Verso l'OOP: struct, puntatori a funzione

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

La necessità di creare nuovi tipi «comodi»

- Immaginate di dover sviluppare un software di grafica, o un videogame
- Software del genere devono sicuramente rappresentare dei punti in un sistema di coordinate reali (**double** in C++)
- Immaginiamo che questo sistema di coordinate sia il piano cartesiano che conoscete dalla Geometria Analitica
- Immaginiamo che, oltre alle coordinate, questi punti debbano avere anche un'altra proprietà, un **colore**, di tipo **char**
- Con gli strumenti che conosciamo potremmo rappresentare questi punti con 3 variabili distinte, ma scollegate. Oppure con un array.
- Ma gli array sono strutture dati omogenee… non possiamo mescolare elementi di tipo diverso…

La necessità di creare nuovi tipi «comodi»

- Altro esempio di grafica: dovete rappresentare in un software il concetto di «retta nel piano cartesiano», con un certo *spessore*
- Da cosa è caratterizzata una retta nel piano cartesiano?
- **Hint**: sfruttate le vostre conoscenze matematiche
- Ora un esempio legato alla fisica: dovete costruire un software di simulazione numerica del moto dei pianeti. Quali sono le informazioni relative al pianeta che possono tornarci utili?

- In tutti gli esempi precedenti abbiamo la necessità di raggruppare più variabili in un unico contenitore, a cui daremo un nome

La necessità di creare nuovi tipi «comodi»

```
struct point {
        int x;
        int y;
        char color;
};
```

- **point** è detto *structure tag*
- **x**, **y**, e **color** sono detti *membri*
- dopo questa definizione, abbiamo a disposizione un nuovo **tipo**, di nome **struct point**

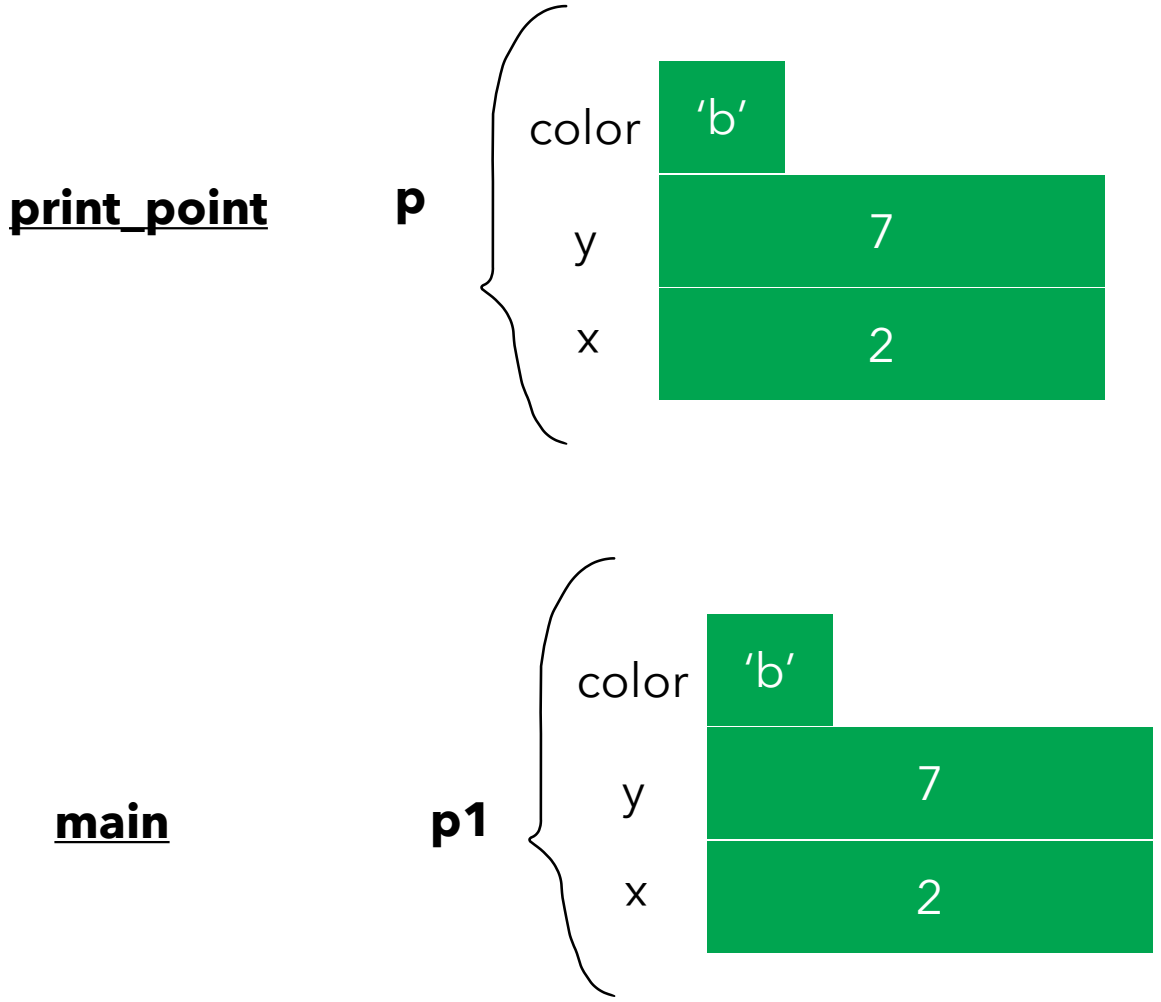
La necessità di creare nuovi tipi «comodi»

```
struct point {
        int x;
        int y;
        char color;
};
void print_point(struct point p) {
        cout << "{" << endl;
        cout << "  x coordinate: " << p.x << "," << endl;
        cout << "  y coordinate: " << p.y << "," << endl;
        cout << "  color: " << p.color << endl;
        cout << "}" << endl;
}

int main() {
        struct point p1 = {2, 7, 'b'};
        print_point(p1);
}
```

