

# Object-oriented programming (Programmazione ad oggetti) Parte 1

**Liceo G.B. Brocchi**  
**Classi terze Scientifico - opzione scienze applicate**  
Bassano del Grappa, Ottobre 2022  
Prof. Giovanni Mazzocchin

# Come unire dati e operazioni in un unico costrutto

- I tipi predefiniti di C++ sono detti **built-in**:
  - *char, int, float, double*
- 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

# Come unire dati e operazioni in un unico costrutto

- Immaginate una **rubrica telefonica**
- Sarebbe molto utile poter creare il tipo **Rubrica**
- Questo tipo non è solo caratterizzato 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 sui dati

# 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*

# Le classi

```
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 dall'esterno della classe

Come fa a funzionare se non prende nessun parametro?  
Come fa a sapere che i membri x e y sono dell'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;  
}
```



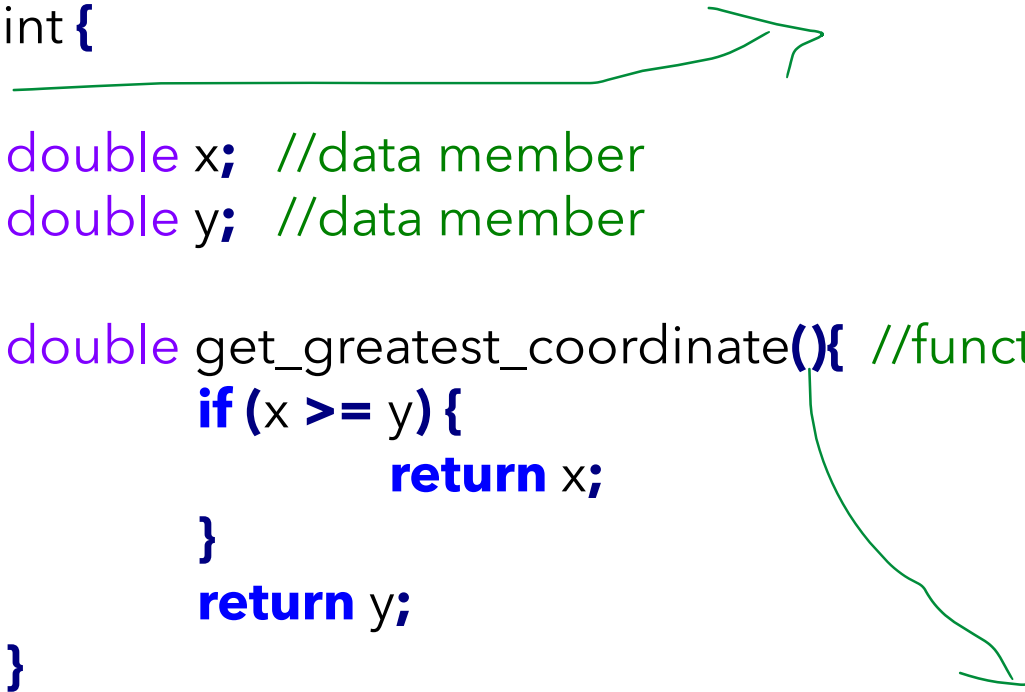
**Qui il parametro è specificato.  
Si vede chiaramente che vengono  
confrontate le coordinate del punto p.**

