

Object-oriented programming (Programmazione ad oggetti) Parte 1

Liceo G.B. Brocchi - Bassano del Grappa (VI)
Liceo Scientifico - opzione scienze applicate
Giovanni Mazzocchin

Unire dati e operazioni in un unico costrutto

- I tipi predefiniti di C++ sono detti **tipi built-in**:
 - char, int, float, double etc...
- Il compilatore sa come rappresentarli in memoria e conosce le operazioni che possono essere fatte su di essi
- I tipi definiti dall'utente sono detti **User-Defined Types**
 - per crearli abbiamo utilizzato le **struct**
 - ma le struct avevano un limite: non potevamo associare dati e operazioni all'interno del nuovo tipo
- L'**OOP** (**O**bject-**o**riented **P**rogramming) permette di superare questo limite. È il paradigma di programmazione più utilizzato nello sviluppo software al giorno d'oggi

Unire dati e operazioni in un unico costrutto

- Nell'OOP, un oggetto è caratterizzato da:
 - uno **stato**
 - un **insieme di operazioni**
 - un'**identità** (tipo)
- Ad esempio, in un videogioco sviluppato ad oggetti, l'oggetto **player_1** può avere:
 - **stato**: armi possedute, livello raggiunto etc...
 - **operazioni**: salta, corri, spara etc...
 - **identità**: **player** (ci possono essere diversi oggetti (*istanze*) di tipo `player`)

Unire dati e operazioni in un unico costrutto

- Immaginate una **rubrica telefonica** (*phone book*)
- Sarebbe molto utile poter creare il tipo **phone_book**
- Questo tipo non è caratterizzato soltanto dai dati (i numeri di telefono associati ai nomi), ma anche da diverse operazioni sui dati:
 - ricerca, aggiunta, rimozione, aggiornamento
- Vogliamo creare un tipo che rappresenti i dati e definisca le operazioni possibili su di essi
- Le operazioni esposte all'esterno costituiscono l'**interfaccia** del tipo

Le classi

```
class Point {  
    double x;  
    double y;  
  
    double get_greatest_coordinate() {  
        if (x >= y) {  
            return x;  
        }  
        return y;  
    }  
};  
  
int main() {  
    Point p;  
    p.x = 5;  
}
```

```
class.cpp  
class.cpp(18): error C2248:  
'Point::x': impossibile accedere al  
membro privato dichiarato nella  
classe 'Point'
```

NON COMPILA: i membri di una classe, di default, non sono accessibili dall'esterno. Si dice che sono *incapsulati*

```
class Point {  
public:
```

```
    double x; //data member  
    double y; //data member
```

```
    double get_greatest_coordinate() { //function member ("method")  
        if (x >= y) {  
            return x;  
        }  
        return y;  
    }  
};
```

```
int main() {  
    Point p; //object of class Point  
    p.x = 5; //access to data member  
    p.y = 8; //access to data member
```

```
    cout << p.get_greatest_coordinate(); //access to function member ("method")  
}
```

la keyword `public` rende visibili i membri all'esterno della classe

come fa a funzionare se non prende nessun parametro?
come fa a sapere che i membri `x` e `y` appartengono all'oggetto `p` del `main`?

Classi vs struct

```
struct point {  
    double x;  
    double y;  
    double (*get_greatest_coordinate_ptr) (struct point*);  
};
```

```
double get_greatest_coordinate(struct point* p) {  
    if (p->x >= p->y) {  
        return p->x;  
    }  
    return p->y;  
}
```

**qui il parametro è specificato;
si vede chiaramente che vengono
confrontate le coordinate del punto p**

Classi vs struct

```
int main(){  
    struct point p = {9, 1, &get_greatest_coordinate};  
    cout << (*(p.get_greatest_coordinate_ptr))(&p);  
}
```



```

class Point {
public:
    double x;
    double y;

    double get_greatest_coordinate() {
        if (x >= y) {
            return x;
        }
        return y;
    }
};

int main() {
    Point p;
    p.x = 5;
    p.y = 8;
    cout << p.get_greatest_coordinate();
}