Abbiamo creato un nuovo tipo: **struct point**

passaggio per valore

La necessità di creare nuovi tipi «comodi»

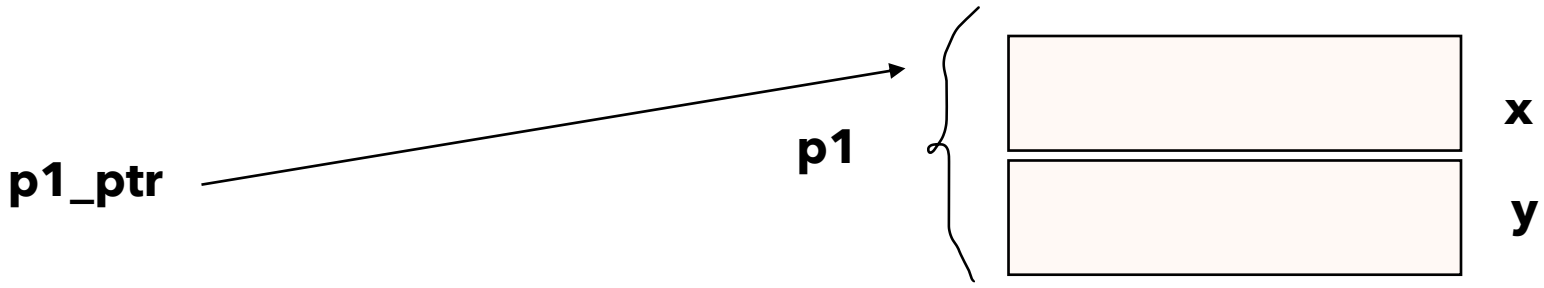


p è copia di p1. Vivono in aree di memoria diverse. È il solito passaggio per valore

di fatto la struct dovrebbe occupare 9 byte… in realtà ne occupa di più per motivi di allineamento. Ma per ora lasciamo stare…

La necessità di creare nuovi tipi «comodi»

```
struct point p1 = {2, 7};
struct point* p1_ptr = &p1;
cout << (*p1_ptr).x << ' ';   //dot operator has higher priority than * operator
cout << (*p1_ptr).y << '\n';
cout << p1_ptr -> x << ' ';   //-> is the 'arrow operator': just syntactic sugar for (*p1_ptr).
cout << p1_ptr -> y << '\n';
```



Struct annidate (*nested structs*)

```
struct point {
      double x;
      double y;
};

struct circle {
      struct point centre;
      double radius;
};
```

```
int main() {
      struct circle c1;
      c1.centre.x = 0;
      c1.centre.y = 0;
      c1.radius = 1;
}
```