# Classi vs struct

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

# Le classi

```
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")  
}
```

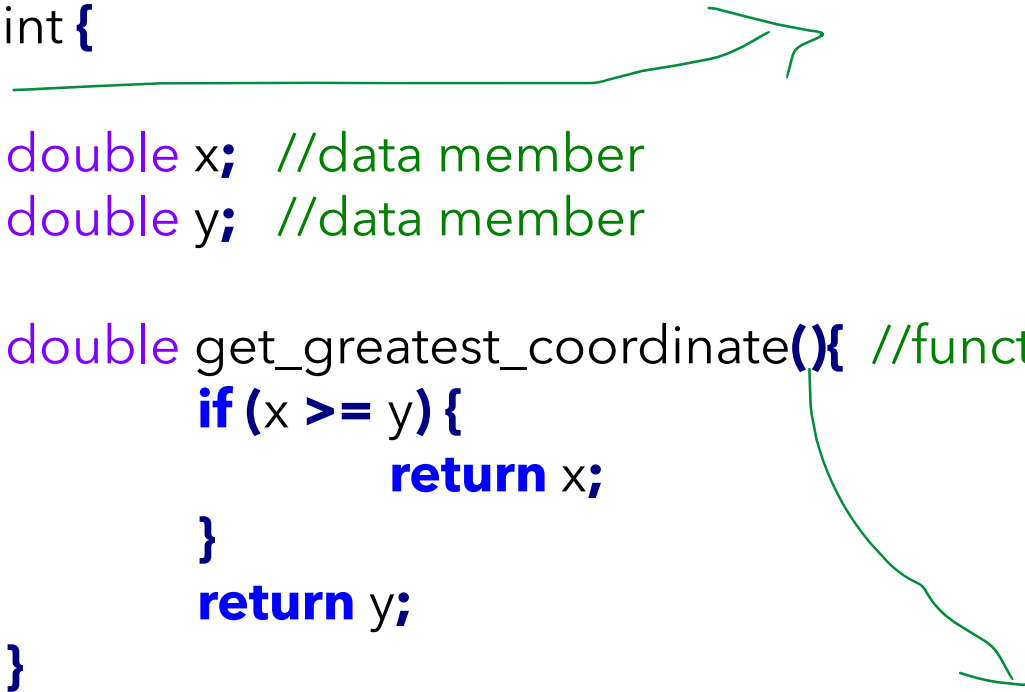


**È come se ci fosse un  
parametro nascosto che  
permette di accedere ai  
membri di p del main**



# Le classi

```
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")  
}
```



In realtà un parametro c'è,  
solo che è sottinteso.  
Si chiama this ed è un  
puntatore a Point