```

**è come se ci fosse un
parametro nascosto che
permette di accedere ai
membri di p del main**

```

class Point {
public:
    double x;
    double y;

    double get_greatest_coordinate() {
        if (x >= y) {
            return x;
        }
        return y;
    }
};

int main() {
    Point p;
    p.x = 5;
    p.y = 8;
    cout << p.get_greatest_coordinate();
}

```

**in realtà un parametro c'è,
solo che è sottinteso:
si chiama this ed è un
puntatore a Point**

```

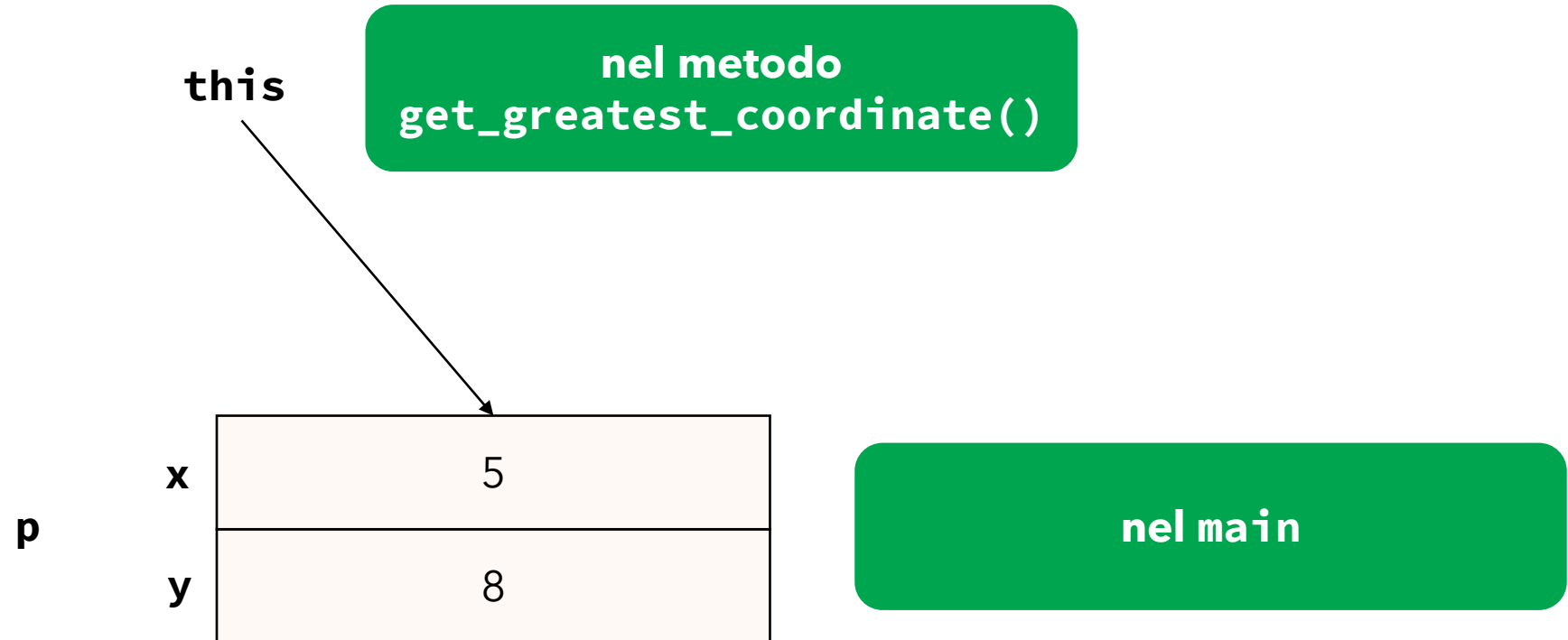
class Point {
public:
    double x;
    double y;
    double get_greatest_coordinate() {
        cout << "value of this pointer in method: " << this << endl;
        if (x >= y) {
            return x;
        }
        return y;
    }
};

int main() {
    Point p;
    p.x = 5;
    p.y = 8;
    cout << "address of p is: " << &p << endl;
    p.get_greatest_coordinate();
}

