

# C++: struct, union, puntatori a funzione

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

# La necessità di creare nuovi tipi *astratti*

- 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
- Oltre alle coordinate, questi punti potrebbero 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 *astratti*

- Altro esempio dalla Geometria: 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?
- 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 *astratti*

```
struct point {  
    double x;  
    double 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**

# Le struct

```
struct point {  
    double x;  
    double 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);
```

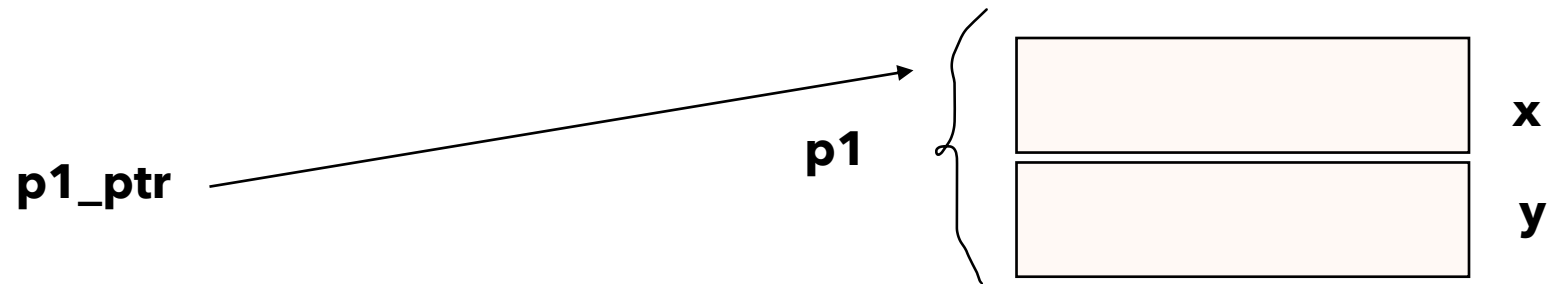
```
}
```

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

**passaggio per valore**

# Le struct

```
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;  
}
```

# Le union

- Nelle **struct** ogni membro ha la sua area di memoria
- Nelle **union**, invece, i membri condividono la stessa area di memoria, che viene acceduta tramite nome del membro
- La dimensione di una variabile di un tipo creato con una **union** è quindi la dimensione del membro più grande (*widest*)

```
union u {  
    unsigned short sh_val; //2 bytes  
    unsigned int ui_val; //4 bytes  
};
```

occupa 4 byte



# Le union

```
union u {  
    unsigned short sh_val; //2 bytes  
    unsigned int ui_val; //4 bytes  
};  
  
int main() {  
    union u u_var;  
    u_var.ui_val = 0xCAFEDEAD;  
    cout << "0x" << hex << u_var.ui_val << endl;  
    cout << "0x" << hex << u_var.sh_val << endl;  
}
```

4 byte, lo spazio c'è

solita area di  
memoria, vista come  
unsigned int

solita area di  
memoria, vista come  
unsigned short

cosa succede se chiedete  
&u\_var.ui\_val e  
&u\_var.sh\_val?

# Puntatori a funzione (*function pointers*)

- Le funzioni non sono variabili, ma sono contenute in aree di memoria che ovviamente partono da un determinato indirizzo (NB: le istruzioni, come i dati, sono memorizzate in qualche locazione di memoria)
- 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) ();  
}
```

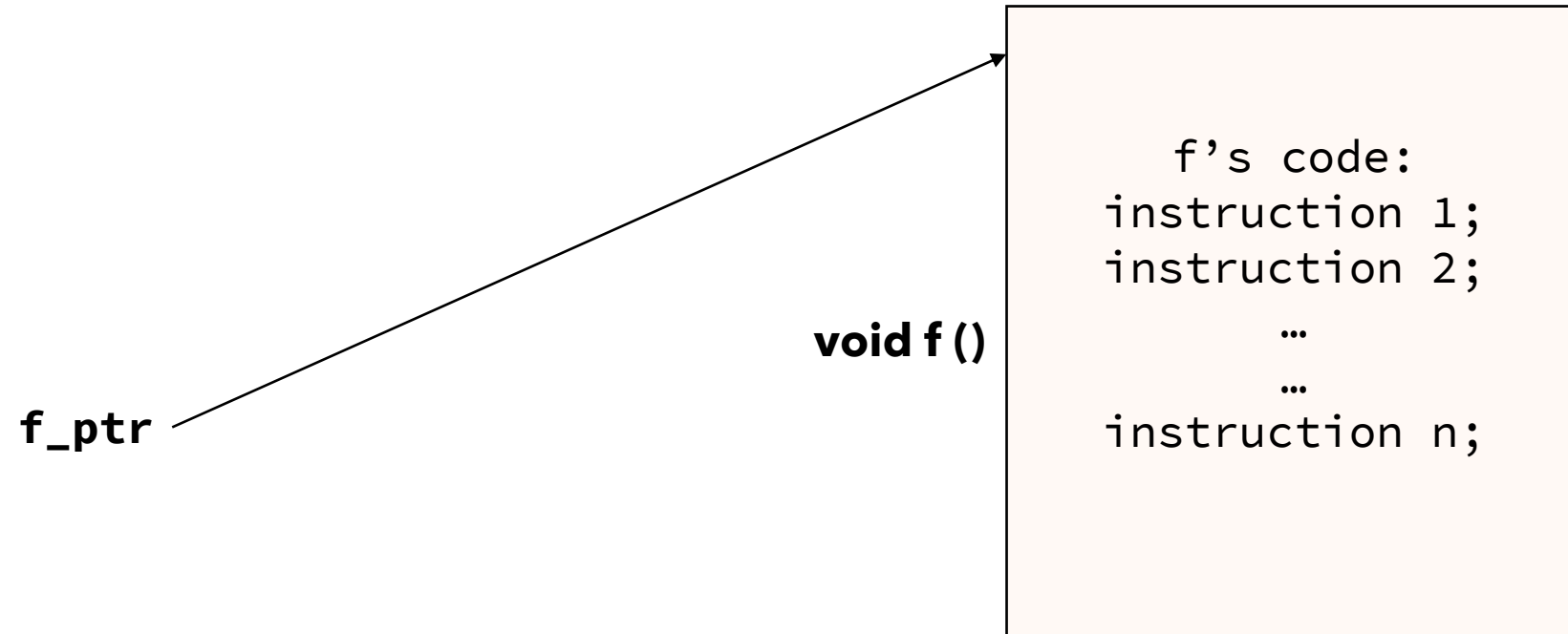
## Attenzione alla sintassi:

**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 ad f per  
dereferenziazione di f\_ptr

chiamata diretta ad f

# Puntatori a funzione



# 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!**

```

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

```

is\_sorted verifica se ar è ordinato in senso crescente se order == 'a', decrescente se order == 'b'

# 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

# 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  $\geq$  del secondo

# Puntatori a funzione

```
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 la distanza tra 2 punti 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**



# Puntatori a funzione

```
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**

# Puntatori a funzione

```
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->y, 2));
}

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