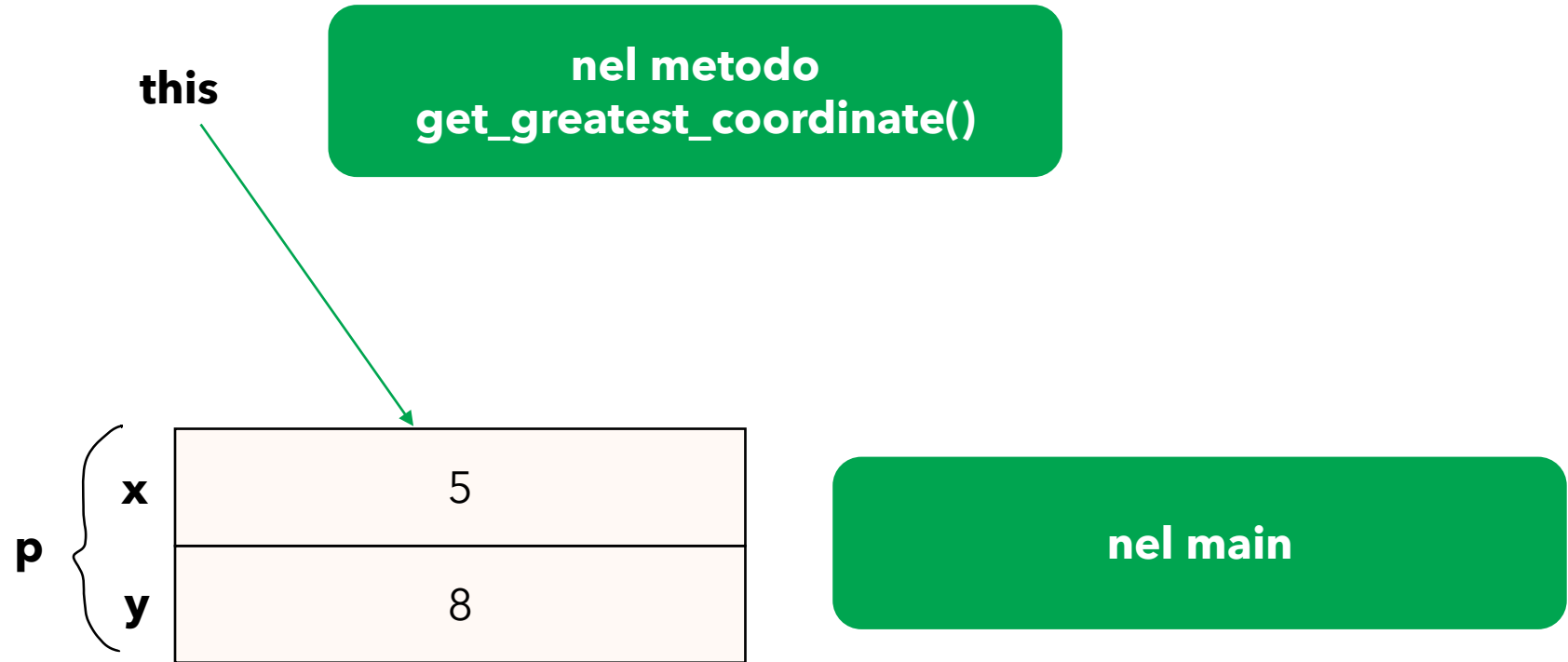
# Il puntatore this

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

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

```
int main(){  
    Point p; //object of class Point  
    p.x = 5; //access to data member  
    p.y = 8; //access to data member  
  
    cout << "address of p is: " << &p << endl;  
    p.get_greatest_coordinate();  
}
```

# Il puntatore this



# Il puntatore this

```
class Point {  
public:  
    double x;  
    double y;  
    double get_greatest_coordinate(){  
        //rewrite the code, but use the this pointer in the right way  
        //as with structs, access to members through a pointer is done by dereferencing or  
        //by the arrow operator  
    }  
};
```

# Il puntatore this

```
class Point {  
    private:  
        double x; //data member  
        double y; //data member  
  
    public:  
        double get_greatest_coordinate() { //function member ("method")  
            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 correttamente gli oggetti

```
class Date{  
public:  
    int month;  
    int day;  
    int year;  
};
```

Kind student, here is the date of your exam:

mm	dd	yy
13	-1	2021

```
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;  
}
```

**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**

# Inizializzare correttamente gli oggetti

```
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->month;
    }
    int get_year() {
        return this->month;
    }
};
```

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

**output**

I created an object

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

# Inizializzare correttamente gli oggetti

```
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;  
}
```

## Output

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



# Inizializzare correttamente gli oggetti

- 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

# Inizializzare correttamente gli oggetti

```
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 = day;  
    }  
};
```

```
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**

# Inizializzare correttamente gli oggetti

```
class Date{  
    private:  
        int month;  
        int day;  
        int year;  
    public:  
        Date(int month, int day, int year) {  
            //se non ci fosse this come lo scriveresti?  
        }  
};  
  
int main(){  
    Date d1 = Date(1, 1, 2022);  
    Date d2 = Date(2, 3, 2021);  
}
```

# Inizializzare correttamente gli oggetti

```
class Date{
private:
    int month;
    int day;
    int year;
public:
    Date(int month, int day, int year) {
        if ((month >= 0 && month <= 12) { //various 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;
        }
    }
};
```

# Ripasso: il concetto di 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**

# Progettare nuovi tipi utili

- 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 normali*
- Potremmo volere le seguenti funzionalità:
  - un metodo che verifica se l'array è ordinato
  - un metodo che realizza la ricerca lineare di una chiave sull'array
  - un metodo che realizza la ricerca binaria di una chiave sull'array, senza verificare se l'array è ordinato (chiamiamo questo metodo «insicuro»)
  - un metodo che realizza la ricerca binaria di una chiave sull'array, ma solo dopo aver verificato che l'array è ordinato
  - un metodo che ordina l'array tramite un algoritmo di ordinamento noto
  - un metodo che stampa su stdout il contenuto dell'array

# Progettare nuovi tipi utili

```
int main(){  
    const int dim = 20;  
    int ar[dim] = {};  
    for (int j = 0; j < dim; j++){  
        ar[j] = j;  
    }  
  
    ArrayPlus ap = ArrayPlus(ar, dim);  
    ap.print();  
    cout << ap.is_sorted();  
  
}
```

Prima di tutto allochiamo un normale array di interi di una certa dimensione stabilita staticamente (a *compile-time*).