```

address of p in main is: 0133FDD8
value of this pointer in method: 0133FDD8

Il puntatore this



Il puntatore this

```
class Point {  
public:  
    double x;  
    double y;  
    double get_greatest_coordinate() {  
        //rewrite the code, using the this pointer  
    }  
};
```

Il puntatore this

```
class Point {  
private:  
    double x;  
    double y;  
public:  
    double get_greatest_coordinate() {  
        if (x >= y) {  
            return x;  
        }  
        return y;  
    }  
};
```

Una classe è formata da:

- un'**interfaccia**, costituita dai membri **public**, accessibili dall'esterno della classe
- un'**implementazione**, costituita dai membri **private**, non accessibili dall'esterno della classe

NB: nelle struct era tutto privato o tutto pubblico?

Inizializzare gli oggetti

```
class Date {  
public:  
    int month;  
    int day;  
    int year;  
};  
  
int main() {  
    Date d1;  
    //free initialization of data members, outside the class  
    d1.month = 13;  
    d1.day = -1;  
    d1.year = 2021;  
  
    cout << "Kind student, here is the date of your exam:" << endl;  
    cout << "\tmm" << "\tdd" << "\tyy" << endl;  
    cout << '\t' << d1.month << '\t' << d1.day << "\t" << d1.year << endl;  
}
```

Kind student, here is the date of your
exam:

mm	dd	yy
13	-1	2021

permettere di inizializzare così «allegrementemente» gli oggetti può portare a bug e disastri di vario tipo; sarebbe meglio se i membri dato fossero privati e l'inizializzazione venisse controllata all'interno della classe Date

Costruttori

```
class Date {  
private:  
    int month;  
    int day;  
    int year;  
public:  
    Date() {  
        cout << "I created an object" << endl;  
    }  
  
    int get_month() {  
        return this->month;  
    }  
    int get_day() {  
        return this->day;  
    }  
    int get_year() {  
        return this->year;  
    }  
};
```

```
int main() {  
    Date d1;  
}
```

output: I created an object

Encapsulation
(incapsulamento): i dettagli
dell'implementazione vanno
mantenuti privati

Costruttori

```
class Date {  
private:  
    int month;  
    int day;  
    int year;  
public:  
    Date() {  
        cout << this->month << endl;  
        cout << this->day << endl;  
        cout << this->year << endl;  
  
        cout << "I created an object" << endl;  
    }  
};
```

```
int main () {  
    Date d1;  
}
```

```
15505380  
15505380  
10854400  
I created an object
```

Costruttori

- Il costruttore senza parametri è detto **costruttore di default**
- Se non si intende cambiare l'inizializzazione di default degli oggetti, non serve ridefinirlo, viene aggiunto dal compilatore in automatico
- Inizializzazione di default di un oggetto: inizializzazione di default di ciascun membro

Costruttori

```
class Date {  
    private:  
        int month;  
        int day;  
        int year;  
    public:  
        Date(int month, int day, int year) {  
            this->month = month;  
            this->day = day;  
            this->year = year;  
        }  
};
```

```
int main() {  
    Date d1 = Date(1, 1, 2022);  
    Date d2 = Date(2, 3, 2021);  
}
```

**costruttore a 3 parametri;
sovrascrive quello di default,
che di conseguenza non è più
disponibile**

Costruttori

```
class Date {  
private:  
    int month;  
    int day;  
    int year;  
public:  
    Date(int month, int day, int year) {  
        if (month >= 0 && month <= 11 /*controls*/) {  
            this->month = month;  
            this->day = day;  
            this->year = day;  
        }  
        else {  
            //create a special Date with no meaning  
            this->month = -1;  
            this->day = -1;  
            this->year = -1;  
        }  
    }  
};
```

Overloading

```
void f() {  
    cout << "called f with 0 parameters";  
}  
  
void f(int a) {  
    cout << "called f with 1 parameter";  
}  
  
void f(int a, int b) {  
    cout << "called f with 2 parameters";  
}
```

```
f();  
f(5);  
f(8, 9);
```

Overloading: più funzioni con lo stesso nome. Il compilatore stabilisce quale chiamare in base alle diverse liste di parametri. Si dice anche che il nome della funzione viene *sovraccaricato (overloaded)* di significati

