

F f;

A\* pa = &b;

D\* pd = &f;

B\* pb = dynamic\_cast<B\*>(pa);

C\* pc = dynamic\_cast<E\*>(pd);

E\* pe = static\_cast<E\*>(pd);

}

Es: definire una gerarchia di classi che include:

• Una classe base polimorfa A nella radice della gerarchia;

• Una classe derivata astratta B;

• Una sottoclasse C di B che sia concreta;

• Una classe D definita tramite derivazione multipla con base virtuale.

class A{

public:

virtual void *f*(){ } ⇒ classe base polimorfa

};

class B : virtual public A {

public:

void *f*(){ }

virtual void *metodo*() const = 0; ⇒ funzione virtuale pura -> classe B astratta

};

class C : public B {

public:

void *metodo*() const{ } ⇒ ridefinizione di metodo -> classe C concreta

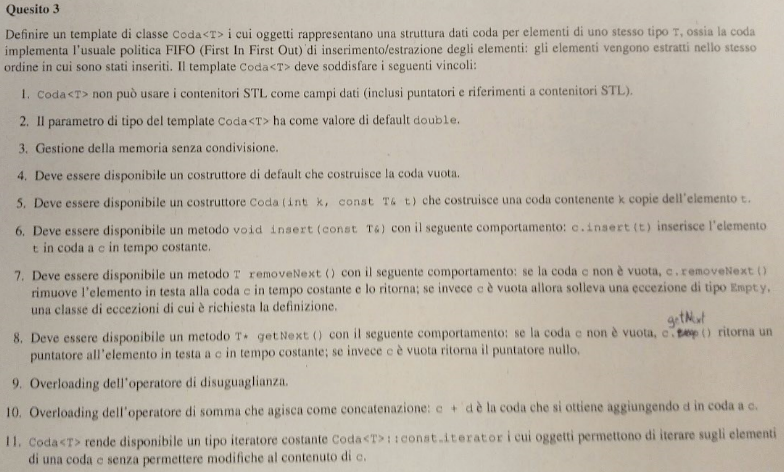
};

class D : virtual public A, public C {

public:

void *f*(){ }

};



#include <iostream>

using namespace std;

class Empty{

public:

Empty();

};

template <class T = double> //double valore di default

class Coda{

friend class const\_iteratore;

private:

class Nodo{

public:

T info;

Nodo\* next;

Nodo(Nodo\* n, T i) : next(n), info(i){ }

Nodo(const T& t) : info(t), next(0){ }

~Nodo(){ delete next; }

Nodo(const Nodo& n) : info(n.info), next(n.next){ }

Nodo& operator=(const Nodo& n){

if(this != &n){

delete \*this;

info = n.info;

next = n.next;

}

return \*this;

}

static bool uguale(Nodo\* n1, Nodo\* n2) const{

if(!n1 && !n2) return true;

else if( (!n1 && n2) || (n1 && !n2)) return false;

return n1 -> info == n2 -> info && uguale(n1 -> next, n2 -> next);

}

};

Nodo\* first;

Nodo\* last;

public:

Coda() : first(0), last(0){ }

Coda(int k, const T& t){ //costruisce Coda con k copie dell'elem t

if(!k) first = last = 0;

else{ first = last = new Nodo(t);

--k;

while( k!= 0){

Nodo\* temp = new Nodo(t);

last -> next = temp;

last = temp;

--k;

}

} }

T removeNext(){ //rimuove elem in testa alla coda

if(is\_empty()) throw Empty();

Nodo\* temp = first;

first = first->next;

T rit = temp->info;

delete temp;

return rit;

}

T\* getNext() const{ //ritorna puntatore all'elem in testa alla coda

if(is\_empty()) return 0;

else return &(first -> info);

}

bool operator!=(const Coda& c) const{ return !uguale(first, c.first); }

class const\_iteratore{ //permette di iterare sugli elementi della coda

private:

const Nodo\* it;

public:

const\_iteratore operator++(){

if(it) it = it -> next;

return \*this;

}

const T& operator\*() const{

return it -> info;

}

const T operator[](const\_iteratore i){

return (i.it) -> info;

}

};

};

bool is\_empty() const{ return first == 0; }

void insert(const T& t){

Nodo\* temp = new Nodo(t);