In seguito, vogliamo che questo array diventi parte di un oggetto di tipo *ArrayPlus* (nome inventato da noi), che realizzerà le operazioni viste nella slide precedente.

# Progettare nuovi tipi utili

```
class ArrayPlus{  
private:  
    int* v;  
    int size;  
  
public:  
    ArrayPlus(int*, int);  
    bool is_sorted();  
    int binary_search(int, int, int);  
    void print();  
};
```

All'interno della classe, l'array viene visto come puntatore ad int. In questo modo possiamo far puntare *v* all'array creato all'esterno, ad esempio nel *main*.

Se avessimo definito *v* come array all'interno di *ArrayPlus* avremmo dovuto dargli una dimensione fissa!

Noi vogliamo invece che la dimensione venga decisa da chi utilizza la classe.

NB: i metodi non sono implementati. La classe contiene solo i prototipi dei metodi, ossia l'*interfaccia pubblica della classe*. L'interfaccia pubblica è costituita dalle funzionalità utilizzabili dall'esterno della classe.

NB: chi utilizza la classe non è interessato a *come è implementata l'interfaccia!*



# Progettare nuovi tipi utili

```
class ArrayPlus{  
private:  
    int* v;  
    int size;  
  
public:  
    ArrayPlus(int*, int);  
    bool is_sorted();  
    int binary_search(int, int, int);  
    void print();  
};
```

NB: chi utilizza la classe non è interessato a *come è implementata l'interfaccia!*

Ad esempio, a chi utilizzerà *ArrayPlus* non interessa se il metodo *print* è stato realizzato con un ciclo *while* o un ciclo *for*, o se utilizza l'aritmetica dei puntatori o l'accesso tramite parentesi quadre.

Altro esempio: chi utilizza la classe non sarà interessato all'implementazione della ricerca binaria (ricorsiva o iterativa), **gli basta che funzioni correttamente!**

# Progettare nuovi tipi utili

```
class ArrayPlus{  
private:  
    int* v;  
    int size;  
  
public:  
    ArrayPlus(int*, int);  
    bool is_sorted();  
    int binary_search(int, int, int);  
    void print();  
};
```

Noi, in quanto sviluppatori della classe *ArrayPlus*, dobbiamo occuparci dell'implementazione dei metodi e del significato dei membri dati privati.

**Iniziamo!**

# Progettare nuovi tipi utili

```
class ArrayPlus{  
private:
```

```
    int* v;  
    int size;
```

```
public:
```

```
    ArrayPlus(int*, int);  
    bool is_sorted();  
    int binary_search(int, int, int);  
    void print();  
    int get_size();  
    void left_shift_slice(int, int);  
    bool is_equal(ArrayPlus);
```

```
};
```

**l'array da «potenziare»**

**la dimensione dell'array**

**metodo costruttore**

# Progettare nuovi tipi utili

```
ArrayPlus::ArrayPlus(int* vec, int sz){  
    this->v = vec;  
    this->size = sz;  
}  
  
bool ArrayPlus::is_sorted(){  
    bool sorted = true;  
    for (int i = 0; i <= (this->size)-2; i++){  
        if (v[i] > v[i+1]){  
            sorted = false;  
        }  
    }  
  
    return sorted;  
}
```

# Progettare nuovi tipi utili

```
int ArrayPlus::binary_search(int key, int low_index, int high_index){  
    int middle_index = (low_index + high_index) / 2;  
    if (key == this->v[middle_index]){  
        return middle_index;  
    }  
    if (low_index > high_index){  
        return -1;  
    }  
    if (key > this->v[middle_index]){  
        return binary_search(key, middle_index + 1, high_index);  
    }  
    return binary_search(key, low_index, middle_index - 1);  
}
```

# Progettare nuovi tipi utili

```
void ArrayPlus::print(){
    for (int i = 0; i < this->size; i++){
        cout << this->v[i] << '\t';
    }
    cout << endl;
}

int ArrayPlus::get_size(){
    return this->size;
}
```