Overloading del costruttore

```
class A{  
private:  
    char c;  
    int i;  
    double d;  
public:  
    A(){  
        cout << "called default constructor";  
    }  
    A(char c){  
        (*this).c = c;  
        cout << "called 1-argument constructor";  
    }  
    A(char c, int i){  
        (*this).c = c;  
        (*this).i = i;  
        cout << "called 2-argument constructor";  
    }  
    A(char c, int i, double d){  
        (*this).c = c;  
        (*this).i = i;  
        (*this).d = d;  
        cout << "called 3-argument constructor";  
    }  
};
```

Passaggio per riferimento costante

- Abbiamo già visto l'utilità del passaggio per riferimento
- Immaginate di dover creare un catalogo dei libri della biblioteca comunale di una città:
 - soluzione **per valore**: viene creata una biblioteca clone e ve la portano a casa vostra. Comodo vero?
 - soluzione **per riferimento**: vi dicono dove si trova la biblioteca, ci andate, e fate il lavoro che dovete fare
 - il problema è che potreste fare del *side-effect* sulla biblioteca. Ad esempio potreste prendere un libro e scriverci qualcosa dentro (vietato):
 - risolvere la questione del possibile *side-effect* portandovi un clone della biblioteca a casa non è molto pratico...
 - è meglio farvi entrare in biblioteca, ma con qualche controllo che vi impedisca di farle del male

Passaggio per riferimento costante

```
void print(A& a){  
    cout << '[' << a.get_c() << ", " << a.get_i() << ", " << a.get_d() << ']' << endl;  
    a = A('v', 8, 1.61);  
}  
  
int main(int argc, char* argv[]){  
    A a('a', 78, 3.422);  
    print(a);  
    print(a);  
}
```

vi fidereste di una funzione che si chiama print e che fa queste cose al suo interno?

Passaggio per riferimento costante

```
void f(const A& a) {  
    a = A('v', 8, 1.61);  
}
```

const T&: significa che non si può modificare il parametro all'interno della funzione, quindi questo codice non compila perché abbiamo provato a modificarlo

oop_examples.cpp(42): error C2678: '=' binario: non è stato trovato alcun operatore che accetti un operando sinistro di tipo 'const A'. È anche possibile che non vi siano conversioni accettabili.

Passaggio per riferimento costante

**alcune possibili invocazioni di `void print(const A&);`
dalla funzione `main`**

```
A a('a', 78, 3.422);  
print(a);  
print({'b', 87, 43.34});
```

Puntatore a costante

```
void f(const int* p) {  
    if (p) {  
        cout << *p;  
        *p = *p + 1;  
    }  
}
```

error C3892: 'p': impossibile assegnare a una variabile const

`const int*` p significa: p punta ad un intero costante, ossia, l'intero a cui punta p non può essere modificato

Puntatore costante

```
int v[10] = {1, 5, 4, 5, 8, 6, 5, 4, 2, 1};  
int* const p = v;  
*p = 99;  
p++;
```

osservazioni?

Metodi costanti

```
class Person {  
    private:  
        string name;  
        string surname;  
    public:  
        Person(string n, string s) {  
            name = n;  
            surname = s;  
        }  
        string get_name() {  
            return name;  
        }  
        string get_surname() {  
            return surname;  
        }  
};
```

Metodi costanti

```
void hello_person(const Person& p) {  
    cout << "hello " << p.get_name() << " "  
    << p.get_surname() << endl;  
}
```

error C2662: 'std::string Person::get_name(void)': impossibile convertire il puntatore 'this' da 'const Person' a 'Person &'
oop_drills.cpp(63): note: La conversione comporta la perdita dei qualificatori
oop_drills.cpp(51): note: vedere la dichiarazione di 'Person::get_name'
oop_drills.cpp(64): error C2662: 'std::string Person::get_surname(void)': impossibile convertire il puntatore 'this' da 'const Person' a 'Person &'
oop_drills.cpp(64): note: La conversione comporta la perdita dei qualificatori
oop_drills.cpp(54): note: vedere la dichiarazione di 'Person::get_surname'

p è un riferimento costante, quindi non può essere modificato. Ma il compilatore non sa se get_name e get_surname modificano p, quindi non compila. Bisogna dire in qualche modo al compilatore che questi due metodi non possono modificare p