I puntatori a funzione

- Le funzioni non sono variabili, ma sono contenute in aree di memoria dotate di indirizzo iniziale
- se **f** è il nome di una funzione, **&f** è l'indirizzo dell'inizio dell'area di memoria contenente le istruzioni di **f**

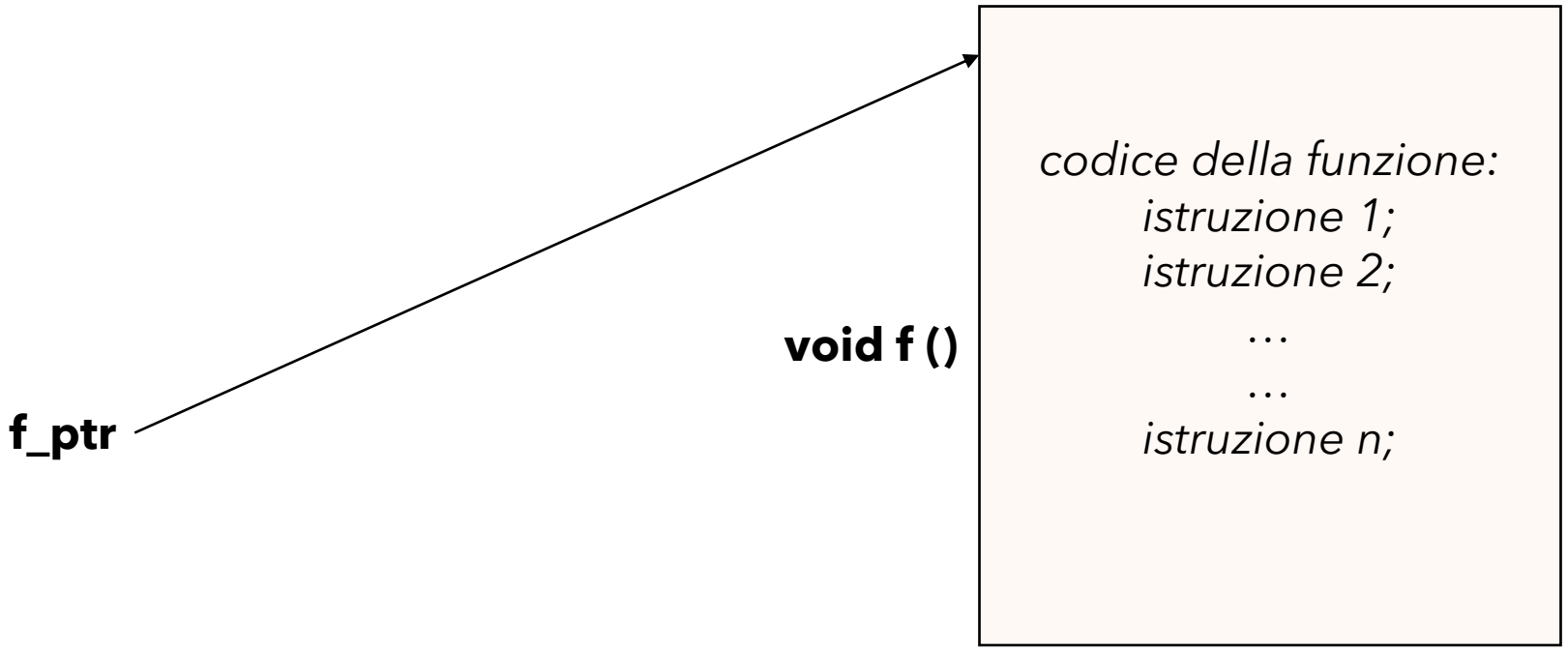
```
void f () {
        cout << "called function f" << endl;
}

int main() {
        void (*f_ptr)() = &f;
        f();
        (*f_ptr)();
}
```

Attenzione alla sintassi:
f_ptr è un puntatore ad una funzione che ritorna **void** e non prende nessun parametro.
Assegniamo ad f_ptr l'indirizzo di **f**. L'istruzione è tipizzata correttamente perché f ritorna **void** e non prende nessun parametro

Chiamata diretta ad f

Chiamata ad f per dereferenziazione di f_ptr

I puntatori a funzione



I puntatori a funzione

```
int inc_1(int n) {
        return n + 1;
}

int main() {
        int (*inc_1_ptr)(int) = &inc_1;
        int n = 0;
        n = inc_1(n);
        cout << n << endl;
        n = (*inc_1_ptr)(n);
        cout << n << endl;
}
```

inc_ptr è un puntatore a funzione che ritorna int e prende un parametro int;
inc_1 è una funzione che ritorna int e prende un parametro int;
È corretto assegnare a inc_ptr l'indirizzo di inc_1!