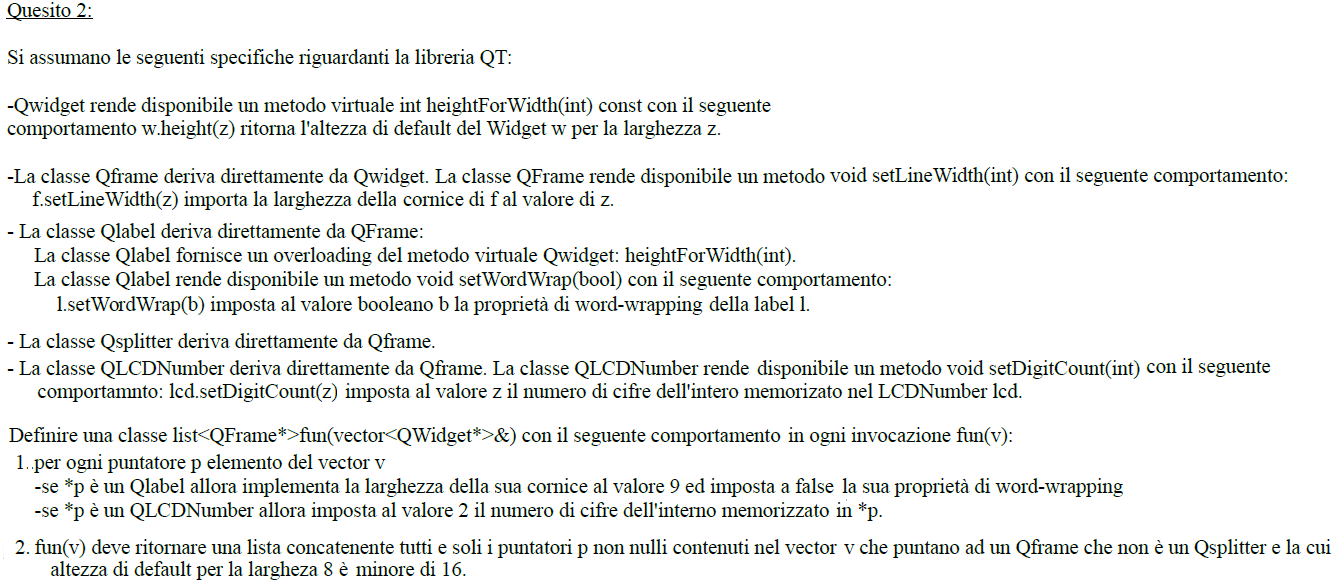
if(is\_empty()) first = last = temp;

else{ last -> next = temp;

last = temp;

}

}



list<QFrame\*>fun(vector<QWidget\*>& v){

list<QFrame\*> rit;

vector<QWidget\*> :: iterator it;

for(it = v.begin(); it!=v.end(); ++it){

QLabel\* pQL = dynamic\_cast<QLabel\*>(\*it);

if(pQL){ pQL -> setLineWidth(9);

pQL -> setWordWrap(false);

}

QLCDNumber\* pQLCD = dynamic\_cast<QLCDNumber\*>(\*it);

if(pQLCD) pQLCD -> setDigitCount(2);

QFrame\* pQF = dynamic\_cast<QFrame\*>(\*it);

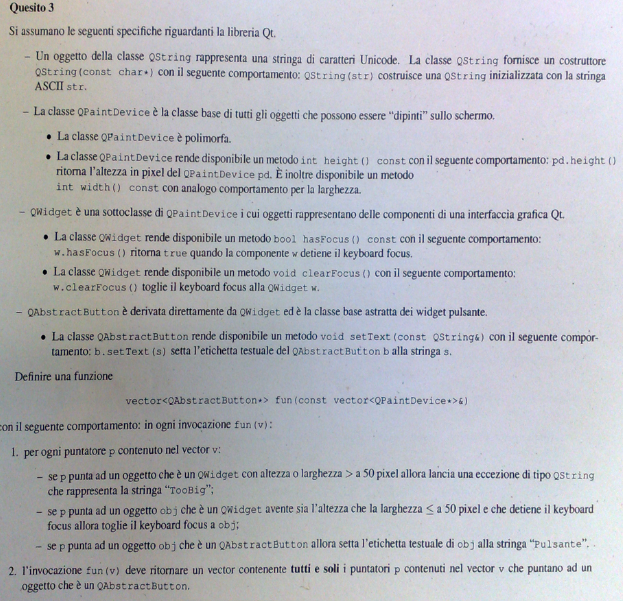
if(pQF && !dynamic\_cast<QSplitter\*>(pQF) && pQF -> *heightForWidth*(8) < 16)

rit.push\_back(pQF);

}

return rit;

}



vector< QAbstractButton\*> fun(const vector<QPaintDevice\*>& v){

vector < QAbstractButton\*> rit;

for(int it = 0; it != v.size(); ++it){

QWidget \* pqw = dynamic\_cast<QWidget\*>(v[it]);

if(pqw){

if(pqw -> heigth() > 50 || pqw -> width() > 50) throw QString("TooBig");

if(pqw -> hasFocus()) pqw -> clearFocus();