# Progettare nuovi tipi utili

```
//left_shift_slice([4, 5, 7, 9], 0, 3) --> [5, 7, 9, INT_MAX]
//left_shift_slice([4, 5, 7, 9], 1, 3) --> [4, 7, 9, INT_MAX]
//left_shift_slice([4, 5, 7, 9], 1, 3) --> [4, 7, 9, INT_MAX]
void ArrayPlus::left_shift_slice(int slice_low, int slice_high){
    //add checks on slice_low and slice_high
    for (int i = slice_low; i <= this->slice_high - 1; i++){
        this->v[i] = this->v[i + 1];
    }
    this->v[this->slice_high] = INT_MAX;
}
```

# 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";
    }
};
```

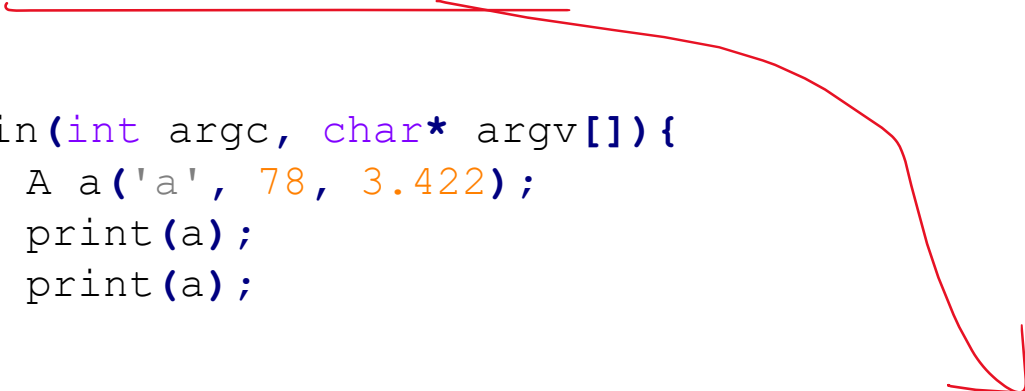


# Passaggio per riferimento costante

- 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 che vi viene portata 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);  
}
```

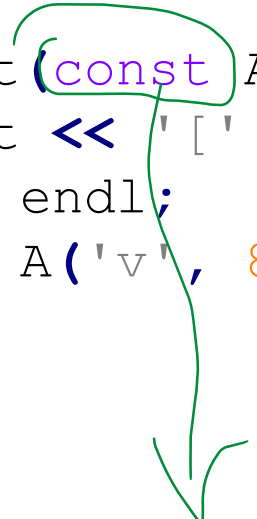


**Vi fidereste di una funzione che si chiama *print* che fa queste cose al suo interno?**

called 3-argument constructor  
[a, 78, 3.422]  
called 3-argument constructor  
[v, 8, 1.61]

# Passaggio per riferimento costante

```
void print(const A& a) {  
    cout << '[' << a.get_c() << ", " << a.get_i() << ", " << a.get_d()  
<< ']' << endl;  
    a = A('v', 8, 1.61);  
}
```



**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

```
void print(const A& a){  
    cout << '[' << a.get_c() << ", " << a.get_i() << ", " << a.get_d() << ']' << endl;  
}
```



**Mi aspetto delle domande su questi metodi della classe A**

# Passaggio per riferimento costante

Alcune possibili invocazioni di *print* dalla funzione *main*

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



**di cosa si tratta?**

# Concetto analogo: 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

# Concetto diverso: puntatore costante

```
int v[10] = {1, 5, 4, 5, 8, 6, 5, 4, 2, 1};  
int* const p = v;  
*p = 99;    //gets compiled  
p++;        //doesn't get compiled
```

**Osservazioni? Farsi aiutare dal titolo della slide**

# Metodi costanti

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



# 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;  
}  
string get_phone_number() const{  
    return name;  
}
```

# Metodi costanti

Quindi, se sapete che un metodo non deve modificare l'oggetto di invocazione, dichiaratelo const

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

```
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**

# Dalla creazione alla distruzione degli oggetti

```
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;  
    }  
};
```