I puntatori a funzione

```
bool is_sorted(int ar[], int size, char order) {
        bool sorted = true;
        for (int i = 0; i < size - 1 && sorted; i++) {
                switch (order) {
                        //false means descending order
                        case 'd':
                                if (ar[i] < ar[i + 1]) {
                                        sorted = false;
                                }
                        break;
                        //false means ascending order
                        case 'a':
                                if (ar[i] > ar[i + 1]) {
                                        sorted = false;
                                }
                        break;
                        default:
                                //should return an error code... left as exercise
                                sorted = false;
                        break;
                }
        }
        return sorted;
}
```

is_sorted verifica se ar è ordinato in senso crescente se order == 'a', decrescente se order == 'b'

I puntatori a funzione

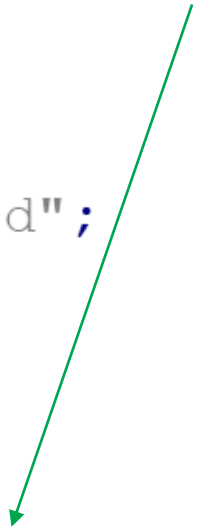
```
//returns true if a <= b
bool cmp_less_than (int a, int b) {
        return a <= b;
}

//returns true if a >= b
bool cmp_greater_than (int a, int b) {
        return a >= b;
}

bool is_sorted_smart(int ar[], int size, bool (*cmp)(int, int))
{
        bool sorted = true;
        for (int i = 0; i < size - 1 && sorted; i++) {
                if ((*cmp)(ar[i], ar[i + 1]) == false) {
                        sorted = false;
                }
        }

        return sorted;
}
```

il «comparatore» è passato come puntatore a funzione. In pratica, possiamo passare una funzione ad una funzione! Nella versione precedente il confronto tra ar[i] e ar[i+1] era interno a is_sorted e avveniva sulla base di un parametro

chiamata alla funzione «comparatore». cmp è un puntatore, quindi va dereferenziato.

I puntatori a funzione

```
int main() {
        int array[5] = {5, 2, 2, 1, 1};
        if (is_sorted_smart(array, 5, &cmp_greater_than)) {
                cout << "array is sorted";
        }
        else {
                cout << "array is not sorted";
        }
}
```



se vogliamo controllare se l'array è ordinato in senso decrescente, passiamo l'indirizzo della funzione che controlla se il suo primo parametro è >= del secondo

I puntatori a funzione: anche no

```
struct point {
      double x;
      double y;
};

double distance(struct point* p1, struct point* p2){
      return sqrt(pow(p1->x-p1->x,2)+pow(p2->y-p2->x,2));
}
```

Sarebbe comodo se la funzione che calcola il di un punto fosse interna a struct point. Associare ad un nuovo tipo anche le operazioni sul tipo all'interno della struct porterebbe alla scrittura di codice molto chiaro e manutenibile.

I puntatori a funzione: anche no

```
struct point {
        double x;
        double y;
        double distance(struct point* p1, struct point* p2){
                return sqrt(pow(p1->x-p1->x,2)+pow(p2->y-p2->x,2));
        }
};
```

Si può fare in C++
Non si può fare in C
In ogni caso, evitiamo di farlo

I puntatori a funzione: anche no

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

double distance(struct point* p1, struct point* p2){
        return sqrt(pow(p1->x-p1->x,2)+pow(p2->y-p2->x, 2));
}



int main() {
        struct point p = {5.6, 1.41};
        struct point pp = {7.6, 8.41};
        p1.distance_ptr = &distance;
        double distance = (*(p1.distance))(&p, &pp);
}
```

Le struct possono contenere puntatori a funzione.
Possiamo assegnare a questi puntatori l'indirizzo di funzioni definite fuori dalla struct.
L'obiettivo era unire dati e operazioni all'interno della struct.
In parte ce l'abbiamo fatta, ma questo codice è un disastro da leggere e manutenere.