Metodi costanti

Esiste un modo per specificare che un metodo non può modificare l'oggetto di invocazione. Sicuramente metodi come i getter non modificano l'oggetto sui quali vengono invocati

```
string get_name() const {  
    return name;  
}  
string get_surname() const {  
    return surname;  
}
```

Metodi costanti

quindi, se sapete che un metodo non deve modificare l'oggetto di invocazione, va dichiarato const

```
string get_name() const {  
    return name;  
}  
string get_surname() const {  
    return surname;  
}
```

```
void set_name(string n) {  
    name = n;  
}
```



**questo metodo setter
naturalmente non è const
perché modifica un membro
dell'oggetto di invocazione**

Lista di inizializzazione del costruttore

- L'inizializzazione dei membri di un oggetto può essere fatta in questo modo, tramite la **lista di inizializzazione del costruttore**

```
class Polynomial_3 {  
private:  
    double coeff_3;  
    double coeff_2;  
    double coeff_1;  
    double coeff_0;  
public:  
    Polynomial_3(double c3, double c2, double c1, double c0):  
        coeff_3(c3), coeff_2(c2), coeff_1(c1), coeff_0(c0) { /*empty body*/ }  
};
```

la lista di inizializzazione riguarda solo i costruttori e viene eseguita prima del corpo del metodo

Distruttori

```
class B {  
private:  
    int x;  
    int y;  
  
public:  
    B() {  
        std::cout << "creating object of class B" << std::endl;  
    }  
  
    ~B() {  
        std::cout << "destroying object of class B" << std::endl;  
    }  
};
```

Distruttori

```
class C {  
private:  
    int x;  
    int y;  
public:  
    C() {  
        std::cout << "creating object of class C" << std::endl;  
    }  
  
    ~C() {  
        std::cout << "destroying object of class C" << std::endl;  
    }  
};
```

Distruttori

```
class D {
private:
    int d;
public:
    D() {
        std::cout << "creating object of class C" << std::endl;
    }

    ~D() {
        std::cout << "destroying object of class C" << std::endl;
    }
};

class E {
private:
    int d;
public:
    E() {
        std::cout << "creating object of class E" << std::endl;
    }

    ~E() {
        std::cout << "destroying object of class E" << std::endl;
    }
};
```

Distruttori

```
int main(int argc, char* argv[]){  
    B b_obj_main;  
    C c_obj_main;  
  
    return 0;  
}
```

```
creating object of class B  
creating object of class C  
destroying object of class C  
destroying object of class B
```

**analizzando le stampe su stdout
sembra che i metodi con la tilde
vengano invocati in automatico**

Distruttori

```
int main(int argc, char* argv[]){  
    B b_obj_main;  
    C c_obj_main;  
  
    return 0;  
}
```

creating object of class B
creating object of class C
destroying object of class C
destroying object of class B

i metodi con la tilde sono detti distruttori (*destructors*) e si occupano della distruzione degli oggetti

ma... abbiamo dovuto invocarli esplicitamente?

No! sono stati invocati in automatico. Perché?

Distruttori

le variabili, e quindi anche gli oggetti
allocati sullo stack (memoria automatica)
vengono deallocati automaticamente

```
void stack_alloc_dealloc() {  
    B b;  
    C c;  
    D d;  
    E e;  
}
```

```
creating object of class B  
creating object of class C  
creating object of class D  
creating object of class E  
destroying object of class E  
destroying object of class D  
destroying object of class C  
destroying object of class B
```

Distruttori

le variabili locali nell'activation record di una funzione vengono allocate e deallocate con politica LIFO, cioè a pila.