# Dalla creazione alla distruzione degli oggetti

```
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;  
    }  
};
```

# Dalla creazione alla distruzione degli oggetti

```
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;
    }
};
```

# Dalla creazione alla distruzione degli oggetti

```
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**



# Dalla creazione alla distruzione degli oggetti

```
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?**

# Dalla creazione alla distruzione degli oggetti

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

```
void stack_allocation_deallocation(){  
    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
```

# Dalla creazione alla distruzione degli oggetti

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_allocation_deallocation(){  
    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
```

# L'allocazione-deallocazione LIFO delle variabili locali

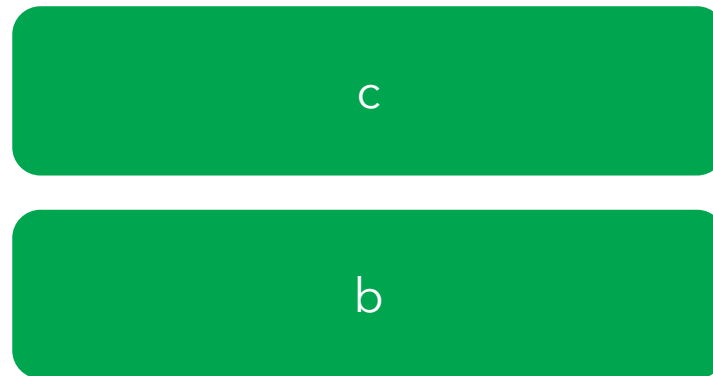
l'ultima variabile allocata è la prima ad essere deallocata  
L'allocazione/deallocazione segue quindi una logica a  
pila (stack)



b

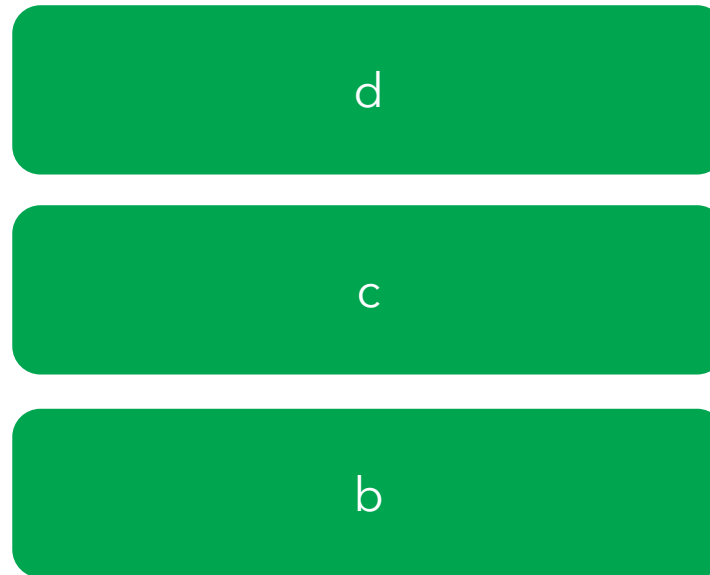
# L'allocazione-deallocazione LIFO delle variabili locali

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



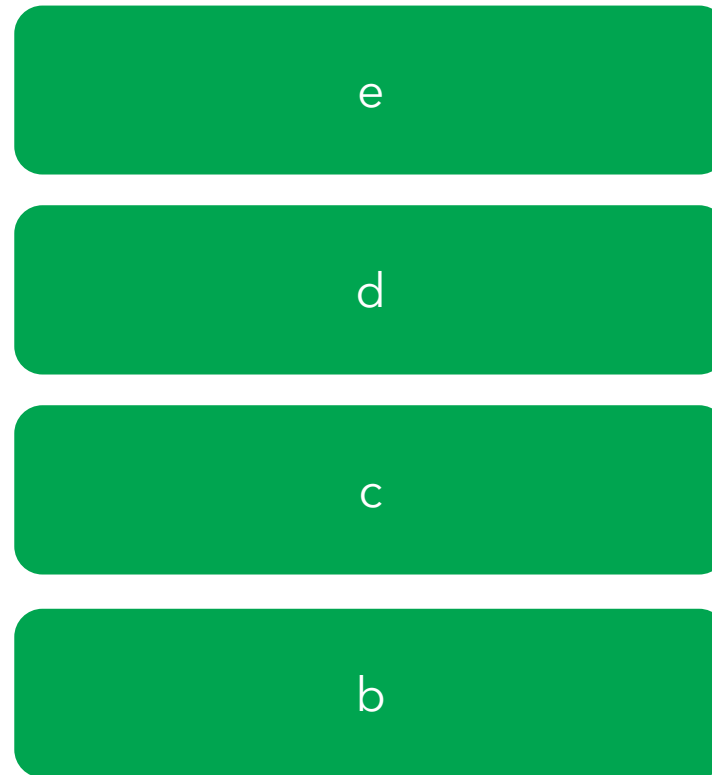