}

QAbstractButton\* pqab = dynamic\_cast< QAbstractButton\*>(v[it]);

if(pqab){

pqab -> setText("Pulsante");

rit.push\_back(pqab);

}

}

return rit;

}

∎ Ridefinire l’operator= di D in modo che si comporti come l’operator= standard:

class Z { private: int x; };

class B { private: Z x; };

class D: public B { private: Z y;

public: D& operator=(const D&); ⇒ ridefinizione di operator=

};

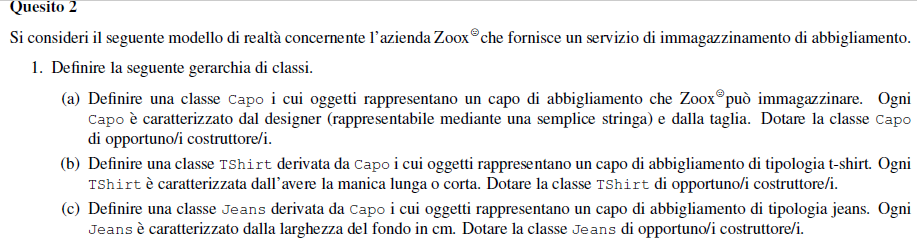
D& D::operator= (const D& x){

this->B::operator=(x);

y = x.y; ⇒ campo dati

return \*this;

}



class Capo{

private:

string brand;

int taglia;

public:

Capo(string b = "noBrand", int t = 40) : brand(b), taglia(t){ }

virtual ~*Capo*() =0; //distr virt puro

virtual bool operator==(const Capo& c) const{

return brand == c.brand && taglia == c.taglia;

}

string getBrand() const{ return brand; }

};

class Jeans : public Capo{

private:

int fondo;

public:

Jeans(string b = "noBrand", int t = 40, int f = 20) : Capo(b,t), fondo(f){ }

virtual bool operator==(const Capo& c) const{

return typeid(const Jeans&) == typeid(c) && Capo::operator==(c) &&

fondo == (static\_cast<const Jeans&>(c)).fondo;

}

int getFondo() const{ return fondo; }

};

class Tshirt : public Capo{

private:

bool corta;

public:

Tshirt(string b = "noBrand", int t = 40, bool c = true) : Capo(b,t), corta(c){ }

virtual bool operator==(const Capo& c) const{

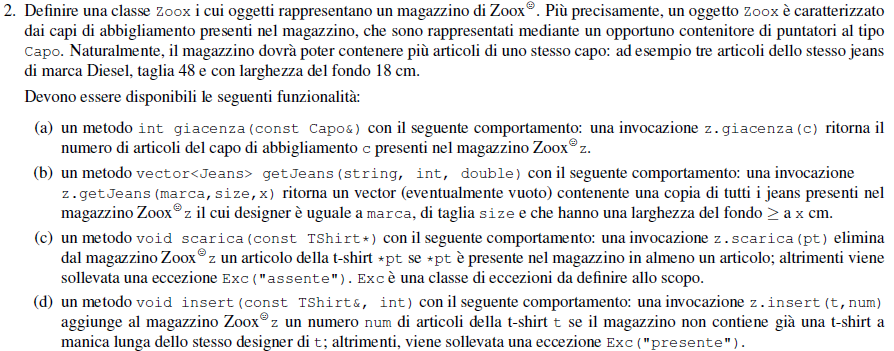
return typeid(const Tshirt&) == typeid(c) && Capo::operator==(c) &&

corta == (static\_cast<const Tshirt&>(c)).corta;

}

bool getManicaL() const{ return !corta; }

};



class Articolo{

friend class Zoox;

private:

Capo\* c; //puntatore polimorfo

int q; //num art presenti in magazzino

public:

Articolo(Capo\* pc, int x = 1) : c(pc), q(x){ }

int getQuantita() const{ return q; }

void scarica(){

if(q>1){ --q; return; }

else delete c;

}

bool uguale(const Capo& c) const{

return \*(this -> c) == c;

}

};

class Exception{

public:

Exception(string s = ""){ }

};

vector<Jeans> getJeans(string m, int s, int f) const{

vector<Jeans> v;

list<Articolo> :: const\_iterator it;

for(it = mag.begin(); it != mag.end(); ++it){

Jeans\* p = dynamic\_cast<Jeans\*>(it->c);

if(p && Jeans(m,s).Capo::operator==(\*(it->c)) && p -> getFondo() >= f)

v.push\_back(\*p);

}

return v;

}

class Zoox{ //classe contenitore

private:

list<Articolo> mag;

public:

int giacenza(const Capo& c) const{

list<Articolo> :: const\_iterator it;

for(it = mag.begin(); it != mag.end(); ++it)

if(it -> uguale(c)) return it -> getQuantita();

return 0;