NB: vengono deallocate appena prima che il controllo del programma ritorni al chiamante

```
void stack_alloc_dealloc() {  
    B b;  
    C c;  
    D d;  
    E e;  
}
```

```
creating object of class B  
creating object of class C  
creating object of class D  
creating object of class E  
destroying object of class E  
destroying object of class D  
destroying object of class C  
destroying object of class B
```


Distruttori

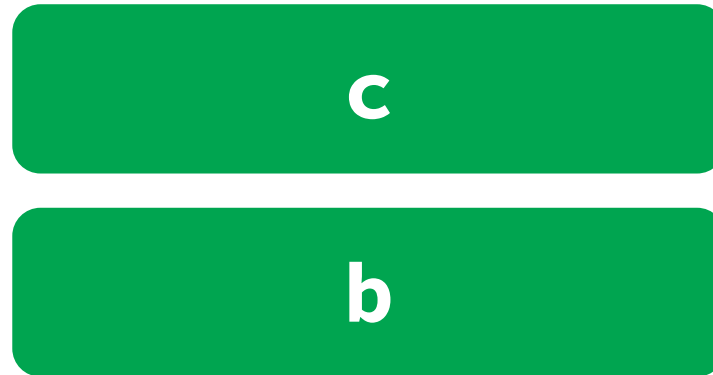
l'ultima variabile allocata è la prima ad essere deallocata
l'allocazione/deallocazione segue quindi una logica a
pila (stack)



b

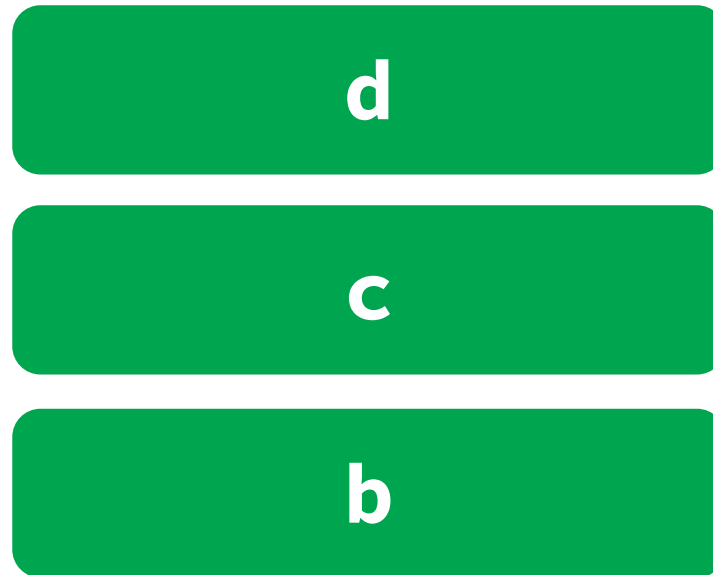
Distruttori

l'ultima variabile allocata è la prima ad essere deallocata
l'allocazione/deallocazione segue quindi una logica a
pila (stack)



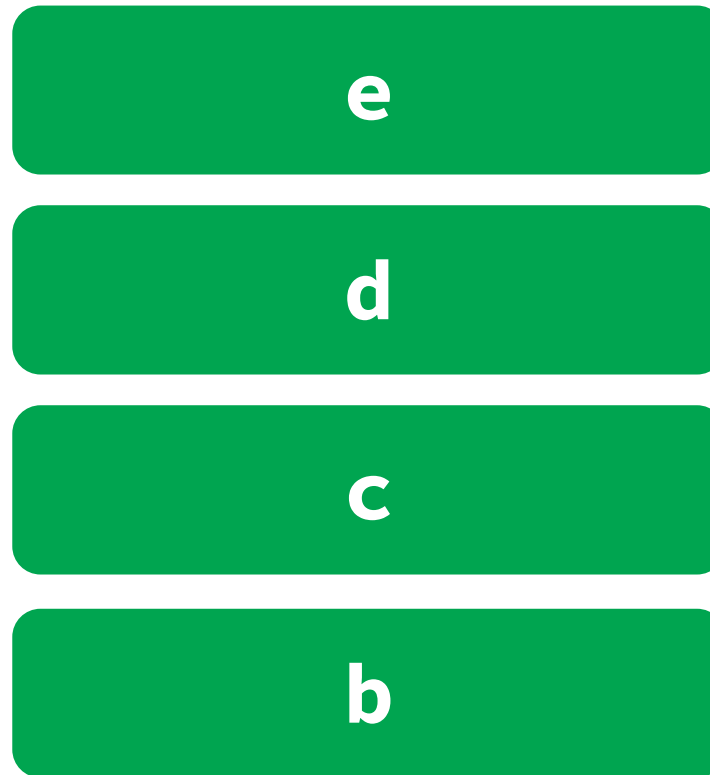
Distruttori

l'ultima variabile allocata è la prima ad essere deallocata
l'allocazione/deallocazione segue quindi una logica a
pila (stack)