# L'allocazione-deallocazione LIFO delle variabili locali

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



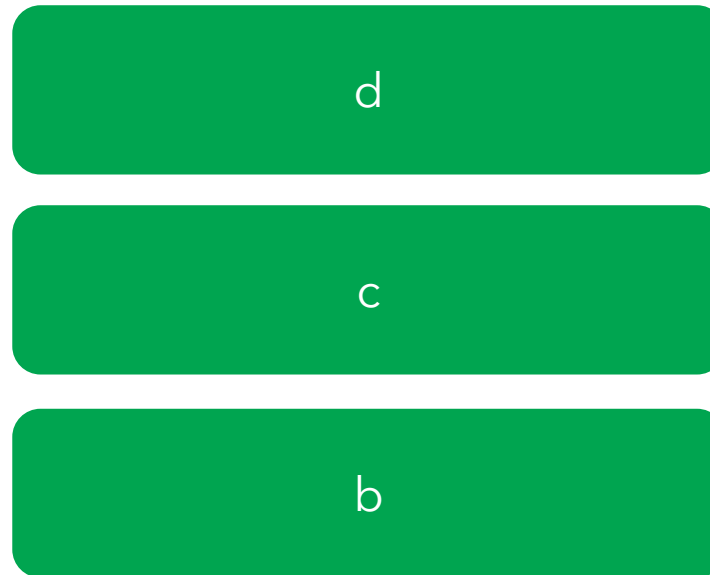
# L'allocazione-deallocazione LIFO delle variabili locali

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



# L'allocazione-deallocazione LIFO delle variabili locali

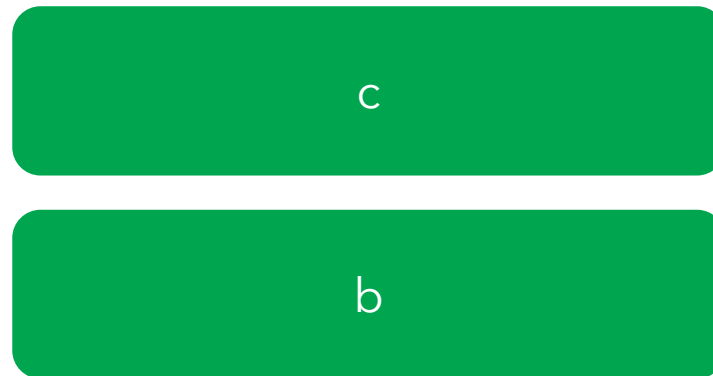
l'ultima variabile allocata è la prima ad essere deallocata  
L'allocazione/deallocazione segue quindi una logica a pila (stack)





# L'allocazione-deallocazione LIFO delle variabili locali

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



# L'allocazione-deallocazione LIFO delle variabili locali

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



b

# L'allocazione-deallocazione LIFO delle variabili locali

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

Ricordatevi che la gestione delle variabili sullo stack è automatica. Viene gestita dal compilatore e dal supporto run-time del linguaggio.

Quindi, per quando ne sappiamo fino ad ora, scrivere i distruttori è inutile. In questi esempi li abbiamo aggiunti soltanto per farli scrivere su standard output (*loggere*), per farci capire quando vengono invocati

*Non vi piacerebbe poter utilizzare la memoria in modo meno rigido e automatico?*