}

void scarica(const Tshirt\* pt){

list<Articolo> :: iterator it;

for(it = mag.begin(); it != mag.end(); ++it){

if(it -> uguale(\*pt)){

it -> scarica();

mag.erase(it);

return;

}

}

throw Exception("assente");

}

void insert(const Tshirt& t, int x){

list<Articolo> :: const\_iterator it;

for(it = mag.begin(); it != mag.end(); ++it)

if(typeid(\*(it -> c)) == typeid(Tshirt)){

Tshirt\* p = static\_cast<Tshirt\*>(it -> c);

if(p->getManicaL() && t.getBrand() == p -> getBrand())

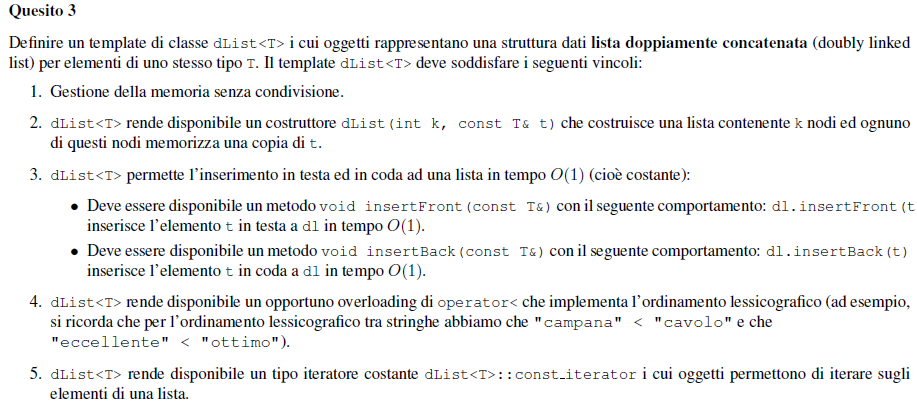
throw Exception("presente");

}

mag.push\_front(Articolo(const\_cast<Tshirt\*>(&t), x));

}

};



template <class T>

class dList{

friend class const\_iterator;

private:

class nodo{

public:

T info;

nodo \*prev, \*next;

nodo(nodo\* p = 0, const T& t = T(), nodo\* n = 0) : prev(p), info(t), next(n){ }

~nodo(){ if(next) delete next; }

};

nodo \*first, \*last;

static nodo\* copia(nodo\* p){

if(p == 0) return 0;

else{ nodo\* f = new nodo(0, p->info, 0);

nodo\* currN = f;

while(p->next != 0){

currN -> next = new nodo(currN, p->next->info, 0);

currN = currN->next;

p = p -> next;

}

return f;

}

}

static nodo\* getLast(nodo\* f){

while(f && f->next) f = f->next;

return f;

}

static bool compareLex(nodo\* p1, nodo\* p2){

if(!p1) return true;

if(!p2) return false;

if(p1->info < p2->info) return true;

if(p2->info > p2->info) return false;

return compareLex(p1->next, p2->next);

}

public:

dList() : first(0), last(0){ }

~dList(){ if(first) delete first; }

dList(const dList& l) : first(copia(l.first)), last(getLast(first)){ }

dList& operator=(const dList& l){

if(this != &l){

if(first) delete first;

first = copia(l.first);

last = getLast(first);

}

return \*this;

}

dList(int k, const T& t){

if(k<=0){ first = last = 0; }

else{

first = new nodo(0,t,0);

nodo\* temp = first;

--k;

while(k>0){

temp -> next = new nodo(temp, t, 0);

temp = temp -> next;

--k;

}

last = temp;

}

}

class const\_iterator{

friend class dList<T>;

private:

const nodo\* punt;

const\_iterator(const nodo\* p = 0) : punt(p){ } //costruttore privato

public:

const T& operator\*() const{

return punt -> info;

}

const T\* operator->() const{

return &(punt->info);

}

const\_iterator& operator++(){

if(punt) punt = punt -> prev;

return \*this;

}

bool operator==(const const\_iterator& i) const{

return punt == i.punt;

}

};

const\_iterator begin() const{

return const\_iterator(first);

}

const\_iterator end() const{

return const\_iterator(0);

}

};

void insertFront(const T& t){

first = new nodo(0,t,first);

if(first->next == 0) last = first;

if(first->next) (first->next)->prev = first;

}

void insertBack(const T& t){

last = new nodo(last,t,0);

if(last->prev != 0) (last->prev)->next = last;

else first = last;

}

bool operator<(const dList& l) const{

return compareLex(first, l.first);

}

JZ