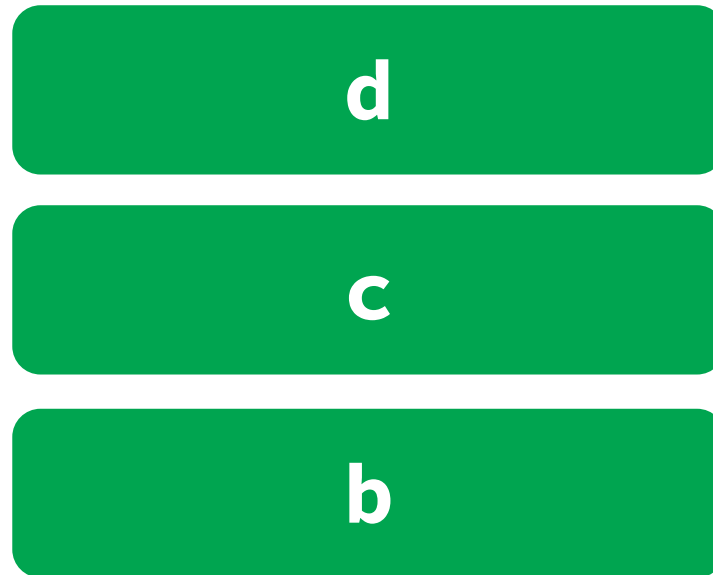
Distruttori

l'ultima variabile allocata è la prima ad essere deallocata
l'allocazione/deallocazione segue quindi una logica a pila (stack)



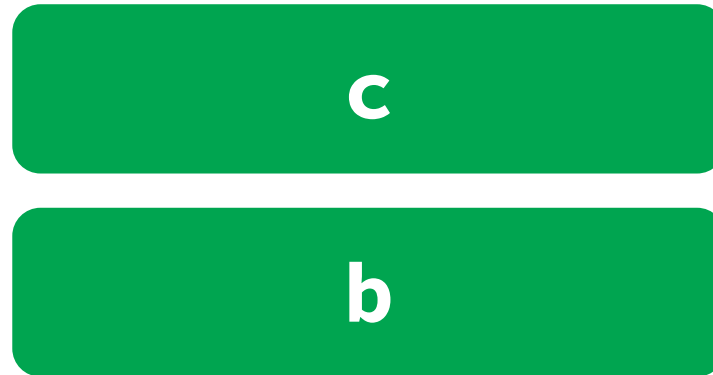
Distruttori

l'ultima variabile allocata è la prima ad essere deallocata
l'allocazione/deallocazione segue quindi una logica a pila (stack)



Distruttori

l'ultima variabile allocata è la prima ad essere deallocata
l'allocazione/deallocazione segue quindi una logica a
pila (stack)



Distruttori

l'ultima variabile allocata è la prima ad essere deallocata
l'allocazione/deallocazione segue quindi una logica a
pila (stack)



b

Progettazione di un array dinamico

- Proviamo a sviluppare una classe che rappresenti il tipo «**array di interi potenziato**»
- Per «potenziato» intendiamo *dotato di alcune funzionalità utili non presenti negli array C-like*
- Potremmo volere le seguenti funzionalità:
 - un metodo che verifica se l'array è ordinato
 - un metodo che realizza la ricerca lineare di una chiave nell'array
 - un metodo che realizza la ricerca binaria di una chiave nell'array
 - un metodo che ordina l'array tramite un algoritmo di ordinamento noto
 - un metodo che stampa su `stdout` il contenuto dell'array
- L'array va allocato sull'heap per dimensionarlo a runtime

Da vedere a casa

- [Pong & Object Oriented Programming - Computerphile](#)